1976

Name management in the construction of large programs

Patricia Ann Carr

Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/rtd

Part of the Computer Sciences Commons

Recommended Citation


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

This material was produced from a microfilm copy of the original document. 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 original submitted.

The following explanation of techniques is provided to help you understand markings or patterns 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 thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

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

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from "photographs" if essential to the understanding of the dissertation. Silver prints of "photographs" may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

5. PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

Xerox University Microfilms
300 North Zeeb Road
Ann Arbor, Michigan 48106
CARR, Patricia Ann, 1946-
NAME MANAGEMENT IN THE CONSTRUCTION OF LARGE PROGRAMS.

Iowa State University, Ph.D., 1976
Computer Science

Xerox University Microfilms, Ann Arbor, Michigan 48106
Name management in the construction
of large programs

by

Patricia Ann Carr

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

1976
# TABLE OF CONTENTS

## CHAPTER I. INTRODUCTION
- Review of the Literature 4
- Statement of the Problem 12
- Plan of the Thesis 13

## CHAPTER II. THE MODULE MANIPULATION CONSTRUCTS
- Modules and Environments 15
- Module Definition Constructs 17
- Module Manipulation Constructs 21
- Environment Definition Constructs 24
- Environment Manipulation Constructs 25

## CHAPTER III. THE IMPLEMENTATION MODEL
- Overview of the Implementation Model 29
- The Base Language, BL 31
- The Input Phase 34
- The Construction Phase 48
  - Module resolution 51
  - Performing a transformation 52
  - Creating a fully-defined module 61
  - An example of the construction process 69
- The Execution Phase 77
  - Overview of the execution phase 77
  - The simulator 79
  - The assignment statement 81
  - The output statement 83
  - The conditional statement 83
  - The branch statement 84
  - The block entry statement 85
  - The block exit statement 86
The procedure call statement 87
The procedure return statement 88
The identifier binding strategy, IBS 89
The environment binding strategy, EBS 91
An example of the execution process 91

CHAPTER IV. A NONTRIVIAL EXAMPLE OF THE IMPLEMENTATION MODEL IN ACTION 105

The Example 109
The Initial Graph of the Program 111
The Construction Process Applied to This Example 116
The Trace of the Program Execution 126

CHAPTER V. CONCLUSIONS AND SUGGESTIONS FOR ADDITIONAL INVESTIGATION 165

BIBLIOGRAPHY 170
ACKNOWLEDGMENTS 173
APPENDIX. AN IMPLEMENTATION OF THE CONSTRUCTION PHASE 174
# LIST OF FIGURES

<p>| Figure 2.1. | Partial BNF grammar for the module manipulation constructs | 28 |
| Figure 3.1. | BNF grammar for BL | 33 |
| Figure 3.2. | An example of the <code>include</code> transformation | 56 |
| Figure 3.3. | An example of interaction between module manipulation constructs | 64 |
| Figure 3.4. | A sample ML program | 70 |
| Figure 3.5. | The initial graph for the program in Figure 3.4 | 71 |
| Figure 3.6. | The subgraph for the local variables of routine main, after transformation | 74 |
| Figure 3.7. | The subgraph for the nested block, after transformation | 75 |
| Figure 3.8. | The program skeleton for the program in Figure 3.4 | 76 |
| Figure 3.9. | S1, the initial record skeleton | 92 |
| Figure 3.10. | S2, the effect of the call to main | 93 |
| Figure 3.11. | S3, the state prior to execution of line 14 | 94 |
| Figure 3.12. | S4, the state prior to execution of line 18 | 95 |
| Figure 3.13. | S5, the state prior to execution of line 19 | 96 |
| Figure 3.14. | S6, the state prior to activation of g | 98 |
| Figure 3.15. | S7, the state prior to execution of line 6 | 99 |
| Figure 3.16. | S8, the state prior to execution of line 20 | 100 |
| Figure 3.17. | S9, the state prior to execution of line 22 | 101 |
| Figure 3.18. | S10, the state just prior to activation of f | 102 |
| Figure 3.19. | S11, the state just prior to execution of line 6 | 103 |</p>
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.20</td>
<td>S12, the state prior to execution of line 23</td>
<td>103</td>
</tr>
<tr>
<td>3.21</td>
<td>S13, the final state of execution</td>
<td>104</td>
</tr>
<tr>
<td>4.1</td>
<td>The source listing of the example</td>
<td>105</td>
</tr>
<tr>
<td>4.2</td>
<td>Graphical representation of module point</td>
<td>111</td>
</tr>
<tr>
<td>4.3</td>
<td>Graphical representation of module line</td>
<td>112</td>
</tr>
<tr>
<td>4.4</td>
<td>Graphical representation of module intrcp</td>
<td>113</td>
</tr>
<tr>
<td>4.5</td>
<td>Graphical representation of module slope</td>
<td>114</td>
</tr>
<tr>
<td>4.6</td>
<td>Initial representation of set of declarations local to routine main</td>
<td>114</td>
</tr>
<tr>
<td>4.7</td>
<td>Initial representation of set of declarations within first nested block</td>
<td>115</td>
</tr>
<tr>
<td>4.8</td>
<td>Initial representation of the global environment</td>
<td>117</td>
</tr>
<tr>
<td>4.9</td>
<td>Effect of is transformation (line 93)</td>
<td>118</td>
</tr>
<tr>
<td>4.10</td>
<td>Effect of names transformations (lines 95, 96)</td>
<td>120</td>
</tr>
<tr>
<td>4.11</td>
<td>Effect of names transformations (lines 99, 100)</td>
<td>121</td>
</tr>
<tr>
<td>4.12</td>
<td>Effect of first stage of include transformation (line 101)</td>
<td>122</td>
</tr>
<tr>
<td>4.13</td>
<td>Effect of secondary include transformation for the module slope (line 50)</td>
<td>124</td>
</tr>
<tr>
<td>4.14</td>
<td>The completely defined sets of declarations nested within the routine main</td>
<td>125</td>
</tr>
<tr>
<td>4.15</td>
<td>Program skeleton for example</td>
<td>127</td>
</tr>
<tr>
<td>4.16</td>
<td>The initial state of the execution</td>
<td>129</td>
</tr>
<tr>
<td>4.17</td>
<td>The state prior to execution of line 94</td>
<td>129</td>
</tr>
<tr>
<td>4.18</td>
<td>The state prior to execution of line 113</td>
<td>129</td>
</tr>
<tr>
<td>4.19</td>
<td>The state prior to execution of line 116</td>
<td>131</td>
</tr>
</tbody>
</table>
Figure 4.20. The state prior to execution of line 117.............. 131
Figure 4.21. The state prior to execution of line 13.............. 132
Figure 4.22. The state prior to execution of line 118............ 134
Figure 4.23. The state prior to execution of line 121............ 134
Figure 4.24. The state prior to execution of line 57.............. 135
Figure 4.25. The state prior to execution of line 42.............. 136
Figure 4.26. The state prior to execution of line 22.............. 138
Figure 4.27. The state prior to execution of line 4.............. 139
Figure 4.28. The state prior to execution of line 23.............. 141
Figure 4.29. The state prior to execution of line 24.............. 142
Figure 4.30. The state prior to execution of line 44.............. 143
Figure 4.31. The state prior to execution of line 76.............. 144
Figure 4.32. The state prior to execution of line 84.............. 145
Figure 4.33. The state prior to execution of line 45.............. 147
Figure 4.34. The state prior to execution of line 48.............. 148
Figure 4.35. The state prior to execution of line 58.............. 149
Figure 4.36. The state prior to execution of line 122............ 149
Figure 4.37. The state prior to execution of line 63.............. 151
Figure 4.38. The state prior to execution of line 123............ 151
Figure 4.39. The state prior to execution of line 104............. 152
Figure 4.40. The state prior to execution of line 104............. 152
Figure 4.41. The state prior to execution of line 13............. 153
Figure 4.42. The state prior to execution of line 105............. 155
Figure 4.43. The state prior to execution of line 106............. 156
Figure 4.44. The state prior to execution of line 30................. 157
Figure 4.45. The state prior to execution of line 76................. 159
Figure 4.46. The state prior to execution of line 84................. 160
Figure 4.47. The state prior to execution of line 35................. 161
Figure 4.48. The state prior to execution of line 108.............. 162
Figure 4.49. The state prior to execution of line 109.............. 163
Figure 4.50. The state prior to execution of line 111.............. 163
Figure 4.51. The state prior to execution of line 126.............. 164
Figure 4.52. The state prior to execution of line 127.............. 164
Figure 4.53. The state prior to execution of line 129.............. 164
Figure 4.54. The state at program termination..................... 164
CHAPTER I. INTRODUCTION

One of the most pressing problems confronting the community of computer users is the high cost of developing and maintaining programs. Falling hardware costs and increasing hardware reliability have both played a major role in increasing reliance on the computer in data management and problem solving. Using the computer requires the existence of programs, which have exhibited a distressing tendency to be wrong and to be extremely expensive. In the transformation of a collection of hardware into a usable computer system, the software development costs heavily dominate the entire expense. Furthermore, rapid technological advances, culminating in the introduction of more sophisticated hardware, have not been met with equally rapid development of software which can make effective use of the hardware.

The computer science community has responded to this problem by instituting investigation into such diverse areas as the psychology of programming, program verification techniques, language design to increase program reliability, programming methodology, and the like, and a new area of research, designated software engineering, has emerged. All of these areas tend to address the questions of how the programming environment might best be configured to promote programming productivity.

Implicit in this approach is the assumption that programming is unaffected by the size of the resulting program; recursive use of the insights gained in any of the above areas of research will yield an
acceptable program. This assumption is not borne out by psychology; rather there exists a level of complexity which represents the highest level of complexity understandable by one person. Especially when dealing with a large program, in order for comprehension of any part to be possible, it becomes psychologically necessary to separate that part from the rest of the program. The greater its interaction with the rest of the program, the less comprehensible the part. The program development process, then, requires the ability to naturally focus on a particular part of a program until solved. This decomposition of a program should reasonably be reflected in the resulting large program to provide for program maintenance and debugging. Programming languages should provide mechanisms to facilitate the use of these natural decompositions in the construction of the whole from its parts.

One approach to supporting natural decomposition is to reduce the artificiality of a problem solution when rendered as a program written in some programming language. This requires that a programming language should provide the formalisms which are naturally used in the transition from a problem to a solution to the problem. At best, this is a long-range goal, requiring different languages for different classes of problems. Furthermore, the selection of an appropriate set of formalisms is fraught with peril, based as it is on psychology, which, as yet, has no answers for how one might determine a suitable set.

A second approach is to provide suitable mechanisms for decomposing a program into semiautonomous parts, reflecting the natural decomposition of the problem, and providing mechanisms for specifying the desired
interactions between the parts. Each part could be defined and developed using any algorithm construction language deemed appropriate. The resulting algorithm would then be rendered in an artificial programming language, but more of the intent of the solution could be provided. This appears to be an achievable short-range goal.

One of the primary capabilities provided by any programming language is that of abstracting the manipulation and storage of information from the physical location in which this information is stored. Data are then referenced by name. The correlation of names appearing within the program text and values arising during the execution of a program is provided by the naming structure of a programming language. Any method of program decomposition is reflected within the naming structure of a programming language in order to provide for any communication between the pieces of the decomposed program. The best known naming structure is block structure as it is defined in ALGOL and its derivatives. Block structure provides two decomposing mechanisms: the procedure and the nested block. In both of these cases, the communication links can be ill-defined, as indicated in Wulf and Shaw (32).

This research is an investigation into the question of extending block structure to provide additional, perhaps more natural, ways of decomposing a large program, and to provide mechanisms for succinctly describing the communication between the parts of the decomposed program.
Review of the Literature

It seems somewhat presumptuous and tedious to review all of the literature pertaining to the decomposition technique of "stepwise refinement." However, since it is probably the best known program decomposition technique, it cannot reasonably be overlooked as part of the body of literature pertaining to the area.

As originally described by Dijkstra (4), stepwise refinement represents a disciplined approach for inducing order from the chaos which results when the level of complexity of a problem exceeds the bounds of comprehension. Dijkstra demonstrated the technique of stepwise refinement in terms of a small program environment, which is induced by the ability of one person to comprehend the whole. Stepwise refinement yields a procedural decomposition of the whole, implying that the parts of the whole are determined by what is done. Refinement of the parts of the program proceed in parallel until such time as the interactions among the parts is well enough defined to allow for further refinement independent of the other parts. This form of decomposition, although well-suited for translation into current decomposition mechanisms, tends to require the definition of data objects before all the operations upon those objects are defined. In particular, implementation details become part of the communication links between the pieces of the decomposition. This tends to add undesirable complexity to the interfaces.

In contrast, Parnas (22, 23) proposed as a decomposition criterion the notion of information hiding. Using this criterion, all of the operations upon a data object (or class of data objects) are defined
and collected within a module. Questions of implementation for the data object are reserved until such time as the operations are programmed. Using the information hiding principle, decomposition leads to the encapsulation of implementation decisions within semiautonomous units, implying that their impact on the program as a whole is severely restricted. Parnas has reported on the efficacy of this technique only as a class project, written in FORTRAN (20). The drawback to this approach, as he has investigated it, is that it has been tried only within the framework of a program which could reasonably be constructed by one person. However, he has demonstrated that, once the decomposition is made, each semiautonomous part can be generated in isolation and that no unexpected side effects accrue. Within a large program environment, this is clearly of value.

Liskov and Zilles (17) have developed a language for structured programming which incorporates the notion of abstract data type. An abstract data type defines a class of data objects which are characterized solely by the operations upon them. These operations are modeled by procedures which manipulate some internal representation for objects of the abstract data type. Collectively, the operations and a description of the internal representation form a cluster. The cluster shields the user from any knowledge of the internal representation while providing access to the operations contained within the cluster. This is an obvious use of Parnas' modules. Clusters are referenced by name; operations within a cluster are referenced unambiguously by a fully qualified name, consisting of both cluster identifier and operation name. Further
restriction is placed on the names appearing within a cluster: the free names must be the names of other clusters only. All cluster names are global to any program written in the language; clusters are collectively stored within a cluster directory. Hence, clusters are compilable in a framework outside any program. In later reports on their work, Liskov and Zilles have focused on the mathematical implications of their work (18).

Earley (5) has considered the question of module definition and manipulation as related to the naming structure of the language defined. Earley defines a module to be a set of declarations of any type supported by an underlying block structured language. Earley proposes two types of modules: those which contain only one routine declaration, a ROUTMOD, and a more general module type, MOD, which contains several declarations. In addition, modules may have module parameters. He defines three module manipulation constructs which are forms of declarations and, thus, eligible for inclusion within a module. Any free names within a module are bound within the context of the module definition; free names are not restricted to be only other module references. Earley provides a mechanism for designating the links by which the module communicates outward. Those declarations which are to be shielded and, hence, do not constitute modular access paths, are tagged.

Three module manipulation operations are defined. The first is used to signal intent to make use of a module. This construct augments the context in which it appears by a set of declarations identical to the set of untagged declarations within the module referenced. In
addition, any module parameters are bound within the context of the construct to those names having the same identifier. These names are determined by the traditional identifier search of block structured languages. When all the module parameters are bound, storage is allocated for the local variables of the module. Those names which appear within a context by virtue of the use of this construct are bound to the names having the same identifier within the module. Compilation of the construct results in the replacement of the construct and module reference by the set of declarations constructed from the module definition. Because names are being created within the context of the module manipulation, a renaming operation is provided, to allow for explicit control over the set of names being implicitly declared. This provides the power to prevent unexpected identifier clashes and unexpected shielding of declarations appearing with a block enclosing that in which the module construct appears. Flexibility is also provided to impact the module parameter binding, allowing the explicit designation of the names to which the parameters are to be bound and allowing the parameters to be bound to constants. Because module parameters may be bound to existing names, this is a form of outward binding, and can be interpreted as a sharing mechanism.

A second module operation is defined to allow the binding of module parameters without exposing the untagged declarations. Any module parameters are bound as described above. The only name implicitly declared is the identifier of the resulting bound module which is the same as the module referenced. This allows reference to the bound module for the purpose of exposing the untagged declarations at a
later time, while the name of the bound module is statically known. Exposure of the untagged declarations is accomplished by recourse to the first operation described. Again, since a name is being implicitly declared, a mechanism is provided for explicitly designating the name which is to become that whose value is the bound module.

The third mechanism is designed for use within a module. If, within a module, the first construct is used, the names which become known within the module are implicitly tagged, thereby shielding them from exposure. The third operation performs just as the first except that the exposed declarations are not tagged.

Under Earley's scheme, modules may be defined in two ways. A module may be defined by enumerating the internal declarations. The second module definition technique provided allows a module to be created from another by binding one or more of the parameters of the original module. The resulting module "has the same status" as the original module.

Module manipulations of the kind defined by Earley clearly affect the naming structure of any block structured language augmented by module manipulations. Johnston has devised an operational model (9, 10, 11, 12) for investigating the naming structure of block structured languages. The contour model displays the program execution as a series of snapshots, taken after the interpretation of each instruction appearing within the program text. Each snapshot contains a program skeleton, which is a static representation of the program text, and a record skeleton, which is a dynamic representation of the record of execution.
The program skeleton is a collection of linked cells of two types. One cell type is the contour cell, which is formed from the set of declarations which comprise either the declarations for the local variables within a block, or the declarations for the formal parameters declared within a procedure header. A contour cell is composed of a set of declaration subcells and also includes an environment link which designates the contour cell generated from the declarations within the immediately enclosing block or, in the case of the contour formed from the formal parameter list of a procedure, the block in which the procedure is declared. The contour cells, together with their environment links, depict the tree induced by block structuring for a given program. The second type of cell, the code cell, consists of a set of instruction subcells which represent a set of instructions appearing within the program text. Each code cell designates the contour cell containing the declarations with which the instructions are associated.

The record skeleton is formed from the interpretive execution of the instructions included within the program skeleton. Assuming invariant program structure, the record skeleton is formed from linked contour cells only. Each contour cell within the record skeleton is formed from an appropriately chosen contour cell within the program skeleton. The set of declaration subcells within a contour cell in the record skeleton is identical to the set of declaration subcells contained within the contour cell from which it was formed. A contour cell within the record skeleton is linked to some other contour within the record skeleton at the time it is created in accordance with the
naming structure being employed. Additionally, a contour within the record skeleton designates the contour in the program skeleton from which it was formed.

Johnston has divided the naming structure of a language into two parts: an identifier binding strategy and an environment binding strategy. The environment binding strategy is used to link a newly created contour in the record skeleton to some existing contour.

Execution of a program is modeled in terms of a processor, which is said to be executing within some contour cell within the record skeleton. This is a reflection of the fact that every instruction within a block structured language is associated with a particular set of declarations within the program text. The processor environment is defined to be that set of declaration subcells which are accessible by virtue of being contained within the set of contour cells selected by traversing the chain of contour environment links which begins with the contour in which the processor is executing.

Every reference to a name appearing in the instruction being modeled is matched to a declaration subcell for that name appearing within the processor environment by the identifier binding strategy being employed. Because the environment binding strategy is employed in the construction of the processor environment, it is obvious that these two functions together model the naming structure of a language.

Johnston has implicitly defined a name to be a triple of property-value pairs. The properties of interest are type, identifier, and value. Johnston provides for two types of values, a label value and a
simple (or nonlabel) value. A label value is an ordered pair \((ip, ep)\), where the \(ip\) designates some instruction subcell of some code cell and the \(ep\) designates some contour within the record skeleton. In his model, Johnston defines the value of a procedure to be a label, the \(ip\) of which designates the first instruction of the entry point coding for the procedure and the \(ep\) of which designates the contour in which the procedure was declared. The value of a label is also an ordered pair \((ip, ep)\), the \(ip\) of which designates the instruction subcell associated with the label and the \(ep\) of which designates the contour within which the label identifier is declared. The environment binding strategy is employed within the execution of any instruction which can alter the processor environment. These instructions are block entry and exit, procedure call and return, and branches. The effect of block entry and exit on the processor environment is well-defined: block entry causes a newly created contour to be linked to the contour in which the block entry instruction is executed; block exit causes the processor to return to the contour designated by the environment link of the contour in which the block exit is executed. In the case of procedure call, the environment binding strategy makes use of both the \(ep\) portion of the referent and the contour in which the processor is executing to determine the value of the environment link for the formal parameter contour. Procedure return can be considered a form of branch if, on procedure call, a label value is generated to be used as the return address. In the case of a branch, the environment binding strategy, using the \(ep\) portion of the label value and the current operating
environment, determines the next contour in which the processor will execute.

The contour model provides a time-dependent framework in which to investigate a fundamental portion of a language definition which, for efficiency reasons, is subsumed within the compilation process.

Statement of the Problem

The thesis of this research is that a set of module manipulation constructs can be appended to a block-structured language without seriously perturbing the simplicity and clarity of the naming structure of the resulting language while providing a much-needed alternative to procedural decomposition.

The resulting language provides two forms of decomposition techniques. In addition to procedural decomposition, modular decomposition, based on the information-hiding principle, is provided. Hence, the programmer, when generating his program, can utilize modules to aid in the solution of his task. These modules shield the programmer from certain implementation details while allowing use of any operations on the data structure provided within the module. In addition, the use of modules allows a concise specification of the communication links between modules, and between a module and the context of its use. A language which supports both modular and procedural decomposition will be termed a module-structured language.

This investigation is motivated by the conviction of the author that the programming language chosen as the vehicle for the communication
of a problem solution to the computer should impose as few constraints as possible on the problem solver. While it is as yet infeasible to allow the problem-solving program to be cast in terms of the set of constructs under which it was derived, whatever decomposition of the problem which seemed most natural should be readily modeled within the programming language. Furthermore, these constructs should be designed with the large program environment in mind, in which many people are involved in the generation of the solution to a problem, since the large program environment probably predominates. Concern with the large program environment has, as a corollary, concern with the efficiency of execution of the large program. It is the author's opinion that any language designed to facilitate the construction of large programs, as is the case with a module-structured language, must not exhibit significant degradation of program execution efficiency when compared to more traditional languages. It will be argued that, although the compilation process becomes somewhat more complex using a module-structured language, program execution is not seriously slowed by the use of modules.

Plan of the Thesis

This research is presented within the following framework. In the next chapter, an informal description of the module manipulation constructs is given. The effect of these constructs upon block structure is described; in their totality, these perturbations, together with block structure, describe the naming structure of the resulting
Chapter III defines the semantics of the module manipulation constructs within an implementation model, which is an extension of the contour model for block-structured languages (9, 10, 11, 12). The implementation model consists of three phases: one for input, and the construction of the initial data structures; one which performs the module manipulations; and one which simulates the execution of the naming structure.

Chapter IV displays the execution of the implementation model on a nontrivial example. Careful attention is given to the level of interaction which can exist between instances of the module manipulation constructs.

Chapter V summarizes the conclusions reached from this research, poses some additional questions, and speculates on future directions of research into the use of more generalized decomposition techniques.
CHAPTER II. THE MODULE MANIPULATION CONSTRUCTS

In this chapter, a module-structured language is defined, where a module-structured language is the result of extending a block-structured language to allow for the use of modules. The module-structured language will be informally described; although a syntax will be presented, it is used only for the purpose of illustration. The semantics of the language are defined in terms of an implementation model, which is presented in the next chapter.

A module is defined to be a set of declarations which are to be manipulated as a unit. Modules exist implicitly within current block-structured languages in two forms: the formal parameters of a procedure form a module, and the local variables declared within a block also form a module. The explicit definition of modules represents a generalization of the method of variable definition and scope as defined within block-structured languages.

Modules and Environments

In this section, the notion of a module is defined and its relationship to the notion of an environment is clarified.

A module consists of a set of declarations to be manipulated as a unit. It is expected that this set may contain declarations of any type supported by the underlying block-structured language. In particular, this set may contain declarations of procedures. In all block-structured languages known to the author, the declaration of a procedure
consists of a declaration of the procedure identifier and the statement of the body of code which is to be the value of the procedure identifier. When discussing the set of declarations which form a module, the value associated with any identifier contained within the set of declarations is not considered as part of the set. Thus, if a procedure, C, having a parameter A and a local variable X, is included within a set of declarations forming a module, the module contains a declaration for C and does not contain declarations for A or X.

One of the functions served by a module definition is the specification of the subset of the set of declarations which may serve as access paths. These access paths designate those declarations which may be referenced extramodularly in order to make use of the module within a program. Any declaration within a module which may serve as an access path is tagged with the attribute accessible. A declaration which has not been tagged must be shielded from any manipulation from outside the module. These shielded declarations represent the set of declarations which hide information within the module.

An environment is defined to be a set of names, existing during program execution, which are related by virtue of the relationship existing between the corresponding declarations within the program text. Hence, a module represents a descriptor for an environment. It is convenient to think of an environment as a data structure implicitly defined within the program text. Since these entities exist during program execution, it seems reasonable to provide a (restricted) set of operations on environments, if they can be named. The environment
created from a module will be explicitly created as a side-effect of some of the module manipulation constructs. By providing mechanisms for associating a name with an environment, greater freedom and greater control is provided the programmer in defining those parts of his program in which an identifier is to be known. This provides some of the utility of the mechanisms proposed by George and Sager (7), by allowing an explicit designation of an environment in which a given name is to be evaluated.

By choosing proper operations on environments, it is possible to resolve all environment references during the compilation process, while still providing complete freedom in designating the environment in which a name is to be evaluated. Thus, environment references appearing within the program text can be viewed as directives to the compiler, used in correlating a reference to an identifier with the declaration of that identifier.

Module Definition Constructs

The process of module definition is one of associating a module name with a set of declarations. Module manipulations provide mechanisms for creating a set of declarations from sets previously associated with module names. This high degree of interaction provides flexibility and power from the constructs, but leaves behind a complicated task of describing the constructs meaningfully.

A set can be defined in one of two ways. The set can be defined by an explicit enumeration of its elements, or it can be defined in
terms of a property which is true of every element within the set, and which can be used to determine whether an object is an element of the set. In a module-structured language, there exist methods of defining sets of declarations which are analogous to each of the set theoretic methods of set definition. The module definition techniques described below provide the mechanisms by which a module name is associated with a set of declarations. Two methods of module definition are provided. The first is used when some subset of the declarations is explicitly enumerated. In this case, the keywords mod and endmod are used to delimit the set of declarations comprising the module. The module name appears immediately following the keyword mod. As an example,

```
mod STACKMOD;

decl STACK:array;

decl TOP:integer;

accessible routine PUSH(ITEM);

begin;
  :
  :
end;

accessible routine POP;

begin;
  :
  :
end;

endmod
```

outlines the definition of the module named STACKMOD, which consists of four declarations, two of which designate access paths. The set of declarations includes the declarations for STACK, TOP, PUSH and POP.
only. ITEM, which is a parameter for the routine PUSH, is not included in the set of declarations referenced by STACKMOD. No declaration contained within the body of either of the routines is included in the set of declarations referenced by STACKMOD.

It is possible that, because of the decomposition inducing the definition of modules, a parameterization for the module is useful. Parameterization allows the module user to transmit some information to the module, to be used during the execution of any routine contained within the module. For example, in the case of STACKMOD, it might be useful to set an upper bound on TOP, when, for a given application, the programmer knows the maximum stack length, and, for testing purposes in particular, desires stack overflow to be signaled when this bound is exceeded. The concept of a module parameter is provided to allow the programmer making use of a module to provide information to it. Module parameters, if used, form a subset of the set of declarations to be associated with a module name. Module parameters are not part of the intermodule access paths, which are used to couple modules together in creating a program. Module parameters may be set to a constant value at the time the module is manipulated, but no access path to the module is produced using a module parameter. The module POSTFIX, outlined below, has a parameter, ERROR, to be used as trap in case of error.
A module name may be associated with a partially enumerated set of declarations. A partially enumerated set of declarations consists of a list of declarations together with a set of module manipulation constructs. Once these module manipulation constructs are resolved (that is, the module operations are performed) a completely enumerated set of declarations will be formed. Description of this form of module definition will be deferred until the module manipulation constructs are described.

A third form of module definition technique is provided. In this case, a module name is associated with a local copy of an existing module. Any of the parameters of the existing module can be made constant, forming a new module having fewer parameters, although this is not necessary. The construct \textit{is} is defined for this type of module definition. The declaration

\begin{verbatim}
STACK is STACKMOD;
\end{verbatim}

creates a local copy of the module STACKMOD, and associates with it the name STACK. The definition of a new module, POLISH, from POSTFIX while setting the parameter ERROR to an existing routine, ERRORHANDLER, is
accomplished by the declaration

POLISH is POSTFIX(ERRORHANDLER);

Module Manipulation Constructs

The module manipulation constructs are considered to be forms of declaration statements. As such, these constructs appear within the context of a set of declarations. Performing the module operation designated by a particular module manipulation construct causes some change to be made to the set of declarations within which the construct appears.

Once a module exists, the most obvious manipulation to be provided is one which allows access to the accessible declarations of a module. This manipulation can be characterized as exposing the accessible declarations by forming access paths from the context in which the construct appears to the set of declarations comprising the module. The include construct is used for this purpose. The effect of performing the manipulation designated by the include construct on the context in which the include construct appeared, is the replacement of the statement in which the include appears with a set of declarations defined from the set of accessible declarations of the referenced module. The declarations so added are linked to the complete set of declarations from which they were generated; when referenced within some instruction, they will be evaluated within the context of their original definition.

At the time of the include, it is possible to set the values of a number of the parameters of the referenced module. These parameters are set to constant values, of any type which may be declared within the module.
itself. The effect of this extended include can be described as the creation of an unnamed module which is then included. Hence, the declaration form

    include STACKMOD;

is replaced by declarations of two routines PUSH and POP, which are linked back into the module STACKMOD; and

    include POSTFIX(TRAP);

is replaced by the declaration of the routine TRANSFORM, linked back to an unnamed module, created from POSTFIX, in which the parameter value has been set to the value of TRAP.

Because declarations are being implicitly made, it is necessary to provide mechanisms to control the set of declarations formed as a result of the use of a module. The effect is to explicitly designate the declaration which is added to the set under construction, while tying that declaration to the one within the module which it replaces. This renaming becomes part of the process of linking the new declaration back to the module from which it was generated. The construct

    include STACKMOD with PUSH-PUSHN,POP-POPN;

is an implicit declaration of two routines, PUSHN and POPN, which are linked back to the declarations for PUSH and POP as they exist in the definition of the module STACKMOD.

