A VERIFIED PROGRAM VERIFIER

by

Larry Calvin Ragland

May 1973

TR-18

This work was supported in part by a National Science Foundation Traineeship and a National Science Foundation Grant GJ-36424.

Technical Report No. 18
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
ABSTRACT

The dissertation describes the construction and verification of a program which generates the inductive assertion method verification conditions. The primary emphasis is on the verification of the program as this represents the first time a verifier has been subjected to a proof of correctness. The verifier is written in Nucleus and operates on programs written in Nucleus.

Nucleus is a programming language designed so that all programs in the language can be subjected to proofs by the inductive assertion method. This verifier is to be the foundation of a sequence of verification systems of increasing sophistication, where each system is used to aid in the verification of the next. The ultimate goal is a verified system which provides the user a full set of verification services.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. INTRODUCTION AND RELATED WORK</td>
<td>1</td>
</tr>
<tr>
<td>I.1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>I.2. Related Work</td>
<td>2</td>
</tr>
<tr>
<td>I.3. Summary of Chapters</td>
<td>5</td>
</tr>
<tr>
<td>II. THE NUCLEUS LANGUAGE</td>
<td>7</td>
</tr>
<tr>
<td>II.1. Introduction</td>
<td>7</td>
</tr>
<tr>
<td>II.2. Design Goals</td>
<td>7</td>
</tr>
<tr>
<td>II.3. Informal Description</td>
<td>8</td>
</tr>
<tr>
<td>II.4. Method of Formal Definition</td>
<td>14</td>
</tr>
<tr>
<td>II.4.1. Syntax</td>
<td>14</td>
</tr>
<tr>
<td>II.4.2. Semantics</td>
<td>21</td>
</tr>
<tr>
<td>III. THE INDUCTIVE ASSERTION METHOD FOR NUCLEUS</td>
<td>27</td>
</tr>
<tr>
<td>III.1. Introduction</td>
<td>27</td>
</tr>
<tr>
<td>III.2. Definitions of Correctness</td>
<td>29</td>
</tr>
<tr>
<td>III.3. Definition of Path</td>
<td>34</td>
</tr>
<tr>
<td>III.4. Informal Description of a Verification Condition</td>
<td>37</td>
</tr>
<tr>
<td>III.5. Verification Condition Terms for Partial Correctness</td>
<td>42</td>
</tr>
<tr>
<td>III.6. Verification Conditions for Partial Correctness</td>
<td>55</td>
</tr>
<tr>
<td>III.7. Verification Conditions for Total Correctness</td>
<td>73</td>
</tr>
<tr>
<td>IV. A NUCLEUS VERIFIER AND ITS PROOF</td>
<td>76</td>
</tr>
<tr>
<td>IV.1. Introduction</td>
<td>76</td>
</tr>
<tr>
<td>IV.2. Methodology</td>
<td>77</td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION AND RELATED WORK

1.1. Introduction

Several techniques have been developed for proving the correctness of computer programs and automatic systems have been built which implement these techniques. Thus, it is now possible to prove the correctness of a sizeable class of computer programs. Unfortunately, a correctness proof which is constructed with the aid of an automatic verification system does not guarantee that the program will always run properly. First, in order for the proof to be valid, the verification system must operate correctly. However, even if the program is absolutely correct, run-time correctness is achieved only if the intervening software functions properly. Therefore, the second requirement for a properly running program is a complete set of correct software.

This research is directed toward the first of these two requirements, a correct program verifier. The goal of this research is to construct and verify an inductive assertion method verification condition generator. This initial verified verifier can then be used in the verification of more sophisticated systems.

The construction and verification of a verification condition generator requires rigorously defined input and implementation languages, a sound theoretical development of the concept of verification
condition, and a solid strategy for the construction and proof of the
verifier. The verifier presented here is written in a programming
language called Nucleus (Chapter II) and operates on programs written
in Nucleus. The Nucleus programming language was designed for the
specific purpose of facilitating proofs of correctness for computer
programs. The Nucleus verification conditions (Chapter III) are
fully defined from the formal definition of the language and are
proved to imply correctness. The proof strategy (Chapter IV) employed
in the verification of the verifier is a unique combination of the
inductive assertion method and equivalence proofs. The portion of the
proof which is based on the inductive assertion method employs an
unproved verification condition generator written in Snobol4 [14].

I.2. Related Work

The inductive assertion method is one of the more successful
of the current methods for proving the correctness of computer programs.
It consists of attaching predicates to certain key points in the
program and showing that the predicates are true each time execution
reaches the point to which the predicate is attached. The basis for
the inductive assertion method was first presented by Floyd [4] with
essentially the same idea presented in a less formal paper by Naur [11]
who used the term "general snapshots" rather than "inductive assertions".
The formulation used in this study is an extension of the approach
presented by Good [5]. A recent and thorough presentation of the
inductive assertion method is contained in Elspas, et. al. [3].
London [10] lists thirteen automatic verification systems of various types which have been implemented. Nine of these are classified as being based on the theoretical foundation presented by Floyd [4], two systems are based on the axiomatic system of Hoare [8], one is based on a variation of Floyd's work formulated by Cooper [2], and one is based on Scott's Logic for Computable Functions (LCF) [12]. All of these systems operate on programs which are already written and restate the question of program correctness in terms of a set of verification conditions.

The first two experimental program proving systems were built by King [9] and Good [5]. These two systems have more than historical significance in that they introduce two basic approaches to verifier construction. King's system is totally automatic and Good's system employs man-machine interaction.

King's program verifier automates the entire inductive assertion method except for the choice of assertions. It operates on a subset of Algol which is restricted to integers, however, it does handle one-dimensional arrays. The assertions are manually supplied and take the form of extended boolean expressions. The verification conditions are automatically constructed and an automatic theorem prover which uses specialized techniques for integers then attempts to prove the verification conditions. This system has been successfully used on a number of small, but non-trivial, programs.

Good's system uses man-machine interaction to complete a
proof. His system also accepts programs from a subset of Algol. It
does not handle arrays or procedures, but does include declarations.
The assertions are manually supplied and can be any text string.
Verification conditions are generated automatically, but the proofs
must be manually supplied and can be any text string. The system
performs a variety of services for the user in the form of elaborate
record keeping. The system keeps track of which assertions need to be
verified along which paths, allows proofs to be modified, retrieves
assertions or proofs, and gives the complete or partial proof at the end.

All of the above systems mentioned operate on programs
which are already written. Snowdon [13] has built an interactive
system called PEARL (Program Elaboration And Refinement Language)
which aids in the construction of correct programs. PEARL provides the
programmer the capability of constructing and proving the correctness
of a program at a very general level in terms of abstract operations
and data types, and then continually refining it until all operations
and data types are reduced to a set of primitives. The PEARL system
allows compilation, execution, and correctness proofs of programs
which are not completely specified, with the programmer supplying
assistance during execution of incompletely specified operations.

