A SEMANTIC FORMALISM AND ASSOCIATED SEMANTIC PROCESS FOR THE SPECIFICATION AND TRANSLATION OF PROGRAMMING LANGUAGES

Marsha Drapkin Hopwood

Department of Information and Computer Science
University of California
Irvine, California 92664
March 1974

TECHNICAL REPORT #43 - MARCH 1974

UNIVERSITY OF CALIFORNIA

Irvine

A Semantic Formalism and Associated Semantic Process for the Specification and Translation of Programming Languages

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Information and Computer Science

by

Marsha Drapkin Hopwood

Committee in charge:

Professor Fred M. Tonge, Chairman

Professor Julian Feldman

Professor Mark Finkelstein

1974

The dissertation of Marsha Drapkin Hopwood is approved, and it is acceptable in quality and form for publication on microfilm:

Committee Chairman

University of California, Irvine

1974

DEDICATION

This dissertation is dedicated to my parents, Ida and Sidney Drapkin.

CONTENTS

CONTENTS

ACKNOWLEDGMENTS

I would like to thank Professor Fred M. Tonge, my advisor, who provided technical direction and guidance as well as unfailing encouragement throughout my graduate education.

There are many others who have assisted in the work described here. In particular, I would like to thank Professor Julian Feldman, who provided advice and support. His tireless efforts as Assistant Chancellor for Computing created the campus computing system on which this dissertation and supporting programs were prepared. Gregory Hopwood, my husband, introduced me to some of the research tools I used and provided critical evaluation of this work. The Carnegie Corporation of New York, the School of Social Sciences, the Department of Information and Computer Science, and the Graduate Division of the University of California provided financial support.

VITA

October 31, 1944 - Born - Flushing, New York

1966 - B.S., Stanford University

1966-1970 - Teaching and Research Assistant, Department of Psychology and Department of Information and Computer Science, University of California, Irvine

1970-1972 - Lecturer, University of California, Irvine

1972-1973 - Research Associate, University of California, Irvine

ABSTRACT OF THE DISSERTATION

A Semantic Formalism and Associated Semantic Process for the Specification and Translation of Programming Languages

by

Marsha Drapkin Hopwood

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1974

Professor Fred M. Tonge, Chairman

Definitions of the semantics of programming languages are often incomplete and ambiguous. In particular, it may be difficult to determine what action is intended by a particular construct in some language. As a consequence, different implementations of that language may produce different results.

To help eliminate those differences which result from incomplete and ambiguous language definitions, a formalism for semantic specification and a semantic process are introduced. A semantic specification for a programming language can be viewed as a set of state transformations. The semantic process applies these state transformations to a program represented as a tree indicating the program structure and produces a computation tree representing the meaning of the program.

The semantic process can be viewed as part of a generalized table-driven translator for programming languages. Although such a translator may not be particularly efficient, its use produces translations which are consistent with the semantic specification and thus match the intent of the specifier. As a tool, the translator can be particularly useful for language debugging and tuning.

To illustrate the use of the semantic formalism, a semantic specification for Algol 60 and the results of its application to several Algol 60 constructs are given. A semantic process coded in Lisp and a cross-reference for the semantic specification by syntactic construct are also given.

Chapter 1

THE SEMANTIC PROBLEM FOR PROGRAMMING LANGUAGES

There are many problems which arise in the study and use of programming languages. Some of the least well understood of these relate to determining what action will be taken when a particular construct in some programming language is executed, that is, what the meaning, or semantics, of the construct is. This semantic problem manifests itself in several forms. Thus we find:

1. It is often difficult to determine what action is intended by a particular construct in some programming language, despite rather detailed specifications given in English. Consider the Algol 60 statements below:

```
x:=true;
a:=true;
if a ∨ f(x) then b:=x
```

Suppose the evaluation of procedure "f" has a side effect which changes the value of "x" to "false." Because of this, the value assigned to "b" depends on how the expression "a ∨ f(x)" is evaluated. If the entire expression is evaluated and then a test is done to determine if the expression is true, "b" will be assigned the value "false." If one notices

that the entire expression evaluates to "true" if any one of the components is "true," then, since "a" evaluates to "true," "f(x)" need not be evaluated. The value "true" will then be assigned to "b."

2. The action taken when a particular construct is executed may differ among the various translators for the language and among the various computers on which the translator is implemented. Various trade magazines have capitalized on these differences by running contests in which readers could submit a description of what they thought would happen when certain Fortran programs were executed. It is interesting to note that the final authority in such contests was frequently not a language specification, but rather a particular Fortran translator for a particular computer.
3. Even if there is no question about the meaning of the constructs in some programming language, producing a translator for that language is usually a major effort. One of the earliest algebraic translators, the Fortran compiler for the IBM 704 computer, took approximately 25 man years to produce. Despite devices like table-driven translators and other advances in our understanding of the processes involved in translation, compiler

construction is often still time-consuming.

The primary purpose of this dissertation is to provide a model for programming language semantics and the semantic aspect of the translation process in order to alleviate the kinds of problems just described.

To resolve the difficulty indicated in the Algol program given earlier, we have adopted an approach similar to that of Garwick (1966), namely, that the meaning of a program is determined by the translator for the programming language in which the program is written. This approach must be extended to provide a means for insuring compatibility among different implementations of translators for the same language. What a program does should not be idiosyncratic to the particular translator.

There are many reasons why different translators for the same language are incompatible. Some of these arise because the implementer may add refinements and extensions to a basic language specification. Elimination of incompatibilities arising in this way is a management problem and is not considered here. Other incompatibilities arise because of differences, such as word size, in the hardware of the target computers. While these differences are a real problem, they are in the domain of numerical analysis and are also not considered here. The remaining incompatibilities arise because implementers are forced to translate a semantic specification from a language like

English to some computer language. Specifications in English are not precise, for it is not the nature of English to encourage or demand precise statement.

The method we propose for eliminating machine-independent incompatibilities among translators for the same programming language is twofold. First, we extend the current notions of table-driven translators to include a table-driven semantic process. Second, we require that the language definition include a semantic specification capable of driving the semantic process. This not only provides a means for unambiguous semantic specification, but also simplifies the production of translators. Once the extended, or generalized, table-driven translator is implemented on a particular computer, specification tables for any programming language can be used to drive it.

The essence of this approach is to view a translator as a collection of language-independent processes and a language specification as a collection of process-independent state descriptions. The translator processes are lexical analysis, syntactic analysis, and semantic analysis. The language specification state descriptions are lexical, syntactic, and semantic specifications of the language to be translated. These serve as inputs to the corresponding processes. Another input to the translator is the program to be translated. The output is a computation tree, a tree structure which indicates in a machine-

independent fashion the actions to be taken when the program is executed. The remainder of the translation process is machine-dependent and involves either interpreting the computation tree or generating machine code for some computer.

While the notion of language-independent lexical and syntactic analysis is not a new one, the incorporation in a translator of a language-independent semantic analysis process which can be driven by a process-independent specification is. The notion of process-independent state descriptions for lexical and syntactic specification of programming languages is also not a new one. We can write lexical and syntactic specifications using formalisms knowing that these specifications can then be used to drive corresponding processes. These specifications are process-independent in that they do not depend on the particular algorithms implemented in the translator. The same should be true for semantic specification.

The simple example which follows is a preview of the notions of semantic specification and semantic analysis which are elaborated in the rest of this dissertation. Suppose we were dealing with the simple expression language whose syntax is given below in Backus Naur Form (Naur 1963).

<exp> ::= <term> | <exp> + <term>

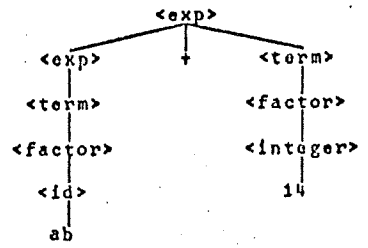
<term> ::= <factor> | <term> x <factor>

<factor> ::= (<exp>) | <id> | <integer>

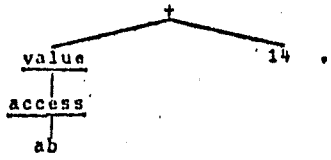
<id> ::= ab | c

<integer> ::= 5 | 14

The output of the syntactic analysis process of our translator for the expression "ab + 14" is the parse tree



The semantic analysis process takes this parse tree and the semantic specification and produces a computation tree as output. The elements of the semantic specification are trees which describe changes to be made to the parse tree by the semantic process. For an expression language these changes include rearranging the parse tree, inserting operators, and removing superfluous parse tree nodes. The resulting computation tree then is



The remaining chapters and appendices develop in detail the ideas of a semantic analysis process analogous to the syntactic analysis process and a semantic specification formalism analogous to the syntactic specification formalisms. Chapter 2 discusses approaches to translation and provides details on the programming language translation processes, with emphasis on semantic analysis. Chapter 3 examines the output of the semantic analysis process, a computation tree, and provides a framework for its interpretation by viewing it as a program for a computer whose basic data and control structures are trees. Chapter 4 describes the semantic specification formalism and illustrates its use with Algol 60 examples. Chapter 5 describes other approaches to semantic specification. Chapter 6 offers some conclusions about the usefulness of this semantic model. Appendix I gives a semantic specification for Algol 60. Appendix II illustrates the application by the semantic analyzer of the semantic specification to an Algol 60 program. Appendix III is a semantic analyzer coded in Lisp. Appendix IV gives a cross-reference of selected Algol 60 syntactic constructs and the semantic specification elements which relate to them.

Chapter 2

A TRANSLATOR MODEL

APPROACHES TO TRANSLATION

There are two basic approaches to writing translators. One approach is to tailor the translator to the particular language being translated and to the particular computer on which the translator is being implemented. Such translators are usually designed either to generate high quality code or to translate rapidly. The techniques used to produce such translators are frequently ad hoc. As a consequence it may take a considerable amount of time to produce a translator and it may be difficult to debug. A translator may also be extremely difficult to modify, since many parts of the translator may be affected by a change in the language being translated.

A second approach to creating translators is to write general purpose processors which are capable of doing some part of the translation process for a large class of programming languages. These processors require descriptions of the particular language being translated. For example, one can write a syntax analyzer, or parser, capable of parsing any language whose syntax can be described in Backus Naur Form (BNF). One such parser has

been described by Earley (1970).

It is quite time consuming to allow a parser to analyze each individual character of an input program. To avoid spending so much time parsing, one can also produce a general lexical analyzer which reads characters from the input program and forms them into basic units, or tokens, of the language being translated. Such a processor requires as input a description of the tokens of the language being translated. The reason time is saved is that the process of creating tokens is much simpler than the process of parsing, and thus each processor need only be powerful enough to do the general task required of it.

Unfortunately, the rest of the translation process is usually far less general. After some syntactically correct construct is recognized, the process of generating code or executing the construct is usually ad hoc. Internal translator structures must be manipulated and code specific to a particular computer must be generated or executed.

Up to the completion of syntactic analysis, this syntax-directed approach avoids the difficulties mentioned for the first ad hoc approach. How to produce lexical and syntactic analyzers of the syntax-directed approach is well understood. Algorithms are available in the computer science literature (Gries (1971) collects many of them); implementations of these algorithms are probably available in many computer installations. Implementation of the

semantics of a language is still time consuming, and the difficulties of debugging and modifying are still present for this component. For the earlier stages of the translation process, however, all that is necessary is to produce a correct specification of the language. Debugging and modifying is simply a matter of correcting a lexical or syntactic specification, not of changing a processor.

If there were no new difficulties introduced by syntax-directed translation, the technique would probably be more widely used. However, the use of general processsors for part of the translation process often provides more power than is necessary for any particular language being translated. Hence these translators may be considerably slower than those produced using the first approach.

A gain in speed may be attained by mechanically producing lexical and syntactic analyzers tailored to the particular language being translated. Johnson, Porter, Ackley, and Ross (1968) describe a system which generates efficient lexical processors for languages from descriptions of the lexicon. Such processors are approximately half as fast as those hand-tailored for the particular language, but the authors claim that this difference can be reduced considerably.

Much improvement can be made in syntactic analysis. Earley's parsing algorithm can handle all context free grammars. For programming language grammars, that is,

unambiguous context free grammars, the time taken to parse a program is proportional to the length or the square of the length of the program, depending on the particular grammar. This is the most efficient general algorithm described to date and appears to be about as good as can be achieved. For particular grammars which possess certain properties, parsers can be constructed mechanically to run in time proportional to the length of the program with a proportionality constant quite a bit smaller than in Earley's algorithm. This gain is possible because the grammar is used to build tables which incorporate the necessary information from the grammar. The grammar is then discarded and the parser operates from these tables. Wirth and Weber (1966) describe a scheme for mechanically constructing parsers for a class of programming languages whose grammars, called precedence grammars, make it possible to parse without backtracking. They describe facilities to detect if the grammar is in the correct form, to indicate where violations of this form (precedence violations) occur, and to construct the precedence tables used to drive a bottom-up parser, the algorithm for which is also provided. Such a parser is as fast, or very nearly so, as would be possible with a hand-tailored parser. Similar schemes are available for top-down parsers. The only disadvantage of these schemes is that the grammar may not be in an acceptable form and the modifications necessary to put the

grammar in that form may either be impossible, hence requiring a more general parser, or may sufficiently distort the grammar to make code generation somewhat difficult.

A GENERAL SEMANTIC ANALYZER

The model for translation proposed here extends the notion of a syntax-directed translator to include a general processor for semantic analysis. This semantic analyzer has two inputs, a parse tree (output by the syntactic analyzer) and a specification of actions modifying the parse tree to be taken when certain subtrees are found in the parse tree. Thus, the parse tree represents the particular program being translated and the specification represents the semantic rules for the language in which the program is expressed. Each element of the semantic specification consists of two trees. The specification may be viewed as a set of state transformations. That is, if some part of the parse tree is in some particular state (reflecting some particular syntactic structure), that part of the parse tree is transformed into some other state (reflecting some modifications to the structure), which expresses the meaning associated with that structure. The output of the semantic processor is a computation tree, which represents the meaning of the program input to the translator. Figure 1 illustrates the semantic process.

The computation tree may be viewed as a program for a "simple" interpreter, since the tree nodes represent simple

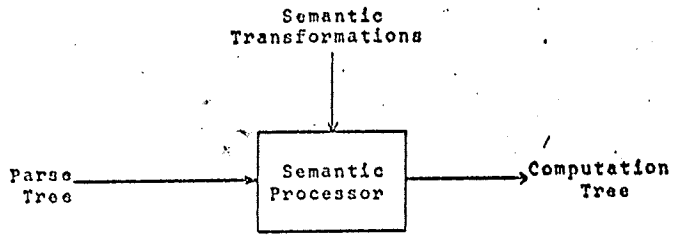


Figure 1. The semantic process.

operators (e.g., addition and transfer of control) or operands. Alternatively, the computation tree may be viewed as a program for a computer which has a tree as its basic data and control structure, as opposed to the linear structure present in most current computers. Berkling (1971) describes a proposed computer based on binary trees.

The semantic processor itself determines how to carry out the state transformations described by the semantic specification and then carries them out. Since each element of the specification consists of two trees, the processor seeks to find the first of these trees in the parse tree and if successful replaces that part of the parse tree by the second tree in the specification element. The elements of the specification are ordered, although this ordering may in most cases be arbitrary. Each element of the specification is applied as many times as it is applicable and then discarded. Then the next element is considered.

The notion to be emphasized is that the semantic specification is analogous to the lexical and syntactic specifications, that is, it describes state transformations. How these transformations are made, the analysis process, is the province of the processors and need not concern the individual who specifies the language.

INCLUSION OF SEMANTICS IN SYNTACTIC SPECIFICATIONS

Let us consider language specification and translation from the standpoint of the separation of specification and process. Usually a language definition includes a syntax specification given in something like BNF and a semantic specification given in English. For example, the syntax of Algol 60 (Naur 1963) defines syntactically correct language constructs. The form of the syntactic specification facilitates its use as input to a parser but the actual details reflect more than syntax. There are at least three kinds of non-syntactic concerns included in the syntax:

1. Many metavariables are included in the syntax so that they may be discussed under semantics.
2. Some metalinguistic formulae (syntactic equations) are structured to facilitate code generation.
3. The meaning of certain constructs is included in the syntax.

Duncan (1966) points out that about one fourth of the metavariables found in the Algol 60 syntax are there so that they can be discussed under semantics. For example,

consider the definition of a <formal parameter>:

<formal parameter> ::= <identifier> .

This definition could easily be eliminated; it adds nothing to the clarity of the syntax. It is there so that some semantics (meaning) can be attached to an occurrence of a formal parameter which is not appropriate to an identifier.

The structure imposed by the Algol 60 syntax is somewhat artificial since many metavariables were introduced to facilitate code generation. For example, consider the definition of a <for statement>:

<for statement> ::= <for clause> <statement>

<for clause> ::= for <variable> := <for list> do .

The definition of a <for clause> appears to have been introduced so that the recognition of a metalinguistic formula can be completed before the recognition of a <statement> in a <for statement>. This allows an internal label to be placed before the <statement> in the code generated for the <for statement>.

Sometimes semantics are imbedded in the Algol 60 syntax. An example of this is seen in the syntax for arithmetic expressions. Ignoring unary operators, one might normally define an expression as an operand, or a sequence of two or more operands separated by operators. The Algol 60 syntax, however, is much more structured, because the precedence of operators is imbedded in the syntax. One can argue that precedence of operators is not a semantic issue

or that because it is easier to handle precedence in syntax than in semantics, precedence considerations belong in the syntax. No matter what viewpoint one takes, however, it is clear that more structure than is necessary to define the form of expressions has been introduced.

Thus it can be seen that language specifications can contain more information than is necessary to define a language. Some of this additional detail may be burdensome to produce. The separation of specification and process as proposed here eliminates some of the need for extra structure. In particular, the additional detail added to the syntax to facilitate code generation may not be necessary, since transformations do not conform to any notions of scope associated with code generators by the designers of Algol 60 and thus may incorporate as much or as little context as is necessary. This does not result in a shift of structure from the syntax to the semantics, but rather allows what structure is required to be introduced where it can be done with the least effort. Further, if the formal language specification designed to drive a translator is distinct from user documentation, it is not necessary to include structure in the formal syntax to facilitate discussion under semantics.

A TRANSLATOR MODEL

Figure 2 is a diagram of the flow of information and control in the proposed translator. The inputs and outputs of the processors have already been described and are indicated by solid lines in the diagram. Flow of control is indicated by broken lines. The lexical processor reads the first several characters of the program, creates a token, and passes the token and control to the syntactic processor. The syntactic processor uses this token in beginning the parsing process. When this processor can proceed no further without an additional token, control is returned to the lexical processor, which then reads a few more characters. If the syntactic processor has recognized its goal, that is, a program, then the process is considered completed. A parse tree is output and control is passed to the semantic processor. The semantic processor then applies all the state transformations to the parse tree which apply and when there are no transformations left to apply, the process is considered completed and a computation tree is output.

In a slightly different model the syntactic processor can pass control and a partial parse tree to the semantic processor each time a syntax equation is recognized. The semantic processor can then apply all the applicable state transformations to the parse tree and return control and the transformed tree to the syntactic processor. When a program has been recognized and all the applicable state

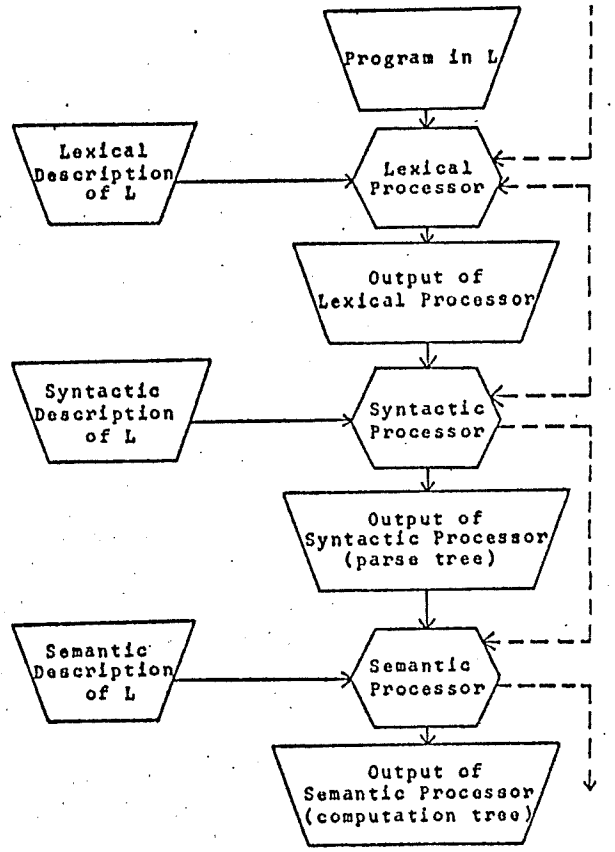


Figure 2. Flow of information and control in the translator.

transformations have been applied, the translation is considered completed and a computation tree is output by the semantic processor.

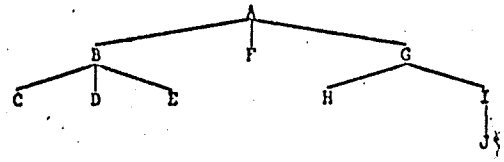
In both models, no claim of efficiency is made. Alternating between syntactic and semantic processing may result in some saving of space, since a complete parse tree need not be constructed before semantic processing begins. Other means of increasing efficiency are discussed later, but the proposed translator is not intended, for now, to compete with production translators.

The only part of this translator that is substantially different from other translators is the semantic processor. This processor is really a sophisticated pattern matching and replacement mechanism similar to that found in languages like Snobol; however, the semantic processor works with tree structures, that is, a parse tree and the elements of the semantic specification, rather than character strings. We shall now consider the details of the operation of the semantic processor.

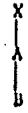
THE SEMANTIC MATCHING PROCESS

In attempting to match an element of the semantic specification to the parse tree, the semantic processor traverses the relevant trees in preorder. (A preorder traversal consists of visiting the root and then traversing the subtrees of that root left-to-right in preorder. The leftmost subtree is traversed first, then the second from

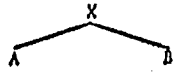
the left and so on until the rightmost subtree is traversed. For the tree



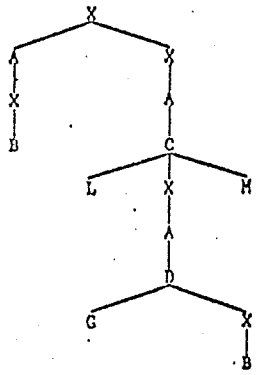
the nodes would be visited in a preorder traversal in the order A, B, C, D, E, F, G, H, I, J.) The processor starts at the root of the parse tree and at the root of the first tree, called the match tree, in the semantic specification element. The parse tree is traversed in preorder until a node is found which matches the root of the match tree. If no such node is found, the processor fails, the current specification element is discarded, and the next element is considered. If such a node is found, the processor continues traversing both trees simultaneously, attempting to match the nodes in the parse tree with the corresponding nodes in the match tree. This matching requires not only a match of symbols stored at the nodes, but a structural match as well. For example, the match tree



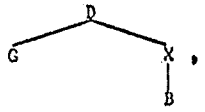
cannot be found in the parse tree



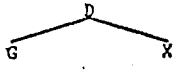
The processor succeeds if this node-by-node comparison matches a subtree of the parse tree, or the entire parse tree, with the match tree. For example, given the parse tree



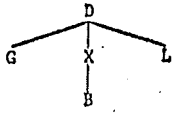
and the match tree



a successful match is possible. However, if the match tree

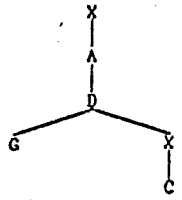


or the match tree



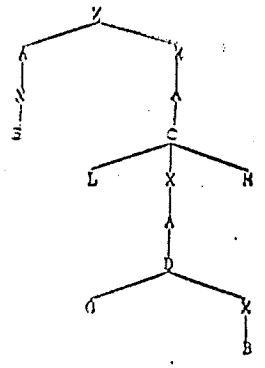
is used, the attempt to match fails.

In the first of these two failures, the match tree does not match a complete subtree of the parse tree. In the second, the parse tree is exhausted while there are still nodes to be matched in the match tree. An initial attempt to match may also fail if corresponding nodes do not match, as would happen if the match tree

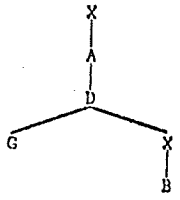


is used. These failures do not mean that the processor fails to make a match using the specification element, but rather that this attempt fails and other areas of the parse tree must be considered.

Subsequent match attempts begin in the parse tree at the node following the node matched with the root of the match tree in the previous attempt. For example, given the parse tree



and the match tree

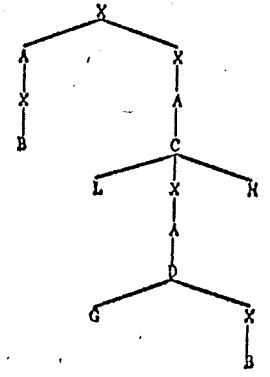


the processor succeeds on the fourth attempt. A match attempt is begun at each of the first four nodes labelled "X" visited in a preorder traversal, but only the match attempt begun at the fourth node "X," i.e., the one which is a son of "C," is successful.

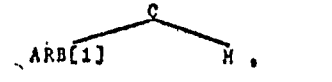
SPECIAL MATCH TREE NODES

It is frequently cumbersome, if not impossible, to specify an entire subtree of the parse tree in the match tree. For this reason eight special node types are defined for use in the match tree. These nodes, which are subscripted to distinguish different instances of them, are:

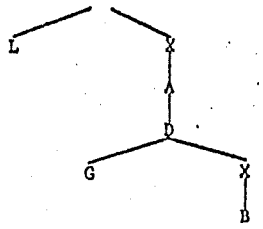
1. NON[I]--A nonterminal node matches any nonterminal symbol of the syntax found in a parse tree. If the syntax is given in BNF, NON matches anything enclosed in "<" and ">".
2. SYM[I]--A symbol node matches any symbol at a node in a parse tree.
3. TERM[I]--a terminal node matches any terminal symbol of the syntax, that is, any symbol of the source language.
4. ARB[I]--The father of an ARB node in a match tree is matched to a parse tree node. ARB matches zero or more sons (and their complete subtrees) of the parse tree node. Thus ARB matches an arbitrary structure. For example, given the parse tree



and the match tree

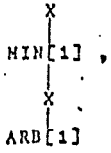


ARB[1] matches

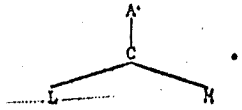


Because it matches complete subtrees, an ARB node does not have any sons.

5. MIN[I]--A minimum node matches the minimum depth (possibly null) section of a subtree between what matches the father of MIN and the son of MIN. What matches the son of the MIN node must be a descendant of a node included in what matches the MIN node. For example, given the same parse tree as above and the match tree



MIN[1] matches

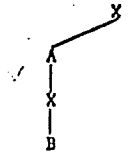


If MIN[1] is matched to

A

in the left subtree of the parse tree, the match

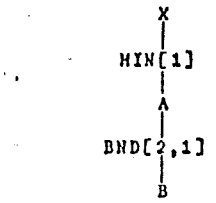
attempt for the entire match tree fails. This failure occurs because a match tree must match an entire subtree of the parse tree and this particular match for MIN[1] causes the match tree to be matched to



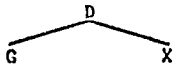
Since that is not a complete subtree of the parse tree, the attempt fails.

A MIN match is determined by two nodes, the node that matches the father of the MIN and the node that matches the son of the MIN. (A MIN node must have exactly one son.) Although the node that matches the father of a MIN node may have several subtrees, only one of these can be involved in a MIN match. The particular subtree involved is the one containing the node matching the son of the MIN node; the other subtrees must be matched by other match tree nodes.

6. BND[J,I]--This is the same as MIN with the additional constraint that the match is bounded, that is, what matches BND cannot include, in order, all the J nearest ancestors of what matches BND. For example, given the same parse tree as above and



MIN[1] matches null and has depth zero. BND[2,1] matches

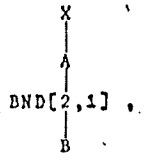


and the 2 nearest ancestors of what matches BND[2,1] are

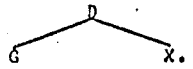
7. IL[I]--It is often necessary in translation to generate unique symbols for use as internal labels. If there is a process, a symbol generator, which returns a unique symbol each time it is invoked, an IL node will match any symbol so generated.

8. S[I1, I2, . . ., In]--"S" is a character string, possibly null, and different from "NON," "SYM," "TERM," "ARB," "MIN," "BND," or "IL." Each of the Ij is either a symbol or the sum or difference of two symbols. S[I1, I2, . . ., In] is called a subscripted node. An example of such a node is bh[i+1,7]. A subscripted match tree node matches a parse tree node if the character strings are the

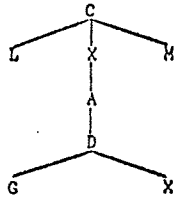
the match tree



BND[2,1] matches



BND[2,1] does not match the structure



since the 2 parse tree ancestors of this structure, namely,

are contained in the structure.

If a MIN node occurs in the J nearest ancestors of a BND node, what matches the MIN node may have a tree depth other than one. This depth is counted in determining the J ancestors. For example, given the same parse tree as above and the match tree

same in both and for each pair of corresponding subscripts, one of the following holds:

a. Both are the same number.

b. The subscript in the match tree node is a string of characters which has been matched during this attempt to a number and that number is the same as the corresponding subscript in the parse tree.

c. The subscript in the match tree node has not been previously matched in this attempt, in which case it is matched to a number, i. e., the corresponding subscript in the parse tree.

For example, the match tree node

M[I,2]

matches the parse tree node

M[5,2].

If the node

M[3,I]

occurs later in the match tree and the corresponding node in the parse tree is

M[3,4],

then the match attempt is not successful since I has been matched to 5 earlier in this attempt.

With the addition of these special nodes, particularly the ARB, MIN, and BND nodes, the matching procedure must be

modified slightly. If at any point in the process a match for the next node in the match tree cannot be made, an attempt is made to extend the most recent ARB, MIN, or BND match. If this is successful, the match process continues from that point. If an ARB, MIN, or BND cannot be found which can successfully be extended, a new attempt to match is made as before.

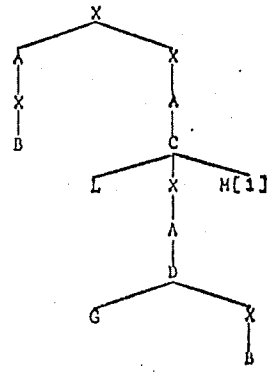
THE SEMANTIC SUBSTITUTION PROCESS

When a match is successfully completed, the second tree, called the substitution tree, of the specification element, is considered. Each node in the substitution tree is visited. If it is not one of the eight types of special nodes described above, no action is required and the next node is visited. If the node is an IL node and it has been matched to a node in the parse tree, it is replaced by the parse tree node. Otherwise, the symbol generator is invoked, the IL node is replaced by the newly generated symbol and a match between the IL node and the generated symbol is recorded. If the node is a subscripted one, each of the non-numeric subscripts is replaced by the number to which it was matched. Simple expressions of the form A + B or A - B are also allowable subscripts and they are evaluated when replacement is done.

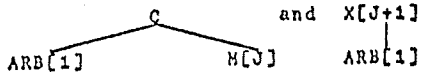
After all the nodes in the substitution tree are visited and replaced as necessary, the substitution tree is inserted into the parse tree in place of the subtree of the

parse tree matched to the match tree.

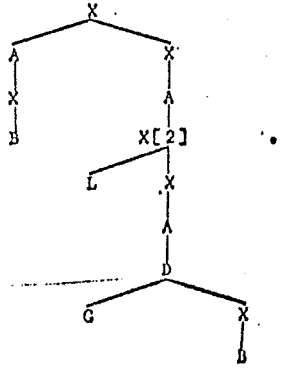
An example of the results of the match and substitution processes is shown below. Given the parse tree



and the match and substitution trees



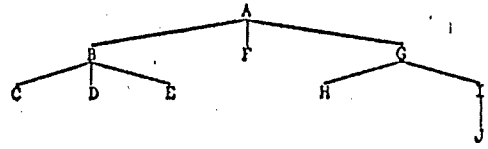
respectively, the resulting tree is



Chapter 3

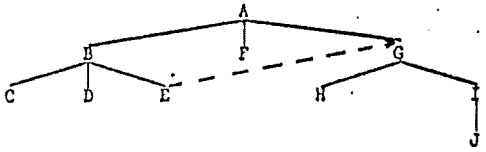
THE TREE MACHINE

A computation tree can be viewed as a program for a computer which has a tree as its basic data and control structure, as was mentioned in the last chapter. The nodes of the tree are operators and operands. The sons of an operator node are its operands. A tree machine program, that is, a computation tree, is executed by traversing the tree in endorder. As each node is encountered, those that are operands are placed on an evaluation stack and those that are operators are executed. An endorder traversal starting at the root of a tree consists of traversing the subtrees of that root, left-to-right, in endorder and then visiting the root. For the tree



the nodes would be visited in an endorder traversal in the order C, D, E, B, F, H, J, I, G, A. The order of execution may be changed by operators which transfer control to other parts of the tree. The execution of these operators, which resemble the branch and subroutine call operators of

conventional machines, is sometimes aided by the inclusion of threads in the computation tree. These threads are auxiliary links and are indicated by dotted lines. In the tree



the node E is threaded to the node G. This thread may be used by an operator to change the execution order. The next node is then obtained using the thread rather than following the normal traversal order. If the operator is a conditional one, the thread may or may not be used.

The tree machine operators perform three kinds of operations: standard operations, passive operations, and runtime administration operations. The standard operations resemble the operations of conventional machines, for example, add, subtract, load, and store.

The passive operations have no analog in conventional machines, although they may be compared with no-operation instructions. The operators of this class serve several purposes. They may serve as a collection point for (ancestor of) a series of related subtrees, such as all the statements in a block. They may connect the components of a construct, such as the clauses in a conditional expression (if-then-else construct). They may also position

computations in the tree so that the desired effect will be achieved by endorder traversal and execution. No matter what the purpose of a passive operator may be, no action is taken when a tree node containing such an operator is visited.

The runtime administration operations, as their name indicates, manipulate the structures necessary to maintain the appropriate environment during execution of the computation tree. Some of these operations correspond to operations often performed by operating systems. An example is the allocation and deallocation of a block of data storage. Many of the basic services provided by operating systems can be treated as extensions of the computer order code; the user, in fact, need not know that these services are not hardware operations.

Other runtime administration operations, such as saving pointers to the current environment when entering a new environment, are not functions normally performed by operating systems, but are nonetheless necessary for the correct execution of a block structured program. At least one computer now in use, the SYMBOL machine (Rice and Smith 1971), includes both the standard and the runtime administration operations in the order code.

Most of the individual operators in the tree machine order code perform only one of the three kinds of operations (standard, passive, or runtime administration) mentioned. A

few operators, however, perform both standard and runtime administration operations. An example is the operator "to," which performs the standard operations involved in transferring control to another location and if necessary also performs the runtime administration operations involved in exiting one or more blocks.

Both the standard and runtime administration operations make use of some auxiliary structures in addition to the computation tree. The standard operations use an evaluation stack, which provides temporary storage for operands, and a temporary variable stack, which provides somewhat longer term storage than the evaluation stack. The runtime administration operations use a block activation stack, which contains information for all blocks which have been entered but not exited, and a block accessibility stack, which contains information necessary to resolve references to variables defined outside the block currently being executed. An additional structure, the tree traversal stack, is part of the control function of the tree machine, but may also be affected by the runtime administration operations.

The evaluation stack functions in the same manner as the stack in a stack (zero address) computer. Each time an operand is encountered it is placed on the evaluation stack; whenever an n-ary operator is to be executed, the topmost n elements of the evaluation stack are the operands. Some

such temporary storage is necessary if the computation tree is not to be modified during execution.

The temporary variable stack provides temporary storage, when necessary, for the values of actual arguments in a procedure call. In most schemes for handling argument passing, addresses of some sort are passed. In some situations, for example when an actual argument is an expression consisting of more than a single variable, the actual argument has no address. The value of the expression, once computed, can be stored on the temporary variable stack and readily referenced. Although temporary storage could be provided by using the evaluation stack or by allocating temporary variables along with programmer-defined variables at block entry, the use of a separate temporary variable stack clearly distinguishes this utilization of storage from others and provides a straightforward solution.

The block activation stack contains a chain of pointers connecting, in reverse order of creation, the data areas for blocks which have been entered but not exited. Each data area contains, in addition to storage for local variables, information relevant to resumption of execution after the completion of a block, such as a return address.

The block accessibility stack contains, for each block B which has been entered but not exited, a chain of pointers to the data areas for those blocks whose data areas would be

accessible from B if B were the block currently being executed. This stack provides information needed by the current block. It is used to resolve global references, that is, references to variables defined outside the current block.

In some instances the chains of pointers in the block accessibility and block activation stacks are the same. Differences in these stacks can occur if nonlocal variables in a block are defined by the environment in which the block is defined, rather than the environment in which the block is executed. In particular, the situation causing differences in the block accessibility and block activation stacks can occur in Algol 60 when a procedure is called. Consider the Algol 60 program below:

```
begin real a;
      procedure X;
L1:   begin real b;
      H;    b := a;
            . . .
      end X;
L2:   begin real a;
      L3:   X;
            . . .
      end
end
```

Assume that the outermost block is labeled "L0." The procedure "X" is defined in block "L0" and the variable "a" in the statement labeled "H" refers to the "a" declared in the first line of the program. The procedure "X" is called in the statement labeled "L3" which occurs in block "L2." There is also a variable "a" declared in block "L2." This

variable is not accessible during the execution of "X." At label "H," the block accessibility stack entry for the current block contains pointers to the data areas for blocks "L1" and "L0," while the block activation stack contains pointers to the data areas for blocks "L1," "L2," and "L0."

The tree traversal stack and the pointer to the current tree node are similar to the instruction pointer (program counter) and pointer update mechanism of conventional computers. Updating the current tree node pointer is the same process as visiting the next node in an endorder traversal.

It should be noted that the various stacks are functionally separate. This does not preclude the possibility of combining several of them in a single physical structure.

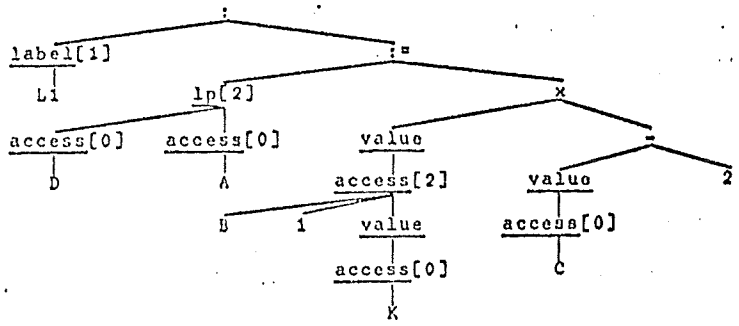
The example which follows illustrates several of the tree machine operators and indicates how the evaluation stack is used. The example gives an Algol 60 construct, the corresponding computation tree, and a step-by-step execution trace. The execution trace indicates the node being visited, the action taken when the node is visited, and the contents of the evaluation stack after the node is visited. The computation tree represents a program fragment and differs in a few details from a computation tree for a complete program. The most noticeable difference is that variable names appear in the example, while in a tree for a

complete program, variable names are replaced by references to data block numbers and displacements within blocks.

EXAMPLE

Algol 60 construct: L1: D:=A:=B[1,K]x(C-2)

Computation tree:



Execution trace:

Node: L1
Action: Push the symbol "L1" onto the evaluation stack.
Stack: L1 <--

Node: label[1]
Action: Pop the evaluation stack 1 time.
Stack: <--

Node: D
Action: Push the symbol "D" onto the evaluation stack.
Stack: D <--

Node: access[0]
Action: Replace the symbol on top of the evaluation stack by the address of the variable named by that symbol.
Stack: address of D <--

Node: A
Action: Push the symbol "A" onto the evaluation stack.
Stack: A <--
address of D

Node: access[0]
Action: Replace the symbol on top of the evaluation stack by the address of the variable named by that symbol.
Stack: address of A <--
address of D

Node: lp[2]
Action: push 2, the number of left parts, onto the evaluation stack.
Stack: 2 <--
address of A
address of D

Node: B
Action: Push the symbol "B" onto the evaluation stack.
Stack: B <--
2
address of A
address of D

Node: 1
Action: Push the number 1 onto the evaluation stack.
Stack: 1 <--
B
2
address of A
address of D

Node: K
Action: Push the symbol "K" onto the evaluation stack.
Stack: K <--
1
B
2
address of A
address of D

Node: access[0]
Action: Replace the symbol on top of the evaluation stack by the address of the variable named by that symbol.
Stack: address of K <--
1
B
2
address of A
address of D

Node: value
Action: Replace the address on top of the evaluation stack by the value of the item at that address.
Stack: value of K <--
1
B
2
address of A
address of D

Node: access[2]
Action: Replace the name (third item from the top) of the 2-dimensional array by the address of the array element described by T3[T2,T1], where T1 is the top of the stack, T2 is the second item from the top of the stack, and T3 is the third item from the top. Pop the evaluation stack twice.
Stack: address of B[1,K] <--
2
address of A
address of D

Node: value
Action: Replace the address on top of the evaluation stack by the value of the item at that address.
Stack: value of B[1,K] <--
2
address of A
address of D

Node: C
Action: Push the symbol "C" onto the evaluation stack.
Stack: C <--
value of B[1,K]
2
address of A
address of D

Node: access[0]
Action: Replace the symbol on top of the evaluation stack by the address of the variable named by that symbol.
Stack: address of C <--
value of B[1,K]
2
address of A
address of D

Node: value
Action: Replace the address on top of the evaluation stack by the value of the item at that address.
Stack: value of C <--
value of B[1,K]
2
address of A
address of D

Node: 2
Action: Push the number 2 onto the evaluation stack.
Stack: 2 <--
value of C
value of B[1,K]
2
address of A
address of D

Node: -
Action: Subtract the value on top of the evaluation stack from the value at the next to the top. Store the result in the next to the top. Pop the stack.
Stack: value of C-2 <--
value of B[1,K]
2
address of A
address of D

Node: ×
Action: Multiply the value at the next to the top by the value on top of the evaluation stack. Store the result in the next to the top of the stack. Pop the stack.
Stack: value of B[1,K]x(C-2) <--
2
address of A
address of D

Node: :=
Action: Assign the value on the top of the evaluation stack to the number of items indicated by the next to the top. Addresses of these items begin two down from the top. Pop the evaluation stack n+2 times where the next to the top of the evaluation stack contains n.
Stack: <--

Node: :
Action: no action
Stack: <--

The following tables list each of the tree machine operators, the number of operands its takes (i.e., the number of sons it has in the computation tree), and its function. The operators are grouped as standard, passive, or runtime administration operators, although those preceded by asterisks appear in more than one group. These operators are sufficient for, and somewhat based on, Algol 60. Other languages might require some different or new operators, although the overall approach would remain the same.

TABLE 1: STANDARD OPERATORS

Operator	Number of Operands (sons)	Function (S1,S2, . . . denote sons left to right)
↑	2	Exponentiation: S1↑S2.
×	2	Multiplication: S1×S2.
/	2	Division: S1/S2.
+	2	Addition: S1+S2.
-	2	Subtraction: S1-S2.
=	2	Equality test: S1=S2.
>	2	Greater than test: S1>S2.
≥	2	Greater than or equal test: S1≥S2.
<	2	Less than test: S1<S2.
≤	2	Less than or equal test: S1≤S2.
≠	2	Inequality test: S1≠S2.
¬	1	Negation: ¬S1.
∧	2	And: S1∧S2.
∨	2	Or: S1∨S2.
⊃	2	Implication: S1⊃S2.
≡	2	Equivalence: S1≡S2.
:=	2	Assignment: Assign the value, S2, to the n locations which are sons of S1.
access[n]	n+1	Get address: Find the address of the variable S1 (if n=0) or array element S1[S2, . . ., S(n+1)] (if n>0).
args[n]	n	Procedure call argument collector: Collect as sons of this node and count the number, n, of arguments in a procedure call.

*call	2-3	Subroutine call: Invoke the subroutine named by the rightmost son with arguments indicated by the sons of the next to the rightmost son. Take all appropriate environment adjustment actions.
erase	1	Erase stack: Erase the temporary stack up to and including the last mark.
halt	0	Stop execution.
if	1	Conditional evaluation: Evaluate S1. If true execute next right brother. If false execute rightmost brother.
label[n]	n	Label collector: Collect as sons of this node and count the n labels, S1, . . ., Sn, on a statement.
lp[n]	1	Left part collector: Collect as sons of this node and count the number, n, of left parts in the left part of an assignment statement.
mark	0	Mark stack: Put a mark in the temporary stack.
null	0	No operation.
pusht	1	Push stack: Push the temporary stack.
*return	0	Subroutine return: Transfer control to the calling routine and restore the previous environment.
*to	1	Unconditional branch: Transfer control to the statement in the computation tree whose label is S1. Take all appropriate block exit actions before control is transferred.
value	1	Get value: Obtain the value at the location indicated by S1.

TABLE 2: PASSIVE OPERATORS

Operator	Number of Operands (sons)	Function (S1,S2, . . . denote sons left to right)
:	2	Labels-statement connector: Associate the labels which are the sons of S1 with the statement S2.
ap	3	Program-symbol tables connector: Connect the computation tree program, the program-symbol table separator, and the symbol table tree for the program.
arg[i]	variable	Argument node for ith argument: Connect all the parts of the ith argument of a procedure call.
bp[n]	n	Array dimension collector: Collect as sons of this node all n lower bound-upper bound pairs given as the dimensions of an array.
cond	3	Conditional expression and conditional statement connector: Connect the condition (S1), the son to execute if the condition is true (S2), and the son to execute if the condition is false (S3).
mixed[n]	n	Statement-expression collector: Collect as sons of this node n statements and expressions, where the collection is to be treated as a unit.
pair	2	Array dimension lower-upper bound connector: Connect the lower bound and upper bound of an array dimension.
pblock	2	Procedure name-formal parameter block connector: Connect a procedure name to the symbol table for the formal parameters in the procedure.
st[i,j,k,l,m]	n	Symbol table (symbol collector): Collect as sons of this node symbols in block i at level j where k = number of units of variables l = number of units of own variables m = block number of father block.

stat[n]	n	Statement collector: Collect as sons of this node n statements, where the collection is to be treated as a unit.
symbol	n	Symbol-attributes connector: Connect a symbol and its n-1 attributes.

TABLE 3: RUNTIME ADMINISTRATION OPERATORS

Operator	Number of Operands (sons)	Function (S1,S2, . . . denote sons left to right)
bh[i,j]	0	Block head: Save the current environment. Allocate space for variables. Allocate space for own variables if this is the first time the block has been entered.
bt[i,j]	variable	Block tail: Deallocate space for variables local to this block. Restore the previous environment.
*call	2-3	Subroutine call: Save the current environment including the return address. Transfer control to the routine being called.
codeh	0	Code head: Indicate the beginning of a code procedure body. The syntax and semantics of this code are machine-dependent and are not specified in Algol 60.
codet	variable	Code tail: Indicate the end of a code procedure body. The syntax and semantics of this code are machine-dependent and are not specified in Algol 60.
*return	0	Subroutine return: Restore environments as necessary. Go back to point of call.
*to	1	Unconditional branch: Deallocate space for variables and restore environments as necessary. Transfer control to statement labelled S1.

The following tables list each of the standard and runtime administration operators and its effect on the various stacks. The passive operators have no effect on any of the stacks.

TABLE 4: STANDARD OPERATORS

Operator	Effect on Stacks (T1, T2, . . . denote evaluation stack elements from the top down.)
↑	T2:=T2↑T1. Pop the evaluation stack.
×	T2:=T2×T1. Pop the evaluation stack.
/	T2:=T2/T1. Pop the evaluation stack.
+	T2:=T2+T1. Pop the evaluation stack.
-	T2:=T2-T1. Pop the evaluation stack.
=	T2:=true if T2=T1, false otherwise. Pop the evaluation stack.
>	T2:=true if T2>T1, false otherwise. Pop the evaluation stack.
≥	T2:=true if T2≥T1, false otherwise. Pop the evaluation stack.
<	T2:=true if T2<T1, false otherwise. Pop the evaluation stack.
≤	T2:=true if T2≤T1, false otherwise. Pop the evaluation stack.
≠	T2:=true if T2≠T1, false otherwise. Pop the evaluation stack.
¬	T1:=true if ¬T1 is true , false otherwise.
∧	T2:=true if T2∧T1 is true, false otherwise. Pop the evaluation stack.
∨	T2:=true if T2∨T1 is true, false otherwise. Pop the evaluation stack.
⊃	T2:=true if T2⊃T1 is true, false otherwise. Pop the evaluation stack.
≡	T2:=true if T2≡T1 is true, false otherwise. Pop the evaluation stack.
:=	Store T1 at the locations whose addresses are given by Ti, for 3≤i≤T2+2. Pop the evaluation stack T2+2 times.

access[n] T(n+1):=address of T1 (if n=0) or address of the array element T(n+1)[Tn, . . . T2, T1] (if n>0). Pop the evaluation stack n times.

args[n] Push the number, n, of arguments in a procedure call onto the evaluation stack.

***call** Save the current environment, including the pointer to the next computation tree node, in the accessibility and activation stacks. Enter the block in which the procedure being called is defined, allocating any space defined for the block (formal parameter space) and creating new entries in the accessibility and activation stacks. Copy the subroutine call arguments from the evaluation stack into the formal parameter slots. Put a block mark in the traversal stack and traverse the computation tree without executing until the ";" of the construct which has the procedure name as label is reached. (This same point is reached by following the thread from the procedure name in the symbol table.)

erase Erase the temporary stack up to and including the last mark.

halt No effect.

if Take the appropriate descendant of the node on top of the traversal stack (the middle son if the top of the evaluation stack is true, the right son if it is false). Pop the traversal stack if the right son is selected. Change the computation tree pointer. Pop the evaluation stack.

label[n] Pop the evaluation stack n times.

lp[n] Push the number, n, of left parts in an assignment statement onto the evaluation stack.

mark Put a mark in the temporary stack.

null No effect.

pusht Push the temporary stack.

***return** Deallocate storage for the procedure block. Pop the accessibility and activation stacks and restore the old computation tree pointer. Pop the traversal stack until a block mark is removed.

*to	Extract the block number and level from T1. If the block number and level are not the same as for the current block, do actions for bt (block tail) and pop traversal stack to last block mark until the block number and level are the same. Go to node bh (block head) for the block in which T1 is defined and traverse the computation tree without executing (or stacking operands) until the ";" of the labeled construct is reached. (This same point is reached by following the thread from the label in the symbol table.)
value	T1:=value at the address T1.

TABLE 5: RUNTIME ADMINISTRATION OPERATORS

Operator	Effect on Stacks
bh[i,j]	Make new entries in the block accessibility and activation stacks. Allocate space for variables local to the block, copying values of own variables if necessary. Put a block mark in the traversal stack.
bt[i,j]	Deallocate space for variables local to the block, saving values of own variables if necessary. Pop the block accessibility and activation stacks.
*call	Same as under STANDARD OPERATORS.
codeh	The effect of code procedures is machine-dependent and hence the effect of this operator is also machine-dependent.
codet	The effect of code procedures is machine-dependent and hence the effect of this operator is also machine-dependent.
*return	Same as under STANDARD OPERATORS.
*to	Same as under STANDARD OPERATORS.

Chapter 4

SEMANTIC SPECIFICATION WITH EXAMPLES FROM ALGOL 60

A complete semantic specification for Algol 60 is given in Appendix I. This chapter illustrates the kinds of functions performed when these specifications are applied to a parse tree. It also gives some indication of the effort required to produce a semantic specification for another language.

