Toward a verification system for the programming language KL-1 using weakest preconditions

Timothy E. Lindquist
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd
Part of the Computer Sciences Commons

Recommended Citation
Lindquist, Timothy E., "Toward a verification system for the programming language KL-1 using weakest preconditions" (1979). Retrospective Theses and Dissertations. 6654.
https://lib.dr.iastate.edu/rtd/6654

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
INFORMATION TO USERS

This was produced from a copy of a document sent to us for microfilming. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help you understand markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure you of complete continuity.

2. When an image on the film is obliterated with a round black mark it is an indication that the film inspector noticed either blurred copy because of movement during exposure, or duplicate copy. Unless we meant to delete copyrighted materials that should not have been filmed, you will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed the photographer has followed a definite method in "sectioning" the material. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For any illustrations that cannot be reproduced satisfactorily by xerography, photographic prints can be purchased at additional cost and tipped into your xerographic copy. Requests can be made to our Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases we have filmed the best available copy.

University Microfilms International
300 N. ZEEB ROAD, ANN ARBOR, MI 48106
18 BEDFORD ROW, LONDON WC1R 4EJ, ENGLAND
LINDQUIST, TIMOTHY E.,
TOWARD A VERIFICATION SYSTEM FOR THE
PROGRAMMING LANGUAGE KL-1 USING WEAKEST
PRECONDITIONS.

IOWA STATE UNIVERSITY, PH.D., 1979
Toward a verification system for the programming language KL-1 using weakest preconditions

by

Timothy E. Lindquist

A Dissertation Submitted to the Graduate Faculty in Partial Fulfillment of The Requirements for the Degree of DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:
Signature was redacted for privacy.

In Charge of Major Work
Signature was redacted for privacy.

For the Major Department
Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa
1979
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>A. Background</td>
<td>1</td>
</tr>
<tr>
<td>B. Overview</td>
<td>7</td>
</tr>
<tr>
<td>II. RELATED WORK</td>
<td>10</td>
</tr>
<tr>
<td>A. The Flowchart Proof Method</td>
<td>11</td>
</tr>
<tr>
<td>B. Proof by Symbolic Execution</td>
<td>16</td>
</tr>
<tr>
<td>C. The Axiomatic Method</td>
<td>22</td>
</tr>
<tr>
<td>D. The Weakest Precondition Method</td>
<td>29</td>
</tr>
<tr>
<td>1. Sequential composition of statements</td>
<td>32</td>
</tr>
<tr>
<td>2. Simultaneous substitution</td>
<td>33</td>
</tr>
<tr>
<td>3. The assignment statement</td>
<td>34</td>
</tr>
<tr>
<td>4. A theorem for loops</td>
<td>35</td>
</tr>
<tr>
<td>5. The proof</td>
<td>36</td>
</tr>
<tr>
<td>III. PROCEDURES</td>
<td>39</td>
</tr>
<tr>
<td>A. External Procedures</td>
<td>45</td>
</tr>
<tr>
<td>B. Internal Procedures</td>
<td>58</td>
</tr>
<tr>
<td>1. Dynamic resolution</td>
<td>66</td>
</tr>
<tr>
<td>2. Static resolution</td>
<td>82</td>
</tr>
<tr>
<td>C. The Abstraction Theorem</td>
<td>92</td>
</tr>
<tr>
<td>1. Abstraction; verifying procedures</td>
<td>96</td>
</tr>
<tr>
<td>2. Abstraction; verifying functions</td>
<td>104</td>
</tr>
<tr>
<td>IV. COMPOSITE DATA TYPES</td>
<td>112</td>
</tr>
<tr>
<td>V. SUMMARY</td>
<td>123</td>
</tr>
<tr>
<td>VI. BIBLIOGRAPHY</td>
<td>128</td>
</tr>
<tr>
<td>VII. ACKNOWLEDGMENTS</td>
<td>133</td>
</tr>
</tbody>
</table>
VIII. APPENDIX A: THE KL-1 GRAMMAR 134
IX. APPENDIX B: THE WEAKEST PRECONDITIONS FOR KL-1 140
X. APPENDIX C: PROOF OF ROOTFINDER 149
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure 2.1.</th>
<th>The flowchart operators</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.2.</td>
<td>The interpreted flowchart for the square root</td>
<td>14</td>
</tr>
<tr>
<td>Figure 2.3.</td>
<td>The fully specified flowchart for the square root function, A</td>
<td>17</td>
</tr>
<tr>
<td>Figure 2.4.</td>
<td>The fully specified flowchart for the square root function, B</td>
<td>18</td>
</tr>
<tr>
<td>Figure 2.5.</td>
<td>Two consecutive IF-THEN-ELSE statements</td>
<td>20</td>
</tr>
<tr>
<td>Figure 2.6.</td>
<td>The execution tree for two IF-THEN-ELSE statements</td>
<td>20</td>
</tr>
<tr>
<td>Figure 2.7.</td>
<td>The square root function for proof by symbolic execution</td>
<td>23</td>
</tr>
<tr>
<td>Figure 2.8.</td>
<td>The symbolic execution tree for the square root function, initialization</td>
<td>24</td>
</tr>
<tr>
<td>Figure 2.9.</td>
<td>The symbolic execution tree for the square root function, loop</td>
<td>25</td>
</tr>
<tr>
<td>Figure 2.10.</td>
<td>The square root function for proof with Dijkstra's and Hoare's methods</td>
<td>27</td>
</tr>
<tr>
<td>Figure 3.1.</td>
<td>A KL-1 program which uses an external procedure</td>
<td>47</td>
</tr>
<tr>
<td>Figure 3.2.</td>
<td>External procedures, a general form</td>
<td>49</td>
</tr>
<tr>
<td>Figure 3.3.</td>
<td>A single external procedure</td>
<td>52</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3.4</td>
<td>Resolving nonlocal identifier references</td>
<td>62</td>
</tr>
<tr>
<td>3.5</td>
<td>Resolving procedure references</td>
<td>67</td>
</tr>
<tr>
<td>3.6</td>
<td>A general form for internal procedures</td>
<td>72</td>
</tr>
<tr>
<td>3.7</td>
<td>The static renaming of identifiers</td>
<td>84</td>
</tr>
<tr>
<td>3.8</td>
<td>ST applied to Figure 3.7</td>
<td>86</td>
</tr>
<tr>
<td>3.9</td>
<td>The program of Figure 3.4 after application of ST</td>
<td>89</td>
</tr>
<tr>
<td>3.10</td>
<td>External procedure for proof with the theorem</td>
<td>100</td>
</tr>
<tr>
<td>4.1</td>
<td>A procedure to exchange two values</td>
<td>118</td>
</tr>
</tbody>
</table>
I. INTRODUCTION

A. Background

It has recently become apparent that software accounts for a majority of the cost of a computer system. Additionally, an overwhelming portion of this software cost is absorbed by long-term maintenance. Therefore, it is no surprise that a great deal of computer science research is aimed at producing software which is both initially correct and which will cost less to support. The intention of current research is to produce programs which will require less corrective maintenance and which can evolve readily with changing requirements. Three largely distinct and complementary areas of research deal almost completely with this problem; they are structured programming, fault tolerance, and software verification.

Of late, structured programming, a loosely defined area of research, has made a significant impact on software development. The term structured programming usually refers to either a methodology for writing programs or to programming language features which support a methodology. The methodology aspect of structured programming attempts to define the steps necessary to compose clear, concise, and correct programs. Many structured design methodologies exist including stepwise refinement, into programming, the data flow design method, and the data structure design method.
Features which support these methodologies can be seen in nearly all recent high level languages. Excellent examples are the data abstraction facilities of both ALPHARD (54) and CLU (38), and the one-in-one-out control constructs employed in most high level languages today.

Structured programming attempts to decrease the cost of maintaining software in two ways. First, programs which are developed using structured techniques are more readable, and thus, more adaptable to changing requirements. Secondly, structured programs are more thoroughly designed, and consequently, require less corrective maintenance.

Fault tolerance is displayed by a system if it has the ability to continue acceptable service despite the occurrence of hardware problems or erroneous input. For example, a peripheral control system which interfaces many terminals to some higher level must be capable of continued operation in the face of hardware problems with one or more devices. In addition, the system must be tolerant of bad or even malicious input originating at any one of the terminals.

Fault tolerance has typically been restricted to the design of operating systems; however, the principles extend in a large sense to software in general. Work in the area of creating more fault tolerant software has identified another way to reduce software maintenance costs. A more fault
tolerant system reduces the number of maintenance changes needed to correct errors which result from an incomplete or poor design.

Another area of research, program verification, is directed toward developing techniques which can be used to validate programs. The underlying concept of this research is that thorough verification during the development process will result in a program which requires less long-term maintenance. The verification methods being developed are used to certify that a program executes according to its design specifications. Three approaches to program verification are taken.

1. The static approach. Verification is done after the software is written and is based only on the syntax of the program and its specifications.

2. The constructive approach. Verification takes place as the program is being written.

3. The execution approach. Verification is done by running the program.

With the static approach, the source and specifications of the completed program are used to construct a mathematical proof of correctness. A static program verifier can be viewed as a black box with two inputs and one output. The first input is the text of the program to be verified and the second is the program's design specification, a description
of the function that the program performs. The latter is normally written in terms of propositions over the program's variables. The output of the verifier is a simple yes or no which tells whether or not the program implements the intended design.

Static verification methods are well documented and define, with a certain amount of rigor, techniques which may be used to prove programs. One of these methods, the weakest precondition method, is used as the foundation for the KL-1 verification system. Other prominent methods of static verification are reviewed in the related works section of this thesis.

The constructive approach to verification involves writing a program and proving it at the same time. The goal of this approach is to know that the program is correct as it is being written. In general, the techniques for proving programs as they are being constructed outline a programming methodology which uses a static verification method.

The best illustration of this approach is given in (9). Here, several examples are considered which demonstrate the process of simultaneous construction of a program and its proof. This approach does not rigorously define techniques for software development and verification. In fact, Dijkstra emphasizes the need for a creative component to be involved in writing programs and constructing their proofs.
It should be noted that the constructive and static approaches to verification do not require that the program be executed. Two important points arise from this fact. First, the execution characteristics (operational semantics) must be derived from the program's syntax only. Thus, static and constructive methods not only outline a means of verifying programs, but also define the operational meaning of statements in a language. Second, the static and constructive verification methods must assume that the translation and execution of a program does not alter its meaning.

The final approach to verification is execution time verification. In general, this approach does not constitute a formal method of verification, but does play an important role in software validation.

The well-known technique of exhaustive testing is included in this category. Using this method, a program can be proven correct providing that each valid execution path is shown to produce the desired results. In a program containing both alternative and repetitive control constructs, there exist several possible execution paths. For any execution of the program, the path taken depends on the values of the input.

To verify a program by exhaustive testing, it must be executed with each possible set of input values. In each
case, the results must agree with the specifications. For most programs this task is not feasible or at the very least, not economical; however, common practice allows selecting a limited number of input values to test. Several studies have been conducted which deal with the testing process (see, e.g. 26, 3). The intent of these studies is to add some rigor to execution time verification.

Another form of execution time verification is provided by allowing more run time control of a program's data. For example, the language EUCLID allows the programmer to place boolean conditions, called assertions, throughout the program text. When these assertions are encountered during execution, they are checked, and thus, provide a means of controlling the values of the program variables.

As another example, the Symbol 2-R system (46) allows the use of On-Blocks to control data values. In essence, On-Blocks are procedures associated by declaration with a data item. The On-Block is invoked by a software trap each time an attempt is made to change the associated data item. Consequently, the On-Block can control the values taken by the program data element.

The debugging tools which are normally built into language interpreters provide other facilities for execution time verification. These tools include the ability to examine and change data during execution, the ability to
single step through a program, and other related control facilities.

Structured programming, fault tolerance, and program verification each focus on an important aspect of software reliability. It should be emphasized that no one or two of these areas alone seem to adequately address the problem of producing software which requires less long-term maintenance. The production of more reliable software depends on developing structured programming methods, making software systems more tolerant to hardware and software faults, and to some extent, verifying that software performs the function for which it was originally designed.

B. Overview

There is no doubt that the execution time approach is the most commonly used method for verifying software. Execution time techniques are deeply rooted in both our educational system and in the world of pragmatic computer use. Currently, support is increasing for the adoption of static and constructive verification techniques; however, the extent to which these techniques should be adopted is an issue for debate (6, 10).

While the economics of employing these verification techniques is being questioned, the benefit derived from both teaching and developing the static and constructive approaches to verification is indisputable. Since the KL-1
language system is a tool for teaching students how to program, it will include facilities for all three types of software verification. The constructive approach can be built into the system by merging a design methodology, functional decomposition, with the Weakest Precondition method of static verification. The static approach will be accommodated by facilities which allow a program's proof to be constructed interactively. Facilities will also be included which encourage a rigorous form of execution time verification.

The language KL-1 is a block structured language based on control constructs designed for conceptual clarity. These "thinking constructs" for programming were developed by Keller (32) and are based on ideas presented by Gries (16) and McKeeman (43) on writing programs. As in PL/1, the language KL-1 allows procedures to be defined both internally and externally. Data is controlled in KL-1 as in most compiled block structured languages e.g., PASCAL, ALGOL, and PL/1. KL-1 does not include either heap or static storage classes and it does not include facilities for programmer defined data types. The syntactic structure of the language is given in Appendix A.

The objective of this thesis is to complete the formal foundation for the KL-1 verification system. This foundation is the Weakest Precondition method of static verification and
is needed for both the constructive and static portions of the KL-1 system. Weakest preconditions have been applied to the control constructs of KL-1 (37). To complete the foundation, the Weakest Precondition method must be applied to the remainder of the language. This is done by detailing the proof mechanisms for KL-1 procedures and arrays.

A large portion of the thesis treats procedure mechanisms in general. This treatment includes proof techniques for KL-1 procedures and provides proof tools for a class of procedures which have not been previously addressed.
II. RELATED WORK

To form an adequate background for the remaining chapters, the major methods of static verification will be reviewed. Included in this review are the Flowchart proof method, proof by Symbolic Execution, the Axiomatic method, and the Weakest Precondition method.

These methods of static proof require as input the program to be verified and the design specifications to which the program is to conform. The verifier, whether automatic or manual, must then decide if the program implements the design specifications. In each of the methods discussed, the assumption is made that specifications of the program can be put into the form of two propositions. These propositions are first order logic expressions over the variables of the program and are called the input and output assertions. This assumption does not cause a problem for relatively simple software; however, its feasibility when considering larger more complex programs might be questioned.

In this chapter, only the proof of partial correctness is considered. Partial correctness proves that a program adheres to its specifications and disregards any guarantee of termination.

With each method reviewed, an example of its use in verifying a simple program is given. The example which is proven for all methods is a program which computes for a
given number the largest natural number which is not greater than its square root. For example with input 100, the function would return the value 10 and with the input 40, the function would return the value 6.

A. The Flowchart Proof Method

The Flowchart method described here is taken from Floyd's paper (13). A flowchart language containing five operators is used to express the syntax of the program to be proven (see Figure 2.1). Flowchart programs are combinations of these operators connected by directed edges.

The program's specifications are expressed by a flowchart interpretation, a term introduced by Floyd. A flowchart is interpreted by labeling its edges with appropriate predicates. The predicates (or propositions) are expressions of first order logic whose free variables are identifiers of the program. The propositions describe conditions which must be true when execution of the flowchart passes the associated edge.

For the Flowchart method to work correctly, propositions must appear in the following places. First, the edge leading out of the start operator must be labeled with a predicate. This proposition corresponds to the input specification for the program and expresses the condition which is assumed true each time the program is entered. Second, a proposition called the output assertion is associated with the edge
The assignment operator

The join operator

The conditional branch operator

The start operator

The halt operator

Figure 2.1. The flowchart operators
leading into the HALT operation. This proposition expresses the desired results of the program. Finally, a proposition must be associated with at least one edge in each loop of the program. This proposition, commonly called the loop invariant or the inductive assertion, expresses a condition which must be satisfied on each pass through the loop.

Figure 2.2 presents the flowchart and its interpretation for the square root function. The flowchart uses the five types of operators described previously and details a solution to the problem. The interpretation of the flowchart describes its intended operation. The input assertion requires that the value of the variable X be non-negative when execution begins. The variable Z represents the function's output. The output assertion specifies that on termination Z has the desired value. That is, Z is not greater than the square root of the input X and Z+1 is greater than the square root of X. Both the input and output assertions are consistent with the description of this function and are used for all proofs of this function.

Floyd's method provides a formal means to verify that the interpretation of the flowchart actually describes the function computed. It must be shown that if the proposition associated with the START operation is true when execution begins, then if the HALT statement is reached, the proposition associated with it will also be true.
Figure 2.2. The interpreted flowchart for the square root.
To verify that the flowchart is consistent with its interpretation, Floyd defines for each of the five operators of the flowchart language a predicate transformer. This transformer takes one assertion and changes it in some specified manner to create a new one. For any operator, its predicate transformer acts as follows. Given the predicate which is assumed to be true before encountering the operator, it generates an assertion which is guaranteed to be true after execution. This assertion is called the strongest verifiable consequent. For example, suppose that the predicate $X > 5$ is true prior to the execution of the assignment operator

$$X := X + 5;$$

then, the transformer for this operator will indicate that after the assignment the assertion $X > 10$ will be true. The predicate transformer produces an assertion which is the strongest verifiable consequent of executing the operator. It is the strongest consequent, because it describes all possible values which the variables may have as a result of executing the operation.

The predicate transformers are applied successively to each operator in the program to prove that the flowchart and its interpretation are consistent. After each application, the resulting assertion is placed on the edge leading out of the operator. The important step in this process is the
generation of the strongest verifiable consequent for the operator whose edge contains the output assertion. If the generated predicate implies the output assertion, then the interpreted flowchart is verified. Figures 2.3 and 2.4 present the flowchart for the square root function after the transformers have been applied to all statements in the program. The Flowchart proof method has strongly influenced other work in program verification. Floyd's work has established a base from which the Weakest Precondition and Axiomatic proof methods have evolved (8, 21).

B. Proof by Symbolic Execution

The Symbolic Execution method of proof is probably the most straightforward of those presented. Symbolic execution involves proving the partial correctness of programs written in a simplified programming language. This language is based on the statement types assignment, if-then-else, and the repetitive construct, while-do. A more detailed explanation of this method appears in (19).

As in the Flowchart method, this method requires that the syntax of the program be accompanied by an interpretation of the program. The interpretation details the desired operation of the program by means of propositions over its variables. The propositions needed for this method are the input assertion, the output assertion, and an inductive assertion for each loop contained in the program. The
\( (X \geq 0) \)

\( (I = 0 \text{ AND } X \geq 0) \)

\( (J = 1 \text{ AND } I = 0 \text{ AND } X \geq 0) \)

\( (K = 1 \text{ AND } J = 1 \text{ AND } X \geq 0 \text{ AND } I = 0) \)

From Figure 2.4 (A)

\( J \leq X? \)

\( Z^2 \leq X \text{ AND } X < (Z + 1)^2 \)

FALSE

TRUE

To Figure 2.4 (B)

Z + I

HALT

Figure 2.3. The fully specified flowchart for the square root function, A
Figure 2.4. The fully specified flowchart for the square root function, B
language contains special statements for placing these propositions in the program's text; namely, ASSUME, PROVE, and ASSERT. The statement

\textbf{ASSUME} (proposition)

is used to express the input assertion and is generally placed before any executable statements in the program. The statement

\textbf{PROVE} (proposition)

specifies the output assertion and is placed immediately before the end statement. Finally, the statement

\textbf{ASSERT} (proposition)

is used to place invariant assertions in loops.