If a module is defined using the include construct, the declarations appearing within the module because of the include are not tagged; and are then local to the module. In constructing large programs, it is expected that modules may be defined in terms of other modules where
this is not desirable; that is, some of the declarations which are not explicitly enumerated are to be part of the intermodule access paths. To accommodate this, the insert operation is defined. Within a module definition, insert has the same effect, and the same capabilities, as include except that the declarations appearing because of the insert are tagged accessible.

The definition of a module which is, itself, defined in terms of some other module, requires that the choice of that other module must be unambiguous. Any module reference appearing within a module definition is a free name; the free names within a module are statically bound within the context of the definition of the module.

The local variables within a block are also represented by a set of declarations. In block-structured languages, this set is explicitly enumerated. Within a module-structured language, this is not necessarily so. The include construct may be used in the creation of the set of declarations for the local variables for a block, with the same effect on the context of the include as defined above. The module reference is statically bound within the context of the include. However, the insert construct may not be used in defining a block since no declarations, other than those appearing as part of the set of declarations associated with a module name, can be tagged accessible.

Modules may be defined globally or within a block. Local definition of a module restricts the scope of the module name, and can be done using any of the module definition constructs defined above. Aside from restricting the scope of the module name, the impact of defining a
module locally is on the binding of the module references appearing within the newly defined module, in the case where the newly defined module is created by use of the `is` construct.

**Environment Definition Constructs**

Both of the constructs defined above have the side-effect of eventually yielding an environment conforming to the set of declarations which comprise the module. Any set of declarations created during the compilation process has this side effect. An environment is a set of names, related during program execution by virtue of their creation from a set of declarations created during the compilation process. In block-structured languages, environments are implicitly created; in module-structured languages, they are explicitly created from the module manipulation constructs and implicitly created from procedure parameter lists, and local declarations within a block. The action of creating an environment from a module can be performed separately and a name associated with the environment so created.

The `bind` construct is used to create an environment from the referenced module, without exposing any of the accessible declarations within the referenced module and without creating any access paths into the module from the context of the `bind` construct. As before, any of the module parameters can be set at the time of the `bind`. Because a data structure is explicitly being created while no mechanism of manipulating that structure is being defined, a name is associated with the resulting structure. If no name for the environment is designated, the name
of the referenced module is used. This has the effect of shielding
the module from further reference by any block nested within the set of
declarations in which the bind appears.

To provide further flexibility in the creation of environment
names, any environment which is defined (other than those formed from
the parameter list of a routine) can be named. The names construct is
used to create the correspondence between a declaration and the environ­
ment which becomes the value of the name, once it is created. The names
construct can be used with the include, the insert and the bind con­
structs. It can appear as part of the set of declarations within a
module, and the name so declared can be tagged accessible. Thus, the
construct

\[
S \text{ names include STACKMOD;}
\]

represents the explicit definition of \( S \) and the implicit definition of
PUSH and POP, which, when referenced, will be evaluated within an en­
vironment, described by STACKMOD, which is the value of \( S \).

The local environment can be named by using the keyword local, as
shown in the construct

\[
S \text{ names local;}
\]

Environment Manipulation Constructs

Environment names may appear within the executable code associated
with any block. They are compile-time directives, used in resolving
identifier references. Within the execution of a statement, the environ­
ment in which a name is to be evaluated can be explicitly designated by
reference to an environment name. In large programs constructed by many people, this provides a mechanism for the explicit designation of the environment in which a name is to be evaluated. This can be used to prevent the unexpected shielding of names, and to make possible the interrogation of a set of names during the debugging process which may not otherwise be accessible. Two operations are defined on environment names: `enter` and `leave`.

Environments may be entered and left anywhere within a single statement. Environments which have been entered, but not yet left, are stacked. The `enter` operation stacks the current environment, and transfers to the one designated by the `enter` operation. The `leave` operation causes transfer to the environment on the top of the environment stack and pops the stack. At the end of statement execution, all environments which have been entered, but not left, will be popped; hence, each statement in a sequence begins execution within the same environment. As an example, if the value of TOP, as declared in STACKMOD, is to be printed and if the declaration, `S names include STACKMOD;` is included within a set of declarations such that

```
print(<enter S> TOP);
```

is included within the scope of `S`, then the value of `TOP` would be printed.

By including the `names` construct, the module-structured language, which has been defined from a block-structured language, allows the compile-time data type environment. This expands the set of declarations which can be used to define a module to include declarations of
the type environment. Therefore, a module parameter may be bound to an entity of type environment.

The syntax for the module manipulation constructs defined is described by the partial BNF grammar of Figure 2.1. In the next chapter, the semantics of these constructs are precisely defined in terms of an implementation model.
Figure 2.1. Partial BNF grammar for the module manipulation constructs
CHAPTER III. THE IMPLEMENTATION MODEL

Overview of the Implementation Model

The implementation model, in which the semantics of the module manipulation constructs are embedded, is described within this chapter. The implementation model is an operational model which has been programmed in PL/I and run on an IBM 360/65. The model consists of three phases, which are run sequentially, to reflect the input of a program, the compilation of a program, and the execution of a program written in a particular module-structured language, ML.

The input phase of the model creates the initial data structure to represent the input program. Input to this phase is in the form of an abstract syntax which can be considered to be the output of a lexical analysis portion of a compiler. The output of this phase is a labeled, rooted, directed graph, in which the set of declarations forming the local variables for a block have been collected into a module-like form. In addition, the formal parameters of a procedure are represented in a module-like form within the graph. The input phase collects together all of the instructions belonging to a block of code.

The second phase performs the module manipulations, forming the complete enumeration of any set of declarations which included one or more module manipulation constructs. During this process, any module referenced, which was defined using any module manipulation construct, is completely defined. Any environment identifiers declared within the input program have their values defined. Once these operations
are complete, the data structure, representing the executable program, is created and input to the third phase.

The third phase of the model is an extension of Johnston's contour model for block-structured languages (9, 10, 11, 12). As such, it interpretively executes the program as output by the second phase, simulating the naming structure of the module-structured language. The data structure input to the third phase is a labeled, rooted, directed graph which consists of two node types. The first, the contour cell, contains a set of declaration subcells which form an environment during execution; the other node type, the code cell, contains a set of instruction subcells, which contain the instructions which form a block of code. This graph is decomposable into a forest of trees. Each procedure which is defined within the global environment is captured by a tree; each module referenced within a procedure (or within a block) is also represented by a tree. This, of course, is caused by the block-structured nature of the underlying language. From any point on the tree, the set of declaration subcells, which can be accessed by being contained within any contour cell which is included on the path from the point on the tree to the root of the tree, forms the set of "vertically visible" names. The forest is transformed into a graph by the inclusion of the links between contour cells which are created by the module manipulation constructs providing "horizontal visibility."

Before proceeding to define the semantics of the module constructs, it is appropriate to define the underlying block-structured language interpreted by the third phase of the model. This base language is
presented in the next section. The three sections following that describe the three phases of the model. The semantics of the module manipulation constructs are defined within the section on the second phase as transformations on the initial graphical representation of the input program. Following the discussion of the model, the semantics and naming structure of the module-structured language, ML, are defined by discussion of an example.

The Base Language, BL

The base language, defined for the third phase of the implementation model, is a simple, imperative, block-structured language. The language was designed to capture the notion of block structure and provide adequate power for the purpose of demonstration but remain small enough for a manageable, rather simple-minded interpreter. BL focuses on those language constructs which, together, provide block structure. The remaining language constructs are included to demonstrate the effect of that block structure. For that reason, BL cannot be considered a viable programming language.

BL provides for the explicit declaration of simple variables and routines. Label constants are declared by their use. No structured variables or constants are provided. Routines may be value-returning. Simple variables are untyped. BL allows for data objects of two types: integer and label. A sufficient set of arithmetic operators is provided for the generation of integer-valued data objects during program execution. Label-valued data objects cannot be generated during program
execution. Both labels and routines are label-valued data objects.

All type checking is dynamic, performed only under the control of the executable statement in which the type of the data object to be manipulated is defined. Hence, the call of a procedure requires a label-valued object as an operand, while assignment performs no type checking.

BL provides eight executable statements, four of which are included to capture the notion of block structure. BL provides assignment statements, output statements, transfer statements, conditional statements of the if-then-else form, as well as procedure call and return, and block entry and exit. Although all conditional statements are treated as if-then-else constructs, the else clause may be elided within the program text. A few more restrictions have been made: (1) all arguments to a procedure are passed by value; (2) only the value of integer-valued simple variables can be output; and (3) the comparison performed as part of the execution of a conditional statement must be between an integer-valued simple variable and either a second integer-valued simple variable or an integer.

A BNF grammar for BL is given in Figure 3.1. The semantics of BL, defined within the third phase of the implementation model, are as expected for a block-structured language derived from ALGOL.
Figure 3.1. BNF grammar for BL
The Input Phase

The input phase of the model creates a graphical representation of a program written in ML, for use by the construction phase in performing the module manipulations. The program as input to this phase of the model is expressed in an abstract form which can be viewed as the output of a lexical analysis. The graph constructed by the input phase is a rooted, labeled, directed graph with all of the arcs having a label. The nodes themselves are not labeled, although some of the nodes contain information.

The graph is constructed from several different types of nodes. The contour node is used in representing a set of declarations, as they appear within the program text. The contour node also plays a part in the representation of a module manipulation construct, since such a construct is considered as part of a set of declarations. A second node type which appears in the graph is the code node. A code node is used in representing a set of instructions. A code node is used, for example, in representing the set of instructions forming the body of a block. A code node designates the contour node used in representing the set of declarations with which the set of instructions is textually related. A third node type is a branching node. This node type is used in building the graph but has no analog within the program text. A fourth node type is a container node. This is the only node which is nonempty and contains different types of objects extracted from the program text, including identifiers, integral constants, and arithmetic expressions.
The contour node will be described first. Within the program text, any set of declarations is decomposable into several disjoint subsets, depending on the type associated with the declaration. For example, the program fragment given:

```plaintext
dcl a;
routine b(c,d);
begin;
end;
include e;
p names bind s;
```

is a set of declarations which can be decomposed into four sets: the set of variables declared (\{a\}), the set of routines declared (\{b\}), the set of `include` statements (\{include e\}), and the set of environment declarations (\{p names bind s\}). To represent a set of declarations, all of this information is placed in a subgraph whose root is the contour node, where a subgraph is defined to be a set of nodes such that any node in the graph has arcs connecting it to only other nodes in the set. In creating the subgraph, the natural decomposition of identifier types is used, since it is of value in the construction phase of the model.
As an example, the set of declarations given by:

```
decl a;
decl b;
routine c;
    begin;
    :
    end;
routine d;
    begin;
    :
    :
    end;
```

would be represented within the graph as:

where the labels on the arcs emanating from the contour node are `vars` (for variables) and `rtns` (for routines) and each of the branching nodes have arcs labeled 1 and 2.

A contour node can be distinguished from any other node in the graph by the labels on the arcs emanating from it. These arcs are labeled from the set `{vars, rtns, mods, incs, names, ins, pars, env}` and are primarily used in decomposing a set of declarations into the sets of
variables declared, routines declared, modules declared, include statements, environment declarations, insert statements, and parameter identifiers, respectively. The arc labeled env will be discussed below. The set of identifiers which are declared as variables are contained in a set of container nodes which are linked to a branching node by arcs emanating from the branching node. These arcs are labeled by positive integers, but the correspondence between the arc label and the identifier is arbitrary. Except for the case of parameters, this arbitrary correspondence exists. Parameters, being positional, require a strictly defined correspondence between arc label and identifier, such that the identifier contained in the node which terminates the arc labeled k is the k\textsuperscript{th} parameter in the parameter list.

All sets of declarations within the program are represented in the initial graph by a subgraph whose root is a contour node. The type of the declaration set determines the nonempty disjoint sets into which the set can be decomposed. For example, if the set of declarations is the set of local variables in a block, both the set of inserted modules and the set of parameters will be empty. If the set of declarations represents the formal parameters of a routine, only the set of parameters may be nonempty; this set is empty in the case of a parameterless procedure. If the set of declarations is a module definition then the set of defined modules will be empty.

A contour node has an arc labeled env emanating from it; in some cases, no other arc emanates from the contour node. This arc connects the contour node with that contour node which is the root of the
subgraph for the set of declarations for the "enclosing block". If the contour node is the root of the subgraph representing the local declarations for a nested block, the arc labeled env terminates in the contour node which is the root of the subgraph for the set of declarations for the statically enclosing block. If a contour node is the root of a subgraph which represents the set of declarations of a module, the arc labeled env emanating from it terminates at the contour node which is the root of the subgraph for the set of declarations in which the module is defined. If a contour node is the root of a subgraph which represents the formal parameters of a routine, the arc labeled env emanating from it terminates at the contour node which is the root of the subgraph for the set of declarations in which the routine is declared. If a contour node is the root of a subgraph which represents the local variables within the body of a routine, the arc labeled env emanating from it terminates at the contour node which is the root of the subgraph for the formal parameters. The relationship of the env-labeled arcs of contour nodes can best be demonstrated with the program fragment below:

```plaintext
dec1 a;
dec1 b;
routine c(d,e);
begin;
);
end;
mod f;
dec1 g;
);
endmod
```

whose subgraph would appear in the initial graph as:
The relationship between a module name and the set of declarations which is its value is evident in the subgraph above. An arc labeled value which emanates from a node containing a module identifier and terminates in a contour node reflects the compile-time binding of a module and its name. The name-value relationship between a routine identifier and its body is also reflected within the initial graph by an arc labeled value. This aspect of the initial graph will be considered later.

In the initial graph, the module manipulation constructs are each represented in tree form, with all of the information contained within the statement appearing in the tree. As an example, an include statement has, basically three pieces of information within it: the name of the module, a (possibly empty), set of arguments and a, (possibly empty) set
of renaming information. An include statement is represented in the initial graph by a tree whose root is a node which has an arc labeled mod emanating from it, and if there are arguments, an arc labeled args and if any renaming is to be performed, an arc labeled withs. In addition, the argument list in an include statement provides implicit information concerning the order of the arguments, which is reflected in the arc labels, and the renaming information implicitly designates both the identifier of a name as it appears in the referenced module, and the identifier of the name to appear in the context of the include statement. As an example, the construct

```
include mod (a,,0) with rl -> r;
```

would be represented within the initial graph by the tree shown below:
All of the renaming information is displayed within the graph as a branching node with arcs terminating in rename nodes, which are empty nodes having arcs labeled old and new emanating from them. Again, the ordering implied by the arc labels emanating from the branching node which is the root of the tree containing the renaming information is arbitrary. The argument information is contained within the initial graph by a tree whose root has a set of integer-labeled arcs which terminate in container nodes. These arcs are labeled by a subset of \([1, 2, \ldots, k \mid k > 0]\) for some integer \(k\).

A contour node has an arc labeled incs emanating from it which terminates in a branching node. By attaching a subtree of the form shown for an include construct to this branching node, the manipulation itself becomes inherent in the graph. So for the module definition fragment given by:

```plaintext
mod q;
  ...
  decl a;
  decl b;
  routine c;
    begin;
      ...
    end;
  include mod(a,0) with rl -> r;
  ...
endmod
```

the subgraph given below would appear within the initial graph.
The `insert` construct consists of the same type of information as the `include` construct, and is represented in the same manner. The only difference is that the tree which represents the construct is attached to the node at which the arc labeled `ins` terminates.

The `names` construct performs two functions simultaneously. It acts as the declaration of an environment identifier and, further, defines how the environment which is to become the value of the environment identifier, is to be created. The `names local` construct is somewhat different and will be treated later. The graph reflects this dual function in the following way: A node which contains the environment identifier has an
arc emanating from it which is labeled with the module manipulation and which terminates in a node which is the root of a subtree. This subtree is the type of subtree used for the include and insert constructs. In the case of a bind, of course, no arc labeled withs exists in the subtree. Thus, the construct

\[
\text{s names bind t(a)}; 
\]

would appear within the graph as a subtree of the form:

Within a subgraph whose root is a contour node, the subtree representing a names construct appears as a subtree with an arc labeled by an integer terminating at its root. This arc emanates from the branching node which terminates the arc labeled names emanating from the contour node. As an example, for the set of declarations given by:

\[
\begin{align*}
\text{decl} \ a; \\
\text{s names insert t;} \\
\text{routine} \ c; \\
\quad \text{begin;} \\
\quad : \\
\quad : \\
\quad \text{end;}
\end{align*}
\]

the subgraph within the initial graph would be given by:
The `bind` construct is defined in terms of the `names` construct. Within the graph, it is represented in its expanded form. Hence the module manipulation

```
bind mod;
```

is reflected in the graph as though it appeared within the program text as

```
mod names bind mod;
```

and the above discussion describes the subgraph which would be used.

The `names local` form of the `names` construct is used to associate an environment identifier with the environment created from the set of declarations in which it appears. To eliminate any possibility of circularity and to reflect the fact that, within the set of declarations, no value can be attached to the identifier in the construct until the set of declarations has been completely enumerated, the identifier is handled as a special form of simple variable and appears in the subtree whose root is the branching node in which the arc labeled `vars` terminates.
To distinguish it from other identifiers in the subtree, the identifier for the local environment is capitalized. For the program fragment given below:

\begin{verbatim}
decl a;
s names local;
decl b;
routine c;
begin;
  ...
end;
\end{verbatim}

the subgraph shown below would appear in the initial graph.

\begin{center}
\begin{tikzpicture}
  \node (vars) {vars} child {node (a) {a} edge from parent node[above] {1}} child {node (S) {S} edge from parent node[below left] {2}} child {node (b) {b} edge from parent node[below right] {3}};
  \node (rtns) {rtns} edge from parent node[below] {3} child {node (c) {c} edge from parent node[above] {1}};
\end{tikzpicture}
\end{center}

Modules are defined by either listing a set of declarations and module manipulations, or by using the \texttt{is} construct. In the later case, the construct is reflected within the initial graph as a subtree. This subtree possesses the same characteristics as that which is used for the \texttt{bind} construct. Hence, the root of the subtree is a node having an arc labeled \texttt{mod} emanating from it, and if there are arguments, has a subtree for the arguments within it. This subtree is linked to a container node by an arc emanating from it labeled \texttt{value}. The module identifier appears within the container node. The subtree rooted by the container node is
attached to the subgraph for the set of declarations in which the module
definition appears within the subgraph whose root is the branching node
which terminates the arc labeled mods emanating from the contour node.
In considering module definitions, some provision must be made to dis­t
inguish those names which have been tagged accessible. For purposes of
illustration, tagged identifiers are underlined within the graph.

A code node is used in the representation of the blocks of code
existing within the program. A code node is a form of branching node,
which has an additional arc labeled decls emanating from it. This arc
terminates at the contour node which is the root of the subgraph for the
set of declarations with which the code is associated. The other arcs
eemanating from a code node are labeled by the set of consecutive positive
integers 1, 2, ..., k for k > 0. Each of the integer-labeled arcs termi­
nates in a container arc, in which an instruction is stored. The node
which terminates the arc labeled n contains the n\textsuperscript{th} instruction in the
block of code. Any node within the graph which contains a routine iden­
tifier has emanating from it an arc labeled value which terminates in a
code node. The instructions within the subtree for which this code node
is the root form the entry point coding for the routine. Interpretation
of these instructions by the third phase of the model causes the parameter
binding and procedure activation. A code node as described above has its
decls-labeled arc terminating in the contour node which is the root of
the subgraph for the formal parameters of the routine. Only two instruc­
tion types are not totally contained within the node in which they appear.
These are the activate instruction, which appears within entry-point
coding for a routine, and the begin instruction. Both of these instructions designate a code node which serves as root for a subtree in which the instructions for the routine body or block appear. Therefore, an arc labeled block emanates from a container node which contains a begin or an activate instruction and terminates in a code node.

During the construction phase, the instructions within the graph are interrogated but not transformed. For this reason, code nodes are pictured as monolithic entities, in which all of the instructions are contained. It is assumed that each instruction can be retrieved using the integer which would be the label of the arc terminating in the node containing the desired instruction.

The root of the initial graph is a contour node. This contour node can be considered the value of an unnamed module whose set of declarations comprises the global environment. The set of declarations for which this contour node is the root of the subgraph represents the global environment. The root has at most two arcs emanating from it. One is labeled rts. The other, if it exists, is labeled mods. This is, to some extent, a reflection of the definition of BL, which defines a program as a set of routines. In other module-structured languages, the global environment might also contain declarations for simple variables. If so, the root of the initial graph could then have an arc labeled vars emanating from it.

The graph which is generated by the input phase for a given program according to the descriptions above is input to the construction phase, which performs the module manipulations as defined within the graph,
transforming the module-structured program into a form which can be executed in the third phase. This construction process defines the compile-time actions used in creating the sets of declarations using the module manipulation constructs. Parts of the compilation process require complete symbol tables; thus, the completion of the module manipulations precedes that portion of the compilation process which has been abstracted to the execution phase.

The Construction Phase

The construction phase of the model performs the module manipulations by performing a series of transformations on the initial graph. These transformations create the environments which might arise during program execution and build the access paths between a set of declarations in which a module is referenced and the set of declarations forming the referenced module. The correlation between an environment identifier and the environment which is to become its value is established during the construction phase. Once the set of transformation has been performed, the resulting graph is used to create the program skeleton, which is input to the third phase, as a description of the program to be executed.

The process of constructing the program to be executed yields a sequence of graphs, \( G_0, G_1, ..., G_n \), where \( G_0 \) is the initial graph and \( G_n \), the final graph, is used to create the program skeleton. This process yields a unique \( G_n \) from a given \( G_0 \). The sequence of graphs which yield \( G_n \) may vary, within certain constraints, depending on the implementation.
chosen for the algorithms describing the transformations. Each module manipulation consists of three parts: (1) module resolution which determines which module was referenced; (2) creation of a copy of the definition of the referenced module, which requires performing all module manipulations within the module; and (3) performance of the module manipulation, which transforms the subgraph in which the construct originally appeared by replacing the subtree describing the construct with the copy of the referenced module which has been created. Only those subgraphs which have a contour node as a root are transformed; the subgraphs which have code nodes as roots are traversed to locate contour nodes, as required.

Before describing the transformations on the graph which define the semantics of the module manipulations, a few definitions are in order. Within the graph, all of the arcs emanating from a node are labeled with distinct labels. Any arc in the graph is uniquely determined by two nodes and, because of the unique labels of the arcs emanating from a node, from a given node an arc unambiguously "selects" another node. As an example, if a contour node is being considered, then from that contour node, the arc labeled \texttt{vars} selects a particular branching node within the graph. If \( n \) stands for a contour node, then \( \text{vars}(n) \) is the node selected by the arc labeled \texttt{vars} emanating from the node \( n \). The operation of composite selection is the process of repeatedly selecting a node by using an arc label. In the subgraph for the set of declarations given by:
if \( n \) designates the contour node which is the root of the subgraph representing this set of declarations, then \( l(\text{rtns}(n)) \) is used as a "selector function" to select the node containing the identifier \( c \). Any node which is selected by an arc emanating from a node is "directly accessible" from the node from which the arc emanates. Thus, the code node which is the root of the subgraph representing the entry-point coding for a routine is directly accessible from the container node in which the routine identifier appears. The notion of accessibility is the transitive closure of the notion of direct accessibility. A node \( n \) is "accessible" from a node \( m \) if there exists a sequence of nodes \( n_1, n_2, \ldots, n_k \) such that

- \( n_1 \) is directly accessible from \( m \)
- \( n_2 \) is directly accessible from \( n_1 \)
  
  \[
  \vdots
  \]
- \( n_k \) is directly accessible from \( n_{k-1} \)
- \( n \) is directly accessible from \( n_k \)

Another way of describing the notions of "direct accessibility" and "accessibility" is to cast the definition in a more active frame. If node \( n_1 \) is directly accessible from node \( n \), then node \( n \) "directly accesses" node \( n_1 \), using the label of the arc which connects them, and if node \( m \) is accessible from node \( n \), then \( n \) "accesses" node \( m \).

The description of the construction process is made in terms of these definitions.
Module resolution

The process of module resolution is one of locating the definition of the module referenced within a module manipulation construct. If more than one module having the same name exists, the reference is to be resolved to the module defined in the smallest block enclosing the one in which the reference is made. This is a form of static binding, which is usually used in block-structured languages. To locate the module to which a module reference is to be resolved, a search is made within specific subgraphs of the graph for the module definition. These subgraphs are defined by repeatedly using the arcs labeled \texttt{env} to select a sequence of contour nodes. For purposes of discussion, let \( m \) be the contour node which is the root of the subgraph containing the module reference. Starting at \( m \), the contour nodes selected in this way can be used to define all of the declarations visible from a given contour node in much the same way that the set of declarations visible from a given block in a program written in an ALGOL-like program can be determined. From any contour node, the subgraph whose root is the branching node selected by \texttt{mods} determines the set of modules defined within that block. From any contour node, then, the set of module names which are visible can be determined by repeated use of the selectors \texttt{mods} and \texttt{env}. The module within a manipulation construct is resolved to the module of the same name defined within the smallest enclosing block in which such a module definition occurs. Thus, in the program fragment given below:
the reference to q within routine r would be resolved to the module q containing the declaration for b, while the reference to q within routine s would be resolved to the module q containing the declaration of a.

Performing a transformation

After resolving the module referenced within a construct, the transformation which defines that construct can be made, provided the module to which the reference was resolved is fully defined, i.e., that the module definition contains no reference to another module. In a legal program, any module definition can be transformed into a fully-defined module. In describing the transformations on the graph, a fully-defined
module is assumed. Transforming a module into a fully-defined module makes use of the transformations presented below.

The procedures defining the transformations are given in the Appendix. In presenting the transformations, the procedures will not be explicitly referenced. For purposes of discussion, it is assumed that the module reference within the construct has been resolved to a fully-defined module and that a copy of that module has been made. After describing the transformations as isolated occurrences, the traversal strategy will be described. The traversal strategy orders the transformations on the graph, determining which to perform next.

The *include* construct is used to make the accessible declarations of the referenced module known in the context of the construct. At the same time, when any of these names are referenced within the code, they must be evaluated within an environment which is created from the description provided by the module in which they were declared. To describe the transformation, suppose $c$ is a contour node, that $x$ is the root of a subtree which describes the construct

```
include mod;
```

and that, for some integer $k$, $k(\text{incs}(c))$ selects $x$. Suppose further, that $v$ is the node selected by $\text{vars}(c)$ and that the node selected by $\text{n}(\text{vars}(c))$ exists but the node selected by $\text{n}+1(\text{vars}(c))$ does not exist. Assume also that mod has $m$ accessible declarations.

The transformation satisfies the first half of the definition in the following way: For each accessible declaration in the referenced module, create a container node for each accessible declaration, placing
in the container node the identifier from the declaration. In this ex-
ample, m container nodes would be created. When a container node is
created, and the identifier placed in it, the container node is linked
to the one in the copy of the module from which it was created by con-
structing an arc from the new container node. This describes the cor-
respondence between a name made known in the context of a module refer-
ence and the name within the module. Each newly created container node
is attached to the subtree whose root is selected by the arc labeled
**vars** from some contour node. In the example, the subtree for which v
is the root would now have n + m nodes directly accessible from v. The
final part of the transformation replaces the subtree for the construct
with the copy of the referenced module.

This description of the transformation has not addressed the issues
of argument list and renaming. Renaming affects the set of names which
are appended to the subgraph whose root is a contour node and in which
the root of the subtree is selected by \( k(\text{incs}(c)) \) for some contour node,
c, and some integer \( k > 0 \). The effect on the above transformation is
one of changing the identifier placed in the newly-created container node.
Figure 3.2 pictorially displays the include transformation, with some
renaming. In Figure 3.2a, a subgraph whose root is a contour node is
given. This subgraph depicts the set of declarations shown below:

```plaintext
decl a;
decl b;
include ex with b→ d;
routine c;
   begin;
   ::
   end;
```
where the hexagon represents the subgraph used to represent the value of c. If the module \( \text{ex} \) has been defined as:

```
mod ex;
  decl a;
  accessible decl b;
  accessible routine f;
  begin;
  ;
  end;
endmod
```

then Figure 3.2b is the graphical representative of \( \text{ex} \). The result of the transformation on the subgraph of Figure 3.2a is shown in Figure 3.2c. The reader will note that the complete subgraph for the module is copied into the subgraph in which the `include` construct originally appeared. Once the transformation is complete, the copied subgraph provides a description of an environment to which access has been defined.

The question of an argument list within the `include` construct still remains. In using an argument list, a new module is implicitly being created. This module has fewer parameters than the one named in the construct. After the process of resolving the module reference, in creating a copy of that module, the argument list is used so that the copy, which will eventually be linked into the subgraph in place of the subtree for the construct, has the parameters set as indicated in the construct. This process can be thought of as creating an unnamed module using the `is` construct. For this reason, an argument list has no effect on the `include` transformation.

In Chapter II, the `insert` manipulation was defined as having the same effect as the `include` except that those names which become known
Figure 3.2. An example of the **include** transformation
in the context of the construct are tagged accessible, so the transforma-
tion for the insert manipulation is basically the same as that for the
include manipulation.

The bind construct is represented within the graph in tandem with
the names construct. A bind causes the same change to a set of declara-
tions as would an include on a module having no accessible declarations.
That is, a copy of the module referenced, subject to any changes caused
by an argument list within the construct, would replace the subtree
defining the construct and have no further affect on the subgraph in
which the construct originally appeared.

The names construct is used to associate an environment identifier
with an environment created by a module reference. In s names bind t;, for example, an environment described by module t is to be created and s
is to identify that environment. In s names include t;, an environment
described by module t is to be created; s is to identify that environment;
and horizontal visibility from the environment in which s appears to the
environment created from t is to be provided in accordance with the def-
inition of the include construct. As an example, if s names include t;
were included within a subgraph whose root is a contour node, the trans-
formation is carried out as described above. The graphical representa-
tion for the include t part of the construct is accessed from the contour
node using a composite selector of the form inc(k(names(contour node))),
for some integer k > 0. The copy of the module which has been used
in the include transformation replaces the graphical representation
of the include t; so that the contour node which is the root of this
copy is accessible from the contour node by \texttt{inc(k(names(contour node)))} for the same \texttt{k}.

Basically, then, the transformation for a \texttt{names} construct consists of deciding which of the transformations for \texttt{include}, \texttt{insert}, and \texttt{bind} to use. This is determined by the label on the arc emanating from the container node in which the environment identifier is stored.