The system to be described in this thesis does not attempt
to provide as many services for the user as the above systems. This
verifier is not intended to be a highly sophisticated system, but is
to be the first verified verifier in a sequence of systems, with each
system proved to be correct and each more sophisticated than the one before. The ultimate goal is a correct verifier which provides the user a full set of verification services.

I.3. Summary of Chapters

Chapter II introduces the Nucleus programming language. Nucleus is designed for the specific purpose of facilitating proofs about programs in the language and to facilitate verification of the Nucleus compiler and verifier. The formal definition of Nucleus in terms of transition networks and axioms forms the base on which the formulation of the inductive assertion method for Nucleus programs and the proof strategy for the verification of a Nucleus verifier are built.

Chapter III presents the theoretical foundation for the verification of Nucleus programs by the inductive assertion method. Partial and total correctness are defined and for each of these types of correctness, a set of verification conditions is defined. The set of language features covered by these verification conditions includes input and output operations. The sets of verification conditions are shown to be sufficient to prove each of the two types of correctness defined.

Chapter IV describes the construction, resulting structure, and verification of a Nucleus program which generates the verification conditions for Nucleus programs. The proof of this verifier employs a unique approach to program correctness which combines the inductive
assertion method and equivalence proofs. The resulting verifier provides a starting point for a sequence of verified verifiers of increasing sophistication.

Chapter V summarizes the results and findings of this research and contains suggestions for future efforts related to the further development of the verifier described in this paper.
CHAPTER II

THE NUCLEUS LANGUAGE

II.1. Introduction

This chapter presents a brief introduction to the Nucleus programming language and the method used to define it formally. The discussion of Nucleus is included to make the dissertation self-contained and because of the close relationship between the formal Nucleus definition and the proof of the verifier. More detailed descriptions of Nucleus and the method of definition are contained in [7].

II.2. Design Goals

Nucleus is designed for the specific purpose of facilitating proofs about computer programs. This objective is more clearly defined by the following design goals of Nucleus.

1. Inductive assertion provability. It must be possible to subject any Nucleus program to a proof of correctness using the inductive assertion method. Thus, the language provides a mechanism for stating the inductive assertions and is limited to features for which verification conditions can be constructed.

2. Verifier correctness. It must be possible to construct and prove the correctness of a verifier which generates verification conditions for Nucleus programs. In order for proofs constructed with the aid of a verifier to be valid, the verifier must be known to
operate properly. A verified verifier assures that this condition is met.

3. **Compiler correctness.** It must be possible to construct and prove the correctness of a compiler for Nucleus programs. Even a correct program will not run properly unless it is compiled correctly.

4. **Rigorous definition.** Both the syntax and semantics of Nucleus must be completely and rigorously defined. This requirement is implicit in the previous goals. In order for the verifier, the compiler, and other programs to be proved, the language must be completely specified.

II.3. Informal Description

Nucleus is a simple language, however, it does contain features which make it non-trivial. It contains input and output operations, parameterless procedures, one-dimensional arrays, a multi-way branch, and a built-in mechanism for stating assertions directly within programs.

**Programs.** The form of a Nucleus program is:

- declarations
- procedures
- **START** identifier

The declarations define the global simple variables and arrays of the program. All variables are global and must be declared to be of type INTEGER, BOOLEAN, or CHARACTER. Arrays have only one subscript with a lower bound of zero and an upper bound declared to be any non-negative integer constant. There may be any number of procedures and the identifier following **START** is the name of the procedure where execution begins.
Procedures. The form of a procedure is

    PROCEDURE identifier ;
    body
    EXIT ;

Procedures may be recursive but have neither parameters nor local variables. Parameterless procedures avoid the problem of parameter passage and represents one of the major temporary concessions for the sake of simplicity of inductive assertion proofs.

Bodies. A body is a sequence of statements and assertions, each of which is terminated by a semicolon. Assertions are made up of the reserved word, ASSERT, followed by any text not containing a semi-colon. Statements may be labelled by any number of label identifiers. Each label is followed by a colon and is local to the procedure in which it appears. The Nucleus statements are described below.

Assignment. leftside := expression.

The leftside can be any variable or array reference and the data type of leftside must match the data type of expression.

Go to. GO TO identifier.

The identifier must be the label of a statement in the procedure in which the go to appears.

If. IF boolean expression THEN body FI
    IF boolean expression THEN body ELSE body FI

The if statement is of the usual form except for the inclusion of "body" where a statement generally appears in other languages.

While. WHILE boolean expression DO body ELIHW.
The while statement is the usual loop control statement, again with "body" where a statement usually occurs.

Case. CASE integer expression OF alternatives ESAC
      CASE integer expression OF alternatives ELSE body ESAC

The case statement is the Nucleus multi-way branch mechanism. The alternatives are bodies which are preceded by numeric labels. When a case statement is encountered during execution, the integer expression is evaluated. If this value matches a numeric label of a body in the alternatives, then control passes to that body. If the value does not match a numeric label, then control passes to the next statement after the case for the first form and to the body following ELSE for the second form of the case statement. From the end of an alternative, control passes to the next statement after the case.

Read. READ array

The array of the read statement must be a type character array. The read statement accesses the standard input file which is composed of a sequence of records numbered 1,2,... Each record is either an end-of-file (eof) record or consists of a sequence of n Nucleus characters (n is the same for all records). The record size n is one of the implementation parameters of Nucleus. That is, a specific value for n is not specified in the formal definition, but rather it is left open to be specified by each particular implementation of Nucleus. At the beginning of program execution, an input record pointer is set to zero and execution of a read statement then proceeds as follows.
1. The input record pointer is increased by one to a value of, say, p.

2. If record p is an eof record, the character "T" is placed in array[0] and the remainder of the array is unchanged.

3. If record p is not an eof record, the character "F" is placed in array[0] and character i of record p is placed into array[i] for all i such that 1 ≤ i ≤ min (array bound, record size). The remainder of the array, if any, is left unchanged.

**Write. WRITE array**

The write statement is closely related in behavior to the read statement. Again the array must be a type character array. The write statement accesses a standard output file similar in structure to the input file, but with possibly a different record size. At the beginning of program execution, an output record pointer is set to zero, and execution of a write statement then proceeds as follows.

1. The output pointer is increased by one to a value of, say, p.

2. If array[0] contains the character "T", then record p becomes an eof record.

3. If array[0] does not contain the character "T", then character i of record p becomes the character in array[i] for all i such that 1 ≤ i ≤ min (array bound, record size). The rest of the characters in the record, if any, become blanks.