Verification of a program using this method begins by placing the input, output, and invariant assertions in the program. Once this has been done, \textit{symbolic execution trees} of the program are created. The number of trees created depends on the number of loops contained in the program. The program is verified when each leaf of each execution tree contains the word "verified".

The use of alternative and repetitive statements in a program introduces the possibility of having many execution paths through the program. For example, suppose that two if-then-else statements occur in a program with one directly following the other (Figure 2.5). There are, based on syntax only, four possible execution paths through these two
IF B1 THEN S1
ELSE S2 ENDIF;

IF B2 THEN S3
ELSE S4 ENDIF

Figure 2.5. Two consecutive IF-THEN-ELSE statements

Figure 2.6. The execution tree for two IF-THEN-ELSE statements
statements. The symbolic execution tree, which corresponds to these two if-then-else statements, also has four paths (Figure 2.6). Each path through the tree corresponds to a path through the if-then-else. In general to verify a program, trees are constructed to certify that all possible execution paths through the program will generate the desired results.

The rules for the symbolic execution of a program define how these trees are built. A separate tree is formed for each ASSUME and ASSERT statement in the program. The symbolic execution of statements proceeds sequentially through the program until either a PROVE or an ASSERT statement is reached. Each program statement causes a new node to be placed in the tree. When a statement which alters a variable is encountered, a new symbolic value for that variable is calculated and the new value is associated with the proper node in the execution tree.

Each path through a symbolic execution tree is described by a predicate known as the path condition (pc). This predicate describes the values necessary to cause this execution path to be taken (for example, see Figure 2.6). When a statement which alters the flow of control is found, the path condition is changed.

The most important step in the construction of the tree occurs when a PROVE or an ASSERT statement is encountered.
This denotes the end of an execution path through the tree. At this point to verify that the program produces the desired results, it must be shown that the predicate for the PROVE or ASSERT statement will be true for all possible executions of the path. This is done by finding the current symbolic value of the predicate and comparing this value with the current path condition. If the path condition logically implies the evaluated proposition, then the path ending at the node is verified. The program for the square root function is shown in Figure 2.7 and the symbolic execution trees which prove the program are presented in Figures 2.8 and 2.9.

C. The Axiomatic Method

The most widely used method of static verification is the Axiomatic method. This method, first presented by Hoare (21), is based on ideas from Floyd's Flowchart method. The Axiomatic method has been applied to various languages; the most notable of which are PASCAL (24) and EUCLID (40). EUCLID is especially interesting since this language is probably the first in which design decisions were made with the intention of simplifying program proofs.

In the mathematical sense, a theory is defined by applying the Axiomatic method to a programming language. The theory consists of a set of theorems, a set of axioms, and a set of rules of inference. A theorem of the theory is a program together with its input and output specifications.
1  ASSUME (X ≥ 0)
2    I := 0
3    J := 1
4    K := 1
5  WHILE (J ≤ X) DO
6      ASSERT ((I+1)**2 ≤ X AND J = (I+1)**2
7                          AND K = 2 * I + 1)
8    I := I + 1
9    K := K + 2
10   J := J + K
11  END WHILE
12  Z := I
13  PROVE (Z**2 ≤ X AND X < (Z + 1)**2)
14  END

**Figure 2.7.** The square root function for proof by symbolic execution
Figure 2.8. The symbolic execution tree for the square root function, initialization
Figure 2.9. The symbolic execution tree for the square root function, loop
Minimally, it is required that all theorems of the system be programs which match their specifications; that is, the system must be sound. To determine whether or not a program and its requirements form a theorem, axioms and rules of inference are defined. If a program and its requirements are derivable from the axioms and rules, then they constitute a theorem.

Syntactically, the theorems of the system take the form

\[ \vdash P[S] Q. \]

Where \( S \) is a statement or set of statements of the programming language and \( P \) and \( Q \) are predicates (assertions) over the variables used in \( S \). The turnstile, \( \vdash \), indicates that \( P[S]Q \) is a theorem of the system. Intuitively, \( P[S]Q \) can be interpreted to mean, if \( P \) is true before execution of \( S \), then \( Q \) will be true after execution provided \( S \) halts.

Figure 2.10 is the square root program for proof with both the Axiomatic method and Dijkstra's Weakest Precondition method. The axioms and rules of inference necessary to construct a proof of this program follow. Only the axioms and rules which relate to the program statements are given. Those dealing with the underlying computer arithmetic and the assertion language are ignored for this treatment.
I := 0
J := 1
K := 1
WHILE (J ≤ X) DO
    I := I + 1
    K := K + 2
    J := J + K
END WHILE
Z := I
END

Figure 2.10. The square root function for proof with
Dijkstra's and Hoare's methods
Axiom of Assignment \( \text{AxO.} \)

\[ |- P(f/X) \{X:=f\} P \]

Where, \( P \) is a predicate, \( P(f/X) \) is \( P \) with all occurrences of \( X \) replaced by \( f \), \( X \) is a variable identifier, and \( f \) is an expression.

Rule of Weaker Consequences \( \text{W-C.} \)

\[ \text{If } |- P \{S\} R \text{ AND } |- R \Rightarrow Q \text{ Then } |- P \{S\} Q \]

Rule of Stronger Antecedents \( \text{S-A.} \)

\[ \text{If } |- P \{S\} R \text{ AND } |- T \Rightarrow P \text{ Then } |- T \{S\} R \]

Rule of Composition \( ; \).

\[ \text{If } |- P \{S_1\} T \text{ AND } |- T \{S_2\} R \]
\[ \text{Then } |- P \{S_1;S_2\} R \]

Rule of Iteration \( \text{WH.} \)

For the statement \( \text{WHILE } B \text{ DO } S. \)

Suppose that \( P \) is an assertion which is always true before \( S \).

\[ \text{If } |- (P \text{ AND } B) \{S\} P \]
\[ \text{Then } |- P \{\text{WHILE } B \text{ DO } S\} \left( \neg B \text{ AND } P \right) \]

The proof of the square root function is now presented. Only the most important steps are included.

\[ |- X \geq 0 \{I:=0\} (X \geq 0 \text{ AND } I=0) \text{ By } \text{AxO.} \] (1)
D. The Weakest Precondition Method

The Weakest Precondition method, the final static verification approach reviewed, will be used as the foundation for the KL-1 verification system. The original work, by Dijkstra, which explains weakest preconditions and applies them to a mini-language can be found in (8). A later
work which applies this method to the control constructs of KL-1 is (37). The details necessary for understanding the proof of the square root function are given in this section and the application of weakest preconditions to KL-1 is summarized in Appendix B.

This proof technique is quite similar to Floyd's Flowchart approach in many respects, but it is built on a programming language rather than flowchart operations. As the Flowchart method, the weakest Precondition method is based on a set of predicate transformers. One transformer is defined for each type of statement in the language. These transformers are grouped together to form the weakest precondition function. This function changes a predicate which describes the desired result of executing a statement into a predicate that characterizes the necessary initial states. The desired result is called the postcondition and the necessary initial states are characterized by the precondition. If the statement is executed from one of these initial states, it is guaranteed to terminate giving the desired result.

For a given postcondition the function yields the weakest precondition; therefore, the function has been called the weakest precondition function. The function has two arguments, a statement and the postcondition; it returns the weakest precondition. This predicate describes all states
and only those states in which activation of the statement will produce the desired result. The term weakest is used to indicate that the precondition places the least amount of constraint on the set of initial states. That is, the precondition describes the largest set of states.

The static approach to verification, using the weakest precondition function, begins with the program's source. Added to the source is an output assertion and an input assertion which describe the intended operation of the program. Next, the weakest precondition for the program is found by invoking the weakest precondition function with arguments the program and the specified output assertion. The resulting weakest precondition describes all the initial states in which the program can begin execution and be guaranteed to halt satisfying the output condition. This precondition is subsequently compared with the specified input assertion. If all the states which satisfy the input assertion also satisfy the generated weakest precondition, then the program is verified.

To find the weakest precondition for an entire program, the function must be defined for each statement type used in the program. In other words, a weakest precondition predicate transformer must be defined for each statement in the language. In addition, a mechanism must be available which provides a precondition for combined statements (an
entire program) rather than only one statement.

Figure 2.10 is the square root program as represented for proof with weakest preconditions. The portions of the weakest precondition function needed for the proof of this program are now introduced.

Suppose that $S$ is some program statement and $R$ is a predicate over the variables manipulated by $S$. The weakest precondition for execution of statement $S$ with desired result $R$ is usually denoted $\text{WP}(S, R)$, i.e. the invocation of the weakest precondition function $\text{WP}$ with arguments $S$ and $R$.

1. Sequential composition of statements

Since programs are built of many individual statements, it is necessary to find the weakest precondition for a group of statements with a single desired result. For this purpose, the rule for sequential composition of statements is defined. This rule provides a means of separating a program so its statements may be treated individually.

If $S_1$ is a statement or sequence of statements and $S_2$ is a statement, then the sequential composition of these statements is denoted by a semicolon as,

$$S_1;S_2.$$

The weakest precondition of these statements with the desired result $R$,

$$\text{WP}(S_1;S_2, R)$$

is defined to be
WP(S1, WP(S2, R)).

A semicolon (;) is used to denote the use of this relation in a program's proof. The weakest precondition for the two statements S1;S2 is the weakest precondition of the first, S1, where the postcondition is the set of states in which activation of S2 will give the result R. From this definition, it can be seen how the desired postcondition for a program is pushed up through the program to generate a precondition.

2. Simultaneous substitution

To define the transformer for the assignment statement, a substitution operator is needed. For any predicate R, the substitution

\[ R(A/X) \]

denotes the predicate obtained by substituting A for all occurrences of X in R. Throughout this thesis, A will be an expression of the programming language and X will be an identifier used in the program.

This operator can be generalized to a multiple simultaneous substitution. The substitution

\[ R(A_1/X_1, A_2/X_2, \ldots, A_n/X_n) \]

denotes the replacement in R of each A for its respective X. The substitution is simultaneous; if an X occurs in some A, then no substitution is made for that instance of X.
3. **The assignment statement**

The weakest precondition for an assignment statement is a predicate which must be true prior to execution of the statement to assure truth of the desired result when the assignment completes.

The general form of the KL-1 assignment statement is

\[ \text{VAR} := \text{EXP}. \]

The expression EXP is evaluated and the result is associated with the variable VAR. For any predicate \( R \), the weakest precondition of an assignment with desired result \( R \), written

\[ \text{WP}(\text{VAR}:=\text{EXP}, R), \]

is defined to be the substituted predicate

\[ R(\text{EXP}/\text{VAR}). \]

The assignment operator (\( := \)) denotes the use of this transformer in a program proof.

The mechanics of this transformer can be seen in a trivial example. Suppose that after the execution of the statement

\[ A := A + 5, \]

the condition \( A > 5 \times B \) should be true. The weakest precondition

\[ \text{WP}(A:=A+5, A>5*B) \]

is the condition which must be true prior to the assignment to assure this result. By the transformer above, this is the
substitution

\[(A > 5 \times B) \land (A + 5 / A)\]

which is the predicate \(A + 5 > B \times 5\).

4. **A theorem for loops**

Another property of the weakest precondition function which Dijkstra calls monotonicity (Property 2), is essential in proofs involving loops.

For all predicates \(Q\) and \(P\) and all statements \(S\),

\[
\text{If } Q \Rightarrow R \text{ Then } WP(S,Q) \Rightarrow WP(S,R).
\]

The symbol \(\Rightarrow\) denotes logical implication and the property states that the weakest precondition function preserves implication.

The final property of the weakest precondition function required for the proof of the square root program is the theorem for loops. Dijkstra presents and proves a version of this theorem as it applies to the repetitive construct of his mini-language (9). The same theorem is presented here, but is changed by applying it to the repetitive construct \(\text{WHILE-DO}\) and eliminating the condition assuring termination.

When nontrivial loops exist in a program, the predicate transformer cannot be applied to determine a weakest precondition. To determine the weakest precondition, the number of iterations taken would have to be independent of the program variables; this is normally not the case.

Instead of directly computing a precondition for a loop, an
inductive argument is used. An invariant proposition which is true for each loop iteration is found, then induction on the number of iterations is employed. The theorem gives the conditions necessary to prove that the inductive assertion is a precondition for the WHILE-DO loop. The invariant need not be the weakest precondition and more commonly it is a stronger assertion.

In the theorem, the boolean expression of the WHILE-DO is abbreviated as B, the statement body as SL, and the invariant as P. The theorem is

\[
\text{if } (P \land B) \Rightarrow WP(SL, P) \quad \text{then } P \Rightarrow WP(\text{WHILE } B \text{ DO } SL, P \land \neg B).
\]

5. **The proof**

The proof of the square root function (Figure 2.10) is now presented using the Weakest Precondition method. Two abbreviations are made; the first three statements of the program are referred to as INIT and the loop is called WHILE-DO.

\[
WP(INIT;\text{WHILE-DO};Z:=I, (Z**2\leq X) \text{ AND } (X<(Z+1)**2))
\]

\[
= WP(INIT;\text{WHILE-DO},(I**2\leq X) \text{ AND } (X<(Z+1)**2)) \text{ BY := .}
\]

\[
= WP(INIT,WP(\text{WHILE-DO},(I**2\leq X) \text{ AND } (X<(Z+1)**2))) \text{ BY ; .}
\]

Let the loop invariant P be

\[
P = ((I**2\leq X) \text{ AND } (J=(I+1)**2) \text{ AND } (K=2*I+1)) .
\]

By substitution and manipulation,
(P AND ¬(J≤X)) => (((I**2≤X) AND (J=(I+1)**2) AND (K=2*I+1))
AND (X<J))
so (P AND ¬(J≤X)) => (((I**2≤X) AND (X<(I+1)**2)) .
Using the property of monotonicity,
WP(WHILE-DO, P AND (J>X)) => WP(WHILE-DO, (I**2≤X) AND
(X<(I+1)**2)) .
We choose to calculate the antecedent
WP(WHILE-DO,P AND J>X) .
From the lemma proven below, we find that
P => WP(WHILE-DO,P AND J>X) .
Again using monotonicity,
WP(INIT,P) => WP(INIT,WP(WHILE-DO, P AND J>X)).
We choose the antecedent to calculate,
WP(INIT, P)
= WP(I:=0;J:=1,(I**2≤X) AN: (J=(I+1)**2) AND (I=0))
= WP(I:=0,(I**2≤X) AND (1=(I+1)**2) AND (I=0))
= 0≤X BY several applications of := and ; .
The following implication has been shown
(X≥0) => WP(INIT;WHILE-DO;Z:=I,(Z**2≤X AND X<(Z+1)**2)) .
X≥0 is a precondition for execution of the program, but it is
not necessarily the weakest precondition.

Proof of Lemma.
To apply the theorem for the WHILE-DO, it must be shown
that
(P AND (J≤X)) => WP(I:=I+1;K:=K+2;J:=J+K, P)
to conclude

\[ P \Rightarrow WP(\text{WHILE-DO}, P \text{ AND } J \geq X) . \]

Using ; and := , it follows that

\[ WP(I := I + 1; K := K + 2; J := J + K, P) \]
\[ = WP(I := I + 1; K := K + 2, (I \times 2 \leq X) \text{ AND } (J + K = (I + 1) \times 2) \text{ AND } (K = I \times 2 + 1)) \]
\[ = WP(I := I + 1, (I \times 2 \leq X) \text{ AND } (J + (K + 2) = (I + 1) \times 2) \text{ AND } (K + 2 = I \times 2 + 3)) \]
\[ = ((I + 1) \times 2 \leq X) \text{ AND } (J + (K + 2) = (I + 2) \times 2) \text{ AND } (K + 2 = (I + 1) \times 2 + 1) . \]

By algebraic manipulation, it can now be shown that

\[ (P \text{ AND } (J \leq X)) \Rightarrow (((I + 1) \times 2 \leq X) \text{ AND } (J + (K + 2) = (I + 2) \times 2) \text{ AND } (K + 2 = 2 \times (I + 1) + 1)) . \]

The theorem for the WHILE-DO can be applied to obtain the result

\[ P \Rightarrow WP(\text{WHILE-DO}, P \text{ AND } J \geq X) . \]

This concludes the proof of the lemma. Because this lemma uses the WHILE-DO theorem, the proof does not show that the program halts.
III. PROCEDURES

The weakest precondition mechanisms for handling procedure declaration and invocation are presented in this chapter. These mechanisms provide a means for constructing formal proofs of programs which contain procedures. Although this chapter deals with procedure facilities which are not in KL-1, it does include the KL-1 procedures.

The syntactic framework for defining and calling procedures and functions in KL-1 is reviewed first. Then, the machinery for verifying a simplified class of both internal and external procedures is presented. In the case of internal procedures, both the dynamic and static identifier referencing mechanisms are discussed. In the last section of this chapter, the approach to simple procedures is extended to accommodate a more general class. Finally, proof methods for function type procedures are given.

Throughout this chapter, examples of KL-1 procedures and their proofs are presented. To aid in understanding these examples, the syntactic structure of KL-1 is presented in Appendix A. The example proofs use the weakest precondition mechanisms for KL-1. These mechanisms have already been discussed and are presented in complete form in Appendix B.

The general form of a KL-1 procedure declaration is:
This declaration is for procedure L with three formal parameters X, Y, and Z and local variables N and O. SL is the set of executable statements of L and the function procedure M is declared within L. Any number of function or procedure declarations may appear within L; others might follow the definition of M.

The conventional call by reference and call by value parameter mechanisms are not supported in KL-1. Instead, a
formal parameter may be used in one of three ways. It may 
provide a value to the procedure, it may be a variable whose 
value is set by the procedure, or it may be a combination of 
these by providing an input to and an output from the 
procedure. For each formal parameter, KL-1 requires that the 
procedure heading include a specification of how it is used. 
This is done by placing the parameter in one or both of the 
inlist (IN) and outlist (OUT). Only those parameters 
declared in OUT can be changed by the procedure. In the 
procedure L, the formal parameter X is an input to L; thus, 
it provides a value which can be used only in right context. 
The formal parameter Y is an output from L and any assignment 
to Y will change the corresponding actual parameter. The 
value of Y may only be changed in L and may not appear in 
right context. The formal parameter Z serves as both input 
to and output from L. The value of Z may be used in L; 
additionally, any assignment to Z will cause the 
corresponding actual parameter to be changed when the 
assignment is made. With this syntactic framework, it is 
senseless to invoke L with an expression as the actual 
parameter for either Y or Z and it is prohibited.