The last construct to be considered is the \texttt{is} construct. This construct is used to create a module whose definition is determined by the definition of some other module. If no arguments exist in the statement, that is, the manipulation statement is of the form

\begin{verbatim}
m is n;
\end{verbatim}

then a copy of the subgraph which is accessed from the node containing \texttt{n} by the arc labeled \texttt{value} is made and this copy replaces the subtree for the \texttt{is} construct within the graph. If an argument list does exist in the manipulation construct, then a new module having fewer parameters must be created. To begin with, any parameter whose value has been set can no longer be accessed from the root of the subgraph by a selector of the form \texttt{k(pars(root))}. Rather, it is to be accessed by a selector of the form \texttt{k(vars(root))}, since that seems the most reasonable of the subsets in which to place the identifier. To accommodate the value which must be associated with the identifier in creating the name, when the environment is constructed, the node containing the identifier directly accesses a node containing the value by an arc labeled \texttt{value}. After all of the parameters for which values have been provided in the argument list have been moved to the subset of variables, the subset of remaining parameters
must be reordered to enforce the correspondence between the label on the arc which terminates at the node containing the identifier for the parameter and the position of that parameter in the new parameter list. Once this has been done the graph representing the new module is complete and can be used in whatever fashion desired. In the case of the *is* construct this new graph replaces the graphical representation of the *is* construct. If this creation is performed as a secondary issue in an include (or insert, or bind) transformation, then the required copy has been created for use.

As an example, suppose the module *mod*, having three parameters is defined as shown below:

```
mod mod (x,y,z);
  decl a;
  routine b;
    begin;
    :
    :
    end;
endmod
```

and that new *is* *mod (1,,3);* is to be performed. Then the effect of the *is* construct is to define a module *new* which names the set of declarations given by:

```
mod new (y);
  decl x initial 1;
  decl z initial 3;
  decl a;
  routine b;
    begin;
    :
    :
    end;
endmod
```

where *initial* has been used to describe the initialization of the values
of \( x \) and \( z \).

The argument evaluation aspect of this process needs clarification. The arguments to a module are compile-time constants. They may be integers, as shown above, or identifiers which have values associated with them during compilation. These identifiers include routines and environment identifiers, as well as simple variables which have been initialized, as above. They do not include modules, since modules cannot be immediately nested within modules. Module arguments may also be arithmetic expressions, in which case only previously initialized module parameters can be referenced. In order to evaluate the argument, all identifier references must be resolved. The resolution strategy follows that described for module resolution. Starting with the set of declarations in which the identifier reference was made, the identifiers are searched for an instance of the referenced identifier, and if it has a value, that value is used. If it has no value, then the program is illegal. If the identifier cannot be found then the search is continued by using the set of declarations for the enclosing block of the one in which the argument appeared. This process of identifier resolution will be expanded upon in the discussion, below, on the strategy used in determining the order of applying the transformations. Once all of the arguments have been evaluated using this static resolution, the new module is created as described above.

As an example, suppose a module, \( m_1 \), is defined by

```
mod ml(x,y);
  decl a;
  decl b;
endmod
```
and that a routine, r1, is also defined, as outlined below:

```
routine r1;
  begin;
    decl a;
    routine r2;
    begin;
      m is ml(2,r3);
    end;
  end;
routine r3;
  begin;
  end;
end;
```

The creation of the module m requires the evaluation of the argument r3. First, the set of declarations which are local to the body of r2 is searched for the identifier r3. Since none is found, the set of declarations for the enclosing block is searched. This set, \{a,r2,r3\} contains the identifier r3, which has a value. The module m will then contain an identifier y whose value is the value of r3.

**Creating a fully-defined module**

To this point, all modules referenced have been assumed to be fully-defined. This is not always the case. The creation of a fully-defined module requires the performance of all of the module manipulations that appear within the module, either in the set of declarations which define the module, or within a routine defined within the module. The order in which the required transformations is defined by the "traversal strategy". The process itself is highly recursive and will be discussed in somewhat
general terms. The reader is referred to the Appendix, in which the proce-
dures for the entire construction process are given.

Once a module reference has been resolved, the module definition
must be completed, if the module is not fully defined. The process of
fully-defining a module is one of completing a set of declarations. The
same method is, basically, applied to any set of declarations within the
graph and will be described with that in mind. Some slight modification
is made to the process to provide error checking. These modifications will
not be discussed.

For any set of declarations, any modules defined within that set
are first interrogated. This is done to facilitate module resolution.
Any module defined by recourse to the _is construct is defined by creat-
ing a copy of the module referenced, with any argument manipulations re-
quired. The module created is any exact copy of the referenced module
in that no internal module manipulations are performed, since module
references have been defined to be statically resolved from the point of
definition. Any module in this set which is defined by a partial enumera-
tion of the set of declarations is not modified. It is possible that a
module, defined by use of the _is construct, is to be passed an argument
which will exist within the context of the _is construct following some
further module manipulation. In first attempting to create the module,
arguments are resolved within the local context only. If this cannot be
done, the module definition is deferred. Once all of the required module
definitions have been attempted, then the _include constructs are tried.
Once the module reference has been resolved, if the referenced module is
still defined as the result of an *is* construct, the attempt is "aborted," and the next *include* construct is tried. If the module referenced is defined in terms of a set of declarations, then that set of declarations is fully defined. Only in the case where the module referenced is defined within the context of the *include* construct can the process of full definition be legitimately aborted. If the referenced module is defined in some context other than the one whose subgraph is being traversed, and that module cannot be fully defined, the program is illegal. Assuming that the referenced module has been fully defined, the arguments, if any, are evaluated within the local context. If that can be done, the include transformation is applied. If not, the attempt to include is legitimately aborted. The set of insert constructs is next interrogated in this same fashion. Following that, the *names* constructs are tried.

Having made one pass through all of the constructs within the subgraph, a second pass is made if any attempt was aborted. The process of repeatedly attempting to transform each of the module constructs continues until either all of the constructs have been transformed, or some subset of them cannot be performed. One additional pass is made through the untransformed constructs, which are now known to hinge on argument evaluation. Beginning with any as yet undefined modules, any arguments are evaluated in the complete environment, as opposed to the local context only. Any module which cannot now be fully defined signals the illegality of the original program. Having now defined all of the modules which can be referenced within the set of *include*, *insert*, and *names* statements, the as yet untransformed *include*, *insert* and *names* representation are again
interrogated, with the local context restriction on argument evaluation. If, again, a set of legitimately untransformed construct representations exist after a number of passes through them (an additional pass is made only if some transformation was performed), the restriction on argument evaluation is lifted and another set of passes is made. In this case, either all of the transformations are performed or, if some subset still cannot be performed, then the program is illegal.

The complexity of the traversal described above is a function of the possibly great interaction between constructs within one set of declarations. The program fragment in Figure 3.3 demonstrates such interaction. Suppose that the modules q and r are defined as shown within some block enclosing that in which the three module manipulations appear. According to the traversal strategy, s is defined first. The argument,

```plaintext
mod q(a,b);
decl c;
accessible routine d;
begin;
::
end;
endmod
mod r(c,d);
accessible decl e;
endmod
::
include q(,e);
s is r(0);
v names include s;
::
```

Figure 3.3. An example of interaction between module manipulation constructs
consisting of a constant, allows this to occur. The include construct is then attempted. No declaration for e exists locally, although after the completion of the transformation for the names construct, one will exist. Since the definition of the constructs imposes no hierarchy on the constructs, the e which will become known locally is the e which should be passed in the argument list for q. Hence, the include transformation is legitimately aborted. In this case, the names transformation is tried next. The module s is defined locally. Since the definition was completed, the transformation can be performed, making e known within the local context. In the second pass, then, the transformation for the statement include q(e); can be performed.

Once all of the transformations for module definitions, include statements, insert statements, and names statements have been performed the local context has been transformed into its final state. Those sets of declarations for which the current one is the enclosing set must still be transformed. Each of the routines defined with the set are next transformed. The definition of a routine can be accessed from a contour node by using a composite selector of the form value(k(rtns(contour node))), for some k > 0. The node so selected is a code node, from which the contour node which is the root of the subgraph for the formal parameters is directly accessible. It is obvious that the set of declarations represented by such a contour node requires no transformation. Instead, the subgraph representing the local variables for the routine body does need interrogation. From the code node representing the entry-point coding for the routine, the root of the subgraph for the local variables
can be accessed by decls(block(n(code node))). This contour node is then traversed, using the strategy being described. Once that traversal is complete, any nested blocks within the body of the routine need to be located. This is accomplished by interrogating the code for the body of the routine and, on encountering a begin instruction, traversing the contour node accessible by decls(block(begin instruction)). Once the traversal is complete, the remainder of the instructions in the body of the routine are interrogated. When all of the instructions of the routine have been interrogated, the traversal of the routine is complete; any other routine which is defined in the same context as the routine whose traversal has just been completed is next traversed. Once all of the routines defined within a set of declarations have been traversed, the traversal for that set of declarations is complete.

The traversal begins with the contour node which is the root of the graph. After the traversal for each of the global routines has been completed, C_\text{n} exists. C_\text{n} is used to create the program skeleton for input to the third phase of the model. To do this the routine definitions are traversed again. At this time, each contour node accessed causes the creation of a contour cell, and each code node accessed causes the creation of a code cell. A contour cell contains a set of declaration subcells, an environment link, and some additional information which is not important at this stage. A declaration subcell contains a name, consisting of an identifier, a value, if any, and the type of the value, if one exists. The set of declarations is used to determine the number of declaration subcells within the contour cell; each parameter, each
variable, each routine and each environment identifier requires a declaration subcell. Creating the name from a parameter or from a simple variable is relatively straightforward, although any simple variable which exists by virtue of a module manipulation or any simple variable which names the local environment requires special handling. Initialized simple variables are handled by placing the value within the declaration subcell. The definition of module parameters which have been bound to nonintegral values is deferred. After the simple values and parameters have been formed in the contour cells, the contour cells which are needed because of environment descriptions contained within the set of declarations being interrogated are formed using the technique being described. After this, the linkages from those simple variables which exist because of the module manipulations are built, linking such a name to the declaration subcell within the contour cell from which it was created. This linkage is not a value, but a reference to a declaration subcell. The environment identifiers are then added to the contour cell and their values established by creating a value reference to the appropriate contour cell. Lastly, the routines are defined. The value of a routine is a label, the ip of which designates the code cell containing the entry point coding. Definition of a routine requires both a traversal of the definition as it appears in the graph, as well as the construction of the label. The code node containing the entry point coding is used to create a code cell to contain the entry point code. A code cell consists of a set of instruction subcells and an environment link designating the contour cell with which the code is associated. In creating a code cell,
each of the instructions contained in nodes accessible from the code node by integral selectors are copied into the instruction subcells, in order. The code cell for the entry point coding is linked to the contour in which the routine identifier is defined. Then the contour cell for the formal parameters is created; its environment link designates the contour cell in which the routine identifier is defined. Next the body of the routine is interrogated. The set of declarations describing the local variables for the body of routine is traversed, creating any required contour cells. The subsidiary contour cells are all given null environment links, indicating that they are distinct from the program text. The contour cell for the routine body has its environment link set to designate the contour cell formed for the formal parameters. At this point, block entry-point coding is constructed, within a code cell. Block entry-point coding is used by the third phase of the model in locating the contour cells needed by a block. The activate instruction in the code cell containing the routine entry-point coding is set to designate this block entry-point coding. Within the block entry-point coding, an instruction is created which designates the code cell of the body of the block. The code cell of the block is built and the link between the block entry-point coding and this code cell is created. The contour cell designated by the block entry-point coding is the contour in which it can be activated. In the case of the block entry-point coding for the body of a routine, this is the contour cell for the formal parameters. After creating the code cell for the body of the routine, the instruction subcells are interrogated for begin statements. Each of these is used to
create a set of contour cells and code cells as outlined here. When the traversal of a routine definition is completed, the label which is the value of the routine name is constructed and stored in the declaration subcell containing the routine identifier. Once all of the routine definitions have been created, the definition of a contour cell is complete.

After the traversal of the graph is complete, and the program skeleton basically formed, the graph is traversed yet another time. This traversal, together with a parallel traversal of the program skeleton, accomplishes the task of linking module parameters which have been set to routines or environments. It is the author's opinion that this last traversal could be eliminated if a slightly different second traversal were used.

An example of the construction process

The effect of the first two phases of the model can be best illustrated by example. This section demonstrates the construction process in terms of a simple example, which appears in Figure 3.4. This program contains one module, quad, which is used to define a quadratic function. In using the module, the parameters must be set to the values of the coefficients of the function. In this case, the module contains only one routine, which evaluates the specific quadratic function, defined by setting the module parameters, for a given value, x. This restriction is placed on the module for purposes of illustration; with a "real" module, other functions might be provided.
The routine main makes use of the module on two separate occasions, defining two quadratic functions. Again, for purposes of demonstration, one of the references to the module assigns an environment name to the environment created by reference to the module. The output of the input phase is shown in Figure 3.5. The numbers next to a node are used in positioning the results of the transformations. As mentioned earlier, the code nodes within the initial graph have been collapsed into monolithic entities rather than pictured as subgraphs. Further, to simplify the graph, only those arcs whose labels must be known are labeled. Thus, sets of nodes accessible from a given node are ordered only when the
Figure 3.5. The initial graph for the program in Figure 3.4
ordering provides significant information. In referring to sets of parameters, for example, this information is significant; when referring to a set of routines declared within a block, this information is not significant and has been omitted.

In executing the construction phase of the model, each contour node is interrogated and, if any reference is made to a module, a transformation is applied. In a similar fashion, each code node is interrogated, in order to locate all of the contour nodes. In traversing the initial graph, the root is the contour node first interrogated. Each module name defined within this contour, that is, each module identifier which is accessible from the root node, r, by a selector of the form k(mods(r)), for some positive integer k, is checked to determine whether it has been defined using the is construct. Any module defined in this manner must be created as described above. Once this has been done, every module name which can be referenced in a module manipulation using one of the labels {incs, ins, names} emanating from the contour node being interrogated, has been defined and is represented in the graph by a subgraph whose root is a contour node. When interrogating the root of the initial graph, it is known that at most two arcs emanate from the node, and that one is labeled rtns. After interrogating the module definitions, the traversal of the routine definitions proceeds. A routine definition is traversed by locating each code node accessible from the node containing the routine identifier. The first code node encountered is the one for the entry-point coding of the routine. The contour node associated with the entry-point coding defines the formal
parameters of the routine, and need not be further considered at this time. The last instruction within the code node designates the code node containing the instructions representing the body of the routine. This code node can be used to select the contour node for the local variables of the routine and must be interrogated. After determining that no local module definitions are made in this example, the module manipulations are performed. In this case, an include construct exists. Let the contour node being interrogated be called n. The effect of the manipulation on the subgraph whose root is n consists of replacing the include construct with a copy of the module formed by setting the parameters of the module quad, and by augmenting the set of nodes directly accessible from vars(n) with a set of identifier-containing nodes, one for each declaration tagged accessible in the referenced module, and building the links from the new container nodes back to the identifier-containing nodes from which they were generated. Figure 3.6 demonstrates the effect of the transformation include on the contour node representing the local variables of the routine main. The subgraph in Figure 3.6 replaces that whose root is "labeled" by 1 in the graph in Figure 3.5. The entire module, including the subgraph whose root is the node containing the identifier for eval, appears locally, since the function eval must be evaluated within an environment in which the name a has a value of 2 (at least initially). No other module manipulations exist within the subgraph whose root is n, the contour node being interrogated.

At this point, the code node for the routine body is interrogated, to locate any begin instructions. As is apparent from the program text,
Figure 3.6. The subgraph for the local variables of routine main, after transformation

in Figure 3.4, one such nested block exists. Once the code node for this block has been located, the contour node associated with it is interrogated. Again, no module definitions are made, and no use is made of the include construct. In this case, however, a names construct is used. Performing the module manipulation causes the replacement of the subgraph describing the module manipulation to be used in creating the environment, with a direct description of the environment, together with the performance of the environment-creating manipulation. In this instance, a module is created from the definition of quad in which the parameters are set to 1, 0, and -1, respectively. Resolution of the referenced module name is accomplished by traversing the chain of contour
nodes selected by repeated use of the env-labeled arcs, locating the module quad defined with the global environment. Figure 3.7 depicts the contour node for the nested begin block following application of the names transformation. Since no further module manipulation constructs exist within the subgraph whose root is the contour node being interrogated the code node associated with it is interrogated for instances of begin instructions. Since none exist, the interrogation of the code node for the enclosing block is continued from the point of the begin instruction. No other nested blocks were defined, so the traversal of the definition of routine main is complete. Returning to the contour node which is the root of the subgraph in which the routine definition appears

Figure 3.7. The subgraph for the nested block, after transformation
(in this case the root of the graph) an attempt is made to traverse any other routines defined. Since none exist, the traversal of the graph is complete; all required transformations have been made.

The construction phase of the model completes its task by traversing the graph to construct the program skeleton for the execution phase. The contour cells comprising the program skeleton are shown in Figure 3.8. Each box represents a contour cell. The arrows linking one box to another form the chain of vertically visible environments induced

Figure 3.8. The program skeleton for the program in Figure 3.4.
by block structuring. Those contours containing the identifier RAId are formal parameter contours. RAId is used during the execution phase as the identifier whose value is the return address to be used when exiting the routine. This identifier is generated during building of the program skeleton as a "compiler generated identifier" and placed in the formal parameter contour, along with the parameter identifiers declared as part of the routine header. The identifiers main and eval have values associated with them representing the code for the routines; f and g also have values linking them to the appropriate contour cells. These have been eliminated for clarity. The value of p has been included, designating the environment formed from the module quad which is its value.

In the next section, the execution phase of the model is described. The example above will be used in demonstrating it.

The Execution Phase

Overview of the execution phase

In this section, the third phase of the implementation model, which simulates program execution, is presented. This part of the model is heavily based on the contour model of block-structured languages (9, 10, 11, 12). The semantics of BL are specified by the execution phase, and present no surprises; the semantics will not be explicitly defined here. The effect of the module manipulation constructs on program execution, and the effect of the environment manipulation constructs (enter and leave) will be described in considerable detail.

The execution phase receives as input an abstract representation of
the input program in terms of contour cells and code cells. This representation is considered to be invariant during program execution and is referred to as the program skeleton. Simulation of the instructions of the program creates the record skeleton, which is a dynamic record of execution. The sequence of record skeletons, produced during program simulation, is a time-dependent model of the execution of the naming structure of ML. Because the program skeleton is invariant, the record skeleton consists only of a set of contour cells, each of which is created during simulation in response to a particular instruction within some code cell within the program skeleton. Each contour cell in the record skeleton has an antecedent contour cell within the program skeleton from which it is created. The declaration subcells within the record skeleton most closely capture the notion of name, as these names arise during program execution. Each record skeleton which arises during program simulation will be considered to be a snapshot of the execution taken immediately after the execution of an instruction as it appears within the program text.

Before proceeding to describe the effect on the record skeleton of the simulation of the program given in Figure 3.4, and whose program skeleton is outlined in Figure 3.8, a short discussion of the execution of the model will be given. The reader who is familiar with Johnston's contour model (9, 10, 11, 12) will find only those sections dealing with the execution affects of the module constructs and the enter and leave commands of interest; the effect of these constructs is illustrated by the example of this chapter and that of chapter 4. Other readers will find the next section of value in understanding the execution phase.
The simulator

The program simulator is founded on the notion that the naming structure of a language is the primary determinator of the effect of the execution of any program written in that language. The naming structure, which determines the correspondence between a reference to a name and the declaration of that name, is perhaps the most fundamental aspect of the compilation process and, as such, cannot be readily investigated. By abstracting the execution of the naming structure from the compilation process, the effect of the naming structure on program execution can be studied. In this model, determining the set of names which can be legitimately referenced within an instruction, as well as determining the corresponding declaration instance of a name for a particular reference is of paramount importance.

The state of the model at any time is described by the record skeleton as mentioned above. In addition, the model contains a processor. The processor contains a plethora of information, including an (ip, ep) pair, the ip of which designates the instruction, within a code cell, which is to be the next instruction simulated, and the ep of which designates the contour cell in the record skeleton in which the processor is executing. The processor also contains a work space, used in linking new contour cells into the record skeleton, and the processor also is used in transferring the value returned by a value-returning function from that function to the context of the call.

The ep stored within the processor is used to define the processor environment, which consists of a subset of the set of contour cells within
the record skeleton. This subset is determined as follows: The contour cell designated by the processor's ep is included in the processor environment, and the contour cell designated by the environment link of any contour cell in the processor environment is also in the processor environment. The processor environment is, then, a chain of contour cells within the record skeleton uniquely determined by the processor's ep.

The naming structure of a language is used in determining the correspondence between a reference to a name and the declaration of that name. In this model, that correspondence is determined by the identifier binding strategy, IBS. The IBS searches the processor environment for a declaration subcell containing a name whose identifier matches that of the reference, according to some search strategy. Once the declaration subcell has been found, the correspondence is established. In the discussions which follow, the first declaration subcell encountered in a static search of the processor environment will be the one chosen. One other aspect of the naming structure needs consideration. When a contour cell is created in the record skeleton, it must have its environment link set. The contour cell to which a newly allocated contour cell is to be linked is determined by the environment binding strategy, EBS. The EBS is also used to determine the value of the processor's ep as a side effect of any instruction simulation which can cause a change in the site of execution.

Contour cells within the record skeleton are created on block entry and procedure call. At the time of block entry, multiple contour cells may be created, if any module manipulation appears as part of the set of
declarations of the block to be entered. The creation of multiple contour cells, in response to a single instruction, is one of the major extensions to the contour model contained within the execution phase. At the time of procedure call, only one contour cell is created within the record skeleton. This contour cell represents the formal parameters (and return address) for the procedure. After the procedure call instruction has been executed, the local environment in which the processor is executing, defined by the contour cell designated by the processor's ep, is changed to the contour cell representing the local variables for the procedure body, which is itself a block. Thus, the multitude of contour cells which can be created during the entire process of calling and activating a procedure are entered into the record skeleton by execution of two instructions one of which is a begin instruction.

In the discussion of the instructions of BL which are recognized by the third phase of the model, below, the discussions concerning block entry, procedure call, and the binding strategies are those which focus on the extensions to block structuring caused by the module manipulation constructs.

The assignment statement

BL allows four types of assignment statements: assignment of a constant, assignment of the value of a name, assignment of the value of an expression, and assignment of a value returned by a function. These
actions have been condensed into one algorithm by virtue of their commonality in the handling of the target of the assignment. Simulation of an assignment statement proceeds in a left-to-right fashion. The first action is to locate in the processor environment the declaration subcell of the target of the assignment, using the IBS. Once found (if not found, execution of the program fails), the source of the assignment statement is evaluated. If the right hand side is a constant, it is treated like an immediate argument. For the sake of discussion, assume that this constant is placed in a temporary location, TEMP. If the right-hand side is a variable name, the declaration subcell of that name is located using the IBS. If the name has no value, execution fails. Otherwise, the value of the name is placed in TEMP. If the right-hand side of the assignment statement is an expression, then each operand is evaluated using the IBS as above. Any name having either no value or whose value is not an integer causes execution to fail. Once all of the operands have been evaluated, the arithmetic expression is evaluated, and the result placed in TEMP. The result of a value-returning function is placed within the processor, as described below during the discussion of the procedure-return statement. This value is retrieved from the processor and placed in TEMP.

Once the value of TEMP has been set, the value field of the declaration subcell of the target is updated to contain the value in TEMP. No restriction has been placed on the type of the value in TEMP, beyond the fact that it exists. The action of updating completes the assignment statement simulation.
The output statement

By the definition of the syntax of BL, only the value of a variable may be output. For this reason, the algorithm for output relies heavily on the IBS and has the additional restriction that only integral-valued variables may have their values output. The simulation of an output statement causes no change to the processor environment. The declaration subcell of the variable whose value is to be output is first located. If no such subcell can be found, execution fails. The type of the value is next interrogated. If no value exists, or if the value is not an integer, execution fails; otherwise, the value is printed on the output device (in this case, the line printer).

The conditional statement

BL provides for conditionals of both the if-then and if-then-else forms, although within the code cells, only the if-then-else is provided. This implies that any instruction input as if-then has a null else clause constructed during parsing. Further, two forms of comparison may be performed. Either the value of a variable is to be compared with a constant (integral) value, or it is to be compared with the value of a variable. The type of comparison to be performed, e.g., equality, is contained within the instruction and plays little part here. As is true of assignment, the conditional statement proceeds in a left-to-right manner. For purposes of discussion, let cl and c2 be temporary locations, which will be used to hold the values of lc, the left comparitor, and rc, the right comparitor.

The simulation of the instruction proceeds by using IBS to locate
the declaration subcell for lc. Again, execution fails if no such subcell can be found. If the declaration subcell contains an integer as value, then the value is retrieved and stored in cl; otherwise execution fails. If the value of lc is to be compared to a constant, that constant is retrieved from the instruction and placed in c2; otherwise, the declaration subcell for rc is located using IBS, and if it contains an integer as value, this value is placed in c2. The same failure conditions as for lc hold. Once both cl and c2 have been set, the comparison is performed, creating the boolean value which determines the next ip setting within the processor. The ip value to be used when executing the else clause is contained within the instruction. The conditional statement has no effect on the processor environment.

The branch statement

The branch instruction requires the retrieval of the label value of a variable. If the IBS cannot return a declaration subcell for the variable referenced, or if that variable does not have a label value, execution fails. A label value consists of an (ip, ep) pair; the ep of which, together with the ep within the processor, is used by the EBS to locate the contour which is to become the site of activity. The implementation model allows any EBS to be defined; the EBS generally used within a block-structured language is the static strategy, in which case the EBS will return the ep from the label value at all times. The branch instruction causes the (ip, ep) pair of the processor to be replaced by that of the label value. No other change is made to the state of execution.
The block entry statement

The simulation of the block entry statement represents one of the major extensions to the contour model (9, 10, 11, 12). The contour model assumes that at most one contour cell may be added to the record skeleton as part of the simulation of an instruction. The module manipulation constructs destroy this correspondence for block entry only. At the time of block entry, a contour cell for the local variables of the block must be created within the record skeleton; this block is linked to the contour cell designated by the processor's ep, which designates the environment in which the block entry statement is executed. At the same time, all of the environments created by module manipulations within the block are created within the record skeleton and the linkages between them are created to provide horizontal visibility. None of the contour cells created are linked into the record skeleton by the environment link; rather all linkage into the record skeleton is reflected by the linkages built in response to the module manipulations. To accommodate multiple contour cell creation, it is essential that block entry-point coding be executed. The block entry-point coding designates those contour cells within the program skeleton which must have copies created in the record skeleton as part of block entry simulation. It also designates the code cell which contains the code for the block being entered. On creating a contour cell within the record skeleton, the names contained within the declaration subcells of that contour cell is defined and, where necessary, the initial values are set. A contour cell within the program skeleton contains all of the information necessary to accomplish this. For
example, a declaration subcell for a routine name within the program skeleton contains a label value, the ip of which designates the entry-point coding for the routine and the ep of which designates the contour cell of the program skeleton in which the routine is declared. From this information, the routine name existing within some contour cell being created in the record skeleton can have its value initialized, as long as the creation activity can match a newly created record with the contour cell within the program skeleton from which it was created.

Any contour cell within the record skeleton contains an antecedent link; this is set at the time of creation and can be used by either the IBS or the EBS. Once all of the required contour cells have been created, and have had the names within them initialized, the block entry statement simulation terminates by updating the processor's ip and ep to begin executing the first instruction of the code for the block within the contour cell created for the local variables of the block.

The block exit statement

Block exit requires a return of the site of activity to a contour cell within the record skeleton which already exists, and a return of execution to an instruction subcell within a code cell in which some of the instructions have already been simulated. The location of the code cell and the instruction subcell containing the instruction next to be simulated is determinable at compilation. The processor ip can be reset by using the value so determined, which is stored as part of the block-exit instruction. Simulation of the block exit instruction need only check that the processor is executing in the contour cell of the block.
to be exited. Unless a failure condition exists, this will be true. The 
site of processor activity is altered by setting the processor ep to 
designate the contour cell within the record skeleton which is designated 
by the environment link of the contour cell in which the block exit in-
struction is executed.

The procedure call statement

Procedure call is the second of the statements which causes the 
creation of contour cells within the record skeleton. Only one contour 
cell is created, for the formal parameters of the procedure, along with 
the return address for the procedure. The value of a routine name has 
been set to designate the entry-point coding for the routine. The pro-
cedure call simulation executes this entry-point coding as well as the 
call statement itself. The call statement is executed in a left-to-right 
fashion. The procedure name is first retrieved using the IBS. If no such 
name can be found, or if the value of the name does not designate the 
entry-point code for some routine, execution fails. Once the entry-point 
coding has been located within the program skeleton, it is used to locate 
the formal parameter contour used to create a contour cell within the 
record skeleton. The instructions within the entry-point coding direct 
the creation of the return address from the processor ip and ep as they 
existed at the time of procedure call. This value is stored within the 
formal parameter contour in the record skeleton as the value of the 
return address identifier, RAId. The parameter values are bound, using 
the entry-point coding (which identifies the parameter names in order) 
and the argument list within the call statement. An argument may be a
constant, a reference to a name, or an arithmetic expression. The evaluation of an argument proceeds in the same manner as the evaluation of the source of an assignment statement, described above. In this case, however, no value returned by a function can be designated. Once the values of the parameters have been bound, the formal parameter contour within the record skeleton is linked into the record skeleton. The choice of contour cell to which the formal parameter contour is to be linked is made in accordance to the EBS, which uses the contour cell in which the procedure was defined and the site of the call statement in making this choice. After linking the formal parameter contour into the record skeleton, the procedure is activated by setting the processor ip to designate the last instruction of the entry-point coding and the processor ep to designate the formal parameter contour. Procedure activation is simulated as a block entry instruction; it is in this way that a procedure call, in its entirety, can cause multiple contour cells to be created in the record skeleton.

The procedure return statement

BL provides for value-returning routines, as well as pure procedures. Return from either of these procedure types is simulated by the procedure return statement. If a value is to be returned, that value is determined within the context of the return statement. Only one value can, of course, be returned, but it may be a label value. If a value is to be returned, it is determined in the same manner as the value of the source within an assignment statement. The value which is placed into TEMP during the simulation of an assignment statement is, in this case,
placed in the processor for transfer.

The return itself is accomplished by determination of a declaration subcell for the name RAId, using the IBS. This declaration subcell had a label value stored within it during the simulation of procedure call. The value is retrieved and decomposed into an ip and an ep, which are then stored as the processor ip and ep, completing the simulation of the return statement.