The functions performed by the application of the semantic transformations are:

1. removing superfluous syntax--eliminating those nodes from a parse tree which are not meaningful in arriving at a computation tree;
2. making operations explicit--putting nodes into a tree which are implied but need to be made explicit at execution;
3. counting--determining the number of instances of a construct and grouping them together so they can be treated as a unit;
4. specifying order of execution--arranging nodes in a tree so that they are in the proper order for execution; and
5. copying--moving information from one part of a tree

to another or replicating information, usually for the purpose of gathering information in one place.

The semantic functions are illustrated below. In the example trees, the metalinguistic variables of Algol 60 have been replaced by two-letter abbreviations. Appendix I gives the correspondence between the abbreviations and the actual metalinguistic variables.

One or more of the semantic functions may be present in an element (pair of trees) of the semantic specification. In the percentages given below for indicating the number of elements including the various functions, each element which contains more than one function is counted once for each function. Thus the percentages total more than one hundred.

REMOVING SUPERFLUOUS SYNTAX

Many of the nodes in the parse tree for an Algol 60 program can be eliminated in producing a computation tree. The nodes may be superfluous for several reasons. These include:

1. The nodes were used in some transformation and are no longer necessary.
2. The nodes are present because syntax was introduced in the language specification to facilitate discussion under semantics or to facilitate some notions held by the designers of the language relating to how and when code should be generated. This usually yields relatively long parse tree

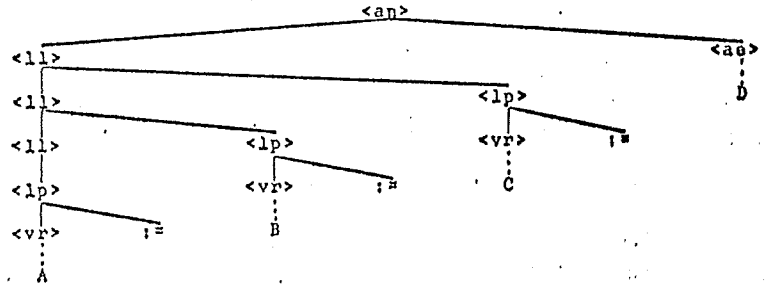
branches in which each node but the leaf has one son. These nodes are not meaningful in arriving at a computation tree.

3. The nodes are present because the syntax notation used, BNF in the case of Algol 60, is recursive. A recursive notation may lead to a left-branching (if left recursion is used) or right-branching (if right recursion is used) subtree. These nodes are not meaningful in arriving at a computation tree.

An example of the removal of superfluous syntax can be seen by considering the assignment statement

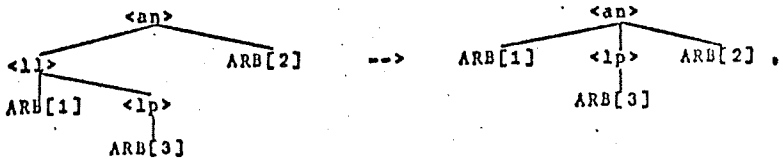
A:=B:=C:=D

which has the left-branching parse tree

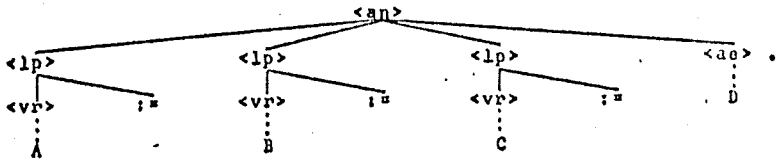


The left-branching structure does not contribute to the meaning of the construct. In particular, all the variables to the left of assignment operators are assigned the value of the right hand side. It would seem that all the left parts should be on the same level, that is, brother nodes.

Transformation 84,



when applied to the parse tree above and the resulting trees three times, removes this left-branching structure, yielding



Approximately forty-five per cent of the tree transformations for Algol 60 include removing superfluous syntax.

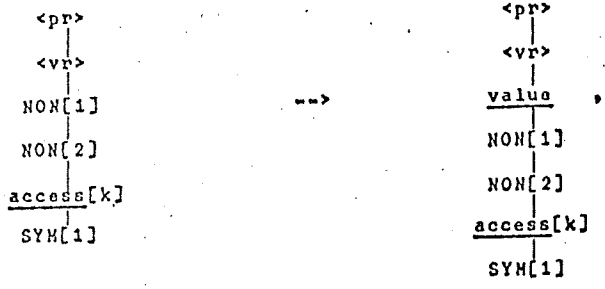
MAKING OPERATIONS EXPLICIT (EXPANSION)

The statements one writes in some high level programming language are usually a shorthand for a detailed sequence of operations. One of the tasks of the semantic specification is to put in those things which are implied but need to be made explicit at runtime. In the assignment statement

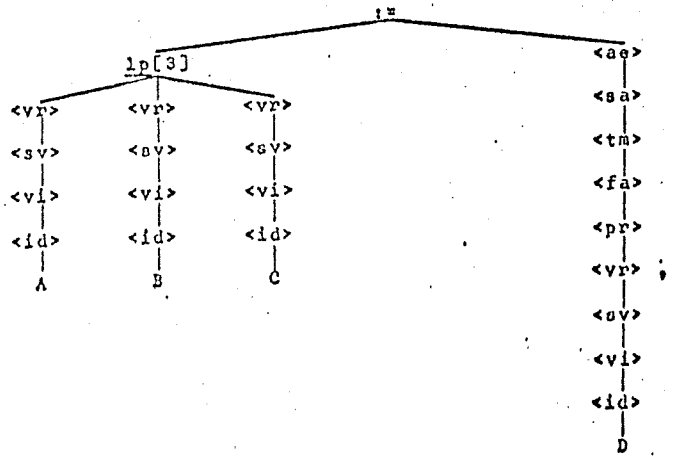
A:=B:=C:=D

the variable "D" is an expression; we have to determine the address of "D" and then the value stored at that address. For the variables "A," "B," and "C," however, all that has to be determined is the address of each. Transformations 87

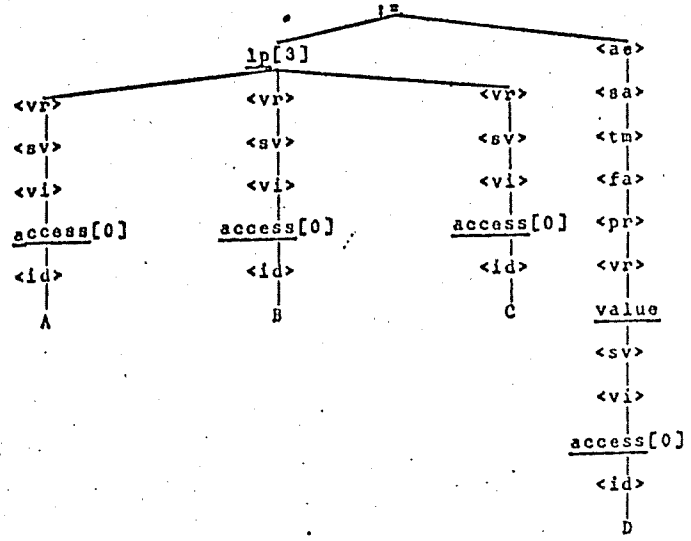
and 88



when applied to the tree



make these details explicit, yielding,



Approximately thirty-five per cent of the tree transformations for Algol 60 include making operations explicit.

COUNTING

There are several situations in which more than one instance of a construct occur and these instances are to be treated as a unit. For example, in the assignment statement

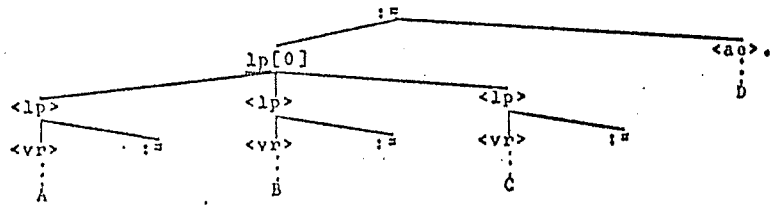
A:=B:=C:=D

the value "D" is to be assigned to the three left parts "A," "B," and "C." In doing the assignment one needs to know how many variables are to be assigned the given value.

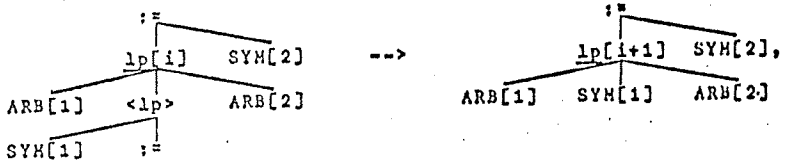
In order to group these instances and count them while

they are being grouped, two different transformations must be applied. The first of these inserts a collector node which has a subscript. The next transformation counts an instance, adds one to the subscript, and makes a change so that the instance cannot be counted again.

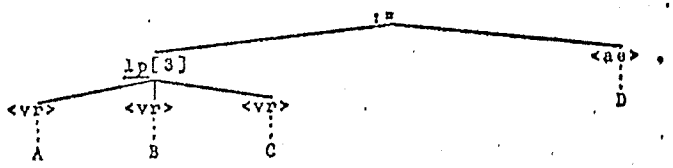
After several transformations, the tree for the assignment statement above is



The collector node lp[0] has already been inserted. Transformation 86,



when applied to this tree and then to the resulting trees as often as applicable, counts the number of left parts in the tree, yielding



Each application of transformation 86 adds one to the

subscript of the lp node and removes one <lp> node. The removal of the <lp> node prevents counting the corresponding left part again using transformation 86.

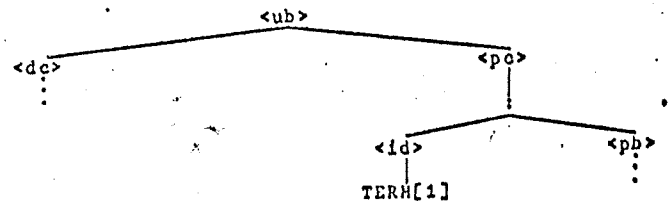
In this example there are superfluous <lp> node which can be removed. If there had not been superfluous <lp> nodes to remove, a marker would have been introduced and used to make changes. The transformation which introduced the node lp[0] would also have placed a marker as the leftmost son of the lp[0] node. Each application of the transformation to do the counting would have added one to the subscript and moved the marker over the son immediately to the right of the marker. Eventually the marker would be the rightmost son of the lp node. Then a third transformation would be used to remove this marker.

Approximately thirty per cent of the tree transformations for Algol 60 include a counting function.

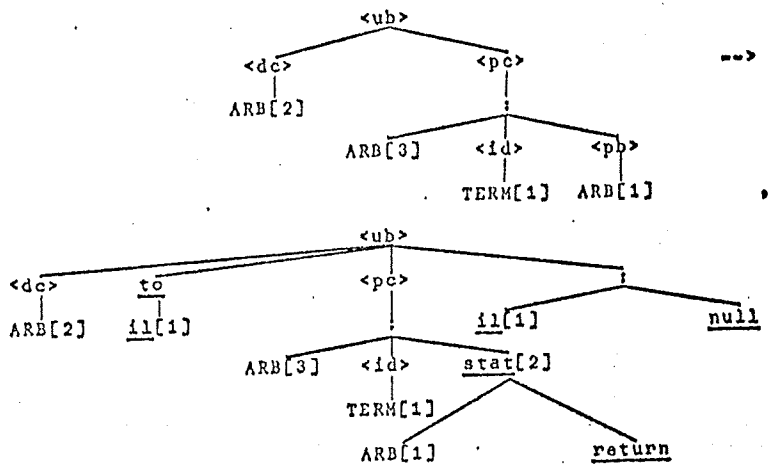
SPECIFYING ORDER OF EXECUTION

The order in which information appears in a source program is not always the order in which it should appear in the object program. For example, declarations must precede statements in an Algol 60 block. If one of these declarations is a procedure declaration, then some provision must be made to avoid "falling into" the procedure upon entering the block in which it is declared.

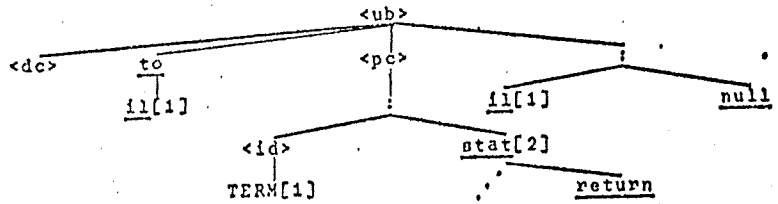
After several transformations a partial tree corresponding to a procedure declaration would be



Transformation 43

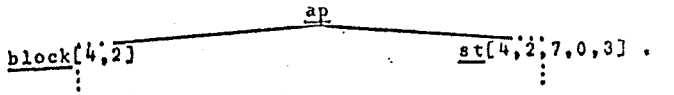


when applied to the tree above, inserts a transfer around the procedure body, yielding

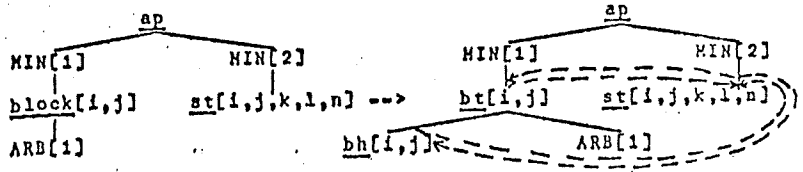


This transformation eliminates the possibility of

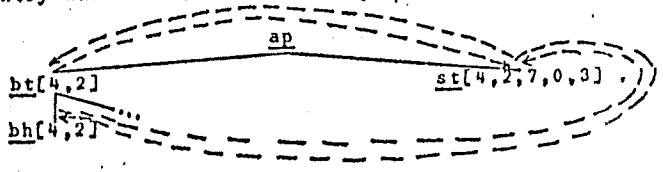
"falling into" a procedure declaration. The problem of referencing a procedure from various locations in a program is still present. In fact, there is still the general problem that information needed for continuing the execution of a computation is not always available at the next node. Another specific example of this occurs at block entry, when it is necessary to reference information pertaining to storage allocation for the block. To avoid repeating the information at each node where it might be referenced, linking is done. For example, part of a parse tree for a program after many transformations might be



Transformation 212



links block allocation information (right branch) and the block entry and exit (left branch) yielding

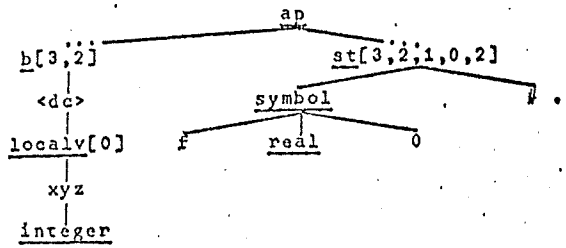


Approximately fifteen per cent of the tree transformations for Algol 60 include specifying order of execution.

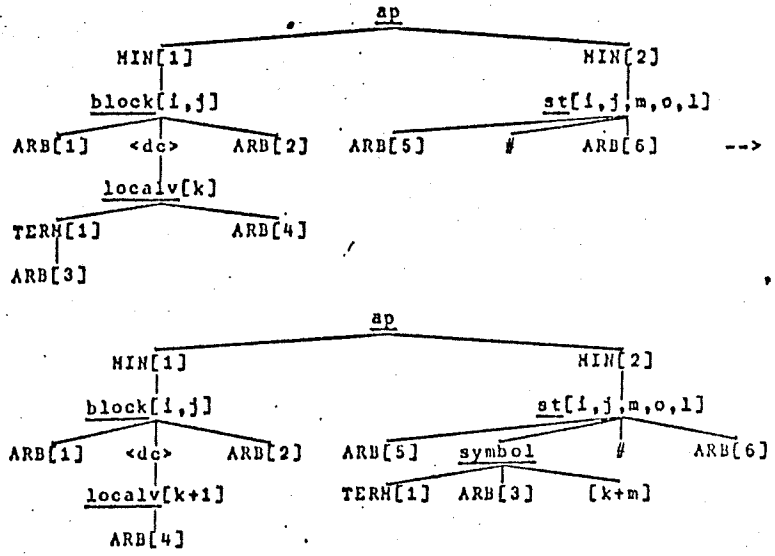
COPYING

It is sometimes necessary to copy information from one spot in a program to another, in some cases erasing the original copy, in some cases not. For example, the symbols defined in an Algol 60 declaration are entered in a symbol table; the actual declarations are discarded.

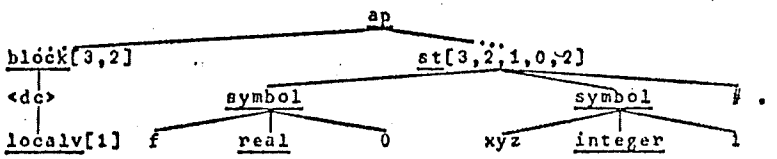
Part of a parse tree for a program after many transformations might be



Transformation 163



when applied to the tree above, copies the integer variable declared in the left branch into the symbol table of the right branch, yielding



Approximately twenty-five per cent of the tree transformations for Algol 60 include copying.

CREATING ANOTHER SEMANTIC SPECIFICATION

Producing a semantic specification for a programming language using this model is a matter of determining what

the computation trees for programs written in the language should look like. Many of the transformations in a semantic specification correspond to particular syntax equations. The reasoning required to produce these is similar to that required to do direct code generation for those equations. These local transformations eliminate syntax devices (like left and right recursion) which have no semantic content and produce operators and operands essential to express the meaning of the syntactic constructs.

The global transformations are required for those situations where the information provided by a single syntax equation is not sufficient to identify what computation tree segment should be produced. This is the same situation in which direct equation-by-equation code generation is inadequate. A few global transformations may also be used to replace a group of local transformations when it is convenient to do so. For example, rather than have one transformation for each syntax equation which creates superfluous parse tree nodes, it is more convenient to have a few global transformations handle most of the elimination of superfluous nodes.

Another consideration in producing a specification is the generality of the transformations. It is possible for a single generalized transformation to apply to several constructs. Alternatively, more specific transformations which each apply to an individual construct may be used.

The former, more general transformations yield a more compact specification, but the latter, more specific transformations often make the semantic analysis process operate faster. In particular, the tree matching algorithm in the analysis process may do a lot of needless backtracking if there are few specific nodes in the transformations. This is especially noticeable if the next node visited after an ARB, MIN, or BND node is a SYN, NON, TERM, or other nonspecific node.

Ordering the transformations must also be considered in producing a semantic specification. For most of the transformations in the semantic specification for Algol 60 given in Appendix I, ordering is not important. No matter in what order the transformations are applied, the results are the same. These transformations are assigned an order strictly for convenience.

Some of the transformations do require ordering. For a few, ordering is mandatory. For example, transformation 176 removes all remaining nonterminal symbols from the tree. This transformation must come after all other transformations which must include nonterminal symbols to identify the context in which they operate.

Another kind of ordering occurs because two or more transformations may apply to a particular construct and each changes the construct in some way. Because all but the first of these transformations apply to a changed subtree,

the transformations must by ordered so that each incorporates the changes made by the previous transformations. For example, transformation 212 replaces a node block[i,j] by two other nodes, one of which is bt[i,j]. Transformation 167 includes a node block[i,j]. If the order of these two were reversed, and 212 came first, then transformation 167 would include a node bt[i,j] rather than block[i,j].

EXAMPLE

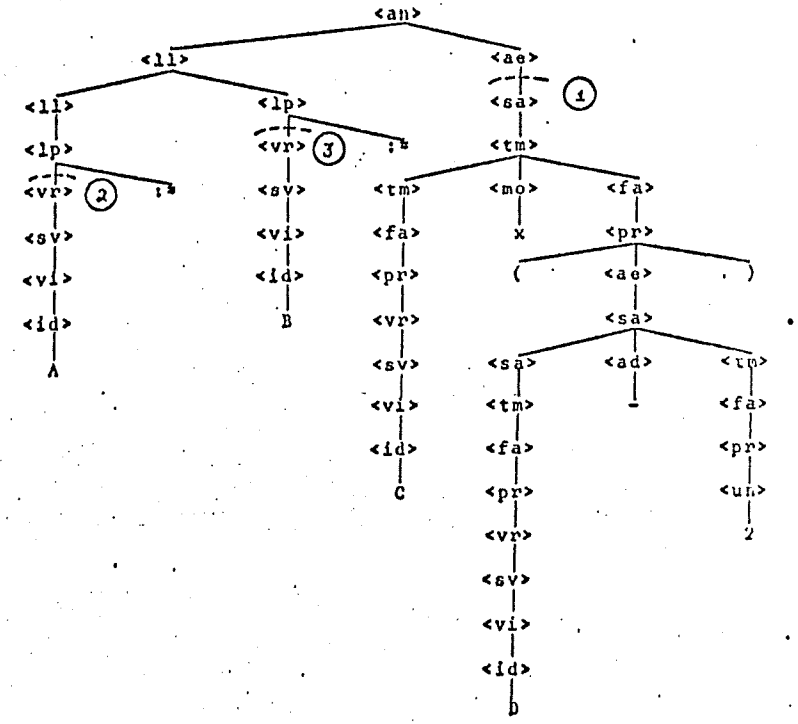
The elements of the semantic specification for Algol 60 are ordered so that subtrees of a parse tree may first be handled separately (perhaps in parallel). When all these local transformations are completed, the later, more global, elements of the specification are applied.

The example which follows illustrates the application of the semantic specification to a parse tree. To make it easier to see the effects of each transformation on the parse tree, certain subtrees are not explicitly given in each tree. Thus in one tree one or more subtrees may be shown with a dashed line separating the subtree from its root and with a number enclosed in a circle at the side. In the next tree, the encircled number appears in place of the subtree. Finally, in the tree to which a transformation affecting the subtree is to be applied, the complete subtree is reintroduced. In this example, the semantic specification is applied to the parse tree for the

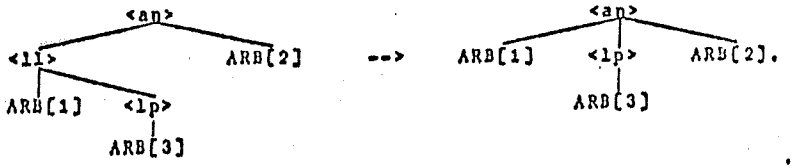
assignment statement

A:=B:=Cx(D-2).

The parse tree for this statement is

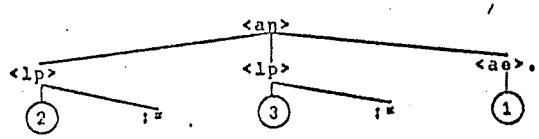


The first applicable transformation is number 84

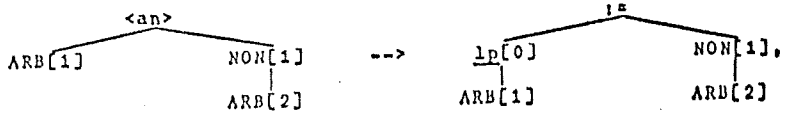


This transformation collects the left parts (the variables

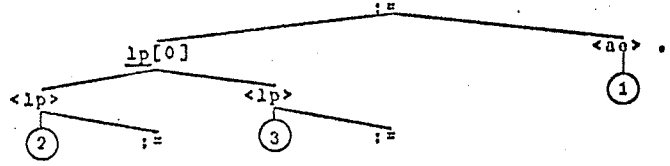
to which the value of the right part is assigned) at the same level in the tree. Two applications of this transformation yield the tree



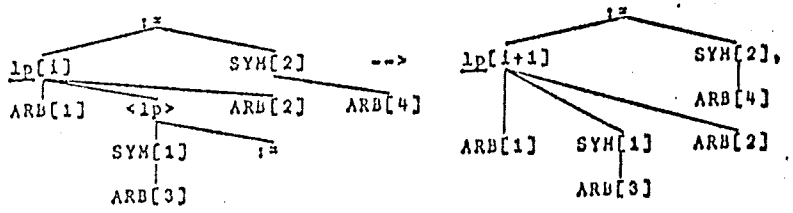
After these two applications, transformation 84 is no longer applicable. The next applicable transformation is 85,



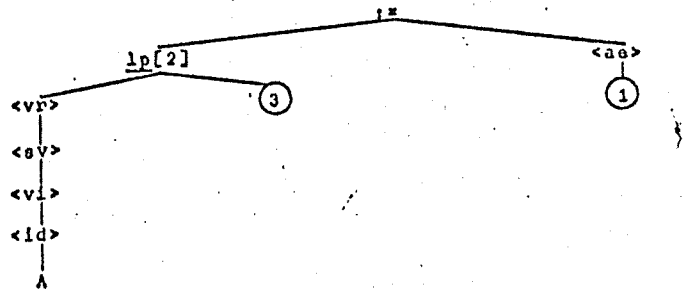
the main purpose of which is to insert into the tree a node used to count left parts. The application of this transformation yields the tree



Now the number of left parts in the assignment statement can be counted. The next applicable transformation, number 86

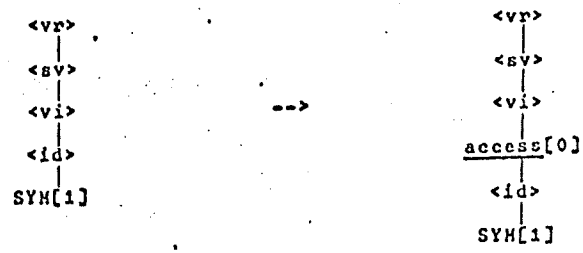


does this counting and also eliminates superfluous assignment operators, yielding

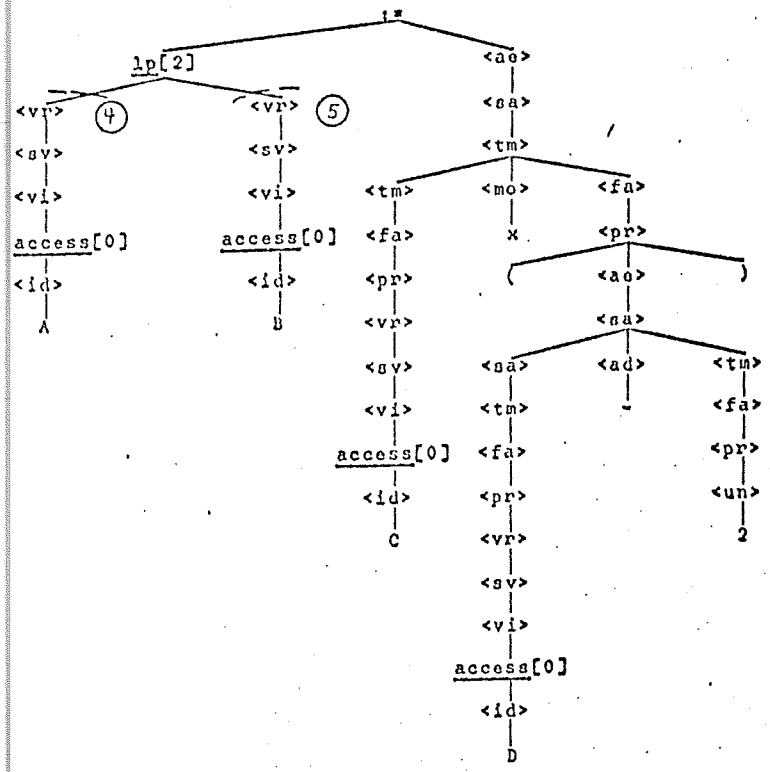


after two applications.

The next two transformations are concerned with inserting operators to fetch the address and contents of a location. Transformation 87

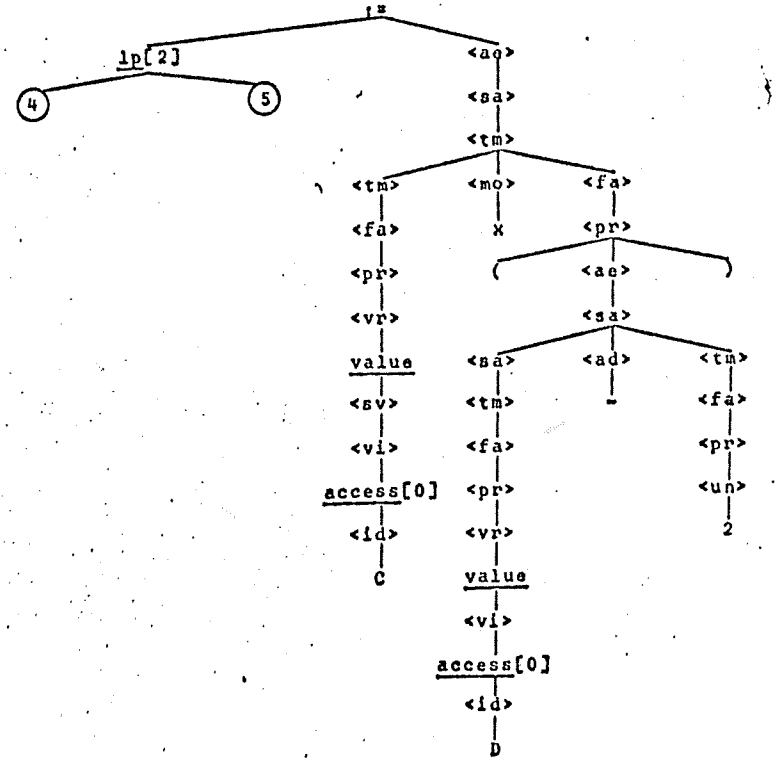


inserts the address fetch (access) operator before simple variables. This transformation applies at four different places in the tree, yielding



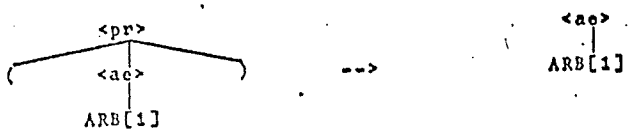
after four applications. Transformation 88

inserts the contents fetch (value) operator in arithmetic expressions. This transformation applies in two different places in the tree, yielding

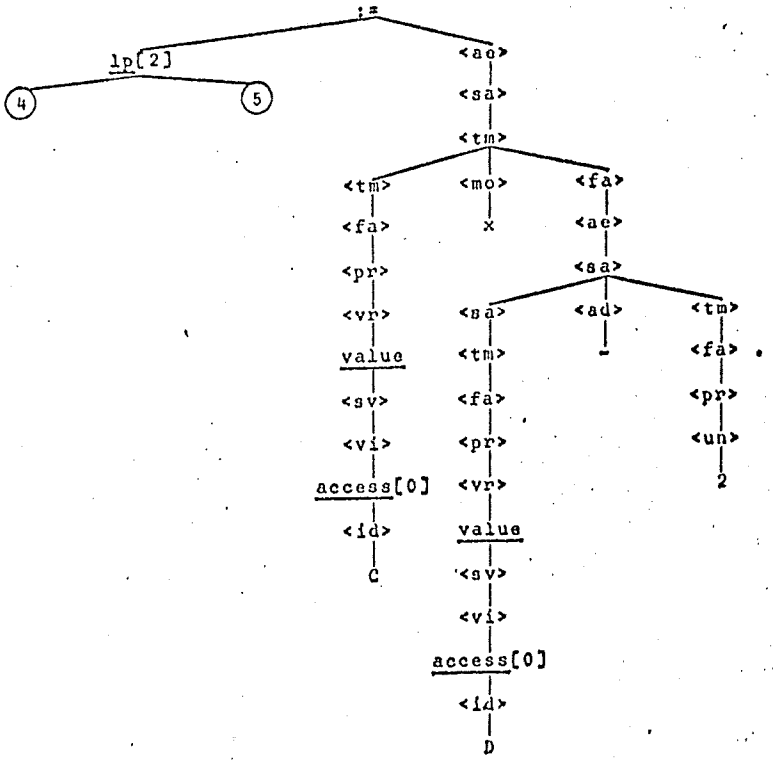


after two applications.

The next five transformations manipulate the form of arithmetic expressions. Transformation 90

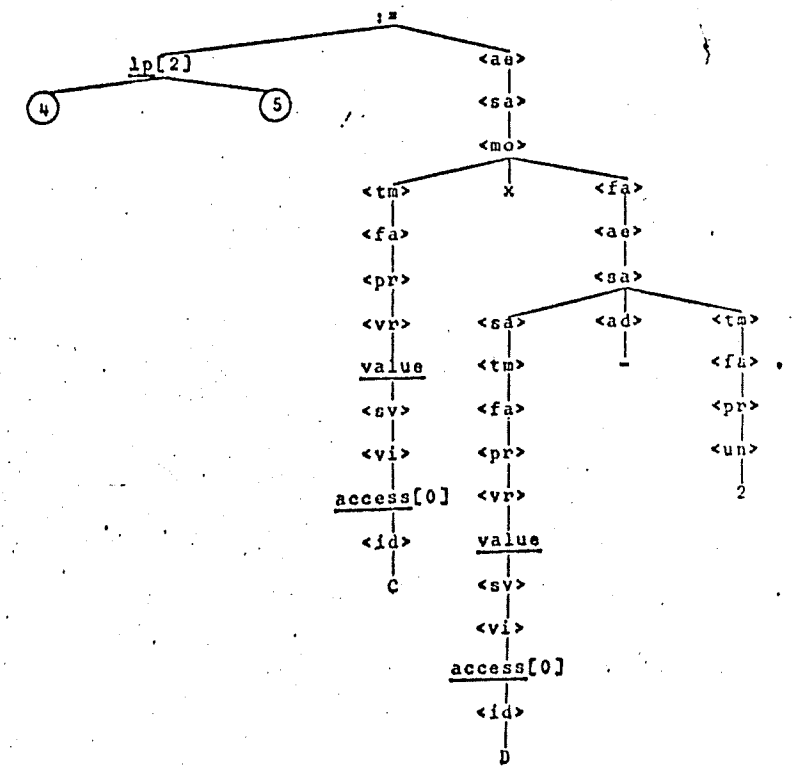


removes parentheses, yielding the tree



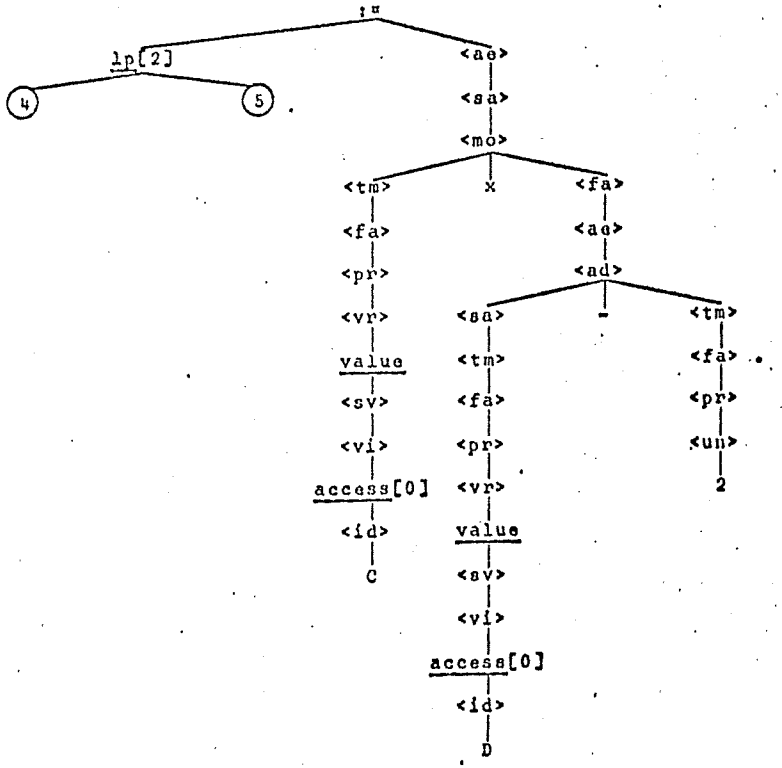
Transformation 117

moves the node which is the class operator for multiplication (i.e., <multiplying operator>) to a position so that it is a father node of its operands. Application of this transformation yields the tree



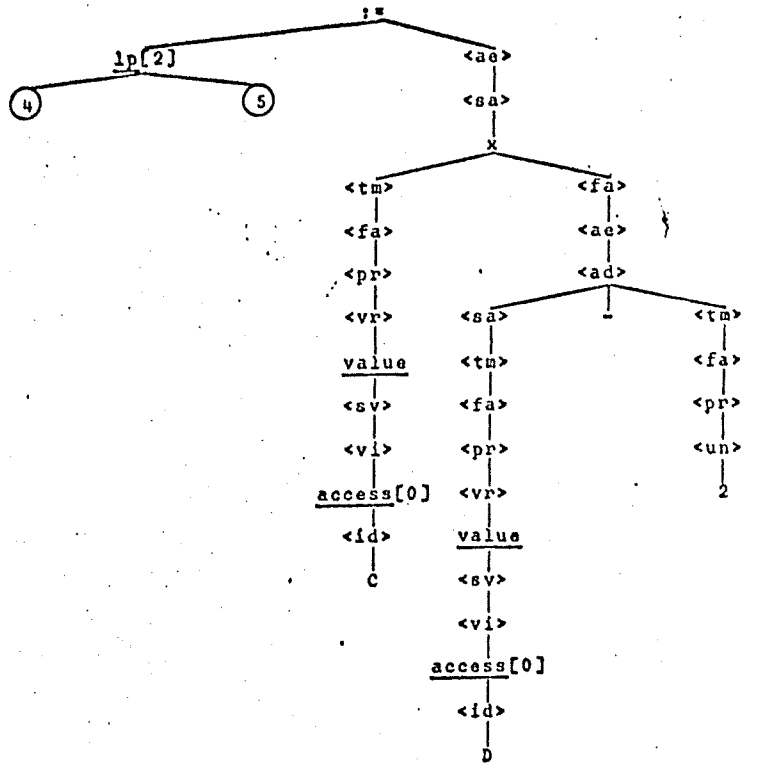
Transformation 118

is the same as the previous transformation except that it applies to <adding operator>. Application of this transformation yields the tree



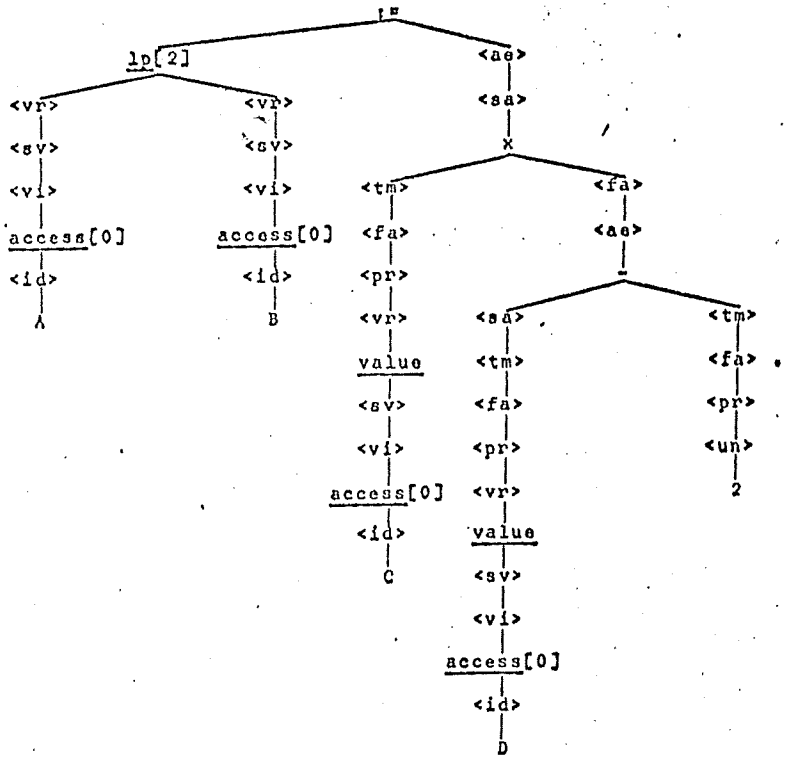
Transformation 124

replaces a class operator, in this case the multiplication class operator, with the actual operator, yielding the tree

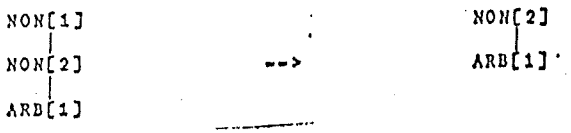


Transformation 125

is the same as the previous transformation except that it applies to the addition class operator, yielding the tree

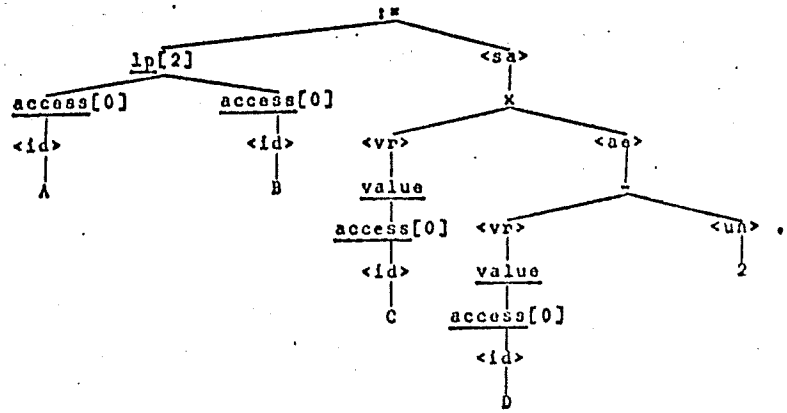


The last two transformations which apply eliminate superfluous nodes from the tree. Transformation 151



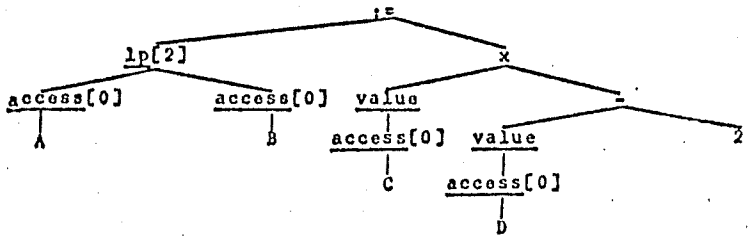
replaces two nonterminal nodes by a single nonterminal node. This transformation can be applied twenty-two times to the tree above, yielding, after the twenty-second application,

the tree



The last transformation which applies, number 176,

eliminates all remaining nonterminal nodes. After nine applications, the resulting computation tree is



If this assignment statement were included in a program, the resulting computation tree would not include the variable names "A," "B," "C," and "D." These names would have been replaced by references to the blocks in

which they were declared and to the displacements within the blocks. One or more of transformations 167-173 would have been used to do this replacement. Other transformations, including some from 1-30 and 130-166, would have been used to build the table of information used in the replacement. An example for a complete program is given in Appendix II.

Chapter 5

HISTORICAL PERSPECTIVES OF SEMANTIC FORMALISMS FOR PROGRAMMING LANGUAGES

Semantic formalisms for programming languages have their roots in three overlapping areas of concern. These are:

1. Language modeling--the desire to specify both natural and artificial languages completely and to examine their properties;
2. Automated translation--the desire to mechanize the translation process for programming languages; and
3. Proofs of correctness--the desire to examine notions of equivalence of programs culminating in the proofs of correctness of computer programs.

This chapter reviews some of the early developments in these three areas and describes some more recent work in formalisms for programming language description which is relevant to the work of this dissertation.

LANGUAGE MODELING

One of the earliest attempts to provide a complete specification of a programming language began in the late 1950's and culminated in the "Revised Report on the Algorithmic Language ALGOL 60" (Naur 1963). The

specification for Algol 60 included distinct syntax and semantic descriptions along with examples. It made use of the BNF formalism for syntax description. Semantics were given in English and the weakness of English as a semantic description language continued to be pointed out for several years in a number of articles, e.g., Knuth (1967), indicating gaps in the specification.

Another significant development was the introduction of the programming language Lisp (McCarthy 1962). While Lisp had an informal syntax specification (and a very simple syntax), it had a rigorous interpreter-defined semantic specification, the formalism for which was the lambda calculus. This has provided the inspiration for a major portion of the work done on semantic specification up to the present time.

The lambda calculus model for language specification was developed further by Landin (1965) and others (see, for example, Steel (1966)). Landin established a correspondence between both declarative and imperative constructs of Algol 60 and expressions of a structurally simpler language of applicative expressions. He also described an abstract machine for evaluating applicative expressions.

A third development was the natural language modeling done by Chomsky (1963). In addition to categorizing grammars based on the capability of the grammar to generate specific kinds of languages, he also introduced the notion

of transformational grammars. Because the standard grammars were not powerful enough to make a finite representation of all the structural variations of English feasible, transformations were introduced. These were used to take representations of English sentences and transform them into semantically equivalent but structurally different sentences. For example, the sentence "The girl threw the ball." (represented as a tree) might be changed by a passive voice transformation into the sentence "The ball was thrown by the girl." Although this work was done to help formalize the syntax of a natural language, the transformational model has since been used as a semantic formalism for programming languages.

AUTOMATED TRANSLATION

Efforts to automate the translation process for programming languages are surveyed in Feldman and Gries (1968). These efforts began with the development of techniques to construct syntactic analyzers automatically. In general, an automatically constructed syntactic analyzer, or recognizer, consists of a set of tables representing the syntax of the language whose statements are to be recognized and a set of routines to use these tables. There is also a construction algorithm for building these tables from some representation of the syntax as a grammar. (Alternatively, both the tables and the routines to use them may take the form of a program written in a specialized language.)

Automatic construction of recognizers has the advantages of saving time for human translator writers and guaranteeing that the recognizers follow the syntax of the language. Also, most constructing algorithms are capable of detecting ambiguity in grammars, and thus make it possible to eliminate potential problems which might not be detected until much later in the process of constructing a translator.

There are several potential disadvantages to these automatically constructed recognizers. They may not be very efficient. They may require substantial changes to the syntactic specification of the language. For a particular language, it may not be possible to modify the syntactic specification to fit the restrictions of certain recognizers. In such a case the language itself must be changed or the use of these recognizers is precluded. Finally, either the grammar or the recognizer may have to be altered if semantics are to be associated with the recognizer. Whether these potential disadvantages are serious actual disadvantages in any situation is dependent on the particular language being translated and the particular recognizer being used.

Once automatic construction of recognizers became possible, the next step in automating the production of translators was the development of techniques to handle the postsyntactic, or semantic, aspects of translation. These

efforts can be roughly divided into two categories: syntax-directed symbol processors and compiler-compilers.

The syntax-directed symbol processors provide a means for automatic construction of a recognizer and facilities for embedding calls on semantic routines within the recognition process. They also provide a collection of programs used during the execution of the output of these processes. Examples of this class of processors include the so-called metacompilers Meta-II and Tree-Meta.

The compiler-compilers provide some means for automatic construction of a recognizer and also attempt to automate the postsyntactic aspects of translation. A good representative of this class is the FSL compiler-compiler system developed by Feldman (1966) making use of FSL (Formal Semantic Language) to describe the semantics of languages to be translated.

The FSL system provides a syntactic metalanguage and semantic metalanguage. The formal specification of a language is written using these metalanguages. The syntactic and semantic specifications are taken as inputs by corresponding constructors which produce tabular representations of the syntax and semantics. Once these tables are produced, the constructors and the formal specifications can be discarded. These tables along with a collection of routines to use these tables form a compiler.

The syntactic metalanguage is a production language

which allows the embedding of calls to semantic routines. FSL, the semantic metalanguage in which these semantic routines are written, is an attempt to provide a machine-independent metalanguage incorporating facilities which are likely to be needed in writing a compiler. Machine-dependent aspects are handled by primitive routines. Statements in the semantic metalanguage are translated into machine language code. This code consists primarily of calls on the primitives routines. Thus, a compiler produced using this system consists of a syntax table, a semantics table (consisting primarily of calls on primitives), and a large collection of primitive, machine-dependent routines.

Other efforts, including meta-assemblers and extendible compilers, are relevant to a discussion of automated translation, but have not contributed to the model presented in this dissertation and are not discussed further.

PROOFS OF CORRECTNESS

Semantic models which have been developed for examining notions of equivalence of programs are primarily axiomatic, rather than constructive, in nature. These models deal with abstractions of concepts of programs, e.g., assignments, conditional constructs, and sequencing, rather than with complete programming languages and as such are not directly applicable to the approach taken in this dissertation.

De Bakker (1969) surveys several axiomatic approaches to semantics. These models consider a set of variables, a

set of unspecified functions depending on these variables, and a formalization of a property of programs, e.g., assignment, to be studied. Each provides a set of axioms and inference rules which then can be used to examine the equivalence in terms of effect of different sequences of the property, e.g., sequences of assignment statements. Other approaches make use of flow diagrams (directed graphs) to investigate sequencing, but are not essentially different from the axiomatic methods.

An example of the application of these techniques to complete programming languages is found in Floyd (1967). He uses an informal, as opposed to mechanical, approach to the problem of program verification. He considers a program represented as a directed graph, or flowchart, and associates a transformation with each node and a proposition with each edge. Each transformation is a computation step involving the variables of the program. Each proposition specifies the conditions to be satisfied by the variables of the program in order for program control to pass along the associated edge.

Program verification is considered in terms of two problems. First is an attempt to prove that the program yields the correct result, second is an attempt to prove that the program terminates. The first problem is treated as a sequence of subproblems, that is, starting with an initial proposition, the attempt to prove correctness is a

sequence of attempts to prove that each subsequent proposition is implied by the previous proposition and transformation. The final proposition should describe the intent of the program and if this proposition is implied by the sequence of steps, the program can be assumed to yield the correct result if the program terminates. Proving that the program terminates involves showing that there is an interpretation of the program variables under which the final proposition is satisfied.

The effectiveness of this technique depends on the fallibility of the person supplying the propositions and carrying out the proof. The use of mechanical verification techniques may eliminate human error, but contemporary mechanical theorem provers are not yet sophisticated enough to carry out, in a reasonable amount of time and space, the subtle manipulations often required in the program verification process (Elspas, Levitt, Waldinger, and Waksman 1972).

FORMAL SPECIFICATION OF PROGRAMMING LANGUAGES

The remainder of this chapter describes two models which are directly relevant to the work presented in this dissertation. These are the Vienna Definitional Language developed at the IBM Vienna Laboratory to define PL/I (Lucas, Lauer, and Stigleitner 1968) and the augmented grammar model of Cheatham (1968).

VIENNA DEFINITIONAL LANGUAGE

The description of the Vienna Definitional Language (VDL) presented here is due to Neuhold (1971). It is a simplification of the actual model and does not exhibit the generality required to handle all of PL/I, for example. Nevertheless, the description does indicate the significant concepts of the original.

The Vienna Definitional Language is a formalism for describing both the syntax and semantics of programming languages in a machine-independent and implementation-independent fashion. The premise of this method is that the interpretive execution of a program constitutes a semantic description of that program. The essence of the semantic specification method is tied to the definition of the interpreter, that is, the definition of its instructions and functions and of the information it retains as it executes a program.

In order to make the interpretation process as clear and simple as possible and to maintain machine independence, the interpreter and its inputs are abstractions rather than concrete realizations. The interpreter is an abstract or conceptual machine. Its mechanisms are the same for all language definitions, although the information it must retain as it interprets programs, i.e., the state of the interpreter, varies with the language. To simplify the task of the interpreter, considerations of form which are not

relevant to the substance (meaning) of a program are eliminated from the program before interpretation. Thus the abstract interpreter operates on an abstract program.

An abstract program is represented as a tree. Each branch has a name associated with it which identifies the abstract program segment it represents. These names are used as selectors and, when applied to a tree, produce the subtree descendant from that branch. Further, there are a set of test functions, or predicates, to test if a tree segment is of a certain class. Elementary predicates are defined for describing elementary tree segments (i.e., the leaves or terminal nodes of trees) and additional predicates can be defined as combinations of the elementary predicates. The abstract syntax for a language is written in terms of these selectors and predicates.

The abstract interpreter operates on a tree structure and performs transformations upon that structure. The behavior of the interpreter constitutes an interpretation and hence a semantic specification of the abstract program contained in the tree structure.