A procedure is invoked by using its name with 
parenthesized arguments in a call statement; a function is 
invoked by the occurrence of its name with actual parameters 
in an expression.
Recall that *simultaneous substitution*

\[ R(A/B,C/D) \]

was defined in chapter II as the predicate \( R \) with all occurrences of \( B \) replaced by \( A \) and all occurrences of \( D \) replaced by \( C \). The notion of substitution is now expanded to include replacement in both program source and predicate transformers. Intuitively, the new form of substitution will be used for associating actual procedure parameters to their corresponding formal parameters. With simultaneous substitution, the object of the replacement, \( R \), is restricted to be a predicate. This restriction is lifted to provide a more powerful form of substitution. The new operator is denoted by square brackets, [ ], and is called *textual substitution*. The general form of this operation is

\[ A[X_1/Y_1,X_2/Y_2,\ldots,X_n/Y_n]. \]

The replacement string, \( X_1/Y_1,X_2/Y_2,\ldots,X_n/Y_n \), has the same form as in simultaneous substitution. The \( X \) component may be any expression of the language or any predicate over the variables of the program and the \( Y \)'s are identifiers of the program. The meaning of this operation is also the same; each \( X_i \) is simultaneously substituted for all occurrences of \( Y_i \) in \( A \).

The object of the replacement, \( A \), may have two forms. First, \( A \) may be a statement or sequence of statements in the language. In this case, all selected identifiers in \( A \) are
replaced by the proper expression. For example, the textual substitution

\[(A := X + 2; \ Y := X + 3 \times 7) [A + B / X]\]
denotes the statements

\[A := A + B + 2; \ Y := A + B + 3 \times 7.\]

This form of textual substitution will be used in a later section of this chapter.

The second form of the object, A, is a predicate transformer. In this case, A is a combination of logical and substitution operators which describe the effect of executing a set of statements. For example, A may be the sequence of **simultaneous substitutions**, 

\[R(X_1/Y_1)(X_2/Y_2)\ldots(X_n/Y_n),\]
defined on the predicate R. The simultaneous substitution operator, ( ), associates left, so the above assertion is found by first replacing all occurrences of Y1 in R by X1 and then to the result, substituting X2 for Y2, and so on.

Notice that the sequence of substitutions above may produce radically different results than one simultaneous substitution

\[R(X_1/Y_1, X_2/Y_2, \ldots, X_n/Y_n).\]

For example, consider the sequence of substitutions

\[A > 5(X + B / A) (20 / B).\]

The result is \(X > -15\). However as one simultaneous substitution
A > 5(X + B/A, 20/B),
the result is X + B > 5. When textual substitution is defined
over a predicate transformer, the textual replacement is
carried out prior to any simultaneous substitutions which may
occur. Textual substitution is defined this way so that it
may be used to supply a procedure with its actual parameters.
For example, the textual substitution

\[(R(Y/X) \text{ OR } R(Z/A)(A+B/X))[3/Z,C/X]\]

denotes the predicate transformer

\[(R(Y/C) \text{ OR } R(3/A)(A+B/C)).\]

Textual substitution also associates to the left, so the
sequence of textual substitutions

\[R(A+1/X)[B/X][B>10/R]\]

evaluates as follows.

\[R(A+1/B)[B>10/R]\]
\[= B>10(A+1/B)\]
\[= A>9\]

When the definition of textual substitution is used in a
program proof, it is denoted by brackets, [], and when the
definition of simultaneous substitution is used, it is
denoted by parentheses, ()

The predicate transformer for a variable declaration is
now defined.

\[WP (\text{DECLARE VAR X OF T ENDDECLARE, R})\]
\[= R(Ut/X),\]
where $U_t$ is the undefined value of type $T$ and $R$ is any assertion. This predicate transformer assures that all variables declared in a block are initialized to an undefined value. Because of the mechanics of procedure proof, this transformer also requires that all variables be given undefined values every time the procedure is entered. In this thesis, a shorthand notation is used; instead of

$$WP(\text{DECLARE VAR } X \text{ OF } T \text{ ENDDECLARE}, R),$$

the abbreviation $WP(DCL X, R)$ is written.

A. External Procedures

One motivation for procedures is to have a completely separate name space for each routine declared. That is, the same identifier may be used in separate procedures with different meanings. In addition to this feature, external procedures are unique in the control they exert on data communication. Two external procedures have no way of accessing common data except through parameters. Thus, externally declared procedures have disjoint data spaces. The FORTRAN language is one example in which all procedures must be external. Contrary to the above restriction, the designers of FORTRAN realized that completely disjoint data spaces are not always desirable. Consequently, FORTRAN uses COMMON blocks to allow procedures to share large amounts of data.
Two procedures are considered to be external to each other when their declarations are textually disjoint. Figure 3.1 is an example of a KL-1 program in which all procedures are external. In KL-1, data may be shared by two external procedures only through parameters. The proof mechanisms for the interaction of externally defined procedures are now detailed.

The weakest precondition approach to program verification centers around the concept of a predicate transformer. For each statement type of the language, a predicate transformer exists. This transformer changes a predicate which describes the desired result of executing a statement into another predicate which describes the necessary precondition to produce that result.

There can be no single predicate transformer for procedure invocation. The necessary precondition for executing a CALL statement depends not only on the desired result, but also on the procedure being invoked and the actual parameters supplied to the procedure. Intuitively, the approach presented here is to develop a predicate transformer for each procedure of the program. This transformer details the effect of executing the procedure in terms of its formal parameters and a generalized postcondition. To find the weakest precondition for a procedure call, the CALL rule applies the predicate
PROGRAM TEST;

    DECLARE VAR X, A, B, C OF NUMBER ENDDECLARE;

    A := 5;
    C := 10;
    CALL P(A, B, C);
    X := B + C;
    POST(X = 40);

END;

END TEST

PROCEDURE P(VAR L, M, N OF NUMBER);

    IN:L, N;
    OUT:M, N;

    DECLARE VAR X OF NUMBER ENDDECLARE;

    X := N + L;
    N := X + N
    M := X;

END P
transformer to the specific postcondition and set of actual parameters. Using this approach, each procedure defined in a program is assigned a meaning by developing its predicate transformer. Each invocation of the procedure results in an application of the procedure's predicate transformer to the specifics of the call. This approach is adequate for simple procedures; however, it fails for more complex procedures and is expanded in a later section.

Consider the program A of Figure 3.2. In this figure, L and M are external procedures to the program A and POST(Q) specifies that the assertion Q is to hold after the execution of the statements SA. We require that Q may reference only variables which are defined in the declarations of A. To verify A, the weakest precondition

\[ WP(\text{PROGRAM A, Q}) \]

must be found to be the predicate TRUE. Such a result indicates that regardless of the initial state, execution of A will terminate with A's variables satisfying the predicate Q. To find this precondition, the following rule is applied. The weakest precondition of a program A with desired result Q is found by applying the WP function to the body of A. In detailing this precondition, the predicate transformers for each of the procedures declared external to A may be used. Before formalizing this rule, the creation of a predicate transformer for a procedure is discussed.
The predicate transformer for the procedure L in Figure 3.2 is written:

$$\text{EL} = \text{WP} (\text{PROCEDURE L}, R).$$

EL is read, "the effect of the procedure L". The effect of procedure L (its operational semantics) details the transformation necessary to change a postcondition for L into a precondition. This is true in the same sense that the
transformer for the assignment statement
\[ WP(VAR:=EXP, R) = R(EXP/VAR) \]
details how a precondition can be formed from a postcondition. In the same way that VAR, EXP, and R are parameters to the assignment rule, the predicate transformer for a procedure also has arguments. These arguments are the actual parameters to the procedure and the postcondition. The postcondition details the desired result of executing the procedure in terms of the identifiers accessible from the point of call. Thus, the effect of L, EL, interprets the action carried out by L in terms of a generalized postcondition (the symbol R) and L's formal parameters. EL is the predicate transformer for the procedure L and is found by using the following rule.

\[ EL = WP(PROCEDURE L, R) \]
\[ = WP(DECLARATIONS_L; SL, R) \]

R is assumed to be a predicate expressing the desired result of executing the procedure. It is assumed that R does not exist as an identifier in the procedure L. For example, the predicate transformer for procedure P of Figure 3.1 is found as follows.

\[ EP = WP(PROCEDURE P(), R) \]
\[ = WP(DCL X; X:=N+L; N:=X+N; M:=X, R) \]
\[ = WP(DCL X; X:=N+L; N:=X+N, WP(M:=X, R)) \]
\[ = WP(DCL X; X:=N+L; \hat{N}:=X+N, R(X/M)) \]
The sequence of substitutions, $R(X/M)(X+N/N)(N+L/X)(U_i/X)$, is the predicate transformer for the procedure $P$. It details how to find the weakest precondition of $P$ with the generalized result $R$ using $P$'s formal parameters.

The rule for finding the weakest precondition of a program is now formalized. The transformer for a PROGRAM depends on the desired result of the program as detailed by a POST statement. In addition, the transformer depends on the declarations and executable statements in the program as well as the specific predicate transformers for all procedures which may be either directly or indirectly invoked by the program. Using the general form for external procedures (Figure 3.2), the weakest precondition for a program is defined by the following rule.

$$WP(\text{PROGRAM } A, Q) = WP(\text{DECLARATIONS}_A; S_A, Q)$$

given $EL$ AND $EM$.

This rule states that the weakest precondition for a program can be found by detailing the weakest precondition for its body using if necessary the meanings of all procedures which may be invoked.
The CALL rule must apply the proper transformer to the specifics of the procedure invocation. In this case, the rule applies the procedure's predicate transformer to the actual parameters and the desired postcondition. The result is a precondition for the call. This rule is now presented.

PROCEDURE L(VAR X,Y,Z OF T);
  IN: X,Z;
  OUT: Y,Z;
  DECLARE VAR I,J OF T ENDDECLARE;
  SL;
  END;
END L

Figure 3.3. A single external procedure

Suppose that we have the procedure declaration of Figure 3.3. Assuming L is called, CALL L(A,B,C), from some external procedure or main block. The desire is to find the weakest precondition,

\[ WP(\text{CALL } L(A,B,C), Q), \]

where Q is a specific predicate from the calling environment. It is important to note that Q is a proposition over the variables declared in the calling environment. Q may make assertions about the actual parameters, but it may not refer to any of the variables declared in L. The rule for the call
must apply the predicate transformer for the procedure \( L \) to the actual parameters \( A, B, \) and \( C \) and to the postcondition \( Q \). The rule is defined as the following substitution

\[
EL[A/X, B/Y, C/Z, Q/R]
\]

and is referred to as \textit{EXT\_CALL}' when used in a program proof. The substitution causes the predicate transformer \( EL \) to act on the actual parameters \( A, B, \) and \( C \) instead of the formal parameters and to act on the actual postcondition \( Q \) rather than the symbolic postcondition \( R \). It must be emphasized that the substitution of the actual postcondition must take place simultaneously with or after the argument substitution. This requirement is necessary when \( Q \) contains a reference to a variable declared in the calling environment which has the same name as a formal parameter. A hypothetical substitution which demonstrates this situation is

\[
EG[X/B, (A\times B>5)/R].
\]

The identifier \( X \) should not be substituted for \( B \) in the postcondition \( A\times B>5 \).

A problem caused by the disjoint name spaces of external procedures must be resolved. Commonly, an identifier defined in a procedure may also be declared in the calling environment. This variable may occur in the actual postcondition and may also be manipulated by the procedure's predicate transformer. Thus, the predicate transformer may
change a variable occurrence in the postcondition which should not be changed since it refers to a variable declared in the calling environment. To illustrate this, suppose we have the following program definition.

PROGRAM B

DECLARE VAR A, B OF NUMBER ENDDECLARE;

...
CALL X(B);
POST(A>5);
END;
END L

PROCEDURE X(VAR Y OF NUMBER)
IN: Y;
OUT: Y;

DECLARE VAR A OF NUMBER ENDDECLARE;
A := Y;
Y := A + 3 * Y;
END;
END X

Note that the problem occurs for the call CALL X(B) and that the call is incorrectly verified with the rule EXT_CALL'. The predicate transformer for the procedure X is: EX = WP(PROCEDURE X, R)
\[
= P(A+3*Y/Y) (Y/A) (Ut/A, Ut/C)
\]
Using the textual substitution from EXT_CALL', we would get:

\[
W(P(CALL X(B), A>5))
\]

\[
= R(A+3*Y/Y) (Y/A) (Ut/A, Ut/C)[B/Y, A>5/R]
\]

\[
= A>5 \ (A+3*B/B) (B/A) (Ut/A, Ut/C)
\]

\[
= A>5 \ (B/A) (Ut/A, Ut/C)
\]

\[
= B>5
\]

This precondition is incorrect. The A in the postcondition A>5 refers to the A declared in program B. The occurrences of A in the predicate transformer

\[
R(A+3*Y/Y) (Y/A)
\]
s should refer to the A declared in the procedure X and not to the A declared in the calling environment. The weakest precondition for the call should be A>5 since the A declared in program B is not changed by X.

Similiar problems occur when an actual parameter has the same name as a variable declared in the procedure. Problems of this nature are caused when a single identifier has separate meanings in separate procedures. The CALL rule must be changed to eliminate these problems. This is done by renaming all variables declared in the procedure. The modified external call rule, EXT_CALL, for the procedure L of Figure 3.3 is:

\[
WP(CALL L(A, B, C), Q)
\]

\[
= EL[ A/X, B/Y, C/Z, UN(I)/I, UN(J)/J, Q/R].
\]
This rule is the same as EXT_CALL' except that the substitutions UN(I)/I, UN(J)/J have been added. The function UN is used to create a unique identifier name to correspond with its argument. UN(J) may return J' on one call and J'' on a subsequent call. The process of substituting a unique name for all local variables manipulated by a procedure eliminates the problem of the same identifier name referring to separate variables. In terms of describing the meaning of procedure mechanisms, EXT_CALL states that the same variable names (identifiers) used in separate procedures refer to different variables.

The use of the unique identifier function, UN(id), to create new names for all variables declared in a procedure eliminates the possibility of a redeclared identifier referencing storage in the calling environment; however, a reference to an undeclared variable (seemingly, a nonlocal reference) in a procedure may affect that variable as declared in the calling environment. Since externally declared KL-1 procedures share data only through parameters, the proof rules must reflect the fact that no nonlocal references may exist. To prevent this situation and equivalently require that all local variables be declared, the following restriction is imposed. The only variables that a procedure may reference are those it declares and the parameters passed to it. The enforcement of this restriction
by the proof rules is now discussed in terms of procedure L of Figure 3.3 and the call, CALL L(A,B,C).

The only identifiers from the calling environment which may be referenced in L are the actual parameters A, B, and C. To insure this, we require that the intersection of the free variables occurring in the substituted transformer

\[ E_L[A/X, B/Y, C/Z, \text{UN}(I)/I, \text{UN}(J)/J] \]

with the free variables in the postcondition Q must be included in the set \{A,B,C\}. Since the formal parameter X provides an input to the procedure, it is possible that the actual parameter A is an expression. In this case, the intersection must be included in the set \{B,C\}.

This section is concluded with a proof of the program in Figure 3.1.

We want \( WP(\text{PROGRAM TEST, } X=40) \)

\[ = WP(\text{DCL } X, A, B, C; A:=5; C:=10; \text{CALL } P(A,B,C); X:=B+C, X=40) \]

GIVEN \( EP \)

\[ = WP(\text{DCL } X, A, B, C; A:=5; C:=10; \text{CALL } P(A,B,C), B+C=40) \]

BY \( ; \) AND \( := \).

\[ = WP(\text{DCL } X, A, B, C; A:=5; C:=10, WP(\text{CALL } P(A,B,C), B+C=40)) \]

BY \( ; \).

(1)

We now find \( WP(\text{CALL } P(A,B,C), B+C=40) \)

\[ = EP[A/L, B/M, C/N, \text{UN}(X)/X, B+C=40/R] \] BY EXT_CALL.

The predicate transformer for \( P \) was found previously to be

\( EP = R(X/M) (X+N/N) (N+L/X) (U_i/X) \) so
\[ WP(CALL P(A,B,C), B+C=40) \]
\[ = EP[A/L,B/M,C/N,UN(X)/X, B+C=40/R] \]
\[ = R(X/M) (X+N/N) (N+L/X) (Ui/X)[A/L,B/M,C/N,UN(X)/X, B+C=40/R] \]
\[ = B+C=40 (X''/B) (X''+C/C) (C+A/X") (Ui/X") BY [ ]. \]
\[ = X''+C=40 (X''+C/C) (C+A/X") (Ui/X") BY (). \]
\[ = X''+X''+C=40 (C+A/X") (Ui/X") BY (). \]
\[ = C+A+C+A+C=40 \]
\[ = 3C + 2A = 40 \] (2)

Substituting (2) in for \( WP(CALL P(A,B,C), B+C=40) \) in (1) we get:
\[ WP(DCL X,A,B,C; A:=5; C:=10; CALL P(A,B,C), B+C=40) \]
\[ = WP(DCL X,A,B,C; A:=5; C:=10, 3C+2A=40) \]
\[ = WP(DCL X,A,B,C, 30+10=40) \]
\[ = TRUE \]

This concludes the proof of Figure 3.1.

B. Internal Procedures

External procedures offer a means of declaring disjoint data spaces in a program. In KL-1, all communication between external procedures is accomplished by parameters. However as many language designers have recognized, there is often a need to share more data than can be conveniently passed via parameters. This need, along with the desire to limit the accessibility of certain subroutines, has resulted in the use of internal procedures. An internal procedure is one whose declaration textually appears within another routine.
Internally defined procedures may reference some identifiers which are neither declared within nor passed as parameters. Further, they may be used to hide subprogram definitions from other program parts.

KL-1 includes internally defined procedures to augment the data communication facilities provided by external routines. This section is devoted to verification techniques for programs which use internal procedures. The verification of internal procedures is complicated by their ability to access identifiers declared in another environment. Such accesses are called nonlocal references and are now discussed in more detail.

An identifier for the purposes of this treatment is a name for a variable, an array, a formal parameter, or a procedure.

A binding (or an association) is a pair \((X, p)\) where \(X\) is an identifier and \(p\) represents the storage or program element which \(X\) names.

CREATE and DESTROY, two operations involving associations, are important for this discussion. In KL-1, the CREATE operation takes place when the declaration statements of a procedure are encountered. When a KL-1 program element (main program or procedure) is entered, an association is created for each identifier declared.
In most high level languages, bindings are created on block entry, but there are exceptions. Associations for the PL/1 CONTROLLED and BASED storage classes are one example. In this case, encountering the declaration of a variable which is either CONTROLLED or BASED does not cause a binding to be created, but forms only a name for that variable. The binding is created later when an ALLOCATE statement is executed.

The second binding operation is the DESTROY operation. This operation removes an association from the system. The associations created on invocation remain in the system until block exit. If control should temporarily leave the block, for example if another procedure is called, the associations still exist on return of control. When a procedure or program is exited, all associations created in that block are destroyed.

When executing a program, it is common to have more than one active association for a given identifier. For example, consider the program L in Figure 3.4. The second time execution reaches the assignment statement

\[ T := B \times 5 \]

two bindings for the identifier T exist. The second execution of this assignment is a result of the call,

\[ \text{CALL Q(A)}, \]

in program L. One of these bindings was created when the
execution of L began and the other was created when the procedure Q was called from L. The question posed at this point is: Which of these bindings should be used for the reference to T in the assignment statement? An answer is provided by the language's identifier binding strategy. This strategy is the set of rules built into a language to resolve nonlocal references to the proper association. The two most common strategies, static and dynamic (most recent), are now considered.