The identifier binding strategy, IBS

The function of the IBS is to define the correspondence between a reference to an identifier within an instruction, the reference occurrence, and a declaration subcell for a name having that identifier, the declaration occurrence. In a block-structured language, this is done by a search of the contour cells which form the processor environment. Choice of a particular declaration subcell within the processor environment is a function of the exact specification of the naming structure. In those block-structured languages known to the author, the first declaration subcell containing the desired name which is encountered by a linear search of the processor environment, beginning with the site of processor execution is defined as the naming structure. In a module-structured language, providing both horizontal visibility and the explicit designation of environments, this does not suffice. The IBS is extended to accommodate these features.

A reference occurrence to an identifier may have associated with it a set of enter and leave directives. Each of these directives affects the resolution of the reference occurrence to a declaration subcell, and
can have further effect on subsequent uses of the IBS. Hence, the IBS must have access to a stack of environments, as well as access to the processor environment. To accommodate this, the processor maintains an environment stack. The IBS alters that stack, as well as the processor environment in response to enter and leave directives. Within an enter directive, a secondary reference occurrence appears, which must be resolved by the IBS, before proceeding to resolve the reference occurrence of an identifier. Considering all identifier references within a reference occurrence as "id occurrences," the IBS proceeds to resolve each id occurrence in turn.

For any id occurrence, IBS searches the current processor environment, locating a declaration occurrence for the name, using some specified search strategy. If the id occurrence exists by virtue of an enter directive and if the value of the name is of type environment, then the processor ep is changed to designate this value, after stacking the current processor ep. The next id occurrence is then resolved within this new processor environment. If the value is not of type environment, execution fails. If the id occurrence is not part of an enter directive, it refers to the name whose declaration subcell is to be returned by IBS. In each of the above, it is possible that a declaration subcell which exists by virtue of an include or an insert is found. In this case, the name is linked to another name within some other contour cell. The IBS locates this other name, the one designated by the link, and uses that declaration subcell in lieu of the one located directly. The other operations within IBS, whether as part of an enter directive, or as the
identifier resolution, remain unperturbed by this level of indirection. To execute a **leave** directive, the IBS pops the environment stack, replacing the processor ep with the environment designator with that which was the top of the stack.

In defining the IBS for a given module-structured language, variability exists within the search strategy employed. The semantics of the **enter** and **leave** directives, as well as the resolution of linkages between environments results from the definition of the module manipulations. It is the author's opinion that, at least initially, the search strategy should be a static search, in accordance to that defined for traditional block-structured languages.

**The environment binding strategy, EBS**

The environment binding strategy actually performs two separate functions. One is involved with the execution of the branch instruction, in which case the site of activity, existing after simulation of the branch, is determined by the EBS. The other case exists when a newly created contour cell is to be linked into the record skeleton. None of the extensions to block-structured languages defined within this research affect the execution of the EBS.

**An example of the execution process**

This research has led to the definition of a naming structure which is not supported in any block-structured language known to the author. In discussing any language, it becomes necessary to unambiguously define the naming structure and its effect on program execution. While the
definition of any identifier-binding strategy and environment-binding strategy will capture the naming structure, it is appropriate to demonstrate their effect using an example. To do this, the program in Figure 3.4 whose program skeleton is outlined in Figure 3.8, will be used. In this discussion, the line numbers given in Figure 3.4 will be used to designate the instruction whose effect is depicted. In addition, the environment link of a contour cell will be illustrated by nesting. The contour cell designated by the processor's ep will be designated by the location of $\pi$ within the figure.

A program in ML is defined as a set of global routines, one of which is designated as the main routine. In the example, this routine has also been named main, although this is not required. Before execution of the program can begin, the global environment must be created, and the processor initialized to execute in that global environment. Figure 3.9 depicts this state. This contour consists of only one declaration subcell, defining the routine main. The first field of the declaration subcell holds the identifier of the name. The other fields represent the label value of the name: The first subfield indicates that the code for the routine appears within the program text as the body of the

![Figure 3.9. S1, the initial record skeleton](image)
routine named main. The second subfield indicates that the routine is
defined within the environment depicted by the contour cell, in this
case the global environment. The contour cell has been labeled P1 for
reference within the discussion.

Figure 3.10 depicts the effect of a (system) call to the routine
main. In this case, the formal parameter contour has been created. It
is placed within the contour for the global environment, since it executes
within that environment, and the return address, named RAId, has been
created. The value of RAId indicates that, on returning from main, the
processor is to return to the global environment.

\[
\begin{align*}
\text{Figure 3.10.} \quad & \text{S2, the effect of the call to main} \\
& \text{In calling a procedure, after creating the formal parameter contour} \\
& \text{and performing any parameter binding, the procedure must be activated,} \\
& \text{which requires creation of the contour cells for the body of the proce-} \\
& \text{dure, as well as any subsidiary contour cells required because of the} \\
& \text{use of a module manipulation construct in the program text. The body of} \\
& \text{main has such a subsidiary contour, resulting from the declaration in}
\end{align*}
\]
line 13. Figure 3.11 displays the state of execution just before execution of the begin instruction in line 14. The contour created for quad is placed outside the global environment to display the fact that no identifier referenced within quad can be a free name, defined in the global environment. The value of f is a link to the value of eval within the contour created from the module quad. All links of this type are represented as a two-field value, the first of which designates the name to which the link is made, and the second of which designates the contour in which the name appears.

Execution of the begin instruction of line 14 requires the creation of the contour for the names local to the block, as well as any subsidiary contours. As is true of any block-structured language, an entered block is nested within the block from which it is entered. Only the value of

![Diagram](image)

Figure 3.11. S3, the state prior to execution of line 14
p needs discussion here. p is to name the environment created from the include in line 17. Figure 3.12 displays the state of the model prior to execution of line 18. In this figure, the contour for the environment which p is to name has been designated E2, and this designation appears as the value of p, for clarity. An arrow, similar to those used as the ep portion of the label which is the value for an eval could have been used.

The execution of line 18 requires a search of the processor environment for a declaration subcell defining the name x. The IBS locates the declaration subcell for x within P4, and assignment causes the value of

Figure 3.12. S4, the state prior to execution of line 18
x to be set to 3, as shown in Figure 3.13, the snapshot taken just prior to execution of line 19.

\[
\begin{array}{ccc}
\text{P1:} & \text{main} & [\text{main}] \rightarrow \\
\text{P2:} & \text{RAld} & \\
\text{P3:} & x & \\
& y & \\
& f (\text{eval}) & E1 \\
\text{P4:} & x & 3 \\
& y & \\
& g (\text{eval}) & E2 \\
& p & E2 \\
\end{array}
\]

\[
\begin{array}{ccc}
\text{E1:} & a & 2 \\
& b & -5 \\
& c & 4 \\
\text{P2:} & \text{Eval} & [\text{eval}]) \rightarrow \\
\text{E2:} & a & 1 \\
& b & 0 \\
& c & -1 \\
\text{P2:} & \text{Eval} & [\text{eval}]) \rightarrow \\
\end{array}
\]

Figure 3.13. S5, the state prior to execution of line 19

The statement in line 19, \( y := g(x) \), requires use of the extensions to the naming structure provided by the module manipulation constructs. In this case, the IBS first locates the declaration subcell for \( y \). This is found in P4. Next, the IBS locates the declaration subcell for \( g \). This is also found in P4. The IBS interrogates the value of \( g \), finding that it is a link to the routine eval contained in E2. This requires a transfer to E2, to locate the entry-point coding for eval. The formal parameter contour for eval is created and linked, by the EBS, to E2.
The argument for g is to be evaluated in P4, according to line 19. Thus, the processor must return to P4 to locate the value of x, using the IBS, then transfer this value to the formal parameter contour for eval, binding the parameter. This type of environment straddling during argument evaluation and parameter binding is common in block-structured languages and normally subsumed in the semantics of procedure call. Figure 3.14 depicts the state of execution just prior to activation of the routine eval, which is the activation of the routine designated by g in line 19. The value of RAId indicates that execution of a return within eval returns the processor to the environment labeled P4.

Execution of eval has been described as the evaluation of a quadratic function of the form \( ax^2 + bx + c \). The particular quadratic function to be evaluated, as defined by E2, is \( x^2 - 1 \) which, for \( x = 3 \), is 8. Figure 3.15 displays the state of execution just prior to line 6, in which the procedure has been activated and the assignment to y made.

Return from a value-returning function requires the transfer of a value by the processor, as well as execution of a return. In executing line 6, the value of y is first stored in the processor. Then the value of RAId is located and used in the return. The value of RAId designates the environment to which the return is to be made, and also designates the instruction to first be executed after the return. The ip portion has been elided in the snapshots for clarity, since a return could be to the middle of an instruction, as is true in this case. In this case, return to P4 from eval, and assignment of the returned value to the name y, as determined by the IBS as indicated in the above discussion for S5.
Figure 3.14. S6, the state prior to activation of g
Figure 3.15. S7, the state prior to execution of line 6
(Figure 3.13), are treated as having one, albeit large, effect on the record skeleton. Figure 3.16 displays the state just prior to execution of line 20. The contours which were created in the record skeleton in executing eval (or g) have been removed from the snapshot for clarity. In reality, they are either retained or deleted according to the retention strategy employed by the block-structured language augmented by the module manipulation constructs.

Figure 3.16. S8, the state prior to execution of line 20

Execution of line 20 causes the output of the value 8, and has no effect on the record skeleton.

Line 21 causes a change in the processor environment, by returning
the processor to P3. Figure 3.17 displays the state prior to execution of line 22. Again, the contours of P4 and E2 have been removed from the snapshot for clarity; the contours are deleted or retained in the record skeleton in accordance to the retention strategy employed by the particular module-structured language, as a reflection of the retention strategy employed by the underlying block-structured language. The contour E2 has a lifetime determined by the lifetime of P4, since E2 is the pictorial representation of an environment created in response to the needs of the environment represented by P4, and in some respects can be considered an extension to the more traditional interpretation of an environment created by a block.

Figure 3.17. S9, the state prior to execution of line 22
Execution of line 22 requires the same actions as did the execution of line 19. In this case, however, control is transferred with $E_1$, since a different instance of eval is linked to $f$ than was linked to $g$. Figures 3.18 and 3.19 mirror Figures 3.14 and 3.15 for this activation of eval.

Figure 3.20 displays the state prior to execution of line 23. The execution of $y := f(6)$ parallels the execution of line 19, to a significant extent and, for this reason, is not further enlarged upon.

Execution of line 23 causes the value 46 to be output.

Line 24 triggers a return from the routine main. Since this is a return to the global environment, it also signals a return to the system code which caused the call and activation of main. A return to the system code signals the termination of execution. Figure 3.21, included for completeness, represents the final state of execution, in which the processor has returned to the system code.

Figure 3.18. $S_{10}$, the state just prior to activation of $f$
Figure 3.19. S11, the state just prior to execution of line 6

Figure 3.20. S12, the state prior to execution of line 23
Figure 3.21. S13, the final state of execution
CHAPTER IV. A NONTRIVIAL EXAMPLE OF THE IMPLEMENTATION MODEL IN ACTION

This chapter is devoted to a second example demonstrating both the power and complexity of the module manipulation constructs, as well as the implementation model. The example used was created with those goals in mind; the choice of modularization induced is not defended in any other way. The example consists of four modules, having a fair degree of interaction, and one routine, and performs some relatively simple geometric manipulations. The program text is displayed in Figure 4.1, and will be described briefly before the actions of the implementation model on this input program are discussed.

```
1. mod point(x,y,error);
2. accessible routine get(coord);
3. begin;
4. if coord = 1 then return(x);
5. if coord = 2 then return(y);
6. call error;
7. end;
8. accessible routine set(coord,val);
9. begin;
10. if coord = 1 then x := val;
11. else if coord = 2 then y := val;
12. else call error;
13. return;
14. end;
15. endmod
```

Figure 4.1. The source listing of the example
mod line(p1,p2);
  accessible decl inf;
decl xa;
decl ya;
routine point;
  begin;
    xa := <enter p1> get(1);
    ya := <enter p1> get<leave>(2);
    return;
  end;
accessible routine getx(y);
  begin;
    decl m;
    decl x;
call point;
bigest := inf;
m := slope;
  if m = 0 then return(inf);
  if m = inf then return(xa);
x := (y - ya)/m + xa;
return(x);
end;
accessible routine gety(x);
begin;
decl m;
decl x;
call point;
bigest := inf;
m := slope;
  if m = inf then return(inf);
  if m = 0 then return(ya);
y := m * (x - xa) + ya;
return(y);
end;
include slope(p1,p2) with inf -> bigest;
endmod

Figure 4.1 (Continued)
mod intrcp;
  accessible decl a;
  accessible decl b;
  accessible routine yint(gety);
    begin;
    b := gety(0);
    return;
  end;
  accessible routine xint(getx);
    begin;
    a := getx(0);
    return;
  end;
endmod

mod slope(p1,p2);
  accessible decl inf;
  accessible routine slope;
    begin;
    decl xl;
    decl yl;
    decl x2;
    decl y2;
    decl dx;
    decl dy;
    xl := <enter p1> get(1);
    yl := <enter p1> get(2);
    x2 := <enter p2> get(1);
    y2 := <enter p2> get(2);
    dy := y1 - y2;
    if dy = 0 then return(0);
    dx := x1 - x2;
    if dx = 0 then return(inf);
    return(dy/dx);
  end;
endmod

Figure 4.1 (Continued)
routine main;
begin;
routine trap;
begin;
goto endrun;
end;
point is point(,,trap);
begin;
p1 names include point with get → geta, set → seta;
p2 names include point with get → getb, set → setb;
routine mirror(xint,yint);
begin;
begin;
begin;
begin;
begin;
The Example

The first module, point, is used to store a description of a point in a plane. Geometrically, a point on a plane can be represented as an ordered pair \((x,y)\). The module, providing routines for storing the coordinates defining a point, and retrieving those coordinates, serves a data structure manipulating module in which the choice of representation for a point is hidden from the user. This usage of a module is quite close to the use of an operation cluster, as described by Liskov and Zilles (17). The module, point, has a parameter list, which can be used to define a point at the time of module reference, by using a statement of the form

```cpp
include point(3,5);
```

for example. The third parameter is used as an error trap and can be set or left unset as desired by the user of the module. Of course, if the third parameter has not been given a value, and an error occurs within either get or set, execution of the program fails "ungracefully".

The module line is used to define a line from a line segment, which is defined by two distinct points. The module requires two points, contained in environments \(p_1\) and \(p_2\) to be provided. Using these two points, the line is defined and, from that, any point on the line can be determined, given one of the coordinates. If no value for the other coordinate can be uniquely determined, which can occur on lines parallel to the axes, the routines in line return a value, here named \(\text{inf}\), which must be set by the user since it is a flag to the user that a point could not
be uniquely determined. Line makes use of the module slope in performing its calculations; this does affect the storage requirements of any user of line, but not the capabilities of that user.

The module intrcp is used to find the x-intercept and y-intercept of a line, if they exist. No checking is provided internally to determine whether or not the intercepts exist. Each of the routines within intrcp require as an argument a routine from a line module, to provide the correct line on which the point is to be found.

The last module, slope, finds the slope of a line as defined by the points defined within p1 and p2, the module parameters.

In looking at the program text, the reader will note the increased level of interaction between the effects of the module manipulation constructs. For example, parameters to one module are used as arguments to a module referenced internally (module line and its include slope statement), and nested use of module references.

The main routine defines a line from the two points (4,6) and (-5,-3) and finds the intercepts of that line. Its internal routine, mirror, defines the line, perpendicular to the first, which passes through the x-intercept of the first line, and evaluates a few points on it. Further, it designates the value of infinity to be 32768, which is too large to be stored in a half word on the IBM 360/65.

As stated above, the program was designed for demonstrating the module manipulation constructs and implementation model; it is not particularly well-suited for BL, which performs only integer arithmetic necessitating the careful selection of the lines used to guarantee
integral slopes.

The Initial Graph of the Program

Because of the complexity of the initial graph of this program, it is presented in pieces, in Figures 4.2, 4.3, 4.4, 4.5, 4.6 and 4.7. For clarity, arc labels have been eliminated where they provide an artificial ordering, or where they are obvious, and accessible declarations have been underlined. Figure 4.2 displays the graphical representation of the module point. The contour node, which has a 1 next to it is the root of

Figure 4.2. Graphical representation of module point
Figure 4.3. Graphical representation of module line
Figure 4.4. Graphical representation of module intrcp

the subgraph representing the module. The 1 is used in constructing the
full initial graph from its pieces. Figure 4.3 displays the graphical
representation of the module line. Figures 4.4 and 4.5 contain the
graphical representations of the modules intrcp and slope, respectively.
Figure 4.6 displays the graphical representation of the set of declara-
tions local to the body of the routine main. The reader should note the
handling of the label endrun, which appears as a simple variable having
the value 2. This is a simplification of the handling of labels at this
stage. Just as identifiers which are appended to the subtree whose root
Figure 4.5. Graphical representation of module slope

Figure 4.6. Initial representation of set of declarations local to routine main
Figure 4.7. Initial representation of set of declarations within first nested block
is accessed by \texttt{vars(contour node)} during the execution of a module manipulation construct are distinguishable from other identifiers in that subtree, label constants can be distinguished by some additional information in the container node.

Figure 4.7 displays the graphical representation of the variables declared local to the block which encompasses lines 94-127 of the program text in Figure 4.1. The complete initial representation of the routine mirror is included in Figure 4.7.

Figure 4.8 outlines the complete initial graph. The complete graph can be constructed by replacing the contour nodes labeled 1-6 with their expansions, given in Figures 4.2-4.7, respectively.

The Construction Process Applied to This Example

In applying the construction process to this example, more of the algorithms are exercised. In describing this process, some false starts will be mentioned to demonstrate the effect the level of interaction among the constructs has on the construction process. The first transformation applied to the initial graph is that for the \texttt{is} construct in line 93, defining the local module point. The reader who studies the procedures in the Appendix will notice the care which needs to be taken in creating a new module from one having the same name, and also the deferral of the binding of trap, creating a partially-bound parameter-argument pair. Figure 4.9 illustrates the result of defining the local module point on the set of declarations in which the construct originally appeared. The back link from the binding of error to the argument trap
Figure 4.8. Initial representation of the global environment
is not resolved until after the construction of the program skeleton.

The construction process next transforms the set of declarations for
the block which begins at line 94. Transformations for both of the
names constructs are performed, in an arbitrary order. Both of the
references to the module point are resolved to the module defined in
line 93. The copies of the modules each provide an environment link back
to the set of declarations in which the referenced module was located.
Figure 4.10 illustrates the effect of these transformations on the
subgraph representing the set of local declarations for this block, whose representation within the initial graph was given in Figure 4.7. The hexagons are used as place holders, representing the entire graphical representation of the value, which is discernible from a previous figure. This is done to simplify the figures.

Once the transformations shown above are complete, the definition of the routine, mirror, is interrogated. This requires locating the contour node which is the root of the subgraph for the variables local to the body of mirror. Following the approach outlined in Chapter III, an attempt to perform the transformation for the include in line 101 is made. Neither p1 or p2 have had their definitions resolved, so the attempt is legitimately aborted, and the transformations for the names constructs are tried. In this case, both can be performed as encountered, with the references to point resolved to the module defined in response to line 93. Figure 4.11 shows the effect on the set of local declarations within the body of the routine of these transformations. The transformation for the include in line 101 is now reattempted. The module line is located in the global environment, and the transformation for the include is performed, yielding the subgraph shown in Figure 4.12 as the graphical representation of the first stage of the transformation in which a copy of the representation of line has been attached and the parameters semibound outward. As is the case with routine parameters, the parameter binding must be deferred until the environments have been created in the program skeleton. To complete the transformation, the copy of line requires full definition. This entails performing the
Figure 4.10. Effect of names transformations (lines 95, 96)
Figure 4.11. Effect of names transformations (lines 99, 100)
Figure 4.12. Effect of first stage of include transformation (line 101)
include transformation by resolving the reference to the module slope. The module is found in the global environment, and the transformation performed. Once completed, the transformation to perform the inclusion of line, in line 101, is completed. Figure 4.13 displays the final result of the transformation required by line 101. In this figure, the reader can see the effect of the renaming in line 50, as well as a back chaining effect on the arguments p1 and p2, passed to the module line, and then passed again to the module slope. This completes the set of transformations required in the definition of mirror, since it contains no nested blocks.

Returning to the block which begins at line 94, the declaration of no other routines signals the end of processing for the set of declarations local to the block. The code for the block is now interrogated for any nested blocks. The block which encompasses lines 113-126 is found, and the contour node which is the root of the subgraph for the set of declarations local to this block is located. The order in which the transformations for the two \texttt{include} statements (lines 114, 115) is not significant. Considering line 114 first, the same recursive resolution as described above for line 101 occurs, yielding the same basic result, except for the binding of the parameters of the module line. In this case, the binding must be made to p1 and p2 as declared in lines 95 and 96. The transformation required for line 115 is straightforward. The result of all of the transformations for the block defined in lines 94-127 is shown in Figure 4.14, with the results of the transformations required by the block defined in lines 113-126 the most recent
Figure 4.13. Effect of secondary include transformation for the module slope (line 50)
Figure 4.14. The completely defined sets of declarations nested within the routine main.
transformations applied to the graph.

The rest of this stage of the construction process is an unwinding back to the global environment where, since no other global routine has been defined, this stage terminates.

The second and third stages of the construction process build the program skeleton for input to the third phase of the model. The result of these stages, less the code cells, is given in Figure 4.15. The resolution of module parameters whose arguments were environments is explicitly shown in Figure 4.15; the resolution of the third parameter of the module point, in which the parameter error was bound to the routine trap, is not explicitly detailed, primarily because no code cells are shown in the figure. However, this binding has been indicated by the use of [trap] with every declaration subcell for error.

This completes the construction process for the example in Figure 4.1.

The Trace of the Program Execution

In tracing the execution of this program, the effect of the interactions among the module manipulations, as evident in the source of the program displayed in Figure 4.1, as well as the effect of the enter and leave commands (see lines 22, 23, 104, 105 in Figure 4.1 for example) are of primary interest. For this reason, a full trace of program execution is not presented. The part of the trace displayed, in Figures 4.16-4.54, focuses on the issues for which the program was initially designed.
Figure 4.15. Program skeleton for example
As is true of the example in Chapter III, the simulation begins with the processor executing system code within the global environment, as displayed in Figure 4.16. This system code calls the routine main. Figure 4.17 displays the state of the execution following the execution of the entry-point coding for the routine, prior to the execution of line 94. The reader will note that, as part of the execution of this entry-point coding, the value of the label endrun has been initialized to a label value, the ep of which designates the contour cell in which its declaration subcell appears. The nesting of the contour cells visually displays the effect of the EBS, as before. Again, the IBS and EBS are assumed to have been defined in terms of the static naming structure of traditional block-structured languages.

Figure 4.18 displays the state of execution after simulation of line 94. In this case, execution of the block entry-point coding causes the creation of three contours within the record skeleton, designated P4, E1 and E2. As part of the process of creating these contour cells, the values of the names p1 and p2 in P4 are set to designate the environments E1 and E2, respectively. Thus, the module constructs in lines 95 and 96 have now had their run-time effect described. As part of this creation process, the values of error within E1 and E2 have been set. The ep portion of these values are set to designate the contour in the record skeleton in which a declaration subcell for the routine trap appears. This is done without recourse to the IBS, from information contained within the block entry-point coding and the processor environment at the time of block entry.
Figure 4.16. The initial state of the execution

Figure 4.17. The state prior to execution of line 94

Figure 4.18. The state prior to execution of line 113
The execution of line 113 requires the allocation of four additional contours within the record skeleton: one for the block itself, which requires the initialization of all its names since no name is explicitly declared; one for the environment implicitly defined in line 115, designates as E5; one for the environment implicitly defined in line 114, designated E3; and one for the subsidiary environment defined within the set of declarations comprising the module line, designated E4 in Figure 4.19. All of these contour cells require some of their declaration subcells to have values created during the allocation process. Figure 4.19 displays the state of execution prior to execution of line 116.

The assignment in line 116 requires the IBS to resolve the linkage in order to locate the declaration subcell in which the declaration of inf appears. To do so, the IBS must interrogate E3, to locate the appropriate declaration subcell. The result of executing line 116, shown in Figure 4.20, is a change within the environment designated E3, which is external to the processor environment, but horizontally visible as a result of the include statement in line 114.

Lines 117-120 are used in initializing the set of two points, designating the line on which the geometric manipulations are performed. In simulating the execution of line 117, the IBS, in resolving the reference to seta, must search the declaration subcells in P5, the active contour, then follow the environment link of P5 to P4 before locating a declaration for seta. At this point, the IBS must resolve the linking information, with the value of seta being located in E1. Performing the call
Figure 4.19. The state prior to execution of line 116

Figure 4.20. The state prior to execution of line 117
instruction causes the creation of two additional contours in the record skeleton: one for the formal parameters and return address; and one for the names local to the body of set. After creating the formal parameter contour and setting the return address, execution of the entry-point coding of set binds the parameters and causes procedure activation. The activation of the procedure causes creation of the contour cell for the set of names local to the body of set, which is empty. The processor's ip is set to the first executable statement in the code for the body of set (lines 10-12), and the processor's ep is set to the contour for the local names. Execution of lines 10-12 results in the record skeleton as shown in Figure 4.21.
On executing the `return` in line 13, the IBS is used to locate the return address and the processor is returned to the environment labeled P5. Both of the contours created as part of the execution of line 117 have been removed from the snapshot of the record skeleton, in Figure 4.22, taken prior to execution of line 118. The removal of these contours is done for clarity; the contours would be removed from the record skeleton only if the language being modeled utilized an environment deletion strategy.

Execution of lines 118-120, which follow the pattern of line 117 (except that lines 119 and 120 cause execution within the environment designated E2) results in the definition of the points (4,6) and (-5,-3). Figure 4.23 displays the state following execution of line 120.

Lines 121 and 122 appear to present a more complex problem in calling a procedure since, in both, the argument designated is itself a linked name, however, no such added complexity arises. The snapshot in Figure 4.24 represents the state of execution immediately following execution of lines 121 and 56 and just prior to execution of line 57. The reader will note that the parameter `gety`, whose declaration subcell appears in the formal parameter contour for `yint`, within E5, has been set to the value `gety` within environment E3, implying that on evaluating an argument, the IBS performs in a manner identical to any reference occurrence of a name.

Execution of line 57 necessitates locating the declaration subcell for `b`, which is found within E5, the environment described by the module `intrcp`. After that, the routine `gety` must be activated as part of the
Figure 4.22. The state prior to execution of line 118

Figure 4.23. The state prior to execution of line 121
Figure 4.24. The state prior to execution of line 57

(implicit) procedure call. The value of gety, above, indicates that it is defined within the environment designated by E3 which was created from the description provided by the module line. The EBS, which is a static binding strategy, causes the execution of gety to be performed within the context of E3. The snapshot in Figure 4.25 depicts the state of execution prior to execution of line 42, in which the value of RAId, in the formal parameter contour of gety has been set to a label value, the ep portion of which is E51, the environment in which the call
Figure 4.25. The state prior to execution of line 42
instruction was executed.

Line 42, which causes the routine point, defined in lines 20-25, to become active is of interest primarily because of the code which constitutes the body of the routine. In considering the execution of the routine point, the snapshot of the activation of point is included in the trace as displayed to allow for a somewhat more graceful study of its execution. This snapshot was taken prior to execution of line 22, and is depicted in Figure 4.26.

In executing line 22, xa := \langle enter pl \rangle get(l), the model must perform a variety of actions. After locating the declaration subcell for xa, contained in the contour cell designated E3, the processor must locate the environment identified by pl. The IBS locates the declaration sub-cell for pl, determining the environment, El, which is its value. This causes the processor to transfer to the environment El and call upon the IBS to resolve the remaining part of the identifier reference. The IBS locates the declaration subcell for get and the processor, using this information, executes the entry-point coding for the routine get as defined within El, binding its parameters and activating the routine. This state is depicted in Figure 4.27. The processor then begins execution of the code which constitutes the body of get. At this point, the five routines get, point, gety, yint, and main can be considered active.

The execution of get requires the return of the current value of x, as line 4 indicates. The value, 4, is transported back to environment E31, in which the execution of the assignment xa := \langle enter pl \rangle get(l) is completed. This requires the value of xa, in E3, to be set to 4,
Figure 4.26. The state prior to execution of line 22
Figure 4.27. The state prior to execution of line 4
as shown in Figure 4.28.

Execution of line 23 follows the same, rather convoluted, pattern, yielding an execution state as shown by the snapshot of Figure 4.29, taken prior to the execution of line 24, in which the value of ya in E3 has been set to 6.

Figure 4.30 displays the snapshot after execution of lines 24 and 43. Execution of line 24, the return in routine point, causes the processor's site of activity, as defined by its ip and ep, to be set to the contour for the local variables of gety, designated E31 in Figure 4.29, ready to execute the assignment bigest := inf. Execution of line 43, while exercising the IBS, is relative straightforward, resulting in the value of inf, in E4, being set to the current value of the vertically visible name inf, which is 32768.

Execution of line 44 requires an activation of the routine slope, defined within environment E4, as shown in Figure 4.31. Once activated, the execution of slope makes extensive use of the enter command to access the coordinates of the two points defining a line segment. It should be noted that, in executing the statements in lines 76-79, no anomalous behavior occurs, since every statement in a sequence has been defined to execute in the same environment, during the description of the enter and leave commands in Chapter II. Hence, each of xl, y1, x2, and y2 are evaluated within the correct environment by the IBS. Figure 4.32 displays the state of execution prior to execution of line 84, in which the slope of a line which is not parallel to an axis, is computed and returned.
Figure 4.28. The state prior to execution of line 23
Figure 4.29. The state prior to execution of line 24
Figure 4.30. The state prior to execution of line 44
Figure 4.31. The state prior to execution of line 76
Figure 4.32. The state prior to execution of line 84
The return in line 84 causes the processor to transfer the value 1 back to the environment labeled E31 in Figure 4.32. This causes the completion of line 44, whose effect is to set the value of m to 1, the returned value, as shown in Figure 4.33.

Figure 4.34 displays the state prior to execution of line 48. Neither line 45 nor 46 have any effect on the state of execution. Line 47 computes the value of y, using a form of an equation which can be algebraically generated from an equational statement that the slope of a straight line can be determined by any two points on the line. Once the value of y has been calculated, the execution of the program returns that value within the processor, transferring to the environment labeled E51, and completing the statement in line 57, which sets the value of b to 2. The state prior to execution of line 58 is depicted in Figure 4.35; Figure 4.36 displays the result of executing line 58, simulated by setting the processor's ep to P5 and the processor's ip to designate the instruction subcell in which the instruction generated from line 122 is stored.