**Enter. ENTER identifier**

This is a recursive call of the procedure named identifier.

**Return. RETURN**

The return statement causes a jump to the end of the procedure in which it occurs.
Null. NOP

The null statement causes a jump to the next statement in sequence.

Halt. HALT

The halt statement causes immediate termination of the program execution.

Expressions.

Expressions are built from primaries in the usual way. The operators that are available are given in Table II.1, and each operator may be applied only to operands of the appropriate type. The

TABLE II.1

Nucleus Expression Operations

<table>
<thead>
<tr>
<th>Operator</th>
<th>Priority</th>
<th>Operand Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>unary+,-</td>
<td>1</td>
<td>INTEGER</td>
</tr>
<tr>
<td>*,/,% (modulo)</td>
<td>2</td>
<td>INTEGER</td>
</tr>
<tr>
<td>binary+,-</td>
<td>3</td>
<td>INTEGER</td>
</tr>
<tr>
<td>&lt;,&lt;=,&gt;,&gt;=,=,#</td>
<td>4</td>
<td>see below</td>
</tr>
<tr>
<td>∼</td>
<td>5</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>∧</td>
<td>6</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>∨</td>
<td>7</td>
<td>BOOLEAN</td>
</tr>
</tbody>
</table>

relational operations may be applied to operands of any type, provided both operands are of the same type. If operands of type boolean or character are used, the transfer function to type integer is applied automatically.

If an expression would evaluate to a value v such that the implementation parameter inrange(v) = false, then the value of the
expression becomes undefined (Axioms 22-27). The expression also becomes undefined upon divide or modulo by zero (Axioms 26, 27).

Primaries.

A primary may be a constant (a NUMBER, TRUE, FALSE, or CHARACTERCONSTANT token), a simple variable, an array reference, an expression enclosed in parentheses, or the application of one of the type transfer functions, INTEGER, BOOLEAN, or CHARACTER. In an array reference, IDENTIFIER [expression], if the expression falls outside the array bounds, the value of the array reference is undefined (Axiom 11). The type transfer functions have as an argument an expression of any type. The functions for transfers between all possible pairs of types are given in Appendix C.

The following example of a Nucleus program illustrates the program structure, the form of assertions, and several types of statements. This program sums the absolute values of the first N elements of array A and places this value in SUM. The main procedure is ADD which does the summation and calls procedure ABS to set element I of array A to its absolute value. This example will be called the ADD example when referenced in later sections.

```
INTEGER I,N,SUM;
INTEGER ARRAY A[100];

PROCEDURE ADD;
ASSERT 0 ≤ N.0 ≤ 100;
SUM:=0;
I:=0;
ASSERT X ≥ I → A[X]=A.0[X];
ASSERT SUM=TOTAL X=0 TO I-1 OF ABS(A.0[X]);
ASSERT N=N.0;
```
ASSERT 0 ≤ I ≤ N+1;
WHILE I ≤ N DO
    ENTER ABS;
    SUM:=SUM+A[I];
    I:=I+1;
EXIT;

ELIHV;
ASSERT SUM=TOTAL X=0 TO N.0 OF ABS(A.0[X]);
EXIT;

PROCEDURE ABS;
ASSERT 0 ≤ I.0 ≤ N.0;
IF A[I] < 0
    THEN A[I]:=−A[I];
FI;
ASSERT A[I.0]=ABS(A.0[I.0]);
ASSERT X#I.0 → A[X]=A.0[X];
EXIT;

START ADD

II.4. Method of Formal Definition

The formal definition of Nucleus consists of two components, syntax and semantics. The syntax of Nucleus is a set of rules that determine whether or not any given character string is a Nucleus program. For any string of characters which is a legal program, the semantics of Nucleus specify the executions of that program. The formal definition uses transition networks, Woods [15], to define the syntax, and transition networks and axioms, Burstall [1], to define semantics. The choice of these mechanisms was strongly influenced by the goals of proving the Nucleus verifier.

II.4.1. Syntax

The definition of the Nucleus syntax consists of two distinct transition networks, a scanning network and a parsing network. The
scanning network reads the input string of Nucleus characters and

groups these together to form the input string of Nucleus tokens for

the parsing network.

The transition networks employed in the definition of

Nucleus are a modified form of the "augmented transition network


These networks are based on finite state transition diagrams. The

language defined by the grammar is the set of strings accepted by the

network. This amounts to defining the language by defining its

recognizer and provides two significant advantages in proving the

recognizer component of the Nucleus verifier. First, the transition

networks define completely the Nucleus syntax, including such restrictions

as no identifier may be declared more than once, and + may only be

applied to expressions of type integer. Second, since the transition

networks specify explicitly a recognition procedure, the correctness of

the recognizer component of the verifier can be stated in terms of an

equivalence with the transition networks. This greatly simplifies the

proof of the recognizer.

The networks employed in the Nucleus definition are simpler

in operation than the ones described by Woods for two reasons. First,

we do not need all of the features that are necessary to cope with

natural languages. For example, these networks have no backtracking

mechanism. Second, the Nucleus networks are sufficiently simple so

that their operation can be defined by a set of simple axioms. These
axioms are another important advantage in proving the recognizer component of the verifier. The axiomatic description of the transition networks appears in Appendix A. The remainder of this section gives an informal description of the transition networks and their operation.

A transition network is a directed graph with labelled nodes and arcs. The labelled nodes make up the set of states. One state is designated as the initial state, and some set of states is specified as the set of recognition states. The network also has associated with it a return stack for saving arcs, an input string, an input pointer, and a set of registers. The registers form the memory for the network. They contain values that can be manipulated and tested during the operation of the network. Each arc in the network is labelled with either an input string character, nil, or the name of some state. Each arc also has associated with it a test, a set of actions, and a scan flag. The test is a condition defined on the registers, and the actions are sequences of assignment operations on the registers.

The operation of the network begins at the initial state with the input pointer pointing to the first character of the input string and the return stack empty and proceeds as follows for any state that is attained. First, the arcs leaving the state are examined to find a traversable arc. To determine a traversable arc, all arcs labelled with input string characters are considered first. If the character labelling the arc matches the character pointed to by the input pointer and the test associated with the arc is satisfied, then
the arc is traversable. If there are no traversable arcs labelled with input characters, arcs labelled with "nil" are considered next. An arc labelled with "nil" is traversable if its test is satisfied. If still no traversable arc is found, then state-labelled arcs are considered. In the Nucleus networks there is at most one such arc leaving any state. This arc is saved on the return stack and the next state attained is the state used to label the arc. If the network proceeds from that state to a recognition state, the arc on the top of the return stack is reconsidered and is traversable if its test is satisfied. If so, it is removed from the stack. In order to be certain that each state can have at most one traversable arc, the following restrictions are imposed on the network.