The static binding strategy is employed by most block structured languages including ALGOL, PL/1, and PASCAL. The textual form of the program determines which association is used when this strategy is employed. To resolve a nonlocal reference, each containing block is examined. The search begins at the innermost block which contains the reference and continues outward. The search ends when a block is encountered which has an association for that identifier and that binding is used. This strategy creates a one way hierarchical structure for nonlocal referencing, because the accessibility of a name extends to all procedures internal to the block declaring it. With this strategy, the reference to T in the assignment in question (Figure 3.4)

\[ T := B \times 5 \]

is resolved to the outermost association created on entry to L.
PROGRAM L;

DECLARE VAR T, A OF NUMBER ENDDECLARE;
A := 1;
CALL R(A);
CALL Q(A);
POST(T=5 AND A=56); // FOR DYNAMIC BINDING.//
END; //T=25 AND A=56 FOR STATIC.//

PROCEDURE R(VAR B OF NUMBER);
IN: B;
OUT: B;
T := B * 5;
B := T;
END;
END R;

PROCEDURE Q(VAR X OF NUMBER);
IN: X;
OUT: X;

DECLARE VAR T OF NUMBER ENDDECLARE;
CALL R(X);
T := X + 5;
X := T + X + 1;
END;
END Q;

END TEST

Figure 3.4. Resolving nonlocal identifier references
The most notable languages which employ the dynamic strategy are APL, LISP, and SNOBOL. With this strategy, the binding used to resolve an identifier reference is the one most recently created. With the dynamic strategy, the reference to T from within procedure R (the second time R is called in Figure 3.4) uses the binding for T created on entry to Q.

Suppose a given identifier reference is encountered several times in the execution of a program. The dynamic strategy may resolve that reference to a different declaration each time. On the other hand, when the static strategy is employed a given reference will always resolve to the same identifier declaration. Each reference is resolved based only on the static program structure.

A liberty is being taken which should be explained. A reference does not resolve to an identifier declaration, rather it resolves to one of many associations for a declaration. Since a procedure may be entered and left many times during the execution of a program, several associations will be created and destroyed for each of its variables. We are concerned with the declaration whose association a reference uses. Consequently, we say that a reference resolves to a declaration.

The programming language EUCLID is interesting, because it was designed so that its programs may be proven correct.
The Axiomatic method of static verification has been applied to EUCLID by London (40). The proof rules London gives for EUCLID's procedures are based on Hoare's work on procedures and parameters (23). To facilitate the proof of procedures, certain restrictions have been placed on the language. One of these restrictions limits the nonlocal referencing environment for a procedure. In EUCLID, references to identifiers declared outside a procedure are controlled by an IMPORT list. All nonlocal identifiers accessed in a procedure must appear in the list. The IMPORT list provides the ability to enforce the following rule. No procedure may invoke a routine which imports identifiers not accessible from the point of call.

This restriction reduces the nonlocal environment for EUCLID procedures so that the static and dynamic strategies produce the same results. As other block structured languages, EUCLID specifies that nonlocal references are bound using the static strategy. Because of the rules imposed by the IMPORT list, the nonlocal environment, although based on the static strategy, is restricted to the point that the two strategies are indistinguishable. That is, all valid EUCLID programs will operate the same using the static strategy as the dynamic strategy.

The program in Figure 3.4 is an example in which the two strategies produce separate results. This program would not
be a legal EUCLID program; it violates the above rule. By this rule, the procedure Q may not invoke R since R references the variable T (declared in L) to which Q does not have access.

The remainder of this section details the weakest precondition mechanisms for procedures which do not limit nonlocal references in the context of the static and dynamic strategies. The reader should not infer from the treatment of these strategies that the language KL-1 provides both; KL-1 uses the static strategy. The justification for treating both is two-fold. First, it seems desirable to have proof rules for procedures which do not depend on restricting the nonlocal referencing environment. The proof mechanisms which follow do lift this restriction. No attempt is made to evaluate how these restrictions affect the "complexity of a proof". When a language is designed for verifiability, this evaluation should be made and assessed with regard to the effect the restrictions have on the "flexibility of expression". The second justification for this treatment is that the mechanisms presented define the operational semantics of procedure use for both the static and dynamic referencing strategies.
1. Dynamic resolution

Previously, the approach has been to construct a predicate transformer for each procedure used in a program and then apply the transformer to every invocation of the procedure. This approach has the advantage that only one scan through a procedure is necessary to verify any number of calls. Unfortunately, a predicate transformer cannot be used when the dynamic strategy is employed; by examining procedure identifiers, it becomes clear that an alternative to the predicate transformer approach must be taken.

The ability to use a single predicate transformer to prove all invocations of a procedure is based on a key assumption. That is, the procedure has the same meaning (operational characteristics) every time it is called. This assumption no longer holds when procedure identifiers are bound to an association using the dynamic strategy. Figure 3.5 is an example which illustrates this.

In this figure, the reference to the procedure A inside of B is resolved to a different A each time B is called. B is called once from an environment in which the outermost procedure A will be used and once when the A declared internal to procedure C will be used. A predicate transformer which describes the effect of calling procedure B cannot be detailed since the statements within the procedure
PROGRAM TEST

DECLARE VAR X OF NUMBER ENDDECLARE;
X := 5;
CALL B;
CALL C;
POST(X=12);  //X=12 FOR DYNAMIC BINDING.//
END;  //X=13 FOR STATIC BINDING.//

PROCEDURE A();
    X := X + 3;
END A;

PROCEDURE B();
    X := X + 1;
    CALL A;
END B;

PROCEDURE C();
    CALL B;
END;

PROCEDURE A();
    X := X + 2;
END A;

END C;

END TEST

Figure 3.5. Resolving procedure references
may have a different meaning for every invocation. Clearly, the meaning of procedure B cannot be defined by a predicate transformer when the dynamic strategy is used. The approach to verifying procedure invocation with this strategy is to detail the meaning of a procedure each time it is invoked.

The question naturally arises: How can a static verification method accurately model a binding strategy which relies on the dynamic nature of the program? It has already been established that separate executions of an identifier reference may be resolved to different declarations when the dynamic strategy is employed.

By the static structure of a program, we can determine which declaration each execution of a reference will use. The truth of this claim depends on two assumptions made about the language. First, the language does not allow procedure variables or procedures as parameters. Secondly, the language does not support a dynamic storage allocation feature such as PL/1's CONTROLLED and BASED storage classes. To show that the claim is true, two observations must be made.

1. When an identifier reference is executed, the procedure history is all that is needed to resolve it. This history details all blocks which will be active when the reference is made. With the assumption that dynamic storage allocation is not supported, the current associations can be
determined from the history since the creation and
destruction of bindings depend only on block entry and exit.
Certainly given the associations, the reference can be
resolved and the first observation is valid.

2. The second observation is that the procedure history
can be determined by static program analysis. By the
assumption that procedure identifiers may not be variables,
statements of the following form are illegal and do not
complicate the analysis.

DECLARE VARIABLE A OF PROCEDURE(NUMBER);
  READ A;
  CALL A(20); ...

This assumption permits determination of all possible
execution paths through a program from its text. From the
execution path, the procedure history can certainly be
derived. Thus, the declaration that an identifier reference
will use can be determined from the static structure of the
program.

The problem of detailing the weakest precondition for a
statement which may contain dynamically bound nonlocal
references is now considered. Suppose the weakest
precondition for the assignment (in procedure R of Figure
3.4)
\[ T := B \times 5 \]

with the postcondition

\[ T = 5 \]

is desired. The variable \( T \) in this assignment is certainly a nonlocal reference. As stated above, this reference resolves to different declarations on separate executions of the assignment. To calculate this weakest precondition, we must know whether the \( T \) in the assignment resolves to the same declaration as the \( T \) in the postcondition. To determine if the same declaration for \( T \) is used, the predicate transformer needs more information than just the statement and desired result. Additionally, the active associations are needed. This information is conveyed by a substitution list (environment) which is passed to the transformer as an extra argument. The weakest precondition for any set of statements \( S_1 \) in the environment \( A \) with the desired result \( R \) is denoted as

\[ WP(S_1, \{A\}, Q). \]

\( A \) is the substitution list and is also called the association list or the environment. \( A \) consists of a sequence of replacements in the same form as the textual and simultaneous substitution operators. This list maintains the set of identifiers which constitute the local and nonlocal referencing environments and can be viewed as providing the procedure history to each statement executed. When a
procedure is entered, the list is updated by the new associations created in that block. In addition, a textual substitution is made which resolves all references in the block to the proper declaration.

The weakest preconditions for language statements must be altered to use the environment parameter. In fact, the predicate transformers for language constructs are only slightly changed by the additional argument. All manipulation of the substitution list is done by the transformer for procedure call. The predicate transformer for the assignment statement is unchanged by the addition of the association list. For example,

\[ WP(VAR:=EXP, [E], R) = WP(VAR:=EXP, R) = R(EXP/VAR). \]

The rule for the sequential composition of statements is changed to pass the same environment to each invocation of the function WP. The composition of two statements S1 and S2 is:

\[ WP(S1; S2, [E], R) = WP(S1, [E], WP(S2, [E], R)). \]

The predicate transformers for the repetitive and alternative statements are extended in a similar way to accommodate the association list. For example,

\[ WP(IF B THEN ST ELSE SF ENDIF, [E], R) = (B AND WP(ST, [E], R)) OR (\neg B AND WP(SF, [E], R)). \]

The association list is not changed for these constructs and is passed on by the recursive calls. Changes of this nature
must be made to all transformers when the dynamic strategy is employed. Since these changes are straightforward, they are not presented.

The rules for dynamic resolution of variable and formal parameter references are now given. These rules are then extended to include procedure identifiers. The predicate transformers for procedure call and procedure entry make necessary changes to the association list. The predicate transformer for a procedure call adds the actual parameters to the list and passes the modified list to the procedure.

```
PROCEDURE L(VAR X,Y,Z OF T)
    IN: X,Z;
    OUT: Y,Z;
    DECLARE VAR I,J OF T ENDDECLARE;
    SL;
    END;
PROCEDURE D(VAR W OF T)
    IN: W;
    DECLARE VAR K OF T ENDDECLARE;
    SD;
    END;
END D;
END L
```

Figure 3.6. A general form for internal procedures
Consider the procedure declaration in Figure 3.6. The weakest precondition for calling procedure L with desired result Q given the current environment E is:

$$WP(CALL\ L(A,B,C),\ E,\ Q)$$

$$= WP(PROCEDURE\ L,\ {A/X,B/Y,C/Z,F},\ Q)$$

where the association list F is derived from the list E by removing any entries for the identifiers X, Y, and Z. In program proofs, the use of this rule will be denoted by D.1. The transformer for the call updates the environment to include the actual parameters and invokes the transformer for a procedure. The statement argument to the WP function, PROCEDURE L, is intended to represent the entire program element for the procedure L. Thus in defining the weakest precondition function for a statement of type PROCEDURE, all of its declarations and body are accessible.

The weakest precondition for procedure entry given procedure L,

$$WP(PROCEDURE\ L,\ E,\ Q),$$

is responsible for updating the association list E by all identifiers declared in L. This transformer performs the operation CREATE binding (and implicitly DESTROY binding) by defining new associations for each identifier declared. In addition, it makes these bindings accessible to L and all procedures which may be called by L. This is done by creating unique identifier names and substituting those names
for all references to the declared variables. For each identifier declared, a unique name for that identifier is found using the function UN(id). This identifier is then substituted into the text for all references which resolve dynamically to that declaration. Suppose UN(I)=I' and UN(J)=J' then, the weakest precondition for the procedure L of Figure 3.6 is defined as follows.

\[
WP(\text{PROCEDURE L}, \{E\}, Q) = WP((DCL I, I; SL)[I'/I, J'/J, F], \{I'/I, J'/J, F\}, Q)
\]

where F is derived from E by removing any associations for either I or J. In program proofs, the use of this rule will be denoted by D.2'. It details the weakest precondition for execution of procedure L coming from environment E with desired result Q. It is important to note that the actual parameters have already been placed into the association list by the CALL rule D.1. The effect of D.2' is two-fold.

First, it updates the environment to include all local and nonlocal identifiers which are accessible. The local identifiers are placed in the list as unique names. If a substitution already exists for a name, then the new one is added and the old one is removed. Since this is done whenever a name is added, the list never has multiple substitutions for a single identifier; it contains only the most recent associations.
Secondly, this transformer performs a textual substitution of all the names in the association list into the procedure body. This has the effect of resolving all references (including nonlocal references) in the procedure. The predicate transformer for the CALL statement, D.1, will pass these associations to any procedure which is called from L.

The final predicate transformer for the dynamic strategy is used to initiate the process of finding the weakest precondition for a complete program with some desired result. Since this is the transformer which initializes the environment, the invocation of the function WP only has the program's text and the specified output assertion as arguments. The predicate transformer is

\[ WP(PROGRAM P, Q) \]

where \( P \) is the name of a program and \( Q \) is the desired postcondition. The transformer is

\[ WP(PROGRAM P, Q) = WP(DECLARATION_P;SP, \{ \}, Q). \]

When used in a program's proof, this transformer will be referenced by D.3'. This transformer specifies that the identifiers declared in the program \( P \) do not need the unique name operator UN. Therefore, the null association list specifies that all identifier references in \( P \)'s body, SP, are local.
To illustrate the transformers D.1, D.2', D.3', the program of Figure 3.4 is now proven. In this figure, the references to the variable T from within procedure R resolve to different bindings for T on separate calls to R. The first time R is called from the body of L the references to T use the binding created in L. R is called a second time by procedure Q and the references to T from this call use the binding created in Q. In the next two proofs, use of the assignment transformer, :=, and the composition relation, ;, refer to the transformers which have been altered to accommodate the environment.

\[
WP(\text{PROGRAM L, T=5 AND A=56})
\]
\[
= WP((\text{DCL T, A; A:=1; CALL R (A); CALL Q(A)}), \{\},
\]
\[
T=5 \text{ AND A=56}) \text{ BY D.3'}. 
\]
\[
= WP((\text{DCL T, A; A:=1; CALL R (A)}), \{\}, WP(\text{CALL Q(A)}, \{\},
\]
\[
T=5 \text{ AND A=56})) \text{ BY ;}. \tag{1} \]

We use D.1 and D.2' to find

\[
WP(\text{CALL Q(A)}, \{\}, T=5 \text{ AND A=56})
\]
\[
= WP(\text{PROCEDURE Q, [A/X]}, T=5 \text{ AND A=56}) \text{ BY D.1}. 
\]

Assuming UN(T) is T'
\[
= WP((\text{DCL T; CALL R(X); T:=X+5; X:=T+X+1)[T'/T,A/X]},
\]
\[
[T'/T,A/X], T=5 \text{ AND A=56}) \text{ BY D.2'}. 
\]
\[
= WP((\text{DCL T'; CALL R(A); T':=A+5; A:=T'+A+1)},[T'/T,A/X],
\]
\[
T=5 \text{ AND A=56}) \text{ BY [}]. 
\]
\[
= WP((\text{DCL T'; CALL R(A)}),[T'/T,A/X], T=5 \text{ AND 2A+6=56}) 
\]
BY ; and:=.

= WP(DCL T', {T'/T, A/X}, WP(CALL R(A), {T'/T, A/X},
   \[T=5 \text{ AND } 2A=50]) \text{ BY ::.} \hspace{1cm} (2)

We now find

WP(CALL R(A), {T'/T, A/X}, T=5 AND 2A=50)
= WP(PROCEDURE R, {A/B, T'/T, A/X}, T=5 AND 2A=50) \text{ BY D.1.}
= WP((T:=B*5; B:=T)[A/B, T'/T, A/X], {A/B, T'/T, A/X},
   \[T=5 \text{ AND } 2A=50]) \text{ BY D.2'.}
= WP((T':=A*5; A:=T'), {A/B, T'/T, A/X}, T=5 AND 2A=50)
= T=5 AND 10A=50 \hspace{1cm} (3)

Note that the references to T in R use Q's binding for T.

Substituting (3) into (2)

WP(CALL Q(A), {}, T=5 AND A=56)
= WP(DCL T', {T'/T, A/X}, T=5 AND 10A=50)
= T=5 AND 10A=50 \hspace{1cm} (4)

Substituting (4) into (1)

WP(PROGRAM TEST, Q)
= WP((DCL T, A; A:=1; CALL R(A)), {}, T=5 AND 10A=50)
= WP((DCL T, A; A:=1), {}, WP(CALL R(A), {},
   \[T=5 \text{ AND } 10A=50)) \text{ BY ::.} \hspace{1cm} (5)

We now find

WP(CALL R(A), {}, T=5 AND 10A=50)
= WP(PROCEDURE R, {A/B}, T=5 AND 10A=50) \text{ BY D.1.}
= WP((T:=B*5; B:=T)[A/B], {A/B}, T=5 AND 10A=56)
= WP((T:=A*5; A:=T), {A/B}, T=5 AND 10A=50)
Substituting (6) into (5):

\[
\text{WP}((\text{DCL T, A; A:=1}, \{\}, 5A=5 \text{ AND } 50A=50) \text{ BY } ; \text{ and } :=. = \text{ TRUE}
\]

This completes the proof of Figure 3.4.

This technique is now expanded to include dynamic binding of procedure identifiers. To accommodate these identifiers, an association must be made for each procedure in a block when the block is entered. This is accomplished by adding to the predicate transformer \( D.2' \) an association for each internally declared procedure. Two additions to the predicate transformer are necessary to incorporate an association. First, a unique name is generated for each procedure and added to the environment list. This causes each new procedure name to be substituted into all subsequent references. Secondly, an informal association is made between the new name for a procedure and its corresponding program element. The new predicate transformer \( D.2 \) based on procedure \( L \) in Figure 3.6 is:
\[ WP(\text{PROCEDURE} \ L, \ \{E\}, \ Q) \]
\[ = WP((DCL \ I,J; \ SL)[D'/D,I'/I,J'/J,P], \ \{D'/D,I'/I,J'/J,P\}, \ Q) \]
where: \( D' \) refers to the procedure \( D \) declared in \( L \).
\( F \) is derived from \( E \) by removing any substitutions for \( D, I \) or \( J \).
\( D', I', J' \) are unique names for the variables \( D, I, J \) respectively.

To accommodate procedure identifiers, the same change is made to the predicate transformer for a program \( (D.3') \)
\[ WP(\text{PROGRAM} \ P, \ Q) \]
Again, an association is made for each internally declared procedure. Suppose that procedures \( A \) and \( B \) are declared internal to some program \( P \). The predicate transformer for the program creates associations for these procedures informally as
\[ WP(\text{PROGRAM} \ P, \ Q) \]
\[ = WP(\text{DECLARATIONS}_P;SP, \ \{\}, \ Q) \]
where: \( A \) refers to the procedure \( A \) declared in \( P \).
\( B \) refers to the procedure \( B \) declared in \( P \).

In program proofs, the use of this rule will be denoted by \( D.3 \). Note that this rule does not need to create unique names for the procedures \( A \) and \( B \). Therefore, \( A \) and \( B \) are not placed in the substitution list.