Execution of line 122 follows the same basic pattern as the execution of line 121. In this case, however, the routines xint and getx are activated instead of yint and gety. Figure 4.37 displays the state prior to the execution of line 63, after the parameter, getx, has been activated, and the value of the x-intercept, a, has been calculated. The reader will note that the execution of the routine point in line 30, is redundant within the context of E3, as the values retrieved are already known. However, the effect of lines 76-77 are critical as the
Figure 4.33. The state prior to execution of line 45
Figure 4.34. The state prior to execution of line 48
Figure 4.35. The state prior to execution of line 58

Figure 4.36. The state prior to execution of line 122
reactivation of the routine slope creates entirely new contour cells for
the environments needed by the execution of the routine. Figure 4.38 dis­
plays the effect of the return in line 63, which depicts the state prior
to the execution of line 123.

Lines 123 and 124 causes the output of the values -2 and 2 respec­
tively, after a mild amount of exercising the IBS.

The execution of line 125, the call to mirror, causes the creation
of six contour cells within the record skeleton. First, the contour
for the formal parameters and return address for the routine is created,
and nested within P4, as a sibling of P5, the block in which the call is
executed. In activating the routine, four subsidiary contour cells are
created in addition to the one needed for the names local to the body of
the routine. Lines 99 and 100 each require the creation of an environ­
ment, designated E6 and E7, respectively, within Figure 4.39. The values
of the local names P1 and P2 are set to reflect this. Line 101 requires
the creation of two contour cells, one for the module line and one for
its subsidiary environment, described by the module slope, which be­
comes necessary as a result of line 50. Within E8, the contour for the
module line, the values of bigest and slope have been set to refer to
E9, created from the description provided by the module slope. Fig­
ure 4.40 displays exactly the same state as Figure 4.39. Because of the
complexity of the record skeleton as displayed in Figure 4.39, those
contour cells to which no reference is made during the execution of the
routine mirror have been depicted without details.

Execution of lines 104 and 105 again demonstrate the effect of the
Figure 4.37. The state prior to execution of line 63

Figure 4.38. The state prior to execution of line 123
Figure 4.39. The state prior to execution of line 104

Figure 4.40. The state prior to execution of line 104
enter command and, in addition, demonstrate the effect of the leave command. Figure 4.41 displays the state prior to execution of line 13. In the transition from the state displayed in Figure 4.40 to that of Figure 4.41, the following events occur. First, the IBS is used to locate the declaration subcell in which P1 is defined. E6, the value of p1, is retrieved, and the processor's ep is set to this value, transferring the site of processor activity, and, hence, changing the processor environment. Within the environment designated by E6, the declaration subcell for set is located. The processor's ip is changed, to cause execution of the entry-point coding for the routine, as designated within the value of set. Execution of that entry-point coding causes the binding of the parameters for set. The evaluation of the first argument is trivial as it is a constant; hence, the binding of the parameter coord is

Figure 4.41. The state prior to execution of line 13
straightforward. Note, however, that the processor was executing in E6 when this evaluation was performed. Evaluation of the second argument is begun by executing the leave command, which causes the processor's ep to have the contour designation P7 reinstated. It is within this processor environment that the IBS establishes the correspondence between the reference occurrence of xint and the declaration occurrence of xint, found in P6. Once this declaration subcell has been located, the binding of the parameter val is completed, and set activated. Execution of the procedure activation causes the creation of the contour cells nested within the contour designated by E6 in Figure 4.41. Following the execution of the assignment appearing as the then clause in line 11, the state as displayed in Figure 4.41 has been created.

Execution of the return comprising line 13, causes the processor's ep to be set to P7, by using the value of the return address created during the execution of the procedure call. Figure 4.42 displays the state prior to execution of line 105.

Execution of line 105 proceeds in much the same way as did execution of line 104. In executing line 105, the IBS is used to locate the declaration subcell in which the name p2 is defined. The value of p2 is used to transfer the site of processor execution to E7. In this environment, the name set is located using the IBS. Once found, execution of its entry-point coding is begun after its formal parameter has been created. This entry-point coding causes the evaluation of the arguments. Before evaluating any argument, however, the leave command is executed, which reinstates P7 as the value of the processor's ep. Again, the
evaluation of the first argument is trivial. Evaluation of the second argument is a more complex process, since the argument is an arithmetic expression. In evaluating an arithmetic expression, each operand must be evaluated. In this case, the value of \( \text{yint} \) is retrieved from the declaration subcell for \( \text{yint} \) in \( P6 \), in accordance with the IBS. The binding of the parameter \( \text{val} \) is done after the negation of 2 is performed. Thus, the value of \( \text{val} \) is set to \(-2\), as a result of the execution of the routine \( \text{set} \), the value of \( x \), in \( E7 \), is set to \(-2\). The snapshot depicted in Figure 4.43 displays the state prior to execution of line 106.

Figure 4.44 displays the snapshot taken after the execution of
Figure 4.43. The state prior to execution of line 106

line 107 has begun, prior to execution of line 30, the first executable instruction in the text of routine getx, declared within the module line. The effect of line 106 is to set the value of inf, within E8, to 32768. The call and activation of getx results in the creation of the contour cells for the formal parameters and names local to the body of getx "within" E8.

Execution of line 30 results in the values of xa and ya, in E8 being set to the values of the coordinates of the point stored in E6. Execution of line 31 sets the value of inf within E9 to the value of inf within E8. Line 32 requires the activation of the routine slope, whose definition appears in E9, as determined by the IBS. Figure 4.45
Figure 4.44. The state prior to execution of line 30
contains the snapshot taken prior to execution of line 76, the first executable statement of the routine slope. Lines 76-79 have the effect of retrieving the coordinates of the two points defining the line segment whose slope is to be calculated. These points, stored in E6 and E7, respectively, are retrieved after the same type of exercise of the IBS as described above in conjunction with Figures 4.31 and 4.32. Once this has been accomplished, lines 80-83 are executed, setting the values of dx and dy, as shown in Figure 4.46.

The slope of the line is calculated and returned, to be used in setting the value of m, as shown in Figure 4.47, as the result of executing line 84 and completing the execution of line 32. The value of x is calculated in line 35 and returned in the execution of line 36. This allows the execution of line 107 to be completed. The snapshot in Figure 4.48 displays the state prior to the execution of line 108, in which the value of the name x, whose declaration subcell appears in P7.

Figure 4.49, which displays the state after execution of line 108, requires the same activities in generating the state from that of Figure 4.48, as did generating the state in Figure 4.47 from that of Figure 4.43.

Execution of lines 109 and 110 causes the values 1 and -6 to be output, and cause no change to the state as displayed within a snapshot. Figure 4.50 contains the complete snapshot displayed in Figure 4.49, preparatory to executing line 111, the return from mirror.

To simulate the return from mirror, the processor's ip and ep are set using the value of RAId, found in P6. This causes the processor's
Figure 4.45. The state prior to execution of line 76
Figure 4.46. The state prior to execution of line 84
Figure 4.47. The state prior to execution of line 35
ep to be set to designate P5, and the processor's ip to designate the instruction subcell containing the instruction generated from line 126. As noted earlier, although the snapshot now no longer contains the contour cells labeled E6, E7, E8, E9, P6 and P7, whether or not they are actually removed from the record skeleton is determined by the environment retention or deletion strategy utilized by the underlying base language. Figures 4.51, 4.52, 4.53 and 4.54 contain the snapshots taken after execution of lines 111, 126, 127, and 129 respectively, returning the processor to execution of the system code within the global environment, signaling program termination.
Figure 4.49. The state prior to execution of line 109

Figure 4.50. The state prior to execution of line 111
Figure 4.51. The state prior to execution of line 126

Figure 4.52. The state prior to execution of line 127

Figure 4.53. The state prior to execution of line 129

Figure 4.54. The state at program termination
CHAPTER V. CONCLUSIONS AND SUGGESTIONS FOR ADDITIONAL INVESTIGATION

This research has investigated the feasibility of providing an additional decomposition technique, termed modular decomposition, in programming languages which support the more traditional decomposition techniques of procedures and nested blocks. This is done in an attempt to ease the more global problems associated with the high cost of software development, both by providing a cleaner way to combine algorithms together to form the program and by providing ways of effectively utilizing algorithms previously developed in the solution of the current problem. The concept of modular decomposition was originally proposed by Parnas (23), and is not well-captured in existing programming languages. While no programming language can fully capture the algorithms developed by a problem solver, it is the author's belief that if the program which results from translating a set of algorithms into a programming language for communication with a machine can reflect more of the intent of the algorithms, then the job of maintaining that program is eased. Modular decomposition appears to be of particular utility in the large program environment, in which many are involved in defining the algorithms and writing the programs, and in which the resulting program is expected to have a long lifetime and be subject to modifications which are often drastic. All well-structuring techniques do, of course, attempt to ease the problems inherent in the large program environment, but as the soaring cost of software indicates, they are still dominating the development of software.
In this dissertation, a module has been defined as a set of declarations forming a unit, which model the solution to a problem or set of related problems. A set of module manipulation constructs to augment a block-structured language have been defined, providing the mechanisms for making use of these modules. An implementation model has been developed to define the semantics of these constructs and to provide a vehicle for investigating the effects of module interaction and modular decomposition on the naming structure of the underlying block-structured language. Investigation into this last issue has been conducted in terms of Johnston's contour model (9, 10, 11, 12), which itself was developed to investigate the naming structures in block-structured language. The phase of the implementation model which deals with the module manipulation constructs is defined by the source listing of the program, contained in the Appendix. The rest of the implementation model has been written either to create the initial data structures for the construction phase, or was written as an implementation of Johnston's contour model to exhibit the effect of the module manipulation constructs on the naming structure of a module-structured language and are not included. It has been argued that the effect of module manipulations on the run time environment is restricted only to an expansion of the system activity required during block entry and exit, creating and deleting a multitude of environments at these times instead of only one, as is currently done during block entry and exit in block-structured languages.

The set of constructs defined have been chosen to facilitate the
types of module manipulations which the author feels are needed, both by a program developer and by the developer of a module to be used by others. Only the actual use of a module-structured language can fully address the question of the completeness of this set of constructs. In an attempt to address this issue, the author has defined a simple time-sharing operating system, based on Halstead's PILOT operating system (8), for use on a PDP 11/40 minicomputer. In that instance, the modularity of the operating system was enhanced by use of a module-structured approach, and an underlying hierarchical structure of the operating system more clearly exhibited. This operating system has not been reported upon in this dissertation because, in describing any software of this nature, the architectural characteristics of the host machine dictate all of the details of the program, obscuring the benefits of a module-structured approach to the problem when presented in its final detailed form.

Modular decomposition appears to present a much-needed added dimension to the decomposition of programs, by clearly specifying communication paths between parts of the program and by removing some of the complexities which exist, in block-structured languages, in which implementation details must be prematurely considered (for example, when several procedures must manipulate the same data structure). Modular decomposition provides a mechanism by which at least some of the disadvantages inherent in block structure can be overcome. In many cases, in a block-structured language, names are declared within a context within which they are not needed, simply to defeat block
structuring. Using modules, names declared in this way can be removed from a context in which they are not needed, protecting their integrity and the integrity of the routines referencing that name. This is a form of protection which has been addressed, and partially solved, using traditional decomposition techniques.

Modular decomposition techniques do suffer from one drawback, however, in that a program references names not visibly declared. The binding on those names, while well-defined, does tend to violate some of the dictates of structured programming techniques. It is the view of the author that, through careful documentation practice, this liability can become less costly than some of the surprises that arise under dynamic binding of variable references.

Modular decomposition also provides for the association of a name with an environment, allowing the programmer to clearly specify the environment in which a name is to be evaluated. This has a side advantage of facilitating investigation of the state of program execution, without recourse to a dump, when recursive procedures are used.

The use of modular decomposition requires further investigation, as an extension to a viable underlying block-structured language. The question of the complexity of the resulting compilation process requires study, in an attempt to determine:

1) the difficulty of resolving identifier references

2) the cost of module manipulations during compilation

3) the efficiency of the generated object code.

Additional questions concerning the run-time costs of the use of module
manipulations need to be considered:

1) the cost of block entry and exit
2) the effect on the run-time stack, in terms of size
3) the effect on the display or chaining techniques within the run-time stack.

Modular decomposition techniques need also to be investigated as a tool in the definition of a program under development. This can, probably, only be done by extensive usage. Attempted use of these techniques can also resolve the question of the completeness of the set of constructs, and their utility in promoting the rapid development of reliable programs, facilitating the use of previously done work, where appropriate.

The broader question of the appropriate forms of decomposition techniques required to facilitate the production of large programs is still open. The modular approach appears to hold promise as a valuable tool to be used in addition to the more traditional procedural decomposition techniques.
BIBLIOGRAPHY


13. Johnston, J. B.; Berry, D. M.; and Murphy, D. P. "Expression Stack Management in Nested Declaration Computations." Unpublished paper, Computer Science Department, New Mexico State University, 1974.


ACKNOWLEDGMENTS

In the production of this dissertation, the support received has ranged from technical to moral. All of it was important. The roles played during the research and writing of this dissertation can best be summarized by:

| Directors                          | Roy F. Keller                  |
|                                   | Charles T. Wright, Jr.         |
| Technical assistance coordinator  | Charles T. Wright, Jr.         |
| Typist                            | Gwen Ethington                 |
| "Sanity preservers"               | Karen Ethington                |
|                                   | Patricia Gates                 |
|                                   | David Seidman                  |
|                                   | Carol Smith                    |
|                                   | Sharilyn Thoreson              |
|                                   | Charles T. Wright, Jr.         |
| Private fan club                  | Arthur A. Carr                 |
|                                   | Mary Ann Carr                  |
|                                   | Robert A. Carr                 |
|                                   | Janice A. Carr                 |
APPENDIX. AN IMPLEMENTATION OF THE CONSTRUCTION PHASE

The construction phase of the implementation model is presented in complete detail in this Appendix. To do so, the program listing is included as the most concise and precise description of all of the ramifications of the level of interaction between module manipulation constructs allowed. Before presenting this description, the storage structures used to represent any of the graphs in the sequence $G_0, \ldots, G_n$ which are generated during this phase are discussed. The reader may find it beneficial to refer to the description of the construction presented in Chapter III.

The Storage Structures

Each graph in the sequence has been defined as a rooted, labeled, direct graph, in which the arcs are labeled. These arc labels are used either to order the nodes they select, or to convey some information about the contents or type of the node it selects. In creating the storage structures to be used to represent the graph, the arc label information is subsumed within one or more of the fields defined within the storage structures. Some additional information, of a descriptive nature, may also be contained within the storage structures to facilitate the implementation of the transformations. This will be enlarged upon below.

The contour node is used within the graph to define a set of declarations. Each of these sets is decomposed into seven disjoint
sets, some of which may be empty. In addition, a contour node has an
arc labeled \textit{env} emanating from it, designating the contour node for the
enclosing block. Within the implementation model, a contour node is
represented by a based structure defined by:

\begin{verbatim}
1 DECL_HEADER,
2 @ENC_BLOCK POINTER,
2 HEADER,
3 @VARS POINTER,
3 @RTNS POINTER,
3 @MODS POINTER,
3 @INCS POINTER,
3 @NAMES POINTER,
3 @INS POINTER,
3 @PARS POINTER,
3 #DECLARATIONS FIXED BINARY(15),
3 #ACCESSIBLES FIXED BINARY(15);
\end{verbatim}

where the last two fields provide some auxiliary information, useful
in performing the transformations. Each of the terminal fields has
subsumed the analagous arc label. This implies that \texttt{@ENC\_BLOCK}
can be used to locate the structure representing the contour node for
the enclosing block, and that each of \texttt{@VARS}, \texttt{@RTNS}, \texttt{@MODS}, \texttt{@INCS},
\texttt{@NAMES}, \texttt{@INS} and \texttt{@PARS} are used in locating the set of container
nodes within the set. The representation of a subtree of the type
having a branching node as root, selected by one of the set of arc
labels \{\texttt{vars}, \texttt{rtns}, \texttt{mods}, \texttt{incs}, \texttt{names}, \texttt{ins}, \texttt{pars}\} requires storing a
small bush of nodes. Within the program, a linked list has been used
to do this, eliminating from the program a direct analog of this use
of branching node. Each element in the linked list is stored in the
internal representation of a container node, defined by:
Within the NAME substructure, @VAL is used to locate the value associated with a name, if any. The TYPE field is used in dividing the set into subsets where this is appropriate. For example, it can be used to distinguish between simple variables declared within a set of declarations, identifiers included by virtue of an insert transformation, and an identifier to name the local environment. Additional use of the NAME_STRUCTURE storage structure will be described later.

A code node within a graph has been depicted as a monolithic entity. This is reflected within the storage of the graph also; a code node is stored within the structure:

```plaintext
1 CODE,
2 @ENV POINTER,
2 #INST FIXED BINARY(15),
2 INST(N REFER(#INST)),
3 OPCODE BIT(16) ALIGNED,
3 PAD FIXED BINARY(15),
3 @RAND1 POINTER,
3 @RAND2 POINTER;
```

where @ENV is used to locate the stored representation of the contour node associated with the code node being represented.

The information within a graph which describes a module manipulation construct is represented by using NAME_STRUCTUREs. For example, to represent the subgraph which has been used within the graph for

```
p names bind m(,0);
```

a NAME_STRUCTURE in which the ID field contains 'p' exists. In this
case, NAME.VAL is set to point to an additional NAME_STRUCTURE, in
which NAME.ID contains 'm', NAME.TYPE indicates the bind manipulation,
and NAME.VAL points to a reformations structure, REFORMATION. This
last structure, defined by

```
1 REFORMATION,
  2 @PARAMSETS       POINTER,
  2 @RENNAMES        POINTER;
```
is used to locate the information which, in the graph, is selected by the
arcs labeled args and withs, respectively. In this @PARAMSETS would
point to a linked list of structures in which the argument information
is stored; @RENNAMES would, of course, be set to NULL.

In a graph, renaming information is represented by a subtree, the
root of which has a set of integer-labeled arcs emanating from it; each
of which selects an empty node, from which two arcs, labeled old and new
emanate. A set of renames are stored as a linked list, each element of
which contains one rename (i.e., the old and the new identifiers).
Thus, the structure for renaming information has been defined by:

```
1 RENAME_STRUCTURE,
  2 NEXT           POINTER,
  2 RENAME,
    3 ID           CHARACTER(6)
    3 ALIAS        CHARACTER(6);
```

where ALIAS designates the name to be used within the context of the
module manipulation construct within which the renaming was defined.

Argument lists to be used in creating a module, either named or
unnamed, are stored as linked lists, each element of which contains the
label information, designating the position of argument within the list.
Three argument structures have been defined, the use of which is
dependent upon the type of the argument. Using structure mapping techniques and based storage attributes, each of these structures are overlaid so that the "value" field in each, while having different declared attributes, occupy the same storage. The structure for an integral argument is defined by:

1 REFORMATION,  
2 NEXT  
2 REFORM,  
3 TYPE  
3 PARAM  
3 PAD  
3 CONSTANT

POINTER,  
FIXED BINARY(15),  
FIXED BINARY(15),  
FIXED BINARY(31),  
FIXED BINARY(31);

If the argument is an identifier reference, that identifier occupies the last six bytes of the storage allocated for the substructure REFORM, and if the argument is an arithmetic expression, a pointer to the internal representation is stored in the field named CONSTANT in the substructure REFORM.

The method by which an integer value of a variable is stored still needs clarification. Because all of the structures, above, are allocated in the heap, it is possible to define one structure over another. In the graph, only names within the subgraph whose root can be selected by an arc labeled vars can have identifiers having integers as values. The subset of such identifiers is disjoint from the set of identifiers appearing as the result of an include or an insert transformation. Hence, for those identifiers having initial values, this value is stored in place of the pointer NAME.@VAL, and the name is flagged as initialized within NAME.TYPE. NAME.TYPE is also used to distinguish the accessible identifiers from those which are to be shielded from
exposure.

Arithmetic expressions are stored as trees, in which all terminal
nodes are either identifier references or integers; all of the operators
are encoded within the branching nodes of the trees. Only those arith­
metic expressions passed as arguments to modules are evaluated during
the construction phase of model execution.

The Procedures, Briefly

The construction phase of the model, as implemented, consists of
twenty procedures whose functions can be used to form five separate
groups of procedures: (1) the driver, which controls the construction
phase; (2) the eleven procedures used to perform the transformations in
a specified order; (3) two procedures used in copying a module defini­
tion; (4) five procedures used in building the program skeleton; (5) one
procedure used in completing the binding of module parameters to routine
and environment arguments after the construction of the program skeleton.

The driver

The driver performs only a few functions. It serially causes the
routines within the global environment to be traversed, directs the
building of the program skeleton by causing the contour cell for the
global environment to be built, calls for the final state of the binding
of those module parameters to which routines or environments were passed
as arguments, and builds part of the system code, which locates the con­
tour cell within the program skeleton which defines the global environ­
ment, for use by the execution phase of the model.
The transformation performers

The eleven procedures in this category fall naturally into four subcategories: (1) one procedure used to resolve module references, RESOLVE; (2) three routines which perform the actual transformations: INCLUDE, INSERT, and CREATE_MODULE; (3) four which direct the serial application of specific transformations: INCLUSIONS, INSERTIONS, MODULES, and ENVIRONMENTS; (4) three which are used to interrogate and traverse the graph: TRANSFORM_BLOCK, TRANSFORM_ROUTINES, and TRANSFORM_MODULE. Of these, CREATE_MODULE deserves additional comment. CREATE_MODULE is used in performing the is transformations, as well as being used to create any unnamed module, which arises whenever a module reference includes an argument list.

The copiers

Two procedures fall into this category: COPY and BLOCK_COPY. BLOCK_COPY is used to copy the code associated with any block, while interrogating that code for the existence of any nested blocks. COPY is used to copy the complete internal representation of sets of declarations, and also copies the entry-point coding of routines, since this code node is known to contain no nested block, but does contain an easily-located activate instruction.

The program skeleton builders

The program skeleton is built by the efforts of five procedures: BUILD_BLOCK, BUILD_ROUTINE, BUILD_RPC, BUILD_CONTOURS, and SET_LABELS, while traversing the final graph. The last is included because the
construction of a label value requires the proper setting of the ep portion of the value, requiring special handling within the definition of the contour and code cells representing a routine definition within the program skeleton. It is the author's opinion that this could be subsumed within BUILD_CONTOURS, which initializes the declaration sub-cells, if a different order of traversal of the final graph were used in constructing the program skeleton.

One of the functions of this set of procedures is the creation of the block entry-point coding. This is done in BUILD_CONTOURS, which builds the set of horizontally visible environments, and creates any linkages between them. In doing so, BUILD_CONTOURS amasses all of the information to build the required block entry-point coding. Because the formal parameter contour of a routine can have no horizontally visible environments, its contour is created in BUILD_FPC, since the generality of BUILD_CONTOURS is not required.

Setting the parameters

The final procedure is used to complete the binding of a module parameter to a routine or environment argument; as such it requires all of the visible contours to be complete before attempting to perform its function. The present order by which these contours are completed precludes this activity until the program skeleton is completed. SET_PARAMS traverses the program skeleton to locate those names requiring this special attention; the reader will note that this is the only procedure in this phase which traverses the program skeleton.
The Program

The implementation model has been programmed in PL/I and run on an IBM 360/65, using the PL optimizing compiler. The language supported by this compiler offers some features in addition to those provided by the more standard form of PL/I offered by the F-compiler. These are used sparingly, but as necessary. In particular, to facilitate output, all allocations are performed within areas. Using the optimizing compiler, the conversions between pointer and offset are automatically performed. This is heavily used throughout the program, which is presented in the listing which follows.
PERFORM CONSTRUCTION PHASE

DECLARE /* PROGRAM STRUCTURES */
  1 PROGRAM_AREA BASED(@P),
  2 P_LEN FIXED BINARY(15),
  2 PAD CHARACTER(6),
  2 PROGRAM AREA(N REFER(P_LEN)),

  1 PROGRAM_AREA_1 BASED(@P),
  2 PAD CHARACTER(12),
  2 PROGRAM_EXTENT FIXED BINARY(31),

@P POINTER,
@PS Pointer EXTERNAL;

DECLARE /* SCRATCH STRUCTURES */
  1 SCRATCH_AREA BASED(@S),
  2 SC_LEN FIXED BINARY(15),
  2 PAD CHARACTER(6),
  2 SCRATCH AREA(N REFER(SC_LEN)),

  1 SCRATCH_AREA_1 BASED(@S),
  2 PAD CHARACTER(12),
  2 SCRATCH_EXTENT FIXED BINARY(31),

@S POINTER,
@SS Pointer EXTERNAL;

DECLARE BADMOD CONDITION EXTERNAL;
DECLARE BADPROG CONDITION EXTERNAL;

DECLARE /* CODE CELL */
  1 CODE BASED(@CC),
  2 @ENV POINTER,
  2 #INST FIXED BINARY(15),
  2 INST(N REFER(CODE,#INST)),
  3 OP CODE BIT(16) ALIGNED,
  3 PAD FIXED BINARY(15),
  3 @RAND1 POINTER,
  3 @RAND2 POINTER,

@CC POINTER;
DECLARE
1 STARTUP EXTERNAL,
2 @ENV POINTER,
2 #INST FIXED BINARY(15),
2 START; OPCODE BIT(16) ALIGNED,
3 PAD FIXED BINARY(15),
3 &RAND1 POINTER,
3 &RAND2 POINTER;
DECLARE @ESD POINTER EXTERNAL;
DECLARE @ALLOC POINTER EXTERNAL;
DECLARE SDUMP ENTRY(POINTER,FIXED BINARY(31));
DECLARE @EPC POINTER;
DECLARE NULL BUILTIN;

@S=ASS; @P=APS;
@CC=0ESD;

/* RESOLVE THE GLOBAL ENVIRONMENT. */
CALL TRANSFORM_BLOCK(@CC);
@ALLOC=NULL;

/* BUILD THE PROGRAM SKELETON. */
@EPC=BUILD_BLOCK(@CC,NUL1));

/* SET THE VALUES OF THE MODULE PARAMETERS WHICH HAVE BEEN BOUND TO ROUTINE OR ENVIRONMENT ARGUMENTS. */
CALL SET_PARAMS;

/* BUILD THE SYSTEM CODE. */
START.OPCODE=a1000000000000000'B;
START.&RAND1=&EPC;
START.&RAND2=&EPC->CODE.INST(&EPC->CODE.#INST).&RAND2;

STARTU.P.@ENV=NULL; STARTUP.#INST=1;

/* OUTPUT THE PROGRAM SKELETON. */
CALL PLIDUMP("TBNF","AFTER COMPILE");
CALL SDUMP(ADDR(PROGRAM),PROGRAM_EXTENT);
RETURN;

/* END COMPILE; */
PERFORM ANY MODULE MANIPULATIONS WITHIN A BLOCK

TRANSFORM BLOCK:
PROCEDURE(@CC) RECURSIVE:

DECLARE @CC POINTER;

DECLARE /* DECLARATIONS HEADER */
1 DECL_HEADER BASED(@HEADER),
2 @ENC BLOCK POINTER,
2 HEADER,
3 @VARS POINTER,
3 @RNS POINTER,
3 @MODS POINTER,
3 @INS POINTER,
3 @NAMES POINTER,
3 @INS POINTER,
3 @PARS POINTER,
3 #DECLARATIONS FIXED BINARY(15),
3 #ACCESSIBLES FIXED BINARY(15),
@HEADER POINTER;

DECLARE /* NAME STRUCTURE */
1 NAME_NODE BASED(@NAME),
2 LINK POINTER,
2 NAME,
3 TYPE FIXED BINARY(15),
3 PAD FIXED BINARY(15),
3 @MDEF POINTER,
3 @VAL POINTER,
@NAME POINTER;

DECLARE /* CODE CELL */
1 CODE BASED(@CC),
2 @ENV POINTER,
2 #INST FIXED BINARY(15),
2 INST(N REFER(CODE,#INST)),
3 OPCODE BIT(16) ALIGNED,
3 PAD FIXED BINARY(15),
3 @RAND1 POINTER,
3 @RAND2 POINTER;
DECLARE /* INSTRUCTION SUBCELL */

1 INST BASEC(INST),
2 CPDOE BIT(16) ALIGNED,
2 FAD FIXED BINARY(15),
2 RAND1 POINTER,
2 RAND2 POINTER,
@INST POINTER,
@INST FIXED BINARY(31) DEFINED(INST);

DECLARE MOD FIXED BINARY(15);
DECLARE INC FIXED BINARY(15);
DECLARE ENV FIXED BINARY(15);

DECLARE IS_NOT_COMPLETE FIXED BINARY(15) STATIC INITIAL(0);
DECLARE IS_PROGRESSING FIXED BINARY(15) STATIC INITIAL(1);
DECLARE IS_COMPLETE FIXED BINARY(15) STATIC INITIAL(2);

DECLARE IS_INSERT FIXED BINARY(15) STATIC INITIAL(3);
DECLARE LOCAL BIT(1) ALIGNED STATIC INITIAL('1'B);
DECLARE GLOBAL BIT(1) ALIGNED STATIC INITIAL('0'B);

DECLARE BLOCK_ENTRY BIT(16) ALIGNED STATIC INITIAL('100000000000000'B);

DECLARE @THIS POINTER;
DECLARE I FIXED BINARY(15);

@HEADER=CODE.@ENV;

/* CHECK FOR INVALID INSERTIONS. */

IF @INS = NULL THEN SIGNAL CONDITION(BADMOD);

@THIS=@NAMES;

DO WHILE(@THIS = NULL);
    @NAME=@THIS->NAME.@VAL;
    IF NAME.TYPE = IS_INSERT THEN SIGNAL CONDITION(BADMOD);
    @THIS=@THIS->LINK;
END;

/* RESOLVE ALL MODULE REFERENCES. */

CALL RESOLVE(@HEADER);
/* PERFORM THE TRANSFORMATIONS IN THE BLOCK. */

IF @MODS = NULL THEN MOD=IS_COMPLETE; ELSE MOD=IS_PROGRESSING;
IF @INCS = NULL THEN INC=IS_COMPLETE; ELSE INC=IS_PROGRESSING;
IF @NAMES = NULL THEN ENV=IS_COMPLETE; ELSE ENV=IS_PROGRESSING;