The tree structure on which the interpreter operates contains all the information which determines the state of the interpreter. In addition to the abstract program, the tree contains a statement counter, storage for values of program variables, a control store, and a library. The library contains definitions of all the interpreter

instructions. These definitions are particular to the language being defined. The control store is dynamic and contains those interpreter instructions which are to be executed.

The mechanisms of the interpreter can be better understood by considering how the interpretation progresses from an initial state. Initially, the state consists of the abstract program to be interpreted, an initialized statement counter, an instruction to interpret a program in the control store, an empty value store, and a library containing all the interpreter instructions. The execution control, i.e., the interpreter, selects a leaf of the control tree. This specifies the instruction to be found in the library and then executed. The execution of the instruction causes the abstract program tree to be examined, the statement counter to be updated, and the control store to be modified. This control store modification may include deletion of the selected leaf, insertion of the result returned by the execution as an argument in the instruction where it is to be used, and the insertion of additional instructions. The value store may also be modified. The execution control then selects another leaf of the control tree and the interpretation process continues until the control store is empty.

The formal definition of a language using the Vienna model has several parts. A formal syntax specification in

BNF, for example, and a specification of the mapping of source programs into abstract programs are required. The semantic component includes the instruction definitions in the library, which are particular to the language, and the abstract interpreter mechanisms. These mechanisms include the sequencing concepts of the interpreter, elementary objects and functions, means for composition of objects and functions, and operators for tree modification.

The formal definition of a language using the transformational model presented in this dissertation has many of the same parts required using the Vienna model. The transformational model requires a formal syntax specification. The model deals with concrete (as opposed to abstract) programs, and thus there is nothing which corresponds directly to the mapping of source programs into abstract programs. The transformational model does include some preliminary manipulations, however, since it presumes an adequate lexical analyzer to take character strings and produce symbol tokens.

The significant differences between the models are in the semantic component. The Vienna model assigns meaning to a program solely on the basis of the interpretive execution of an abstraction of that program. The transformational model assigns meaning to a program based on the results of an analysis process applied to that program and the results of the interpretation of the output of that analysis

process. The library of the Vienna model and the set of tree transformations of the transformational model play similar roles. In the latter model, however, the information contained in the transformations is introduced into the tree to be interpreted by a separate semantic analysis process. The mechanisms of the abstract interpreter of the Vienna model have analogs in the mechanisms found in the analysis process and interpreter of the transformational model. For example, the semantic analysis process mechanisms include tree modification operations, the interpreter mechanisms include elementary operations, and both include sequencing concepts.

THE CHEATHAM MODEL

The work of Cheatham indicates that the notion of the semantics, or meaning, of a program has two aspects. The first aspect assigns meaning to a program based on a structural representation resulting from an analysis process. This process takes a program whose language is described by context free grammar rules and associated interpretations and produces data structures representing the meaning of the program. These structures include a computation tree representing the imperatives of the program and data graphs representing the declaratives of the program. This is a representation of meaning to the extent that the components of the resulting structures are understood.

The second aspect of semantics assigns meaning to a program based on results produced when the data structures yielded by the analysis process are interpreted. In this context the interpreter is an abstract or conceptual machine for which the data structures are a machine language program.

Because the work of this dissertation has much in common with the Cheatham model, the details of both approaches are somewhat similar. The significant differences are enumerated below.

1. The Cheatham model associates an interpretation with each syntax equation. His interpretations are tree transformations which are highly dependent on previous and subsequent transformations.

   The transformational model associates a body of transformations (very similar to Cheatham's interpretations) with the entire syntax. Semantics that are not generally tied to a particular syntax equation are readily expressible. For example, this approach makes it easier to express such things as number and type checking of arguments in a procedure call and data type conversions. When Cheatham considers certain issues like context sensitive interpretation of references and argument checking, he is forced to expand the syntax so that there are equations with which to associate the

interpretations.

2. In associating interpretations with syntax equation recognition, the Cheatham model limits the languages which can be modeled to those which can be parsed without backtracking. Without this restriction, it would be necessary to examine the problem of undoing transformations when backtracking occurs.

   This question does not arise in the transformation model, since the program is completely parsed (and complete parse tree built) before the transformations are applied. At any point during the parse and transformation phases, however, the tree is likely to contain more, perhaps many more nodes than in the Cheatham model, since his approach eliminates nodes which are no longer semantically useful as soon as they are discovered.

3. In the Cheatham model, interpretations are implicitly ordered by the syntactic hierarchy. That is, when a syntax equation is recognized, the corresponding interpretation is carried out.

   In the transformational model, transformations are explicitly numerically ordered, but the ordering is probably quite similar to Cheatham's since the first transformations consider the

grosser syntactic entities in the parse tree. Using the transformational model, however, one can place a few housekeeping transformations at the end to eliminate superfluous nodes, rather than consider their elimination at each level in the tree.

4. Cheatham uses an ideal language to illustrate his model and points out that his model serves mainly to point out the desirability of a symmetric, consistent language like the one he has used and the usefulness of Ambit/G (a two-dimensional graphic language) as a semantic host language.

   The transformational model is illustrated using Algol 60. This language is not as suitable for the model as Cheatham's and hence indicates the generality of the technique. The semantic host language is two-dimensional and uses several tree pattern matching mechanisms. In addition, it introduces operators into the computation tree. These operations are, by and large, commonly accepted as the kinds of basic operations computers can perform. Of primary concern is that both the syntactic and semantic specifications should be capable of driving a translator.

5. The interpretations for some declarations provided in the Cheatham model are conditional programs,

i.e., if the first interpretation described cannot be performed, a second interpretation is attempted. This introduces another mechanism for sequencing through the interpretations.

   The sequencing mechanism for the transformational model is to start with the first transformation, apply it repeatedly until it no longer applies, and then go to the next transformation.

6. The symbol table in the Cheatham model is a collection of data graphs which are highly dependent on Ambit/G. The attributes of symbols are indicated by the shapes of boxes in which the associated values are stored and by the orientation of links between boxes.

   The transformational model includes symbol information in the computation tree. The organization used is somewhat similar to the property list used in Lisp. Each symbol has various attribute-value pairs associated with it. In addition there are symbol tables for each block (denoted by a symbol table number and level) and each table contains a reference to the previously regnant table.

7. The Cheatham model introduces macros to handle conventional variable declaration forms (integers,

arrays, and so forth) and thus translates them into the language being defined.

The transformational model treats the handling of declarations as just another set of transformations which introduces the appropriate information into the symbol table and computation tree.

8. Because the Cheatham model associates interpretations, which are at least in part the semantic specification of a language, with syntax equations, the notion of a semantic analyzer is not distinguished from a syntactic analyzer. Further, the domain on which semantic analysis operates is determined by syntactic analysis. The kinds of operations which are performed in producing the computation tree and data graphs of the Cheatham model are embedded in the operations of the host semantic language.

In the tranformational model, the semantic analyzer is a separate process, distinct from the syntactic analyzer. The kinds of operations the semantic analyzer performs, namely structural pattern matching and replacement, are made explicit in the description of the process.

9. That aspect of semantics which assigns meaning to a program based on results produced by the

interpretation of the computation tree and associated data graphs is handled similarly in both models. In the Cheatham model, the interpreter is defined, primarily, by an Ambit/G program. Explanations of interpreter functions and Ambit/G mechanisms are provided in varying degrees of formality and precision.

The transformational model also defines an interpreter. The basic control and data structures and operations of the interpreter are described in detail. This description is independent of any specific implementation. Thus the description does not specify precisely how the various structures are to be implemented, but rather concentrates on detailed specification of what manipulations are to be performed.

In summary, the principle difference between the transformational model and the other semantic models is that the transformational model defines a distinct semantic analysis process which can be used in conjunction with a formal semantic specification to produce a structural representation of the meaning of a program. While the Cheatham model and the transformational model both use an analysis process, the latter provides a definition of a separate semantic analysis process and shows its application to a widely used programming language.

Chapter 6

CONCLUSION

In the preceding chapters several aspects of the semantics of programming languages have been presented. This chapter summarizes the results of the earlier chapters, indicates several uses for this work, and points out some possible extensions and future research directions.

In addition to information of historical interest, the preceding chapters deal with two main topics. The first of these is a formalism for the specification of the semantic components of programming languages. A semantic specification for a programming language can be viewed as a set of state transformations. Each state transformation is a pair of trees indicating an initial state and a final state into which the initial state is to be transformed. The first tree of each pair describes a particular structure and the second tree describes a modification to that structure to make explicit its meaning. Appendix I shows the use of this formalism to give a semantic specification for Algol 60.

The second main topic presented is a semantic process which makes use of a formal specification of semantics to perform part of the translation process for programming

languages. The process has two inputs: a structural representation of the program being translated (e.g., a parse tree) and the set of pairs of trees which constitute a semantic specification for the programming language in which the program is written. As output the process produces a computation tree, which is a representation of the meaning of the program being translated. Appendix II illustrates the application of the semantic process to an Algol 60 program.

The primary purpose of this work is to model programming language semantics and the semantic analysis process. The model helps to solve some problems present in and arising from less formal programming language specifications. The use of the semantic description formalism and the translator model can eliminate the ambiguities in programming language definitions and help to eliminate machine-independent incompatibilities among different implementations of the same language.

While the primary purpose of this work is to model semantics, an implementation of the model for programming language translation described in Chapter 2 has some practical applications. In order to do the implementation, several pieces of software have to be written. First, a lexical analyzer and syntactic analyzer which produces a parse tree as output are required. These may be time-consuming, although not technically difficult to

produce. Second, a semantic analyzer to operate on the parse tree is required. Producing one is somewhat more difficult than producing a syntactic analyzer, primarily because of the complexity of the matching procedures required to handle the special nodes. The matching procedures do not always match one node to another, but rather may have to match a special node to sections of a tree. An implementation of a semantic analyzer, coded in Lisp, is given in Appendix III. Finally, a code generator or interpreter to operate on the output of the semantic analyzer, i.e., the computation tree, is required.

Up to the code generation or interpretation stage, translation is primarily a machine-independent symbol manipulation problem. A code generator, however, is a machine-dependent translator component in that the object code it emits is specific to some target machine. An interpreter in this context is really an implementation of the conceptual tree machine; computation trees are programs for the tree machine. Although an interpreter written in a high level programming language may be a machine-independent program, that is, independent of the physical machine on which it executes, it is dependent on the conceptual machine defined by the high level language. Furthermore, the interpreter embodies the translation of tree machine code to a form executable on a particular machine.

A variation of the translation model mentioned in

Chapter 2 involves alternation between syntactic analysis and semantic analysis. This could readily be accomplished by keying transformations to syntax equations. The recognition of a syntax equation would cause the corresponding transformation to be carried out. This is a procedure commonly followed in syntax-directed translation.

The approach which is likely to be most efficient in terms of both time and space is to alternate between syntactic analysis and semantic analysis for many of the transformations and apply the remainder of the transformations after the interleaved syntactic/semantic analysis is completed. Some of the transformations relate directly to syntactic constructs and applying them at the time and place the syntactic structure is recognized can eliminate much futile tree searching, thus saving time, and many unnecessary nodes, thus saving space. Other transformations are really of a global nature, for example, those whose context includes more than a single statement. These are more readily applied after the completion of syntactic analysis.

As was mentioned in Chapter 2, generalized table driven translation may not compare favorably in terms of efficiency with more ad hoc means of translation. Improvements can be made by selecting a syntactic analysis method no more powerful than necessary. Some generality could also be removed from semantic analysis, e.g., eliminating

backtracking in tree matching, if it could be shown that this generality is not needed for a particular semantic specification. Nonetheless, such tailoring may still not make the speed of table driven translation comparable to that of some other means of translation.

A generalized table driven translator as described in Chapter 2 does have several uses. An implementation of one is an ideal tool for language (as opposed to program) debugging. One could readily manipulate syntax and semantic specifications and observe the effect of such manipulations on language constructs as they are passed through the translator. The availability of a tool to debug and tune a programming language definition should simplify the design process. The availability of an implementation, albeit a relatively inefficient one, as soon as the formal specifications are completed makes it possible to consider carefully the strategies to be employed in a production version before a sizeable implementation effort is undertaken.

There are some additional consequences of the semantic formalism and semantic process which relate to parallel processing and program transferability. Many of the transformations in the semantic specification for Algol 60 are order-independent. If a parse tree can be separated appropriately into several subtrees, several separate semantic analyzers (one for each subtree) could operate in

parallel to apply transformations. (Alternatively, each semantic analyzer could, in turn, be used to apply certain transformations to each subtree.) After the transformations dealing with local context are applied, the resulting subtrees can be recombined and the remaining global transformations can be applied.

Program transferability arises as a consequence of programming language portability. A language which has a complete lexical, syntactic, and semantic specification using the formalisms described earlier can be immediately transferred to any system which has a skeleton translator (lexical, syntactic, and semantic analyzers and code generator) available.

Chapter 3 includes several tables of tree machine operators and Appendix I includes an Algol 60 semantic specification which incorporates those operators. If semantic specifications for some other programming languages are written, some new operators may be required. For a language with a larger selection of data types and allocation possibilities, e.g., PL/I, several additional access operators as well as explicit allocation and deallocation operators are required. A semantic specification for a character string processing language like Snobol might use substring extraction and concatenation operators. For a language like Lisp, where a variable is defined by its execution context rather than its definition

context, different access mechanisms as well as different allocation-deallocation mechanisms are required. In general, the additional operators required to give a semantic specifications for other languages fall into two categories: storage management and data manipulation. Additional storage management operators are necessary to handle variable binding times and consequent allocation strategies different from those of Algol 60. Additional data manipulation operators are necessary to handle a richer variety of data types, e.g., character strings and Cobol-like structures, than are found in Algol 60.

In addition to the use of transformations illustrated earlier, there are at least two other potential uses for transformations as part of the semantic aspect of translation. These uses are error detection and optimization.

The use of transformations to accomplish error detection has two aspects: detection and correction of what are often classified as compile time errors and manipulation of computations to facilitate detection and correction of what are often classified as run time errors. Transformations to detect compile time errors are really checks for consistency. Depending on the programming language under consideration, such transformations might include checks to determine if all variables and labels referenced are also defined, if the arguments of a procedure

call correspond in number and type to the parameters of the procedure, and if variables in an expression or assignment statement are of compatible types. If errors are found, the transformation could modify the computation tree to indicate an error and, depending on the kind of error, attempt to make a correction. For example, if a variable is referenced but not defined, a transformation could indicate the error (via an error return in the semantic analyzer or via a node inserted in the tree) and insert a declaration of the variable in the local symbol table.

Transformations to facilitate detection and subsequent programmer correction of run time errors involve the insertion of additional nodes in the computation tree to monitor computations. Such insertions might include PL/I-like condition handling features, for example monitoring array subscript usage to insure that the declared range is not exceeded and handling overflow conditions.

Transformations to accomplish computation tree optimization are of the global type, that is, they are applied after the completion of syntactic analysis. Such transformations rearrange a computation tree into another tree which is computationally equivalent but, by some measure, more efficient. That is, the rearrangement should leave the results of executing the tree unchanged, but should cause some improvements in the time or space needed to compute the results. Optimization really involves

changing the meaning of a program as represented in a computation tree and, as in all optimization, there is the danger that the rearrangement may affect the computation in some unanticipated way.

Another potential extension to the work presented here deals with data handling. The transformations for Algol 60 given in Appendix I build a block structured symbol table containing relevant attribute information. For programming languages in which data types are determined at compile time, additional transformations might be included to indicate the appropriate operator and required type conversion functions necessary to evaluate a particular expression. For example, the addition of an Algol 60 integer variable and an Algol 60 real variable might include the conversion of the integer to real and the use of a real (floating point) addition operator. Other transformations might concern access functions and additional aspects of data modeling.

What seems to be the most interesting extension to the semantic modeling presented here is the development of metrics for programming language semantics. Just as with syntax, there can be many sets of semantics for a language. It would be theoretically interesting and potentially of practical value to have some measure of how good a particular semantic description of a language is for some purpose. Work with formal languages has resulted in some

classifications of grammars according to their generality (Hopcroft and Ullman (1969) survey these results.). Further efforts, e.g., De Remer (1971) and Earley (1970), have established some theoretical bounds on the time and space required for syntactic analysis using different levels of generality in the syntax description. Analogous results would be desirable for semantics.

There are several items one could examine in attempting to determine suitable metrics. The number of transformations included in a semantic description of a programming language might give some measure of the power of the language being described, but the number alone does not give a good indication of the complexity of the transformations. Classification of transformations using some complexity measure might be analogous to classification of formal language grammars. A starting point for developing such a classification for semantics using this model would include a thorough examination of the functions embodied in the transformations. These functions could then be classified according to the context they require, the complexity of the operations they perform, the number of special matching nodes they use, and so forth. A transformation would then derive its complexity measure from the measures of its component functions plus any factors about the transformation which might prove relevant. A semantic specification then could be classified by the

complexity of its most complex transformation. This could lead to considerations of equivalence of transformations in an attempt to show that one transformation is equivalent to another embodying functions of lesser complexity.

Another measure might consider the time or space required for semantic analysis for different semantic descriptions. As with syntactic analysis, bounds might be obtained on the time and space requirements necessary to do semantic analysis using transformations of particular complexities.

All of these considerations of metrics are, of course, quite sketchy. More work needs to be done in refining the semantic model before much progress can be made in measurement. However, it seems to be an interesting and potentially very fruitful area for further research in programming languages.

BIBLIOGRAPHY

Berkling, Klaus J. "A Computing Machine Based on Tree Structures." IEEE Transactions on Computers, Vol. C-20, No. 4, April 1971, pp. 404-418.

Boyle, J. M. and A. A. Grau. "An Algorithmic Semantics for ALGOL 60 Identifier Denotation." Journal of the Association for Computing Machinery, Vol. 17, No. 2, April 1970, pp. 361-382.

Cheatham, T. E., Jr. "A Semi-Formal Model for the Syntactic and Semantic Specification of Programming Languages." Massachusetts Computer Associates, Wakefield, Massachusetts, 1968.

Cheatham, T. E., Jr. and K. Sattley. "Syntax Directed Compiling." Programming Systems and Languages. Ed. Saul Rosen. New York: McGraw-Hill, 1967, pp. 264-297.

Chomsky, Noam. Aspects of the Theory of Syntax. Cambridge: M. I. T. Press, 1965.

Chomsky, Noam. Syntactic Structures. The Hague: Mouton, 1957.

Chomsky, Noam and M. P. Schutzenberger. "Introduction to the Formal Analysis of Natural Languages." Handbook of Mathematical Psychology, Vol. 2. Eds. R. R. Bush, E. H. Galanter, and R. D. Luce. New York: John Wiley, 1963, pp. 269-322.

De Bakker, J. W. "Semantics of Programming Languages." Advances in Information Systems Science. Ed. J. T. Tou. New York: Plenum Press, 1969, pp. 173-227.

DeRemer, Franklin L. "Simple LR(k) Grammars." Communications of the Association for Computing Machinery, Vol. 14, No. 7, July 1971, pp. 453-460.

Duncan, F. G. "Our Ultimate Metalanguage." Proceedings of the IFIP Working Conference on Formal Languages for Computer Programming. Ed. T. B. Steel. Amsterdam: North-Holland, 1966, pp. 295-299.

Earley, J. "An Efficient Context-Free Parsing Algorithm."

Elson, M. and S. T. Rake. "Code-Generation Technique for Large-Language Compilers." IBM Systems Journal, Vol. 9, No. 3, 1970, pp. 166-188.

Elspas, Bernard, Karl N. Levitt, Richard J. Waldinger, and Abraham Waksman. "An Assessment of Techniques for Proving Program Correctness." Computing Surveys, Vol. 4, No. 2, June 1972, pp. 97-147.

Feldman, J. A. "A Formal Semantics for Computer Languages and its Application in a Compiler-Compiler." Communications of the Association for Computing Machinery, Vol. 9, No. 1, January 1966, pp. 3-9.

Feldman, J. A. and D. Gries. "Translator Writing Systems." Communications of the Association for Computing Machinery, Vol. 11, No. 1, January 1968, pp. 77-147.

Floyd R. W. "Assigning Meanings to Programs." Mathematical Aspects of Computer Science, Vol. 19. Ed. J. T. Schwartz. Providence: American Mathematical Society, 1967, pp. 19-32.

Garwick, J. V. "The Definition of Programming Languages by their Compilers." Proceedings of the IFIP Working Conference on Formal Languages for Computer Programming. Ed. T. B. Steel. Amsterdam: North-Holland, 1966, pp. 139-147.

Gries, David. Compiler Construction for Digital Computers. New York: John Wiley, 1971.

Hopcroft, John E. and Jeffrey D. Ullman. Formal Languages and their Relation to Automata. Reading, Massachusetts: Addison-Wesley, 1969.

Ingerman, P. Z. "Thunks--A Way of Compiling Procedure Statements with Some Comments on Procedure Declarations." Communications of the Association for Computing Machinery, Vol. 4, No. 1, January 1961, pp. 59-60.

Irons, Edgar T. "A Syntax Directed Compiler for Algol 60." Communications of the Association for Computing Machinery, Vol. 4, No. 1, January 1961, pp. 51-55.

Johnson W. L., J. H. Porter, S. I. Ackley and D. T. Ross. "Automatic Generation of Efficient Lexical Processors Using Finite State Techniques." Communications of the Association for Computing Machinery, Vol. 11, No. 12, December 1968, pp. 805-813.

Knuth, Donald E. The Art of Computer Programming, Vol. 1. Reading, Massachusetts: Addison-Wesley, 1968.

Knuth, Donald E. "The Remaining Trouble Spots in ALGOL 60." Communications of the Association for Computing Machinery, Vol. 10, No. 10, October 1967, pp. 611-618.

Landin, P. J. "A Correspondence Between ALGOL 60 and Church's Lambda-Notation." Communications of the Association for Computing Machinery, Vol. 8, No. 2, February 1965, pp. 89-101 and Vol. 8, No. 3, March 1965, pp. 158-165.

Lauer, P. "Formal Definition of ALGOL 60." IBM Laboratory Vienna, Technical Report TR 25.088, December 13, 1968.

Lindsey, C. H. and S. G. van der Meulen. Informal Introduction to ALGOL 68. Amsterdam: North-Holland, 1971.

Lucas, P., P. Lauer, and H. Stigleitner. "Method and Notation for the Formal Definition of Programming Languages." IBM Laboratory Vienna, Technical Report TR 25.087, June 1968.

Lucas, P. and K. Walk. "On the Formal Description of PL/I." Annual Reviews of Automatic Programming, Vol. 6, Part 3, 1969, pp. 105-181.

McCarthy, John, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. The LISP 1.5 Programmers' Manual. Cambridge: M.I.T. Press, 1965.

Naur, P., ed. "Revised Report on the Algorithmic Language ALGOL 60." Communications of the Association for Computing Machinery, Vol. 6, No. 1, January 1963, pp. 1-17.

Neuhold, E. J. "The Formal Description of Programming Languages." IBM Systems Journal, Vol. 10, No. 2, 1971, pp. 86-112.

Randell, B. and D. J. Russell. ALGOL 60 Implementation. London: Academic Press, 1964.

Rice, R. and W. R. Smith. "SYMBOL: A Major Departure from Classic Software-Dominated von Neumann Computing Systems." Proceedings of the Spring Joint Computer Conference, Vol. 38, 1971, pp. 575-587.

Smith, W. R., R. Rice, G. D. Chesley, T. A. Lalliotis, S. F. Lundstrom, M. A. Calhoun, L. D. Gerould, and T. G. Cook. "SYMBOL: A Large Experimental System Exploring Major Hardware Replacement of Software." Proceedings of the Spring Joint Computer Conference, Vol. 38, 1971, pp. 601-616.

Steel, T. B., ed. Proceedings of the IFIP Working Conference on Formal Languages for Computer Programming. Amsterdam: North-Holland, 1966.

Warshall, S. and R. M. Shapiro. "A General-Purpose Table-Driven Compiler." Programming Systems and Languages. Ed. Saul Rosen. New York: McGraw-Hill, 1967, pp. 332-341.

Wirth, Nicklaus and Helmut Weber. "EULER: A Generalization of ALGOL, and its Formal Definition." Communications of the Association for Computing Machinery, Vol. 9, No. 3, January 1966, pp. 13-23, 25 and Vol. 9, No. 4, February 1966, pp. 89-99.

Zwicky, Arnold M., Joyce Friedman, Barbara C. Hall, and Donald E. Walker. "The MITRE Analysis Procedure for Transformational Grammars." Proceedings of the Fall Joint Computer Conference, Vol. 27, 1965, pp. 317-326.

Appendix I

SEMANTIC SPECIFICATION OF ALGOL 60

The table below gives the abbreviations used throughout this work for metalinguistic variables and the corresponding variables. In the text these abbreviations are enclosed within angle brackets ("<" and ">").

ABBREVIATION	METALINGUISTIC VARIABLE
ac	actual parameter
ad	adding operator
ae	arithmetic expression
ai	array identifier
al	array list
an	assignment statement
ao	arithmetic operator
ap	actual parameter part
ar	array declaration
as	array segment
at	actual parameter list
ba	basic symbol
be	Boolean expression
bf	Boolean factor
bh	block head
bk	bracket
bl	block
bn	bound pair list
bo	Boolean secondary
bp	Boolean primary
br	bound pair
bs	basic statement
bt	Boolean term
cn	conditional statement
co	code
cs	compound statement
ct	compound tail
dc	declaration
de	designational expression
df	decimal fraction
di	digit

dl	delimiter
dn	decimal number
dr	declarator
ds	dummy statement
em	empty
ep	exponent part
ex	expression
fa	factor
fc	for clause
fd	function designator
fe	for list element
fl	formal parameter list
fo	formal parameter
fp	formal parameter part
fs	for statement
ft	for list
gs	go to statement
ic	if clause
id	identifier
il	identifier list
im	implication
in	integer
is	if statement
la	label
lb	lower bound
le	letter
ll	left part list
lo	logical operator
lp	left part
ls	letter string
lt	local or own type
lv	logical value
mo	multiplying operator
no	number
op	operator
os	open string
pa	parameter delimiter
pb	procedure body
pc	procedure
pd	procedure declaration
pg	proper string
ph	procedure heading
pi	procedure identifier
po	program
pr	primary
ps	procedure statement
rl	relation
ro	relational operator
sa	simple arithmetic expression
sb	simple Boolean
sc	specificator
sd	simple designational expression

se	subscript expression
sg	string
si	switch identifier
sl	subscript list
sn	switch declaration
so	sequential operator
sp	specification part
sr	specifier
ss	switch list
st	statement
su	subscripted variable
sv	simple variable
sw	switch designator
td	type declaration
tl	type list
tm	term
ty	type
ub	unlabelled block
uc	unlabelled compound
ui	unsigned integer
ul	unlabelled basic statement
un	unsigned number
up	upper bound
us	unconditional statement
vi	variable identifier
vp	value part
vr	variable

The remainder of this appendix is concerned with a semantic specification for Algol 60. The table below summarizes the functions performed by the application of the 218 transformations of that specification to Algol 60 programs. The headings indicating the functions refer to removing superfluous syntax (remove), making operations explicit (insert), counting, specifying order of execution (arrange), and copying, as explained in Chapter 4.

FUNCTION	TRANSFORMATION NUMBER												
	1	2	3	4	5	6	7	8	9	10	11	12	13
REMOVE		X	X	X	X	X	X	X	X	X	X	X	X
INSERT						X	X						
COUNT				X		X	X	X	X	X	X	X	X
ARRANGE													
COPY	X	X											
	14	15	16	17	18	19	20	21	22	23	24	25	26
REMOVE	X	X	X	X	X	X		X		X	X	X	X
INSERT			X	X									
COUNT	X	X			X		X	X	X	X	X	X	
ARRANGE													
COPY													
	27	28	29	30	31	32	33	34	35	36	37	38	39
REMOVE	X		X		X			X		X		X	X
INSERT		X											
COUNT				X					X		X		
ARRANGE													
COPY				X		X	X	X			X		
	40	41	42	43	44	45	46	47	48	49	50	51	52
REMOVE	X	X	X					X					X
INSERT		X	X	X	X		X		X	X	X		
COUNT						X		X	X	X	X	X	
ARRANGE				X	X								
COPY		X	X										
	53	54	55	56	57	58	59	60	61	62	63	64	65
REMOVE	X	X	X	X	X	X	X	X		X	X		X
INSERT									X			X	X
COUNT													
ARRANGE													
COPY									X		X		
	66	67	68	69	70	71	72	73	74	75	76	77	78
REMOVE	X	X	X		X	X	X	X	X	X	X	X	X
INSERT										X			
COUNT	X			X	X	X					X		
ARRANGE							X	X		X			
COPY													X

	79	80	81	82	83	84	85	86	87	88	89	90	91
REMOVE	X				X	X		X				X	X
INSERT		X	X	X					X	X	X		
COUNT					X		X	X					
ARRANGE		X	X	X			X						
COPY													
	92	93	94	95	96	97	98	99	100	101	102	103	104
REMOVE	X	X	X	X	X	X			X	X	X	X	X
INSERT	X	X					X	X					
COUNT			X	X	X	X			X	X	X		
ARRANGE											X	X	X
COPY													
	105	106	107	108	109	110	111	112	113	114	115	116	117
REMOVE	X	X	X	X				X	X	X	X		X
INSERT			X	X	X	X	X			X	X	X	
COUNT													
ARRANGE		X											X
COPY													
	118	119	120	121	122	123	124	125	126	127	128	129	130
REMOVE	X	X	X	X	X	X	X	X	X	X			X
INSERT					X					X			
COUNT												X	X
ARRANGE	X	X	X			X					X		
COPY													
	131	132	133	134	135	136	137	138	139	140	141	142	143
REMOVE													
INSERT	X												
COUNT	X	X	X	X							X	X	
ARRANGE													
COPY		X	X	X	X	X	X	X	X	X			X
	144	145	146	147	148	149	150	151	152	153	154	155	156
REMOVE								X					
INSERT							X		X			X	X
COUNT						X					X	X	X
ARRANGE													
COPY	X	X	X	X	X					X			

	157	158	159	160	161	162	163	164	165	166	167	168	169
REMOVE		X						X		X	X	X	
INSERT	X										X	X	X
COUNT	X		X	X	X	X	X		X				
ARRANGE													
COPY			X	X	X	X	X	X	X				

	170	171	172	173	174	175	176	177	178	179	180	181	182
REMOVE				X			X						
INSERT	X	X	X		X	X							
COUNT								X	X	X			
ARRANGE													
COPY											X	X	X

	183	184	185	186	187	188	189	190	191	192	193	194	195
REMOVE													
INSERT	X	X	X	X	X	X	X	X	X	X			
COUNT											X	X	
ARRANGE			X	X	X	X	X	X	X	X			
COPY											X	X	X

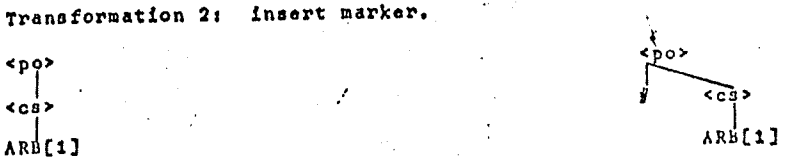
	196	197	198	199	200	201	202	203	204	205	206	207	208
REMOVE													
INSERT	X	X	X	X	X	X	X	X	X	X	X	X	X
COUNT													
ARRANGE													
COPY					X	X	X	X	X	X	X	X	X

	209	210	211	212	213	214	215	216	217	218
REMOVE										
INSERT	X	X								X
COUNT			X						X	
ARRANGE				X	X	X	X	X		
COPY	X	X								

A semantic specification for Algol 60, containing 218 transformations, is given below. Each transformation is a pair of trees, the first a match tree and the second a substitution tree as described in Chapter 2. A grouping of the transformations by the particular syntactic constructs on which they operate is contained in Appendix IV.

Transformation 1: insert marker.

Transformation 2: insert marker.

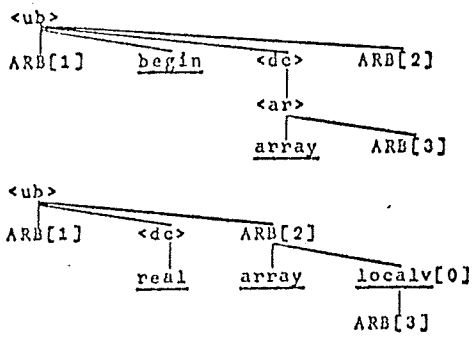


Transformation 3: bring all the statements in a compound tail to the same level.

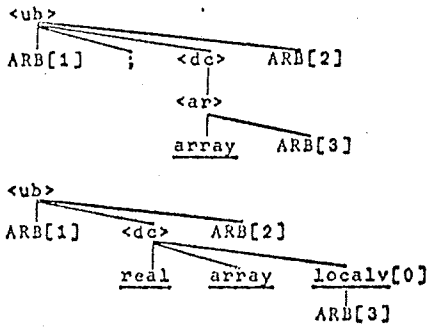
Transformation 4: insert statement count node for counting to be done in transformations 70 and 71.

Transformation 5: bring all the declarations in a block head to the same level.

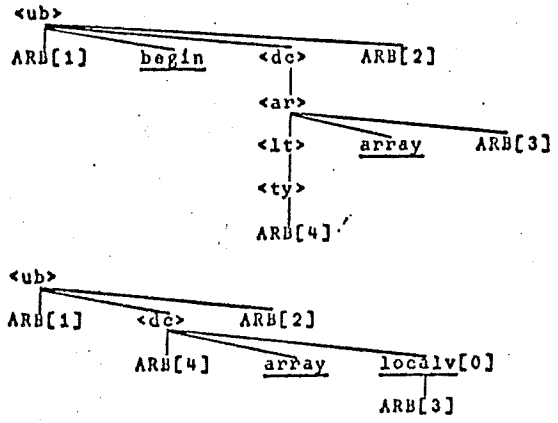
Transformation 6: insert local variable count node for array declaration and make the default array type (real) explicit.



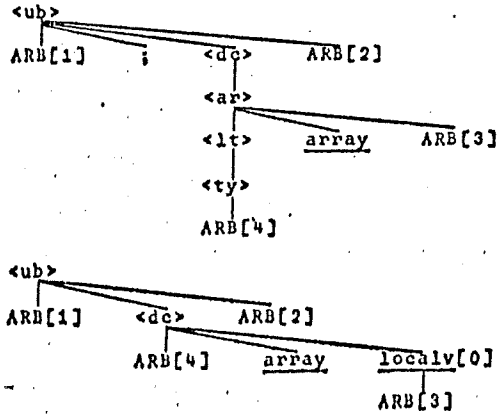
Transformation 7: insert local variable count node for array declaration and make the default array type (real) explicit.



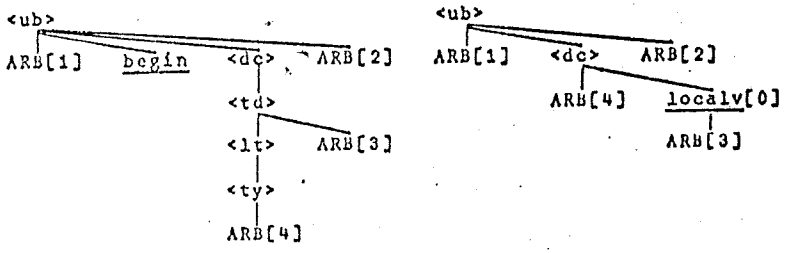
Transformation 8: insert local variable count node for array declaration.



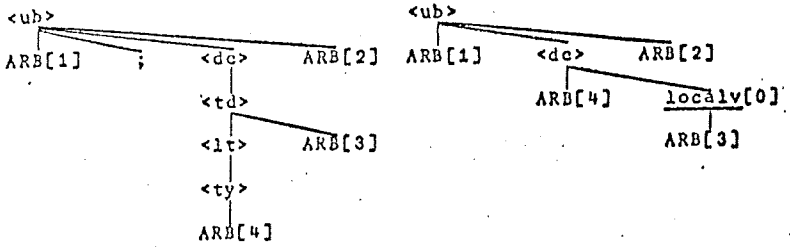
Transformation 9: insert local variable count node for array declaration.



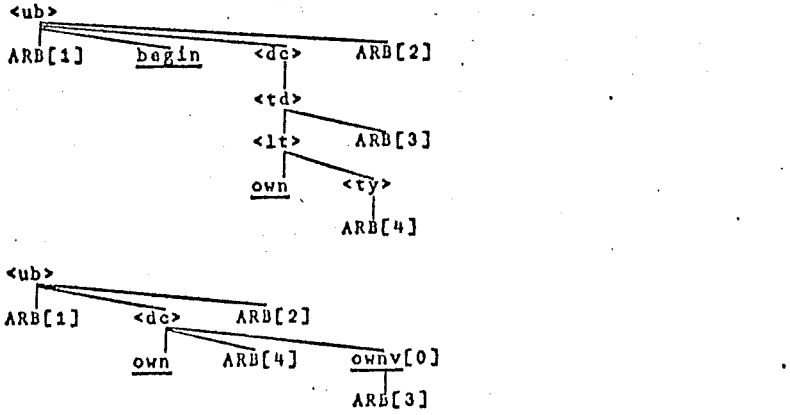
Transformation 10: insert local variable count node for simple variable declaration.



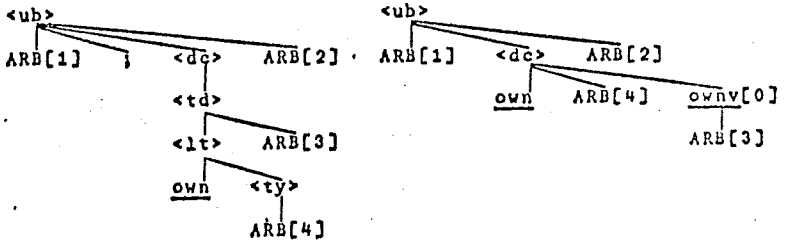
Transformation 11: insert local variable count node for simple variable declaration.



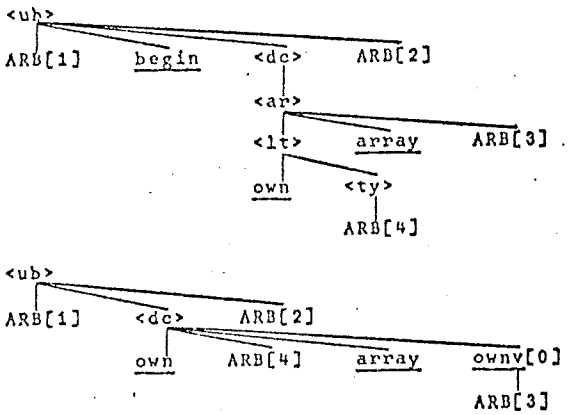
Transformation 12: insert own variable count node for own variable declaration.



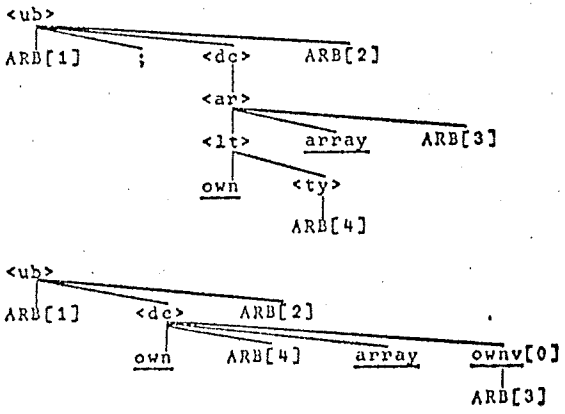
Transformation 13: insert own variable count node for own variable declaration.



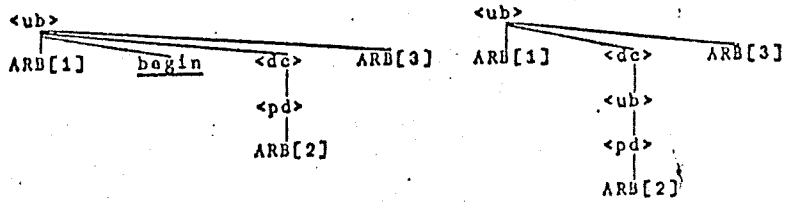
Transformation 14: insert own variable count node for own array declaration.



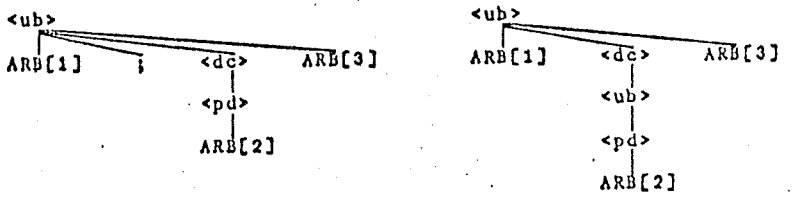
Transformation 15: insert own variable count node for own array declaration.



Transformation 16: make separate block for procedure declaration.



Transformation 17: make separate block for procedure declaration.

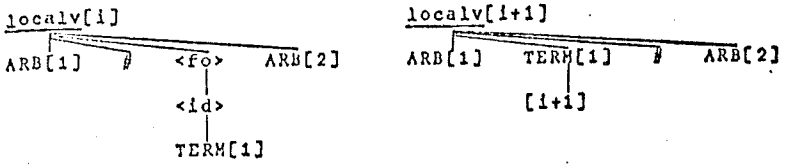


Transformation 18: insert local variable count node for formal parameter list.

Transformation 19: bring all the formal parameters in a formal parameter list to the same level and remove parameter delimiters.

Transformation 20: insert markers in preparation for counting formal parameters.

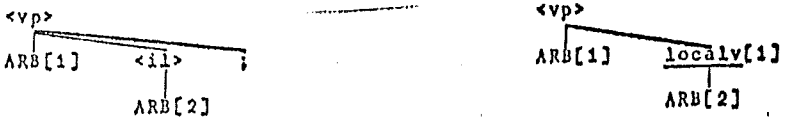
Transformation 21: count formal parameter, associate displacement number with it, and move marker to prevent the parameter from being counted again.



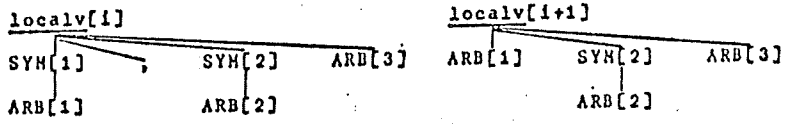
Transformation 22: counting done, remove markers.

Transformation 23: insert local variable count node for specification part.

Transformation 24: insert local variable count node for value part.



Transformation 25: count specification or value variable and make change to prevent the instance from being counted again.



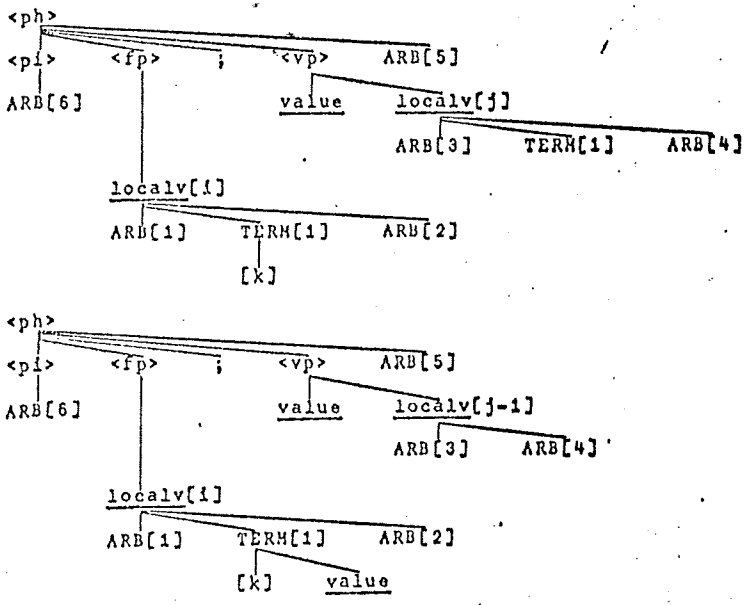
Transformation 26: remove superfluous identifier node.

Transformation 27: bring all specifiers to same level.

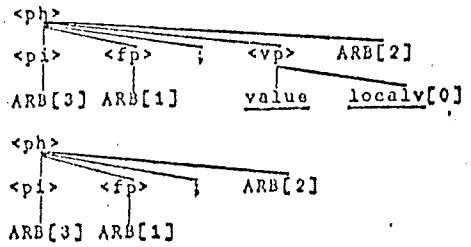
Transformation 28: make the default array specifier type (real) explicit.

Transformation 29: remove superfluous type node.

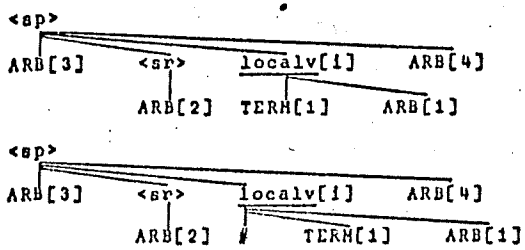
Transformation 30: copy value attribute to formal parameter, remove formal parameter name from value list, and decrement value list count.



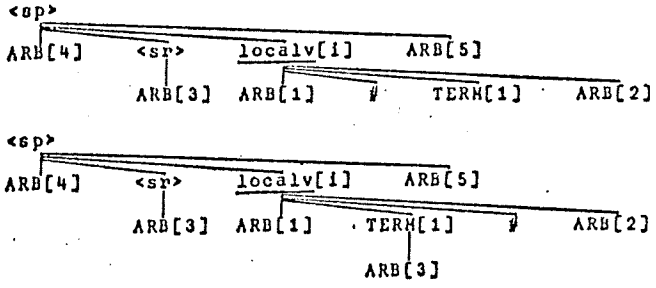
Transformation 31: all formal parameter names removed from value list, remove value part.



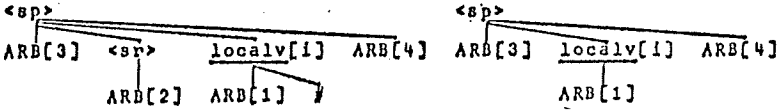
Transformation 32: insert marker in preparation for copying specifier to specification variable.



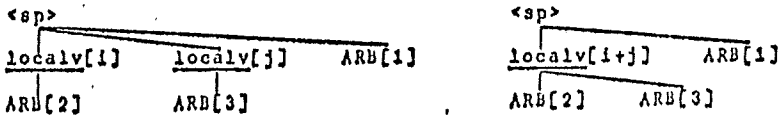
Transformation 33: copy specifier to specification variable and move marker over that variable.



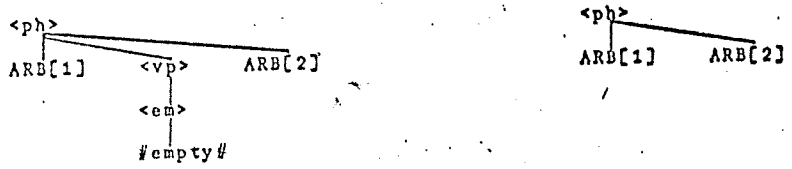
Transformation 34: copying done, remove marker.



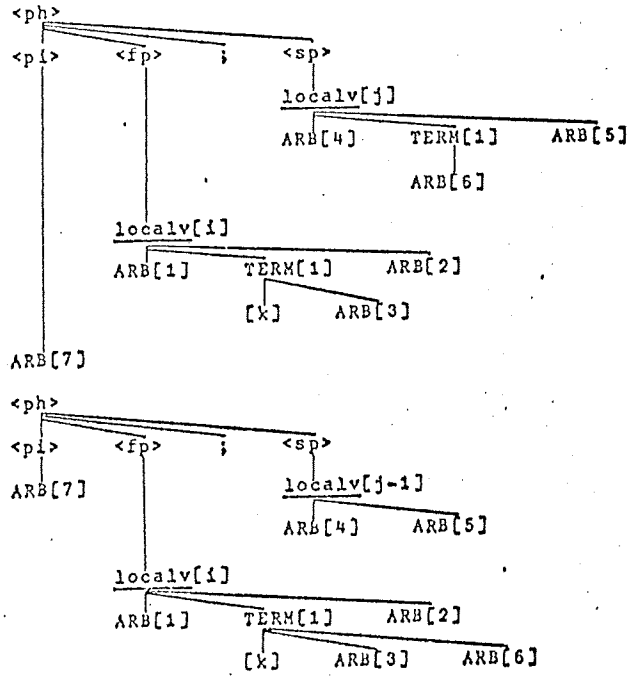
Transformation 35: combine specification variable lists.



Transformation 36: remove empty value part.



Transformation 37: copy attribute of formal parameter named in specification list to formal parameter, remove name from specification list, and decrement specification list count.

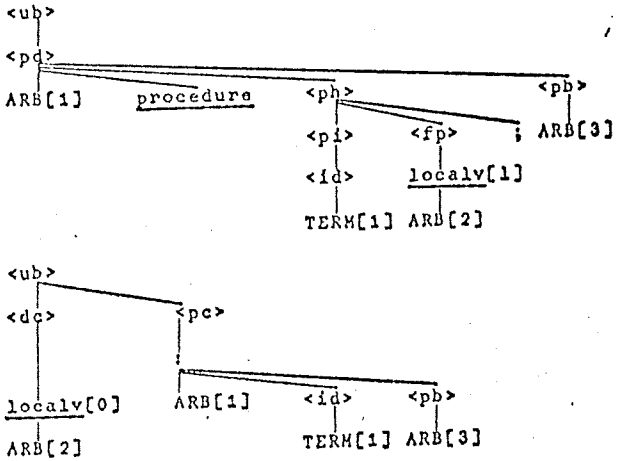


Transformation 38: all formal parameters on specification list removed, remove specification part.

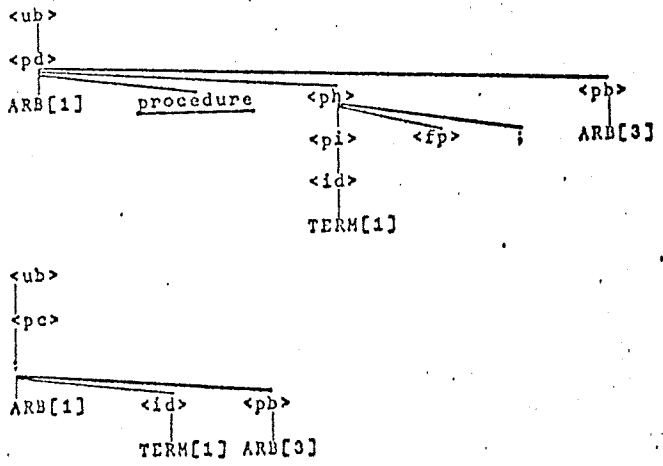
Transformation 39: remove empty specification part.

Transformation 40: remove superfluous type node.

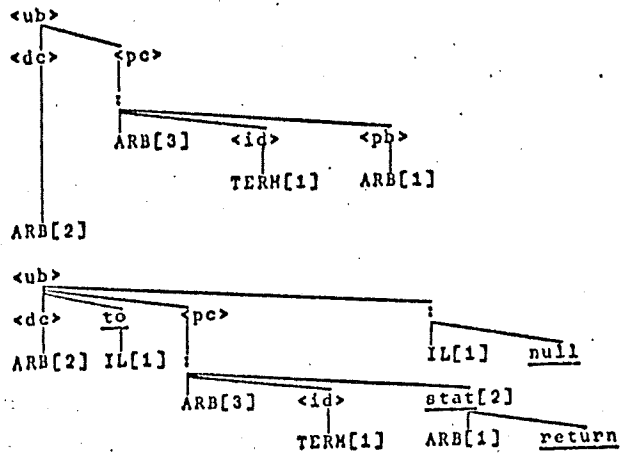
Transformation 41: make formal parameter part into declaration local to procedure block; make procedure name a label on the procedure body.