This section is concluded with the proof of a program in which a single procedure reference is resolved to different
procedures. The program in Figure 3.5 is used. In this program, the reference to procedure A from within procedure B uses different bindings for A on separate calls to B.

\[
\text{WP(PROGRAM TEST, } X=12) \\
= \text{WP(DCL X; } X:=5; \text{ CALL B; CALL C, }\{}\text{, } X=12) \\
\text{Where: A refers to procedure A declared in TEST.} \\
B \text{ refers to procedure B.} \\
C \text{ refers to procedure C. BY D.3.} \\
= \text{WP(DCL X; } X:=5; \text{ CALL B, }\{}\text{, WP(CALL C, }\{}\text{, } X=12)\text{)} \\
\text{BY } ;. \quad (1)
\]

We now find

\[
\text{WP(CALL C, }\{}\text{, } X=12) \\
= \text{WP(PROCEDURE C, }\{}\text{, } X=12) \quad \text{BY D.1.} \\
= \text{WP(CALL B[A'/A], }\{}\text{A'/A}\text{), } X=12) \\
\text{Where: A' refers to procedure A declared in C. BY D.2.} \\
= \text{WP(CALL B, }\{}\text{A'/A}\text{), } X=12) \quad \text{BY } [\ ]. \\
= \text{WP(PROCEDURE B, }\{}\text{A'/A}\text{), } X=12) \quad \text{BY D.1.} \\
= \text{WP(X:=X+1; CALL A [A'/A], }\{}\text{A'/A}\text{), } X=12) \quad \text{BY D.2.} \\
= \text{WP(X:=X+1; CALL A', }\{}\text{A'/A}\text{), } X=12) \quad \text{BY } [\ ]. \\
= \text{WP(X:=X+1, }\{}\text{A'/A}\text{), WP(CALL A', }\{}\text{A'/A}\text{), } X=12)\text{)} \quad \text{BY } ;. \quad (2)
\]

We now find

\[
\text{WP(CALL A', }\{}\text{A'/A}\text{), } X=12 \\
\text{Notice that this reference to procedure A' binds to the procedure A declared internal to C.}
\]
= WP(PROCEDURE A', {A'/A}, X=12) BY D.1.
= WP(X:=X+2[A'/A], {A'/A}, X=12) BY D.2.
= WP(X:=X+2, {A'/A}, X=12) BY [ ].
= X=10 BY :=. (3)

Substituting (3) into (2)

WP(CALL C, {}, X=12)
= WP(X:=X+1, {A'/A}, X=10) which is:
= X=9 BY :=. (4)

Substituting (4) into (1) we get

WP(PROGRAM TEST, X=12)
= WP(DCL X; X:=5; CALL B, {}, X=9)
= WP(DCL X; X:=5, {}, WP(CALL B, {}, X=9)) BY ;. (5)

We now find

WP(CALL B, {}, X=9)
= WP(PROCEDURE B, {}, X=9) BY D.1.
= WP(X:=X+1; CALL A, {}, X=9) BY D.2.
= WP(X:=X+1, {}, WP(CALL A, {}, X=9)) BY ;. (6)

We now find

WP(CALL A, {}, X=9)

Notice that this reference to procedure A binds to the A declared in TEST.

= WP(PROCEDURE A, {}, X=9) BY D.1.
= WP(X:=X+3, {}, X=9) BY D.2.
= X=6 BY :=. (7)

Substituting (7) into (6)
WP(CALL B, {}, X=9)  
= WP(X:=X+1, {}, X=6)  
= X=5  BY :=.  

Substituting (8) into (5)  
WP(PROGRAM TEST, X=12)  
= WP(DCL X; X:=5, {}, X=5)  
= WP(DCL X, {}, 5=5)  
= TRUE  

This completes the proof of Figure 3.5 and the treatment of the dynamic referencing strategy.

2. **Static resolution**

Using the static strategy, nonlocal identifier references are resolved by searching the physically containing blocks until an association for the identifier is found. This strategy bases the decision of which binding a reference uses on the textual structure of the program only. When identifiers are resolved to a binding using the static strategy, each reference uses the same declaration throughout the execution of the program. Unlike the dynamic method in which different executions of a reference may use separate declarations, the static strategy ties each identifier reference to one declaration. Because a reference uses one declaration, all identifiers which statically resolve to the same declaration can be given a common name. To aid in verifying procedures, the proof mechanisms defined below will
rename the identifiers in a program so that the same name does not appear in two separate declarations. The same goal could be accomplished by requiring programmers to use unique identifiers in all of a program's declarations, but this is not a practical solution since there is a definite need to reuse names when writing programs.

Renaming, as a part of verification, is justified in the sense that proof techniques define the operational semantics of procedure mechanisms. When the same identifier is declared in separate blocks, it is intended to have a distinct meaning in each block. Indeed, the implementation of a translator for a block structured language includes this renaming process.

The renaming process can be used to simplify the proof mechanisms for procedures which do not limit nonlocal references and use the static strategy. Suppose that we have the program A in Figure 3.7. To calculate the weakest precondition for program A, it is first submitted to the renaming process. This process is performed by the static renaming function ST. The function ST has two arguments, the source text and level number. The text may be either an entire program or a procedure element. In either case, it is assumed that all the declarations and statements of the text are available. The modified text returned by ST is passed to the weakest precondition function.
PROGRAM A;

DECLARE VAR X OF NUMBER ENDDECLARE;

SA;

POST(Q);

END;

PROCEDURE B(VAR Y OF NUMBER);

IN: Y;

OUT: Y;

DECLARE VAR Z OF NUMBER ENDDECLARE;

SB;

END;

PROCEDURE C();

DECLARE VAR W OF NUMBER ENDDECLARE;

SC;

END;

END C;

END B;

END A

Figure 3.7. The static renaming of identifiers
The function $ST$ is defined for program $A$ of Figure 3.7 as follows.

$$ST(\text{PROGRAM } A, 0) = (\text{PROGRAM } A; \ldots; \text{END})$$

$ST$ creates a unique name for each identifier declared in a block. The name is substituted for each occurrence of the identifier in the declaring block and for each unresolved occurrence of the identifier in the procedures which are declared internal to the block. $ST$ creates unique names by extending the identifier with two objects. The extended identifier $X.0A$ refers to the identifier $X$ declared at level 0 in the block $A$. Level refers to the nesting level of the variable's declaration. Each time an internally defined procedure is entered, the level is incremented. Block indicates the procedure element in which the identifier declaration occurs. The static renaming of all procedures directly internal to $A$ is defined as follows.

$$ST(\text{PROCEDURE } B(Y); \ldots; \text{END } B, 1) = (\text{PROCEDURE } B(Y); \ldots; \text{END}; \text{PROCEDURE } C(); \ldots; \text{ENDC, 2})[Z.1B/Z,Y.1B/Y,C.1B/C] \text{ END } B;.$$  

Finally, for a procedure which has no internal routines such as $C$, $ST$ is defined as

$$ST(\text{PROCEDURE } C(); \ldots; \text{END } C, 2) = (\text{PROCEDURE } C(); \ldots; \text{END } C)[W.2C/W].$$
(PROGRAM A;
    DECLARE VAR X OF NUMBER ENDDECLARE;
    SA;
    POST(Q);
    END;

(PROCEDURE B(VAR Y OF NUMBER);
    IN: Y;
    OUT: Y;
    DECLARE VAR Z OF NUMBER ENDDECLARE;
    SB;
    END;

(PROCEDURE C();
    DECLARE VAR W OF NUMBER ENDDECLARE;
    SC;
    END;

END C;)

END B;)

END A;

Figure 3.8. ST applied to Figure 3.7
The result of applying the renaming function to program A of Figure 3.7 is shown in Figure 3.8. Figure 3.8 shows program A after ST has been applied, but before any textual substitution has taken place. Parentheses are used in this figure to enclose the formal parameters in a procedure declaration and to denote the range of the textual substitution operator. Observe from Figure 3.8 that any reference to the variable X (or the procedure B) from procedure C will resolve to the variable X.OA (or the procedure B.OA) and also that declaring a new variable X in the procedure B would cause a reference to X in C to resolve to the binding for X in B. The weakest precondition for the renamed program A can now be found.

\[ WP(\text{PROGRAM } A, Q) \]
\[ = WP(\text{ST(PROGRAM } A, 0), Q') \]
where \( Q' \) is the predicate Q after applying the function ST. The weakest precondition of this new program is found using the rules presented for external procedures. Again, the transformer for a program is

\[ WP(\text{PROGRAM } A, Q) \]
\[ = WP(\text{DCL } X; \text{ SA};, Q) \]
given \( EB \).

\( EB \) refers to the predicate transformer for the procedure B. The predicate transformer for a procedure is found in the same manner as described in the section on external procedures.
The weakest precondition for invocation of a procedure whose predicate transformer has been found is now reviewed. Suppose the procedure P is declared as follows.

PROCEDURE P(VAR X,Y,Z OF NUMBER);
    IN: X,Z;
    OUT: Y,Z;
    DECLARE VAR I OF NUMBER ENDDO;
    SP;
END;
END P

The weakest precondition of a call to P with actual parameters A, B, and C and postcondition Q is

\[ WP(\text{CALL P}(A,B,C),Q) = RP[A/X,B/Y,C/Z,Q/R] \]

This rule is the external call rule, EXT_CALL', presented earlier.

To demonstrate the rules for static binding, they are now used to prove the program in Figure 3.4. To find,

\[ WP(\text{PROGRAM L, T=25 AND A=56}) \]

we first find

\[ ST(\text{PROGRAM L, 0}) \]

This has been done and is given in Figure 3.9. Weakest preconditions are now applied to prove this program.
PROGRAM L;

DECLARE VAR T.OL, A.OL OF NUMBER ENDDECLARE;
A.OL := 1;
CALL R.OL(A.OL);
CALL Q.OL(A.OL);
POST(T.OL=25 AND A.OL=56);
END;

PROCEDURE R.OL(VAR B.1R OF NUMBER);
  IN: B.1R;
  OUT: B.1R;
    T.OL := B.1R * 5;
    B.1R := T.OL;
END;
END R.OL;

PROCEDURE Q.OL(VAR X.1Q OF NUMBER);
  IN: X.1Q;
  OUT: X.1Q;
    DECLARE VAR T.1Q OF NUMBER ENDDECLARE;
    CALL R.OL(X.1Q);
    T.1Q := X.1Q + 5;
    X.1Q := T.1Q + X.1Q + 1;
END;
END Q.OL;

END L

Figure 3.9. The program of Figure 3.4 after ST
First, the predicate transformers for the procedures \( R \) and \( Q \) are found.

\[
\begin{align*}
\mathcal{E}R.0L &= WP(\text{PROCEDURE } R.0L, R) \\
 &= WP(T.0L:=B.1R*5; B.1R:=T.0L, R) \\
 &= R(T.0L/B.1R) (B.1R*5/T.0L)
\end{align*}
\]

The predicate transformer for \( Q \) is:

\[
\begin{align*}
\mathcal{E}Q.0L &= WP(\text{PROCEDURE } Q.0L, R) \\
 &= WP(DCL T.1Q; \text{CALL } R.0L(X.1Q); T.1Q:=X.1Q+5; \\
&\quad X.1Q:=T.1Q+X.1Q+1, R) \\
 &= WP(DCL T.1Q; \text{CALL } R.0L(X.1Q), R(T.1Q+X.1Q+1/X.1Q) \\
&\quad (X.1Q+5/T.1Q)) \\
 &= WP(DCL T.1Q, \mathcal{E}R.0L[X.1Q/B.1R, (R(T.1Q+X.1Q+1/X.1Q) \\
&\quad (X.1Q+5/T.1Q))/R]) \text{ BY } \text{EXT}_\text{CALL}'.
\end{align*}
\]

Substituting in the predicate transformer for \( R \) and performing the textual substitution we have

\[
\begin{align*}
&WP(DCL T.1Q, R(T.1Q+X.1Q+1/X.1Q) (X.1Q+5/T.1Q) (T.0L/X.1Q) \\
&\quad (X.1Q*5/T.0L)) \\
&= R(T.1Q+X.1Q+1/X.1Q) (X.1Q+5/T.1Q) (T.0L/X.1Q) \\
&\quad (X.1Q*5/T.0L) (Un/T.1Q).
\end{align*}
\]

Having the predicate transformers for \( Q \) and \( R \), the weakest precondition for the main routine can be found.

\[
WP(DCL T.0L,A.0L; A.0L:=1; \text{CALL } R.0L(A.0L); \text{CALL } Q.0L(A.0L), \\
T.0L=25 \text{ AND } A.0L=56)
\]
\begin{equation}
= \text{WP}(\text{DCL } T.0L, A.0L; A.0L := 1; \text{ CALL } R.0L(A.0L), \text{ CALL } Q.0L(A.0L), T.0L = 25 \text{ AND } A.0L = 56))
\end{equation}

Evaluating the postcondition

\begin{align*}
&= \text{WP}(\text{CALL } Q.0L(A.0L), T.0L = 25 \text{ AND } A.0L = 56) \\
&= \text{EQ.0L}[A.0L/X.1Q, (T.0L = 25 \text{ AND } A.0L = 56)/R] \\
&\quad \quad \text{BY EXT_CALL}^*.
\end{align*}

\begin{align*}
&= T.0L = 25 \text{ AND } A.0L = 56 (T.1Q + A.0L + 1/A.0L) (A.0L + 5/T.1Q) \\
&\quad \quad \quad \quad \quad (T.0L/A.0L) (A.0L*5/T.0L) (T.0L/T.1Q) \text{ BY [ ]}.
\end{align*}

\begin{align*}
&= T.0L*5 = 25 \text{ AND } A.0L*5 + 5 + A.0L*5 + 1 = 56 \\
&= A.0L = 5 \text{ AND } 10A.0L = 50
\end{align*}

Substituting (2) into (1):

\begin{align*}
&= \text{WP}(\text{DCL } T.0L, A.0L; A.0L := 1; \text{ CALL } R.0L(A.0L), \text{ CALL } Q.0L(A.0L), \\
&\quad T.0L = 25 \text{ AND } A.0L = 56) \\
&= \text{WP}(\text{DCL } T.0L, A.0L; A.0L := 1; \text{ CALL } R.0L(A.0L), \\
&\quad A.0L = 5 \text{ AND } 10A.0L = 50) \\
&= \text{WP}(\text{DCL } T.0L, A.0L; A.0L := 1, \text{ WP}(\text{CALL } R.0L(A.0L), \\
&\quad A.0L = 5 \text{ AND } 10A.0L = 50))
\end{align*}

Evaluating the postcondition of (3):

\begin{align*}
&= \text{WP}(\text{CALL } R.0L(A.0L), A.0L = 5 \text{ AND } 10A.0L = 50) \\
&= \text{EQ.0L}[A.0L/B.0L, (A.0L = 5 \text{ AND } 10A.0L = 50)/R] \text{ BY EXT_CALL}^*.
\end{align*}

\begin{align*}
&= A.0L = 5 \text{ AND } 10A.0L = 50 (T.0L/A.0L) (A.0L*5/T.0L) \text{ BY [ ]}.
\end{align*}

\begin{align*}
&= A.0L*5 = 5 \text{ AND } 10*A.0L*5 = 50 \\
&= A.0L = 1
\end{align*}

Substituting (4) into (3):

\begin{align*}
&= \text{WP}(\text{DCL } T.0L, A.0L; A.0L := 1; \text{ CALL } R.0L(A.0L); \text{ CALL } Q.0L(A.0L), \\
&\quad A.0L = 5 \text{ AND } 10A.0L = 50)
\end{align*}
\[ T.0L=25 \text{ AND } A.0L=56 \]
\[ = \text{WP}(T.0L, A.0L, \text{WP}(A.0L:=1, A.0L=1)) \]
\[ = \text{WP}(T.0L, A.0L, \text{TRUE}) \]
\[ = \text{TRUE}. \]

This completes the proof of Figure 3.9.

C. The Abstraction Theorem

To this point, the treatment of procedures with the static binding strategy has been limited to a special class of routines. The limiting factor has been the ability to detail a predicate transformer for the procedure. Such a transformer can only be created when the procedure is sufficiently short and contains no loops. Certainly, the concept of a transformer provides an operational definition of procedure facilities; however when a general class of procedures is considered, the predicate transformer becomes difficult to detail. The proof mechanisms defined in this section are used when a predicate transformer cannot be found.

In programming, it is quite common to group several statements together and to treat them as one entity. Procedures are an example of this practice. A group of statements which perform a single logical function are referred to by a single name and may be used several times in the program. Usually, a procedure's meaning is informally inferred by its name or by comments which describe its
operation. The name of a procedure implies the effect of executing its body. The name ADD_SYMBOL, for example, abstracts the meaning of the procedure which adds a new element to the symbol table. This informal way of attaching a meaning to a group of statements is not rigorous enough for verification.

One form of procedural abstraction has already been used to present the proof mechanisms for both external and internal procedures which use static binding. A predicate transformer was found in these cases to describe the effect of executing the procedure. This transformer served to abstract the meaning of the procedure in a form applicable to each invocation. This method can be applied to the generalized case in which a single statement or group of statements is to be considered as a single entity, say SL. A predicate transformer may be found to describe the effect of executing the statements SL. This transformer

\[ T = \text{WP}(SL, P), \]

where \( P \) is any predicate, can be applied to verify a specific use of SL with desired result \( R \). The verification is performed by substituting the postcondition \( R \) for all occurrences of \( P \) in the transformer \( T \).

Unfortunately when SL contains a construct which must be proven by induction, the predicate transformer for SL can no longer be detailed. Another form of procedural abstraction
is now presented to allow proofs of statements for which no predicate transformer may be found.

To detail the operational meaning of the statements SL, the statement types PRE and POST are added to the language. The statement POST(P), where P is a predicate, has been used to describe the desired result of executing a program. This statement may also appear at the end of a procedure to specify the desired result of procedure execution. The statement PRE(P), where P is an assertion, will be used to specify the expected state of the variables prior to execution of a procedure. The statements PRE and POST can be used in the general case to describe the effect of executing the statements SL as follows:

\[ \text{PRE}(P); \]
\[ \text{SL}; \]
\[ \text{POST}(Q); \]

Adequate choices for the predicates P and Q must be made by the programmer. P and Q must be adequate in two senses. First, it must be shown that P really is a valid precondition for execution of SL with result Q, i.e. P => MP(SL,Q). Secondly, P and Q must satisfy a "reasonable" property. This "reasonable" property requires that together P and Q describe the operation of statements SL. An example of predicates P and Q which aren't reasonable is:
\texttt{PRE(FALSE);}
\texttt{SL;}
\texttt{POST(FALSE);}

Clearly, \texttt{FALSE} $\Rightarrow$ \texttt{WP(SL, FALSE)} regardless of the statements \texttt{SL}; however, the precondition \texttt{FALSE} and the postcondition \texttt{FALSE} give no information concerning the operation of \texttt{SL}. If \texttt{P} and \texttt{Q} conform to these two criteria, then they may be used to aid in proving each instance of \texttt{SL} in the program.

If \texttt{SL} is used in many places in a program, then to verify the program, the weakest precondition \texttt{WP(SL,R)} must be found for different predicates \texttt{R}. The Abstraction Theorem presented below defines the use of \texttt{SL} in terms of the predicates \texttt{P}, \texttt{Q}, and \texttt{R}. The theorem allows a precondition to be found without calculating the weakest precondition. Assume that \texttt{P} and \texttt{Q} are reasonable with respect to \texttt{SL}. The Abstraction Theorem is then

\texttt{If P $\Rightarrow$ WP(SL, Q)}

\texttt{then (P AND (Q $\Rightarrow$ R)) $\Rightarrow$ WP(SL, R).}

\texttt{PROOF.}

This theorem follows immediately from the property of monotonicity. Suppose that

\texttt{P and (Q $\Rightarrow$ R).}

By monotonicity it follows that

\texttt{WP(SL, Q) $\Rightarrow$ WP(SL, R).}

The antecedent of the theorem says
P => WP(SL, Q).