DO WHILE ((MOD = IS_PROGRESSING) | (INC = IS_PROGRESSING) | 
   (ENV = IS_PROGRESSING));
   IF MOD -= IS_COMPLETE THEN MOD=MODULES(@HEADER,LOCAL);
   IF INC -= IS_COMPLETE THEN INC=INCLUSIONS(@HEADER,LOCAL);
   IF ENV -= IS_COMPLETE THEN ENV=ENVIRONMENTS(@HEADER,LOCAL);
   END;

IF MOD -= IS_COMPLETE THEN MOD=IS_PROGRESSING;
IF INC -= IS_COMPLETE THEN INC=IS_PROGRESSING;
IF ENV -= IS_COMPLETE THEN ENV=IS_PROGRESSING;

DO WHILE ((MOD = IS_PROGRESSING) | (INC = IS_PROGRESSING) | 
   (ENV = IS_PROGRESSING));
   IF MOD -= IS_COMPLETE THEN MOD=MODULES(@HEADER,GLOBAL);
   IF INC -= IS_COMPLETE THEN INC=INCLUSIONS(@HEADER,GLOBAL);
   IF ENV -= IS_COMPLETE THEN ENV=ENVIRONMENTS(@HEADER,GLOBAL);
   END;

IF (MOD = IS_NOT_COMPLETE) | (INC = IS_NOT_COMPLETE) | 
   (ENV = IS_NOT_COMPLETE) THEN SIGNAL CONDITION(BADMOD);

CALL TRANSFORM_ROUTINES(@HEADER);

/* RESOLVE ANY NESTED BLOCKS. */

@INST=ADDR(CODE,INST(1));

DO I=1 TO CODE,#INST;
   IF INST.OPCODE = BLOCK_ENTRY THEN 
      CALL TRANSFORM_BLCCK(INST.@RAND2);
      @INST=#INST+12;
   END;

RETURN;

END TRANSFORM_BLOCK;
/* PERFORM THE MODULE MANIPULATIONS WITHIN THE BODY OF THE ROUTINES */

TRANSFRM_ROUTINES:
PROCEDURE(@HDR) RECURSIVE;

DECLARE @HDR POINTER;

DECLARE /* DECLARATIONS HEADER */
1 DECL_HEADER BASED(@HEADER),
2 @ENC_BLOCK POINTER,
2 HEADER,
3 @VARS POINTER,
3 @RINS POINTER,
3 @MODS POINTER,
3 @INS POINTER,
3 @NAMES POINTER,
3 @INS POINTER,
3 @PARS POINTER,
3 @DECLARATIONS FIXED BINARY(15),
3 #ACCESSIBLES FIXED BINARY(15),
@HEADER POINTER;

DECLARE /* NAME STRUCTURE */
1 NAME_NODE BASED(@NAME),
2 LINK POINTER,
2 NAME,
3 TYPE FIXED BINARY(15),
3 ID CHARACTER(6),
3 @VAL POINTER,
@NAME POINTER;

DECLARE /* CODE CELL */
1 CODE BASED(@CC),
2 @ENV POINTER,
2 #INST FIXED BINARY(15),
2 INST(N REFER(CODE,#INST)),
3 OPCODE BIT(16) ALIGNED,
3 PAC CHARACTER(6),
3 @RAND POINTER,
@CC POINTER;
@NAME=@HDR->@RINS;

DO WHILE(@NAME ^= NULL);
   @CC=NAME.@VAL;
   @HEADER=CODE.@ENV;
   IF @VARS ^= NULL THEN SIGNAL CONDITION(BADMOD);
   IF @RINS ^= NULL THEN SIGNAL CONDITION(BADMOD);
   IF @MODS ^= NULL THEN SIGNAL CONDITION(BADMOD);
   IF @INCS ^= NULL THEN SIGNAL CONDITION(BADMOD);
   IF @NAME ^= NULL THEN SIGNAL CONDITION(BADMOD);
   IF @INS ^= NULL THEN SIGNAL CONDITION(BADMOD);
   IF #ACCESSIBLES ^= 0 THEN SIGNAL CONDITION(BADMOD);
   @CC=CODE.@INST(CODE.@INST).@RAND;
   CALL TRANSFORM_BLOCK(@CC);
   @NAME=NAME.LINK;
END;

RETURN;

END TRANSFORM_ROUTINES;
PERFORM THE MODULE MANIPULATIONS WITHIN MODULE DEFINITIONS

TRANSFORM MODULE:
PROCEDURE(@HDR) RECURSIVE:

DECLARE @HDR POINTER;

DECLARE /* DECLARATIONS HEADER */
1 DECL HEADER
2 ENC BLOCK
3 HEADER,
4 @VARS POINTER,
5 @RNS POINTER,
6 @MODS POINTER,
7 @INCS POINTER,
8 @NAMES POINTER,
9 @INS POINTER,
10 @PARS POINTER,
11 #DECLARATIONS FIXED BINARY(15),
12 #ACCESSIBLES FIXED BINARY(15),
@HEADER POINTER;

DECLARE /* NAME STRUCTURE */
1 NAME_NODE BASED(@NAME),
2 LINK POINTER,
3 NAME,  
4 TYPE FIXED BINARY(15),
5 PAD FIXED BINARY(15),
6 @MDEF POINTER,
7 @VAL POINTER,
@NAME POINTER;

DECLARE INC FIXED BINARY(15);
DECLARE INS FIXED BINARY(15);
DECLARE ENV FIXED BINARY(15);

DECLARE IS_NOT_COMPLETE FIXED BINARY(15) STATIC INITIAL(0);
DECLARE IS_PROGRESSING FIXED BINARY(15) STATIC INITIAL(1);
DECLARE IS_COMPLETE FIXED BINARY(15) STATIC INITIAL(2);

DECLARE LOCAL BIT(1) ALIGNED STATIC INITIAL('*1'8);
DECLARE GLOBAL BIT(1) ALIGNED STATIC INITIAL('*0'8);
DECLARE @THIS POINTER;
/* CHECK FOR INVALID MODULE DEFINITIONS. */

IF @MODS IS NULL THEN SIGNAL CONDITION(BADMOD);

/* PERFORM THE TRANSFORMATIONS IN THE MODULE DEFINITION. */

IF @INCS IS NULL THEN INC=IS_COMPLETE; ELSE INC=IS_PROGRESSING;
IF @INS IS NULL THEN INS=IS_COMPLETE; ELSE INS=IS_PROGRESSING;
IF @NAMES IS NULL THEN ENV=IS_COMPLETE; ELSE ENV=IS_PROGRESSING;

DO WHILE ((INC = IS_PROGRESSING) OR (INS = IS_PROGRESSING) OR (ENV = IS_PROGRESSING));

IF INC IS COMPLETE THEN INC=INCLUSIONS(@HEADER, LOCAL);
IF INS IS COMPLETE THEN INS=INSERTIONS(@HEADER, LOCAL);
IF ENV IS COMPLETE THEN ENV=ENVIRONMENTS(@HEADER, LOCAL);

END;

IF INC IS COMPLETE THEN INC=IS_PROGRESSING;
IF INS IS COMPLETE THEN INS=IS_PROGRESSING;
IF ENV IS COMPLETE THEN ENV=IS_PROGRESSING;

DO WHILE ((INC = IS_PROGRESSING) OR (INS = IS_PROGRESSING) OR (ENV = IS_PROGRESSING));

IF INC IS COMPLETE THEN INC=INCLUSIONS(@HEADER, GLOBAL);
IF INS IS COMPLETE THEN INS=INSERTIONS(@HEADER, GLOBAL);
IF ENV IS COMPLETE THEN ENV=ENVIRONMENTS(@HEADER, GLOBAL);

END;

IF (INC IS NOT COMPLETE) OR (INS IS NOT COMPLETE) OR (ENV IS NOT COMPLETE) THEN SIGNAL CONDITION(BADMOD);

CALL TRANSFORM_ROUTINES(@HEADER);

RETURN;

END TRANSFORM_MODULE;
CREATE LOCAL MODULE DEFINITIONS

MODULES;

PROCEDURE(@HDR, LOCAL_ONLY) RETURNS(FIXED BINARY(15)) RECURSIVE;

DECLARE /* PARAMETERS */
@HDR
LOCALONLY
POINTER,
BIT(1) ALIGNED;

DECLARE /* DECLARATIONS HEADER */
1 DECL_HEADER
BASED(@HEADER),
2 EEC_BLOCK
POINTER,
HEADER,
3 @VARS
POINTER,
3 @RINS
POINTER,
3 @MODS
POINTER,
3 @INC
POINTER,
3 @NAMES
POINTER,
3 @INS
POINTER,
3 @PARS
POINTER,
3 #DECLARATIONS
FIXED BINARY(15),
3 #ACCESSIBLES
FIXED BINARY(15),
@HEADER
POINTER;

DECLARE /* NAME STRUCTURE */
1 NAME_NODE
BASED(@NAME),
2 LINK
POINTER,
2 NAME,
3 TYPE
FIXED BINARY(15),
3 PAD
FIXED BINARY(15),
3 @DEF
POINTER,
3 @VAL
POINTER,
@NAME
POINTER;

DECLARE /* REFCRM STRUCTURE */
1 REFORM_STRUCTURE
BASED(@REFCRMS),
2 @PARAMSETS
POINTER,
2 @RENAME
POINTER,
@REFCRMS
POINTER;

DECLARE IS IMPLICIT
FIXED BINARY(15) STATIC INITIAL(0);
DECLARE IS EXPLICIT
FIXED BINARY(15) STATIC INITIAL(1);
DECLARE IS NOT COMPLETE
FIXED BINARY(15) STATIC INITIAL(0);
DECLARE IS PROGRESSING
FIXED BINARY(15) STATIC INITIAL(1);
DECLARE IS COMPLETE
FIXED BINARY(15) STATIC INITIAL(2);
DECLARE DONE
FIXED BINARY(15);
DECLARE TOTAL
FIXED BINARY(15);
DECLARE RESULT FIXED BINARY(15);
DECLARE @MOD POINTER;
DECLARE @THIS POINTER;
DECLARE @DEF POINTER;
DECLARE @PARAMS POINTER;
DECLARE NEW_MOD BIT(1) ALIGNED STATIC INITIAL('1'8);

@HEADER=@HDR;
DONE.TOTAL=0;
@MOD=@MODS;
DO WHILE (@MOD -= NULL);
   @THIS=@MOD->NAME.@VAL;
   IF @THIS->NAME.TYPE = IS_EXPLICIT THEN DO;
   /* A MODULE DEFINED BY AN IS CCNSTRCT HAS BEEN FOUND. */
      TOTAL=TOTAL+1;
      @NAME=@THIS->@MDEF;
      IF NAME.TYPE = IS_EXPLICIT THEN DO;
      @PARAMS=@THIS->NAME.@VAL;
      IF @PARAMS -= NULL THEN
           IF @RENAME = NULL THEN SIGNAL CONDITION(BADMOD);
      ELSE @PARAMS = @PARAMSETS;
      ELSE @PARAMS = NULL;
      @DEF=CREATE_MODULE(@MDEF,@PARAMS,@HEADER,LOCAL_ONLY,NEW_MOD);
      IF @DEF = NULL THEN DC;
      DONE=DONE+1;
      @THIS->@MDEF=@DEF;
      @THIS->NAME.TYPE=IS_EXPLICIT;
      ENC;
   END;
   @MOD = @MOD->LINK;
END;
IF DONE = TOTAL THEN RESULT=IS_COMPLETE;
ELSE IF DONE = 0 THEN RESULT=IS_NOT_COMPLETE;
ELSE DONE=IS_PROGRESSING;
RETURN(Result);
END MODULES;
/* DIRECT THE PERFORMANCE OF THE INCLUDE TRANSFORMATIONS */

INCLUSIONS:

PROCEDURE(@HDR, LOCAL_ONLY) RETURNS(FIXED BINARY(15)) RECURSIVE;

DECLARE /* PARAMETERS */
@HDR POINTER,
LOCAL_ONLY BIT(*) ALIGNED;

DECLARE /* DECLARATIONS HEADER */
1 DECL_HEADER BASEC(@HEADER),
2 @ENC_BLOCK POINTER,
3 HEADER POINTER,
4 @VARS POINTER,
5 @RINS POINTER,
6 @MODS POINTER,
7 @INCS POINTER,
8 @INS POINTER,
9 @INSPOINTER,
10 #DECLARATIONS FIXED BINARY(15),
11 #ACCESSIBLES FIXED BINARY(15),
@HEADER POINTER;

DECLARE /* NAME STRUCTURE */
1 NAME_NODE BASEC(@NAME),
2 LINK POINTER,
3 NAME POINTER,
4 TYPE FIXED BINARY(15),
5 PAD FIXED BINARY(15),
6 @MDEF POINTER,
7 @VAL POINTER,
8 @NAME POINTER;

DECLARE
1 REFORMS_STRUCTURE BASEC(@REFORMS),
2 @PARAMSETS POINTER,
3 @RENAMES POINTER,
4 @REFORMS POINTER;
DECLARE
1 PARAMSET
2 NEXT
2 REFORM,
3 TYPE
3 FARAM#
3 PAD
3 CONSTANT
@REFORM
DECLARE IS_NOT_DONE
DECLARE IS_DONE
DECLARE IS_EXPLICIT
DECLARE IS_NOT_COMPLETE
DECLARE IS_PROGRESSING
DECLARE IS_COMPLETE
DECLARE RESULT
DECLARE DONE
DECLARE TOTAL
DECLARE @THIS
DECLARE @PARAMS
DECLARE @RENAME
DECLARE @DEF
DECLARE @MOD
DECLARE OLD_MOD
DECLARE @HEADER
DECLARE @HDR;
DECLARE @NAME=SINCS;
DECLARE IS_COMPLETE
DECLARE @HEADER
DECLARE @HDR;
DECLARE @NAME=SINCS;
CC WHILE(@NAME != NULL);

    IF @NAME.TYPE = IS_NCT_DOME THEN CC;
    TOTAL=TOTAL+1;
    @THIS=@MDEF;

    IF @THIS->NAME.TYPE = IS_EXPLICIT THEN CC;
    @REFORMS=NAME@VAL;
    IF @REFORMS = NULL THEN DO;
        @PARAMS=@RENAMES=NULL;
    END;
    ELSE DO;
        @PARAMS=@PARAMSETS;
        @RENAMES=@RENAMES;
    END;

    @THIS=@THIS->@MDEF;

    @MCD=CREATE_MODULE(@THIS,@PARAMS,@HEADER,LOCAL_ONLY,OLD_MOD);

    IF @MOD = NULL THEN DO;
        /* CREATE THE COPY OF THE REFERENCED MODULE. */
        @DEF=COPY(@MOD);
        @DEF->@ENC_BLOCK=@MOD->@ENC_BLOCK;
        /* PERFORM ANY TRANSFORMATIONS WITHIN THE REFERENCED MODULE. */
        CALL RESOLVE(@DEF);
        @DEF->@ENC_BLOCK=NULL;
        CALL TRANSFORM_MODULE(@DEF);
        /* PERFORM THE INCLUDE TRANSFORMATION. */
        CALL INCLUDE(@DEF, @RENAMES, @HEADER);
        @MDEF=@DEF;
        NAME.TYPE=IS_CCNE;
        DONE=DONE+1;
        ENC;
    END;

    @NAME=LINK;

    END;

    IF DONE = TOTAL THEN RESULT=IS_COMPLETE;
    ELSE IF DONE = 0 THEN RESULT=IS_NOT_COMPLETE;
    ELSE RESULT=IS_PROGRESSING;

    RETURN(RESULT);

END INCLUSIONS;
DIRECT THE PERFORMANCE OF THE INSERT TRANSFORMATIONS

INSERTIONS:
PROCEDURE(@HDR,LOCAL_ONLY) RETURNS(FIXED BINARY(15)) RECURSIVE;

DECLARE /* PARAMETERS */
@HDR POINTER,
LOCALONLY BIT(*) ALIGNED;

DECLARE /* DECLARATIONS HEADER */
1 DECL_HEADER BASED(@HEADER),
2 ENC_BLOCK POINTER,
3 HEADER POINTER,
4 @VARS POINTER,
5 @RTNS POINTER,
6 @MODS POINTER,
7 @INCS POINTER,
8 @NAMES POINTER,
9 @INS POINTER,
10 @PARS POINTER,
11 #DECLARATIONS FIXED BINARY(15),
12 #ACCESSIBLES FIXED BINARY(15),
@HEADER POINTER;

DECLARE /* NAME STRUCTURE */
1 NAME_NODE BASED(@NAME),
2 LINK POINTER,
3 NAME POINTER,
4 TYPE FIXED BINARY(15),
5 PAD FIXED BINARY(15),
6 @MDEF POINTER,
7 @VAL POINTER,
@NAME POINTER;

DECLARE
1 REFORMS_STRUCTURE BASED(@REFORMS),
2 @PARAMSETS POINTER,
3 @RENAMEs POINTER,
@REFORMS POINTER;
DECLARE
1 PARAMSET BASED(SREFORM),
2 NEXT POINTER,
2 REFORM,
3 TYPE FIXED BINARY(15),
3 PARAM# FIXED BINARY(15),
3 PAD FIXED BINARY(31),
3 CONSTANT FIXED BINARY(31),
SREFORM POINTER.

DECLARE IS_NOT_DONE FIXED BINARY(15) STATIC INITIAL(0);
DECLARE IS_DONE FIXED BINARY(15) STATIC INITIAL(1);
DECLARE IS_EXPLICIT FIXED BINARY(15) STATIC INITIAL(1);
DECLARE IS_NOT_COMPLETE FIXED BINARY(15) STATIC INITIAL(0);
DECLARE IS_PROGRESSING FIXED BINARY(15) STATIC INITIAL(1);
DECLARE IS_COMPLETE FIXED BINARY(15) STATIC INITIAL(2);
DECLARE RESULT FIXED BINARY(15);
DECLARE DONE FIXED BINARY(15);
DECLARE TOTAL FIXED BINARY(15);
DECLARE THIS POINTER;
DECLARE @PARAMS POINTER;
DECLARE @RENAME POINTER;
DECLARE @DEF POINTER;
DECLARE @MOD POINTER;
DECLARE CLD_MCD BIT(1) ALIGNED STATIC INITIAL("0B");

@HEADER=@HDR;
DONE,TOTAL=0;
@NAME=INS;
CC WHILE(@NAME ≠ NULL):

    IF NAME.TYPE = IS_NCT_DONE THEN DO:
        TOTAL=TOTAL+1;
        @THIS=@MDEF;

        IF @THIS->NAME.TYPE = IS_EXPLICIT THEN DC:
            @REFORMS=NAME.@VAL;
            IF @REFORMS = NULL THEN DO:
                @PARAMS.@RENAME=NULL;
                END;
            ELSE DO:
                @PARAMS=@PARAMSETS;
                @RENAME=@RENAES;
                END;
            @THIS=@THIS->@MDEF;

            @MOD=CREATE_MODULE(@THIS.@PARAMS.@HEADER.LOCAL_CNLY.OLC_MOD);

            IF @MOD = NULL THEN DO:
                /* CREATE THE COPY OF THE REFERENCED MODULE. */
                @DEF=COPY(@MOD);
                @DEF->@ENC_BLOCK=@MOD->@ENC_BLOCK;
                /* PERFORM ANY TRANSFORMATIONS WITHIN THE REFERENCED MODULE. */
                CALL RESOLVE(@DEF);
                @DEF->@ENC_BLOCK=NULL;
                CALL TRANSFORM_MODULE(@DEF);
                /* PERFORM THE INCLUDE TRANSFORMATION. */
                CALL INSERT@aDEF.@RENAME.@HEADER);
                @MDEF=@DEF;
            NAME.TYPE=IS_DONE;
            DONE=DONE+i;
            ENC;
            END;
            END;
        END;

    @NAME=LINK;
    END;

    IF DONE = TOTAL THEN RESULT=IS_COMPLETE;
    ELSE IF CONE = 0 THEN RESULT=IS_NOT_COMPLETE;
    ELSE RESULT=IS_PROGRESSING;
    RETURN(Result);
    END INSERTIONS;
PERFORM THE ENVIRONMENT DEFINITION TRANSFORMATIONS

PROCEDURE (@HDR, LOCAL_ONLY) RETURNS (FIXED BINARY(15)) RECURSIVE;

DECLARE /* PARAMETERS */
@HDR POINTER,
LOCAL_ONLY BIT(*) ALIGNED;

DECLARE /* DECLARATIONS HEADER */
DECL_HEADER BASED (@HEADER),
@ENC_BLOCK POINTER,
@HEADER,
@VARS POINTER,
@RTNS POINTER,
@MODS POINTER,
@INCS POINTER,
@NAMES POINTER,
@INS POINTER,
@PARS POINTER,
#DECLARATIONS FIXED BINARY(15),
#ACCESSIBLES FIXED BINARY(15),
@HEADER POINTER;

DECLARE /* NAME STRUCTURE */
NAME_NODE BASED (@NAME),
LINK POINTER,
NAME,
@NAME POINTER,
@TYPE FIXED BINARY(15),
@PAD FIXED BINARY(15),
@MDEF POINTER,
@VAL POINTER;

DECLARE
REFORMS_STRUCTURE BASED (@REFORM),
@PARAMSETS POINTER,
@RENAMES POINTER,
@REFORMS POINTER;

DECLARE
PARAMSET BASED (@REFORM),
NEXT POINTER,
REFORM,
@TYPE FIXED BINARY(15),
@PARAM# FIXED BINARY(15),
@PAD FIXED BINARY(31),
@CONSTANT FIXED BINARY(31),
@REFORM POINTER;
DECLARE IS_DONE FIXED BINARY(15) STATIC INITIAL(0);
DECLARE IS_EXPLICIT FIXED BINARY(15) STATIC INITIAL(1);
DECLARE NAMES_BIND FIXED BINARY(15) STATIC INITIAL(1);
DECLARE NAMES_INCLUDE FIXED BINARY(15) STATIC INITIAL(2);
DECLARE NAMES_INSERT FIXED BINARY(15) STATIC INITIAL(3);
DECLARE IS_NOT_COMPLETE FIXED BINARY(15) STATIC INITIAL(0);
DECLARE IS_PROGRESSING FIXED BINARY(15) STATIC INITIAL(1);
DECLARE IS_COMPLETE FIXED BINARY(15) STATIC INITIAL(2);
DECLARE DONE FIXED BINARY(15); DECLARE TOTAL FIXED BINARY(15);
DECLARE RESULT FIXED BINARY(15);
DECLARE @THIS POINTER; DECLARE @PARAMS POINTER;
DECLARE @RENAME POINTER; DECLARE @DEF POINTER;
DECLARE @MOD POINTER;
DECLARE OLD_MOD BIT(1) ALIGNED STATIC INITIAL("0’B);

@HEADER=@HDR;
DONE,TOTAL=O;
@THIS=@NAMES;
DO WHILE (@THIS := NULL);
@NAME = @THIS->NAME.@VAL;

IF NAME.TYPE := IS_DONE THEN DC;
TOTAL = TOTAL + 1;
@DEF = @MDEF;

IF @DEF->NAME.TYPE := IS_EXPLICIT THEN DC;
@REFORMS = @NAME.@VAL;
IF @REFORMS := NULL THEN DO;
@PARAMS, @RENAME = NULL;
END;
ELSE DO;
@PARAMS = @PARAMSETS;
@RENAME = @RENAMES;
END;
@MOD = CREATE_MODULE(@DEF->@MDEF, @PARAMS, @HEADER, LOCAL_ONLY, OLD_MOD);
IF @DEF := NULL THEN DO;
/* CREATE THE FULLY-DEFINED MODULE FROM WHICH THE ENVIRONMENT IS TO BE CREATED, THEN PERFORM THE DESIRED TRANSFORMATION. */
@DEF = COPY(@MCD);
@DEF->@ENC_BLOCK = @MOD->@ENC_BLOCK;
CALL RESOLVE(@DEF);
@DEF->@ENC_BLOCK = NULL;
CALL TRANSFORM_MODULE(@DEF);
IF NAME.TYPE := NAMES_BIND THEN;
ELSE IF NAME.TYPE := NAMES_INCLUDE THEN
CALL INCLUDE(@DEF, @RENAME, @HEADER);
ELSE IF NAME.TYPE := NAMES_INSERT THEN
CALL INSERT(@DEF, @RENAME, @HEADER);
ELSE SIGNAL CONDITION(BADMOD);
@MDEF = @DEF;
NAME.TYPE = IS_DONE;
DONE = DONE + 1;
END;
END;
END;
@THIS = @THIS->LINK;
END;

IF DONE = TOTAL THEN RESULT = IS_COMPLETE;
ELSE IF DONE = 0 THEN RESULT = IS_NOT_COMPLETE;
ELSE RESULT = IS_PROGRESSING;
RETURN(Result);

END ENVIRONMENTS;
/* CREATE MODULE DEFINITION, EVALUATING ARGUMENTS IN THE REFERENCE */

CREATE MODULE:
PROCEDURE(aDEF,aPARAMSETS,aENV,LOCAL_ONLY,NEW_MOD) RETURNS(POINTER) RECURSIVE;

DECLARE /* PARAMETERS */
aDEF POINTER,
aPARAMSETS POINTER,
aENV POINTER,
LOCAL_ONLY BIT(*) ALIGNED,
NEW_MOD BIT(*) ALIGNED;

DECLARE /* DECLARATIONS HEADER */
DECL HEADER BASED(SHEADER),
aENC BLOCK POINTER,
aVARS POINTER,
RTNS POINTER,
MODS POINTER,
INCS POINTER,
NAMES POINTER,
INS POINTER,
PARS POINTER,
DECLARATIONS FIXED BINARY(15),
DECLARATIONS FIXED BINARY(15),
DECL HEADER POINTER;

DECLARE /* CONTOUR CELL */
CONTOUR BASED(SCONTOUR),
ENV_LINK POINTER,
ANT_LINK POINTER,
SP POINTER,
DECLS FIXED BINARY(15),
CECLS(N REFER(CONTOUR,DECLS)),
TYPE FIXED BINARY(15),
ID CHARACTER(6),
SIMPLE_VALUE FIXED BINARY(31),
VALUE POINTER,
CONTOUR POINTER;