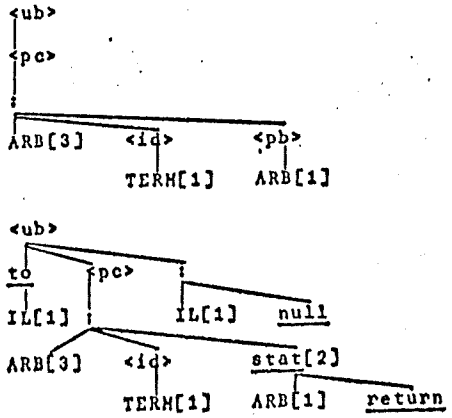
Transformation 42: make procedure name a label on the procedure body.



Transformation 43: insert jump around procedure declaration and insert return in procedure.

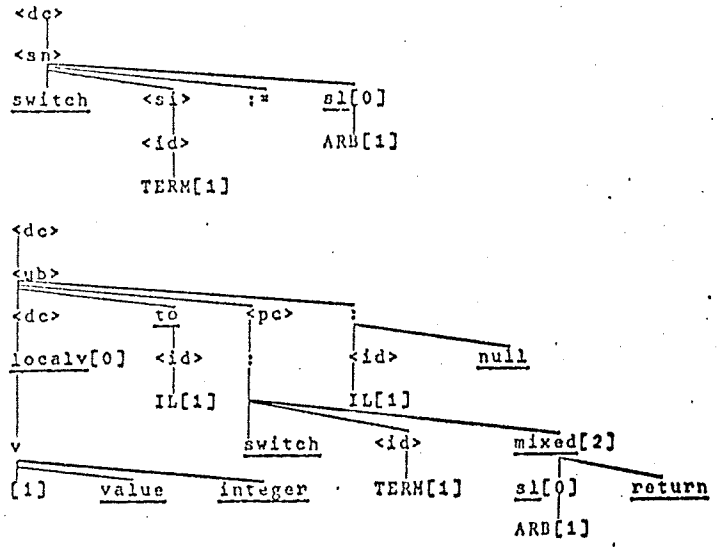


Transformation 44: insert jump around procedure declaration and insert return in procedure.



Transformation 45: insert count node in preparation for counting designational expressions on a switch list.

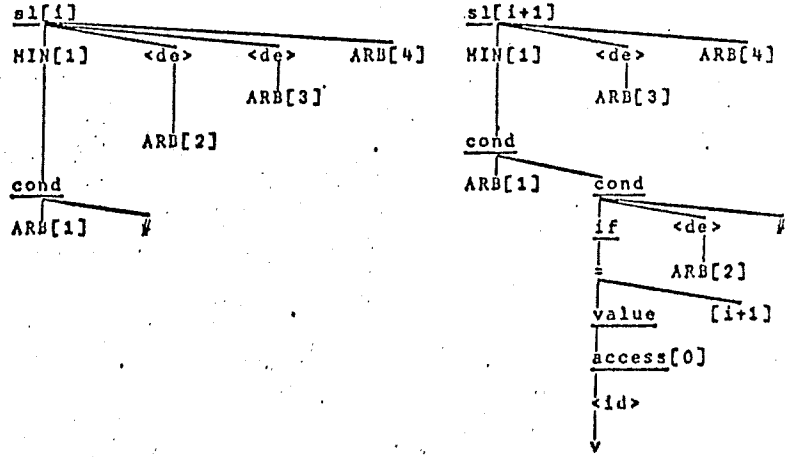
Transformation 46: make switch declaration into procedure declaration and create a separate block for it.



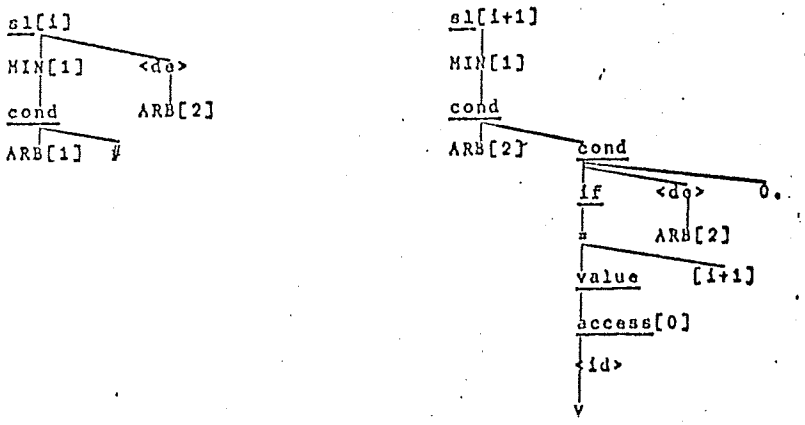
Transformation 47: bring all designational expressions in a switch list to the same level.

Transformation 48: make first designational expression into conditional expression and count.

Transformation 49: make designational expression in a switch list into conditional expression and count.

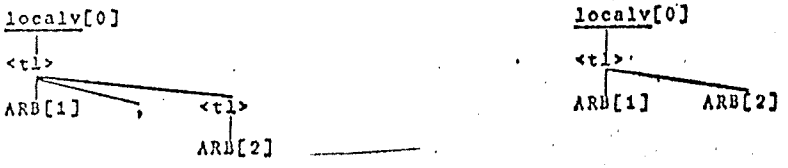


Transformation 50: make last designational expression in a switch list into conditional expression and count.



Transformation 51: remove count node for designational expressions.

Transformation 52: bring all local variables in type list to same level.

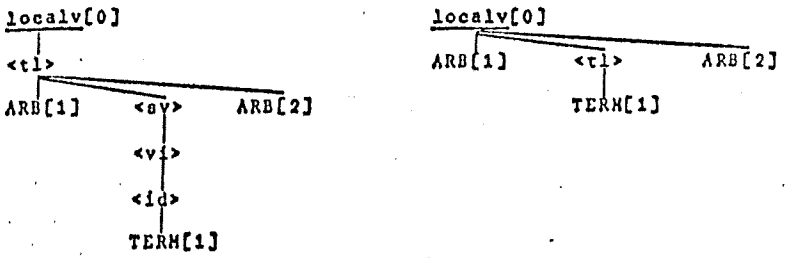


Transformation 53: bring all own variables in type list to same level.

Transformation 54: bring all local variables in array list to same level.

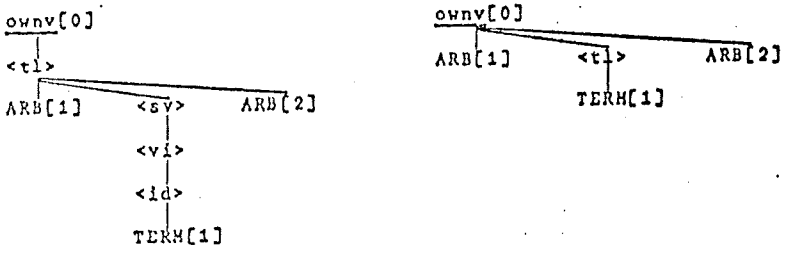
Transformation 55: bring all own variables in array list to same level.

Transformation 56: remove superfluous nodes in type lists.



Transformation 57: remove type list.

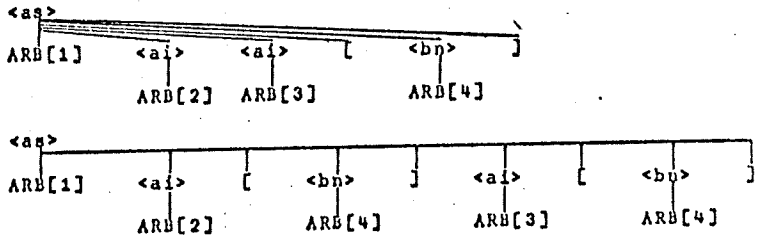
Transformation 58: remove superfluous nodes in type lists.



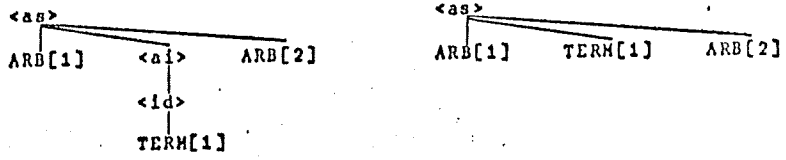
Transformation 59: remove type list.

Transformation 60: remove array declaration delimiters.

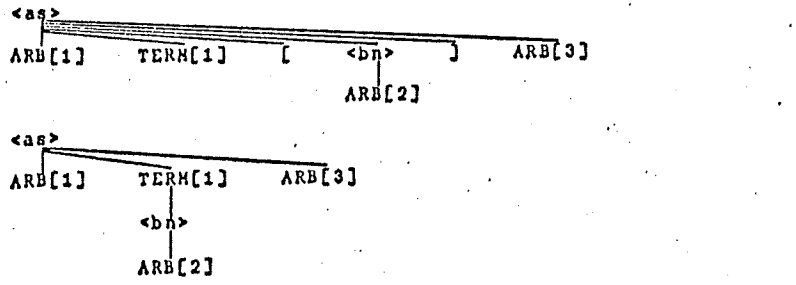
Transformation 61: copy subscript range to each array identifier to which it applies.



Transformation 62: remove superfluous nodes.



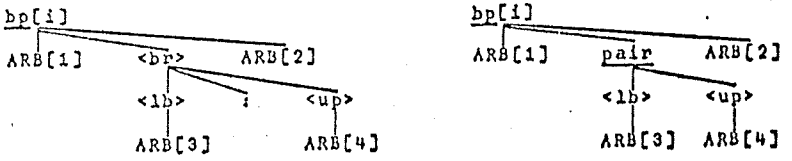
Transformation 63: remove superfluous nodes and make subscript range an attribute of array identifier.



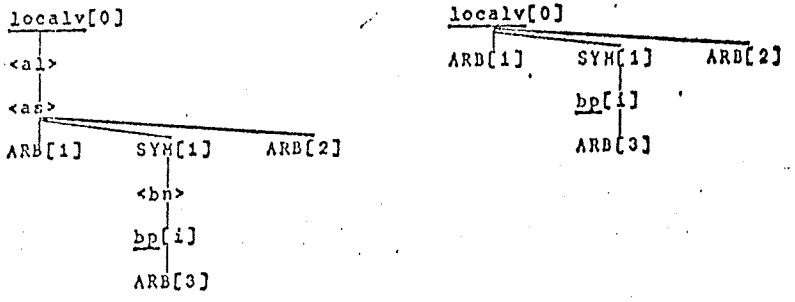
Transformation 64: insert bound pair count node.

Transformation 65: count bound pair and remove delimiter to prevent that instance from being counted again.

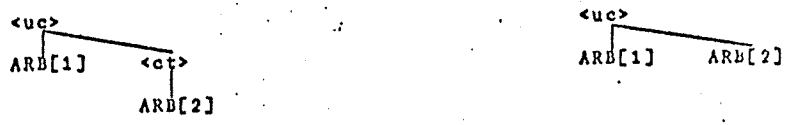
Transformation 66: insert lower-upper bound pair connector node and remove superfluous nodes.



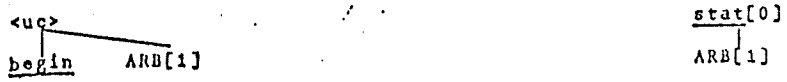
Transformation 67: remove superfluous nodes in array declaration.



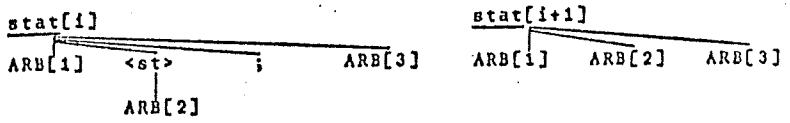
Transformation 68: bring all statements in a compound tail to the same level.



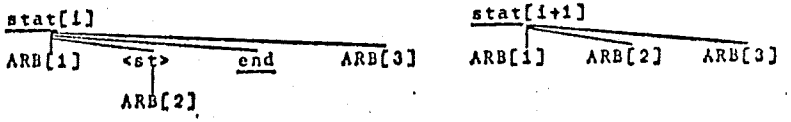
Transformation 69: insert statement count node.



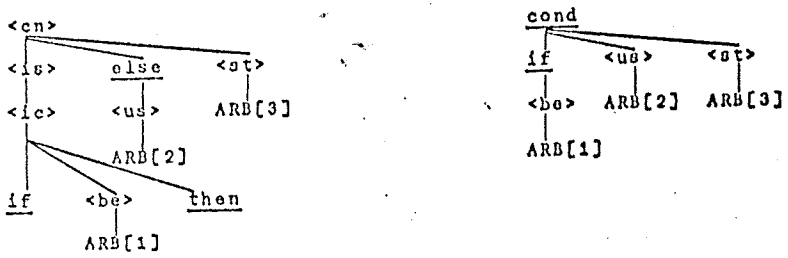
Transformation 70: count statement and remove nodes to prevent the statement from being counted again.



Transformation 71: count statement and remove nodes to prevent the statement from being counted again.



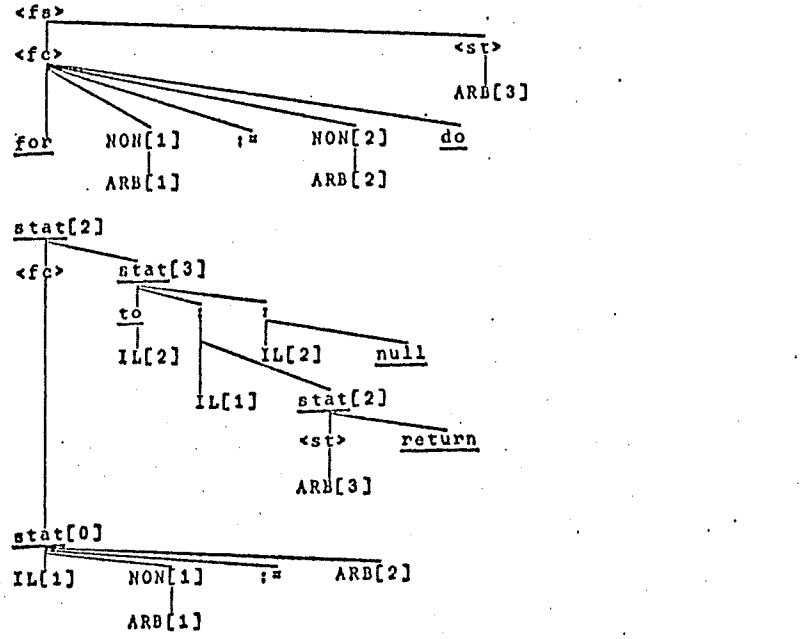
Transformation 72: convert if-then-else conditional statement to computation tree form and remove superfluous nodes.



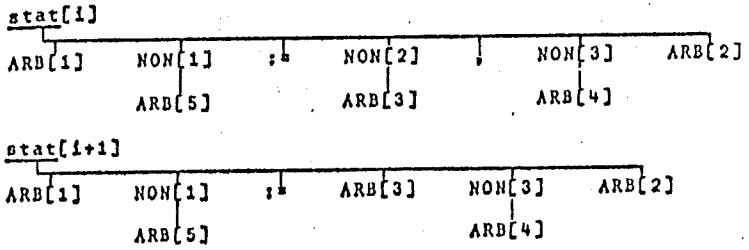
Transformation 73: convert if-then conditional statement to computation tree form, supply default else condition, and remove superfluous nodes.

Transformation 74: convert go to statement to computation tree form and remove superfluous nodes.

Transformation 75: convert for statement to computation tree form, making order of execution explicit, and insert for list element count node and internal label.

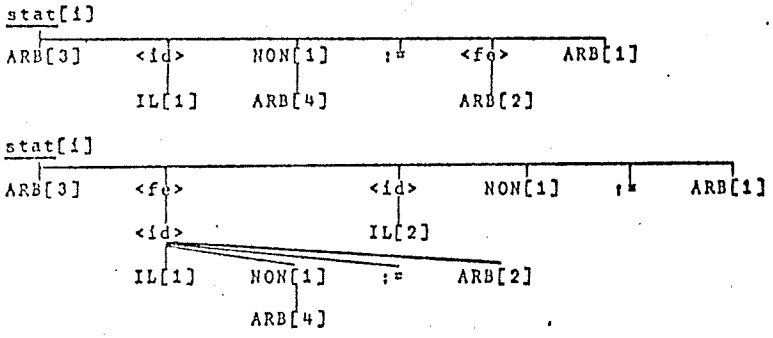


Transformation 76: count elements of for list and remove delimiter to prevent that element from being counted again.

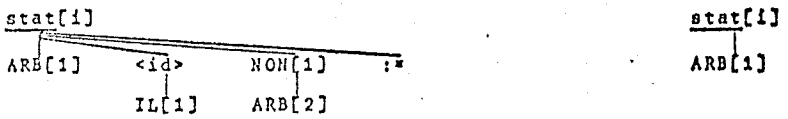


Transformation 77: remove superfluous for list node.

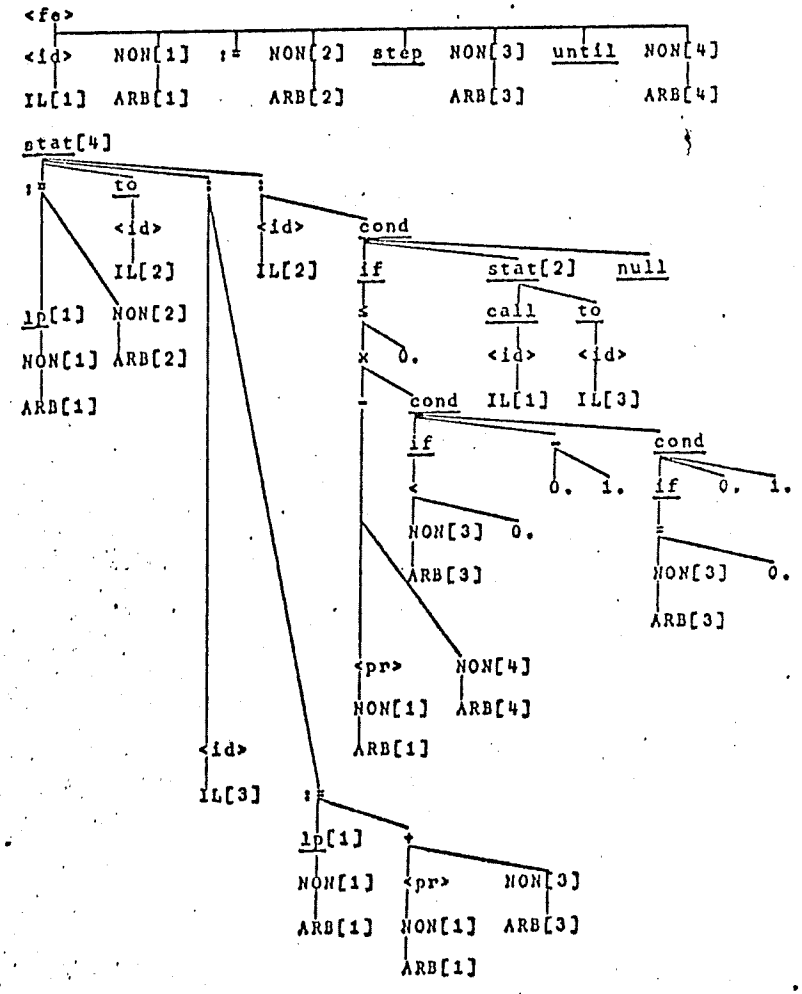
Transformation 78: copy and label for loop variable for each for list element and make change to prevent copying for that element again.



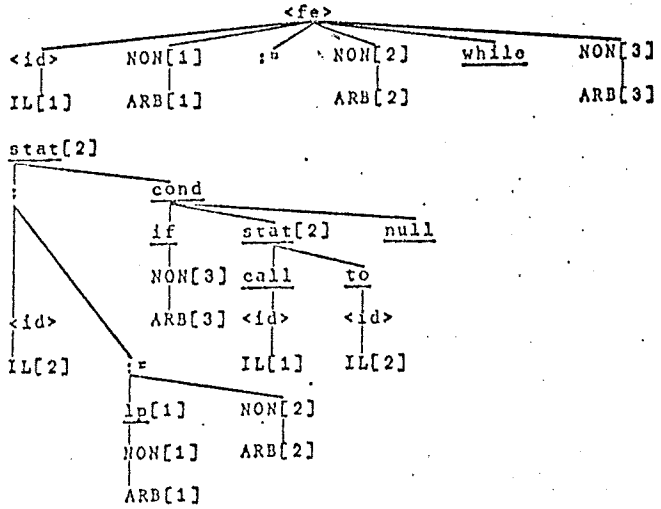
Transformation 79: copying and labeling done, remove superfluous nodes.



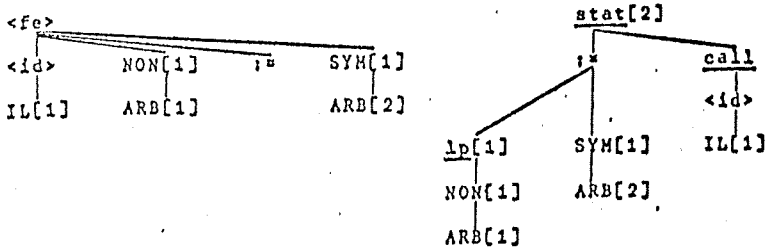
Transformation 80: convert step-until for list element to computation tree form, making operations and order of execution explicit.



Transformation 81: convert while for list element to computation tree form, making operations and order of execution explicit.



Transformation 82: convert remaining for list elements to computation tree form, making operations and order of execution explicit.

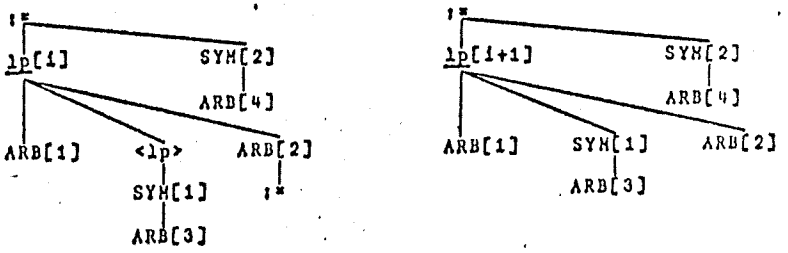


Transformation 83: combine for lists in a for clause.

Transformation 84: bring left parts in an assignment statement to the same level.

Transformation 85: convert assignment statement to computation tree form, making order of execution explicit and insert left part counter node.

Transformation 86: count left part and remove superfluous nodes.



Transformation 87: insert address fetch node in simple variable expression.

Transformation 88: insert value fetch node in primary.

Transformation 89: insert value fetch node in Boolean primary.

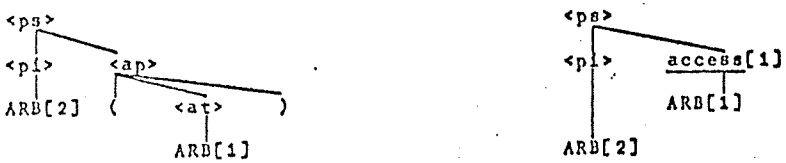
Transformation 90: remove parentheses from primary.

Transformation 91: remove parentheses from Boolean primary.

Transformation 92: convert procedure call statement with no arguments to computation tree form.

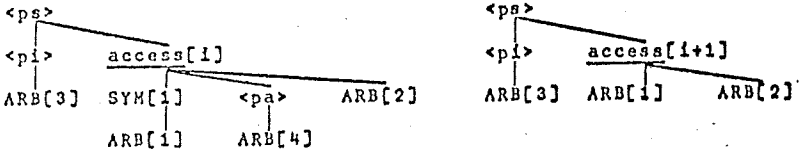
Transformation 93: convert function designator with no arguments to computation tree form.

Transformation 94: insert argument count node in procedure call statement and remove parentheses around argument list.

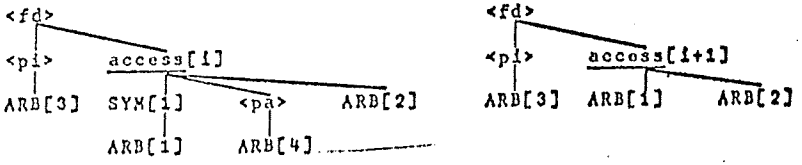


Transformation 95: insert argument count node in function designator and remove parentheses around argument list.

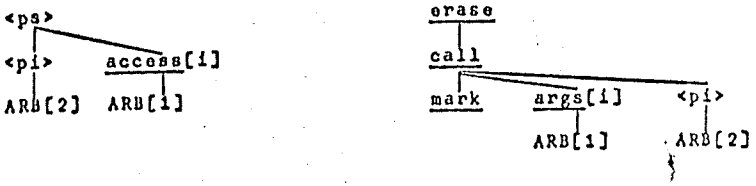
Transformation 96: count arguments in procedure call statement and remove delimiters.



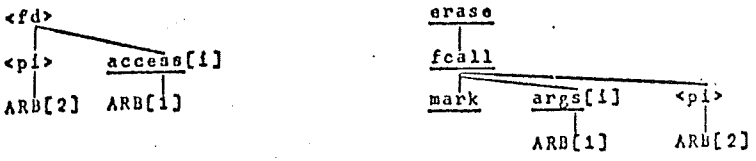
Transformation 97: count arguments in function designator and remove delimiters.



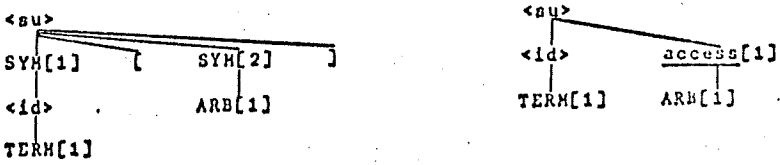
Transformation 98: convert procedure call statement with arguments to computation tree form.



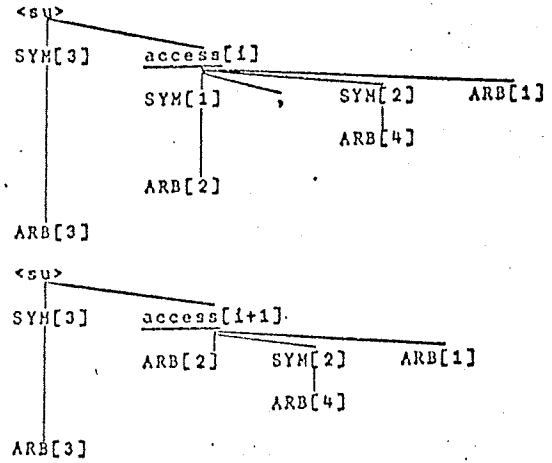
Transformation 99: convert function designator with arguments to computation tree form.



Transformation 100: insert count node for subscript expressions and remove superfluous nodes.

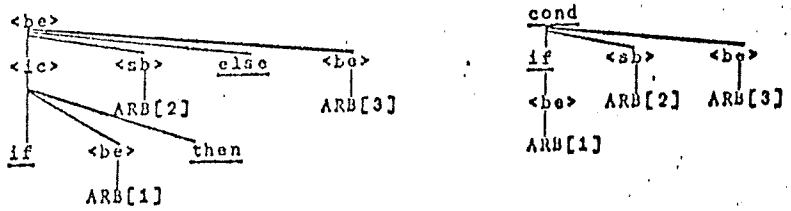


Transformation 101: count subscript expression and remove nodes to prevent the expression from being counted again.

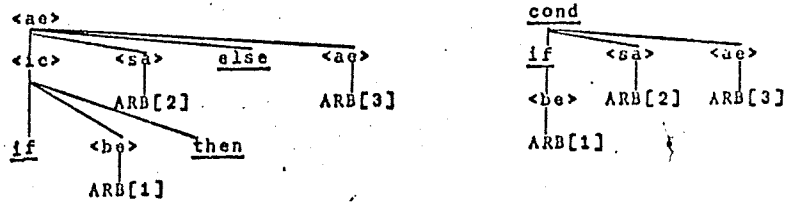


Transformation 102: counting done, remove superfluous nodes and convert to computation tree order.

Transformation 103: convert if-then-else Boolean expression to computation tree form and remove superfluous nodes.

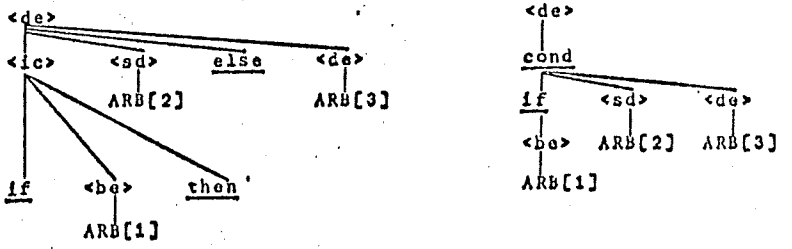


Transformation 104: convert if-then-else arithmetic expression to computation tree form and remove superfluous nodes.



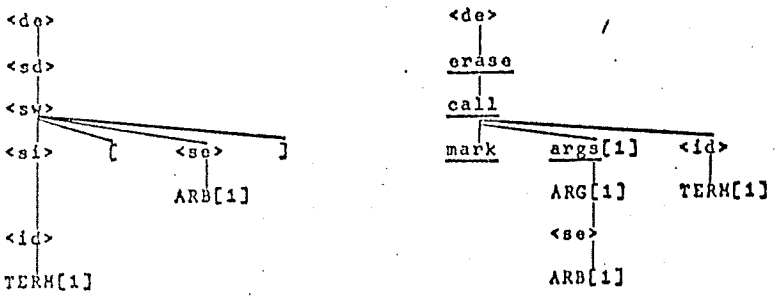
Transformation 105: remove superfluous nodes in designational expression.

Transformation 106: convert if-then-else designational expression to computation tree form and remove superfluous nodes.



Transformation 107: remove superfluous nodes and convert label to designational expression.

Transformation 108: convert switch designator to call on switch procedure.



Transformation 109: convert actual parameter node to argument node.

Transformation 110: convert array identifier argument nodes to dope vector indicator.

Transformation 111: convert procedure identifier argument nodes to procedure indicator.

Transformation 112: remove superfluous nodes from simple variable argument.

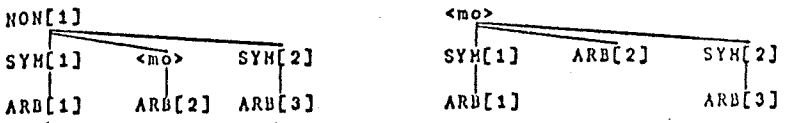
Transformation 113: remove superfluous nodes from Boolean variable argument.

Transformation 114: remove superfluous nodes from numeric argument and convert to constant indicator.

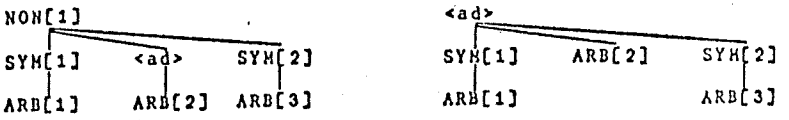
Transformation 115: remove superfluous nodes from Boolean constant argument and convert to constant indicator.

Transformation 116: convert designational expression node to designatonal indicator.

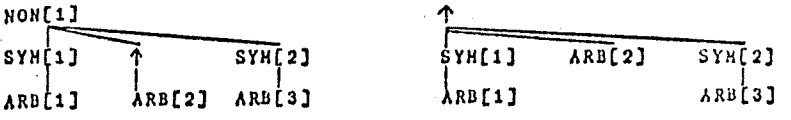
Transformation 117: convert expression involving multiplication or division to computation tree form.



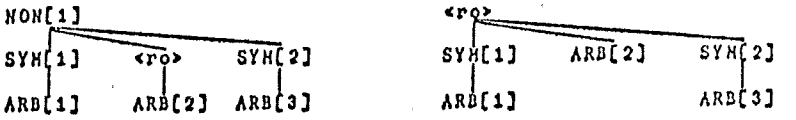
Transformation 118: convert expression involving addition or subtraction to computation tree form.



Transformation 119: convert expression involving exponentiation to computation tree form.



Transformation 120: convert expression involving a relational operator to computation tree form.



Transformation 121: remove unary plus.

Transformation 122: convert unary minus to subtraction from zero.

Transformation 123: convert negation to computation tree form.

Transformation 124: remove superfluous multiplication class nodes.

Transformation 125: remove superfluous addition class node.

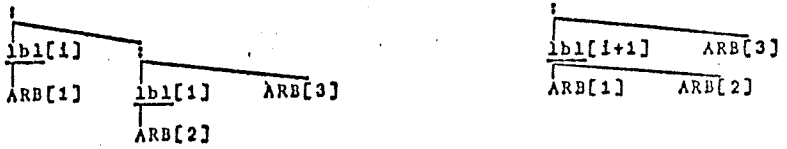
Transformation 126: remove superfluous relational class node.

Transformation 127: replace empty by null statement.

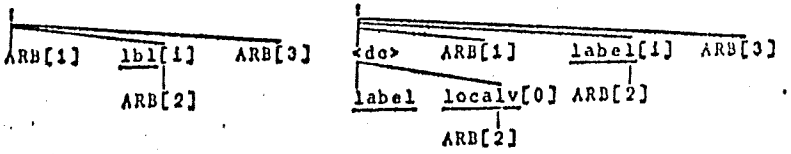
Transformation 128: convert labeled statement to computation tree form and insert label count node.

Transformation 129: insert internal label count node.

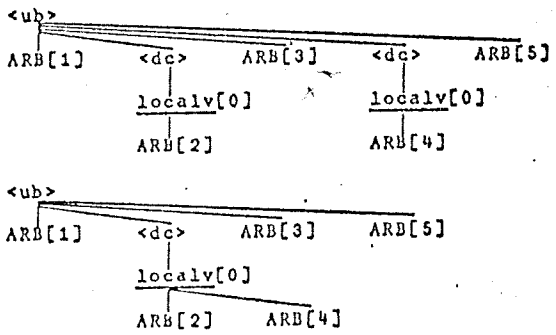
Transformation 130: count labels on a statement and bring to the same level.



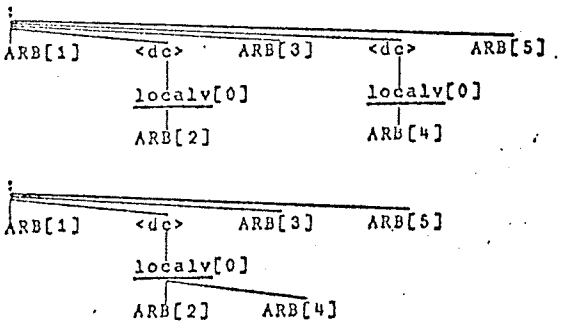
Transformation 131: insert declaration for label and declaration count node.



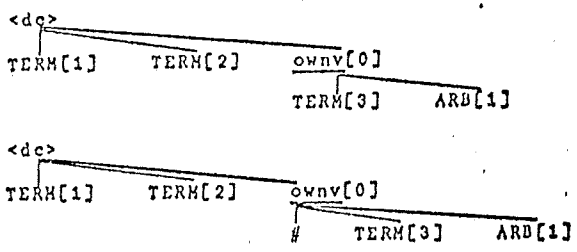
Transformation 141: combine local declarations.



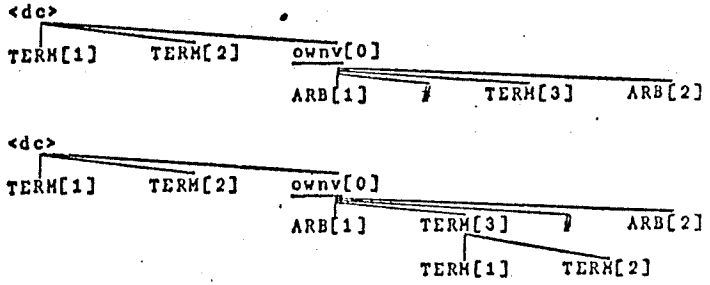
Transformation 142: combine label declarations.



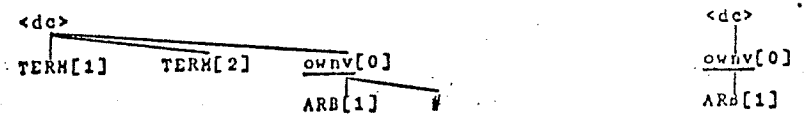
Transformation 143: insert marker in preparation for copying type attribute to own variable.



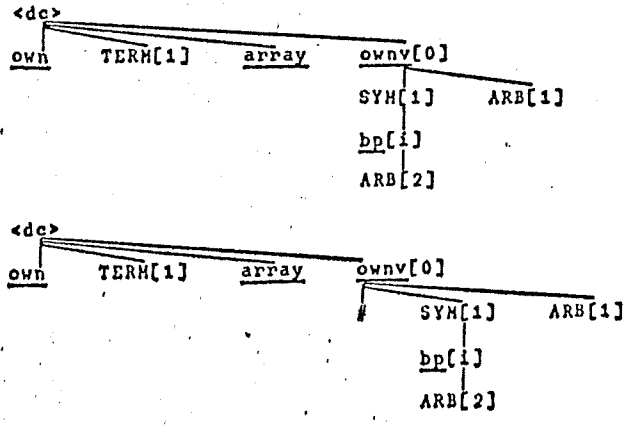
Transformation 144: copy type attribute to own variable.



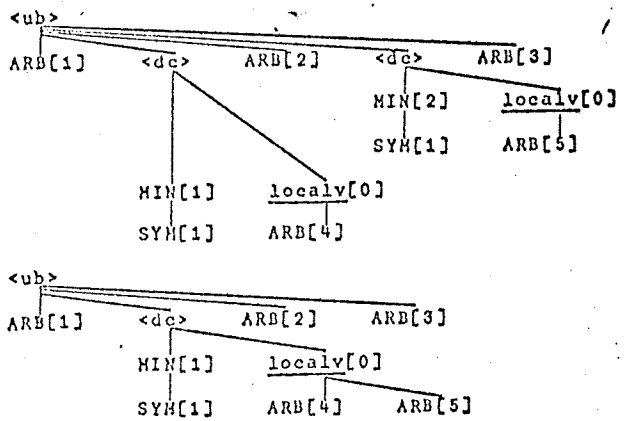
Transformation 145: copying done for own variable, remove marker and attribute.



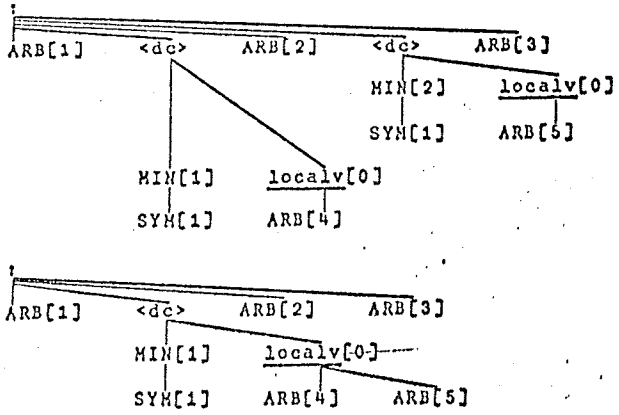
Transformation 146: insert marker in preparation for copying type attribute to own array.



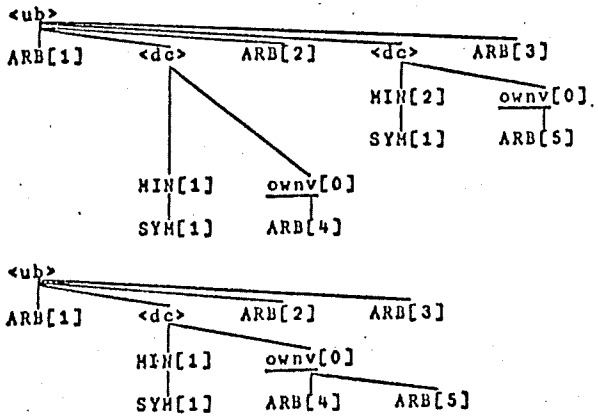
Transformation 132: combine declarations for local variables of same type in a block.



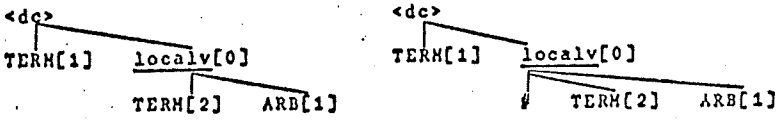
Transformation 133: combine declarations for labels on a statement.



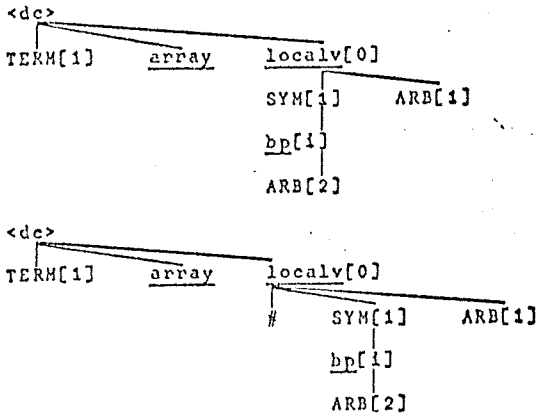
Transformation 134: combine declarations for own variables of same type in a block.



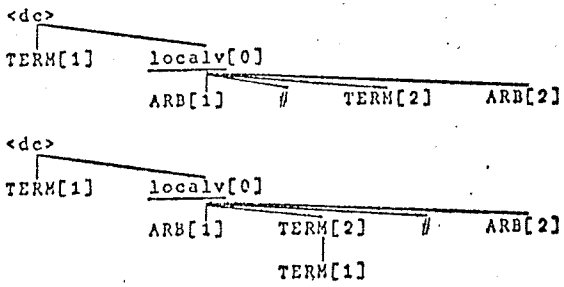
Transformation 135: insert marker in preparation for copying type attribute for local variable.



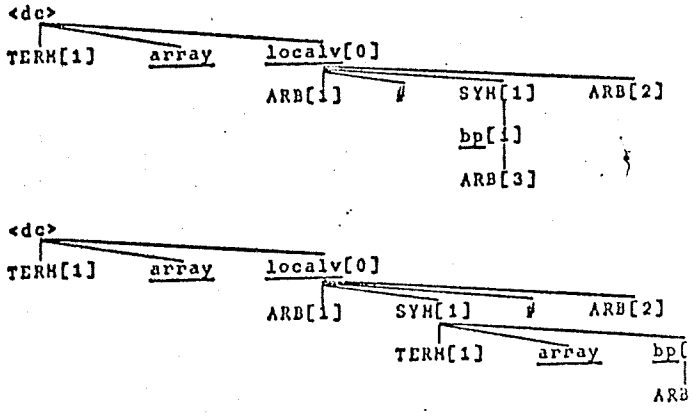
Transformation 136: insert marker in preparation for copying type attribute to local array.



Transformation 137: copy type attribute to local variable or label and move marker.

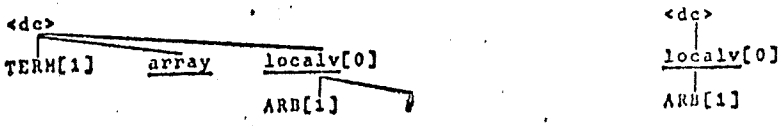


Transformation 138: copy type attribute to local array and move marker.

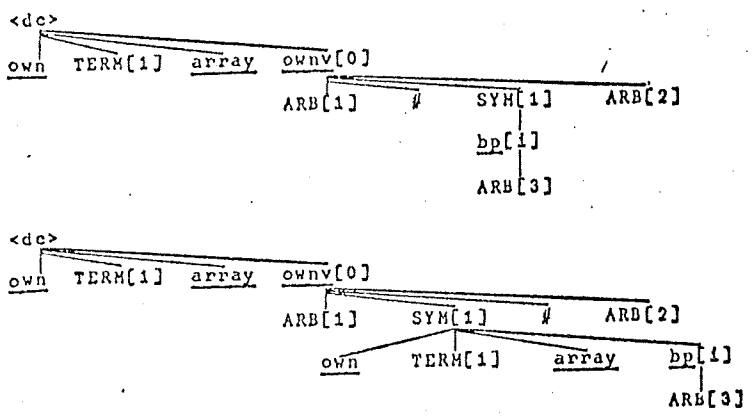


Transformation 139: copying done for variable, remove marker and attribute.

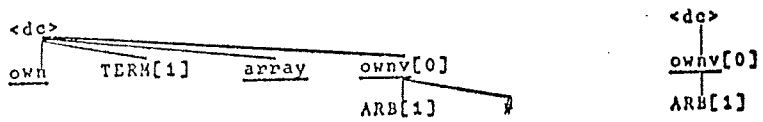
Transformation 140: copying done for array, remove marker and attribute.



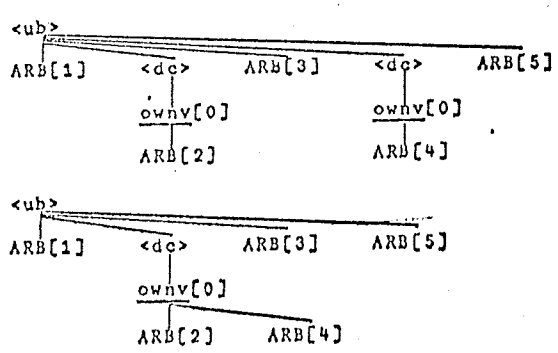
Transformation 147: copy type attribute to own array and move marker.



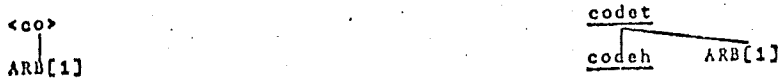
Transformation 148: copying done for array, remove marker and attribute.



Transformation 149: combine own declarations.



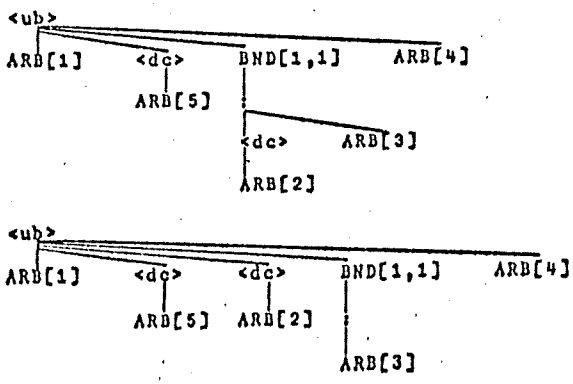
Transformation 150: mark the beginning and end of a code (machine-dependent) procedure body.



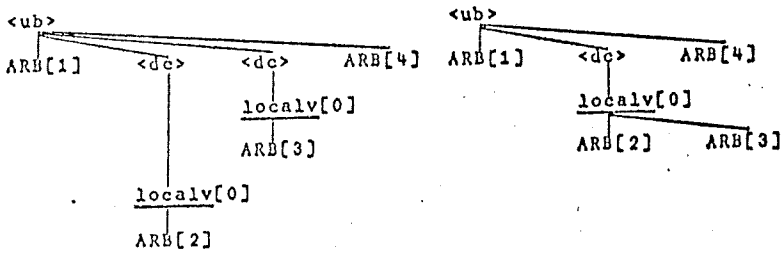
Transformation 151: remove first of two nonterminal nodes.

Transformation 152: augment program tree with symbol table branch.

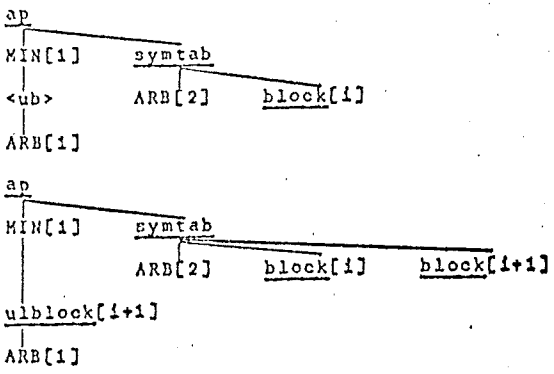
Transformation 153: move label declaration to block in which it is contained.



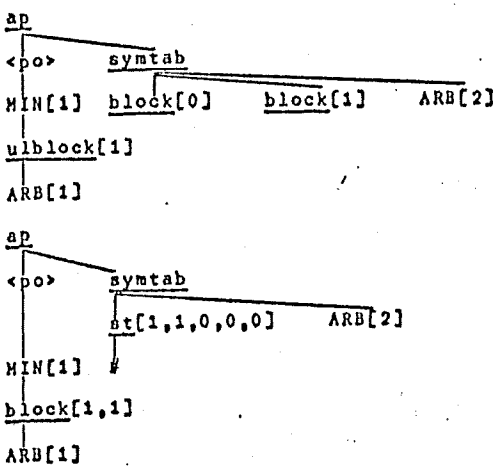
Transformation 154: combine label declaration with other declarations in a block.



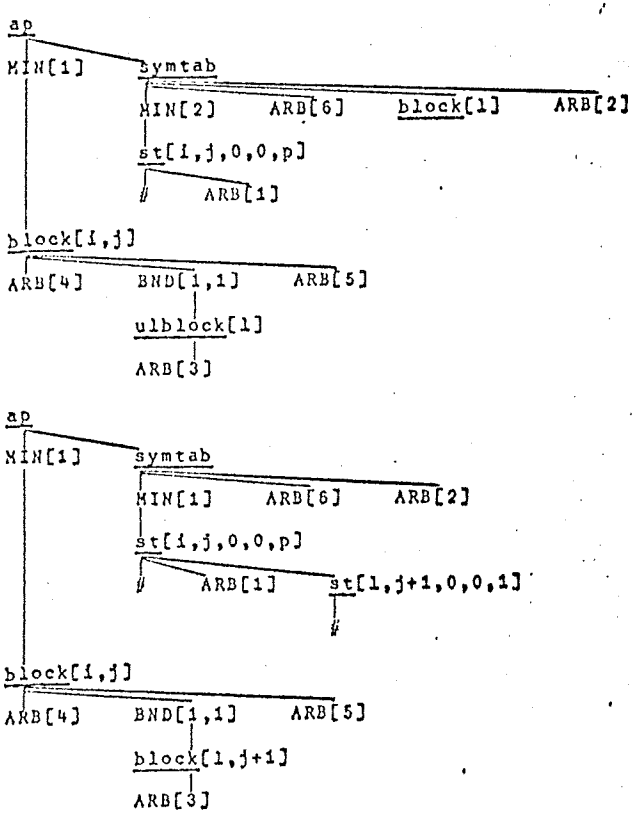
Transformation 155: count program block and insert block symbol table node.



Transformation 156: convert initial program block node to indicate block number and level, convert symbol table block node to indicate number, level, local and own allocation, and number of containing block, and insert marker.

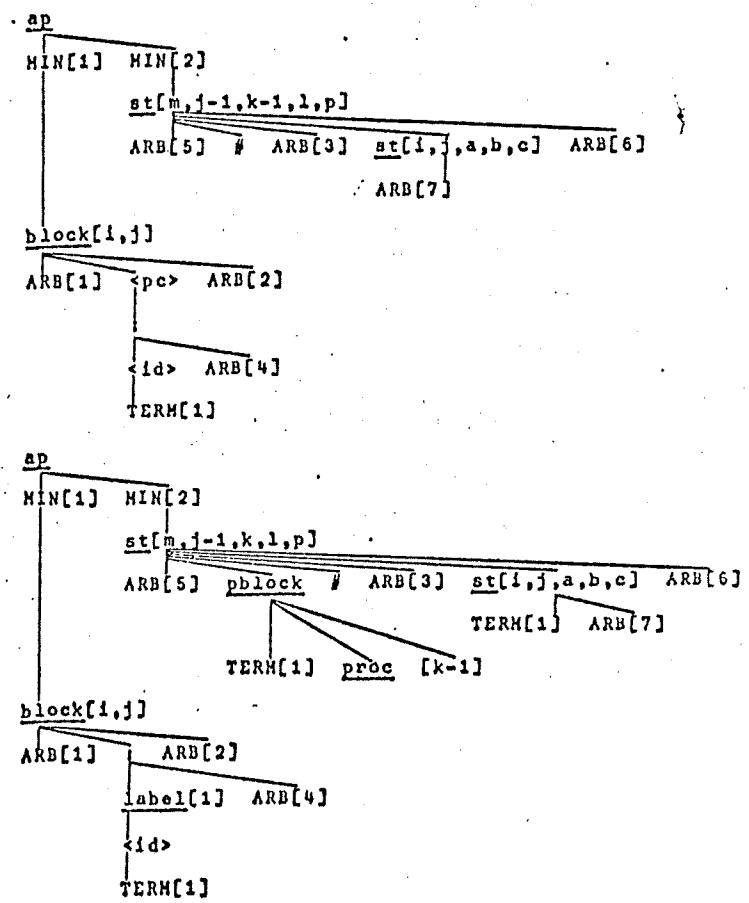


Transformation 157: convert remaining program block nodes to indicate block number and level, convert symbol table block node to indicate number, level, local and own variable allocation, and number of containing block and insert marker.