Additionally, we have assumed P. Thus,

WP(SL, R)

can be established.

1. Abstraction: verifying procedures

The Abstraction Theorem is now applied to allow verification of external procedures and internal procedures with static binding. The procedure proof mechanisms presented below are the tools most commonly used in verification since they are not limited to a specific class of procedures. The Abstraction Theorem is first applied to external procedures, and then extended to internally defined procedures.

Consider the procedure L declared below. To prove the use of procedure L in a program, the programmer must first construct adequate PRE and POST assertions which describe the operation of L. Suppose that the precondition P and the postcondition Q have been placed in the text of L to describe L's effect on its parameters.
PROCEDURE L(VAR X,Y,Z OF TYPE);

IN: X, Z;

OUT: Y, Z;

DECLARE VAR I OF TYPE ENDDECLARE;

PRE (P);

SL;

POST (Q);

END;

END L;

The precondition P is a predicate which describes the initial values of the inputs to L. The procedure L may assume that a special relationship exists among its inputs or that special values have been assigned to them. These constraints may be necessary to ensure proper execution of L and must hold prior to each call. P is an assertion whose free variables may include only X and Za, the inputs to L. Za is used to denote the value of Z upon entry to L.

The postcondition Q is a predicate which describes the values of the variables set by L in terms of L's input. Q may include as its free variables Y and Z, the outputs of L, and both of L's inputs, X and Za. It is important to note that neither P nor Q may reference any variables declared in L.
For each procedure declared in a program, it must be shown that its precondition and postcondition describe the operation of the procedure. This is done for L (and in general) by showing that

\[ P \Rightarrow WP(SL, Q). \]

The proof of this implication dismisses the antecedent of the Abstraction Theorem and allows the consequent of the theorem to be used for each call to the procedure. In the process of proving a program which uses a procedure, say L above, it is necessary to calculate the weakest precondition for calls to L. When the Abstraction Theorem is used, the weakest precondition for the call to L is not calculated, but rather an assertion which implies the weakest precondition. For example for the call

\[ \text{CALL L(A,B,C)}, \]

the weakest precondition

\[ WP(\text{CALL L(A,B,C)}, B) \]

might be needed to prove the program containing the call. Applying the Abstraction Theorem, the following implication holds.

\[ (P(A/X,C/Z) \land (Q(A/X,B'/Y,C/Zn,C''/Z) \Rightarrow R(B'/B,C''/C))) \Rightarrow WP(\text{CALL L(A,B,C)}, B). \]

The antecedent of this implication is a precondition for the call to L. When this implication is used in the proof of a program, it will be denoted by EXT_CALL_ABS. In the
antecedent of the implication, a quoted identifier (") refers to the final value for that identifier and $Z_0$ denotes the initial value for $Z$. The similarity between this rule and the Abstraction Theorem is reflected in its general form. The substitutions are necessary to handle the parameters and to distinguish initial values from final values.

The proof of the program in Figure 3.10 is now given to demonstrate the use of abstraction to verify external procedures. To apply the rule EXT_CALL_ABS we must first show:

\[(L > 0 \text{ AND } N_n > 0) \Rightarrow WP(X := N + L; N := X + N; M := X, M = N_0 + L \text{ AND } N = 2N_0 + L)\]

To show this the consequent of the implication is simplified to:

\[(N + L = N_0 + L \text{ AND } N + N + L = 2N_0 + L)\]

Clearly \[(L > 0 \text{ AND } N_n > 0) \Rightarrow ((N + L = N_0 + L) \text{ AND } (2N + L = 2N_0 + L)).\]

We can now calculate a precondition for the main block of the program.

\[WP(A := 5; C := 10; CALL P(A, B, C); X := B + C, X = 40) = WP(A := 5; C := 10, WP(CALL P(A, B, C), B + C = 40))\]

We get

\[((A > 0 \text{ AND } C > 0) \text{ AND } ((B^n = C + A \text{ AND } C^n = 2C + A) \Rightarrow B^n + C^n = 40))\]

\[\Rightarrow WP(CALL P(A, B, C), B + C = 40) \text{ BY EXT_CALL_ABS.}\]

The antecedent of this implication simplifies

\[((A > 0 \text{ AND } C > 0) \text{ AND } ((B^n = C + A \text{ AND } C^n = 2C + A) \Rightarrow B^n + C^n = 40))\]
PROGRAM TEST;

DECLARE VAR X, A, B, C OF NUMBER ENDDECLARE;

A := 5;
C := 10;
CALL P(A, B, C);
X := B + C;
POST(X = 40);
END;

END TEST

PROCEDURE P(VAR L, M, N OF NUMBER);

IN: L, N;

OUT: M, N;

DECLARE VAR X OF NUMBER ENDDECLARE;

PRE(L>0 AND N>0);
X := N + L;
N := X + N;
M := X;
POST (M=N+L AND N=2N+L);
END;

END P

Figure 3.10. External procedure for proof with the theorem
\[ ((A > 0 \text{ AND } C > 0) \text{ AND } 3C + 2A = 40) \]

So

\[ WP(A := 5; C := 10, A > 0 \text{ AND } C > 0 \text{ AND } 3C + 2A = 40) \]

\[ \Rightarrow WP(A := 5; C := 10, WP(CALL P(A, B, C), B + C = 40)) \text{ BY Monotonicity.} \]

The antecedent of this implication also simplifies

\[ WP(A := 5; C := 10, A > 0 \text{ AND } C > 0 \text{ AND } 3C + 2A = 40) \]

\[ = 5 > 0 \text{ AND } 10 > 0 \text{ AND } 30 + 10 = 40 \text{ BY := and ;} \]

This leaves the result

\[ TRUE \Rightarrow WP(A := 5; C := 10; CALL P(A, B, C); X := B + C, X = 40) \]

which completes the proof.

The use of the Abstraction Theorem to prove procedures has a serious drawback with respect to current high level languages. Most languages accept a procedure call in which one variable is used for two actual parameters. For example, suppose that the call

\[ CALL P(A, C, C) \]

is used to replace the call to \( P \) from the main program in Figure 3.10. This call enables the procedure \( P \) to update the storage for \( C \) through two separate names \( M \) and \( N \). If this call is verified using the proof rule \text{EXT\_CALL\_ABS}, the results may not be valid. The substitution of the post assertion \( Q \) for this call

\[ Q(A/L, C'/M, C/N, C'/N) \]

simplifies to:

\[ M = N + L \text{ AND } N = 2N + L (A/L, C'/M, C/N, C'/N) \]
This postcondition does not reflect that the assignment

\[ M := X; \]

is the last to affect the variable \( C \) (represented by \( C'' \)), but instead indicates that \( P \) returns with two new values for \( C \).

This situation has been appropriately referred to as aliasing by Hoare and others. Aliasing occurs when two identifiers in a procedure can access the same storage element. The programming language EUCLID has taken great care to assure that aliases do not exist in programs. Unfortunately, this requires compile and runtime checking of actual parameters.

The Abstraction Theorem and the CALL rule, \( \text{EXT\_CALL\_ABS} \), closely resemble the form of the rules London and Hoare (40, 23) present for proving procedures. The aliasing problem is introduced to proofs by basing procedure proof mechanisms on the Abstraction Theorem. As demonstrated in the example of aliasing given above, the problem centers on the inability to specify an "order of evaluation" in a procedure's POST assertion. This assertion can only specify the final values of identifiers and does not indicate the sequence in which they receive those values.

Currently, KL-1 does not have the facilities to detect aliases, but the KL-1 verification system will indicate to
the programmer which procedure calls might cause an alias to occur.

The changes necessary to extend the rule EXT_CALL_ABS to allow proofs of internally defined procedures is now detailed. Suppose that the procedure L declared previously is internal to another subroutine or the main program. This supposition changes SL, P, and Q as follows:

SL, the set of statements of L, may now reference nonlocal identifiers.

P, the precondition for L, may now include (in addition to the formal inputs X and Zn) any nonlocal variable whose value is used as an input to L.

Q, the postcondition for L, may reference all variables which provide either input to L or are updated by L. This includes all nonlocal variables accessed in L as well as the initial values X and Zn and the outputs Y and Z.

To prove a program which uses internal procedures, the static renaming function ST is applied to the program prior to verification. The use of the renaming function allows the rule EXT_CALL_ABS to prove internally defined procedures whose nonlocal referencing environment is not limited. That is, a procedure may access nonlocal identifiers as determined by the conventional static scoping rules.
2. Abstraction: verifying functions

Functions are procedures which return a single value to the calling program. Functions do not add expressive power to a language which already supports procedures, but is added to languages as a convenience measure only. A function may be invoked from an expression by referencing the function name and supplying actual parameters. As expected, verifying functions closely resembles the verification process for procedures. The general form for a KL-1 function declaration is:

FUNCTION F(VAR X OF T) OF T;

    IN: X;
    PRE(P);
    SF;
    POST(Q);
END;
END F;

The function F is declared to have one parameter X which is an input. Several formal parameters may exist for a given function; however, all parameters must be inputs. F may be invoked in any expression over operations on type T values since F returns a value of type T. It is implicit that SF,
the statement body of $F$, contains one or more statements of the form $\text{return (EXP)}$. When executed, this statement causes control to be returned to the calling program with the expression $\text{EXP}$ designating the value of the function.

The precondition $P$ is a predicate which describes the expected initial values of the parameters and contains as free variables only the inputs to the function. The postcondition $Q$ is a predicate which describes the value returned by the function and contains as free variables all inputs to the function and $F(X)$ which denotes the returned value. These constraints on the predicates $P$ and $Q$ greatly restrict the power of KL-1 functions as compared to those of most high level block structured languages. KL-1 functions return one value; however, most other languages allow functions to access nonlocal variables, possibly changing their value. When a function alters a nonlocal variable either directly or through another procedure, it is said to have side effects. As recognized by the designers of the language EUCLID, verifying a function which causes side effects can become a difficult task. To avoid problems with order of evaluation in either expressions of the form $F(A)*F(B)$ or in compound statements of the form

\begin{verbatim}
INCASE  $F(A)>9$  DO  $A := F(A) + Y$  ENDDO

$F(B+3)>9$  DO  $A := B+7$  ENDDO

ENDINCASE,
\end{verbatim}
all functions in KL-1 are required to produce only one result. In part, this requirement is enforced by the free variables allowed in P and Q and by eliminating the possibility of nonlocal references. But, nothing restricts a KL-1 function from calling a procedure which has side effects. The proof mechanisms given below will not work properly if a function produces a side effect; however, KL-1 does not incorporate any facilities to aid in detecting side effects.

**EUCLID** eliminates the possibility of side effects by the use of an **IMPORT** list. All nonlocal variables which are accessed either directly or indirectly by a procedure must appear in the IMPORT list. Each variable in the list is declared constant or varying depending on whether it is used in right or left context in the procedure. In **EUCLID**, function type procedures may not have varying parameters or import varying identifiers. These conventions along with the restriction on the nonlocal referencing environment make it impossible for **EUCLID** functions to produce side effects.

Mechanisms are now presented for verifying function use in a **KL-1** program. Although the grammar for **KL-1** (Appendix A) allows a function to be invoked anywhere that an expression is used, the tools given here treat function use in assignment statements, as arguments for type IN procedure
parameters, and in guards of alternative statements.

First, the verification of an assignment statement whose expression invokes a function is considered. The rule used is based on the general form for a KL-1 function, $F$, as given previously and the assignment statement

$$\text{VAR} := \text{EXP}$$

where $\text{EXP}$ contains the function call $F(A)$. The rule is an application of the Abstraction Theorem, so the antecedent of that theorem must first be dismissed. This is done by showing

$$P \implies \text{WP}(\text{SF}, \text{Q})$$

To prove this implication, the following transformer for the RETURN statement in the function $F$ can be used.

$$\text{WP}(\text{RETURN}(\text{EXP}), \text{R})$$

$$= ((P(X)=\text{EXP}) \implies \text{R})$$

where $\text{R}$ is any predicate. Note that the transformer for RETURN assumes that the notation $F(X)$ is used in the postcondition to express the function's value. Once the antecedent of the theorem has been shown, the rule FCN_CALL may be used to verify many invocations of $F$. FCN_CALL, which may be used to find a precondition for execution of the statement $\text{VAR}:=\text{EXP}$ (where $\text{EXP}$ contains $F(A)$) with desired result $\text{R}$, is

$$(P(A/X) \land (Q(A/X) \implies R(\text{EXP}/\text{VAR})))$$

$$\implies \text{WP}(\text{VAR}:=\text{EXP}, \text{R}).$$
As procedures, the renaming function ST must first be applied to a program which is proven using FCN_CALL.

Some concrete examples using this rule to verify assignment statements are now considered. Suppose that the successor function SUCC is defined as shown below.

FUNCTION SUCC(VAR X OF NUMBER) OF NUMBER;
    IN: X;
    PRE(TRUE);
    RETURN(X+1);
    POST(SUCC(X) = X+1);
    END;
END SUCC

As an arbitrary example, suppose that we wish to find a precondition for the assignment

\[ A := SUCC(A) \]

with the desired result \( A > 5 \). The rule FCN_CALL applied to this call gives the following implication.

\[ (TRUE(A/X) \text{ AND } (SUCC(X)=X+1(A/X) \Rightarrow A>5(SUCC(A)/A))) \Rightarrow WP(A:=SUCC(A), A>5). \]

The antecedent of this implication can be simplified.

\[ = (SUCC(A)=A+1 \Rightarrow SUCC(A)>5) \]
\[ = A>4 \]

Thus, \( A > 4 \) is a precondition for execution of \( A := SUCC(A) \) with
desired result $A > 5$. We have

$$ A > 4 \Rightarrow WP(A := \text{SUCC}(A), A > 5). $$

As an example of verifying an assignment whose expression contains more than one function call, consider the adaptation of FCN_CALL to find a precondition for the statement

$$ A := \text{SUCC}(A) + \text{SUCC}(B) $$

with the desired result $A > 5$. Applying FCN_CALL to this assignment results in the predicate

$$(\text{TRUE}(A/X) \text{ AND } \text{TRUE}(B/X) \text{ AND } ((\text{SUCC}(X) = X + 1(A/X) \text{ AND } \text{SUCC}(X) = X + 1(B/X)) \Rightarrow A > 5(\text{SUCC}(A) + \text{SUCC}(B)/A)))$$

implying the weakest precondition

$$ WP(A := \text{SUCC}(A) + \text{SUCC}(B), A > 5). $$

The antecedent of this implication simplifies as follows.

$$ = ((\text{SUCC}(A) = A + 1 \text{ AND } \text{SUCC}(B) = B + 1) \Rightarrow (\text{SUCC}(A) + \text{SUCC}(B) > 5)$$

$$ = A + B > 3 $$

So, $A + B > 3$ is a precondition for execution of

$$ A := \text{SUCC}(A) + \text{SUCC}(B) $$

with desired result $A > 5$.

FCN_CALL can also be applied to verify function invocation in an actual parameter. Suppose we desire a precondition for the assignment

$$ A := \text{SUCC}(\text{SUCC}(A)) $$

with desired result $A > 5$. Applying FCN_CALL to this assignment results in the predicate

$$(\text{TRUE}(\text{SUCC}(A)/X) \text{ AND } \text{TRUE}(A/X) \text{ AND }$$
((SUCC(X) = X+1(SUCC(A)/X) AND SUCC(X) = X+1(A/X)) =>
SUCC(SUCC(A)) > 5))

implying the weakest precondition

WP(A := SUCC(SUCC(A)), A > 5).

The antecedent simplifies as follows.

= (SUCC(SUCC(A)) = SUCC(A)+1 AND SUCC(A) = A+1) => SUCC(SUCC(A)) > 5
= A > 3

Thus, A > 3 is a precondition for the call.

Suppose that the function F as declared above is invoked from a guard in an alternative statement. The rule FCN_CALL given above can be used to detail a precondition for this statement. To detail this precondition, the rule must be changed to reflect the fact that an alternative statement is used rather than an assignment. This is done by replacing the transformer for the assignment B(EXP/VAP) in FCN_CALL by the transformer for the proper alternative. For example, suppose that the boolean expression B in the statement

IF B THEN SL ENDIF

contains the function invocation F(A). Then the form of FCN_CALL used to find a precondition for this statement with desired result R would be

(P(A/X) AND (Q(A/X) =>
((R AND ¬B) OR (B AND WP(SL, R))))
=> WP(IF B THEN SL ENDIF, R).
This concludes the treatment of procedures. The reader is referred to Appendix C for a further example which applies rules presented in this chapter to prove a program which uses procedures.
IV. COMPOSITE DATA TYPES

Arrays provide a means of defining composite organizations of data and exist in nearly every high level language. The array is a set of data (homogeneous data in KL-1) which can generally be accessed in two ways. The first is by a single name which refers to the entire collection of data. The second is by indexing to obtain access individually to each datum in the set or to subsets of the set. Language features to support the definition and access of arrays in both of these modes have developed from a need to manipulate groups of data whose elements share properties and require reference by a unique name.

An array is often defined as a sequence of memory locations which are accessible by index on the array name. Although generally accurate, this definition depends on a sequential implementation of arrays and does not easily accommodate a structure like the SNOBOL Table (which should formally be regarded as an array). The Table structure allows the SNOBOL programmer to define and access array elements using indices from either an ordered or unordered type. With this approach, a two element array may have as its subscripts the strings 'APPLE' and 'PEAR'.

Another common definition treats an array as a special case of a linear list. With this definition, an array is viewed as an ordered sequence of elements. While this
definition is not implementation dependent, it is not general enough to apply to all arrays. Certainly, there are nonempty arrays which do not have a first element; however, there are no such linear lists.

A more precise definition, which is receiving growing acceptance, is to view an array as a set of ordered tuples (INDEX, VALUE). This definition allows relief from the requirement that indices be successive elements of an ordered type. Two special operations, STORE and RETRIEVE, are defined on arrays to provide access to the individual elements of the set.

STORE(NAME, INDEX, VALUE). The store operation creates a pair (INDEX, VALUE) in the array NAME. If a pair already exists with a first component identical to INDEX, it is removed from the set. This last clause restricts the set so that no two elements of the array have the same first component. Because of this restriction, an array may be viewed as a function from the index type into the value type. Without this restriction, the array simply defines a relation.

The retrieve operation provides access to the value component of individual elements of the array. RETRIEVE(NAME,INDEX) looks for a pair (INDEX,VALUE) in the array NAME and if found, it returns the value portion of that tuple. All languages which support arrays furnish these
operations. But rather than denoting them with a function call, they are represented by subscripting the array name. Array subscripting is denoted by brackets as $A[j]$. The store operation is signified by the appearance of a subscripted array name in left context and the retrieve operation is signified by the appearance of a subscripted array name in right context. Thus, the assignment

$$A[I] := J$$

signifies the operation $\text{STORE}(A,I,J)$. The assignment

$$J := A[I]$$

denotes the statement

$$J := \text{RETRIEVE}(A,I).$$

Defining an array as a set of pairs, together with the RETRIEVE and STORE operations, is not implementation dependent. Additionally, this definition fits all structures which should be treated as arrays. It accommodates either the typical sequential representation for arrays or a linked representation.