DECLARE /* DECLARATION SUBCELL */
DECL BASED(SDECL),
TYPE FIXED BINARY(15),
ID CHARACTER(6),
SIMPLE_VALUE FIXED BINARY(31),
VALUE POINTER,
DECL POINTER,
DECL FIXED BINARY(31) DEFINED(SDECL);
DECLARE /* NAME STRUCTURES */
1 NAME_NODE
2 LINK
3 NAME
3 TYPE
3 ID
3 VAL
1 NAME_NODE_1
2 PAD
3 NAME
3 TYPE
3 PAD
3 SVAL
1 NAME_NODE_2
2 PAD
3 NAME
3 PAD
3 INITIAL_VALUE
@NAME
DECLARE /* REFORM STRUCTURES */
1 REFORMATION
2 NEXT
3 REFORM
3 TYPE
3 PARAM
3 PAD
3 CONSTANT
1 REFORMATION_1
2 PAD
3 REF
3 PAD
3 ID
1 REFORMATION_2
2 PAD
3 EXPRESSION
3 PAD
3 expr
@REFORM
DECLARE
  1 LINKED BASIS(@LINK),
  2 @MOD POINTER,
  2 @DECL POINTER,
@LINK PINTER;
DECLARE
  IS_UNDEFINED FIXED BINARY(15) STATIC INITIAL(0);
  IS_SIMPLE FIXED BINARY(15) STATIC INITIAL(1);
  IS_ROUTINE_PARAM FIXED BINARY(15) STATIC INITIAL(8);
  IS_ENV_PARAM FIXED BINARY(15) STATIC INITIAL(9);
DECLARE
  IS_NOT_ACCESSIBLE FIXED BINARY(15) STATIC INITIAL(0);
  NAMES_LCL FIXED BINARY(15) STATIC INITIAL(3);
  NAMES_LCL_ACCESSIBLE FIXED BINARY(15) STATIC INITIAL(4);
  IS_INITIALIZED FIXED BINARY(15) STATIC INITIAL(5);
DECLARE
  IS_CONSTANT FIXED BINARY(15) STATIC INITIAL(1);
  DECLARE IS_VARIABLE FIXED BINARY(15) STATIC INITIAL(2);
  DECLARE IS_EXPRESSION FIXED BINARY(15) STATIC INITIAL(3);
DECLARE
  IDENT CHARACTER(6);
DECLARE @DECL POINTER;
DECLARE @THIS POINTER;
DECLARE @PARM# FIXED BINARY(15);
DECLARE TYPE FIXED BINARY(15);
DECLARE I FIXED BINARY(15);
DECLARE N FIXED BINARY(15);
ALLOCATE DECL_HEADER IN(SCRATCH);
  HEADER=@DEF->HEADER;
  IF NEW_MCD THEN @ENC_BLOCK=@ENV; ELSE @ENC_BLOCK=@DEF->@ENC_BLOCK;
  IF @PARAMSETS = NULL THEN RETURN(@HEADER);
/* TRY TO SET THE PARAMETERS */
  IF @DEF->@PARS = NULL THEN SIGNAL CONDITION(BADMOD);
  @DECL.@NAME=@DEF->@PARS;
  DO N=0 BY 1 WHILE(@DECL -> NULL); @DECL=@DECL->LINK; END;
  ALLOCATE CONTOUR IN(SCRATCH);
  @DECL=ADDR(CONTOUR.@DECLS(N));
  DO WHILE(@NAME -> NULL);
    DECL.TYPE=IS_UNDEFINED;
    DECL.ID=NAME.ID;
    @DECL=@DECL-16;
    @NAME=LINK;
  END;
&REFORM = &PARAM_SETS;
DO WHILE (&REFORM != NULL);
  PARM# = &REFORM. PARM#;
  IF (PARM# < 1) THEN SIGNAL CONDITION (BADMCD);
  &PDECL = ADDR (CCNTOUR. DECLS (PARM#));
  TYPE = &REFORM. TYPE;
  IF TYPE = IS_CONSTANT THEN DO;
    &PDECL -> DECL. TYPE = IS_SIMPLE;
    &PDECL -> DECL. SIMPLE_VALUE = &REFORM. CONSTANT;
  END;
  ELSE IF TYPE = IS_VARIABLE THEN DO;
    IDENT = REF.ID;
    /* TRY TO LOCATE THIS IDENTIFIER. */
    &THIS = &ENV;
    DO WHILE (&THIS != NULL);
      &NAME = &THIS. VARS;
      DO WHILE (&NAME != NULL);
        IF NAME. ID = IDENT THEN DO:
          TYPE = NAME. TYPE;
          IF TYPE = IS_INITIALIZED THEN DO;
            &PDECL -> DECL. TYPE = IS_SIMPLE;
            &PDECL -> DECL. SIMPLE_VALUE = INITIAL_VALUE;
            GOTO NEXT_REFORM;
          END;
          ELSE IF (TYPE = NAMES_LCL) THEN DO;
            &PDECL -> DECL. TYPE = IS_ENV_PARAM;
            UNSPEC (&PDECL -> DECL. SIMPLE_VALUE) = UNSPEC (&THIS);
            &PDECL -> DECL. VALUE = &NAME;
            GOTO NEXT_REFORM;
          END;
          ELSE IF (TYPE = IS_ROUTINE_PARAM) THEN DO;
            &PDECL -> DECL. TYPE = TYPE;
            UNSPEC (&PDECL -> DECL. SIMPLE_VALUE) = UNSPEC (&THIS);
            &PDECL -> DECL. VALUE = &NAME;
            GOTO NEXT_REFORM;
          END;
        ELSE SIGNAL CONDITION (BADMCD);
        END;
        &NAME = LINK;
      END;
    END;
  END;
END;
@NAME=@THIS->@RTNS;
DO WHILE(@NAME != NULL);
  IF NAME.ID = IDENT THEN DO;
    @PODECL->DECL.TYPE=IS_ROUTINE_PARAM;
    UNSPEC(@PODECL->DECL.SIMPLE_VALUE)=UNSPEC(@THIS);
    @PODECL->DECL.VALUE=@NAME;
    GOTO NEXT_REFORM;
  END;
  @NAME=LINK;
END;
@NAME=@THIS->@NAMES;
DO WHILE(@NAME != NULL);
  IF NAME.ID = IDENT THEN DO;
    @PODECL->DECL.TYPE=IS_ENV_PARAM;
    UNSPEC(@PODECL->DECL.SIMPLE_VALUE)=UNSPEC(@THIS);
    @PODECL->DECL.VALUE=@NAME;
    GOTO NEXT_REFORM;
  END;
  @NAME=LINK;
END;
@NAME=@THIS->@MODS;
DO WHILE(@NAME != NULL);
  IF NAME.ID = IDENT THEN SIGNAL CCNDITION(BADMCD);
  @NAME=LINK;
END;
  IF LOCAL_ONLY THEN GOTO FAIL;
@THIS=@THIS->@ENC_BLOCK;
END;
SIGNAL CCNDITION(EADMCD);
END;
ELSE IF TYPE = IS_EXPRESSION THEN DO;
  @NAME=@VARS;
  @PODECL->DECL.SIMPLE_VALUE=TRVERSE(@EXPR);
  @PODECL->DECL.TYPE=IS_SIMPLE;
END;
ELSE SIGNAL CCNDITION(BADMCD);
NEXT_REFCRM:
  @REFCRM=NEXT;
END;
/* ALL PARAMETER RESOLUTIONS PERFORMED. */
$PARS=NULL ;
$DECL=ADDR(DECLS(1)) ;
DO I=1 TO N;
   ALLOCATE NAME_NODE IN(SCRATCH);
   NAME.ID=DECL.ID;
   IF DECL.TYPE = IS_UNDEFINED THEN DC;
      LINK=$PARS; $PARS=NAME;
      END;
   ELSE IF DECL.TYPE = IS_SIMPLE THEN DO;
      NAME.TYPE=IS_INITIALIZED;
      INITIAL_VALUE=DECL.SIMPLE.VALUE;
      LINK=$VARS; $VARS=NAME;
      END;
   ELSE IF DECL.TYPE = IS_ROUTINE_PARAM THEN DO;
      NAME.TYPE=DECL.TYPE;
      ALLOCATE LINKED IN(SCRATCH);
      UNSPEC(LINKED.MOD)=UNSPEC(DECL.SIMPLE.VALUE);
      LINKED.DECL=DECL.VALUE;
      NAME.value=LINK;
      LINK=$VARS; $VARS=NAME;
      END;
   ELSE DO:
      NAME.TYPE=DECL.TYPE;
      ALLOCATE LINKED IN(SCRATCH);
      UNSPEC(LINKED.MOD)=UNSPEC(DECL.SIMPLE.VALUE);
      LINKED.DECL=DECL.VALUE;
      NAME.value=LINK;
      LINK=$VARS; $VARS=NAME;
      END;
      $DECL=$DECL+16;
      END;
      FREE CONTEXT IN(SCRATCH);
      RETURN($HEADER);
FAIL:
   FREE CONTEXT IN(SCRATCH), DECL_HEADER IN(SCRATCH);
   RETURN(NULL);
/* EVALUATE ARITHMETIC EXPRESSION TREE */

TRVERSE:
PROCEDURE(@TREE) RETURNS(FIXED BINARY(31)) RECURSIVE;

DECLARE /* PARAMETERS */
@TREE POINTER;

DECLARE /* ARITHMETIC EXPRESSION TREE NODES */
1 OPERATOR_NODE BASED(@NODE),
2 OP FIXED BINARY(15),
2 PAD FIXED BINARY(15),
2 LPART POINTER,
2 RPART PCINTER,

1 TERMINAL_NODE BASED(@NODE),
2 NODE_TYPE FIXED BINARY(15),
2 PAD FIXED BINARY(15),
2 @REFERENCE PCINTER,
2 CONSTANT FIXED BINARY(31),

@NODE POINTER;

DECLARE /* IDENTIFIER */
1 REF_CELL BASED(@REF),
2 REF_OP FIXED BINARY(15),
2 ENVID CHARACTER(6),
2 IDENT CHARACTER(6),
@REF POINTER;

DECLARE /* NAME STRUCTURE */
1 NAME_NODE BASED(@NAME),
2 LINK POINTER,
2 NAME,
3 TYPE FIXED BINARY(15),
3 ID CHARACTER(6),
3 INITIAL_VALUE FIXED BINARY(31),
@NAME POINTER;

DECLARE IS_INITIALIZED FIXED BINARY(15) STATIC INITIAL(4);
DECLARE VALUE FIXED BINARY(31) STATIC;
DECLARE LVALUE FIXED BINARY(31);
DECLARE RVALUE FIXED BINARY(31);
DECLARE IDENTIFIER CHARACTER(6);
DECLARE @HDR POINTER STATIC;
DECLARE NEGATION_OP FIXED BINARY(15) STATIC INITIAL(2);
DECLARE ADD_OP FIXED BINARY(15) STATIC INITIAL(3);
DECLARE SUB_OP FIXED BINARY(15) STATIC INITIAL(4);
DECLARE MUL_OP FIXED BINARY(15) STATIC INITIAL(5);
DECLARE DIV_OP FIXED BINARY(15) STATIC INITIAL(6);
DECLARE TERMINAL_FLAG FIXED BINARY(15) STATIC INITIAL(1);
DECLARE MIN_OP FIXED BINARY(15) STATIC INITIAL(0);
DECLARE MAX_OP FIXED BINARY(15) STATIC INITIAL(6);
DECLARE OPERATOR FIXED BINARY(15);
DECLARE IS_CONSTANT FIXED BINARY(15) STATIC INITIAL(0);
@CODE=TREE; OPERATOR=OP;

IF (MIN OP <= OPERATOR) && (OPERATOR <= TERMINAL_FLAG) THEN DO;
   /* PROCESS TERMINAL NODE. */
   IF NODE_TYPE = IS_CONSTANT THEN VALUE=CONSTANT;
   ELSE DO:
      /* PROCESS VARIABLE REFERENCE. */
      @REF=@REFERENCE;
      IF REF_OP -= 0 THEN SIGNAL CONDITION(EADMOD);
      IDENTIFIER=IDENT; @HDR=@HEADER;
      DO WHILE(@HDR -= NULL);
         @NAME=@HDR->@VARS;
      DC WHILE(0 NAME -= NULL);
      IF NAME.ID = IDENTIFIER THEN
         IF NAME.TYPE = IS_INITIALIZED THEN DO:
            VALUE=NAME.INITIAL_VALUE;
            GOTO DCNE;
         ELSE SIGNAL CONDITION(BADMOD);
         END;
      ELSE SIGNAL CONDITION(BADMOD);
      END;
      IF LOCALONLY THEN GOTO FAIL;
      @HDR=@HDR->@ENC_BLOCK;
      END;
      SIGNAL CONDITION(BADMOD);
   END;
   ELSE IF OPERATOR -= MAX_CP THEN DO:
      LVALUE=TRAVERSE(LPART);
      IF OPERATOR = NEGATION_OP THEN VALUE=-LVALUE;
      ELSE DO:
         VALUE=TRAVERSE(RPART);
         IF OPERATOR = ADD_OP THEN VALUE=LVALUE+RVALUE;
         ELSE IF OPERATOR = SUB_OP THEN VALUE=LVALUE-RVALUE;
         ELSE IF OPERATOR = MUL_OP THEN VALUE=LVALUE*RVALUE;
         ELSE IF OPERATOR = DIV_OP THEN VALUE=LVALUE/RVALUE;
         ELSE SIGNAL CONDITION(EADMOD);
      END;
   END;
   ELSE SIGNAL CONDITION(BADMOD);
END:
DONE:
RETURN(VALUE);
END TRAVERSE;

END CREATE_MODULE;
/* PERFORM THE INCLUDE TRANSFORMATION */

INCLUDE:
PROCEDURE(@DEF, @RENAME, @HDR) RECURSIVE;

DECLARE /* PARAMETERS */
@DEF POINTER,
@RENAME POINTER,
@HDR POINTER;

DECLARE /* DECLARATIONS HEADER */
1 DECL_HEADER BASED(@HEADER),
2 ENC_BLOCK POINTER,
2 HEADER,
3 @VARS POINTER,
3 @RTNS POINTER,
3 @MODS POINTER,
3 @INCS POINTER,
3 @NAMES POINTER,
3 @INS POINTER,
3 @PARS POINTER,
3 #DECLARATIONS FIXED BINARY(15),
3 #ACCESSIBLES FIXED BINARY(15),
@HEADER POINTER;

DECLARE /* NAME STRUCTURE */
1 NAME_NODE BASED(@NAME),
2 LINK POINTER,
2 NAME,
3 TYPE FIXED BINARY(15),
3 ID CHARACTER(6),
3 @VAL POINTER,
@NAME POINTER;

DECLARE
1 LINK BASED(@LINK),
2 @MCD POINTER,
2 @DECL POINTER,
@LINK POINTER;

DECLARE
1 ACCESSIBLES BASED(@ACCESSIBLES),
2 @NAMES FIXED BINARY(15),
2 ACCESSIBLE(N REFER(@NAMES)),
3 TYPE FIXED BINARY(15),
3 ID CHARACTER(6),
3 @CELL POINTER,
@ACCESSIBLES POINTER;
DECLARE 
  1 DECL 
  2 TYPE 
  2 ID 
  2 &CELL 
@DECL 
#&DECL 

DECLARE 
  1 RENAME_STRUCTURE 
  2 NEXT 
  2 RENAME, 
  3 ID 
  3 ALIAS 
@RENAME 

DECLARE @THIS 
DECLARE @FIRST 

DECLARE OLD_NAME 
DECLARE NEW_NAME 
DECLARE IS_LINKED 
DECLARE IDENT 
DECLARE CASE(3) 
DECLARE I 
DECLARE M 
DECLARE N 
@HEADER=&DEF; 

N=#ACCESSIBLES; M=N; 
ALLOCATE ACCESSIBLES IN(SCRATCH); 

@DECL=ADDR(ACCESSIBLE(1)); @FIRST=@DECL; 

CALL SEARCH(@VARS); 
CALL SEARCH(@RTNS); 
CALL SEARCH(@NAMES); 
IF N > 0 THEN SIGNAL CONDITION(BADMOD);
/* PERFORM ANY RENAMING. */
@RENAME=@RENAME;
DO WHILE(@RENAME /= NULL);
IDENT=@RENAME.ID;
@DECL=@FIRST;
DO I=1 TO M;
IF @DECL.ID = IDENT THEN
IF @DECL.TYPE = CLD_NAME THEN DO:
 @DECL.TYPE=NEW_NAME;
 @DECL.ID=@RENAME.ALIAS;
 GOTO NEXT_RENAME;
 END;
 @DECL=#@DECL+12;
 END;
 SIGNAL CONDITION(SACMOD);
 NEXT_RENAME:
 @RENAME=NEXT;
 END;
@HEADER=&HDR; @DECL=&FIRST;
 DO I=1 TO M;
 ALLOCATE NAME_NODE IN(SCRATCH), LINK IN(SCRATCH);
 NAME.TYPE=IS_LINKED;
 NAME.ID=@DECL.ID;
 NAME.@VAL=@LINK;
 LINK.@MOD=@DEF;
 LINK.@DECL=@DECL.@CELL;
 NAME_NODE.LINK=@VARS; @VARS=@NAME;
 #@DECL=#@DECL+12;
 END;
 #DECLARATIONS=#DECLARATIONS+M;
 FREE ACCESSIBLES IN(SCRATCH);
 RETURN;
SEARCH FOR ACCESSIBLE NAMES

SEARCH:
PROCEDURE(@LIST):

DECLARE /* PARAMETERS */
@LIST POINTER;
DECLARE @CELL POINTER;
DECLARE IS_ACCESSIBLE FIXED BINARY(15) STATIC INITIAL(1);
DECLARE NAMES_LCL_ACCESSIBLE FIXED BINARY(15) STATIC INITIAL(3);
DECLARE IS_INSERTED FIXED BINARY(15) STATIC INITIAL(7);

@CELL=@LIST;
DO WHILE(@CELL != NULL):
  IF (@CELL->NAME.TYPE = IS_ACCESSIBLE) | 
    (@CELL->NAME.TYPE = NAMES_LCL_ACCESSIBLE) | 
    (@CELL->NAME.TYPE = IS_INSERTED) THEN DO;
    IF N = 0 THEN SIGNAL CONDITION(EADMOD);
    DECL TYPE=CLO_NAME;
    DECL.ID=@CELL->NAME.ID;
    DECL.@CELL=@CELL;
    N=N-1;
  END;
  @CELL=@CELL->NAME_NODE.LINK;
END;
RETURN;
END SEARCH;
END INCLUDE;
PERFORM THE INSERT TRANSFORMATION

INSERT:
PROCEDURE(@DEF, @RENAMEs, @HDR) RECURSIVE;

DECLARE /* PARAMETERS */
@DEF POINTER,
@RENAMEs POINTER,
@HDR POINTER;

DECLARE /* DECLARATIONS HEADER */
1 CECL HEADER BASED(@HEADER),
2 ENC_BLOCK POINTER,
2 HEADER,
3 @VARS POINTER,
3 @TNS POINTER,
3 @MODS POINTER,
3 @NCS POINTER,
3 @NAMES POINTER,
3 @INS POINTER,
3 @PARS POINTER,
3 #DECLARATIONS FIXED BINARY(15),
3 #ACCESSIBLES FIXED BINARY(15),
@HEADER POINTER;

DECLARE /* NAME STRUCTURE */
1 NAME_NODE BASED(@NAME),
2 LINK POINTER,
2 NAME,
3 TYPE FIXED BINARY(15),
3 ID CHARACTER(6),
3 @VAL POINTER,
@NAME POINTER;

DECLARE
1 INSERTED BASED(@INSERTED),
2 @MOD POINTER,
2 @DECL POINTER,
@INSERTED POINTER;

DECLARE
1 ACCESSIBLES BASED(@ACCESSIBLES),
2 #NAMES FIXED BINARY(15),
2 ACCESSIBLE(N REFER(#NAMES)),
3 TYPE FIXED BINARY(15),
3 ID CHARACTER(6),
3 @CELL POINTER,
@ACCESSIBLES POINTER;
DECLARE
1 DECL
2 TYPE
2 ID
2 @CELL
DECLARE
ENDDECL
DECLARE
1 RENAME_STRUCTURE
2 NEXT
2 RENAME,
3 ID
3 ALIAS
DECLARE
ENDDECL
DECLARE
RENAME
DECLARE
RENAME STRUCTURE
DECLARE
ENDDECL
DECLARE
IS_INSERTED
DECLARE
IDENT
DECLARE
CASE(3)
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLARE
ENDDECL
DECLA
/* PERFORM ANY RENAMING. */
&RENAMEN=&RENAME;
DO WHILE (&RENAME != NULL);
  IDENT=RENAME.ID;
  &DECL=&FIRST;
  DO I=1 TO M;
    IF DECL.ID = IDENT THEN
      IF DECL.TYPE = OLD_NAME THEN DO:
        DECL.TYPE=NEW_NAME;
        DECL.ID=RENAME.ALIAS;
        GOTO NEXT_RENAME;
      END;
      #DECL=#DECL+12;
    END;
  END;
  SIGNAL CONDITION(BADMOD);
NEXT_RENAME:
  &RENAME=NEXT;
END;

&HEADER=&HEADER; &DECL=&FIRST;
DO I=1 TO M;
  ALLOCATE NAME_NODE IN(SCRATCH), INSERTED IN(SCRATCH);
  NAME.TYPE=INSERTED;
  NAME.ID=DECL.ID;
  NAME.VALUE=INSERTED;
  INSERTED.MCD=DEF;
  INSERTED.OEQL=DECL.OECL;
  LINK=VARS; VARS=NAME;
  #DECL=#DECL+12;
END;

#DECLARATIONS=#DECLARATIONS+W;
#ACCESSIBLES=#ACCESSIBLES+M;
FREE ACCESSIBLES IN(SCRATCH);
RETURN;
/*
SEARCH FOR ACCESSIBLE NAMES
*/

SEARCH:
PROCEDURE (@LIST);

DECLARE /* PARAMETERS */
@LIST POINTERS;
DECLARE @CELL POINTER;
DECLARE IS_ACCESSIBLE FIXED BINARY (15) STATIC INITIAL (1);
DECLARE NAMES_LCL_ACCESSIBLE FIXED BINARY (15) STATIC INITIAL (3);
DECLARE IS_INSERTED FIXED BINARY (15) STATIC INITIAL (7);

@CELL = @LIST;
DO WHILE (@CELL != NULL) |
IF (@CELL->NAME.TYPE = IS_ACCESSIBLE) |
(@CELL->NAME.TYPE = NAMES_LCL_ACCESSIBLE) |
(@CELL->NAME.TYPE = IS_INSERTED) THEN DO;
IF N = 0 THEN SIGNAL CONDITION (BADMOD);
DECL .TYPE=CLD_NAME;
DECL.ID=@CELL->NAME.ID;
DECL @CELL = @CELL;
END;
END; 
@CELL = @CELL->LINK;
END SEARCH;
END INSERT;
RESOLVE MODULE REFERENCES

RESOLVE:
PROCEDURE(OHEADER) RECURSIVE;

DECLARE OHEADER POINTER;

DECLARE /* DECLARATIONS HEADER */
1 DECL_HEADER BASED(OHEADER),
2 @REBLOCK POINTER,
3 HEADER,
4 @VARS POINTER,
5 @RNS POINTER,
6 @MODS POINTER,
7 @INCS POINTER,
8 @NAMES POINTER,
9 @INS POINTER,
10 @PARS POINTER,
11 #DECLARATIONS FIXED BINARY(15),
12 #ACCESSIBLES FIXED BINARY(15);

DECLARE /* NAME STRUCTURES */
1 NAME_NODE BASED(@NAME),
2 LINK POINTER,
3 NAME,
4 TYPE FIXED BINARY(15),
5 ID CHARACTER(6),
6 @VAL POINTER,
7 NAME_NODE_1 BASED(@NAME),
8 PAC POINTER,
9 RESOLVED,
10 TYPE FIXED BINARY(15),
11 PAD FIXED BINARY(15),
12 @MDEF POINTER,
13 @VAL POINTER,
14 @NAME POINTER;
DECLARE /* CODE CELL */
1 CODE
  2 CODE
  2 ENV
  2 #INST
  2 INST(2 REFER(CODE.#INST))
  3 CPCODE
  3 PAD
  3 @RAND
@CC

DECLARE /* INSTRUCTION SLBCELL */
1 INST
  2 CPCODE
  2 PAD
  2 @RAND
  2 @INST
  2 #@INST
INST
INSTRUCTION
SLBCELL

DECLARE IS_IMPLICIT
FIXED BINARY(15) STATIC INITIAL(0);

DECLARE BLOCK_ENTRY
BIT(16) ALIGNED STATIC INITIAL("1CCCC0000000000000000000B");

DECLARE @THIS
POINTER;

DECLARE I
FIXED BINARY(15);
/* RESOLVE LOCAL MODULE DEFINITIONS. */

@THIS=CDSC;
DO WHILE (@THIS = NULL);
   @NAME = THIS->NAME; @VAL;
   IF .TYPE = IS.IMPLICIT THEN
      IF THIS->NAME.ID = NAME.ID THEN
         @MDEF = SEARCH (NAME, ID, @ENC_BLOCK);
         ELSE @MDEF = SEARCH (NAME, ID, @HEADER);
         END;
      END;
   ELSE @MDEF = SEARCH (NAME, ID, @HEADER);
   END;

/* RESOLVE LOCAL INCLUDES. */

@THIS=INCSC;
DO WHILE (@THIS = NULL);
   @THIS->MDEF = SEARCH (@THIS->NAME, ID, @HEADER);
   @THIS = @THIS->LINK;
END;

/* RESOLVE LOCAL ENVIRONMENT NAMES. */

@THIS=NAMESC;
DO WHILE (@THIS = NULL);
   @NAME = @THIS->NAME; @VAL;
   @MDEF = SEARCH (NAME, ID, @HEADER);
   @THIS = @THIS->LINK;
END;

/* RESOLVE LOCAL INSERTS. */

@THIS=IRNS;
DO WHILE (@THIS = NULL);
   @THIS->MDEF = SEARCH (@THIS->NAME, ID, @HEADER);
   @THIS = @THIS->LINK;
END;

RETURN;
/*
SEARCH FOR MODULE REFERENCE
*/

PROCEDURE (ID, &HDR) RETURNS (POINTER);

DECLARE /* PARAMETERS */
  ID CHARACTERS(*)
  &HDR POINTER;

DECLARE /* DECLARATIONS HEADER */
  DECL_HEADER BASED (@HEADER),
  ENC_BLOCK POINTER,
  HEADER,
  E_VARS POINTER,
  &RTNS POINTER,
  &MODS POINTER,
  &INCS POINTER,
  &NAMES POINTER,
  &INS POINTER,
  &S Parsons POINTER,
  DECLARATIONS FIXED BINARY(15),
  ACCESSIBLES FIXED BINARY(15),
  @HEADER POINTER;

DECLARE /* NAME STRUCTURE */
  NAME_NODE BASED (@NAME),
  LINK POINTER,
  NAME,
  TYPE FIXED BINARY(15),
  ID CHARACTER(6),
  &VAL POINTER,
  @NAME POINTER;

DECLARE IDENT CHARACTER(6);

@HEADER=&HDR; IDENT=ID;

DO WHILE (@HEADER NULL);
  @NAME=&MODS;
  DO WHILE (@NAME NULL);
    IF NAME.ID = IDENT THEN RETURN (NAME.@VAL);
    @NAME=LINK;
  ENC;
  @HEADER=@ENC_BLOCK;
END;
SIGNAL CONDITION (BADMOD);

END SEARCH;
END RESOLVE;
COPY:

PROCEDURE(@HDR) RETURNS POINTER) RECURSIVE;

DECLARE @HDR POINTER;

DECLARE /* DECLARATIONS HEADER */
1 DECL_HEADER BASED(@HEADER),
2 &ENC_BLOCK POINTER,
2 HEADER,
3 &VARS POINTER,
3 &RTNS POINTER,
3 &MOOS POINTER,
3 &INCPS POINTER,
3 &NAMES POINTER,
3 @INS POINTER,
3 @PARS POINTER,
3 #DECLARATIONS FIXED BINARY(15),
3 #ACCESSIBLES FIXED BINARY(15),
@HEADER POINTER;

DECLARE /* NAME STRUCTURE */
1 NAME NODE BASED(@NAME),
2 LINK POINTER,
2 NAME,
3 TYPE FIXED BINARY(15),
3 PAD FIXED BINARY(15),
3 @MDEF POINTER,
3 @VAL POINTER,
@NAME POINTER;

DECLARE /* CODE CELL */
1 CODE EASEC(&CC),
2 @ENV PCINTER,
2 @INST FIXED BINARY(15),
2 INST(N REFER(CODE,&INST)),
3 OP_CODE BIT(16) ALIGNED,
3 FAD CHARACTER(6),
3 @RAND POINTER,
@CC POINTER;
DECLARE /* INSTRUCTION SUBCELL */
1 INST BASED(@INST),
  2 CPCODE BIT(16) ALIGNED,
  2 PAD CHARACTER(6),
  2 @RENAMES POINTER,
  @INST POINTER,
  @@INST FIXED BINARY(31) DEFINED(@INST);

DECLARE THIS CHARACTER(6) BASED(@THIS);
  @THIS POINTER EASEC(@THIS),
    Pointer;

DECLARE @ORIG POINTER;
DECLARE @DEF POINTER;
DECLARE @VAL POINTER;
DECLARE @CODE POINTER;

DECLARE @REFORM BASED(@REFORMS),
  1 @REFORMS POINTER,
  2 @PARAMSETS POINTER,
  2 @RENAME POINTER;

DECLARE IS_BIND FIXED BINARY(15) STATIC INITIAL(1);

ALLOCATE DECL_HEADER IN(SCRATCH);
/* HANDLE THE VARIABLE DECLARATIONS. */

@THIS=ADDR(@VARS); @ORIG=@HDR->@VARS;

DO WHILE (@ORIG != NULL);
    ALLOCATE NAME_NODE IN (SCRATCH);
    NAME=@ORIG->NAME;
    THIS=NAME;
    @THIS=ADDR(LINK);
    @ORIG=@ORIG->LINK;
END;

THIS=NULL;

/* HANDLE THE ROUTINE DECLARATIONS. */

@THIS=ADDR(@RTNS); @ORIG=@HDR->@RTNS;

DO WHILE (@ORIG != NULL);
    ALLOCATE NAME_NODE IN (SCRATCH);
    NAME=@ORIG->NAME;
    @CC=LOCK_COPY (@ORIG->NAME, @VAL, @HEADER);
    NAME, @VAL=@CC;
    @INST=ADDR (CCDE, INST, CODE, #INST)));
    INST.@RAND=LOCK_COPY (INST.@RAND, CCDE, @ENV);
    THIS=NAME;
    @THIS=ADDR(LINK);
    @ORIG=@ORIG->LINK;
END;

THIS=NULL;
/* HANDLE ANY LOCAL MODULE DEFINITIONS. */

@THIS=ADDR(@MODS); @ORIG=@HDR->@MODS;

DO WHILE(@CRIG -= NULL);
  ALLOCATE NAME_NODE IN(SMCRATCH), NAME_NODE IN(SMCRATCH) SET(@VAL);
  NAME=@ORIG->NAME;
  @DEF=NAME.@VAL;
  NAME.@VAL=@VAL;
  @VAL->NAME=@DEF->NAME;
  THIS=NAME;
  @THIS=ADDR(LINK);
  @ORIG=@ORIG->LINK;
END;

THIS=NULL;

/* HANDLE INCLUSIONS. */

@THIS=ADDR(@INCS); @CRIG=@HDR->@INCS;

DO WHILE(@CRIG -= NULL);
  ALLOCATE NAME_NODE IN(SMCRATCH);
  NAME=@ORIG->NAME;
  IF NAME.@VAL -= NULL THEN GO;
    ALLOCATE REFORM IN(SMCRATCH);
    @VAL=NAME.@VAL;
    REFORM=@VAL->REFORM;
    NAME.@VAL=REFORMS;
END;

THIS=NAME;
@THIS=ADDR(LINK);
@ORIG=@ORIG->LINK;
END;

THIS=NULL;
/* HANDLE LOCAL ENVIRONMENT DECLARATIONS. */

@THIS=ADDR(@NAMES); @ORIG=@HDR->@NAMES;
DO WHILE (@ORIG != NULL);
   ALLOCATE NAME_NODE IN (SCRATCH), NAME_NODE IN (SCRATCH) SET (@VAL);
   NAME = @CRIG->NAME;
   @DEF = NAME.*@VAL;
   @VAL = NAME.*@DEF->NAME;
   IF @VAL->NAME.TYPE = IS_BIND THEN
      IF @VAL->NAME.*@VAL = NULL THEN CO;
         ALLOCATE REFORM IN (SCRATCH);
         @DEF = @VAL = @REFORM;
         @VAL->NAME.*@VAL = @REFORMS;
      END;
   END;
   @THIS = @NAME;
   @THIS = ADDR (LINK);
   @CRIG = @ORIG->LINK;
END;

@THIS = NULL;

/* HANDLE INSERTIONS. */

@THIS = ADDR (@INS); @ORIG = @HDR->@INS;
DO WHILE (@ORIG != NULL);
   ALLOCATE NAME_NODE IN (SCRATCH);
   NAME = @CRIG->NAME;
   IF NAME.*@VAL = NULL THEN CO;
      ALLOCATE REFORM IN (SCRATCH);
      @VAL = NAME.*@VAL;
      REFORM = @VAL->REFORM;
      NAME.*@VAL = @REFORMS;
   END;
   @THIS = @NAME;
   @THIS = ADDR (LINK);
   @ORIG = @ORIG->LINK;
END;

@THIS = NULL;
HANDLE PARAMETER DECLARATIONS. */

@THIS=ADDR@PARS; @CRIG=@FDR->@PARS;

DO WHILE(@ORIG /= NULL);
    ALLOCATE NAME_NODE IN SCRATCH;
    NAME=@CRIG->NAME;
    THIS=NAME;
    @THIS=ADDR(LINK);
    @ORIG=@ORIG->LINK;
END;

THIS=NULL;

#DECLARATIONS=@FDR->#DECLARATIONS;
#ACCESSIBLES=@FDR->#ACCESSIBLES;

RETURN(&HEADER);

END COPY;
/* BLOCK_COPY: */
PROCEDURE(&CODE,&ENV) RETURNS(POINTER) RECURSIVE;

DECLARE /* PARAMETERS */
&CODE POINTER,
&ENV POINTER;

DECLARE /* DECLARATIONS HEADER */
1 DECL_HEADER BASED(&HEADER),
2 @ENC_BLKCK POINTER,
3 @SARS POINTER,
3 @RTNS POINTER,
3 @MODS POINTER,
3 @INCS POINTER,
3 @NAMES POINTER,
3 @INS POINTER,
3 @PARS POINTER,
3 #DECLARATIONS FIXED BINARY(15),
3 #ACCESSIBLES FIXED BINARY(15),
@HEADER POINTER;