- Two arcs leaving a state may have the same label only if their associated tests can never be satisfied simultaneously.
- A state may have at most one state-labelled arc leaving that state.

Once the traversable arc is determined, the actions associated with that arc are performed. If the scan flag for the arc is set, the input pointer is advanced to the next character on the input string and the next state attained is the state entered by the traversable arc. If a state has no traversable arc and is a recognition state, then the arc at the top of the return stack is reconsidered as mentioned above. An empty stack determines acceptance in the language of the part of the input string preceding the input pointer. If a state has no traversable arc and is not a recognition state, then the
input string is rejected as a sentence in the language.

One additional restriction is placed on the network.

- A recognition state may not have a state-labelled arc leaving that state.

With this restriction, and the fact that every state has at most one traversable arc, we can always make the proper transition or termination decision from any state without any further scanning, either looking ahead or backtracking.

As an example, consider the opening segment of the Nucleus parsing network in Figure II.1. This segment establishes the form of Nucleus programs. This example does not contain multiple arcs from a state, but does illustrate most of the other features of transition networks. Arc labels "START" and "IDENTIFIER" are input characters with respect to the parsing network, while "declarations" and "procedures" are state names which appear in network segments not shown. Any missing tests are assumed to be identically "true" and any missing actions are the identity assignment. State "program" is the designated initial state of the Nucleus parsing network and state 5 is a recognition state.

Traversal of the parsing network begins at state "program" with the return stack empty. The only outgoing arc is labelled "nil" with no test and is thus traversable. The first action, which initializes several of the network registers to the empty set, is now performed. The scan flag is set to "NOSCAN" so the input pointer is not advanced, and the network next attains state 1. The arc leaving state 1 is
program

nil
DEFINED.SIMPLE.SET := {}
DEFINED.ARRAY.SET := {}
TYPE_FUNCTION := {}
DEFINED.PROCEDURE.SET := {}
REFERENCED.PROCEDURE.SET := {}
DEFINED.IDENTIFIER.SET := {}
NOSCAN

1

declarations
NOSCAN

2

procedures
REFERENCED.PROCEDURE.SET = DEFINE.PROCEDURE.SET
NOSCAN

3

START
SCAN

4

IDENTIFIER
TOKEN.STRING ε DEFINED.PROCEDURE.SET
NOSCAN

5

FIGURE II.1. The Opening Segment of the Nucleus Parsing Network
labelled with a state name so this arc is placed on the return stack and operation proceeds to the state labelled "declarations". If a recognition state is then reached, the arc from state 1 to state 2, which is on top of the stack, will be traversable and state 2 is attained. The input pointer will have been advanced to the character beyond the declarations. In the same manner, the network looks for the procedures by beginning operation at the state labelled "procedures". If the procedures are found, then the test on the arc leaving state 2 requires that all procedures referenced in a procedure call must be defined procedures. The network now looks for input characters "START" and "IDENTIFIER". TOKEN.STRING is a register containing the actual input character string constituting the IDENTIFIER, so the test requires that the IDENTIFIER name some defined procedure. The input string then is recognized when the network attains recognition state 5 with the stack empty.

As described in [7] the definition of the Nucleus syntax actually involves two separate networks, a scanner and a parser. These cooperate in passing scanned tokens from the scanner to the parser. The input string for the scanner is the actual string of characters which constitutes the Nucleus program. Thus, character-labelled arcs in the scanner are labelled with members of the basic Nucleus character set and end-of-file, and the scan flag controls the actual advance of the input string.

The operation of the scanner defines and leaves values in two special registers, TOKEN and TOKEN.STRING. TOKEN specifies the type
of Nucleus token recognized, such as an identifier, and TOKEN.STRING contains the actual string of characters making up that token, such as the name of an identifier. These two registers are available to the parser on a read-only basis. There is no input string as such for the parser. Instead, the parser matches character-labelled arcs against the TOKEN register rather than against the actual input string. (Character-labelled arcs in the parser are labelled with tokens such as "IDENTIFIER" and "IF" while arcs in the scanner are labelled with the actual individual characters in the input string such as "X" and "Y".) The scan network is initiated first to define initial values for the TOKEN and TOKEN.STRING registers. Then the parser is initiated, and thereafter the scan flag of the parser controls the initiation of the scanning network. When the parser requests a scan, the scanner is initiated to define the next values of the TOKEN and TOKEN.STRING. The Nucleus tokens are defined so that the scanner always will recognize a token at the head of any string of Nucleus characters.

II.4.2. Semantics

The semantics of Nucleus are defined by the axiomatic method described by Burstall [1]. First, we define a transformation from programs into sentences in the predicate calculus. This transformation will be called the semantic mapping and the set of predicate calculus sentences produced will be called the reduced program. Then we define a set of axioms such that the execution of any Nucleus program on any input can be deduced from its reduced program and the axioms.
In the Nucleus definition, the transition network which defines the syntax also defines the semantic mapping. The Nucleus program is a character string, and each predicate calculus sentence in the reduced program is also a character string. Thus, the transformation is from one string into a set of strings. The Nucleus parsing network defines the semantic mapping through the use of a special action called SENTENCE. The action SENTENCE(x) defines the character string x to be a sentence in the reduced program. Thus, the parsing network not only defines the reduced program, but also gives a procedure for constructing it. The predicates which are used in forming the reduced program are listed and defined in Appendix B. The following example illustrates the reduced program for the ADD example given earlier. The assertions do not appear in the reduced program and are omitted from the example. The left column contains the original Nucleus program and the right column contains the resulting reduced program.

**Nucleus Program**

```
INTEGER I,N,SUM;

INTEGER ARRAY A[100];

PROCEDURE ADD;
SUM:=0;
I:=0;
WHILE I <= N DO
  ENTER ABS;
  SUM:=SUM+A[I];
  I:=I+1;
ELIHW;
EXIT;

PROCEDURE ABS;
```

**Reduced Program**

```
SIMPLE(I)
SIMPLE(N)
SIMPLE(SUM)
ARRAY(A,100)

ASSIGN(ADD:0,SUM,0)
ASSIGN(ADD:1,I,0)
IF(ADD:2,(I) <= (N),3,7)
ENTER(ADD:3,ABS)
ASSIGN(ADD:4,SUM,(SUM)+(A[I]))
ASSIGN(ADD:5,I,(I)+(1))
JUMPTO(ADD:6,2)
EXIT(ADD:7)
EXITPOINT(ADD)=7
```
IF A[I] < 0
  THEN A[I] := -A[I];
FI;
EXIT;
START ADD

IF(ABS:0,(A[I]) < (0),1,2)
  ASSIGN(ABS:1,A[I],-(A[I]))
EXIT(ABS:2)
EXITPOINT(ABS)=2
INITIALPROCEDURE=ADD

The Nucleus axioms are based directly on the concept of state vectors. A state vector, S, is a function from some name space NS into some value space VS. Each member (n,v) of S is a cell, n being the name of the cell and v its value. Thus, S(n) is the value in state vector S of the cell whose name is n. The execution of a Nucleus program is defined as a sequence of state vectors S_0, S_1, S_2, ... . This sequence can be regarded as a function E from the non-negative integers into state vectors (E[i]=S_i). Thus, E[i](n) is the value of the cell whose name is n in the i-th state vector of the program execution. The axioms define the execution of a program by defining the function E and are listed in Appendix C.