The problem of verifying programs which manipulate arrays has been treated by both Hoare and London in the context of the Axiomatic approach to verification (24, 40). Their research provides the basis for the verification of arrays using weakest preconditions. The approach is similar in that arrays are treated as sets, but the weakest precondition treatment, developed below, relaxes the
requirement that indices for arrays be of either scalar or subrange types. No ordering need exist on the index type; it may be any elementary type.

The mechanisms necessary to verify singly dimensioned arrays are first presented, then these mechanisms are expanded to include multiple dimensions.

In KL-1, the form of an array declaration is

```
DECLARE ARRAY A[L:U] OF TYPE ENDDECLARE;
```

where both L and U are numeric. The current version of KL-1 does not support nonnumeric types for array subscripts; however, the mechanisms presented here do allow subscripts of simple nonnumeric types. The array A in the above declaration is defined to be the set of ordered pairs

\[ A = \{(L,V_1) (L+1,V_2) \ldots (U,V_n)\}. \]

Let \((A,i:exp)\) denote the array A and indicate that A has been changed to contain the pair \((i,exp)\). This notation will often be nested and might take the form

\[ ((A,i:exp),j:exp'). \]

This represents the array \((A,i:exp)\) with the value of component \(j\) specified to be \(exp'\). Nesting, as in the above example, specifies the order in which components are assigned values. This order becomes important when the nesting redefines the value of a subscript. In fact, if a nested array specification defines more than one value for a single index, then the outermost prevails. This is illustrated in
the following example

\((A,1:exp,1:exp').\)

By the definition above, this is the array \((A,1:exp)\) where the value of component 1 is specified to be \(exp'\). It is equivalent to

\((A,1:exp').\)

The notation \((A,i:exp)\) provides a means of specifying the array \(A\) and will commonly be subscripted in assertions (not in programs). Thus, the notation

\((A,i:exp)[j]\)

denotes the value of component \(j\) of the array \((A,i:exp)\).

Clearly,

\((A,i:exp)[i] = exp.\)

The weakest precondition predicate transformer for a store operation, an array subscript in left context, is defined to be

\(WP(A[I]:=EXP, R) = R((A,I:EXP)/A).\)

The occurrence of a retrieve operation, the right context use of an array subscript, does not alter the calculation of the weakest precondition. For example,


To demonstrate these mechanisms, consider the proof of the program section:
PRE(TRUE);
X[1] := 5;
X[X[1]] := Y;
POST(X[6] = Y);

The steps in the verification of this program section are:

= WP(X[1]:=5; X[1]:=X[1]+1, (X,X[1]:Y)[6]=Y) 

Since (X,1:X[1]+1)[1] = X[1]+1, we have


Since (X,1:5)[1]=5, we have

 (((X,1:5),1:5+1),5+1:Y)[6]=Y 
= Y=Y 
= TRUE

A problem may arise when a subscripted array occurs as an actual parameter in a procedure call and results from the weakest precondition treatment of both procedures and arrays. For example, consider the procedure SWAP of Figure 4.1. SWAP is used to exchange the values of two program variables. The proofs of certain calls to SWAP will make the parameter call mechanism erroneously appear to be "call by name"
(described below) if subscripted actual parameters are not treated with a special convention. To motivate this convention, the proof of a call to SWAP with simple variables is first discussed.

PROCEDURE SWAP (VAR X,Y OF NUMBER)
IN: X,Y;
OUT: X,Y;
DECLARE VAR T OF NUMBER ENDDECLARE;
T := X;
X := Y;
Y := T;
END;
END SWAP

Figure 4.1. A procedure to exchange two values

To verify that SWAP works correctly for the call,

CALL SWAP(B,C)

a predicate transformer for SWAP is required. This transformer is

R(T/Y) (Y/X) (X/T) (Un/T).

To describe the postcondition for a call to SWAP, a notation for a variable's initial value is used. Let Bn and Cn denote the values of B and C when SWAP is called, then the desired result of the above call is
(B=C^\alpha \text{ AND } C=B^\eta).

When the predicate transformer for \texttt{SWAP} is applied to the actual parameters \( B \) and \( C \) and the actual postcondition \((B=C^\alpha \text{ AND } C=B^\eta)\), the weakest precondition

\[
\text{WP} (\text{CALL SWAP}(B,C), B=C^\alpha \text{ AND } C=B^\eta)
\]

is found to be

\[
C=C^\alpha \text{ AND } B=B^\eta.
\]

This precondition indicates \( C \) has its initial value and \( B \) has its initial value as expected.

The problem mentioned above occurs for the call

\[
\text{CALL SWAP}(I,A[I]).
\]

If the postcondition for this call is not designed carefully, the proof techniques can make the actual parameter \( A[I] \) appear to use the call by name parameter mechanism. If \( \text{SWAP}(I,A[I]) \) is called using the call by name rule, then the \( I \) used to evaluate \( A[I] \) in the statement

\[
Y := T
\]

is the updated value of \( I \). That is, the value of \( I \) set in the statement

\[
X := Y.
\]

\( KL-1 \) does not use this call mechanism; instead, \( A[I] \) is evaluated with the call-time value of \( I \). In general, array elements can be passed to procedures without the call by name parameter mechanism. To do this, the identifiers in the subscript must be marked to denote that the call-time value
is used. The marked subscript is used for the substitution of actual parameters into the procedure's predicate transformer.

For the call, CALL SWAP(I,A[I]) with the KL-1 call mechanism, the actual parameter A[Ia] is used by the predicate transformer. This is the special convention for subscripted actual parameters mentioned above.

The postcondition for this call to SWAP is,

\[ A[Ia] = Ia \text{ AND } I = A[Ia]a. \]

In this assertion, A[Ia]a denotes the call-time value of A[I]. The weakest precondition,

\[ WP(CALL SWAP(I,A[I]), A[Ia] = Ia \text{ AND } I = A[Ia]a) \]

is found by the following substitution on SWAP's predicate transformer.


Using this substitution, the weakest precondition becomes

\[ I = Ia \text{ AND } A[Ia] = A[Ia]a, \]

as expected.

The extension of this proof technique to multiple dimensions is conceptually straightforward. One way of accommodating multiple dimensions is now presented in the context of the definition given previously.

An array with multiple dimensions is still defined to be a set of ordered tuples; however, the value component of each tuple becomes itself an array. Using this definition the two
dimensional array

DECLARE ARRAY A[1:2,1:3] OF TYPE ENDDECLARE;

is represented by the set

\[
A = \{(1,\{(1,V1)\ (2,V2)\ (3,V3)\}) \ (2,\{(1,V4)\ (2,V5)\ (3,V6)\})\}\).
\]

Each tuple in the set A represents a separate row of the array. The tuple with index 1 has as its value the one dimensional array which is the first row of A. The generalization of this approach to many dimensions adds to the level of nesting. Thus in a three dimensional array, each tuple has as its value component a two dimensional array.

Subscripting with multiple dimensions in KL-1 is denoted by enclosing the indices in brackets and separating them by commas. Using the above declaration for A, the subscript 

\[A[I,J]\]

denotes the value component of the tuple with index J in the subarray A[I]. The subscript A[I,J] actually represents the two subscripting operations A[I][J]. For convenience these operations are denoted by one set of brackets as A[I,J].

The previous notation for specifying values in an array is adapted to multiple dimensions. In the case of two dimensions,

\[(A,I: (A[I],J:EXP))\]

denotes the array A whose subarray A[I] contains the pair (J,EXP). Thus, the value of A[I,J] is EXP.
The weakest precondition for an assignment statement with multiple subscripting in left context follows from the rule for one dimension. Consider the assignment statement,

\[ A[I,J] := \text{EXP}. \]

The weakest precondition for execution of this statement with desired result \( R \) is defined as

\[ \text{WP}(A[I,J]:=\text{EXP}, R) = R((A,I:(A[I],J:\text{EXP}))/A). \]

Right context use of many dimensions is the same as right context use of one dimension. A precondition for the assignment

\[ B := A[I,J] \]

with result \( R \) is defined as

\[ \text{WP}(B:=A[I,J], R) = R(A[I,J]/B). \]
V. SUMMARY

This thesis details the basis upon which a verification system for the language KL-1 may be constructed. The application of the weakest precondition proof method to KL-1 forms this basis. Dijkstra originated this proof technique (8) and applied it to his set of control structures (8, 9). Later, the technique was applied to the control constructs of the language KL-1 by Lindquist and Keller (36, 37); this extension is summarized in Appendix B. The research reported in this thesis concludes the application of weakest preconditions to KL-1. This is done by detailing the proof mechanisms for procedures and composite data types, i.e. KL-1 arrays. A majority of the thesis is devoted to applying the weakest precondition technique to general procedure mechanisms and includes the procedures of KL-1.

The programming language EUCLID has been designed so that its programs can be proven correct. To some extent, this philosophy sets a precedent for future language design since there is a growing need to verify software. Most relevant to this thesis are the language design decisions that allow EUCLID procedures to be proven. In the context of the Axiomatic proof method, Hoare and London (23, 40) have detailed rules for proving procedures which require two language restrictions. These restrictions are documented by Lampson and Popek (34, 45) in terms of the language EUCLID.
The first restriction is placed on the nonlocal referencing environment for a procedure. In EUCLID references to identifiers declared outside a procedure are controlled by the IMPORT list. The rules governing this list limit the nonlocal identifiers to which a procedure has access. These rules reduce nonlocal references to the point that both the dynamic (most recent) and static identifier binding strategies produce the same results.

The procedure rules given by London are correct in the context of this restriction (and a second restriction detailed later). Furthermore, these rules represent a significant contribution by providing a means of verifying programs which use procedures. The research reported here augments London's work by providing techniques for proving more general procedures. The weakest precondition definitions developed in this thesis provide tools for proving procedures without restricting the nonlocal referencing environment. The usefulness of these tools go beyond lifting the restrictions on nonlocal references. If language design is to be based to some extent on verifiability, then the cost of any language restriction in terms of flexibility of expression must be compared to the effect the restriction has on the "complexity of proof". In the case of restricting the nonlocal referencing environment, the proof tools presented in this thesis can be used to
evaluate the "complexity of proof" aspect of this trade-off. Indeed, it would be useful for language design to have a measure of the extent that limiting nonlocal references affects the burden of proof.

This thesis also develops separate procedure rules for the static and dynamic binding strategies. These proof techniques are developed in an environment in which nonlocal references are not restricted by an IMPORT list and in which procedure variables and dynamic storage allocation are prohibited. One justification for applying a proof technique to a language is that the proof mechanisms define the operational semantics of the language statements. The Axiomatic rules define the semantics of a limited set of procedures; those procedures for which the most recent and static binding strategies are the same. The mechanisms presented in this thesis define the operational semantics of procedure use with the dynamic and static binding strategies.

The second restriction EUCLID makes is aliasing. Whenever two separate identifiers in a procedure refer to the same storage element, the proof techniques given by London may not work. To eliminate the possibility of aliases in EUCLID, both compile and runtime checks are necessary.

This research defines a class of procedures for which aliasing is not a problem and presents the weakest precondition proof techniques for procedures in this class.
Those procedures for which a "predicate transformer" can be found constitute this class. Specifically, this class is made up of all the procedures for which we can detail an "expression" which displays the transformation of a postcondition into the weakest precondition. The proof techniques defined for this class are restricted to internally and externally declared procedures which use the static binding strategy.

The Abstraction Theorem is presented to aid in proving those procedures for which we cannot detail a predicate transformer. Intuitively, this theorem shows how appropriately chosen PRE and POST assertions can describe the meaning of a set of statements and then, be applied to verify different uses of those statements in a program. The Abstraction Theorem is applied to procedures to create weakest precondition proof techniques for procedures and functions which use the static strategy. These techniques handle procedures whose nonlocal referencing environment is not restricted, but they introduce the aliasing problem to weakest precondition proofs. The weakest precondition proof techniques derived from the Abstraction Theorem have a form similar to the proof rules outlined by London and it is this form which introduces the aliasing problem to procedure proofs.
This thesis does not address the problem of aliasing in the context of the dynamic binding strategy. Due to the nature of this strategy, the Abstraction Theorem is not applied to procedures which use dynamic binding. Future research is necessary to understand if verification techniques are suitable for languages which employ the dynamic strategy. The constraints under which the Abstraction Theorem may be applied to the dynamic strategy must be studied.

The objective of this thesis was to create a framework upon which a language (KL-1) verification system may be built. The intent is to use the weakest preconditions defined for KL-1 as a basis to implement a system to include three types of software verification; constructive, static and execution time verification. The framework completed, future work will involve the application of the techniques defined in this thesis to the development of the verification system.
VI. BIBLIOGRAPHY


VII. ACKNOWLEDGMENTS

Gratefully, I would like to acknowledge Dr. Roy Keller for his unlimited accessibility and careful guidance throughout all the work reported in this dissertation.

Dr. Kris (Ramachandran Krishnaswamy) has provided superb technical assistance and displayed great patience by reviewing preliminary drafts of this work.

I would also like to thank the remainder of my committee Dr. R.M. Stewart Jr., Dr. A.L. Selman and Dr. E.J. Peake Jr. for their editorial comments.

Finally, I thank my wife whose sacrifices have made it possible to devote the needed energy to this thesis. She is gifted with the ability to motivate me when enthusiasm is needed and to kick me when kicking is needed.
VIII. APPENDIX A: THE KL-1 GRAMMAR

The syntactic structure of the language KL-1 is given in this appendix. A derivation of the Backus Naur Form (BNF) is used with the following metasymbols and conventions.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>::=</td>
<td>Represents the definition of a syntactic category (nonterminal).</td>
</tr>
<tr>
<td></td>
<td>Alternation.</td>
</tr>
<tr>
<td>a ::= b</td>
<td>c is a shorthand for;</td>
</tr>
<tr>
<td>a ::= b</td>
<td></td>
</tr>
<tr>
<td>a ::= c.</td>
<td></td>
</tr>
<tr>
<td>{ }</td>
<td>Repetition zero or more times.</td>
</tr>
<tr>
<td>a ::= {b} is a shorthand for;</td>
<td></td>
</tr>
<tr>
<td>a ::= empty</td>
<td>a b.</td>
</tr>
</tbody>
</table>

By convention, nonterminals are contiguous strings of lower case letters. Terminals are any nonblank symbols which are neither metasymbols nor nonterminals.

The syntax of KL-1, as presented here, is not consistent in one respect with the definition of the weakest precondition function given in Appendix B. The grammar includes one generalized repetitive construct REPEAT-ENDREPEAT. This iterative statement can be used as a single exit loop, multiple exit loop, or for bounded iteration (a For loop). The definition of the weakest precondition function in Appendix B includes for multiple
exit loops the REPEATEXIT statement and for single exit loops the WHILE-DO and REPEAT-UNTIL statements. FOR loops can be accommodated by proving their equivalent REPEAT-UNTIL statement.

program ::= PROGRAM newid condec block segroutine

   END id segroutine

segroutine ::= {routine}

routine ::= proc | function

proc ::= PROCEDURE prochead block segroutine END id

prochead ::= newid ( formparm ) in out

formparm ::= empty | fplist

in ::= empty | IN : inlist ;

inlist ::= id {, id}

out ::= empty | OUT : outlist ;

outlist ::= id {, id}

condec ::= empty | CONSTANT conlist

conlist ::= conid {, conid}

conid ::= newid num | newid string

declist ::= iddec {, iddec}

iddec ::= VAR newidlist OF type

   | ARRAY arraylist OF type

fplist ::= fp {, fp}

fp ::= VAR newidlist OF ftype

   | ARRAY fparlist OF ftype

type ::= NUMBER
arraylist ::= array [, array]
ftype ::= NUMBER
STRING ( EXP )
BOOLEAN
STRING ( * )
BOOLEAN
fparlist ::= fpar [, fpar]
array ::= newid [ dimlist ]
fpar ::= newid [ starlist ]
dimlist ::= dimen [, dimen]
starlist ::= * [, *]
dimen ::= exp TO exp
function ::= FUNCTION funchead block segroutine END id
funchead ::= newid ( formparm ) OF type in
block ::= declare seqstat END
declare ::= empty | DECLARE declist END DECLARE
seqstat ::= stat (; stat)
stat ::= empty
selstat
rpstat
iostat
loopstat
asgnstat
block
procall
| return
| proof

selstat ::= if | incase | forcase

if ::= IF exp THEN seqstat ENDIF

| IF exp THEN seqstat ELSE seqstat ENDIF

incase ::= INCASE seqcase ENDINCASE

seqcase ::= case {case}

case ::= casexp DO seqstat ENDDO

casexp ::= empty | exp

forcase ::= FORCASE seqcase ENDFORCASE

rpstat ::= REPEAT range seqstat ENDRPEAT

range ::= empty | FOR var := exp TO exp inc

inc ::= empty | BY exp

loopstat ::= CONT WHILE exp | QUIT quitcond

quitcond ::= empty | WHEN exp

asgnstat ::= var := exp

var ::= id | id [ explist ]

explist ::= exp {, exp}

actparm ::= empty | explist

iostat ::= READ ( varlist ) | WRITE ( explist )

varlist ::= var {, var}

newidlist ::= newid [, newid]

exp ::= const

| var

| pid ( actparm )
\begin{verbatim}
| ( exp )
| exp + exp
| exp - exp
| exp / exp
| exp * exp
| exp ** exp
| - exp
| + exp
| exp AND exp
| exp OR exp
| NOT ( exp )
| exp = exp
| exp IS exp
| exp > exp
| exp < exp
| exp >= exp
| exp <= exp
| exp NOT = exp

pid ::= id
id ::= ID
newid ::= id
procall ::= CALL pid ( actpara )
return ::= RETURN | RETURN ( exp )
const ::= num | TRUE | FALSE | string | CONST
num ::= NUM
\end{verbatim}
string ::= STR
proof ::= ptype ( exp )
ptype ::= PRE | POST | ASSERT
empty ::=
IX. APPENDIX B: THE WEAKEST PRECONDITIONS FOR KL-1

This appendix details in terse form the properties, theorems, and predicate transformers which apply weakest preconditions to KL-1.

The important properties of the weakest precondition function are:

**Excluded Miracle, property 1**  
For any statement $S$,  
$$\text{WP}(S, \text{FALSE}) = \text{FALSE}.$$  
This property says that there are no initial states which will cause $S$ to halt in a state satisfying FALSE. $S$ can not terminate in a state satisfying FALSE since there are no such states.

**Monotonicity, property 2**  
For any statement $S$ and any assertions $Q, R$;  
$$\text{IF } (Q \Rightarrow R) \text{ THEN } (\text{WP}(S, Q) \Rightarrow \text{WP}(S, R)).$$  
The weakest precondition function preserves implication.