DECLARE /* CODE CELL */
1 CODE BASED(&CC),
2 &ENV POINTER,
2 @INST FIXED BINARY(15),
2 @INST(N REFER(&CODE,@INST)),
3 @PCODE BIT(16) ALIGNED,
3 @AD CHARACTER(6),
3 @RAND POINTER,
@CC POINTER;

DECLARE /* INSTRUCTION SUBCELL */
1 INST BASED(@INST),
2 CPCODE BIT(16) ALIGNED,
2 @AD CHARACTER(6),
2 @RAND POINTER,
@INST POINTER,
#@INST FIXED BINARY(31) DEFINED(@INST);

DECLARE BLOCK_ENTRY BIT(16) ALIGNED STATIC
INITIAL("1000000000000000'B);

DECLARE @ FIXED BINARY(15);
N = @CODE->CODE.#INST;
ALLOCATE CODE IN(SCRATCH);
CODE.INST = @CODE->CODE.INST;
@HEADER = COPY(@CODE->CODE.@ENV);
CODE.@ENV = @HEADER;
@ENC_BLOCK = @ENV;
@INST = ADDR(CODE.INST(1));
DO I = 1 TO N;
   IF INST.Opcode = BLOCK_ENTRY THEN
      INST.@RAND = BLOCK_COPY(INST.@RAND, @HEADER);
      @INST = @INST + 12;
   END;
RETURN(@CC);
END BLOCK_COPY;
CREATE BLOCK ENTRY POINT CODING AND BLOCK ENTRY INSTRUCTIONS

BUILD BLOCK:
PROCEDURE(@CODE, @PREV) RETURNS(POINTER) RECURSIVE;

DECLARE /* PARAMETERS */
@CODE  PCINTER,
@PREV  PCINTER;

DECLARE /* DECLARATIONS HEADER */
1 DECL_HEADER BASED(@HEADER),
2 @ERCC_BLOCK POINTER,
2 HEADER,
3 @VARS  POINTER,
3 @ARTNS  POINTER,
3 @MODDS  PCINTER,
3 @INCS  POINTER,
3 @NAMES  POINTER,
3 @INS  PCINTER,
3 @PARS  POINTER,
3 #DECLARATlONS FIXED BINARY(15),
3 #ACCESSIBLES FIXED BINARY(15),
@HEADER  POINTER;

DECLARE /* CODE CELL */
1 CODE BASED(@CC),
2 @ENV  POINTER,
2 #INST FIXED BINARY(15),
2 INST(N REFER(CODE, #INST)),
3 CPCODE BIT(16) ALIGNED,
3 PAD FIXED BINARY(15),
3 @RAND1  POINTER,
3 @RAND2  PCINTER,
@CC;

DECLARE /* INSTRUCTION SUBCELL */
1 INST BASED(@INST),
2 CPCODE BIT(16) ALIGNED,
2 PAD FIXED BINARY(15),
2 @RAND1  POINTER,
2 @RAND2  PCINTER,
@INST  PCINTER,
##@INST FIXED BINARY(31) DEFINED(@INST);
DECLARE /* LABEL STRUCTURE */
  1 LABEL BASED(@LABEL),
  2 IP,
  3 CODE FIXED BINARY(15),
  3 STM# POINTER,
  2 EP POINTER;
DECLARE BLOCK_ENTRY BIT(16) ALIGNED STATIC INITIAL('1000000000000000*B);
DECLARE BLOCK_EXIT BIT(16) ALIGNED STATIC INITIAL('0100000000000000*B);
DECLARE ALLOC BIT(16) ALIGNED STATIC INITIAL('1000000000000000*B);
DECLARE ENTER_BLOCK BIT(16) ALIGNED STATIC INITIAL('10000000001000000*B);
DECLARE @LIST POINTER;
DECLARE @LAST POINTER;
DECLARE @EPC POINTER;
DECLARE @BPC POINTER;
DECLARE @COPY POINTER;
DECLARE @ALLOC POINTER EXTERNAL;
DECLARE 1 CHAIN BASED(@CHAIN),
  2 LINK POINTER,
  2 @EPC POINTER,
  @CHAIN POINTER;
DECLARE I FIXED BINARY(15);
DECLARE N FIXED BINARY(15);
IF @INS == NULL THEN SIGNAL CONDITION(BADPROG);
@LIST=BUILD_CCNTOURS(@HEADER,NULL,@PREV);
@INST=@LIST->INST.@RAND1;
DO N=1 BY 1 WHILE(@INST == NULL); @INST=INST.@RAND1; END;
ALLOCATE CODE IN(PROGRAM);
CODE.@ENV=@PREV;
@INST=ADDR(CODE.INST(1));
DO WHILE(@LIST == NULL);
INST.CPCODE=ALLOC;
INST.@RAND2=@LIST->INST.@RAND2;
@&INST=@&INST+12;
@LIST=@LIST->INST.@RAND1;
END;
@&INST=@&INST-12;

/* CREATE FINAL CODE FORM IN PROGRAM AREA. */
N=CODE->CODE,#&INST;
ALLOCATE CODE IN(PROGRAM) SET(&COPY);
&COPY->CODE.INST=CODE->CODE.INST;

/* BUILD ENTER INSTRUCTION. */
INST.OPCODE=ENTER_BLCK;
INST.@RAND1=&COPY;
@BC=INST.@RAND2;
&COPY->CODE.@ENV=BC;

/* SET LABEL CONSTANTS. */
CALL SET_LABELS(&COPY);
ALLOCATE CHAIN IN(SRATCH);
CHAIN.LINK=&ALLOC; CHAIN.@EPC=CC; &ALLOC=CHAIN;
/* BUILD ANY NESTED BLOCKS. */
@INST=ADDR(ADDR->CODE, INST(1));

DO I=1 TO N;
  IF INST.OPCODE = BLOCK_ENTRY THEN CC;
  @BEPC=BUILD_BLOCK(INST.@RAND2, @BC);
  INST.@RAND1=@BEPC;
  @LAST=ADDR(@BEPC->CODE, INST(@BEPC->CODE, #INST));
  INST.@RAND2=@LAST->INST.@RAND2;

/* CREATE FINAL FORM OF THE BLOCK EXIT INSTRUCTION. */
@LAST=LAST->INST.@RAND1;
@LAST=ADDR(@LAST->CODE, INST(@LAST->CODE, #INST));
IF @LAST->INST.OPCODE = BLOCK_EXIT THEN
  SIGNAL CONDITION(BADFCC);
  ALLOCATE LABEL IN(PROGRAM);
  LABEL.IP->CODE=ADDR;
  LABEL.IP->stmt#=#I+1;
  LABEL.EP=BC;
  @LAST->INST.@RAND1=LABEL;
END;
  #@INSTR=#@INSTR+12;
END;

RETURN(ACC);

END BUILD_BLOCK;
BUILD ROUTINE DEFINITION

BUILD ROUTINE:
PROCEDURE($EP,$PREV) RETURNS(POINTER) RECURSIVE:

DECLARE /* PARAMETERS */
$EPC POINTER,
$PREV POINTER;

DECLARE /* CODE CELL */
1 CODE BASED($CC),
2 $ENV POINTER,
2 #INST FIXED BINARY(15),
2 INST(N REFER(CODE,$INST)),
3 CPCODE BIT(16) ALIGNED,
3 PAD FIXED BINARY(15),
3 $RAND1 POINTER,
3 $RAND2 POINTER,
&CC POINTER;

DECLARE /* INSTRUCTION SUBCELL */
1 INST BASED($INST),
2 CPCODE BIT(16) ALIGNED,
2 PAD FIXED BINARY(15),
2 $RAND1 POINTER,
2 $RAND2 POINTER,
$INST:pointer;

DECLARE $FPC POINTER;
DECLARE $BEPC POINTER;

DECLARE N FIXED BINARY(15);

N=$EPC->CODE.#INST;
ALLOCATE CODE IN(PROGRAM);
CODE.INST=$EPC->CODE.INST;
CPCODE.$ENV=$PREV;

$INST=ADDR(CODE.INST(N));
$FPC=BUILD_FPC($EPC->CODE.$ENV,$PREV);
$BEPC=BUILD_BLOCK(INST.$RAND2,$FPC);
INST.$RAND1=$BEPC;

RETURN($CC);

END BUILD_ROUTINE;
BUILD FORMAL PARAMETER CONTOUR

BUILD_FPCC:
PROCEDURE(@HDR, @ENV) RETURNS(POINTER) RECURSIVE;

DECLARE /* PARAMETERS */
@HDR POINTER,
@ENV POINTER;

DECLARE /* DECLARATIONS HEADER */
1 DECL_HEADER BASED(@HEADER),
2 @ERC_BLOCK POINTER,
2 HEADER,
3 @VARS POINTER,
3 @RTNS POINTER,
3 @NOS POINTER,
3 @ICS POINTER,
3 @NAMES PCINTER,
3 @INS POINTER,
3 @PARS POINTER,
3 DECLARATIONS FIXED BINARY(15),
3 #ACCESSIBLES FIXED BINARY(15),
@HEADER POINTER;

DECLARE /* CONTOUR CELL */
1 CONTOUR BASED(@CONTOUR),
2 ENV_LINK POINTER,
2 ANT_LINK POINTER,
2 SP POINTER,
2 #DECLS FIXED BINARY(15),
2 DECLS(N REFER(CONTOUR.#DECLS)),
3 TYPE FIXED BINARY(15),
3 ID CHARACTER(6),
3 SIMPLE_VALUE FIXED BINARY(31),
3 &:VALUE PCINTER,
@CONTOUR POINTER;

DECLARE /* DECLARATION SUBCELL */
1 DECL BASED(@DECL),
2 TYPE FIXED BINARY(15),
2 ID CHARACTER(6),
2 SIMPLE_VALUE FIXED BINARY(31),
2 &:VALUE POINTER,
@DECL POINTER,
#DECL FIXED BINARY(31) DEFINED(@DECL);
DECLARE /* NAME STRUCTURES */
1 NAME_NODE BASED(@NAME),
2 LINK POINTER,
3 NAME FIXED BINARY(15),
4 TYPE CHARACTER(6),
5 ID POINTER,
6 &VAL FIXED BINARY(15).

DECLARE NODE_1 BASED(@NAME),
2 PAD POINTER,
3 RESOLVED FIXED BINARY(15),
4 TYPE FIXED BINARY(15),
5 PAD FIXED BINARY(15),
6 &MDEF POINTER,
7 &VAL POINTER,
8 NAME POINTER;

DECLARE IS_UNDEFINED FIXED BINARY(15) STATIC INITIAL(0);
DECLARE RETURN_ADDR_ID CHARACTER(6) STATIC INITIAL('#$#RET');

DECLARE I FIXED BINARY(15);
DECLARE N FIXED BINARY(15);

@HEADER=@HDR;
N=#DECLARATIONS+1;
ALLOCATE CCNTOUR IN(PROGRAM);
CCNTOUR.ENV_LINK=@ENV;
DECL=ADDR(CCNTOUR.DECLS(1));
DECL.ID=RETURN_ADDR_ID; DECL.TYPE=IS_UNDEFINED;
DECL=#DECL+16;
NAME=SPARS;
DO I=2 TO N;
  IF @NAME = NULL THEN SIGNAL CONDITION(EADPROG);
  DECL.ID=NAME.ID; DECL.TYPE=IS_UNDEFINED;
  DECL=#DECL+16;
  NAME=LINK;
END;
RETURN(CCNTOUR);
END BUILD_FPC;
BUILD MODULE CONTOURS

PROCEDURE(CHDR, @PREV, @ENV) RETURNS(POINTER) RECURSIVE:

DECLARE /* PARAMETERS */
  @HDR   POINTER,
  @PREV  POINTER,
  @ENV   POINTER;

DECLARE /* DECLARATIONS HEADER */
1 DECL_HEADER  BASED(HEADER),
  @ENC BLOCK  POINTER,
  HEADER,
  3 SAVARS   POINTER,
  3 SRTNS    POINTER,
  3 @MODS    POINTER,
  3 @INCS    POINTER,
  3 @NAMES   POINTER,
  3 @INCS    POINTER,
  3 @NAMES   POINTER,
  3 @NAMES   POINTER,
  3 @NAMES   POINTER;
  3 #DECLARATIONS FIXED BINARY(15),
  3 #ACCESSIBLES FIXED BINARY(15),
  @HEADER    POINTER;

DECLARE /* NAME STRUCTURES */
1 NAME_NODE  BASED(NAME),
  #NAME NODE 1
  2 LINK   POINTER,
  2 NAME,
  3 TYPE   FIXED BINARY(15),
  3 ID     CHARACTER(6),
  3 @VAL   POINTER,

1 NAME_NODE_1 BASED(NAME),
  2 FAC   POINTER,
  2 RESOLVED,  TYPE
  3 TYPE   FIXED BINARY(31),
  3 @DEF   POINTER,
  3 @VAL   POINTER,

1 NAME_NODE_2 BASED(NAME),
  2 PAD   POINTER,
  2 INITIALIZED,  PAD
  3 PAD   CHARACTER(8),
  3 INITIAL_VALUE FIXED BINARY(31),
  @NAME   POINTER;
DECLARE /* CONTOUR CELL */
1 CONTOUR BASED(aCONTOUR),
2 ENV_LINK POINTER,
2 ANT_LINK POINTER,
2 SP POINTER,
2 #DECLS FIXED BINARY(15),
2 DECLS(N REFER(CONTOUR,#DECLS)),
3 TYPE FIXED BINARY(15),
3 ID CHARACTER(6),
3 SIMPLE_VALUE FIXED BINARY(31),
3 @VALUE POINTER.
DECLS)

DECLARE /* DECLARATION SUBCELL */
1 DECL BASED(aDECL),
2 TYPE FIXED BINARY(15),
2 ID CHARACTER(6),
2 SIMPLE_VALUE FIXED BINARY(31),
2 @VALUE POINTER.
DECL)

DECLARE /* INSTRUCTION SUBCELL */
1 INST BASED(aINST),
2 OPCODE BIT(16) ALIGNED,
2 FAD BIT(16) ALIGNED,
2 NEXT POINTER,
2 RAND POINTER,
INST)

DECLARE /* ALIAS */
1 ALIAS BASED(aALIAS),
2 CONTOUR POINTER,
2 DECL# FIXED BINARY(15),
ALIAS)

DECLARE /* LINKED */
1 LINKED BASED(aLINK),
2 @CONTOUR POINTER,
2 @DECL POINTER.
LINK)

DECLARE IS_UNDEFINED FIXED BINARY(15) STATIC INITIAL(0);
DECLARE IS_SIMPLE FIXED BINARY(15) STATIC INITIAL(1);
DECLARE IS_LABEL FIXED BINARY(15) STATIC INITIAL(2);
DECLARE IS_ROUTINE FIXED BINARY(15) STATIC INITIAL(3);
DECLARE IS_ENV FIXED BINARY(15) STATIC INITIAL(4);
DECLARE HAS_ALIAS FIXED BINARY(15) STATIC INITIAL(5);
DECLARE IS_LOCAL_ENV FIXED BINARY(15) STATIC INITIAL(6);
DECLARE
   NAMES_LCL FIXED BINARY(15) STATIC INITIAL(3);
   NAMES_LCL_ACCESSIBLE FIXED BINARY(15) STATIC INITIAL(4);
   IS_INITIALIZED FIXED BINARY(15) STATIC INITIAL(5);
   IS_LINKED FIXED BINARY(15) STATIC INITIAL(6);
   IS_INSERTED FIXED BINARY(15) STATIC INITIAL(7);
   IS_ROUTINE_PARAM FIXED BINARY(15) STATIC INITIAL(8);
   IS_ENV_PARAM FIXED BINARY(15) STATIC INITIAL(9);

DECLARE aMOD POINTER;
DECLARE aid POINTER;
DECLARE &pc POINTER;
DECLARE aRef POINTER;
DECLARE aVal POINTER;
DECLARE CECL# FIXED BINARY(15);
DECLARE alloc BIT(16) ALIGNED STATIC INITIAL("0000000100000000");
DECLARE IDENT CHARACTER(6);
DECLARE I FIXED BINARY(15);
DECLARE a FIXED BINARY(15);

@HEADER=@HDR;
N=#DECLARATIONS;
ALLOCATE CONTOUR IN(PROGRAM), INST IN(SCRATCH);
CONTOUR enumerated=env;
INST .OPCODE=ALLOC; INST .NEXT=PREV; INST .aRAND=CONTOUR;
@ENC_BLOCK=INST;

/* BUILD CONTOURS FOR INCLUDED MODULES. */
@NAME=@INCS;
DO WHILE(@NAME ->= NULL);
   @INST=BUILD_CONTOURS(@MDEF, @INST, NULL);
   @NAME=LINK;
END;

/* BUILD CONTOURS FOR INSERTED MODULES. */
@NAME=@INS;
DO WHILE(@NAME ->= NULL);
   @INST=BUILD_CONTOURS(@MDEF, @INST, NULL);
   @NAME=LINK;
END;
CREATE LOCAL DECLARATION SLBCELLS.

BUILD CONTOURS FOR DEFINED ENVIRNMENTS.

PLACE LOCAL ROUTINE DEFINITIONS IN CONTOUR.

PLACE PARAMETERS IN CONTOUR.
/* PLACE VARIABLE DECLARATIONS, BOTH LOCAL AND EXPOSED, IN CONTOUR. */
@NAME=@VARS;
DO i=1 TO N WHILE(@NAME -= NULL);
DECL.*ID=NAME.*ID;
IF (NAME.*TYPE = NAMES_LCL) | (NAME.*TYPE = NAMES_LCL_ACCESSIBLE)
  THEN DECL.*TYPE=IS_LOCAL_ENV;
ELSE IF NAME.*TYPE = IS_LABEL THEN DO;
  DECL.*TYPE=IS_LABEL;
  DECL.*SIMPLE_VALUE=INITIAL_VALUE;
END;
ELSE IF NAME.*TYPE = IS_INITIALIZED THEN DO;
  DECL.*TYPE=IS_SIMPLE;
  DECL.*SIMPLE_VALUE=INITIAL_VALUE;
END;
ELSE IF (NAME.*TYPE = IS_ENV_PARAM) |
  (NAME.*TYPE = IS_ENV_PARAM) THEN DO;
  DECL.*TYPE=NAME.*TYPE;
  DECL.*VALUE=NAME.*VAL;
END;
ELSE IF (NAME.*TYPE = IS_LINKED) | (NAME.*TYPE = IS_INSERTED)
  THEN DO;
  DECL.*TYPE=HAS_ALIAS;
  ALLOCATE ALIAS IN(PROGRAM);
  DECL.*VAL=ALIAS;
  @LINK=NAME.*VAL;
  @MCD=LINKED.*MCD; @ID=LINKED.*DECL;
/* LOCATE CONTOUR IN WHICH ID APPEARS. */
@PC=@@MCD->ENC_BLOCK->INST->RAND;
IDENT=#ID->NAME.*ID;
DECL.#=SEARCH(IDENT.*PC);
IF DECL.# = 0 THEN SIGNAL CONDITION(EADPROG);
@REF=ADDR(@PC->CONTOUR.*DECLS(DECL.#));
IF @REF=DECL.*TYPE = HAS_ALIAS THEN DO;
  @VAL=REF->DECL.*VALUE;
  ALIAS=VAL->ALIAS;
END;
ELSE DO;
  ALIAS.*CONTOUR=@PC;
  ALIAS.*DECL=#DECL#;
END;
END;
ELSE DECL.*TYPE=IS_UNDEFINED;
#DECL=#DECL+16;
@NAME=LINK;
END;
RETURN(@INST);
SEARCH FOR DECLARATION SUBCELL

SEARCH:
PROCEDURE (ID, @ENV) RETURNS (FIXED BINARY (15));

DECLARE /* PARAMETERS */
ID CHARACTER (*),
@ENV POINTER;

DECLARE /* CONTOUR CELL */
1 @CCNTCUR BASED (@CCNTCUR),
2 ENV_LINK POINTER,
2 ANT_LINK POINTER,
2 SP POINTER,
2 #DECLS FIXED BINARY (15),
2 DECLS (N REFER (@CCNTCUR, #DECLS)),
3 TYPE FIXED BINARY (15),
3 ID CHARACTER (6),
3 SIMPLE_VALUE FIXED BINARY (31),
3 &VALUE POINTER,
@CCNTCUR POINTER;

DECLARE /* DECLARATION SUBCELL */
1 DECL BASED (@DECL),
2 TYPE FIXED BINARY (15),
2 ID CHARACTER (6),
2 SIMPLE_VALUE FIXED BINARY (31),
2 &VALUE POINTER,
&DECL POINTER,
@@DECL FIXED BINARY (31) DEFINED (@DECL);

DECLARE IDENT CHARACTER (6);

DECLARE I FIXED BINARY (15);
DECLARE N FIXED BINARY (15);
ACCTOUR=ENV; IDENT=ID;
N=CONTGUR.#DECLS;
IF N = 0 THEN RETURN(0);
DECL=ADDR(CCNTCUR.DECLS(1));
DO I=1 TO N;
   IF DECL.10 = IDENT THEN RETURN(I);
  DECL=#DECL+16;
END;
RETURN(0);
END SEARCH;
END BUILD_CCNTGURS;
/* INITIALIZE LABEL CONSTANTS */

SET_LABELS:
PROCEDURE(\@CC):

DECLARE \@CC \&CC:

DECLARE /* CODE CELL */ CODE
1 CODE BASED(\@CC),
2 \&ENV PCINTER,
2 \#INST FIXED BINARY(15),
2 INST(N REFER(Code, \#INST)),
3 \&OPCODE BIT(16) ALIGNED,
3 \&PAD FIXED BINARY(15),
3 \&RAND1 PCINTER,
3 \&RAND2 PCINTER;

DECLARE /* CONTOUR CELL */ CONTOUR
1 CONTOUR BASED(\@CONTOUR),
2 ENV_LINK PCINTER,
2 ANT_LINK PCINTER,
2 SP PCINTER,
2 \#DECLS FIXED BINARY(15),
2 DECLS(N REFER(CONTOUR, \#DECLS)),
3 \&TYPE FIXED BINARY(15),
3 \&ID CHARACTER(6),
3 \&SIMPLE_VALUE FIXED BINARY(31),
3 \&VALUE PCINTER,
\&CONTOUR PCINTER;

DECLARE /* DECLARATION SUBCELL */ DECL
1 DECL BASED(\@DECL),
2 \&TYPE FIXED BINARY(15),
2 \&ID CHARACTER(6),
2 \&SIMPLE_VALUE FIXED BINARY(31),
2 \&VALUE PCINTER,
\&DECL PCINTER,
\#DECL FIXED BINARY(31) DEFINED(\@DECL);

DECLARE IS_LABEL FIXED BINARY(15) STATIC INITIAL(2);
DECLARE I FIXED BINARY(15);
DECLARE N FIXED BINARY(15);
$COUNTER=CODE*.ENV$
N=CONCUR*#DECLS;

IF N > 0 THEN DCL:
  $DECL=ADDR(CONTUR*DECLS(1))$;
  DO I=1 TO N;
    IF DECL.TYPE = IS_LABEL THEN DECL.VALUE=$CC$;
    $DECL=#DECL+16$;
  END;
  END;
  RETURN;
END SET_LABELS;
SET ROUTINE AND ENVIRONMENT TYPE MODULE PARAMETERS

SET PARAMS;

DECLARE @ALLOC POINTER EXTERNAL;

DECLARE /* BLOCK ENTRY POINT CODING CHAIN */
  1 CHAIN BASED(@CHAIN),
  2 LINK POINTER,
  2 @EPC POINTER;
@CHAIN POINTER;

DECLARE /* CONTOUR CELL */
  1 CONTOUR BASED(@CCNTCUR),
  2 ENV_LINK POINTER,
  2 ANT_LINK POINTER,
  2 SP POINTER,
  2 #DECLS FIXED BINARY(15),
  2 DECLS(N REFER(CONTOUR.#DECLS)),
  3 TYPE FIXED BINARY(15),
  3 ID CHARACTER(6),
  3 SIMPLE_VALUE FIXED BINARY(31),
  3 @VALUE POINTER,
@CONTOUR PCINTER;

DECLARE /* DECLARATION SLBCELL */
  1 DECL BASED(@DECL),
  2 TYPE FIXED BINARY(15),
  2 ID CHARACTER(6),
  2 SIMPLE_VALUE FIXED BINARY(31),
  2 @VALUE POINTER,
  #DECL POINTER,
  #DECL FIXED BINARY(31) DEFINED(@DECL);

DECLARE /* CODE CELL */
  1 CODE BASED(@CC),
  2 ENV POINTER,
  2 #INST FIXED BINARY(15),
  2 INST(N REFER(CODE.#INST)),
  3 OPCODE BIT(16) ALIGNED,
  3 PAD CHARACTER(6),
  3 @RAND POINTER,
@CC POINTER;
DECLARE /* INSTRUCTION SUBCELL */
  1 INST BASED(@INST),
  2 OP CODE BIT(16) ALIGNED,
  2 PAD CHARACTER(6),
  2 GRAND POINTER,
  @INST POINTER,
  &@INST FIXED BINARY(31) DEFINED(@INST);

DECLARE /* DECLARATION HEADER */
  1 DECL HEADER BASED(@HEADER),
  2 ENC BLOCK POINTER,
  2 HEADER POINTER,
  3 @VARS POINTER,
  3 @RTNS POINTER,
  3 @MODS POINTER,
  3 @INCS POINTER,
  3 @NAMES POINTER,
  3 @INS POINTER,
  3 @PANS POINTER,
  3 #DECLARATIONS FIXED BINARY(15),
  3 #ACCESSIBLES FIXED BINARY(15),
  @HEADER POINTER;

DECLARE /* NAME STRUCTURE */
  1 NAME NODE BASED(@NAME),
  2 NAME POINTER,
  2 NAME POINTER,
  3 TYPE FIXED BINARY(15),
  3 ID CHARACTER(6),
  3 @VAL POINTER,
  @NAME POINTER;

DECLARE IS_ROUTINE_PARAM FIXED BINARY(15) STATIC INITIAL(8),
IS_ENV_PARAM FIXED BINARY(15) STATIC INITIAL(5);
DECLARE IS_ROUTINE FIXED BINARY(15) STATIC INITIAL(3);
DECLARE IS_ENV FIXED BINARY(15) STATIC INITIAL(4);
DECLARE @MOD POINTER;
DECLARE @ID POINTER;
DECLARE @PC POINTER;
DECLARE @DECL POINTER;
DECLARE DECL# FIXED BINARY(15);
DECLARE IDENT CHARACTER(6);
DECLARE I FIXED BINARY(15);
DECLARE J FIXED BINARY(15);
DECLARE K FIXED BINARY(15);
DECLARE N FIXED BINARY(15);
@CHAIN=@ALLOC:

DO WHILE(@CHAIN ^= NULL);
@CC=@CHAIN@EPC;
N=CODE@INST;
IF N > 1 THEN DO;
@INST=ADDR(CODE@INST(1));
DO I=1 TO N-1;
@CONTOUR=INST@RAND;
K=CONTOUR@DECLS;
IF K > 0 THEN DO;
DECL=ADDR(CONTOUR@DECLS(1));
DO J=1 TO K;
IF DECL@TYPE = IS_ROUTINE_PARAM THEN DO;
@DECL=@DECL;
DO WHILE(@OLDDECL@DECL@TYPE = IS_ROUTINE_PARAM);
@LINK=@DECL@DECL@VALUE;
@MCN=LINKED@MOD; @IC=LINKED@DECL;
@PC=@MOD@ENC_BLOCK@INST@RAND;
IDENT=@ID@NAME.ID;
@DECL=SEARCH(IDENT@PC);
IF @DECL ^= NULL THEN SIGNAL CONDITION(EACPROG);
DECL@TYPE=IS_ROUTINE;
DECL@VALUE=@DECL@DECL@VALUE;
END;
ELSE IF DECL@TYPE = IS_ENV_PARAM THEN DO;
@DECL=@DECL;
DO WHILE(@OLDDECL@DECL@TYPE = IS_ENV_PARAM);
@LINK=@DECL@DECL@VALUE;
@MCN=LINKED@MOD; @ID=LINKED@DECL;
@PC=@MOD@ENC_BLOCK@INST@RAND;
IDENT=@ID@NAME.ID;
@DECL=SEARCH(IDENT@PC);
IF @DECL ^= NULL THEN SIGNAL CONDITION(BADPROG);
DECL@TYPE=IS_ENV;
DECL@VALUE=@DECL@DECL@VALUE;
END;
#DECL=#DECL+16;
END;
#INST=#INST+12;
END;
END;
@CHAIN=CHAIN@LINK;
ENC;
RETURN;
SEARCH FOR DECLARATION SUBCELL

PROCEDURE (ID, @ENV) RETURNS (POINTER);

DECLARE /* PARAMETERS */
  ID CHARACTER(*),
  @ENV POINTER;

DECLARE /* CONTOUR CELL */
  1 CCNTOUR BASED(@CCNTOUR),
  2 ENV_LINK POINTER,
  2 ANT_LINK POINTER,
  2 SP POINTER,
  2 #DECLS FIXED BINARY(15),
  2 DECLS(N REFER(CCNTOUR,#DECLS)),
  3 TYPE FIXED BINARY(15),
  3 ID CHARACTER(6),
  3 SIMPLE_VALUE FIXED BINARY(31),
  3 @VALUE POINTER,
  @CCNTOUR POINTER;

DECLARE /* DECLARATION SUBCELL */
  1 DECL BASED(@DECL),
  2 TYPE FIXED BINARY(15),
  2 ID CHARACTER(6),
  2 SIMPLE_VALUE FIXED BINARY(31),
  2 @VALUE POINTER,
  @DECL POINTER,
  #@DECL FIXED BINARY(31) DEFINED(@DECL);

DECLARE IDENT CHARACTER(6);
DECLARE I FIXED BINARY(15);
DECLARE N FIXED BINARY(15);
ACCTOUR=ENV; IDENT=ID;
N=CONTOUR.#DECLS;
IF N = 0 THEN RETURN(NULL);
@DECL=ADDR(CONTOUR.DECLS(1));
DO I=1 TO N:
   IF DECL.ID = IDENT THEN RETURN(@DECL);
      @DECL=DECL+16;
   END;
RETURN(NULL);
END SEARCH;
END SET_PARAMS;
END COMPILE;