There are three classes of axioms in the Nucleus definition, declaratives, evaluatives, and imperatives. The declarative axioms define the name space of the state vectors in the execution of a program, the evaluative axioms describe the evaluation of expressions on an arbitrary state vector, E[i], and the imperatives define the execution sequence by specifying E[i+1] in terms of E[i] and by defining the termination conditions.

In Nucleus, every state vector in every program has the same value space. This value space consists of the union of a number of
disjoint sets: the set of integers, the set of boolean values, true and false, the 64 basic characters of Nucleus, and the set of character strings of the form I:D where I is an identifier and D is a digit string. The value space also contains an undefined element, U, which is distinguishable from every other element in the set. Since the value space is the same for all programs, it is not defined by the axioms.

Each Nucleus program has an associated name space which serves as the name space for every state vector in the execution of that program. The elements of this name space are character strings defined by the declarative axioms. In addition to the undefined element, U, the name space of every program contains the elements :LOC, :LVL, :RDHD, :WTHD, :RTNPT[0], ..., :RTNPT[maxstacksize]. The colons are included in these names to avoid confusion with the declared variables for the program. The names represent the location counter, return stack level, read record pointer, write record pointer, and return stack. The quantity "maxstacksize" is another of the Nucleus implementation parameters.

Table II.2 gives a sample execution sequence for the ADD example that sums the absolute values of array A. The column on the left lists the name space for the program execution, listing the special names which are in every name space first and the declared names last. Each column on the right is a state vector. The initial state vector E[0] has a value of :LOC which corresponds to point zero of the initial
### TABLE II.2

Sample State Vector Sequence for the ADD Example

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>:LVL</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>:RDHD</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>:WTHD</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>?</td>
<td>?</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>N</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SUM</td>
<td>?</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>A[0]</td>
<td>-5</td>
<td>-5</td>
<td>-5</td>
<td>-5</td>
<td>-5</td>
<td>-5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

25
procedure (Axiom 17), \( \text{:LVL} = -1 \) (Axiom 18), \( \text{:RDHD} = 0 \) (Axiom 19), and \( \text{:WTHD} = 0 \) (Axiom 20). For simplicity in this example it is assumed that the values of N and A[0] are initially 0 and -5. Since the statement ASSIGN(ADD:0, SUM, 0) appears in the reduced program and since \( E[0](\text{:LOC}) = \text{ADD}:0 \) then by Axiom 48 we get \( E[1](\text{:LOC}) = \text{ADD}:1, E[1](\text{SUM}) = 0 \) and for all other names, \( x, E[1](x) = E[0](x) \). Execution terminates at state vector \( E[11] \) since EXIT(ADD:7) is in the reduced program and \( E[11](\text{:LOC}) = \text{ADD}:7 \) (see Axiom 51).
CHAPTER III

THE INDUCTIVE ASSERTION METHOD FOR NUCLEUS

III.1. Introduction

This chapter presents the theoretical basis for the verification of Nucleus programs by the inductive assertion method. First, we define what is meant by partial and total correctness of a Nucleus program and then, for each of these types of correctness, we define a set of verification conditions which are sufficient to prove correctness.

This presentation of correctness includes several unique features.

1. Input and output operations are included in the set of language features for which verification conditions are defined. The addition of input and output operations significantly expands the set of possible programs to which the inductive assertion method can be applied.

2. The validity of the verification conditions is proved from the formal axiomatic definition of Nucleus. The proof demonstrates that the verification conditions defined are sufficient to prove correctness of Nucleus programs.

3. Two sets of verifications conditions are defined, one set for partial correctness proofs and another set for total correctness proofs. The two types of correctness involve different treatments of termination. This difference is reflected in the verification conditions.

27
Conditions which result in termination of execution are explicitly
treated in the verification conditions.

4. A mechanism for dealing with termination proofs is
built into the verification condition definitions. The variable :STEP
is treated as a counter for the number of steps in an execution and can
be referenced in assertions.

5. A distinction is made between two kinds of termination.
Termination which results from execution of a halt or exit is termed
normal, and termination which results from an error condition such as
division by zero is termed abnormal.

6. The assertions and verification conditions provide for
explicit treatment of the set of system variables. Thus, the assertions
may reference such system variables as the location counter, :LOC, and
return point stack level, :LVL.

The inductive assertion method consists of attaching predicates
to certain key points in the program and showing that the predicates
are true each time execution reaches the point to which the predicate
is attached. Section III.2 describes how these predicates (assertions)
are associated with program points and how the statement of correctness
is made via the initial assumption and desired result. Partial and total
correctness are defined in terms of the initial assumption, the desired
result, and normal termination. In Section III.3, the concept of a
path is developed and a correspondence between points along a path
and sequences of state vectors is formulated. Section III.4 gives
an informal description of verification conditions including an example of a verification condition. In Section III.5, a set of verification condition terms for partial correctness is defined for each reduced program statement. In Section III.6, these terms are used to define verification conditions for use in partial correctness proofs and the validity of these verification conditions is proved. Section III.7 discusses a set of verification conditions sufficient to prove total correctness.

III.2. Definitions of Correctness

The correctness of Nucleus programs is defined in terms of a relationship between initial and final values of the elements of the state vector. Partial correctness, with respect to a given initial assumption and desired result, can be informally stated as, "If the initial state vector satisfies the initial assumption and if the program terminates normally, then the final state vector satisfies the desired result." Total correctness is the condition, "If the initial state vector satisfies the initial assumption, then the program terminates normally and the final state vector satisfies the desired result."

We will now define a series of terms which leads up to the formal definition of partial and total correctness. As indicated in Chapter II, the Nucleus axioms define a state vector sequence E[0], E[1], E[2], ... for any Nucleus program. We will use the term execution procedure of E[i] to refer to PNAME(E[i](LOC)) where the function PNAME is defined in Axiom 45. If E[i](LOC)=P:Q then the execution
procedure of \( E[i] \) is procedure \( P \).