**Property 2A**  
For any statement $S$ and assertions $Q, R$, since  
$$(Q \text{ AND } R) \Rightarrow Q$$  
we have by PROPERTY 2  
$$\text{WP}(S, Q \text{ AND } R) \Rightarrow \text{WP}(S, Q).$$

**Property 3**  
For any statement $S$ and assertions $Q, R$  
$$\text{WP}(S, Q \text{ AND } R) = \text{WP}(S, Q) \text{ AND } \text{WP}(S, R).$$
Simultaneous substitution
For an assertion R, expressions A, B, C and variables X, Y, Z

\[ R(A/X, B/Y, C/Z) \]

denotes the assertion obtained by substituting the expressions A, B, C for all occurrences of the variables X, Y, Z respectively in the assertion R. Note that the substitution is simultaneous in the following sense. If Y occurs free in A then B is not substituted for those occurrences.

Sequential composition
Sequential statement composition is defined for any statement S1 (simple or compound) and any simple statement S2. Sequential composition is denoted by the semicolon, \( S_1; S_2 \). The weakest precondition of sequentially composed statements \( S_1 \) and \( S_2 \) where the assertion R is the desired result is:

\[ WP(S_1; S_2, R) = WP(S_1, WP(S_2, R)) \]

Concurrent composition
The concurrent composition of two statement lists \( S_{L1} \) and \( S_{L2} \) is denoted by the colon \( (S_{L1}: S_{L2}) \). The following sequence of steps are taken to obtain the weakest precondition, \( wp(S_{L1}: S_{L2}, R) \):

1) a) Find the range variables of \( S_{L1} \) and \( S_{L2} \) (i.e. the set of all variables which appear in left context).

b) Replace all left context occurrences of the range variables in \( S_{L1} \) and \( S_{L2} \) with unique temporary marking variables to form the new statement lists *\( S_{L1} \) and *\( S_{L2} \).
c) Replace all occurrences of the range variables in the post condition R with the temporary marking variables to form the new post condition *R.

2) Find the condition wp(*SL1:*SL2,*R).

3) Restore all remaining occurrences of the temporary marking variables in wp(*SL1:*SL2,*R) with the corresponding range variables yielding wp(SL1:SL2,R). Note that wp(*SL1:*SL2,*R) = wp(*SL2:*SL1,*R) if no range conflicts exist between SL1 and SL2. To generalize this technique to n statement lists in step 1 we find *SL1, ..., *SLn in the same manner as outlined, and in step 2 we find wp(*SL1:*SL2; ... ;*SLn,*R) instead of wp(*SL1:*SL2,*R).

The predicate transformers which define the weakest precondition function for the constructs of KL-1 are:

**The assignment** The assignment statement is of the form

\[ \text{VAR} := \text{EXP} \]

where, \( \text{VAR} \) is a variable defined in the program, and \( \text{EXP} \) is an expression whose value is to be assigned to \( \text{VAR} \). The weakest precondition of an assignment with desired result \( R \) is

\[ \text{WP(\text{VAR}=\text{EXP}, R) = R(\text{EXP}/\text{VAR}).} \]

**The alternatives** Four alternative constructs exist in KL-1. The need for the first construct arises from the desire to execute a sequence of statements depending on a
condition of the current state of the program. This is accomplished by using the construct **IF-THEN**, 

\[ \text{IF } B \text{ THEN } SL \text{ ENDIF} \]

where, \( B \) represents a boolean expression and \( SL \) represents a single or compound statement. The weakest precondition function for the **IF-THEN** is:

\[
WP(\text{IF-THEN-ENDIF}, R) =
(R \text{ AND } \neg B) \text{ OR } (B \text{ AND } WP(SL, R)) .
\]

The second alternative construct reflects the desire to execute exactly one of two possible sets of statements. This is accomplished by the **IF-THEN-ELSE** statement,

\[ \text{IF } B \]

\[ \text{THEN } SL_1 \]

\[ \text{ELSE } SL_2 \text{ ENDIF}. \]

The weakest precondition of the **IF-THEN-ELSE** is defined as:

\[
wp(\text{IF-THEN-ELSE-ENDIF}, R) =
(B \text{ AND } WP(SL_1, R)) \text{ OR } (\neg B \text{ AND } WP(SL_2, R)).
\]

The next construct arises from the desire to choose exactly one sequence of statements from many. Again the sequence chosen depends on the current state of the program. For this purpose a special **INCASE** statement is used rather than nested **IF-THEN-ELSE** or the typical **CASE-OF** statements found in most current languages.
The form of the INCASE statement is:

```plaintext
INCASE
   B1 DO SL1 ENDDO
   B2 DO SL2 ENDDO ...
   Bn DO SLn ENDDO
ENDINCASE.
```

The semantics of the statement require that the predicate

\[(B_1 \text{ OR } B_2 \text{ OR } \ldots \text{ OR } B_n)\]

be equivalent to the predicate TRUE (i.e. the boolean expressions must cover the state space). This condition requires that at least one set of statements be chosen for execution. In addition the boolean guards must characterize disjoint subsets of the state space. That is,

\[(B_i \text{ AND } B_j) = \text{FALSE} \text{ for all } i \neq j.\]

This condition requires that at most one set of statements be chosen. With these restrictions on the guards the weakest precondition function is:

\[
wp(\text{INCASE-ENDINCASE}, R) = (B_1 \text{ OR } \ldots \text{ OR } B_n) \text{ AND } \\
[ (B_1 \text{ AND } \neg B_2 \text{ AND } \ldots \text{ AND } \neg B_n \text{ AND } wp(SL1,R)) \]

\text{ OR } (B_2 \text{ AND } \neg B_1 \text{ AND } \neg B_3 \text{ AND } \ldots \text{ AND } \neg B_n \text{ AND } wp(SL2,R))

\ldots

\text{ OR } (B_n \text{ AND } \neg B_1 \text{ AND } \neg B_2 \text{ AND } \ldots \text{ AND } \neg B_{n-1} \text{ AND } wp(SL_n,R))].

The final alternative construct is the FORCASE. The impetus for this construct comes from the need to select for
execution zero or possibly many sets of statements. The semantics of the FOBCASE can best be described as the relaxation of the two conditions outlined for the INCASE statement. That is, any number of the boolean guards may evaluate true. It must also be mentioned that when more than one sequence of statements is executed, the order of execution is not defined. This requirement makes the FORCASE a parallel, as well as alternative construct. Care must therefore be taken that no conflicts occur in the statement lists. That is, for each variable in the range of a statement list, that variable cannot be used in the range of any of the other statement lists.

The form of the FORCASE is:

```
FORCASE
  B1 DO SL1 ENDDO
  B2 DO SL2 ENDDO ...
  Bn DO SLn ENDDO
ENDFORCASE
```

Since possibly several guards can be true, and hence their corresponding statement lists executed, the weakest precondition function contains a component for each possible combination of the statement lists. The number of combinations expressed by the function is $2^n$. Note that the first disjunct represents the case in which no alternatives are executed, and each other disjunct
corresponds to one of the other possible combinations. The weakest precondition function is:

$$wp(\text{FORCASE-ENDFORCASE}, R) =$$

$$(R \text{ AND } -\neg(B_1 \text{ OR } B_2 \text{ OR } B_3 \text{ OR } ... \text{ OR } B_n))$$

OR $$(B_1 \text{ AND } -\neg B_2 \text{ AND } -\neg B_3 \text{ AND } ... \text{ AND } -\neg B_n \text{ AND } wp(SL_1, R))$$

OR $$(B_1 \text{ AND } B_2 \text{ AND } -\neg B_3 \text{ AND } ... \text{ AND } -\neg B_n \text{ AND } wp(SL_1:SL_2, R))$$

$$...$$

OR $$(B_1 \text{ AND } ... \text{ AND } B_n \text{ AND } wp(SL_1:SL_2: ... :SL_n, R))$$

OR $$(B_2 \text{ AND } -\neg B_1 \text{ AND } -\neg B_3 \text{ AND } ... \text{ AND } -\neg B_n \text{ AND } wp(SL_2, R))$$

$$...$$

OR $$(B_n \text{ AND } -\neg B_1 \text{ AND } ... \text{ AND } -\neg B_{n-1} \text{ AND } wp(SL_n, R))$$

The repetitive constructs of the language consist of two single exit loops, the \textit{REPEAT-UNTIL}, and \textit{WHILE-DO}, and a multiple exit loop the \textit{REPEATEXIT}.

The form of the \textit{WHILE-DO} is:

\begin{verbatim}
WHILE B
  DO SL
ENDWHILE.
\end{verbatim}

The weakest precondition function for the while is defined in terms of the weakest precondition for a given number of iterations. We define:

$$h_0(R) = R \text{ AND } -B$$

$$h_k(R) = B \text{ AND } wp(SL, h_{k-1}(R)) \text{ for } k > 0.$$  
The index k represents the number of iterations through the
loop. As defined $H_k(R)$ is the weakest precondition for $k$ iterations of the while with post condition $R$. Thus:

$$wp(\text{WHILE-DO}, R) = H_0(R) \text{ OR } H_1(R) \text{ OR } ...$$

The \text{REPEAT-UNTIL} statement has the form:

\text{REPEAT SL}
\until B

Repeat the statement list SL until condition B is true. The predicates $H_k(R)$ are again defined to correspond to the weakest precondition for exit after the $k$-th iteration of the loop.

$$H_1(R) = wp(SL, R \text{ AND } B)$$

and $H_k(R) = wp(SL, \neg B \text{ AND } H_{k-1}(R))$ for $k>1$.

We now define:

$$wp(\text{REPEAT-UNTIL}, R) = H_1(R) \text{ OR } H_2(R) \text{ OR } ...$$

The final repetitive construct is the \text{REPEAT-EXIT}. The \text{REPEAT-EXIT} is a loop which allows many exits to a common point. The form is:

\text{REPEATEXIT}

SL1

\text{IF } B1 \text{ THEN EXIT } ...

SLn

\text{IF } Bn \text{ THEN EXIT }

SLn+1

\text{ENDRREPEATEXIT}

Each iteration of the loop has $n$ possible exits thus we
define:

\[ \text{wp}(\text{REPEATEXIT}, R) = H01(R \text{ AND } B1) \text{ OR } H02(R \text{ AND } B2) \text{ OR } \ldots \text{ OR } H0n(R \text{ AND } Bn) \text{ OR } H11(R \text{ AND } B1) \text{ OR } \ldots \]

The predicate \( H_{ji}(R \text{ AND } B_i) \) is the weakest precondition such that the post condition \( R \) is satisfied after \( j \) complete iterations of the loop, at the \( i \)-th exit. To define \( H_{ji}(R) \) we first define:

\[ K_{0i}(R) = \text{wp}(SL_i, R) \quad \text{for } 1 \leq i \leq n \]

and \( K_{11}(R) = \text{wp}(SL_{n+1}; SL_1, R) \).

The predicates \( H \) can now be defined as:

\[ H01(R) = K01(R) \]

\[ H_{j1}(R) = H_{j-1n}(K_{11}(R) \text{ AND } \neg B_n) \quad \text{for } j \geq 1 \]

and \( H_{ji}(R) = H_{ji-1}(K_{0i}(R) \text{ AND } \neg B_{i-1}) \quad \text{for } 1 < i \leq n \text{ and } j \geq 0. \)
The proof of the program ROOTFINDER is given as an example. The program is given below and its proof follows. ROOTFINDER uses the secant method to approximate the root of the function $F$. The PRE and POST statements for ROOTFINDER specify that a root exists in the interval $A, B$ and that the final value of $X$ is a good approximation of the root. The proof uses the predicate transformers, theorems and properties of the weakest precondition method which are presented in this thesis and in Appendix B. Notice that identifiers have been carefully chosen so that the renaming function $ST$ is not needed, and so that substitutions are unnecessary.
PROGRAM ROOTFINDER;
DECLARE VAR X,A,B,E OF NUMBER ENDDDECLARE;
PRE ((A*A+2*A-3)*(B*B+2*B-3)<0 AND E>0);
READ(A,B,E);
CALL SECANT(A,B,E,X);
POST (|X*X+2*X-3|<E);
END;
PROCEDURE SECANT(VAR A,B,E,X OF NUMBER);
IN: A,B,E;
OUT: X;
PRE (F(A)*F(B)<0 AND E>0);
X:=(A*F(B)-B*F(A))/(F(B)-F(A));
WHILE (|F(X)|>=E) DO
  ASSERT (F(A)*F(B)<0 AND E>0);
  INCASE
    F(A)*F(X)<0 DO B:=X; ENDDO;
    F(B)*F(X)<0 DO A:=X; ENDDO;
  ENDINCASE
  X:=(A*F(B)-B*F(A))/(F(B)-F(A));
ENDWHILE
WRITE(X);
POST (|F(X)|<E);
END;
FUNCTION F(VAR X OF NUMBER) OF NUMBER;
IN: X;
PRE (TRUE);
RETURN (X*X+2*X-3);
POST (F(X)=X*X+2*X-3);
END F
END SECANT
END ROOTFINDER

PROOF OF THE PROGRAM ROOTFINDER.

To prove program rootfinder it must be shown that the precondition PRE implies the weakest precondition. That is:

(E>0 AND (A*A+2*A-3)*(B*B+2*B-3)<0)

=> WP(CALL SECANT(A,B,E,X),|X*X+2*X-3|<E)

To calculate

WP(CALL SECANT(A,B,E,X),|X*X+2*X-3|<E)
the procedure call rule, EXT_CALL_ABS, is used. Lemmas 1 and 2 (the proofs of which follow below) verify that
\[(F(A) \times F(B) < 0 \text{ AND } E > 0) \Rightarrow WP(\text{SECANT'S BODY, } |F(X)| < E)\]
so we can apply the procedure rule to get:
\[(F(A) \times F(B) < 0 \text{ AND } E > 0 \text{ AND } (|F(X)| < E \Rightarrow |X^2 + 2X - 3| < E))\]
=> WP(CALL SECANT(A, B, E, X), |X^2 + 2X - 3| < E)
The proof of this program is simplified with regard to the verification of statements which invoke the function F. This simplification allows us to now substitute into the antecedent of the above implication, the expression computed by the function for the denotation of the function invocation. The result is
\[(A^2 + 2A - 3) \times (B^2 + 2B - 3) < 0 \text{ AND } E > 0 \text{ AND } (|X^2 + 2X - 3| < E \Rightarrow |X^2 + 2X - 3| < E)\]
Which can be reduced to
\[=(A^2 + 2A - 3) \times (B^2 + 2B - 3) < 0 \text{ AND } E > 0\]
This is the simplified precondition. Note that this condition is not the weakest precondition for ROOTFINDER. However, the precondition PRE implies this condition and thus, PRE implies
\[WP(\text{BODY OF ROOTFINDER, } |X^2 + 2X - 3| < E)\].
This leaves only the lemmas to be proven.

**LEMA 1.** Show that the precondition for secant implies its weakest precondition.
\[(F(A) \times F(B) < 0 \text{ AND } E > 0) \Rightarrow WP(\text{BODY OF SECANT, } |F(X)| < E)\]
PROOF.

\[
WP(X := (A*F(B) - B*F(A))/(F(B) - F(A))); \text{WHILE-DO; WRITE, } |F(X)| < E)
= WP(X := (A*F(B) - B*F(A))/(F(B) - F(A)); \text{WHILE-DO, } |F(X)| < E)
= WP(X := (A*F(B) - B*F(A))/(F(B) - F(A)), WP(\text{WHILE-DO, } |F(X)| < E)) \quad (5)
\]

\((F(A)*F(B) < 0 \text{ AND } E > 0) \implies WP(\text{WHILE-DO, } |F(X)| < E) \quad (6)\)

BY LEMMA 2.

\[
WP(X := (A*F(B) - B*F(A))/(F(B) - F(A)), F(A)*F(B) < 0 \text{ AND } E > 0)
\implies WP(X := (A*F(B) - B*F(A))/(F(B) - F(A)), WP(\text{WHILE-DO, } |F(X)| < E))
\]

BY proposition 2 on lines (5), (6).

But the antecedent of this condition can be reduced.

\[
WP(X := (A*F(B) - B*F(A))/(F(B) - F(A)), F(A)*F(B) < 0 \text{ AND } E > 0)
= F(A)*F(B) < 0 \text{ AND } E > 0 \quad \text{BY the := transformer. So}
\]

\[
(F(A)*F(B) < 0 \text{ AND } E > 0) \implies
WP(X := (A*F(B) - B*F(A))/(F(B) - F(A)), WP(\text{WHILE-DO, } |F(X)| < E))
\]

This proves the lemma.

LEMMA 2. \(P \implies WP(\text{WHILE-DO, } |F(X)| < E)\)

WHERE: \(P = (F(A)*F(B) < 0 \text{ AND } E > 0)\).

PROOF.

The theorem used to prove this WHILE-DO is presented in the Related Works section of this thesis. To apply this theorem, it must be shown that:

\((P \text{ AND } |F(X)| < 2E) \implies WP(\text{INCASE;}X := (A*F(B) - B*F(A))
/(F(B) - F(A)), P)\)

Application of the rule will give us.

\(P \implies WP(\text{WHILE-DO, } P \text{ AND } |F(X)| < E)\).
WP(INCASE; X := (A*F(B) - B*F(A))/(F(B) - F(A)) / P)
= WP(INCASE, P((A*F(B) - B*F(A))/(F(B) - F(A)) / X))

BY ; =. But

P((A*F(B) - B*F(A))/(F(B) - F(A)) / X) = P BY manipulation.

So we want WP(INCASE, P)
= WP(INCASE, F(A)*F(B)<0 AND E>0)
= (F(A)*F(X)<0 OR F(B)*F(X)<0) AND
  (((F(A)*F(X)<0 AND F(B)*F(X)<0 AND F(A)*F(X)<0 AND E>0)
  OR (F(B)*F(X)<0 AND F(A)*F(X)<0 AND E>0))

BY the incase and := transformers.

= (F(A)*F(X)<0 OR F(B)*F(X)<0) AND
  (((F(A)*F(X)<0 AND F(B)*F(X)<0 AND E>0)
  OR (F(B)*F(X)<0 AND F(A)*F(X)<0 AND E>0))

BY simplification.

To apply the WHILE-DO theorem it must be shown that the above
condition is implied by
(F(A)*F(B)<0 AND E>0 AND |F(X)|>E)

An informal argument is used to show this implication.

Observe that the first conjunct of the antecedant,
F(A)*F(B)<0, says that F(B) and F(A) are both nonzero and of
opposite signs. The remainder of the antecedant says that X
is not an acceptable root; that is, F(X) is sufficiently
different from zero. The antecedant implies the left
conjunct of the consequent, (line 1). This is true since (1)
says that \( F(X) \) must be of the same sign as either \( F(A) \) or \( F(B) \). The antecedant also implies the right conjunct of the consequent, that is, (2) and (3). To see this, assume that all three of \( F(A) \), \( F(B) \), and \( F(X) \) are different from zero and that \( F(A) \) and \( F(B) \) are of opposite signs. It follows that \( F(X) \) has the same sign as the function at one of the endpoints and a different sign than the other.

The implication holds and thus, the WHILE theorem can be applied. The result is

\[
P \Rightarrow WP(\text{WHILE-DO}, P \text{ AND } |F(X)| < E).
\]

BUT

\[
WP(\text{WHILE-DO}, P \text{ AND } |F(X)| < E) \Rightarrow WP(\text{WHILE-DO}, |F(X)| < E)
\]

SO,

\[
P \Rightarrow WP(\text{WHILE-DO}, |F(X)| < E)
\]

Which proves the lemma.