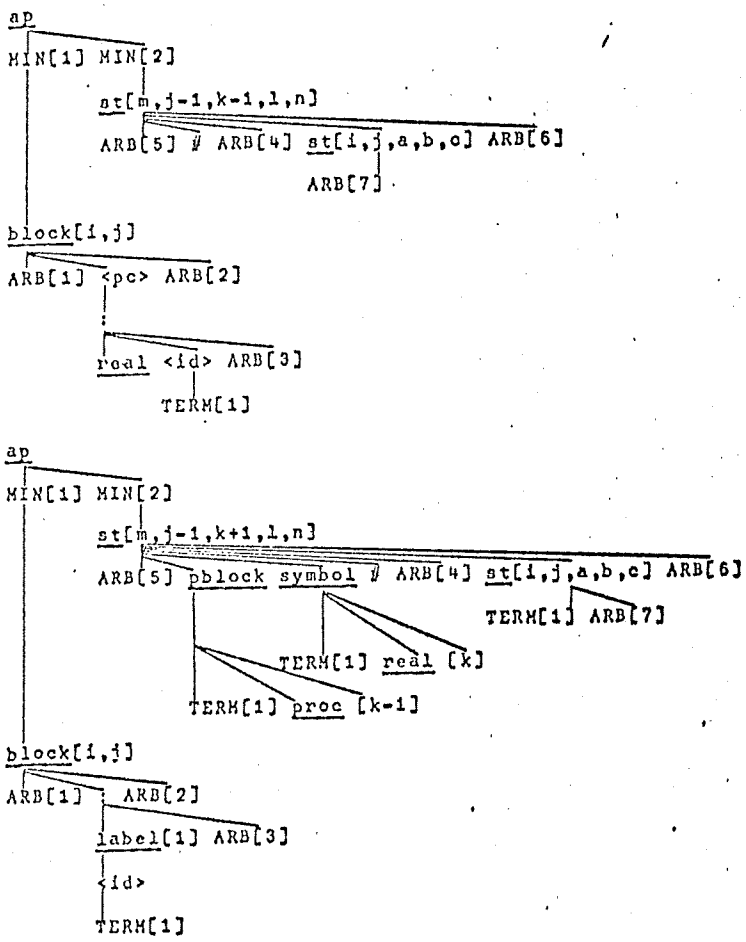


Transformation 158: remove-superfluous node.

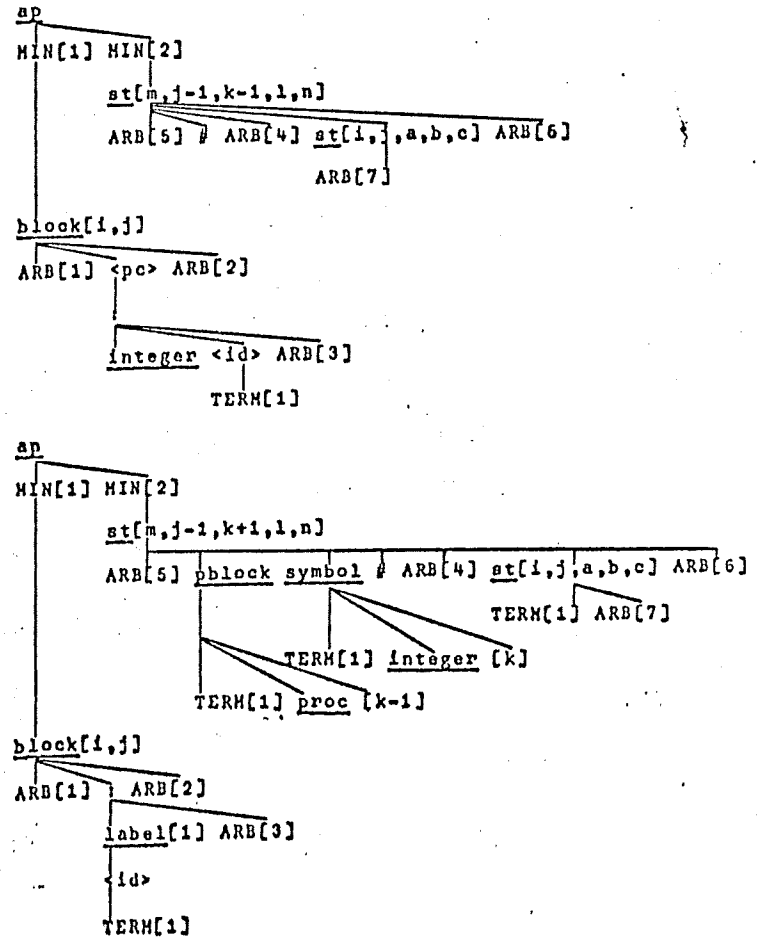
Transformation 159: copy procedure name declaration to symbol table, increment local allocation, and move marker.



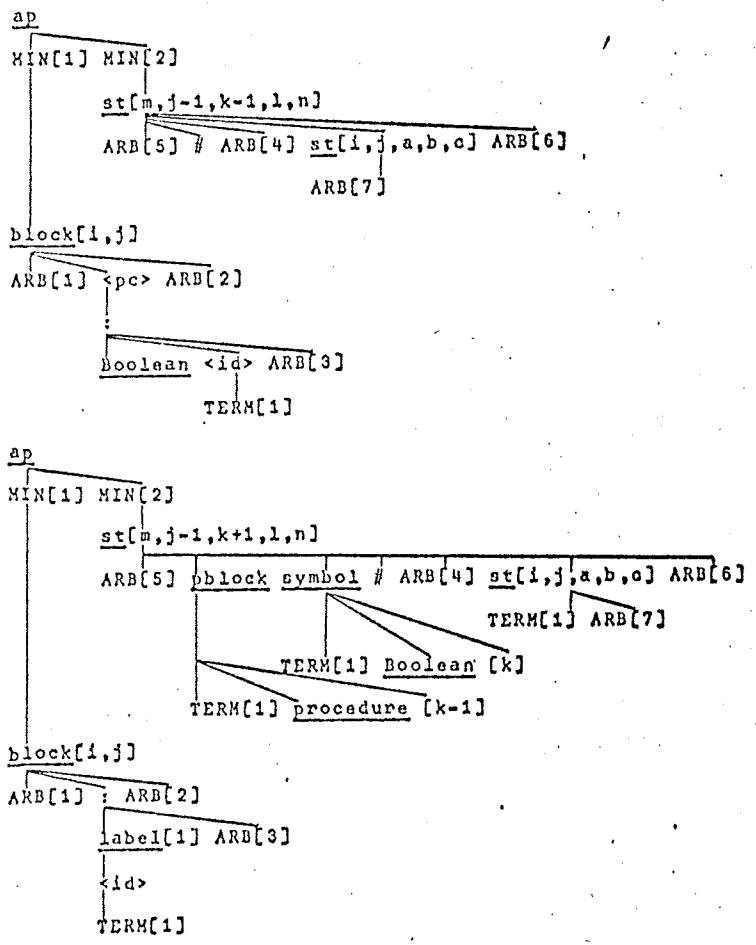
Transformation 160: copy real function procedure name declaration to symbol table, increment local allocation by two, and move marker.



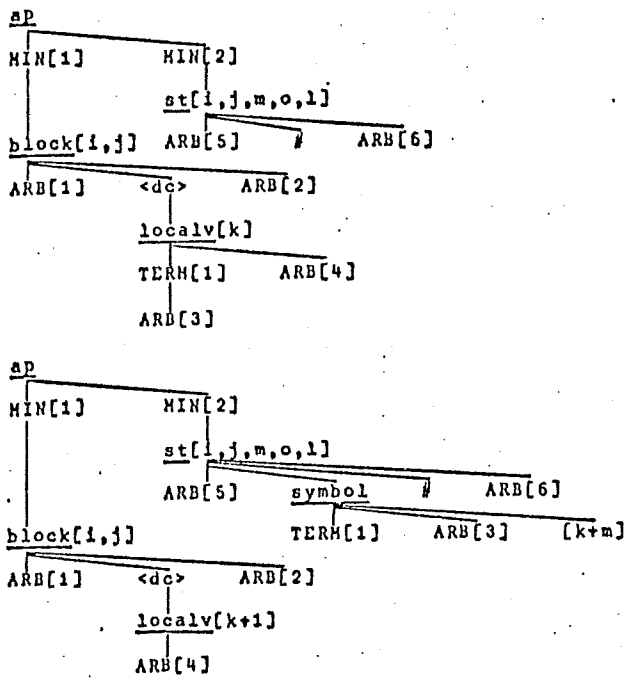
Transformation 161: copy integer function procedure name declaration to symbol table, increment local allocation by two, and move marker.



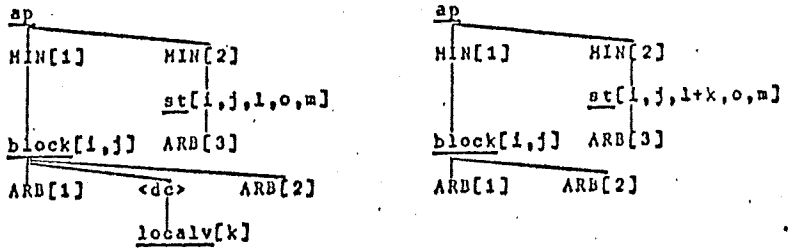
Transformation 162: copy Boolean procedure name declaration to symbol table, increment local allocation by two, and move marker.



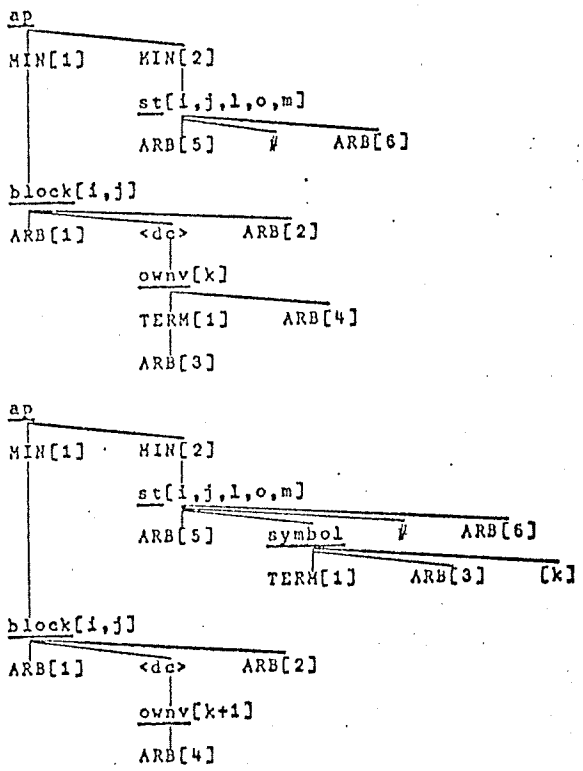
Transformation 163: copy local declaration element to symbol table, increment local allocation, and move marker.



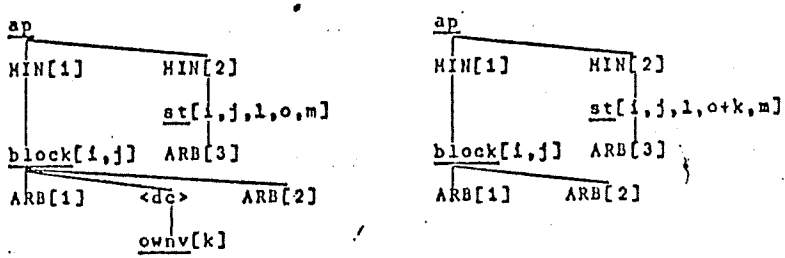
Transformation 164: all local declaration elements for block copied, remove declaration collector.



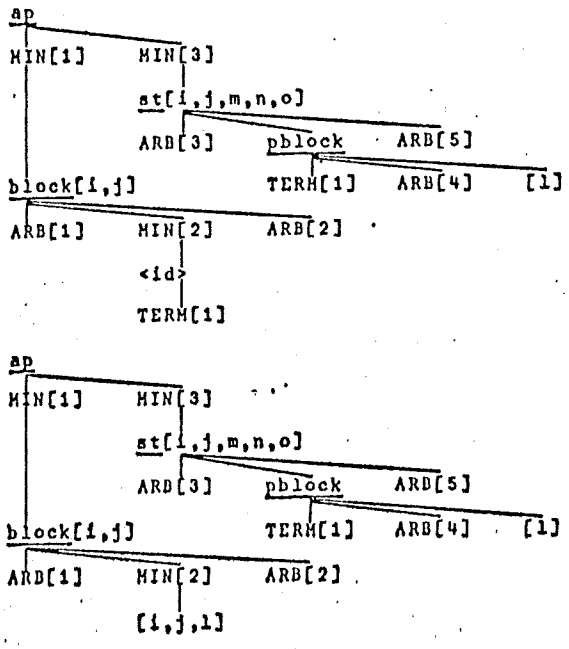
Transformation 165: copy own declaration element to symbol table, increment own allocation, and move marker.



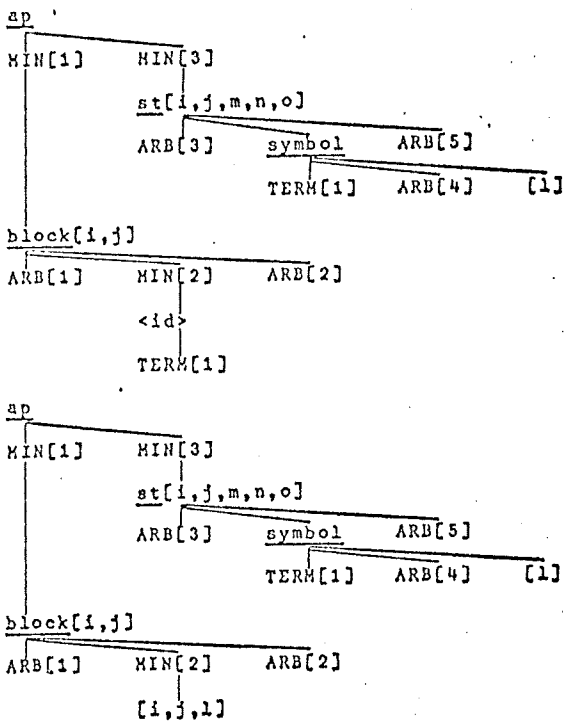
Transformation 166: all own declaration elements for block copied, remove declaration collector.



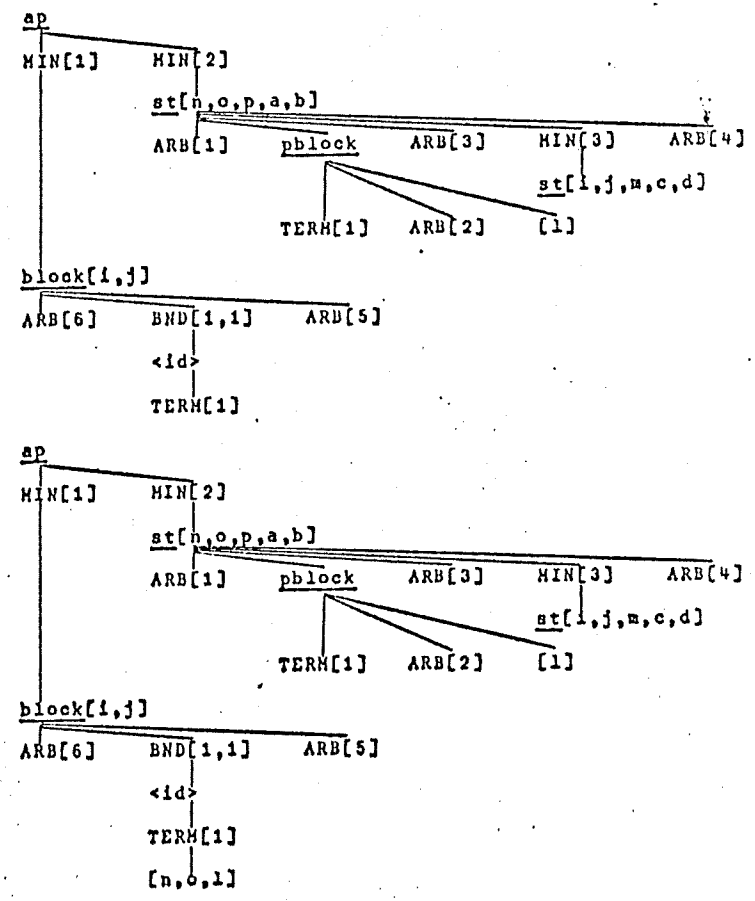
Transformation 167: replace local procedure name by block number, level, and displacement.



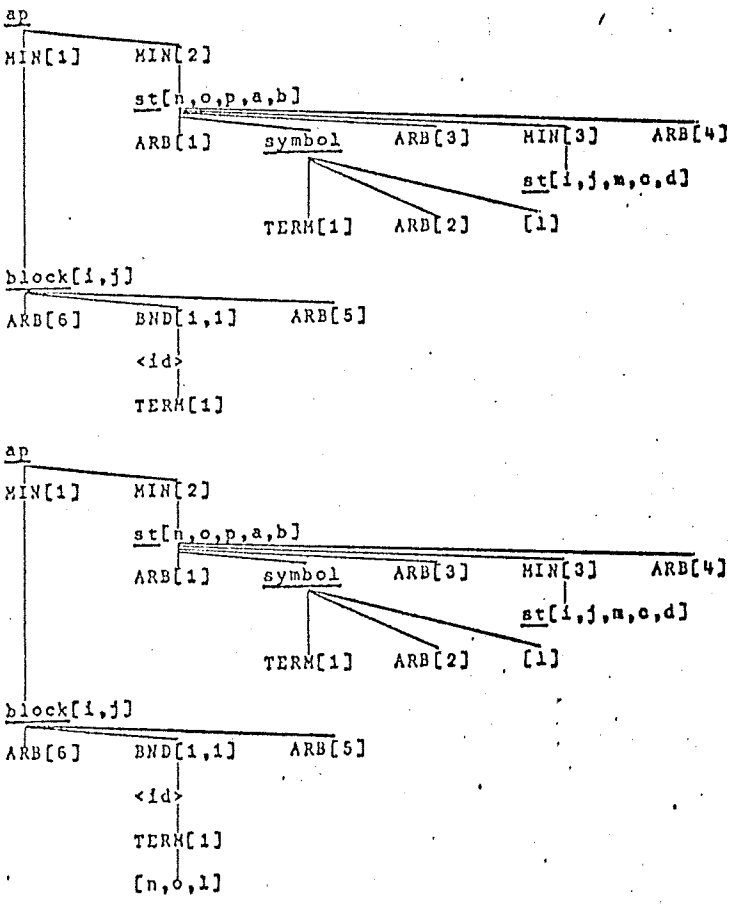
Transformation 168: replace local symbol name by block number, level, and displacement.



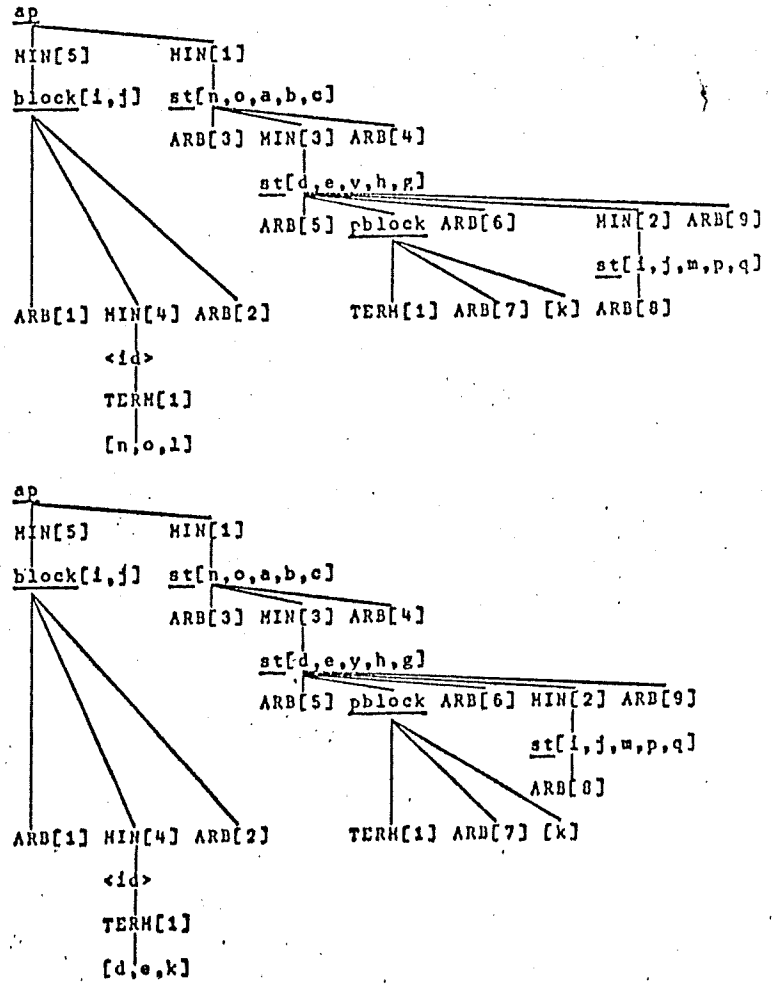
Transformation 169: add to procedure name referenced in block other than defining one the block number, level, and displacement of outermost block containing a declaration of that procedure name.



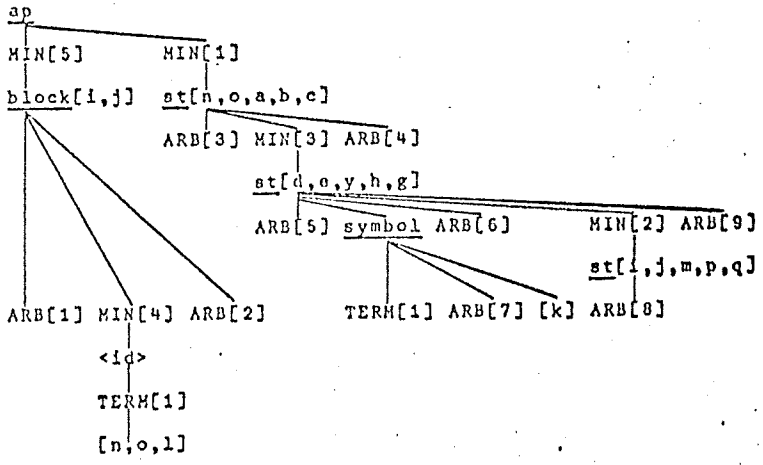
Transformation 170: add to symbol name referenced in block other than defining one the block number, level, and displacement of outermost block containing a declaration of that symbol.

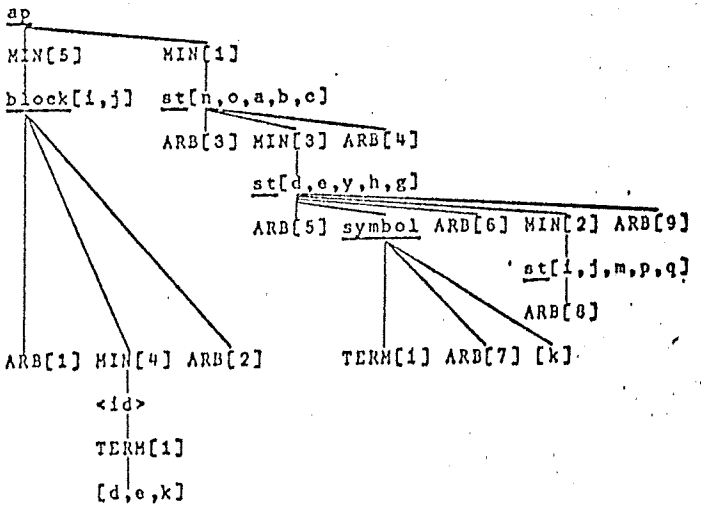


Transformation 171: if there is a matching procedure declaration in a block between the block in which the procedure is referenced and the block whose number, level, and displacement is currently attached to it, replace the attachment by one for the intervening block.



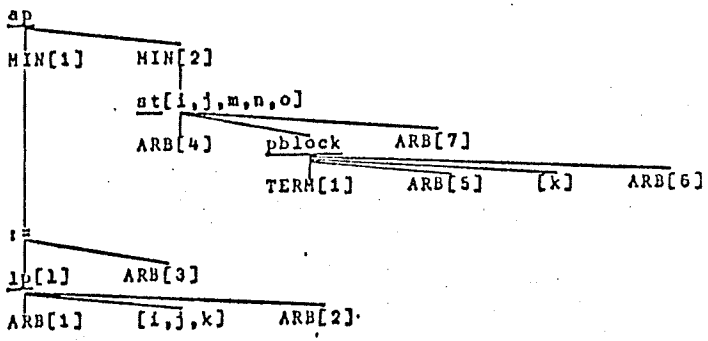
Transformation 172: if there is a matching symbol name declaration in a block between the block in which the symbol is referenced and the block whose number, level, and displacement is currently attached to it, replace the attachment by one for the intervening block.

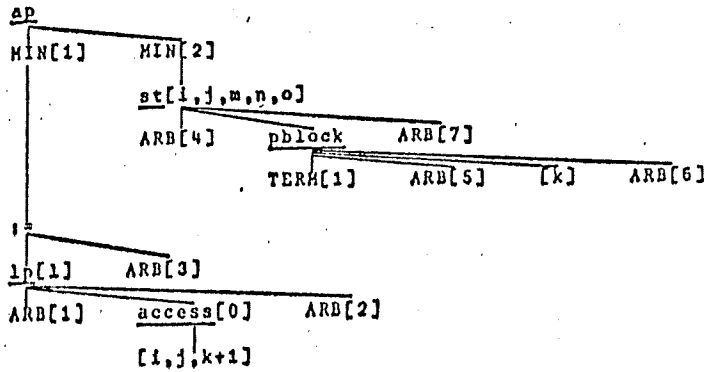




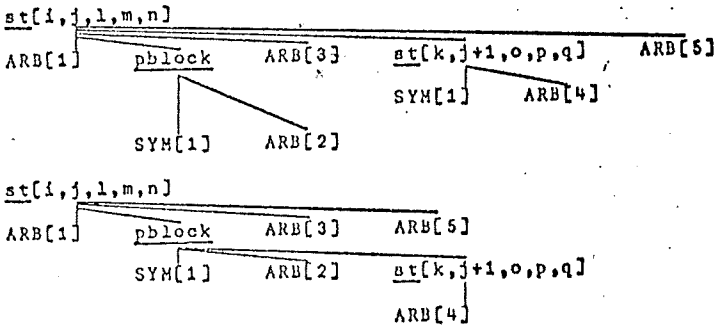
Transformation 173: appropriate association made, delete symbol name.

Transformation 174: change assignment to function procedure name to reference second of two words allocated for it.





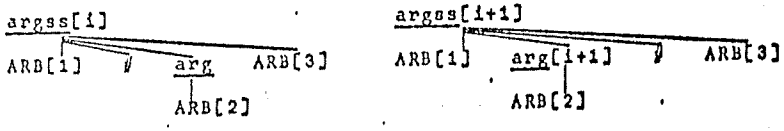
Transformation 175: attach procedure symbol table to procedure name declaration.



Transformation 176: remove all nonterminal nodes.

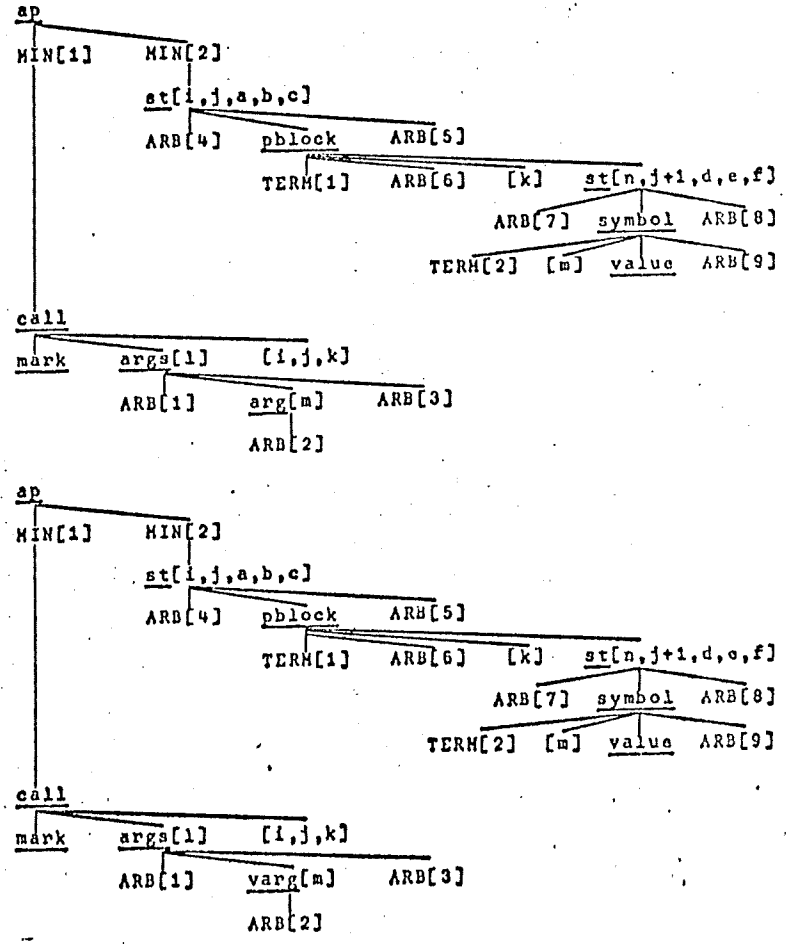
Transformation 177: insert marker in preparation for counting arguments.

Transformation 178: count argument, number particular argument, and move marker.

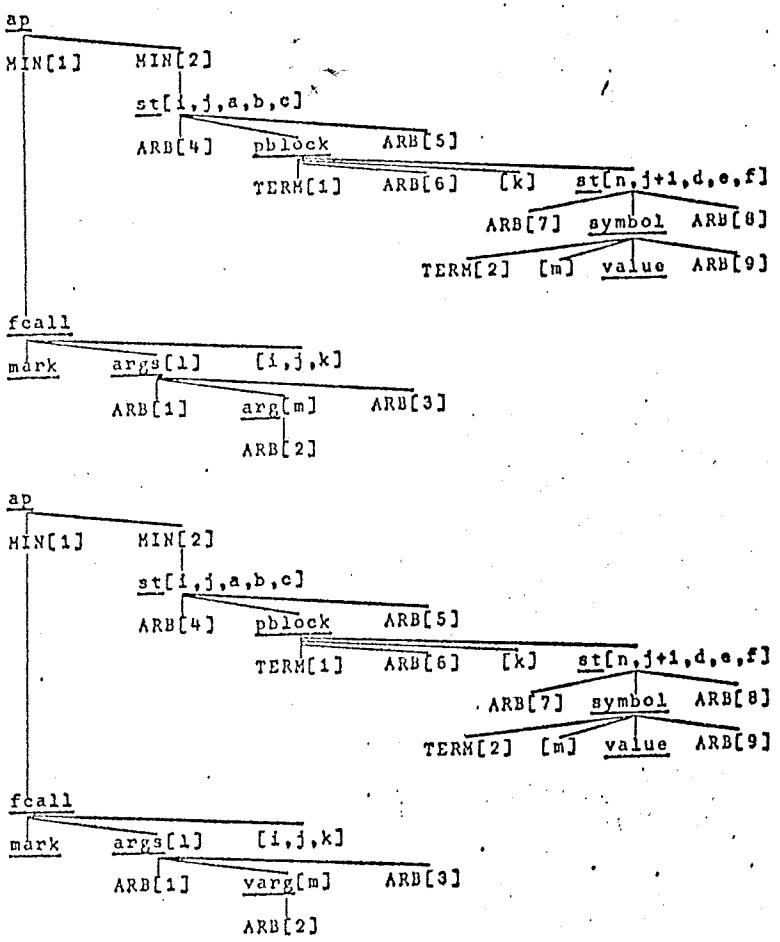


Transformation 179: counting done, remove marker.

Transformation 180: mark value argument.



Transformation 181: mark value argument.

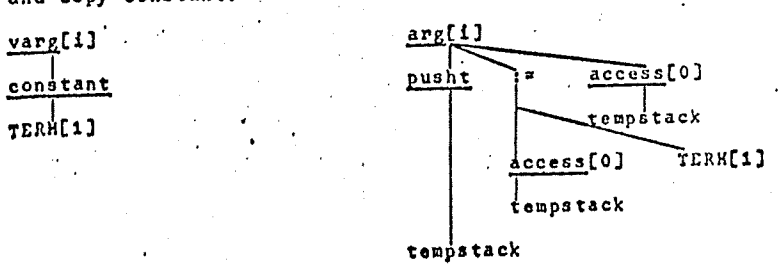


Transformation 182: mark all remaining arguments as name arguments.

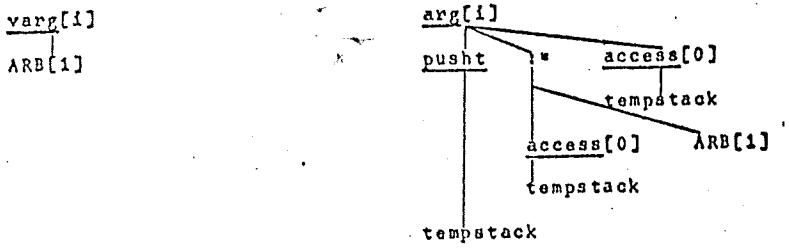
Transformation 183: convert array value arguments to computation tree form.

Transformation 184: convert simple or subscripted variable value arguments to computation tree form.

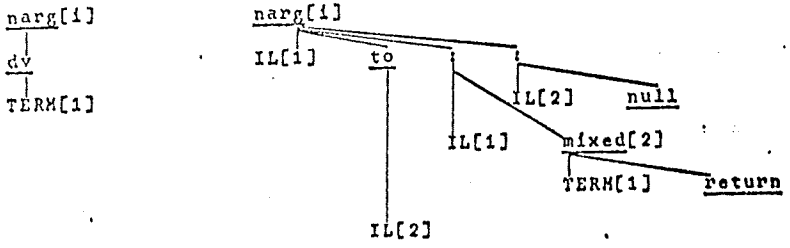
Transformation 185: convert constant value argument to computation tree form, inserting nodes to allocate space for and copy constant.



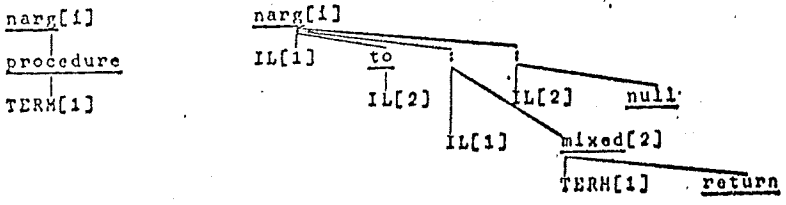
Transformation 186: convert expression value argument to computation tree form, inserting nodes to allocate space for and copy expression value.



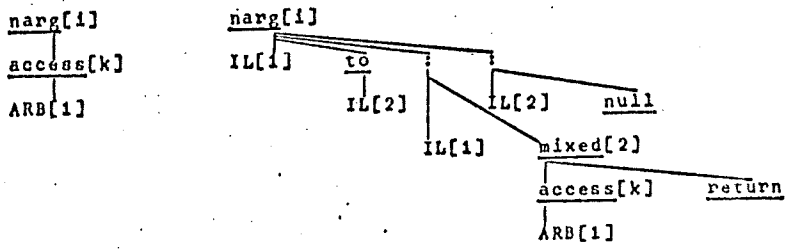
Transformation 187: convert array name argument to computation tree from, inserting nodes to create an evaluation procedure.



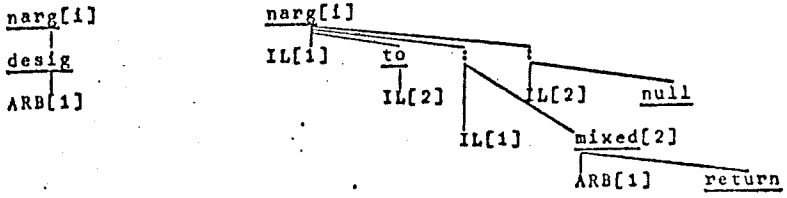
Transformation 188: convert procedure name argument to computation tree form, inserting nodes to create an evaluation procedure.



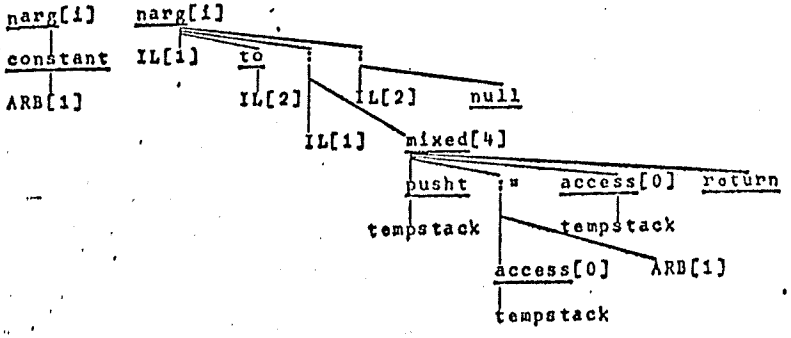
Transformation 189: convert simple or subscripted variable name argument to computation tree form, inserting nodes to create an evaluation procedure.



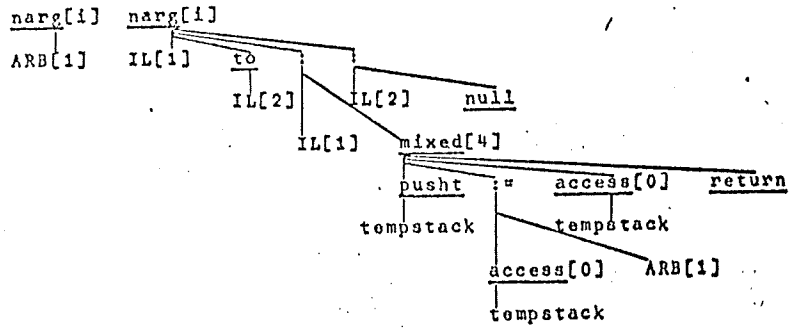
Transformation 190: convert designational expression argument to computation tree form, inserting nodes to create an evaluation procedure.



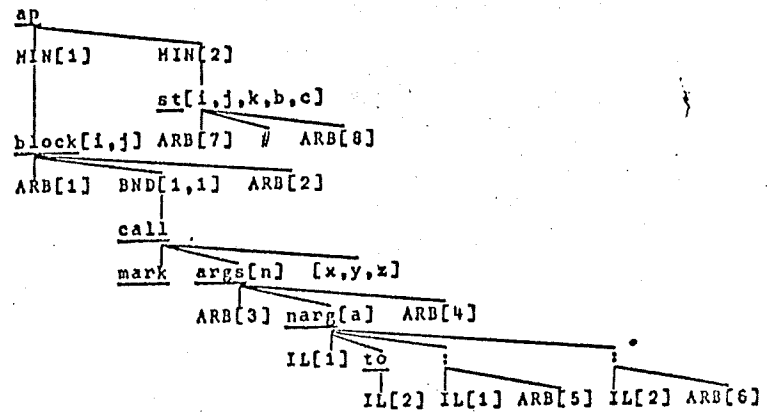
Transformation 191: convert constant name argument to computation tree form, inserting nodes to allocate space and create an evaluation procedure.

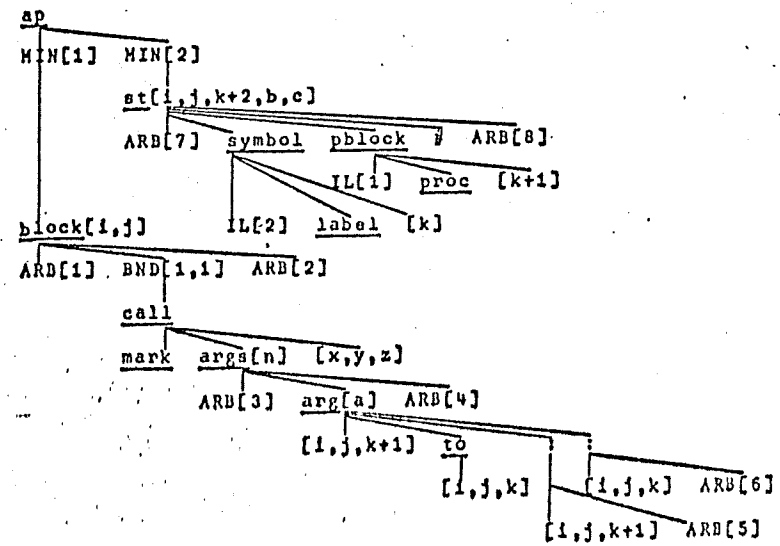


Transformation 192: convert expression name argument to computation tree form, inserting nodes to allocate space and create an evaluation procedure.

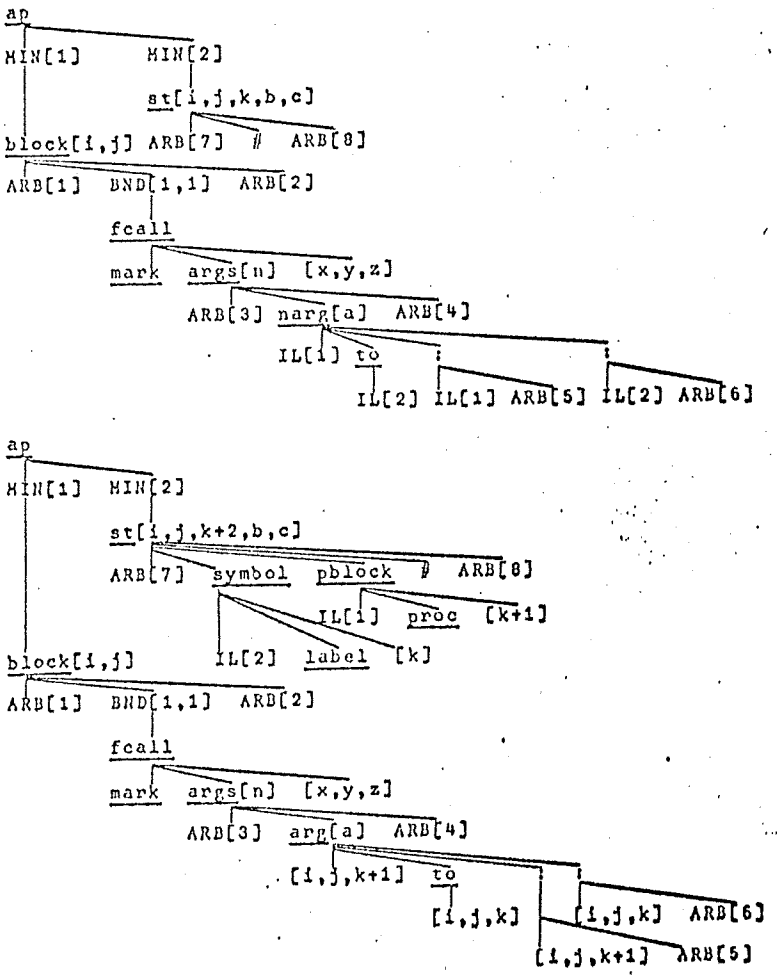


Transformation 193: copy internal labels used for argument evaluation to symbol table, copy evaluation procedure names to symbol table, increment local variable allocation, and replace symbol references by block number, level, and displacement.





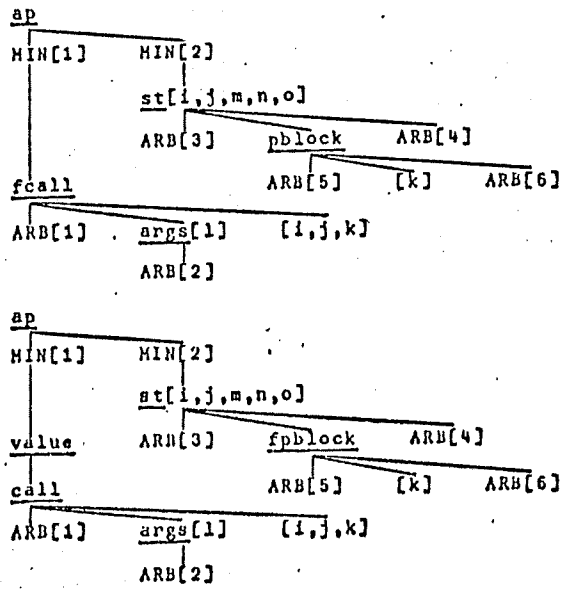
Transformation 194: copy internal labels used for argument evaluation to symbol table, copy evaluation procedure names to symbol table, increment local variable allocation, and replace symbol references by block number, level, and displacement.



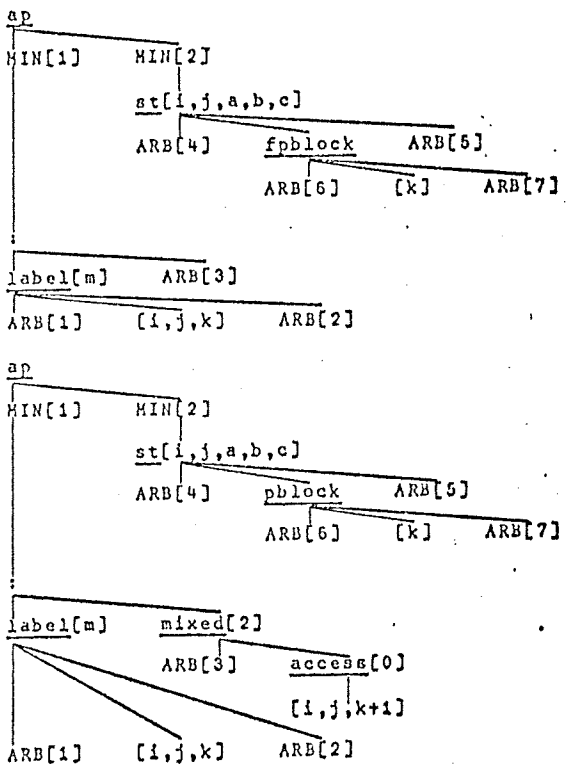
Transformation 195: all symbols replaced by block number, level, and displacement, remove marker.

Transformation 196: insert argument collector for procedure call with no arguments.

Transformation 197: convert function procedure call to obtain value of address returned from call.

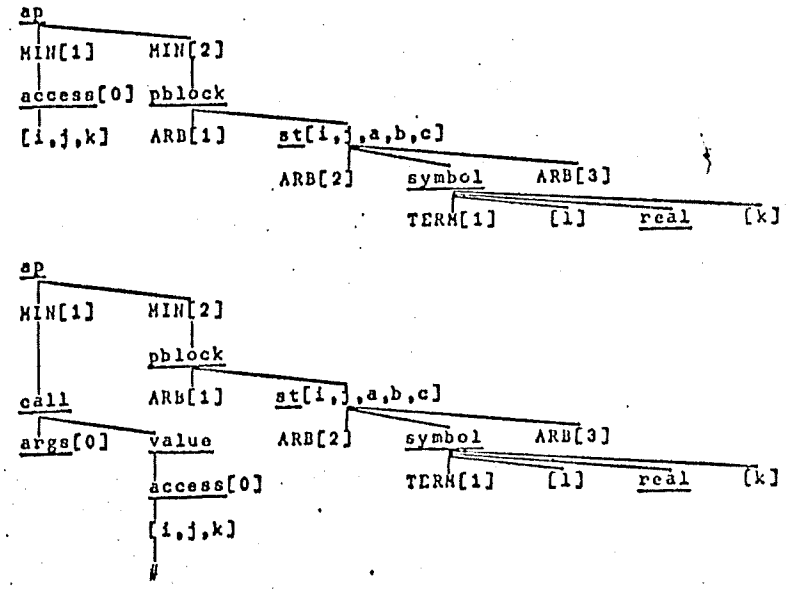


Transformation 198: insert nodes to cause function procedure to return address.

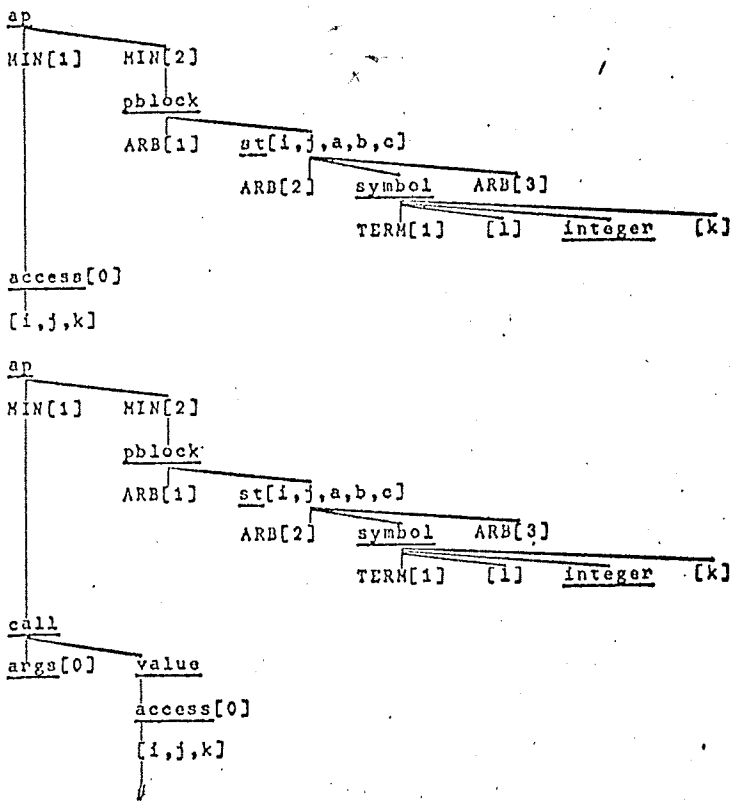


Transformation 199: insert nodes to cause function procedure to return address.

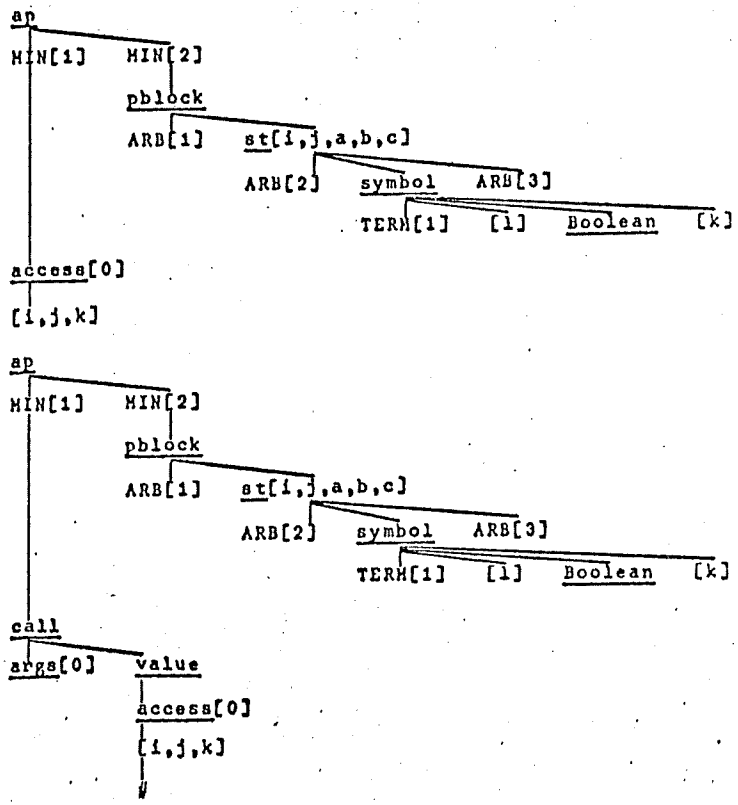
Transformation 200: insert call of evaluation procedure for real variable name argument in referencing procedure.