The following two terms are used extensively throughout the remaining definitions and theorems so a thorough understanding of them is essential. The terms \( \text{entry}(i) \) and \( \text{exit}(i) \) denote functions from one state vector index to another. They are defined so that the state vector \( E[\text{entry}(i)] \) is at the entry to the execution procedure of \( E[i] \) and if the procedure reaches its exit point, \( E[\text{exit}(i)] \) is the state vector at exit. For any given state vector sequence, we define

\[
\begin{align*}
\text{entry}(i) & = \begin{cases} 
0 & \text{if } E[i](:LVL) = -1 \\
\max \{ j | \text{ENTRY}(E[j-1](:LOC), \text{PNAME}(E[i](:LOC))) \land E[i](:LVL) = E[j](:LVL) \} & \text{otherwise}
\end{cases} \\
\text{exit}(i) & = \begin{cases} 
\min \{ j | \text{EXIT}(E[j](:LOC)) \land E[i](:LVL) = E[j](:LVL) \} & \text{if the set is not empty} \\
\text{undefined} & \text{otherwise}
\end{cases}
\end{align*}
\]

In the definition of \( \text{entry}(i) \), the first line defines entry to the initial procedure to be at state vector zero. The term "\( j \leq i \)" in the second line requires that state vector \( E[\text{entry}(i)] \) occurs before \( E[i] \) or is \( E[i] \) itself. The next term requires that \( E[\text{entry}(i)] \) is a state vector immediately following a call of the current execution procedure and the last term requires \( E[\text{entry}(i)] \) to be at the same level as \( E[i] \). The "\( \max \)" operator makes certain that \( E[\text{entry}(i)] \) is defined at the most recent entry to the current execution procedure at this level.

Consider now the definition of \( \text{exit}(i) \). The term "\( j \geq i \)" indicates that \( E[\text{exit}(i)] \), if it exists, occurs after \( E[i] \) or is \( E[i] \)
itself. The next two terms require the exit state vector to occur at an exit from the current level. The "min" operator assures that this is the next such exit. The exit state vector is not defined if the execution procedure does not attain its exit point. In the ADD example, entry(6) = 4, entry(4) = 4, exit(1) = 11, and exit(6) = 6.

A state vector sequence in which all state vectors have the same entry state vector will be termed in execution sequence for their common execution procedure. For example, the sequence E[0],E[1],E[2], E[3],E[7],E[8],E[9],E[10],E[11] from the ADD example is an execution sequence for procedure ADD. The gaps in execution sequences correspond to the execution sequences of called procedures. Since every state vector has an entry state vector, then every state vector in a program execution appears in an execution sequence of some procedure. This property plays an essential role in the proof of the validity of the verification conditions.

We distinguish between two types of termination of Nucleus programs. When program execution terminates as a result of reaching the exit point of the initial procedure with the return stack empty (Axiom 51) or by executing a halt statement (Axiom 52), the termination will be referred to as normal termination, and we define predicate NT(n) to be true iff normal termination occurs at state vector E[n]. All other termination is called abnormal termination. Abnormal termination is caused by the occurrence of an undefined value in evaluating an expression in any statement (Axioms 48-50, 53). Undefined values result
from division or modulo by zero, array subscript out of bounds
(including return point stack overflow), or integer out of range.

The set of assertion variables for a Nucleus program is the
set of strings \( \{ x | \text{SIMPLE}(x) \lor \text{ARRAY}(x,b) \} \cup \{ : \text{STEP} \} \). In the definitions
and theorems which follow, \( X \) will denote a vector whose components are
all the elements in the set of assertion variables. The symbol \( X.0 \)
will refer to the same vector except with each variable name followed
by ".0". The notation \( E[i](X) \) refers to the vector of the values of
the elements of \( X \) at state vector \( i \). If \( X = (X_1,X_2,\ldots,X_j) \) then
\( E[i](X) = (E[i](X_1),E[i](X_2),\ldots,E[i](X_j)) \) where if \( X_k \) is array name \( A \) with bound
\( b \) then \( E[i](X_k) = E[i](A) = E[i](A[0]),E[i](A[1]),\ldots,E[i](A[b]) \). The
value of \( E[i](X.0) \) is \( E[\text{entry}(i)](X) \).

We will allow a predicate of the form \( B(X.0,X) \) to be associated
with any state vector \( E[i] \). The value of \( B(X.0,X) \) at \( E[i] \) is denoted
\( E[i](B(X.0,X)) \) and is defined by \( E[i](B(X.0,X)) = B(E[i](X.0),E[i](X)) = B(E[\text{entry}(i)](X),E[i](X)) \). For example, if \( B(X.0,X) \) is the predicate
\( \text{NUM} \leq \text{NUM.0} \) where \( \text{NUM} \) is an assertion variable, then the value of
\( B(X.0,X) \) at state vector \( E[i] \) is the value of the predicate \( E[i](\text{NUM}) \leq \)
\( E[\text{entry}(i)](\text{NUM}) \).

We may also associate predicates of the form \( B(X.0,X) \) with
points in the reduced program. A predicate associated with a reduced
program point is called an assertion and the point is said to be tagged.
If assertion \( B(X.0,X) \) is associated with point \( P:Q \) then "tag[P:Q](X.0,X)"
will denote \( B(X.0,X) \). All entry, exit, and halt points are assumed to
be tagged. If no assertion is explicitly specified, then the assertion "TRUE" is assumed.

The mechanism for stating the initial assumption and desired result of a program is now described in terms of assertions. The program assertion at point zero of the initial procedure is taken to be the initial assumption of the program and is denoted \( A(X.0,X) \). The disjunction of the assertion at the exit point of the initial procedure and the assertions at all halt statements in all procedures is taken to be the desired result of the program and is denoted \( R(X.0,X) \). For example, if the exit point of the initial procedure is tagged with assertion \( B(X.0,X) \) and if there are \( j \) halt statements tagged with assertions \( H_1(X.0,X), \ldots, H_j(X.0,X) \), then \( R(X.0,X) = B(X.0,X) \lor H_1(X.0,X) \lor \ldots \lor H_j(X.0,X) \). Note that if any of these assertions is satisfied, then \( R(X.0,X) \) is satisfied.

Precise meaning is now given to the terms partial and total correctness. A program is **partially correct** with respect to initial assumption \( A(X.0,X) \) and desired result \( R(X.0,X) \) if for all executions of the program, \( E[0](A(X.0,X)) \land \text{NT}(n) \rightarrow E[n](R(X.0,X)) \). As mentioned earlier, partial correctness can be informally stated as, "If the initial state vector satisfies the initial assumption and if the program terminates normally, then the final state vector satisfies the desired result."