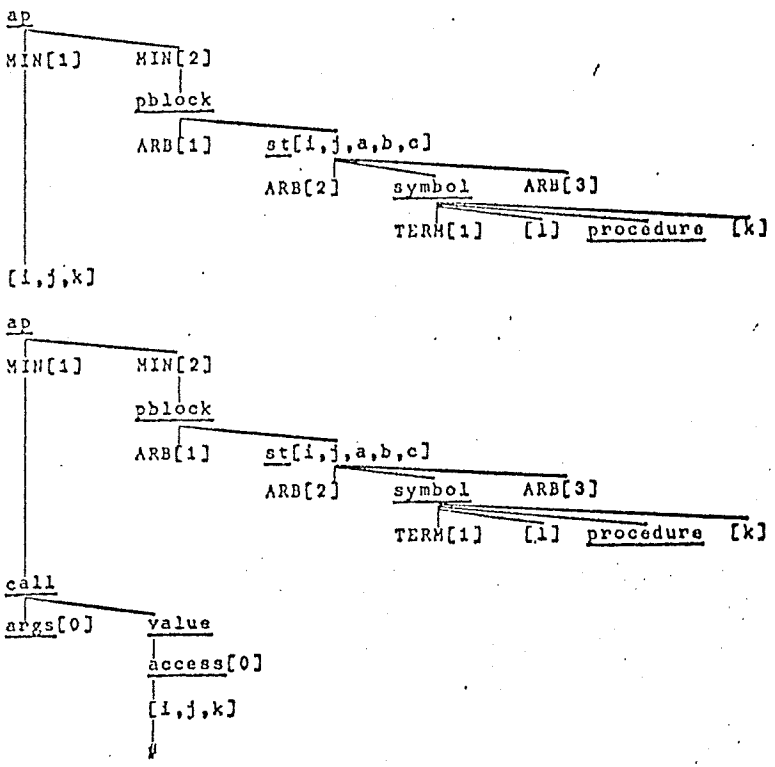
Transformation 201: insert call of evaluation procedure for integer variable name argument in referencing procedure.



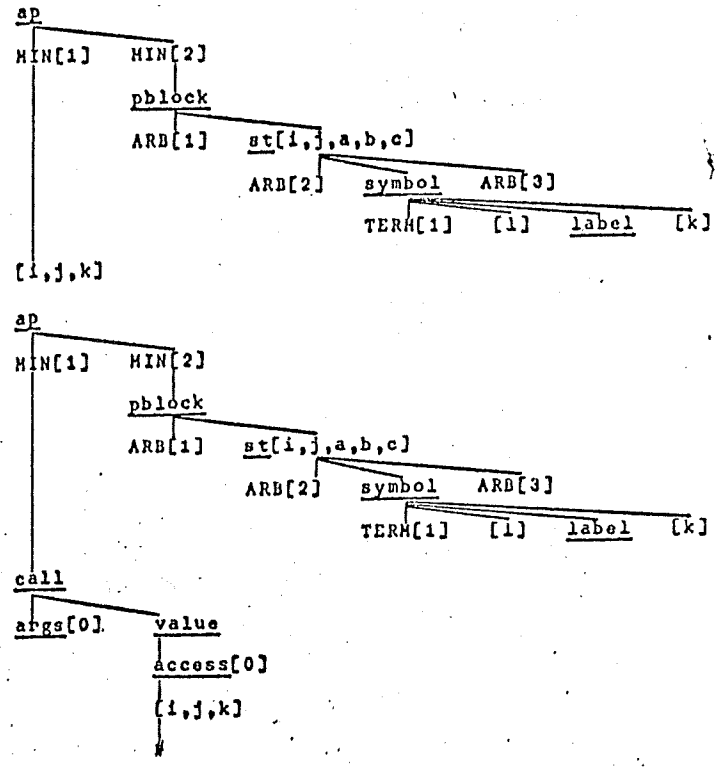
Transformation 202: insert call of evaluation procedure for Boolean variable name argument in referencing procedure.



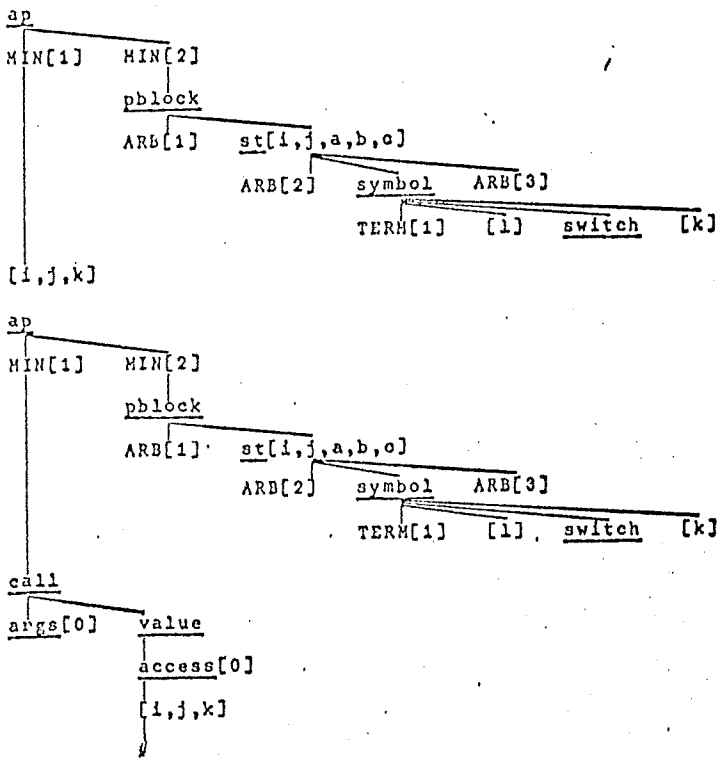
Transformation 203: insert call of evaluation procedure for procedure variable name argument in referencing procedure.



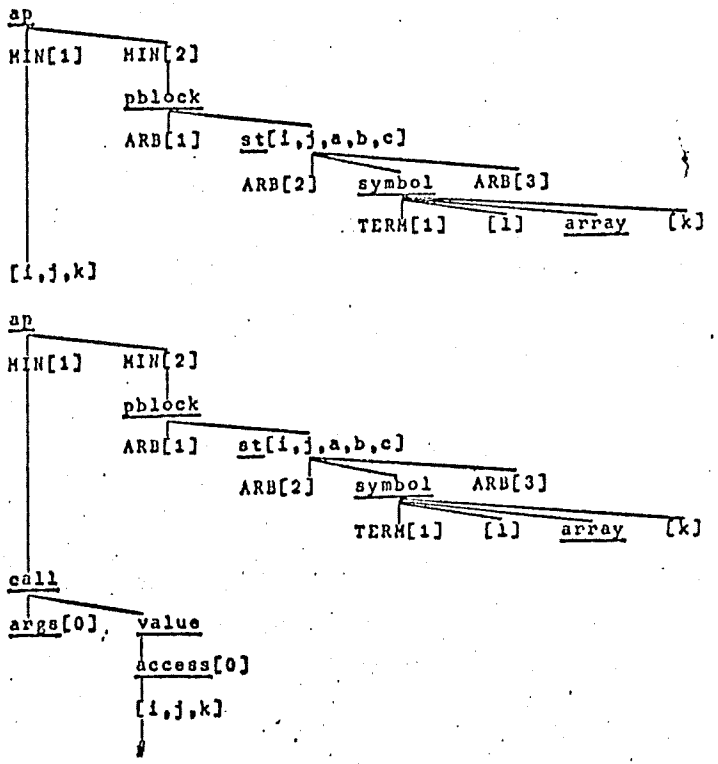
Transformation 204: insert call of evaluation procedure for label variable name argument in referencing procedure.



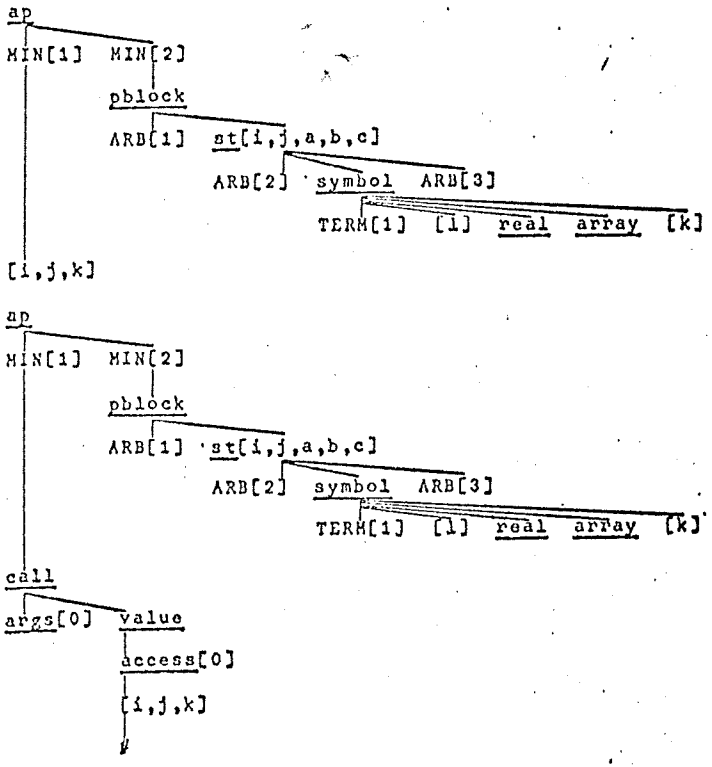
Transformation 205: insert call of evaluation procedure for switch variable name argument in referencing procedure.



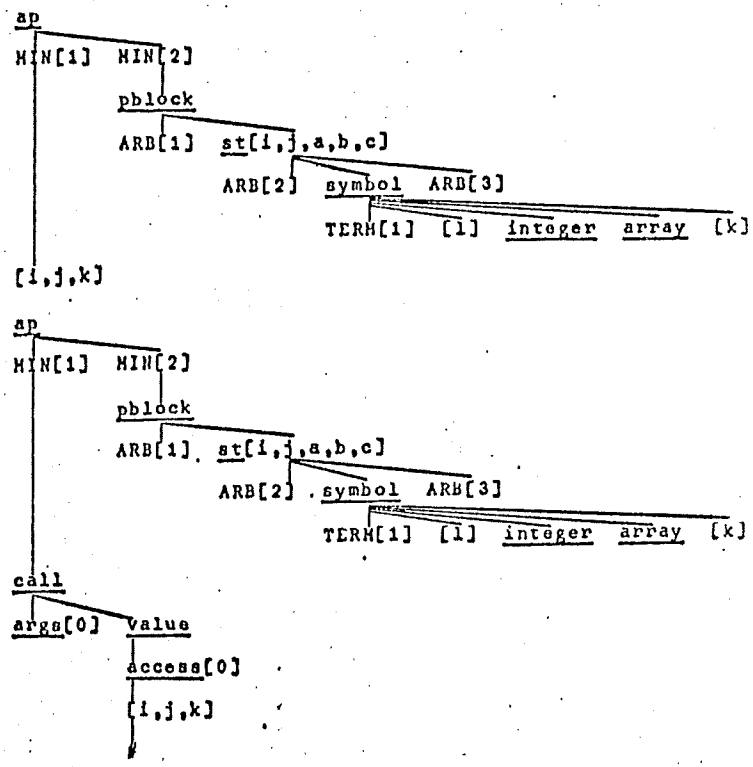
Transformation 206: insert call of evaluation procedure for array variable name argument in referencing procedure.



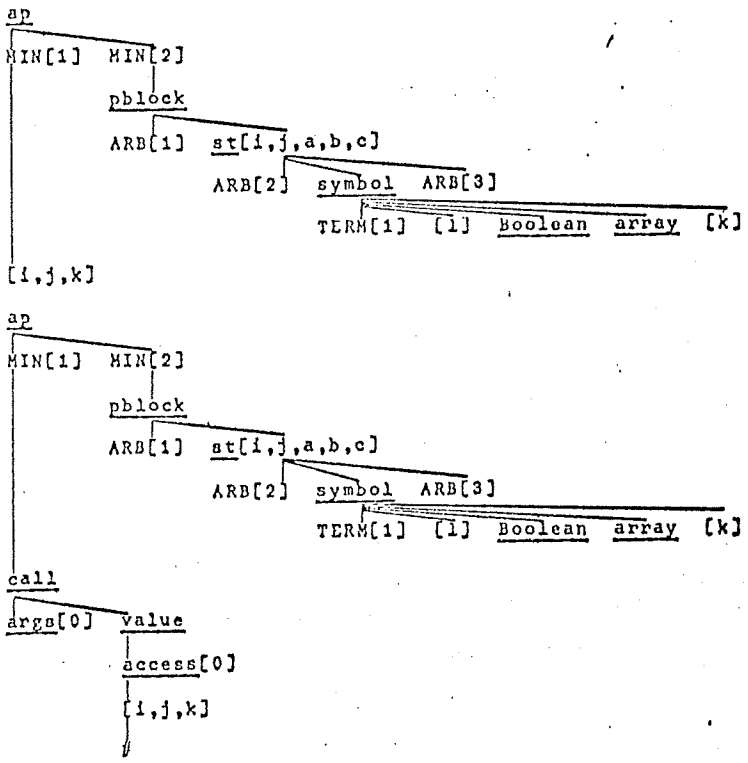
Transformation 207: insert call of evaluation procedure for real array variable name argument in referencing procedure.



Transformation 208: insert call of evaluation procedure for integer array variable name argument in referencing procedure.



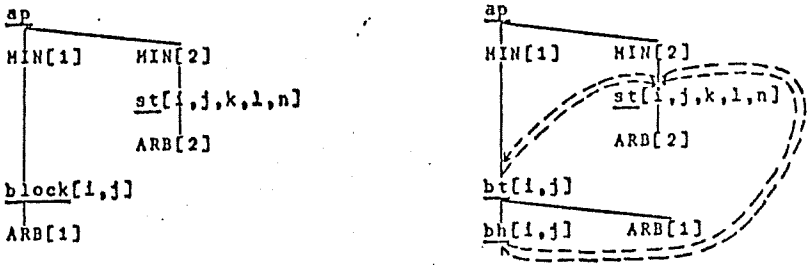
Transformation 209: insert call of evaluation procedure for Boolean array variable name argument in referencing procedure.



Transformation 210: all references marked, remove marker.

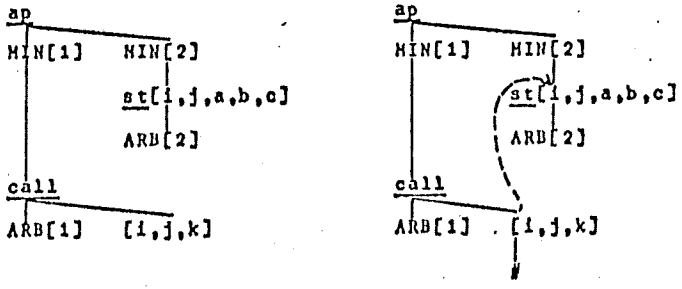
Transformation 211: insert label count node.

Transformation 212: insert program block head and tail nodes and thread to and from block symbol table.



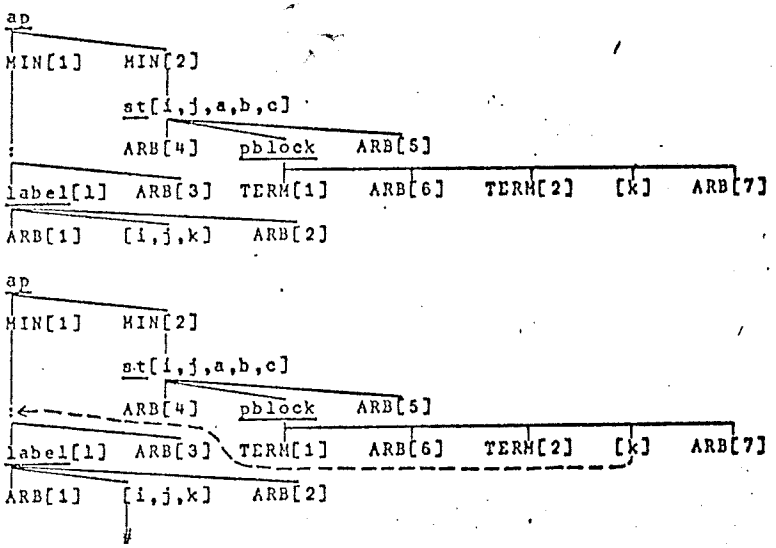
Threads:
thread from bt[i,j] to st[i,j,k,l,n]
thread from st[i,j,k,l,n] to bt[i,j]
thread from bh[i,j] to st[i,j,k,l,n]
thread from st[i,j,k,l,n] to bh[i,j]

Transformation 213: thread procedure call to symbol table for block containing procedure declaration.



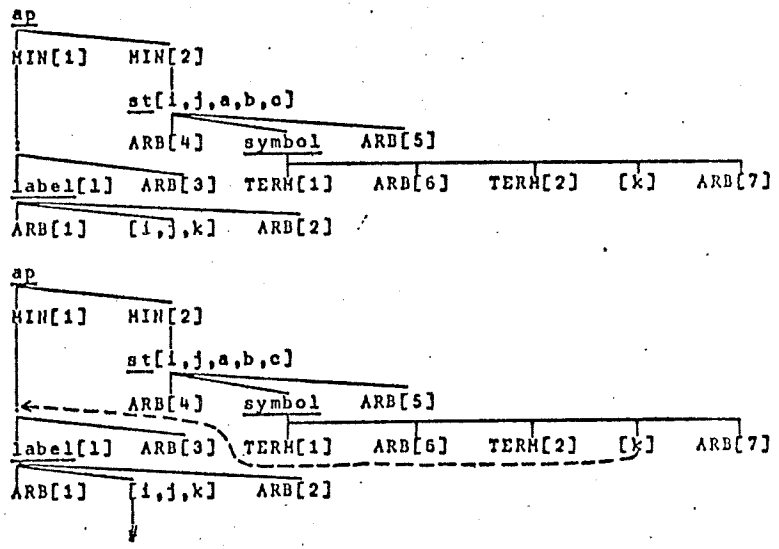
Threads:
thread from [i,j,k] to st[i,j,a,b,c]

Transformation 214: thread procedure name declaration to procedure name as label on procedure body.



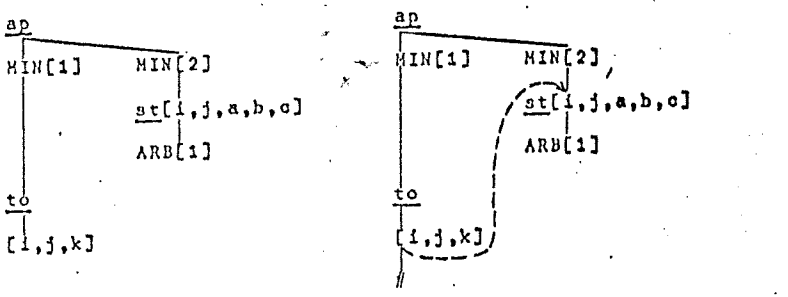
Threads:
thread from [k] to :

Transformation 215: thread label declaration to label usage on statement.



Threads:
thread from [k] to :

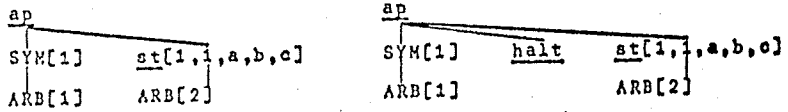
Transformation 216: thread label reference to symbol table block in which the label is declared.



Threads:
thread from [i,j,k] to st[i,j,a,b,c]

Transformation 217: insert label count node.

Transformation 218: insert halt between program branch and symbol table branch of computation tree.



Appendix II

AN EXAMPLE OF THE APPLICATION OF SEMANTIC TRANSFORMATIONS TO AN ALGOL 60 PROGRAM

This appendix illustrates the application of semantic transformations to the parse tree for the following Algol 60 program:

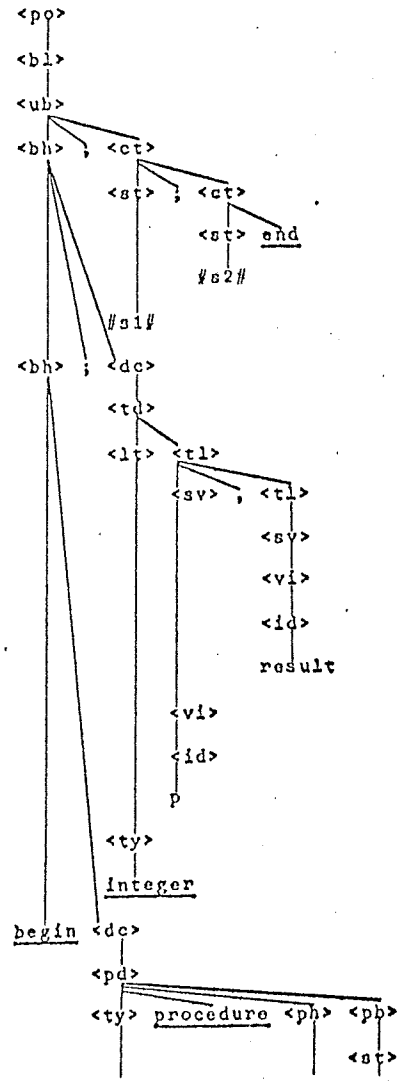
```
begin
   integer procedure factorial(n); value n; integer n;
   if n≤1 then factorial:=1
          else factorial:=n x factorial(n-1);
   integer p, result;
   p:=6;
   result:=factorial(p)
end
```

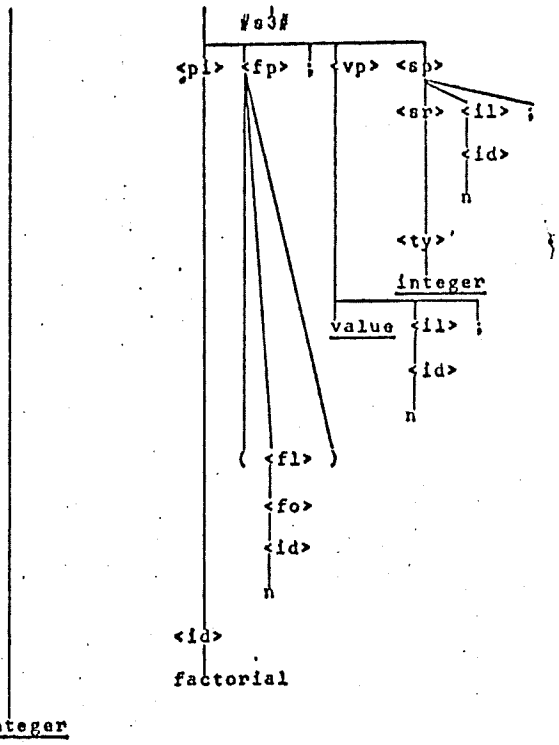
The parse tree for this program has been partitioned into six separate trees and the transformations through 151 are applied to each of these trees separately. Then the resulting trees are recombined into a single tree and the remaining transformations are applied to that tree.

The transformed trees in this appendix are the output of the program in Appendix III. (A few trees have been rearranged by hand to accomodate page width restrictions.) Trees are shown for the successful application of groups of transformations, rather than for each transformation or for just the final transformation.

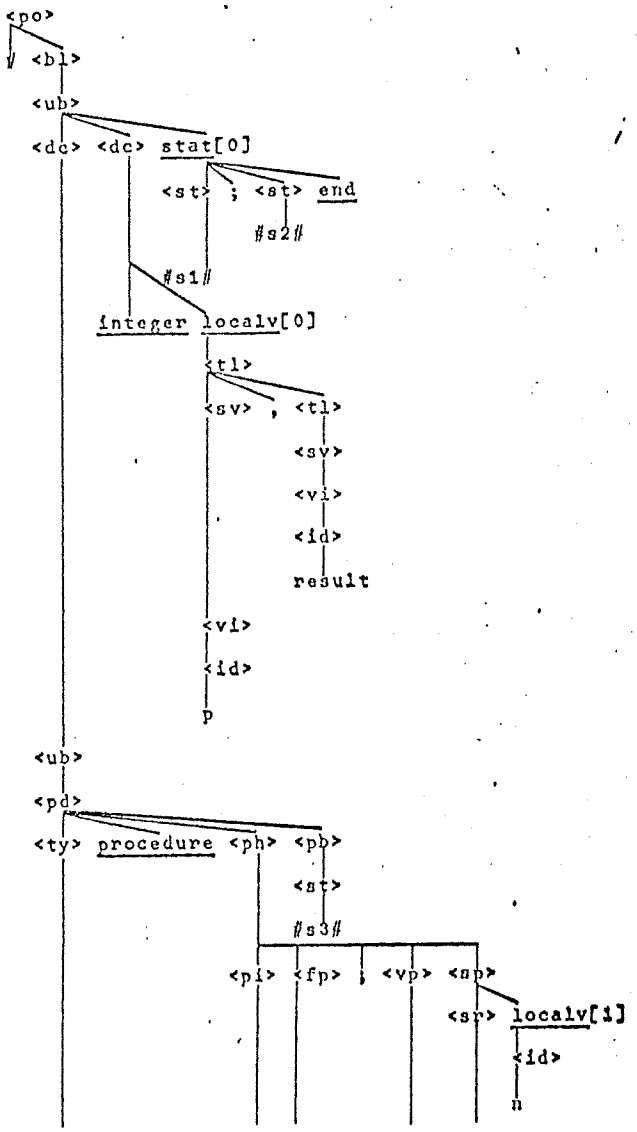
The parse tree for the main body of the program is given below. In that tree #s1#, #s2#, and #s3# represent

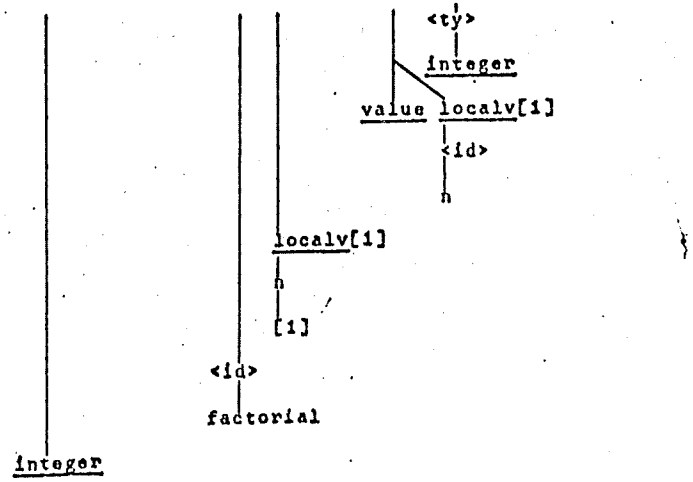
statements, each of which is treated later.



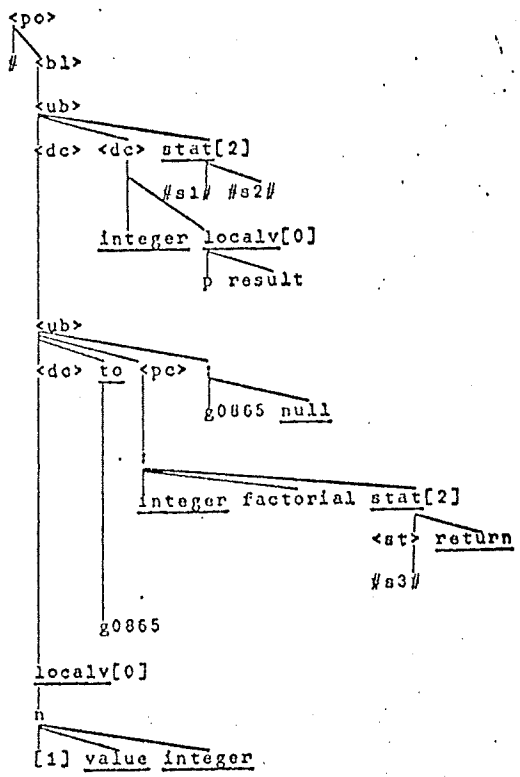


The successful application of transformation 1, 3 (twice), 4, 5 (twice), 11, 16, 18, 20, 21, 22, 23, and 24 collects and counts statements, collects local variables, creates a block for a procedure declaration, collects, counts, and associates displacements with formal parameters, and inserts count nodes for specification and value variables. The resulting tree is given below.

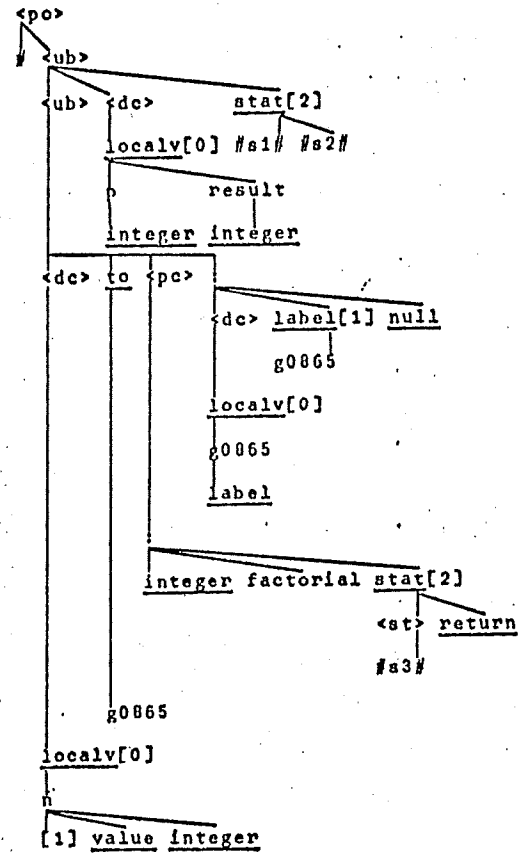




The successful application of transformations 26 (twice), 29, 30, 31, 32, 33, 34, 37, 38, 40, 42, 43, 52, 56 (twice), 57, 70, and 71 collects and counts specification and value variables, copies specification and value attributes to formal parameters and removes specification and value parts, creates a declaration for formal parameters local to the procedure block, labels the procedure with its name, inserts a jump around the procedure, inserts a return statement, and counts statements. The resulting tree is given below.

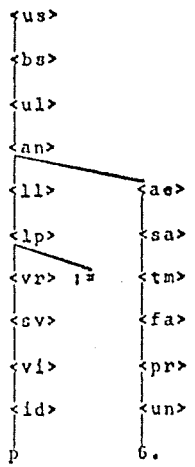


The successful application of transformations 129, 131, 135 (twice), 137 (three times), 139 (twice), and 151 (twice) counts labels, inserts declarations for labels, copies type attributes to local variables, and removes superfluous nodes. The resulting tree is given below.

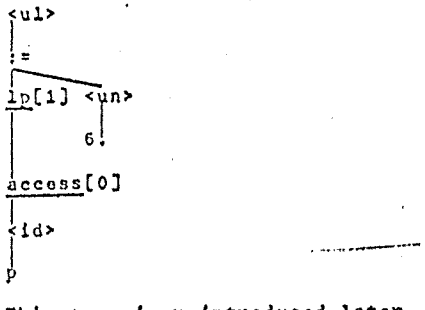


The transformations 1-151 have now been applied. The resulting tree is reintroduced later.

The next parse tree, given below, is for an assignment statement #s1#.

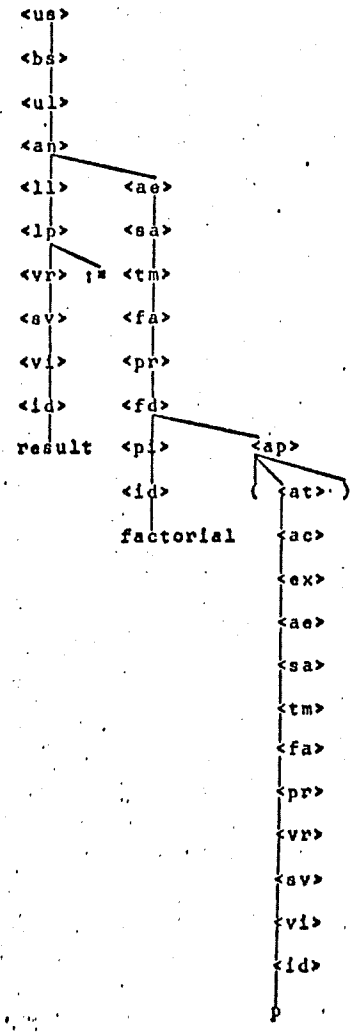


The successful application of transformations 84, 85, 86, 87, and 151 (ten times) collects and counts left parts in an assignment statement, rearranges the format of the statement, inserts an address fetch for a left part variable, and removes superfluous nodes. The resulting tree is given below.

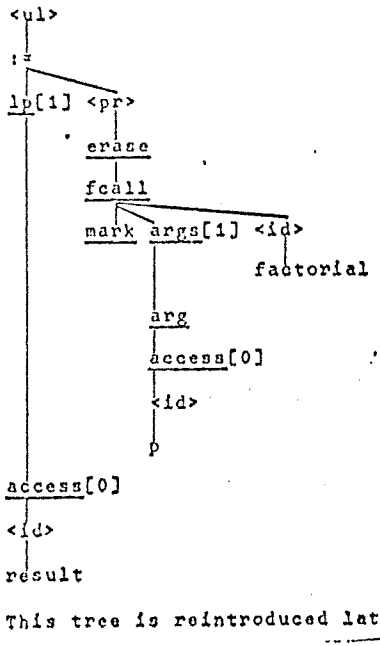


This tree is reintroduced later.

The next parse tree, given below, is for an assignment statement #s2#.

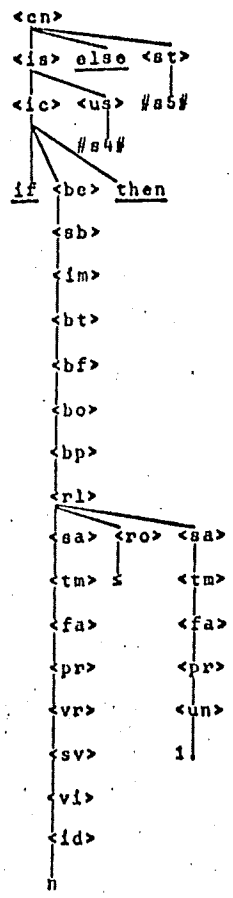


The successful application of transformations 84, 85, 86, 87 (twice), 88, 95, 99, 109, 112, and 151 (eighteen times) collects and counts left parts in an assignment statement, inserts value and address fetch operators for variables, collects, counts, and marks actual parameters in a function designator, rearranges the format of a function designator, and removes superfluous nodes. The resulting tree is given below.



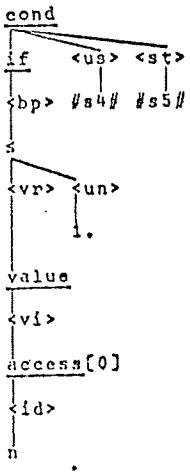
This tree is reintroduced later.

The next parse tree, given below, is for the conditional statement #s3#. The symbols #s4# and #s5# represent two statements which are treated later.



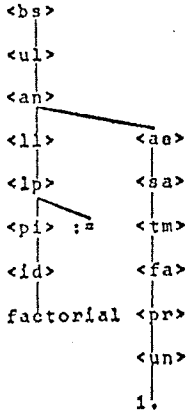
The successful application of transformations 72, 87, 88, 120, 126, and 151 (sixteen times) rearranges the format of a conditional statement, inserts address and value fetch operators for a variable, rearranges the format of an arithmetic expression, and removes superfluous nodes. The

resulting tree is given below.

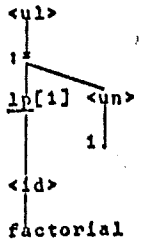


This tree is reintroduced later.

The next parse tree, given below, is for the assignment statement #s4#.

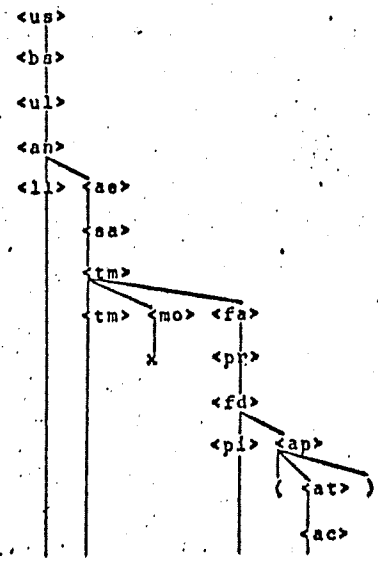


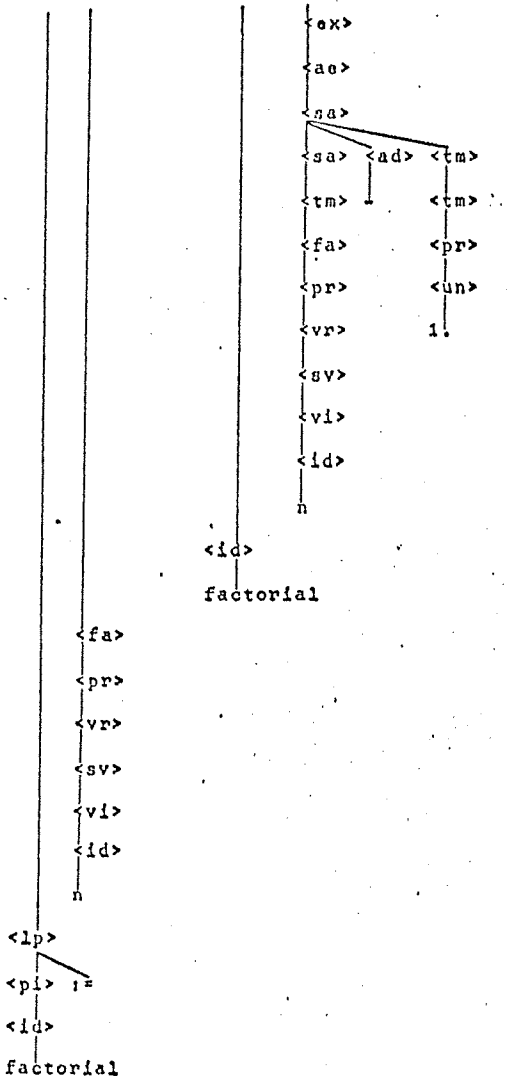
The successful application of transformations 84, 85, 86, and 151 (seven times) collects and counts left parts in an assignment statement and removes superfluous nodes. The resulting tree is given below.



This tree is reintroduced later.

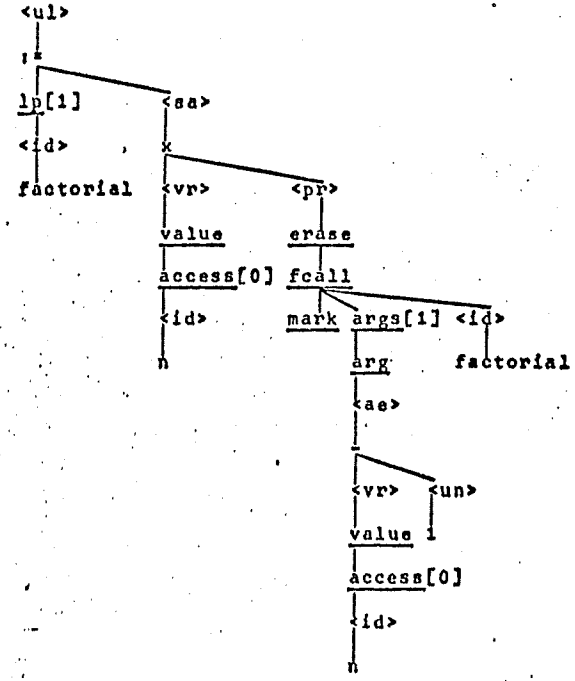
The last of the separate parse trees, given below, is for the assignment statement #s5#.





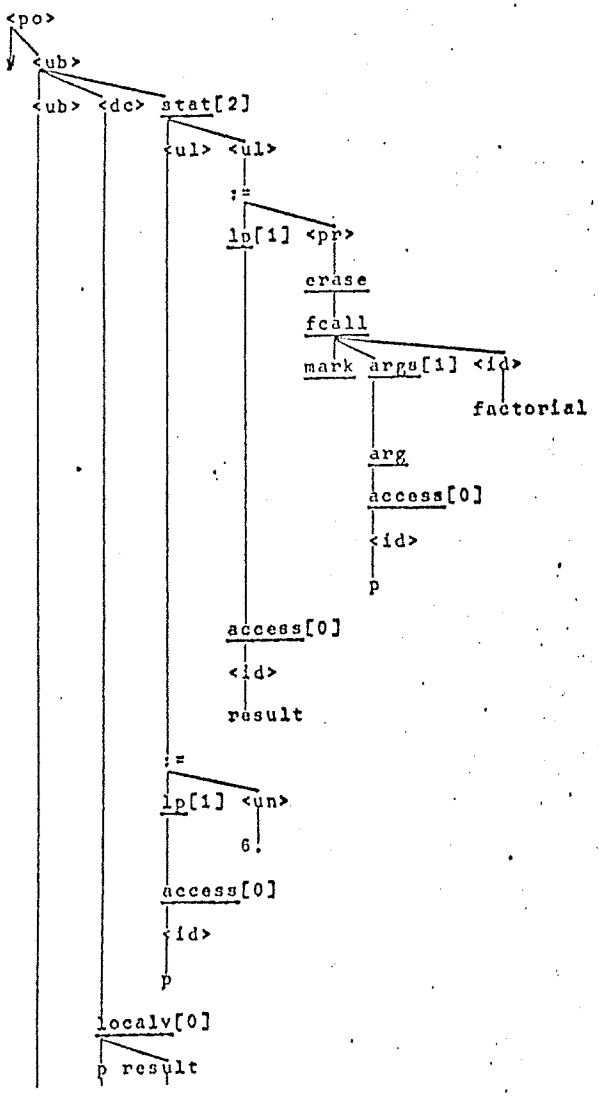
The successful application of transformations 84, 85,

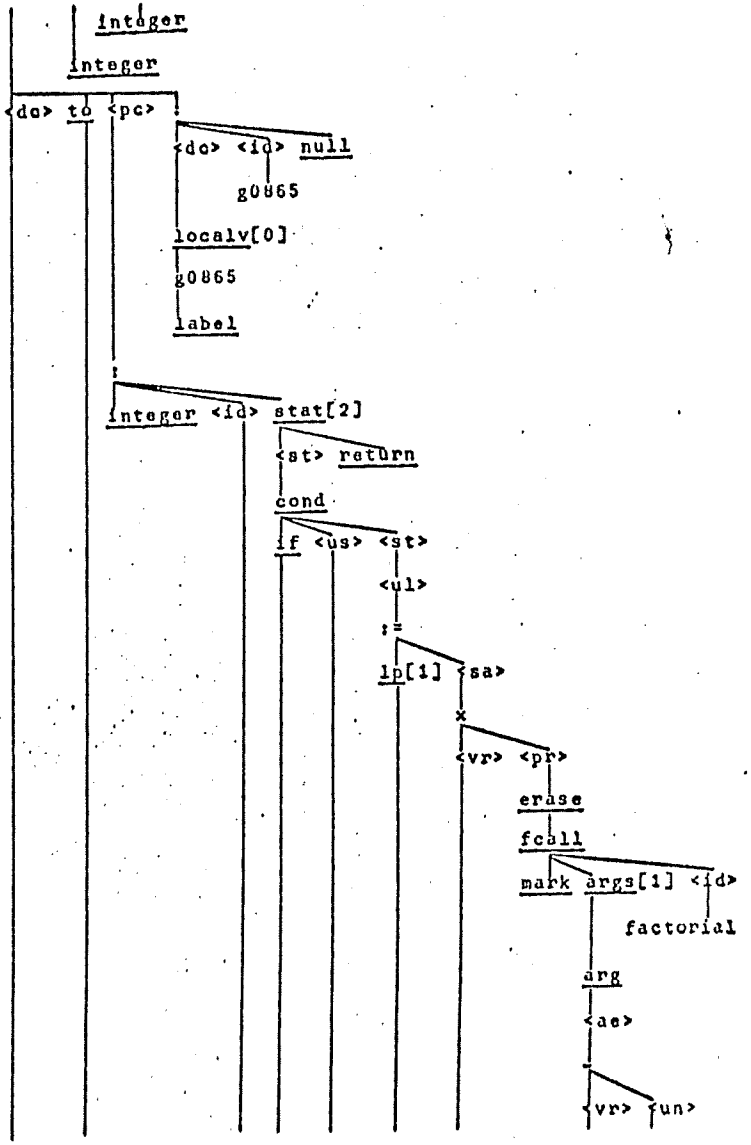
86, 87 (twice), 88 (twice), 95, 99, 109, 117, 118, 124, 125, and 151 (twenty-one times) collects and counts left parts in an assignment statement, inserts address and value fetch operators for variables, collects, counts, and marks actual parameters in a function designator, rearranges the format of a function designator, rearranges the format of an arithmetic expression, and removes superfluous nodes. The resulting tree is given below.

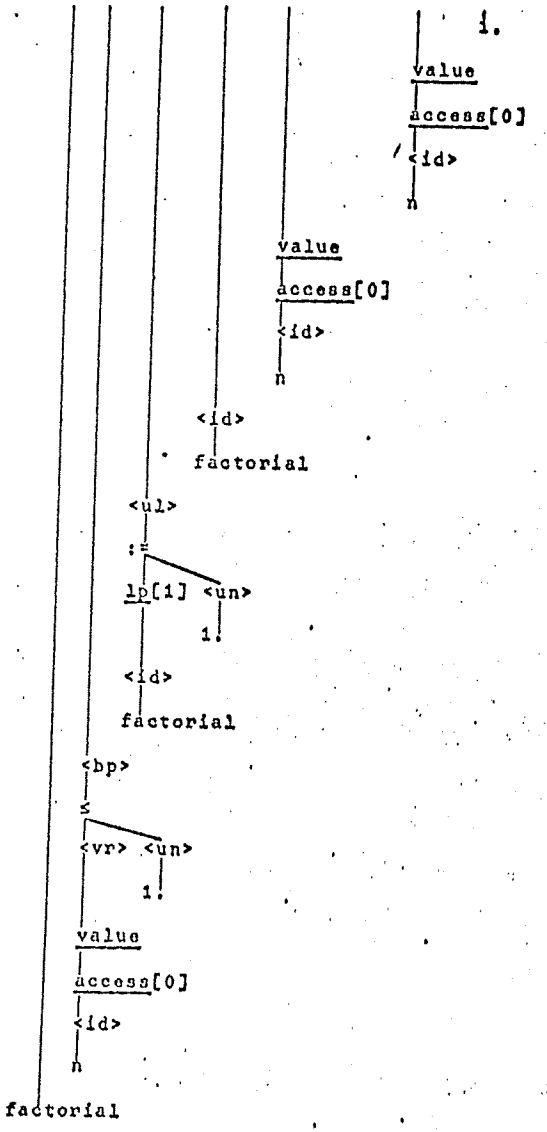


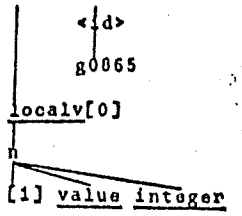
The transformations 1-151 have now been applied to the separate parse trees. The tree given below is the

recombination of those transformed trees.

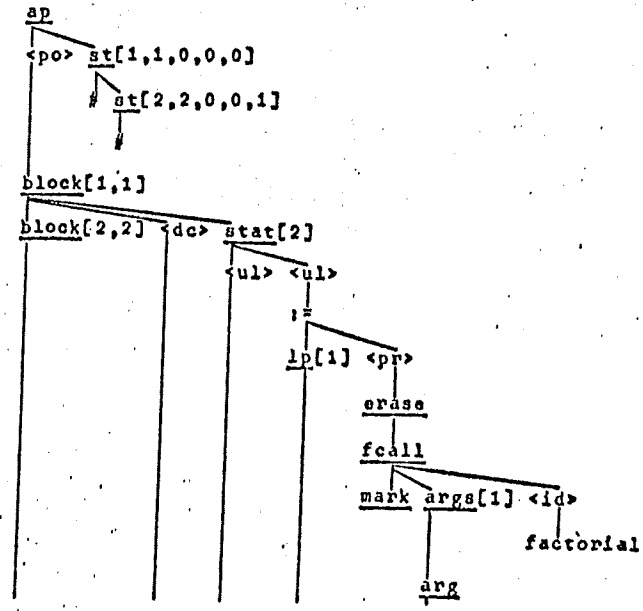


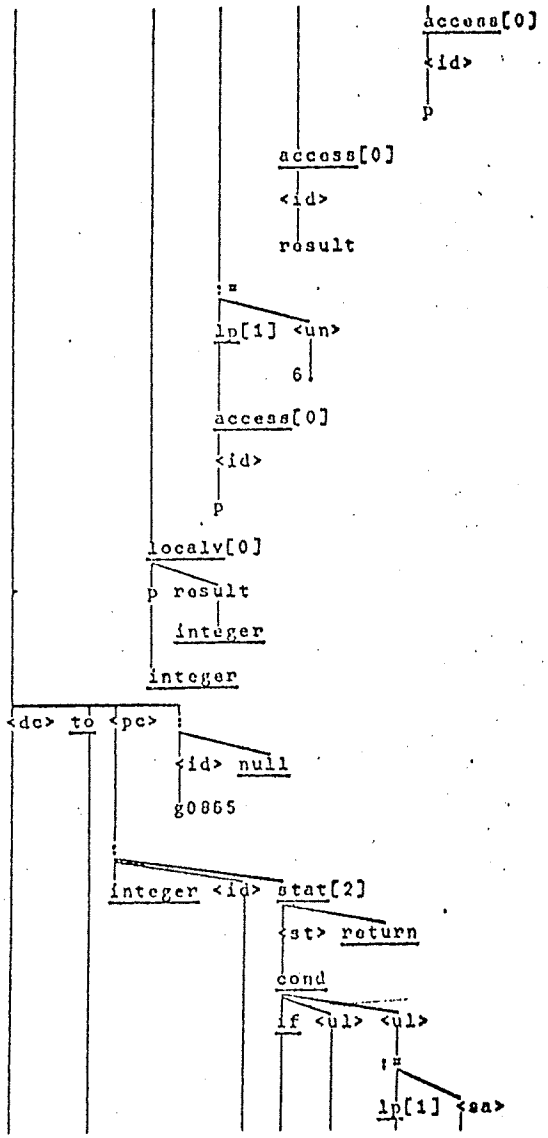


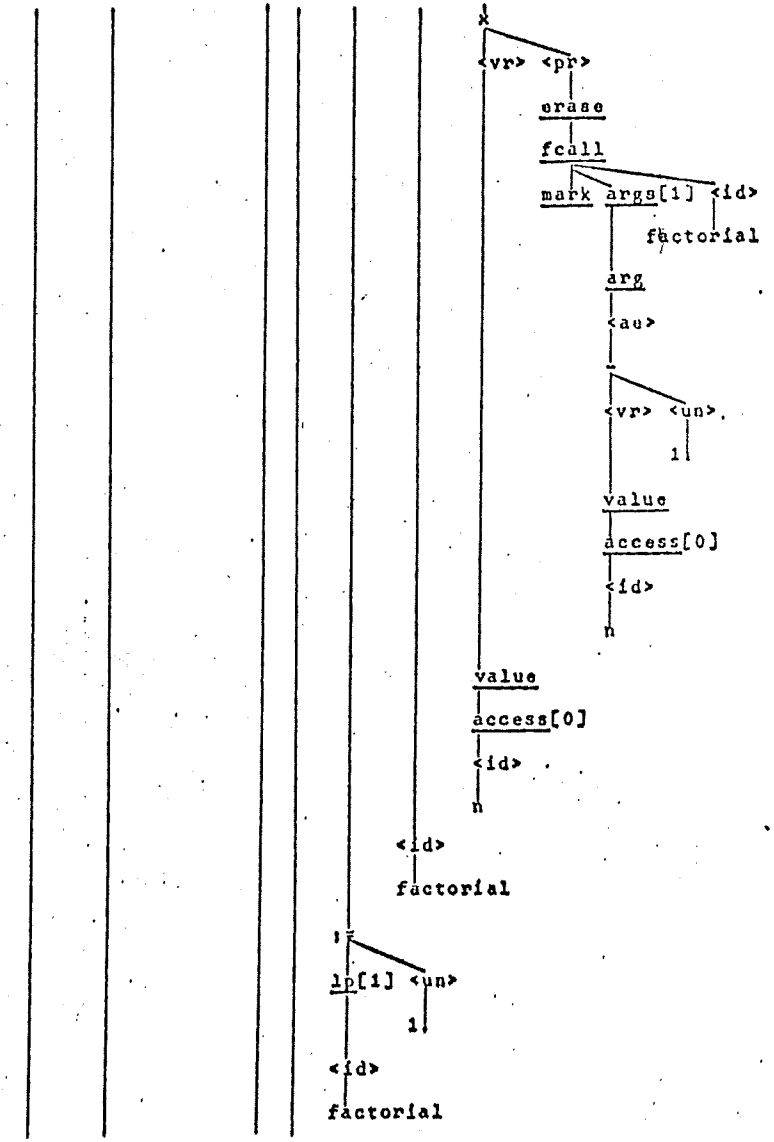


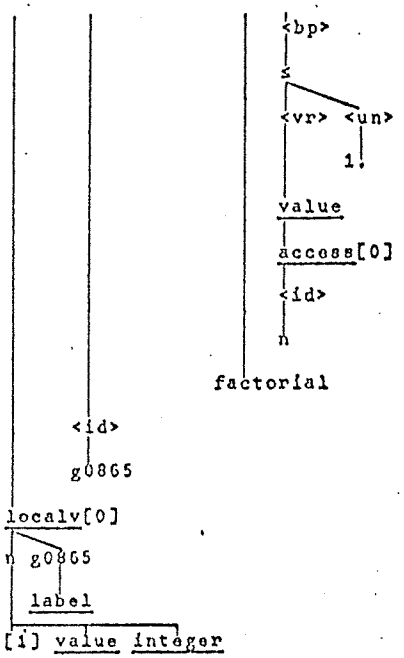


The successful application of transformations 151 (twice), 152, 153, 154, 155, 156, 157, and 158 removes superfluous nodes, collects and combines declarations, and counts and numbers blocks and inserts corresponding symbol table headers into a symbol table. The resulting tree is given below.

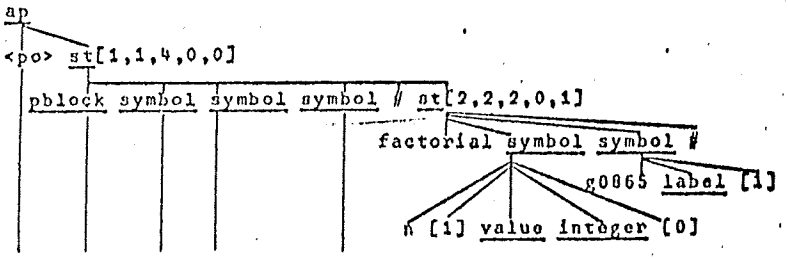


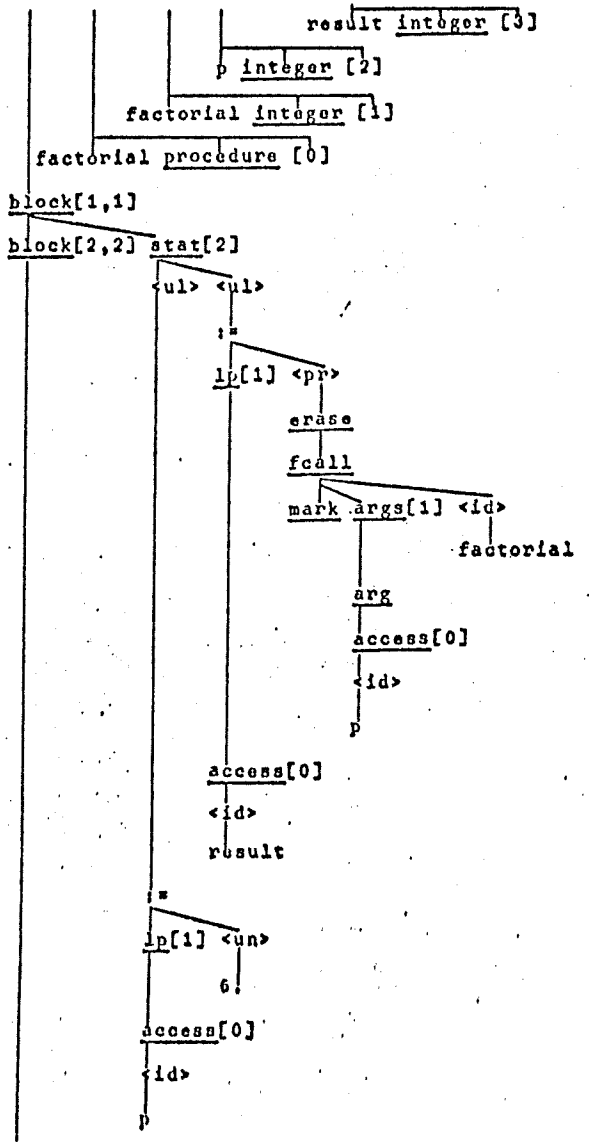

access[0]
<id>
p
access[0]
<id>
result
:=
lp[1] <un>
6
access[0]
<id>
p
localv[0]
p result
integer
integer
<dc> to <pc>
<id> null
g0865
integer <id> stat[2]
<st> return
cond
if <ul>
:=
lp[1] <sa>

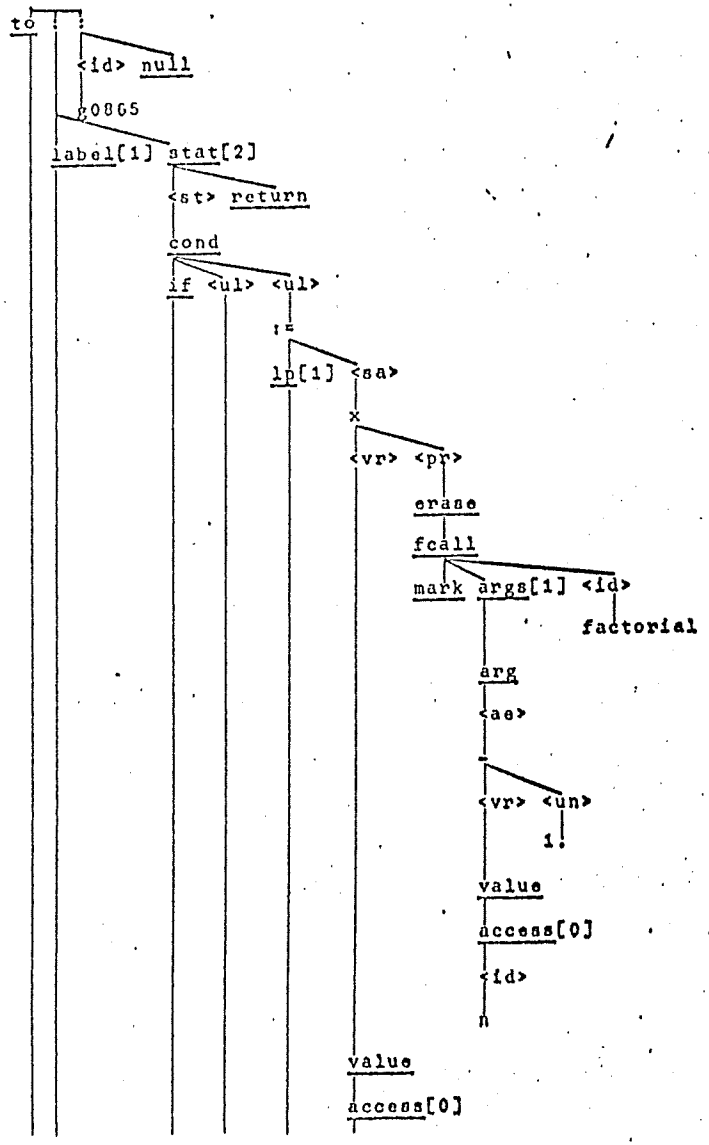

x
<vr> <pr>
erase
fcall
mark args[1] <id>
factorial
arg
<sa>
-
<vr> <un>
1
value
access[0]
<id>
n
value
access[0]
<id>
n
<id>
factorial
:=
lp[1] <un>
1
<id>
factorial

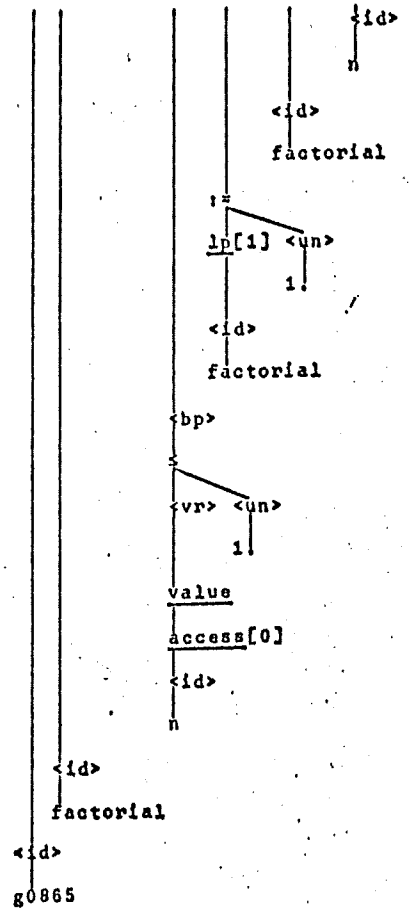


The successful application of transformations 161, 163 (four times), and 164 (twice) copies local symbol declarations into the symbol table for the block in which the symbols are declared and then removes the declarations. The resulting tree is given below.



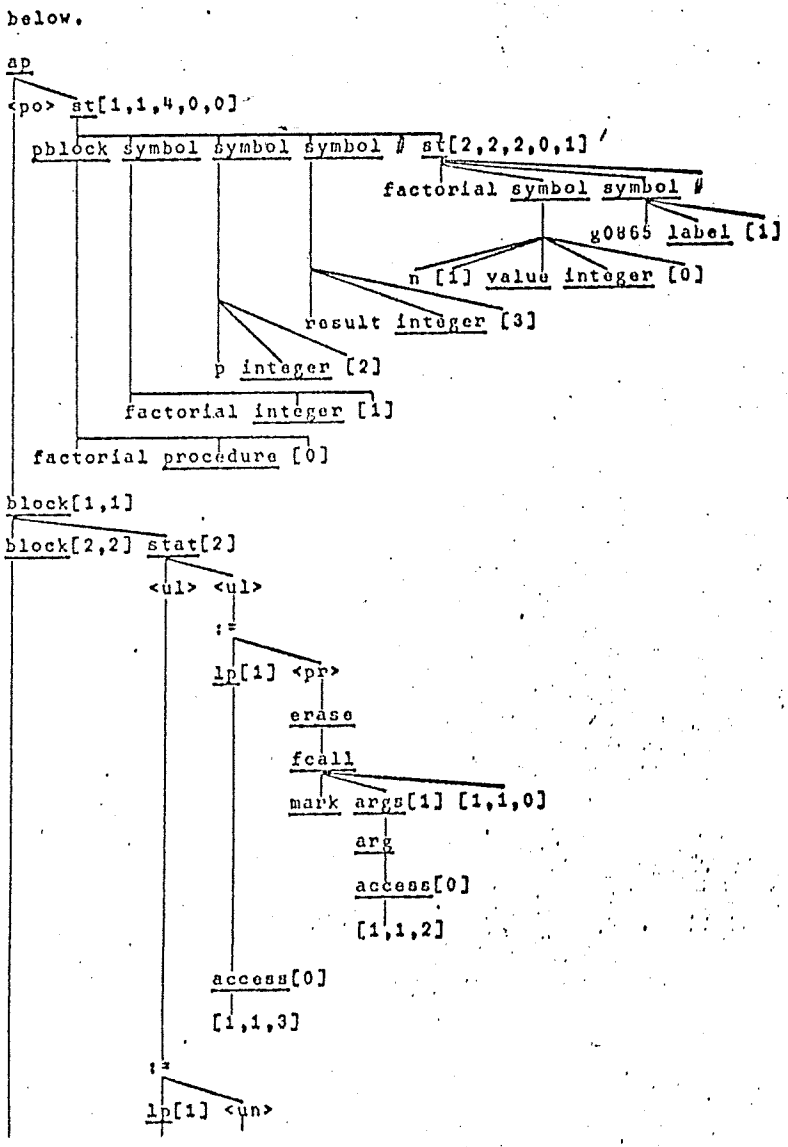




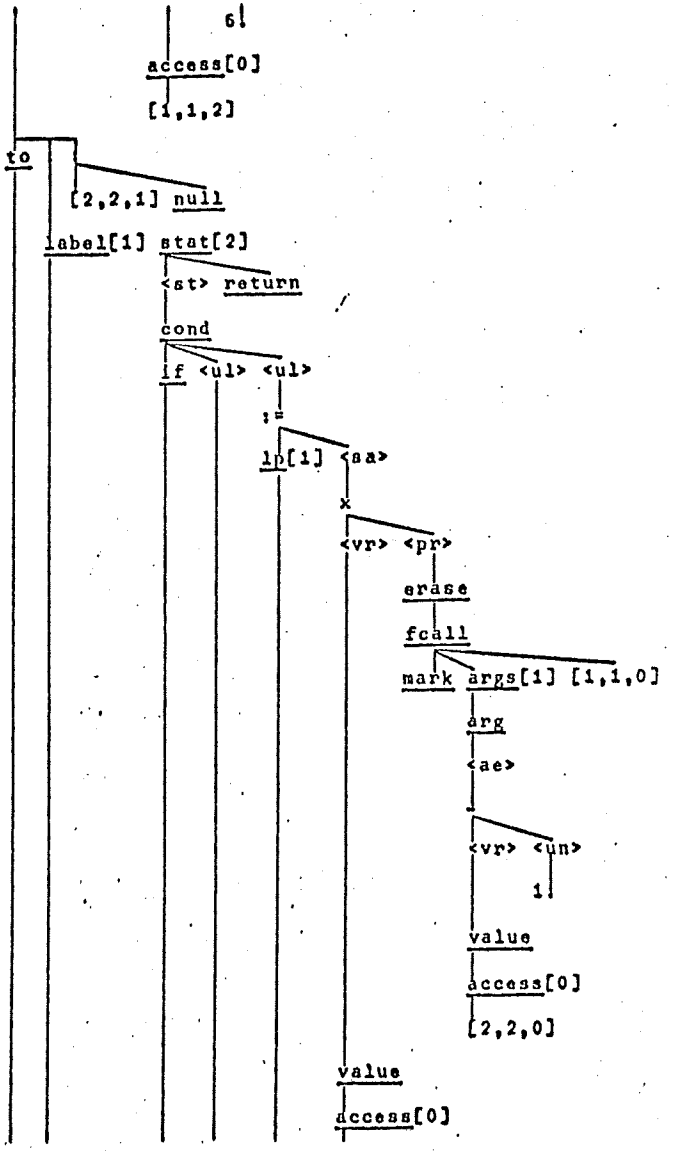


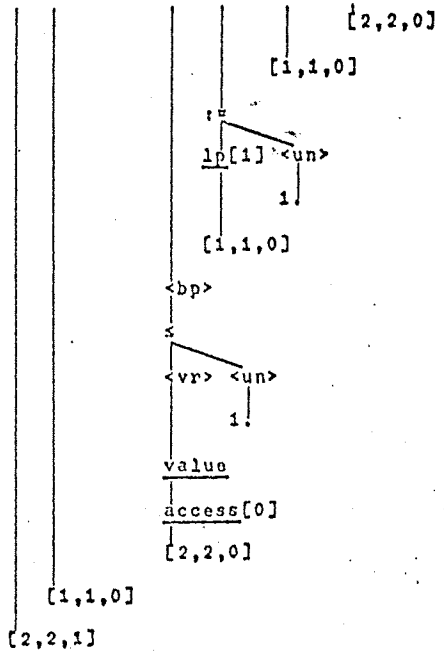
The successful application of transformations 167 (five times) and 168 (eight times) looks up symbols in the symbol table and replaces each by the number and level of the block in which it is declared and by a displacement within the storage area for the block. The resulting tree is given

below.

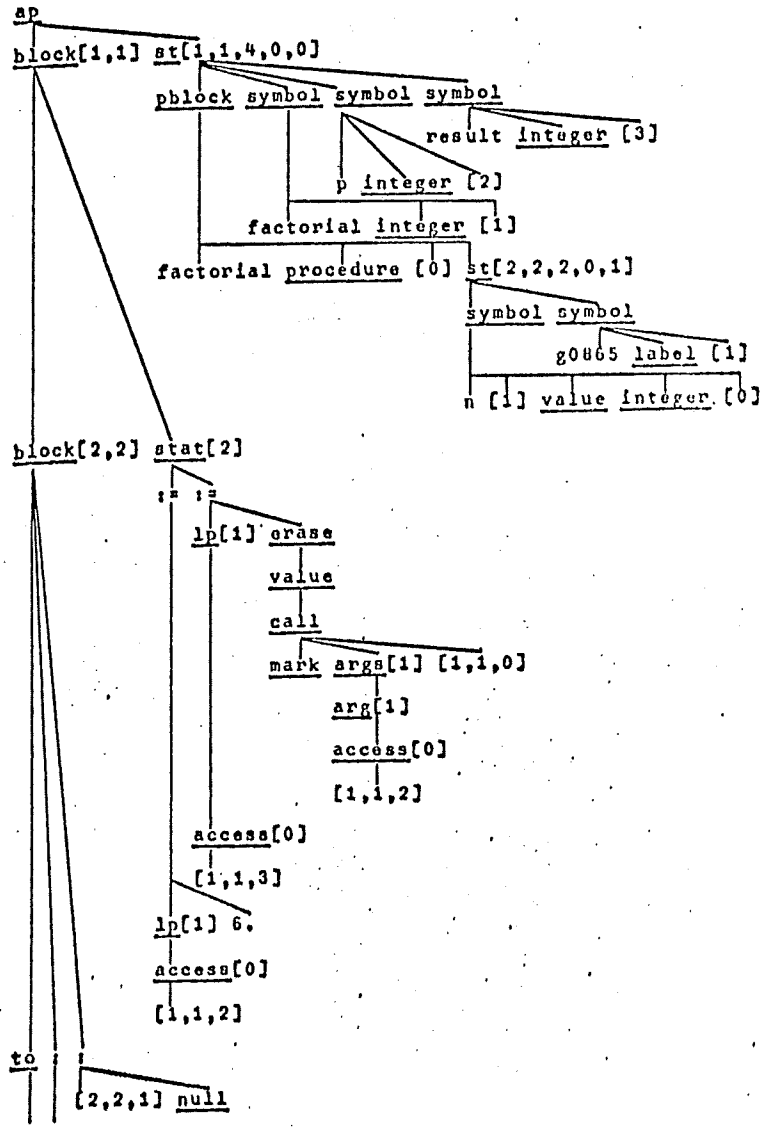

ap
<po> st[1,1,4,0,0]
pblock symbol symbol symbol # st[2,2,2,0,1]
factorial symbol symbol #
g0865 label [1]
n [1] value integer [0]
result integer [3]
p integer [2]
factorial integer [1]
factorial procedure [0]
block[1,1]
block[2,2] stat[2]

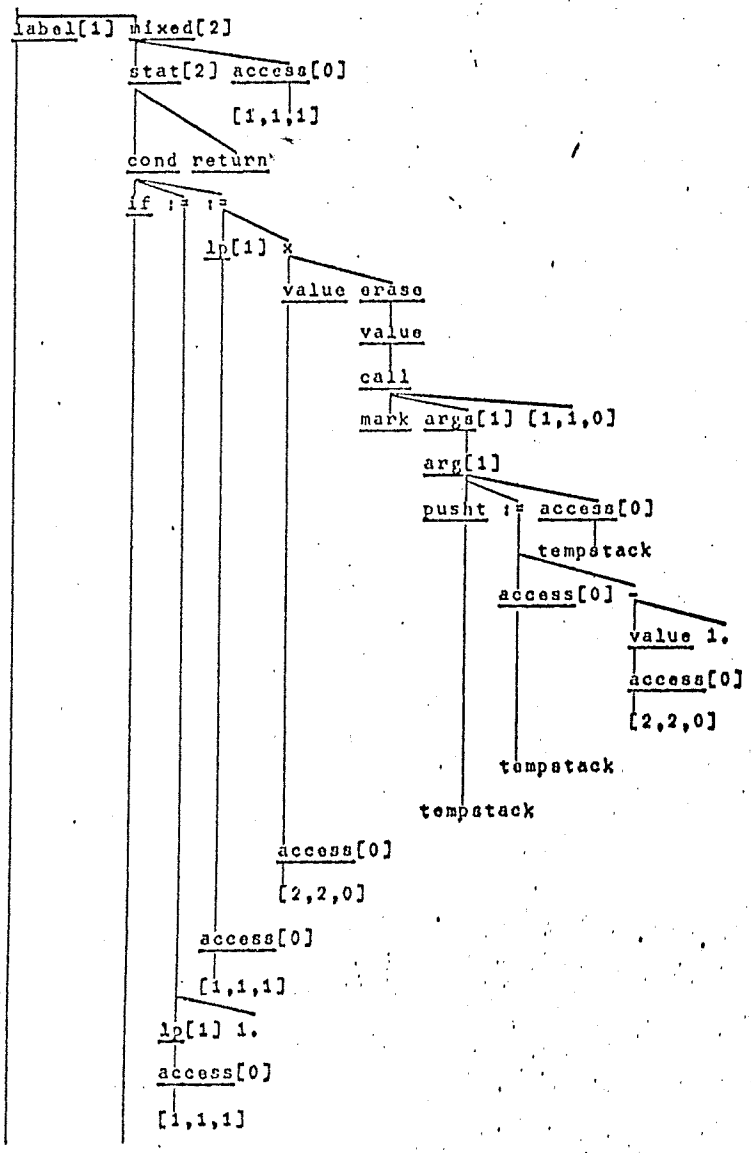
:=
lp[1] <pr>
erase
fcall
mark args[1] [1,1,0]
arg
access[0]
[1,1,2]
access[0]
[1,1,3]
:=
lp[1] <un>

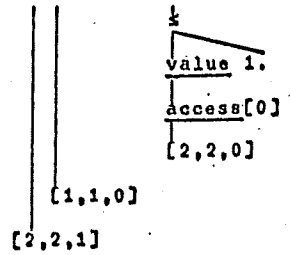

6
access[0]
[1,1,2]
to
[2,2,1] null
label[1] stat[2]
<st> return
cond
if <ul>
:=
lp[1] <aa>
x
<vr> <pr>
erase
fcall
mark args[1] [1,1,0]
arg
<ae>
<vr> <un>
1
value
access[0]
[2,2,0]
value
access[0]



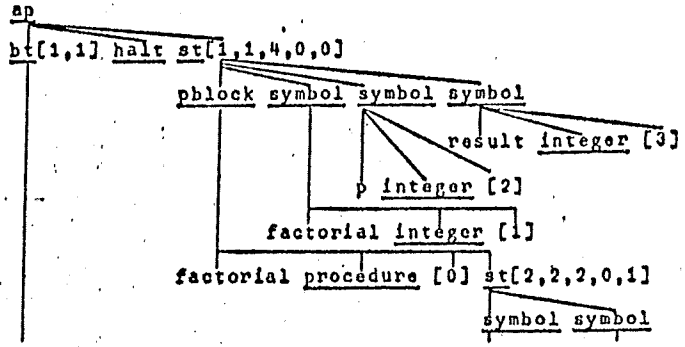
The successful application of transformations 174 (twice), 175, 176 (eighteen times), 177 (twice), 178 (twice), 179 (twice), 181 (twice), 184, 186, 195 (twice), 197, 198, and 199 adjusts access to function procedure names, rearranges the symbol table for a procedure, removes all remaining nonterminal nodes, marks and counts actual arguments in a function procedure call, expands argument expressions, and inserts operators and operands to make explicit the return of a value by a function procedure. The resulting tree is given below.

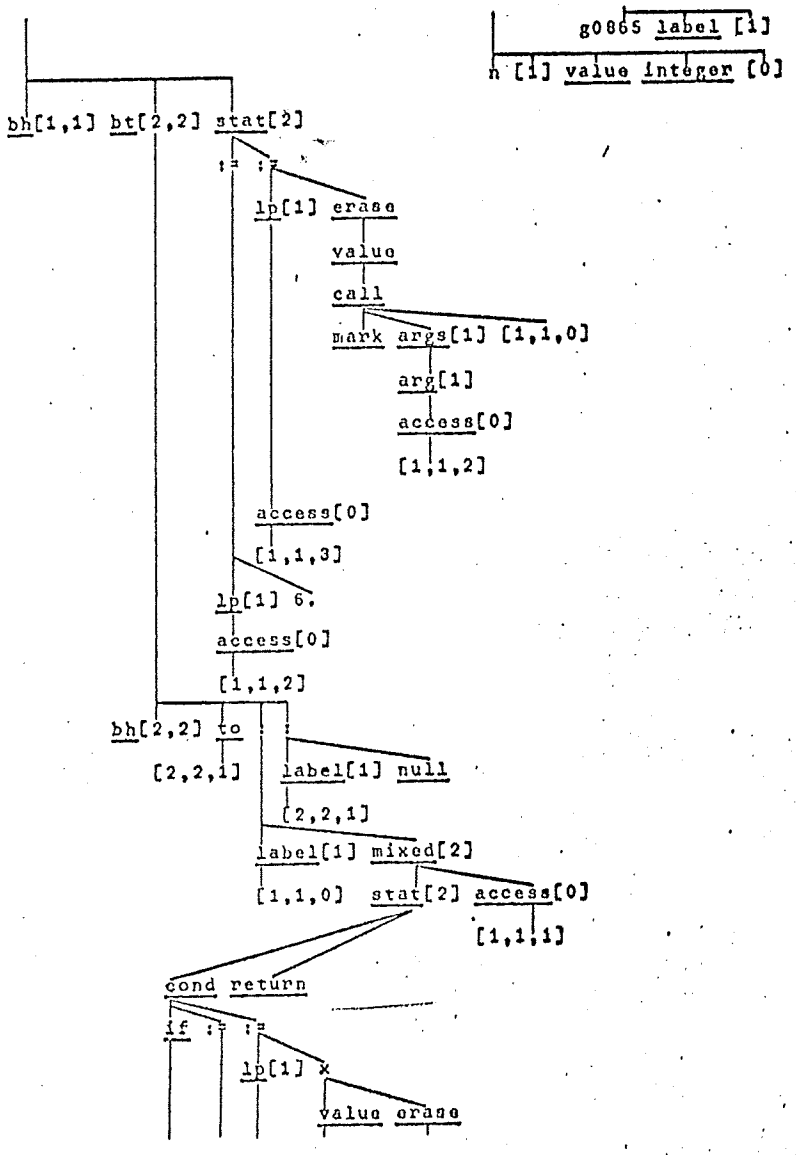


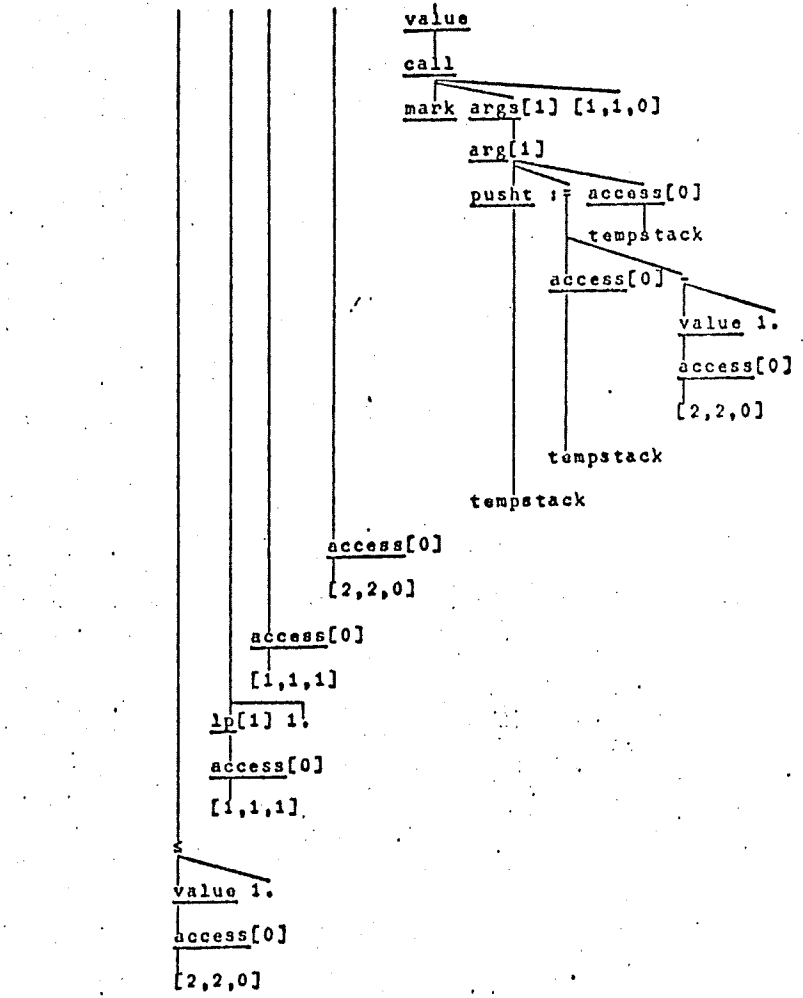




The successful application of transformations 211, 212 (twice), 213 (twice), 214, 215, 217 (five times), and 218 inserts a label collector node, threads blocks to corresponding symbol tables, threads procedure calls to the symbol table for the block of the procedure declaration, threads symbol table entries for symbols which are used as labels to the labeled constructs, threads label references to the symbol tables for the blocks of the label definitions, removes superfluous nodes, and inserts a halt operator. All the transformtions have now been applied and the resulting tree, given below, is the computation tree for the program.







thread from bt[1,1] to st[1,1,4,0,0]
thread from bh[1,1] to st[1,1,4,0,0]
thread from bt[2,2] to st[2,2,2,0,1]
thread from bh[2,2] to st[2,2,2,0,1]
thread from [2,2,1] to st[2,2,2,0,1]
thread from [1,1,0] to st[1,1,4,0,0]

```
thread from [1,1,0] to st[1,1,4,0,0]
thread from st[1,1,4,0,0] to bt[1,1]
thread from st[1,1,4,0,0] to bh[1,1]
thread from [0] to :
thread from st[2,2,2,0,1] to bt[2,2]
thread from st[2,2,2,0,1] to bh[2,2]
thread from [1] to :
```