A program is **totally correct** with respect to initial assumption \( A(X.0,X) \) and desired result \( R(X.0,X) \) if for all executions of the program,
\[ E[0](A(X.0,X)) \rightarrow \exists n[\text{NT}(n) \land E[n](R(X.0,X))] \]  

Total correctness is the condition, "If the initial state vector satisfies the initial assumption, then the program terminates normally and the final state vector satisfies the desired result."

III.3. Definition of Path

In this section we define the concept of a path from one tagged point to the next. In later sections, the method of constructing a verification condition for each path is described and it is shown that the verification condition is sufficient to prove that whenever the tagged point at the front of the path is reached with its assertion satisfied and execution proceeds along the path, then the assertion at the end of the path is satisfied when its tagged point is reached.

The set of all strings of the form P:Q, where P names a defined procedure and Q is a point in procedure P, is the set of control points for the reduced program. Control points are the elements on which we define the concept of a path. The set of all successors for a control point P:Q1 is defined in Table III.1. If the reduced program contains the instruction on the left then the set of successors is given at the right.

Table III.1 introduces a notational convention which serves as a grouping symbol for strings. Any string of characters which is underlined is to be evaluated before being concatenated to the rest of the string. Thus, P:Q1+1 means concatenate "P:" and the value of Q1+1.
\begin{align*}
\text{instruction} & \quad \text{successors} \\
\text{ASSIGN}(P:Q_1,N,V) & \quad \{P:Q_1 + 1\} \\
\text{CASE}(P:Q_1,\text{EXP},L) & \quad \{P:\text{POINTLABELLEDWITH}(W) \mid W \in \text{CASELABELSET}(P:Q_1) \} \cup \{P:L\} \\
\text{ENTER}(P:Q_1,C) & \quad \{P:Q_1 + 1\} \\
\text{EXIT}(P:Q_1) & \quad \{\}\} \\
\text{HALT}(P:Q_1) & \quad \{\}\} \\
\text{IF}(P:Q_1,\text{EXP},T,F) & \quad \{P:T,P:F\} \\
\text{JUMPTO}(P:Q_1,N) & \quad \{P:N\} \\
\text{READ}(P:Q_1,A) & \quad \{P:Q_1 + 1\} \\
\text{WRITE}(P:Q_1,A) & \quad \{P:Q_1 + 1\}
\end{align*}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\text{TABLE III.1 Definition of Control Point Successors} \\
\hline
\end{tabular}
\end{table}

Note the relationship between successive points and successive values of :LOC in the execution sequence of a procedure. If $E[i](:\text{LOC}) = P:Q_1$ then for all statements except an enter statement, $E[i+1]$ is the next state vector in the procedure execution sequence and $E[i+1](:\text{LOC})$ is a successor of $P:Q_1$ (see Axioms 48–58). For the enter statement, the top of the return point stack in state vector $E[i+1]$ is $P:Q_1 + 1$ (Axiom 50) which means that the value of :LOC upon exit from the called procedure will be $P:Q_1 + 1$ (Axiom 51). The state vector following the exit from the called procedure is the successor of $E[i]$ in the execution sequence and its value of :LOC is a successor of $P:Q_1$. Therefore, the successor relationship between control points corresponds to the order in which they can occur as values of :LOC within a
procedure execution sequence.

Control point sequence \(P:Q_1, \ldots, P:Q_r\) is a path iff \(P:Q_i + 1\)
is a successor of \(P:Q_i\) for \(i = 1, \ldots, r-1\), \(P:Q_1\) and \(P:Q_r\) are both
tagged, and no other points in the sequence are tagged. A path will
sometimes be denoted \((Q_1, \ldots, Q_r)\) if the procedure is clear from context.

If a program execution terminates normally, then the state
vectors in a procedure execution sequence which ends with the execution
of an exit or halt can be divided into sequences of state vectors
whose :LOC values form paths. The last state vector in one sequence
will be the first state vector in the next sequence. Each such state
vector sequence is said to correspond to the path formed by its :LOC
values. The only execution sequences not included above are those whose
last state vector occurs at a call of a procedure which leads to
termination at a halt without returning. These procedure execution
sequences can be divided into sequences which correspond to paths and
one sequence which corresponds to the front of the path which was
interrupted by the procedure call. If we define the front portion of
a path through a procedure call to be a procedure entry path then any
procedure execution sequence can be divided into a series of sequences
which correspond to paths and possibly one more sequence which corresponds
to a procedure entry path. The state vector sequence \(E[2], E[3], E[7],
E[8], E[9], E[10]\) corresponds to path \(2, 3, 4, 5, 6, 2\) in procedure ADD of
the ADD example and the sequence \(E[2], E[3]\) corresponds to procedure
entry path \(2, 3\) in the same example. Notice that there may be several
procedure entry paths in a single path and the same procedure entry path may head several different paths.

A program is said to be properly tagged if all paths are finite in length, that is if all loops contain at least one tagged point. This is clearly always possible since tagging all points breaks all loops. Recall that all entry, exit, and halt points are assumed to be tagged, and if no assertion is explicitly specified for these points, then the assertion "TRUE" is assumed.

For an arbitrary properly tagged program, we have a potentially infinite set of possible executions and we have a finite set of paths and procedure entry paths. Since each execution can be broken down into sequences which correspond to paths and procedure entry paths, we now have a mechanism for reducing proofs about a potentially infinite set of executions to proofs about a finite set of paths and procedure entry paths.

III.4. Informal Description of a Verification Condition

The verification condition for each path is a conjecture constructed in a way such that the verification condition is sufficient to prove that whenever the tagged point at the front of the path is reached with its assertion satisfied and execution proceeds along the path, then the assertion at the end of the path is satisfied when its tagged point is reached. The verification condition also requires that all procedures called along the path are entered with their initial assertion satisfied. Thus, the verification condition is also
sufficient to prove that for any of its procedure entry paths, whenever
the tagged point at the front of the procedure entry path is reached
with its assertion satisfied and execution proceeds along the procedure
entry path, then the procedure called at the end of the procedure
entry path is entered with its initial assertion satisfied. Since any
execution can be broken down into paths and procedure entry paths,
then proof of the set of all verification conditions is sufficient to
prove that if the initial assumption is satisfied at state vector
zero, then each assertion is satisfied each time its tagged point is
reached during execution. Thus, if the program terminates normally,
the assertion at the final state vector is satisfied and the program
is partially correct.

The following verification condition example is presented
as a preview to clarify the later definitions. The example program
(which will be called the DIV example) contains a single procedure
which performs division by repeated subtraction. The initial value
of N is some non-negative integer and D is a positive integer. The
procedure places the integer-valued quotient of N and D in Q and the
remainder in R. In the assertions, variable names with a suffix of
".0" are references to initial values. The numbers in parentheses
at the left are not part of the program, but are provided to show
the correspondence between points in the Nucleus program and points in
the reduced program.