Appendix III

AN IMPLEMENTATION OF THE SEMANTIC ANALYZER

This appendix contains a semantic analyzer coded in the University of California, Irvine, dialect of Lisp 1.6.

```
(DEFPROP FCNLIST
 (FCNLIST DRIVER
                  APLACE
                  ARB
                  ARB1
                  BAKMAT
                  BF
                  BND
                  BUILD
                  BUILD1
                  BUILD2
                  COMMENT
                  EXTEND
                  FAIL
                  FATHER
                  FIXN
                  LINK
                  LLINK
                  MAKLST
                  MARKT
                  MATCH
                  MIN
                  MIN1
                  MPOP
                  NANODE
                  NOFUN
                  POP
                  PUSH
                  REPLACE
                  RLINK
                  RSPND
                  SCRIPT
                  SEARCH
                  SEARCH1
                  SPLACE
                  STRIP
                  SUBC
```

```
          SUBNODE
          SUBS
          TAB
          THREAD
          THREADPRINT
          TPR
          TPRINT
          VAL
          WALK)
VALUE)

(DEFPROP DRIVER
 (LAMBDA NIL
  (PROG (PRE PTRN)
        (SETQ BASE 12)
        (NCONC GRINPROPS (QUOTE (COMMENT)))
        (SETQ **NIL (QUOTE (*NIL NIL NIL NIL NIL NIL)))
        (INPUT TREES DSK: (TREE . DAT))
   NXPAR:
        (INC (QUOTE TREES) NIL)
        (COND
         ((EQ (SETQ PRE (ERRSET (READ))) (QUOTE $EOF$))
          (OUTC (QUOTE TRACK) NIL)
          (OUTC NIL T)
          (RETURN T))
         (T (SETQ PRE (CAR PRE))))
        (INC NIL NIL)
        (SETQ TMODE NIL)
        (SETQ TR (BUILD PRE))
        (OUTPUT TRACK DSK: (XMPL . TXT))
        (OUTC (QUOTE TRACK) NIL)
        (LINELENGTH 1750)
        (PRINT (QUOTE (PARSE TREE IS:)))
        (TERPRI)
        (TPRINT (LLINK TR) 1)
        (TERPRI)
        (OUTC NIL NIL)
        (INPUT TRANS DSK: (TTTT . DAT))
   NXPTN:
        (INC (QUOTE TRANS) NIL)
        (COND
         ((EQ (SETQ NEXT (ERRSET (READ))) (QUOTE $EOF$))
          (SETQ MSTK NIL)
          (THREAD)
          (PRINT (QUOTE (ALL TRANSFORMATIONS TRIED)))
          (OUTC (QUOTE TRACK) NIL)
          (PRINT (QUOTE (COMPUTATION TREE IS:)))
          (TERPRI)
          (TPRINT (LLINK TR) 1)
          (TERPRI)
          (SETQ TP TR)
          (THREADPRINT TP)
```



```
          (TERPRI)
          (OUTC NIL NIL)
          (GO NXPAR:))
         (T (SETQ NEXT (CAR NEXT))
            (SETQ PTRN (CAR NEXT))
            (SETQ SBS (CADR NEXT))
            (SETQ NUMB (CADDR NEXT))))
        (COND
         ((NOT (NUMBERP NUMB))
          (SETQ TMODE T)
          (SETQ TLST NUMB)
          (SETQ NUMB (CADDDR NEXT)))
         (T (SETQ TLST NIL)))
        (INC NIL NIL)
        (SETQ PR (BUILD PTRN))
NXTRY:
        (SETQ
         DUMMY
         (QUOTE
          (NIL (*NIL NIL NIL NIL NIL NIL) NIL NIL NIL NIL)))
        (SETQ PFLAG 0)
        (SETQ FFLAG 0)
        (SETQ MSTK NIL)
        (SETQ TP TR)
        (SETQ PP PR)
        (SETQ ST NIL)
        (SETQ STP NIL)
        (SETQ ROOT (LLINK TP))
        (SETQ ROOT1 ROOT)
        (SETQ RSTK (CONS TP NIL))
L1:     (COND
         ((MATCH (QUOTE TP)
                 (QUOTE ST)
                 (QUOTE PP)
                 (QUOTE STP))
          (COND
           ((NOT (OR (NULL TP) (NULL PP))) (GO L1:))
           ((AND (NULL TP) (NULL PP))
            (NOFUN (QUOTE DRIVER))
            (COND
             ((REPLACE SBS)
              (PRINT (LIST NUMB (QUOTE S)))
              (OUTC (QUOTE TRACK) NIL)
              (PRINT (LIST NUMB (QUOTE S)))
              (COND
               ((RSPND) (SETQ MSTK NIL)
                        (TERPRI)
                        (TPRINT (LLINK TR) 1)
                        (TERPRI)))
              (OUTC NIL NIL)
              (NOFUN (QUOTE DRIVER))
              (GO NXTRY:))
```

```
            (T (OUTC (QUOTE TRACK) NIL)
               (OUTC NIL T)
               (PRINT (QUOTE (REPLACE FAILED)))
               (RETURN NIL))))
         (T (GO L2:)))))
L2:  (NOFUN (QUOTE DRIVER))
     (COND
      ((FAIL) (GO L1:))
      (T (PRINT (LIST NUMB (QUOTE F)))
         (COND
          ((RSPND) (OUTC (QUOTE TRACK) NIL)
                   (TERPRI)
                   (TPRINT (LLINK TR) 1)
                   (OUTC NIL NIL)))
         (NOFUN (QUOTE DRIVER))
         (GO NXPTN:)))))
EXPR)

(DEFPROP DRIVER
 "INITIALIZES, READS, AND WRITES SOURCE TREE, READS
TRANSFORMATION, AND CALLS MATCH.  IF MATCH IS SUCCESSFUL,
CALLS REPLACE TO INSERT SUBSTITUTION TREE.  IF REPLACE
FAILS, THEN DRIVER HALTS, OTHERWISE IT PRINTS SUCCESS
AND, OPTIONALLY, THE RESULTING TREE AND TRIES AGAIN
WITH THE SAME TRANSFORMATION.  IF MATCH FAILS, DRIVER
PRINTS FAILURE AND, OPTIONALLY, THE CURRENT SOURCE
TREE AND READS THE NEXT TRANSFORMATION.  WHEN ALL THE
TRANSFORMATIONS HAVE BEEN READ, DRIVER PRINTS THE
COMPUTATION TREE, AND READS THE NEXT SOURCE TREE.
WHEN ALL SOURCE TREES ARE PROCESSED, DRIVER HALTS.

THIS PROGRAM IS INITIATED BY CALLING THE MAIN ROUTINE, I.E.,
(DRIVER).  THE FOLLOWING COMMANDS MUST BE EXECUTED BEFORE
INITIATING THE MAIN PROGRAM SO THAT LEFT AND RIGHT
BRACKETS AND COMMAS MAY BE WRITTEN IN THE OUTPUT FILE:
  (MODCHR 133 (MODCHR 101 NIL))
  (MODCHR 135 (MODCHR 101 NIL))
  (MODCHR 54 (MODCHR 101 NIL)).

THE PROGRAM REQUIRES DISK FILES TREE.DAT CONTAINING SOURCE
TREES, TTTT.DAT CONTAINING TRANSFORMATIONS, AND
XMPL.TXT FOR OUTPUT.  SOURCE TREES ARE CONVERTED TO BINARY
FORM (AS IN KNUTH).  SOURCE TREE INPUT IS OF THE FORM
  (ROOT LEFTSUBTREE RIGHTSUBTREE)
WHERE EACH SUBTREE CAN BE AN ATOM OR LIST.  TRANSFORMATION
INPUT IS OF THE FORM
  (MATCHTREE SUBSTITUTIONTREE NUMBER)
WHERE EACH TREE HAS THE FORM GIVEN FOR SOURCE TREES.
THE TWO LETTER ABBREVIATIONS FOR ALGOL 60
METALINGUISTIC VARIABLES GIVEN IN APPENDIX I ARE USED IN
THE INPUT PRECEDED BY A DOLLAR SIGN ($).  AN
EXCLAMATION POINT (!) IS USED TO INDICATE TERMINAL SYMBOLS
```



```
OF THE TARGET AND SOURCE LANGUAGES.  SUBSCRIPTS ARE
ENCLOSED IN ANGLE BRACKETS (<>) AND ARE SEPARATED BY
COMMAS.  FOR EXAMPLE, A BLOCK HEAD NODE IS WRITTEN
  <I/,J>!BH
AND <IDENTIFIER> IF WRITTEN
  $ID.

THE OUTPUT IS GIVEN AS AN N-ARY TREE WITH CONTROL
CHARACTERS FOR THE RUNOFF TEXT PROGRAM SO THAT
SELECTIVE CAPITALIZATION AND UNDERLINING MAY BE DONE.
THE GENERAL FORM AND CONTENT ARE SHOWN IN THE TREES OF
APPENDIX II, ALTHOUGH PAGE PLACEMENT HAS BEEN MODIFIED
BY HAND FOR SOME OF THEM.  THIS OUTPUT FILE SHOULD BE RUN
IN RUNOFF NOFILL MODE."
COMMENT)

(DEFPROP APLACE
 (LAMBDA(SPA STSA PA V)
  (PROG (PA1 L1)
        (COND
         ((NULL PA) (RETURN NIL))
         ((AND (EQ V (QUOTE ARB))
               (NOT (NULL (LLINK (EVAL SPA)))))
          (PRINT (QUOTE (ERROR: ARB NODE CANNOT HAVE SON)))
          (RETURN NIL))
         ((AND (NULL (CADR PA)) (EQ V (QUOTE ARB)))
          (LINK SPA (RLINK (EVAL SPA)))
          (NOFUN (QUOTE APLACE1))
          (RETURN T))
         ((AND (EQUAL (CADR PA) (CADDR PA))
               (OR (EQ V (QUOTE MIN)) (EQ V (QUOTE BND))))
          (NOFUN (QUOTE APLACE2))
          (COND
           ((NULL (RLINK (EVAL SPA)))
            (COND
             ((NULL (LLINK (EVAL SPA)))
              (LINK SPA NIL)
              (RETURN T))
             (T (LINK SPA (LLINK (EVAL SPA))) (RETURN T))))
           ((NULL (LLINK (EVAL SPA)))
            (LINK SPA (RLINK (EVAL SPA)))
            (RETURN T))
           (T (LINK SPA (LLINK (EVAL SPA)))
              (SETQ PA1 (LLINK (EVAL SPA)))
              (GO RIGHT:))))
          (NOFUN (QUOTE APLACE3))
          (LINK SPA (CADR PA))
          (COND ((NOT (NULL (CADDR PA))) (GO PART:)))
          (COND ((NULL (RLINK (EVAL SPA))) (RETURN T)))
          (SETQ PA1 (CADR PA))
    RIGHT:
          (NOFUN (QUOTE APLACE4))
```

```
        (COND
         ((NOT (NULL (RLINK PA1)))
          (SETQ PA1 (RLINK PA1))
          (GO RIGHT:))
         (T (RPLACA (CDDR PA1) (RLINK (EVAL SPA)))
            (SETQ L1 (GENSYM))
            (SET L1 PA1)
            (RPLACA (CDDDR (RLINK (EVAL SPA))) L1)
            (RETURN T)))
    PART:
        (NOFUN (QUOTE APLACE5))
        (COND
         ((EQ V (QUOTE ARB))
          (LINK (QUOTE (CADDR PA)) (RLINK (EVAL SPA)))
          (RETURN T))
         (T (LINK (QUOTE (CADDR PA)) (LLINK (EVAL SPA)))
            (NOFUN (QUOTE APLACE6))
            (COND
             ((NOT (NULL (RLINK (CADDR PA))))
              (RPLACA (CDDR (LLINK (EVAL SPA)))
                      (RLINK (CADDR PA)))
              (SETQ L1 (GENSYM))
              (SET L1 (LLINK (EVAL SPA)))
              (RPLACA (CDDDR (RLINK (CADDR PA))) L1)))
            (NOFUN (QUOTE APLACE7))
            (COND
             ((NULL (RLINK (EVAL SPA))) (RETURN T))
             (T (RPLACA (CDDR (CADR PA)) (RLINK (EVAL SPA)))
                (SETQ L1 (GENSYM))
                (SET L1 (CADR PA))
                (RPLACA (CDDDR (RLINK (EVAL SPA))) L1)
                (RETURN T)))))))
EXPR)

(DEFPROP APLACE
 "AFTER SUBSTITUTION TREE IS INSERTED IN
SOURCE TREE, APLACE REPLACES ARB, MIN, AND BND NODES
BY WHAT MATCHES THEM."
COMMENT)

(DEFPROP ARB
 (LAMBDA(AT1 AST1 AT2 AST2)
  (PROG (PST1 PST2 X Y AFLAG)
        (SETQ PST1 (EVAL AST1))
        (SETQ PST2 (EVAL AST2))
        (SETQ AFLAG (ARB1 AT1 (QUOTE X) AT2 (QUOTE Y)))
        (COND ((NULL AFLAG) (RETURN NIL))
              ((NULL (SEARCH1 (CAAR HSTK) (CDR HSTK)))
               (GO DONE:))
              (T (PRINT
                   (QUOTE
                    (ERROR: TWO ARBS WITH SAME SUBSCRIPT)))
                 (SETQ HSTK (POP HSTK))
                 (RETURN NIL)))
    DONE:
        (NOFUN (QUOTE ARB))
        (COND ((NOT (NULL X)) (SET AST1 (*APPEND X PST1))))
        (COND ((NOT (NULL Y)) (SET AST2 (*APPEND Y PST2))))
        (RETURN T)))
EXPR)
```



```
(DEFPROP ARB
 "SETS UP FOR AND CALLS ARB1 TO DO MATCH FOR ARB
NODE.  IF MATCH FAILS, RETURNS NIL, OTHERWISE CHECKS TO
SEE IF ARB MATCH USING SAME SUBSCRIPT HAS BEEN MADE.
IF SO, RETURNS NIL, OTHERWISE RETURNS TRUE."
COMMENT)

(DEFPROP ARB1
 (LAMBDA(A1T1 A1ST1 A1T2 A1ST2)
  (PROG (APT1 APT2 AP2T2 AP2T1 APT1B AP2T1B COUNT)
        (SETQ COUNT FFLAG)
        (SETQ APT1 (EVAL A1T1))
        (SETQ APT2 (EVAL A1T2))
   L3:  (WALK A1T2 A1ST2)
        (SETQ AP2T2 (EVAL A1T2))
        (COND ((NULL AP2T2)
               (COND ((EQUAL APT1 **NIL) (SETQ APT1 NIL)))
               (SETQ HSTK
                     (CONS (LIST APT2 APT1 NIL (QUOTE NO))
                           HSTK))
               (SET A1T1 **NIL)
               (SET A1T2 **NIL)
               (RETURN T))
              ((EQ (VAL AP2T2) (QUOTE *NIL)) (GO L3:))
              ((AND (NULL APT1) (NOT (NULL AP2T2)))
               (RETURN NIL))
              (T (SETQ APT1B (STRIP (VAL APT1)))))
        (COND ((EQ (VAL AP2T2) APT1B)
               (COND ((EQ FFLAG 0)
                      (SETQ HSTK
                            (CONS (LIST APT2
                                        NIL
                                        APT1
                                        COUNT
                                        ST
                                        STP)
                                  HSTK))
                      (RETURN T))
                     (T (SETQ FFLAG (SUB1 FFLAG))
                        (GO L4:))))
              ((ATOM (VAL AP2T2)) (GO L4:))
              (T (SETQ HSTK
                       (CONS (LIST APT2
```

```
                                NIL
                                APT1
                                COUNT
                                ST
                                STP)
                          HSTK))
               (COND ((NANODE A1T1 A1ST1 A1T2 A1ST2)
                      (COND ((EQ FFLAG 0) (RETURN T))
                            (T (SETQ FFLAG (SUB1 FFLAG))
                               (MPOP (VAL APT2))
                               (GO L4:))))
                     (T (SETQ HSTK (POP HSTK))
                        (GO L4:)))))
L4:  (NOFUN (QUOTE ARB1))
     (WALK A1T1 A1ST1)
     (SETQ AP2T1 (EVAL A1T1))
     (COND ((NULL AP2T1)
            (COND ((ATOM (VAL AP2T2)) (RETURN NIL))
                  ((EQ (CAR AP2T2) (QUOTE ARB))
                   (SETQ HSTK
                         (CONS (LIST APT2
                                     NIL
                                     NIL
                                     (QUOTE NO))
                               HSTK))
                   (SETQ MSTK
                         (CONS (LIST AP2T2
                                     NIL
                                     NIL
                                     (QUOTE NO))
                               HSTK))
                   (SET A1T1 **NIL)
                   (RETURN T))
                  (T (RETURN NIL))))
           ((EQ (VAL AP2T1) (QUOTE *NIL)) (GO L4:))
           (T (SETQ AP2T1B (STRIP (VAL AP2T1)))))
     (COND ((EQ (VAL AP2T2) AP2T1B)
            (COND ((EQ FFLAG 0)
                   (SETQ MSTK
                         (CONS (LIST APT2
                                     APT1
                                     AP2T1
                                     COUNT
                                     ST
                                     STP)
                               HSTK))
                   (RETURN T))
                  (T (SETQ FFLAG (SUB1 FFLAG))
                     (GO L4:))))
           ((ATOM (VAL AP2T2)) (GO L4:))
           (T (SETQ MSTK
                    (CONS (LIST APT2
```



```
                                APT1
                                AP2T1
                                COUNT
                                ST
                                STP)
                          HSTK))
               (COND ((NANODE A1T1 A1ST1 A1T2 A1ST2)
                      (COND ((EQ FFLAG 0) (RETURN T))
                            (T (SETQ FFLAG (SUB1 FFLAG))
                               (MPOP (VAL APT2))
                               (GO L4:))))
                     (T (SETQ HSTK (POP HSTK))
                        (GO L4:)))))))
EXPR)

(DEFPROP ARB1
 "DOES ARB MATCH."
COMMENT)

(DEFPROP BAKMAT
 (LAMBDA NIL
  (PROG (PK PK1 PK2)
        (SETQ PK (SEARCH1 (CAAAR MSTK) (CDR MSTK)))
        (COND ((NULL PK) (RETURN T))
              (T (SETQ PK1 (STRIP (CADAR MSTK)))
                 (SETQ PK2 (STRIP (CADR PK)))))
        (COND ((EQUAL PK1 PK2) (SETQ MSTK (POP MSTK))
                               (RETURN T))
              (T (SETQ MSTK (POP MSTK)) (RETURN NIL))))))
EXPR)

(DEFPROP BAKMAT
 "IF A SUBSCRIPTED NODE OTHER THAN ARB, MIN, OR BND HAS
BEEN MATCHED, BAKMAT CHECKS TO SEE THAT THERE IS NO OTHER
MATCH FOR THIS NODE HAVING THE SAME SUBSCRIPT BUT MATCHING
A DIFFERENT STRUCTURE.  IF MATCHES ARE NOT IDENTICAL, IT
POPS MATCH STACK AND RETURNS NIL, OTHERWISE TRUE."
COMMENT)

(DEFPROP BF
 (LAMBDA(NO BFT2)
  (PROG (BFX)
        (COND ((EQ NO 0) (RETURN NIL)))
        (SETQ BFX BFT2)
   BF1: (COND ((EQ (LLINK (EVAL (FATHER BFX))) BFX)
               (RETURN
                (CONS (VAL (EVAL (FATHER BFX)))
                      (BF (SUB1 NO) (EVAL (FATHER BFX))))))
              (T (SETQ BFX (EVAL (FATHER BFX)))
                 (GO BF1:))))))
EXPR)
```

```
(DEFPROP BF
 "BUILDS LIST OF ANCESTORS TO USE IN CHECKING
BOUND CONDITIONS IN BND MATCH."
COMMENT)

(DEFPROP BND
 (LAMBDA(BT1 BST1 BT2 BST2)
  (PROG (BX BY FLAG BLST W Z1 Z2 Z3 Z4 Z6 S1 B B1)
        (SETQ FLAG (MIN1 BT1 (QUOTE BX) BT2 (QUOTE BY)))
        (COND ((NULL FLAG) (RETURN NIL)))
        (SETQ Z6 MSTK)
   FIND:
        (COND ((EQ (CAAAAR Z6) (QUOTE BND))
               (SETQ BLST
                     (REVERSE
                      (BF (CAADR (CAAAR Z6)) (CAAR Z6)))))
              (T (SETQ Z6 (CDR Z6)) (GO FIND:)))
        (COND ((NULL BLST) (GO TRUE:)))
        (SETQ W (CAR Z6))
        (SETQ B BLST)
        (SETQ Z1 (CADR W))
        (SETQ Z2 (CADDR W))
        (SETQ Z4 Z1)
        (COND ((OR (NULL Z1) (EQUAL Z1 Z2)) (GO TRUE:)))
        (COND ((NULL BLST) (GO TRUE:)))
        (SETQ B1 (CAR BLST))
   L10: (COND
         ((EQ B1 (VAL Z1))
          (SETQ B (CDR B))
          (COND ((NULL B) (SETQ MSTK (CDR Z6-) (RETURN NIL))
                (T (SETQ B1 (CAR B))
                   (SETQ Z1 (LLINK Z1))
                   (COND ((OR (NULL Z1) (EQUAL Z1 Z2))
                          (GO TRUE:))
                         (T (GO L10:)))))))
   L9:  (SETQ Z3 Z1)
        (WALK (QUOTE Z1) (QUOTE S1))
        (COND ((EQUAL Z3 Z4) (SETQ S1 NIL)))
        (COND ((NULL Z1) (GO TRUE:))
              ((EQUAL (VAL Z1) (QUOTE *NIL)) (GO L9:))
              ((EQUAL Z1 Z2) (GO TRUE:))
              (T (GO L10:)))
   TRUE:
        (COND ((NULL (SEARCH1 (CAAAR Z6) (CDR Z6)))
               (GO DONE:))
              (T (PRINT
                  (QUOTE
                   (ERROR: TWO BNDS WITH SAME SUBSCRIPT)))
                 (SETQ MSTK (CDR Z6))
                 (RETURN NIL)))
   DONE:
        (NOFUN (QUOTE BND))
```



```
        (COND
         ((NOT (NULL BX))
          (SET BST1 (*APPEND BX (EVAL BST1)))))
        (COND
         ((NOT (NULL BY))
          (SET BST2 (*APPEND BY (EVAL BST2)))))
        (SETQ PFLAG 2)
        (NOFUN (QUOTE BND))
        (RETURN T)))
EXPR)

(DEFPROP BND
 "SETS UP FOR DOING BND MATCH WITH SOURCE TREE BT1
AND MATCH TREE BT2 AND CORRESPONDING MATCH TREE STACKS BST1
AND BST2.  CALLS MIN1 TO DO MATCH AND THEN CHECKS BOUND
CONDITIONS.  ADJUSTS STACK APPROPRIATELY AFTER MATCH."
COMMENT)

(DEFPROP BUILD
 (LAMBDA(L)
  (PROG (X)
        (COND
         ((NULL L) (RETURN (LIST NIL NIL NIL NIL NIL NIL NIL))))
        (SETQ X (LIST NIL (BUILD1 L) NIL NIL NIL NIL))
        (BUILD2 X NIL)
        (RPLACA (CDDDR X) NIL)
        (RETURN X)))
EXPR)

(DEFPROP BUILD
 "TAKES S-EXPRESSION L AND CONSTRUCTS
TOP LEVEL INTERNAL TREE REPLRESENTATION.  EACH
TREE NODE HAS SIX FIELDS: VALUE, LEFTLINK,
RIGHTLINK, FATHER, THREAD1, AND THREAD2."
COMMENT)

(DEFPROP BUILD1
 (LAMBDA(L)
  (COND ((NULL L) NIL)
        ((ATOM L) (LIST (SUBNODE L) NIL NIL NIL NIL NIL))
        (T
         (LIST (SUBNODE (CAR L))
               (BUILD1 (CADR L))
               (BUILD1 (CADDR L))
               NIL
               NIL
               NIL))))
EXPR)

(DEFPROP BUILD1
 "CONSTRUCTS INTERNAL TREE REPRESENTATION
INSERTING VALUE, LEFTLINK, AND RIGHTLINK FIELDS."
```

```
COMMENT)

(DEFPROP BUILD2
 (LAMBDA(TREE H)
  (PROG (X)
        (COND ((NULL TREE) (RETURN NIL)))
        (SETQ X (GENSYM))
        (SET X H)
        (RPLACA (CDDDR TREE) X)
        (BUILD2 (CADR TREE) TREE)
        (BUILD2 (CADDR TREE) TREE)
        (RETURN NIL)))
EXPR)

(DEFPROP BUILD2
 "INSERTS FATHER LINKS IN TREE."
COMMENT)

(DEFPROP COMMENT
 (LAMBDA(X)
  (PROG NIL
        (PUTPROP (CAR X) (CADR X) (QUOTE COMMENT))
        (RETURN NIL)))
FEXPR)

(DEFPROP COMMENT
 "PUTS COMMENTS ON PROPERTY LIST."
COMMENT)

(DEFPROP EXTEND
 (LAMBDA(TEX1)
  (PROG NIL
        (NOFUN (QUOTE EXTEND))
        (SETQ PP (CAAR MSTK))
        (SETQ TP (CADAR MSTK))
        (COND ((NULL TP) (SETQ TP (CADDAR MSTK))))
        (COND ((EQ TEX1 (QUOTE ARB))
               (SETQ ST (CADR (CDDDAR MSTK)))
               (SETQ STP (CADDR (CDDDAR MSTK))))
              (T (SETQ ST (CADDDR (CDDDAR MSTK)))
                 (SETQ STP (CAR (CDDDDR (CDDDAR MSTK))))
                 (SETQ PFLAG
                       (CADR (CDDDDR (CDDDAR MSTK))))))
        (SETQ MSTK (POP MSTK))
        (SETQ PFLAG (ADD1 PFLAG))
        (RETURN
         ((EVAL (QUOTE TEX1))
          (QUOTE TP)
          (QUOTE ST)
          (QUOTE PP)
          (QUOTE STP)))))
EXPR)
```



```
(DEFPROP EXTEND
 "IF A NODE-BY-NODE MATCH HAS FAILED, AND AN ARB, MIN,
OR BND NODE HAS BEEN MATCHED TO A SOURCE TREE NODE, FAIL
CALLS EXTEND TO ATTEMPT TO ENLARGE THAT MATCH TO INCLUDE
ADDITIONAL SOURCE TREE NODES.  RETURNS TRUE ON SUCCESSFUL
EXTENSION, NIL OTHERWISE."
COMMENT)

(DEFPROP FAIL
 (LAMBDA NIL
  (PROG (TMP)
   NEWH:
        (SETQ PFLAG 0)
        (NOFUN (QUOTE FAIL))
        (COND
         ((NULL MSTK)
          (COND ((NULL ROOT) (RETURN NIL))
                (T (GO NUROOT:))))
         (T (SETQ TMP (CAAAAR MSTK))
            (COND
             ((OR (EQ TMP (QUOTE TERM))
                  (EQ TMP (QUOTE NON))
                  (EQ TMP (QUOTE SYM))
                  (EQ TMP (QUOTE IL)))
              (SETQ MSTK (POP MSTK))
              (GO NEWH:))
             ((EQ TMP (QUOTE ARB))
              (COND
               ((EQ (CAR (CDDDAR MSTK)) (QUOTE NO))
                (SETQ MSTK (POP MSTK))
                (GO NEWH:))
               (T (SETQ PFLAG (CAR (CDDDAR MSTK)))
                  (COND
                   ((EXTEND TMP) (SETQ PFLAG 0) (RETURN T))
                   (T (GO NEWH:))))))
             ((OR (EQ TMP (QUOTE MIN)) (EQ TMP (QUOTE BND)))
              (COND
               ((EQ (CADDR (CDDDAR MSTK)) (QUOTE NO))
                (SETQ MSTK (POP MSTK))
                (GO NEWH:))
               (T (SETQ PFLAG (CADDR (CDDDAR MSTK)))
                  (COND
                   ((EXTEND TMP) (SETQ PFLAG 0) (RETURN T))
                   (T (GO NEWH:))))))
             (T (SETQ MSTK (POP MSTK)) (GO NEWH:)))))
   NUROOT:
        (WALK (QUOTE ROOT) (QUOTE RSTK))
        (COND
         ((NULL ROOT) (RETURN NIL))
         ((EQ (VAL ROOT) (QUOTE *NIL)) (GO NUROOT:))
         (T (SETQ
```

```
            TROOT
            (LIST (VAL ROOT)
                  (LLINK ROOT)
                  NIL
                  (FATHER ROOT)
                  (CAR (CDDDDR ROOT))
                  (CADR (CDDDDR ROOT))))
           (RPLACA (CDR DUMMY) TROOT)
           (SETQ TP DUMMY)
           (SETQ ST NIL)
           (SETQ PP PR)
           (SETQ STP NIL)
           (SETQ ROOT1 ROOT)
           (SETQ FFLAG 0)
           (RETURN T)))))
EXPR)

(DEFPROP FAIL
 "AFTER NODE-BY-NODE MATCH OF SOURCE AND MATCH
TREES HAS FAILED, FAIL MOVES TO NEST NODE IN SOURCE
TREE FROM WHICH MATCH MAY BE RETRIED, RESETS STARTING
POINTERS, AND RETURNS TRUE.  IF THERE IS NO NEXT NODE
FROM WHICH TO BEGIN RETRY, FAIL RETURNS NIL."
COMMENT)

(DEFPROP FATHER
 (LAMBDA (X) (CADDDR X))
EXPR)

(DEFPROP FATHER
 "RETURNS FATHER LINK FIELD OF CURRENT TREE NODE."
COMMENT)

(DEFPROP FIXN
 (LAMBDA(NODE TLEFT2)
  (PROG (F1 F2 F3 F4)
        (SETQ F4 NIL)
        (COND ((ATOM NODE) (GO LF4:)))
        (SETQ F3 0)
        (SETQ F1 (CAR NODE))
        (COND ((NOT (ATOM F1)) (GO LF5:)))
        (SETQ F4 F1)
        (SETQ F1 (CADR NODE))
        (SETQ F2 (EXPLODE F4))
        (COND ((EQ (CAR F2) (QUOTE !))
               (SETQ F4
                     (APPEND (QUOTE (| _))
                             (CDR F2)
                             (QUOTE (| _))))
               (SETQ F3 (SUB1 (LENGTH F2))))
              (T (SETQ F4
                       (APPEND (QUOTE (| -))
```



```
                               F2
                               (QUOTE (| -))))
                 (SETQ F3 (LENGTH F2))))
   LF5: (SETQ F2 (LIST (CAR F1)))
   LF6: (SETQ F1 (CDR F1))
        (COND ((NULL F1)
               (SETQ NODE
                     (READLIST
                      (APPEND F4
                              (LIST (QUOTE /[))
                              F2
                              (LIST (QUOTE /])))))
               (SETQ F3 (*PLUS F3 3))
               (GO LF3:))
              (T (SETQ F2
                       (APPEND F2
                               (QUOTE (/,))
                               (LIST (CAR F1))))
                 (SETQ F3 (*PLUS F3 2))
                 (GO LF6:)))
   LF4: (SETQ F1 (EXPLODE NODE))
        (SETQ F2 (CAR F1))
        (SETQ F3 (LENGTH F1))
   LF1: (COND ((AND (EQ F2 (QUOTE //)) (EQ F3 2))
               (SETQ F3 1))
              ((EQ F2 (QUOTE $))
               (SETQ NODE
                     (READLIST
                      (APPEND (QUOTE (<))
                              (CDR F1)
                              (QUOTE (>)))))
               (SETQ F3 (ADD1 F3)))
              ((EQ F2 (QUOTE !))
               (SETQ NODE
                     (READLIST
                      (APPEND (QUOTE (| _))
                              (CDR F1)
                              (QUOTE (| _)))))
               (SETQ F3 (SUB1 F3))))
   LF3: (PRINC NODE)
        (RETURN (*DIF TLEFT2 F3))))
EXPR)

(DEFPROP FIXN
 "TAKES VALUE FIELD IN INTERNAL FORM AND CONVERTS TO
OUTPUT FORM WITH RUNOFF CONTROL CHARACTERS."
COMMENT)

(DEFPROP LINK
 (LAMBDA(SPL PL)
  (PROG (LBL LSPL)
        (SETQ LSPL (EVAL SPL))
```

```
        (COND ((EQ LSPL (LLINK (EVAL (FATHER LSPL))))
               (RPLACA (CDR (EVAL (FATHER LSPL))) PL))
              ((EQ LSPL (RLINK (EVAL (FATHER LSPL))))
               (RPLACA (CDDR (EVAL (FATHER LSPL))) PL)))
        (COND ((NULL PL) (RETURN NIL)))
        (SETQ LBL (GENSYM))
        (SET LBL (EVAL (FATHER LSPL)))
        (RPLACA (CDDDR PL) LBL)))
EXPR)

(DEFPROP LINK
 "INSERTS FATHER LINKS IN THE TREE RESULTING
FROM REPLACING WHAT MATCHES THE MATCH TREE BY THE
SUBSTITUTION TREE."
COMMENT)

(DEFPROP LLINK
 (LAMBDA (X) (CADR X))
EXPR)

(DEFPROP LLINK
 "RETURNS LEFTLINK FIELD OF CURRENT TREE NODE."
COMMENT)

(DEFPROP MAKLST
 (LAMBDA(MKL1 MKL2)
  (COND ((NULL MKL1) MKL2)
        ((NULL MKL2) MKL1)
        ((OR (ATOM MKL1) (EQUAL (STRIP MKL1) MKL1))
         (COND ((ATOM MKL2) (LIST MKL1 MKL2))
               (T (APPEND (NCONS MKL1) MKL2))))
        (T
         (COND ((ATOM MKL2) (APPEND MKL1 (LIST MKL2)))
               (T (APPEND MKL1 MKL2))))))
EXPR)

(DEFPROP MAKLST
 "CREATES A LIST OF NODE AND THREAD MARKER ELEMENTS."
COMMENT)

(DEFPROP MARKT
 (LAMBDA(SUBT TLIST)
  (PROG (FROM TO STORE THP THX X)
        (SETQ X SUBT)
   MORE:
        (COND ((NULL TLIST) (RETURN T))
              (T (SETQ FROM (SUBNODE (CAAR TLIST)))
                 (SETQ TO (SUBNODE (CADAR TLIST)))))
        (SETQ SUBT X)
        (SETQ TLIST (CDR TLIST))
   WALK1:
        (WALK (QUOTE SUBT) (QUOTE STORE))
```



```
        (SETQ THX (VAL SUBT))
        (NOFUN (QUOTE MARKT1))
        (COND ((NULL SUBT) (RETURN NIL))
              ((AND (NOT (ATOM THX))
                    (NOT (ATOM (CDR THX)))
                    (NOT (ATOM (CADR THX)))
                    (EQ (CAADR THX) (QUOTE TH))
                    (EQUAL (CAR THX) TO))
               (SETQ THP (GENSYM))
               (RPLACA
                SUBT
                (APPEND
                 (VAL SUBT)
                 (NCONS (LIST (QUOTE TH) THP (QUOTE TO))))))
              ((EQUAL THX TO)
               (SETQ THP (GENSYM))
               (RPLACA
                SUBT
                (LIST (VAL SUBT)
                      (LIST (QUOTE TH) THP (QUOTE TO)))))
              (T (GO WALK1:)))
        (SETQ STORE NIL)
        (SETQ SUBT X)
   WALK2:
        (WALK (QUOTE SUBT) (QUOTE STORE))
        (SETQ THX (VAL SUBT))
        (NOFUN (QUOTE MARKT2))
        (COND ((NULL SUBT) (RETURN NIL))
              ((AND (NOT (ATOM THX))
                    (NOT (ATOM (CDR THX)))
                    (NOT (ATOM (CADR THX)))
                    (EQ (CAADR THX) (QUOTE TH))
                    (EQUAL (CAR THX) FROM))
               (RPLACA
                SUBT
                (APPEND
                 (VAL SUBT)
                 (NCONS
                  (LIST (QUOTE TH) THP (QUOTE FROM)))))
               (GO MORE:))
              ((EQUAL THX FROM)
               (RPLACA
                SUBT
                (LIST (VAL SUBT)
                      (LIST (QUOTE TH) THP (QUOTE FROM))))
               (GO MORE:))
              (T (GO WALK2:)))))
EXPR)

(DEFPROP MARKT
 "ASSOCIATES THREAD MARKERS WITH VALUE FIELDS OF
APPROPRIATE NODES."
```

```
COMMENT)

(DEFPROP MATCH
 (LAMBDA(TREE STK1 PTN STK2)
  (PROG (MTX T1 T2 T1B)
        (WALK TREE STK1)
        (WALK PTN STK2)
        (SETQ T1 (EVAL TREE))
        (SETQ T2 (EVAL PTN))
        (COND ((NOT (NULL T2)) (SETQ MTX (VAL T2)))
              (T (COND ((NOT (NULL T1)) (RETURN NIL)))))
        (COND
         ((NULL T1)
          (COND
           ((NULL T2) (RETURN T))
           (T
            (COND
             ((AND (NOT (ATOM MTX))
                   (EQ (CAR MTX) (QUOTE ARB)))
              (RETURN (ARB TREE STK1 PTN STK2)))
             (T (RETURN NIL))))))
         (T (SETQ T1B (STRIP (VAL T1)))))
        (COND
         ((EQ T1B MTX)
          (COND
           ((NOT (EQUAL (VAL T1) T1B))
            (SETQ MSTK
                  (CONS (LIST (LIST T1B) (VAL T1)) MSTK))))
          (RETURN T))
         (T
          (COND
           ((NOT (ATOM MTX))
            (SETQ FF (NANODE TREE STK1 PTN STK2))
            (NOFUN (QUOTE MATCH))
            (RETURN FF))
           (T (RETURN NIL)))))))
EXPR)

(DEFPROP MATCH
 "CONTROLS NODE-BY-NODE MATCHING OF
PARSE AND MATCH TREES."
COMMENT)

(DEFPROP MIN
 (LAMBDA(MT1 MST1 MT2 MST2)
  (PROG (PST1 PST2 MX MY MFLAG)
        (SETQ PST1 (EVAL MST1))
        (SETQ PST2 (EVAL MST2))
        (SETQ MFLAG (MIN1 MT1 (QUOTE MX) MT2 (QUOTE MY)))
        (NOFUN (QUOTE MIN))
        (COND ((NULL MFLAG) (RETURN NIL))
              ((NULL (SEARCH1 (CAAR MSTK) (CDR MSTK)))
```



```
               (GO DONE:))
              (T (PRINT
                   (QUOTE
                    (ERROR: TWO MINS WITH SAME SUBSCRIPT)))
                 (SETQ MSTK (POP MSTK))
                 (RETURN NIL)))
   DONE:
        (COND
         ((NOT (NULL MX)) (SET MST1 (*APPEND MX PST1))))
        (COND
         ((NOT (NULL MY)) (SET MST2 (*APPEND MY PST2))))
        (SETQ PFLAG 2)
        (RETURN T)))
EXPR)

(DEFPROP MIN
 (LAMBDA (L) (*EXPAND L (QUOTE *MIN)))
MACRO)

(DEFPROP MIN
 "SETS UP FOR DOING MIN NODE MATCH WITH
SOURCE TREE MT1 AND MATCH TREE MT2 AND CORRESPONDING
MATCH STACKS MST1 AND MST2.  CALLS MIN1 TO DO MATCH
AND ADJUSTS STACK APPROPRIATELY AFTER MATCH."
COMMENT)

(DEFPROP MIN1
 (LAMBDA(M1T1 M1ST1 M1T2 M1ST2)
  (PROG (PT1 PT2
             P2T1
             P2T2
             TARY1
             TARY2
             TARY3
             P2T1B
             COUNT
             PSAVE)
        (SETQ COUNT FFLAG)
        (SETQ PSAVE PFLAG)
        (SETQ PT1 (EVAL M1T1))
        (SETQ PT2 (EVAL M1T2))
        (COND ((NULL PT_) (RETURN NIL))
              ((EQ (VAL PT_) (QUOTE *NIL)) (RETURN NIL)))
   L5:  (WALK M1T2 M1ST2)
        (SETQ P2T2 (EVAL M1T2))
        (COND ((NULL P2T2) (RETURN NIL))
              ((EQ (VAL P2T2) (QUOTE *NIL)) (GO L5:))
              (T (GO L7:)))
   L6:  (WALK M1T1 M1ST1)
   L7:  (SETQ P2T1 (EVAL M1T1))
        (COND ((NULL P2T1) (RETURN NIL))
              ((EQ (VAL P2T1) (QUOTE *NIL)) (GO L6:))
```

```
      (T (SETQ P2T1B (STRIP (VAL P2T1)))))
(COND
 ((EQ P2T1B (VAL P2T2))
  (COND
   ((EQ FFLAG 0)
    (SETQ
     MSTK
     (CONS
      (LIST PT2 PT1 P2T1 NIL NIL COUNT ST STP PSAVE)
      MSTK))
    (SET
     M1ST1
     (PUSH
      (PUSH
       NIL
       (LIST (VAL PT1) NIL (RLINK PT_) NIL NIL NIL))
      (LIST (VAL P2T1)
            (LLINK P2T1)
            NIL
            NIL
            NIL
            NIL)))
    (SET
     M1ST2
     (PUSH
      (PUSH
       NIL
       (LIST (VAL PT2) NIL (RLINK PT2) NIL NIL NIL))
      (LIST (VAL P2T2)
            (LLINK P2T2)
            NIL
            NIL
            NIL
            NIL)))
    (RETURN T))
   (T (SETQ FFLAG (SUB1 FFLAG)) (GO L6:))))
 ((ATOM (VAL P2T2)) (GO L6:))
 (T (SETQ TARY1 (CAR (VAL P2T2)))
    (COND
     ((OR (EQ TARY1 (QUOTE ARB))
          (EQ TARY1 (QUOTE MIN))
          (EQ TARY1 (QUOTE BND)))
      (PRINT
       (LIST (QUOTE ERROR:)
             TARY1
             (QUOTE CANNOT)
             (QUOTE FOLLOW)
             (CAR (VAL PT2))))
      (RETURN NIL))
     (T (SETQ
         MSTK
         (CONS
```



```
          (LIST PT2
                PT1
                P2T1
                NIL
                NIL
                COUNT
                ST
                STP
                PSAVE)
          MSTK))
        (SETQ TARY2 (EXPLODEC P2T1B))
        (SETQ TARY3 (CAR TARY2))
        (COND
         ((EQ TARY1 (QUOTE SYM))
          (COND
           ((EQ P2T1B (QUOTE *NIL))
            (SETQ MSTK (POP MSTK))
            (GO L6:))
           (T (SETQ
               MSTK
               (CONS (LIST P2T2 (VAL P2T1)) MSTK))
              (COND
               ((BAKMAT) (GO L65:))
               (T (SETQ MSTK (POP MSTK))
                  (GO L6:))))))
         ((EQ TARY1 (QUOTE TERM))
          (COND
           ((OR (EQ TARY3 (QUOTE !))
                (EQ TARY3 (QUOTE $))
                (EQ P2T1B (QUOTE *NIL)))
            (SETQ MSTK (POP MSTK))
            (GO L6:))
           (T (SETQ
               MSTK
               (CONS (LIST P2T2 (VAL P2T1)) MSTK))
              (COND
               ((BAKMAT) (GO L65:))
               (T (SETQ MSTK (POP MSTK))
                  (GO L6:))))))
         ((EQ TARY1 (QUOTE NON))
          (COND
           ((EQ TARY3 (QUOTE $))
            (SETQ
             MSTK
             (CONS (LIST P2T2 (VAL P2T1)) MSTK))
            (COND
             ((BAKMAT) (GO L65:))
             (T (SETQ MSTK (POP MSTK)) (GO L6:))))
           (T (SETQ MSTK (POP MSTK)) (GO L6:))))
         ((AND (EQ TARY1 (QUOTE IL))
               (EQ TARY3 (QUOTE G))
               (NOT (NULL (CDR TARY2)))
```

```
                    (NUMBERP (READLIST (CDR TARY2))))
                (SETQ MSTK
                      (CONS (LIST P2T2 (VAL P2T1)) MSTK))
                (COND
                 ((BAKMAT) (GO L65:))
                 (T (SETQ MSTK (POP MSTK)) (GO L6:))))
               ((SCRIPT M1T1 M1T2) (GO L65:))
               (T (SETQ MSTK (POP MSTK)) (GO L6:)))))))
  L65: (COND
        ((EQ FFLAG 0)
         (SET
          M1ST1
          (PUSH
           (PUSH
            NIL
            (LIST (VAL PT1) NIL (RLINK PT1) NIL NIL NIL))
           (LIST (VAL P2T1) (LLINK P2T1) NIL NIL NIL NIL)))
         (NOFUN (QUOTE MIN1))
         (SET
          M1ST2
          (PUSH
           (PUSH
            NIL
            (LIST (VAL PT2) NIL (RLINK PT2) NIL NIL NIL))
           (LIST (VAL P2T2) (LLINK P2T2) NIL NIL NIL NIL)))
         (NOFUN (QUOTE MIN1))
         (RETURN T))
        (T (SETQ FFLAG (SUB1 FFLAG))
           (MPOP (VAL PT2))
           (GO L6:)))))
EXPR)

(DEFPROP MIN1
 "DOES MIN AND BND MATCH."
COMMENT)

(DEFPROP MPOP
 (LAMBDA(ITEM)
  (COND ((NULL MSTK) NIL)
        ((EQUAL ITEM (CAAAR MSTK)) (SETQ MSTK (POP MSTK)) T)
        (T (SETQ MSTK (POP MSTK)) (MPOP ITEM))))
EXPR)

(DEFPROP MPOP
 "POPS MATCH STACK UNTIL ITEM IS REMOVED AND RETURNS TRUE
OR UNTIL STACK IS EMPTY AND RETURNS NIL."
COMMENT)

(DEFPROP NANODE
 (LAMBDA(NTREE NSTK1 NPTN NSTK2)
  (PROG (NT1 NT1B A B C REST)
        (SETQ NT1 (VAL (EVAL NTREE)))
```



```
        (SETQ B (CAR (VAL (EVAL NPTN))))
        (SETQ NT1B (STRIP NT1))
        (COND ((EQUAL NT1 NT1B) (SETQ REST NIL))
              (T (SETQ REST (CDR NT1))))
        (SETQ A (EXPLODEC NT1B))
        (SETQ C (CAR A))
        (COND ((EQ B (QUOTE SYM))
               (COND ((EQ NT1B (QUOTE *NIL)) (RETURN NIL))
                     (T (SETQ MSTK
                              (CONS (LIST (EVAL NPTN) NT1)
                                    MSTK))
                        (RETURN (BAKMAT)))))
              ((EQ B (QUOTE TERM))
               (NOFUN (QUOTE NANODE))
               (COND ((OR (EQ C (QUOTE #))
                          (EQ C (QUOTE $))
                          (EQ NT1 (QUOTE *NIL)))
                      (RETURN NIL))
                     (T (SETQ MSTK
                              (CONS (LIST (EVAL NPTN) NT1)
                                    MSTK))
                        (RETURN (BAKMAT)))))
              ((EQ B (QUOTE NON))
               (COND ((EQ C (QUOTE $))
                      (SETQ MSTK
                            (CONS (LIST (EVAL NPTN) NT1)
                                  MSTK))
                      (RETURN (BAKMAT)))
                     (T (RETURN NIL))))
              ((AND (EQ B (QUOTE IL))
                    (EQ C (QUOTE G))
                    (NOT (NULL (CDR A)))
                    (NUMBERP (READLIST (CDR A))))
               (SETQ MSTK
                     (CONS (LIST (EVAL NPTN) NT1) MSTK))
               (RETURN (BAKMAT)))
              ((OR (EQ B (QUOTE ARB))
                   (EQ B (QUOTE MIN))
                   (EQ B (QUOTE BND)))
               (SETQ GG
                     ((EVAL (QUOTE B)) NTREE
                                       NSTK1
                                       NPTN
                                       NSTK2))
               (NOFUN (QUOTE NANODE))
               (RETURN GG))
              (T (RETURN (SCRIPT NTREE NPTN))))))
EXPR)

(DEFPROP NANODE
 "CONTROLS MATCHING FOR ALL SUBSCRIPTED NODES."
COMMENT)
```

```
(DEFPROP NOFUN
 (LAMBDA (NFF) NIL)
EXPR)

(DEFPROP NOFUN
 "THIS FUNCTION IS USED TO HAVE SOMETHING TO BREAK
ON WHILE DEBUGGING.  IT PERFORMS NO FUNCTION."
COMMENT)

(DEFPROP POP
 (LAMBDA (STAC) (CDR STAC))
EXPR)

(DEFPROP POP
 "POPS TOP ELEMENT FROM STACK."
COMMENT)

(DEFPROP PUSH
 (LAMBDA (STAC ITEM) (CONS ITEM STAC))
EXPR)

(DEFPROP PUSH
 "PUSHES ITEM ONTO STACK."
COMMENT)

(DEFPROP REPLACE
 (LAMBDA(SUBST)
  (PROG (SP SR STS VL P Q VALSP REST VALSPB)
        (SETQ SP (BUILD SUBST))
        (COND
         ((NOT (NULL TLST))
          (COND
           ((NOT (MARKT SP TLST))
            (PRINT (QUOTE (ERROR IN THREAD LIST)))
            (RETURN NIL)))))
        (SETQ SR SP)
   NXT: (WALK (QUOTE SP) (QUOTE STS))
        (COND
         ((NULL SP)
          (LINK (QUOTE ROOT1) (LLINK SR))
          (COND
           ((NOT (NULL (RLINK ROOT1)))
            (RPLACA (CDDR (LLINK SR)) (RLINK ROOT1))
            (SETQ Q (GENSYM))
            (SET Q (LLINK SR))
            (RPLACA (CDDDR (RLINK ROOT1)) Q)))
          (RETURN T))
         (T (SETQ VALSP (VAL SP))))
        (COND
         ((ATOM VALSP)
          (SETQ P (SEARCH VALSP))
```



```
          (COND ((NULL P) (GO NXT:))
                (T (RPLACA SP (CADR P)) (GO NXT:))))
         (T (SETQ VALSPB (STRIP VALSP))
            (COND
             ((NOT (EQUAL VALSPB VALSP))
              (SETQ REST (CDR VALSP)))
             (T (SETQ REST NIL)))))
        (COND
         ((ATOM VALSPB)
          (SETQ P (SEARCH VALSPB))
          (COND
           ((NULL P) (GO NXT:))
           (T (RPLACA SP (APPEND (CADR P) REST))
              (GO NXT:))))
         (T (SETQ VL (CAR VALSPB))))
        (COND
         ((OR (EQ VL (QUOTE SYM))
              (EQ VL (QUOTE NON))
              (EQ VL (QUOTE TERM)))
          (SETQ P (SEARCH VALSPB))
          (NOFUN (QUOTE REPLACE))
          (COND
           ((NULL P) (RETURN NIL))
           (T (RPLACA SP (MAKLST (CADR P) REST))
              (GO NXT:))))
         ((EQ VL (QUOTE IL))
          (SETQ P (SEARCH VALSPB))
          (NOFUN (QUOTE REPLACE))
          (COND
           ((NULL P)
            (SETQ Q (GENSYM))
            (SETQ
             MSTK
             (CONS
              (LIST (LIST (COPY VALSPB) NIL NIL NIL NIL NIL)
                    Q)
              MSTK))
            (COND ((NULL REST) (RPLACA SP Q))
                  (T (RPLACA VALSP Q))))
           (T (RPLACA SP (MAKLST (CADR P) REST))))
          (GO NXT:))
         ((OR (EQ VL (QUOTE ARB))
              (EQ VL (QUOTE MIN))
              (EQ VL (QUOTE BND)))
          (SETQ P (SEARCH VALSPB))
          (NOFUN (QUOTE REPLACE))
          (COND
           ((NULL P) (RETURN NIL))
           ((APLACE (QUOTE SP) (QUOTE STS) P VL) (GO NXT:))
           (T (RETURN NIL))))
         (T (NOFUN (QUOTE REPLACE))
            (COND
```

```
            ((SPLACE VALSPB)
             (SETQ P (SEARCH VALSPB))
             (COND
              ((AND (NULL REST) P) (RPLACA SP (CADR P)))
              ((AND P REST)
               (RPLACA SP (APPEND (CADR P) REST))))
             (GO NXT:))
            (T (RETURN NIL)))))))
EXPR)

(DEFPROP REPLACE
 "BUILDS SUBSTITUTION TREE AND INSERTS IN SOURCE
TREE IN PLACE OF MATCHED NODES."
COMMENT)

(DEFPROP RLINK
 (LAMBDA (X) (CADDR X))
EXPR)

(DEFPROP RLINK
 "RETURNS RIGHTLINK FIELD OF CURRENT TREE NODE."
COMMENT)

(DEFPROP RSPND
 (LAMBDA NIL (COND ((EQ (READ) (QUOTE Y)) T) (T NIL)))
EXPR)

(DEFPROP RSPND
 "TAKES TERMINAL INPUT TO DETERMINE WHETHER TO
PRINT CURRENT SOURCE TREE."
COMMENT)

(DEFPROP SCRIPT
 (LAMBDA(ST1 ST2)
  (PROG (SS1 SS2 SS1B C1 C2 CTR PS XX YY THRD)
        (SETQ CTR 0)
        (SETQ SS1 (VAL (EVAL ST1)))
        (SETQ SS2 (VAL (EVAL ST2)))
        (SETQ SS1B (STRIP SS1))
        (COND ((EQUAL SS1 SS1B) (SETQ THRD NIL))
              (T (SETQ THRD T)))
        (COND
         ((ATOM SS1B) (RETURN NIL))
         ((AND (NOT (ATOM (CAR SS1B)))
               (NOT (ATOM (CAR SS2))))
          (SETQ SS1B (CAR SS1B))
          (SETQ SS2 (CAR SS2))
          (GO TEST:))
         ((NOT (EQ (CAR SS1B) (CAR SS2))) (RETURN NIL)))
        (SETQ SS1B (CADR SS1B))
        (SETQ SS2 (CADR SS2))
        (GO TEST:)
```



```
NXSUB:
      (SETQ SS1B (CDR SS1B))
      (SETQ SS2 (CDR SS2))
TEST:
      (COND
       ((OR (NULL SS1B) (NULL SS2))
        (COND
         ((AND (NULL SS1B) (NULL SS2))
          (COND
           (THODE
            (COND
             (THRD
              (SETQ
               MSTK
               (CONS (LIST (LIST (CAR SS1)) SS1)
                     MSTK))))))
           (RETURN T))
          (T (GO CLEAR:)))))
       (SETQ C1 (CAR SS1B))
       (SETQ C2 (CAR SS2))
       (COND
        ((NOT (ATOM C2))
         (SETQ XX (CADR C2))
         (SETQ PS (SEARCH (LIST XX)))
         (COND
          ((NULL PS)
           (SETQ YY C1)
           (COND
            ((EQ (CAR C2) (QUOTE *PLUS))
             (SETQ YY (*DIF YY (CADDR C2))))
            (T (SETQ YY (*PLUS YY (CADDR C2)))))
           (SETQ MSTK
                 (CONS (LIST (LIST (LIST XX)) YY) MSTK))
           (SETQ CTR (ADD1 CTR))
           (GO NXSUB:))
          (T
           (COND
            ((EQ (CAR C2) (QUOTE *DIF))
             (COND
              ((EQ C1
                   (EVAL
                    (LIST (QUOTE *DIF)
                          (CADR PS)
                          (CADDR C2))))
               (GO NXSUB:))
              (T (GO CLEAR:))))
            ((EQ (CAR C2) (QUOTE *PLUS))
             (COND
              ((EQ C1
                   (EVAL
                    (LIST (QUOTE *PLUS)
                          (CADR PS)
```

```
                        (CADDR C2))))
                (GO NXSUB:))
               (T (GO CLEAR:))))
             (T (GO CLEAR:))))))
        ((NUMBERP C2)
         (COND ((EQ C1 C2) (GO NXSUB:)) (T (GO CLEAR:))))
        (T (SETQ PS (SEARCH (LIST C2)))
           (COND
            ((NULL PS)
             (SETQ MSTK
                   (CONS (LIST (LIST (LIST C2)) C1) MSTK))
             (SETQ CTR (ADD1 CTR))
             (GO NXSUB:))
            (T
             (COND ((EQ C1 (CADR PS)) (GO NXSUB:))
                   (T (GO CLEAR:)))))))))

   CLEAR:
        (COND
         ((ZEROP CTR) (RETURN NIL))
         (T (SETQ MSTK (POP MSTK))
            (SETQ CTR (SUB1 CTR))
            (GO CLEAR:)))))
EXPR)

(DEFPROP SCRIPT
 "MATCHES SUBSCRIPTED NODES OTHER THAN ARB, MIN, BND, IL,
NON, TERM, AND SYM."
COMMENT)

(DEFPROP SEARCH
 (LAMBDA (ITM) (SEARCH1 ITM MSTK))
EXPR)

(DEFPROP SEARCH
 "SUPPLIES STACK TO SEARCH FOR ITEM AND CALLS SEARCH1."
COMMENT)

(DEFPROP SEARCH1
 (LAMBDA(ITEM1 MSTK1)
  (COND ((NULL MSTK1) NIL)
        ((EQUAL ITEM1 (CAAAR MSTK1)) (CAR MSTK1))
        (T (SEARCH1 ITEM1 (CDR MSTK1)))))
EXPR)

(DEFPROP SEARCH1
 "SEARCHES STACK MSTK1 FOR ITEM1,
RETURNS POINTER TO ITEM1 IF ITEM1 FOUND, NIL OTHERWISE."
COMMENT)

(DEFPROP SPLACE
 (LAMBDA(SACE1)
  (COND ((NULL (CDR SACE1)) (SUBS (CAR SACE1)))
```



```
        (T (SUBS (CADR SACE1)))))
EXPR)

(DEFPROP SPLACE
 "AFTER SUBSTITUTION TREE IS INSERTED IN SOURCE TREE,
SPLACE CALLS SUBS, SUBSCRIPT EVALUATION ROUTINE."
COMMENT)

(DEFPROP STRIP
 (LAMBDA(VALFD)
  (COND (THODE
         (COND ((AND (NOT (ATOM VALFD))
                     (NOT (ATOM (CDR VALFD)))
                     (NOT (ATOM (CADR VALFD)))
                     (EQ (CAADR VALFD) (QUOTE TH)))
                (CAR VALFD))
               (T VALFD)))
        (T VALFD)))
EXPR)

(DEFPROP STRIP
 "TAKES A VALUE FIELD WITH POSSIBLE THREAD MARKERS
AND RETURNS A VALUE FIELD WITH THE MARKERS STRIPPED OFF."
COMMENT)

(DEFPROP SUBC
 (LAMBDA(WC ZC PLC DIC)
  (PROG NIL
        (COND
         ((NOT (NULL (EVAL PLC)))
          (SET PLC NIL)
          (COND
           ((AND (NUMBERP WC) (NUMBERP ZC))
            (RETURN (EVAL (LIST (QUOTE *PLUS) ZC WC))))
           ((NUMBERP ZC)
            (RETURN (LIST (QUOTE *PLUS) WC ZC)))
           (T (RETURN (LIST (QUOTE *PLUS) ZC WC)))))
         ((NOT (NULL (EVAL DIC)))
          (SET DIC NIL)
          (COND
           ((AND (NUMBERP WC) (NUMBERP ZC))
            (RETURN (EVAL (LIST (QUOTE *DIF) ZC WC))))
           ((NUMBERP ZC)
            (RETURN
             (LIST (QUOTE *PLUS)
                   (EVAL (LIST (QUOTE *DIF) 0 WC))
                   ZC)))
           (T (RETURN (LIST (QUOTE *DIF) ZC WC)))))
         (T (RETURN WC)))))
EXPR)

(DEFPROP SUBC
```

```
 "EVALUATES SUBSCRIPT EXPRESSION IN SUBSCRIPTED NODE."
COMMENT)

(DEFPROP SUBNODE
 (LAMBDA(L)
  (PROG (PL DI W X Y Z)
        (SETQ X (EXPLODEC L))
        (COND ((NOT (EQ (CAR X) (QUOTE <))) (RETURN L))
              (T (COND ((NULL (CDR X)) (RETURN L)))))
   L2:  (SETQ X (CDR X))
        (COND ((EQ (CAR X) (QUOTE >))
               (SETQ W (READLIST W))
               (SETQ Y
                     (APPEND
                      Y
                      (LIST
                       (SUBC W Z (QUOTE PL) (QUOTE DI)))))
               (COND ((NULL (CDR X)) (RETURN (CONS Y NIL)))
                     (T
                      (RETURN
                       (LIST (READLIST (CDR X)) Y)))))
              ((EQ (CAR X) (QUOTE /,))
               (SETQ W (READLIST W))
               (SETQ Y
                     (APPEND
                      Y
                      (LIST
                       (SUBC W Z (QUOTE PL) (QUOTE DI)))))
               (SETQ W NIL)
               (GO L2:))
              ((EQ (CAR X) (QUOTE +))
               (SETQ PL T)
               (SETQ Z (READLIST W))
               (SETQ W NIL)
               (GO L2:))
              ((EQ (CAR X) (QUOTE -))
               (SETQ DI T)
               (SETQ Z (READLIST W))
               (SETQ W NIL)
               (GO L2:))
              (T (SETQ W (APPEND W (LIST (CAR X))))
                 (GO L2:)))))
EXPR)

(DEFPROP SUBNODE
 "PARSES SUBSCRIPT EXPRESSIONS."
COMMENT)

(DEFPROP SUBS
 (LAMBDA(ACE1)
  (PROG (ACE2 PB)
   SUBS1:
```



```
        (SETQ ACE2 (CAR ACE1))
        (COND ((NOT (ATOM ACE2))
               (COND ((SUBS (CDR ACE2))
                      (RPLACA ACE1 (EVAL ACE2)))
                     (T (RETURN NIL))))
              ((NOT (NUMBERP ACE2))
               (SETQ PB (SEARCH (LIST ACE2)))
               (COND ((NULL PB) (RETURN NIL))
                     (T (RPLACA ACE1 (CADR PB))))))
        (SETQ ACE1 (CDR ACE1))
        (COND ((NULL ACE1) (RETURN T)))
        (GO SUBS1:)))
EXPR)

(DEFPROP SUBS
 "REPLACES SUBSCRIPT EXPRESSIONS BY ACTUAL SUBSCRIPT
VALUES."
COMMENT)

(DEFPROP TAB
 (LAMBDA(MARG1 TLEFT1)
  (PROG NIL
   RIGHT:
        (COND ((*GREAT MARG1 (*DIF (LINELENGTH NIL) TLEFT1))
               (PRINC (ASCII 40))
               (SETQ TLEFT1 (SUB1 TLEFT1))
               (GO RIGHT:))
              ((EQ MARG1 (*DIF (LINELENGTH NIL) TLEFT1))
               (RETURN TLEFT1))
              (T (TERPRI) (SETQ TLEFT1 1750) (GO RIGHT:)))))
EXPR)

(DEFPROP TAB
 "SPACES TO APPROPRIATE COLUMN FOR PRINTING OF VALUE FIELD
BY TPR."
COMMENT)

(DEFPROP THREAD
 (LAMBDA NIL
  (PROG (THR WLKSTK TOLST FRMLST TOPT TOPF GENS VALT)
        (SETQ THR TR)
   MOREN:
        (WALK (QUOTE THR) (QUOTE WLKSTK))
        (COND ((NULL THR) (GO FROMS:)))
        (SETQ VALT (VAL THR))
        (COND ((EQUAL VALT (STRIP VALT)) (GO MOREN:))
              (T (SETQ VALT (CDR VALT))))
   MORET:
        (NOFUN (QUOTE THREAD1))
        (COND ((NULL VALT) (RPLACA THR (CAAR THR))
                           (GO MOREN:))
              ((EQ (CADDAR VALT) (QUOTE TO))
```

```
                (SETQ TOLST
                      (APPEND
                       TOLST
                       (NCONS
                        (LIST (CADAR VALT)
                              (CONS (CAAR THR) (CDR THR))))))
                (SETQ VALT (CDR VALT))
                (GO MORET:))
               ((EQ (CADDAR VALT) (QUOTE FROM))
                (SETQ FRMLST
                      (APPEND
                       FRMLST
                       (NCONS
                        (LIST (CADAR VALT)
                              (CONS (CAAR THR) (CDR THR))))))
                (SETQ VALT (CDR VALT))
                (GO MORET:)))
FROMS:
     (NOFUN (QUOTE THREAD2))
     (SETQ WLKSTK NIL)
     (SETQ THR TR)
     (WALK (QUOTE THR) (QUOTE WLKSTK))
NXTF:
     (NOFUN (QUOTE THREAD3))
     (COND ((NULL FRMLST) (GO TOS:))
           (T (SETQ TOPF (CAR FRMLST))))
     (SETQ VALT (CADR TOPF))
     (SETQ GENS (CAR TOPF))
NXTN:
     (COND ((EQUAL THR VALT)
            (COND ((NULL (CAR (CDDDDR THR)))
                   (RPLACA (CDDDDR THR) GENS)
                   (SETQ FRMLST (CDR FRMLST))
                   (GO NXTF:))
                  ((NULL (CADR (CDDDDR THR)))
                   (RPLACA (CDR (CDDDDR THR)) GENS)
                   (SETQ FRMLST (CDR FRMLST))
                   (GO NXTF:))
                  (T (PRINT
                      (LIST (QUOTE
                             (TOO MANY THREADS FROM))
                            (VAL THR)))
                     (RETURN NIL))))
           (T (WALK (QUOTE THR) (QUOTE WLKSTK))
              (COND
               ((NULL THR)
                (PRINT
                 (QUOTE (TOO MANY THREADS FOR TREE)))
                (RETURN NIL)))
              (GO NXTN:)))
TOS: (NOFUN (QUOTE THREAD4))
     (COND ((NULL TOLST) (RETURN T))
           (T (SETQ TOPT (CAR TOLST))
              (SET (CAR TOPT) (CADR TOPT))
              (SETQ TOLST (CDR TOLST))
              (GO TOS:))))))
EXPR)

(DEFPROP THREAD
 "INSERTS THREADS IN SUBSTITUTION TREE."
COMMENT)

(DEFPROP THREADPRINT
 (LAMBDA(TREEP)
  (PROG (STORE)
   NEXT:(COND
         ((WALK (QUOTE TREEP) (QUOTE STORE))
          (COND
           ((NULL (CAR (CDDDDR TREEP))) (GO NEXT:))
           (T (PRINC (QUOTE THREAD))
              (PRINC (ASCII 40))
              (PRINC (QUOTE FROM))
              (PRINC (ASCII 40))
              (FIXN (VAL TREEP) 1750)
              (PRINC (ASCII 40))
              (PRINC (QUOTE TO))
              (PRINC (ASCII 40))
              (FIXN (VAL (EVAL (CAR (CDDDDR TREEP)))) 1750)
              (TERPRI)
              (COND
               ((NULL (CADR (CDDDDR TREEP))) (GO NEXT:))
               (T (PRINC (QUOTE THREAD))
                  (PRINC (ASCII 40))
                  (PRINC (QUOTE FROM))
                  (PRINC (ASCII 40))
                  (FIXN (VAL TREEP) 1750)
                  (PRINC (ASCII 40))
                  (PRINC (QUOTE TO))
                  (PRINC (ASCII 40))
                  (FIXN (VAL (EVAL (CADR (CDDDDR TREEP))))
                        1750)
                  (TERPRI)
                  (GO NEXT:))))))
         (T (RETURN T)))))
EXPR)

(DEFPROP THREADPRINT
 "PRINTS THREAD FIELDS IN A TREE."
COMMENT)

(DEFPROP TPR
 (LAMBDA(TREE MARGN TLEFT)
  (SETQ TLEFT (TAB MARGN TLEFT))
  (COND ((NULL TREE) NIL)
```

```
        ((ATOM TREE) (SETQ TLEFT (FIXN TREE TLEFT)))
        (T (SETQ TLEFT (FIXN (STRIP (CAR TREE)) TLEFT))
           (COND
            ((CADDR TREE)
             (TPR (CADDR TREE)
                  (*DIF (ADD1 (LINELENGTH NIL)) TLEFT)
                  TLEFT)))
           (COND
            ((CADR TREE) (TPR (CADR TREE) HARGN TLEFT))))))
EXPR)

(DEFPROP TPR
 "PRINTS VALUE FIELDS OF TREE WITH BROTHERS ON SAME LEVEL
AND LEFTMOST SON IN SAME COLUMN AS FATHER."
COMMENT)

(DEFPROP TPRINT
 (LAMBDA(XTREE HARG)
  (PROG (LEFT) (SETQ LEFT 1750) (TPR XTREE 0 LEFT)))
EXPR)

(DEFPROP TPRINT
 "SETS OUTPUT LINE LENGTH AND CALLS TPR, TREE OUTPUT
ROUTINE."
COMMENT)

(DEFPROP VAL
 (LAMBDA (X) (CAR X))
EXPR)

(DEFPROP VAL
 "RETURNS VALUE FIELD OF CURRENT TREE NODE."
COMMENT)

(DEFPROP WALK
 (LAMBDA(TREE STACK)
  (PROG (TPTR STK)
        (SETQ TPTR (EVAL TREE))
        (SETQ STK (EVAL STACK))
        (NOFUN (QUOTE WALK))
        (COND ((NULL TPTR) (RETURN NIL))
              ((AND (NULL (VAL TPTR)) (NULL (LLINK TPTR)))
               (SET TREE NIL)
               (RETURN NIL))
              ((EQ (VAL TPTR) (QUOTE *NIL))
               (COND ((NULL STK) (SET TREE NIL)
                                 (RETURN NIL))
                     ((NULL (RLINK (CAR STK)))
                      (SET STACK (POP STK))
                      (SET TREE **NIL)
                      (RETURN **NIL))
                     (T (SET TREE (RLINK (CAR STK)))
```



```
                        (SET STACK (POP STK))
                        (RETURN (VAL (EVAL TREE))))))
              (T (COND ((EQ PFLAG 0)
                        (SETQ STK (PUSH STK TPTR)))
                       (T (SETQ PFLAG (SUB1 PFLAG))))
                 (SET STACK STK)
                 (COND ((NULL (LLINK TPTR))
                        (SET TREE **NIL)
                        (RETURN **NIL))
                       (T (SET TREE (LLINK TPTR))
                          (RETURN (VAL (LLINK TPTR))))))))))
EXPR)

(DEFPROP WALK
 "GETS NEXT NODE IN TREE IN A PREORDER
TRAVERSAL, UPDATING THE TRAVERSAL STACK."
COMMENT)
```

Appendix IV

A CROSS-REFERENCE BY SYNTACTIC CONSTRUCT FOR THE SEMANTIC SPECIFICATION

Below is a cross-reference table of selected Algol 60 syntactic constructs and the numbers of the semantic transformations which deal with them.

CONSTRUCT	TRANSFORMATION NUMBER
program	1-17, 41-44, 46, 132, 134, 141, 149, 152-158, 212, 218
declaration	5-67, 105-108, 110, 116, 131-150, 153, 154, 159-166, 175, 198-210
type	52, 53, 56-59, 132, 134, 135, 137, 139, 143-145
array	6-9, 14, 15, 54, 55, 60-67, 136, 138, 140, 146-148
switch	45-51, 105-108, 116
procedure	16-44, 150, 159-162, 175, 198-210
statement	1-4, 70-86, 92, 94, 96, 98, 174, 177-197, 213
assignment	84-86, 174
go to	74
procedure	92, 94, 96, 98, 177-197, 213
conditional	72, 73
for	75-83
expression	87-91, 100-126
arithmetic	87, 88, 90, 100-102, 104, 109-112, 114, 117-119, 121, 122, 124, 125
Boolean	89, 91, 103, 113, 115, 120, 123, 126
designational	105-108, 116
label	128-131, 133, 142, 153, 211, 215, 216
function designator	93, 95, 97, 99
variable	56, 58, 60-63, 74, 78-82, 87-89, 100, 107, 108, 110-113, 129, 159-174