The tagged points in this program are 0,1, and 6 and the paths
are (0,1),(1,2,3,4,1), and (1,5,6). The verification condition for
<table>
<thead>
<tr>
<th>Nucleus Program</th>
<th>Reduced Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER N,D,Q,R;</td>
<td>SIMPLE(N)</td>
</tr>
<tr>
<td>PROCEDURE DIV;</td>
<td>SIMPLE(D)</td>
</tr>
<tr>
<td>ASSERT D.0 &gt; 0;</td>
<td>SIMPLE(Q)</td>
</tr>
<tr>
<td>ASSERT N.0 ≥ 0;</td>
<td>SIMPLE(R)</td>
</tr>
<tr>
<td>(0) Q:=0;</td>
<td>ASSIGN(DIV:0,Q,0)</td>
</tr>
<tr>
<td>ASSERT N.0=Q*D.0+N;</td>
<td></td>
</tr>
<tr>
<td>ASSERT D=D.0;</td>
<td></td>
</tr>
<tr>
<td>ASSERT N ≥ 0;</td>
<td></td>
</tr>
<tr>
<td>(1) WHILE N ≥ D DO</td>
<td>IF(DIV:1,(N) ≥ (D),2,5)</td>
</tr>
<tr>
<td>(2) N:=N-D;</td>
<td>ASSIGN(DIV:2,N,(N)-(D))</td>
</tr>
<tr>
<td>(3) Q:=Q+1;</td>
<td>ASSIGN(DIV:3,Q,(Q)+(1))</td>
</tr>
<tr>
<td>(4) ELIHV;</td>
<td>JUMPTO(DIV:4,1)</td>
</tr>
<tr>
<td>(5) R:=N;</td>
<td>ASSIGN(DIV:5,R,N)</td>
</tr>
<tr>
<td>ASSERT N.0=Q*D.0+R;</td>
<td></td>
</tr>
<tr>
<td>ASSERT 0 ≤ R &lt; D.0;</td>
<td></td>
</tr>
<tr>
<td>(6) EXIT;</td>
<td>EXIT(DIV:6)</td>
</tr>
<tr>
<td>START DIV</td>
<td>EXITPOINT(DIV)=6</td>
</tr>
<tr>
<td></td>
<td>INITIALPROCEDURE=DIV</td>
</tr>
</tbody>
</table>

path (1,2,3,4,1) is shown below. Again, the numbers at the left of each column are not a part of the verification condition, but are provided as references to the lines of the verification condition. Terms written in a column are joined by conjunction and a solid horizontal line denotes implication. Thus

\[
\begin{array}{c}
A \\
B \\
C \\
D
\end{array}
\]

denotes \(A \land B \rightarrow C \land D\).

For any variable \(v\), the string "v.0" denotes the value of \(v\) at entry to the procedure, "v" denotes the value at the front of the path, and "v.i" denotes the value of \(v\) after \(i\) changes in value along the path. The integer \(i\) will be called the value of the alteration
counter of variable v.

Lines 0.1 and 0.2 are the initial assertion of the procedure in which the path occurs and lines 1.1-1.3 are the assertion at the front of the path. The dashed line has no formal significance. It serves as a separator between the lines which come from the assertions just mentioned and the lines which reflect execution along the path.

```
0.1 D.O > 0
0.2 N.O >= 0
1.1 N.O=Q*D.O+N
1.2 D=D.0
1.3 N >= 0
```

```
1   N >= D
1   :LOC.1=DIV:2
1   :STEP.1=:STEP+1
2   inrange(N-D)
2   N.1=N-D
2   :LOC.2=:LOC.1 +1
2   :STEP.2=:STEP.1 +1
3   inrange(Q+1)
3   Q.1=Q+1
3   :LOC.3=:LOC.2 +1
3   :STEP.3=:STEP.2 +1
4   :LOC.4=DIV:1
4   :STEP.4=:STEP.3 +1
1.1' N.O=Q.1*D.0+N.1
1.2' D=D.0
1.3' N.1 >= 0
```

The terms between the dashed and solid lines describe the path execution. These terms are formally defined in Section III.5. The number at the left indicates the reference number of the program statement which generates the verification condition line at the right. For example, consider instruction 1 which is the test for the while statement. Under the assumption that path (1,2,3,4,1) is traversed,
we know that \( N \geq D \) is true and that the value of the location counter :LOC after one alteration along the path is "DIV:2". Also, the value of execution step counter :STEP is incremented by one. Lines 1.1'-1.3' are the assertion at the end of the path with current values properly indicated by the alteration counters. The proof of the verification condition consists of proving terms below the solid line from the terms above it.

The terms "inrange(N-D)" and "inrange(Q+1)" do not appear to match the definitions in Section III.5. The first line of the definition of the verification condition terms for assignment statements (Definition 3.5.1a) states that the right-side expression is not undefined. (If the right-side expression is undefined, then the execution terminates abnormally. This contradicts the assumption of normal termination for partial correctness proofs.) In the DIV example, the right-side expression is defined if its value is in range. For a particular Nucleus implementation, these verification condition terms can be more precisely specified. For example, if the implementation dependent predicate, inrange(x), is defined by specifying lower and upper bounds for x, say inrange(x) \( \equiv \) lower \( \leq \) x \( \leq \) upper, then the above verification condition terms can be written as "lower \( \leq \) N-D \( \leq \) upper" and "lower \( \leq \) Q+1 \( \leq \) upper".

The complete proof of partial correctness for procedure DIV includes the proof of the sample verification condition and the proofs of the verification conditions for paths (0,1) and (1,5,6).
III.5. Verification Condition Terms for Partial Correctness

The verification condition term definitions are accompanied by the definition of a function \( \text{alt}(x, i) \). This function is called the alteration function and counts the number of times each assertion variable \( x \) has been altered in traversing the current path up to point \( i \). A variable other than a system variable may be altered only by appearing on the left of an assignment statement or in a read statement. The value of \( \text{alt}(N, 3) \) for path \( (1, 2, 3, 4, 1) \) of the DIV example is the number of times \( N \) has been altered when the third point along the path is reached and is called the alteration counter of \( N \) at the third path point. Whenever the value of \( N \) at this point is to be referenced, it may be written "\( N.\text{alt}(N, 3) \)" or since \( \text{alt}(N, 3) = 1 \) we may write "\( N.1 \)".

For an array \( A \), the value of \( \text{alt}(A, i) \) indicates the number of alterations of array \( A \) up to point \( i \) where an array is considered to be altered whenever any element is altered. The array reference \( A.\text{alt}(A, i)[5] \) refers to the value of \( A[5] \) at the \( i \)-th point along the current path.

In the verification conditions, the symbol for a variable at the beginning of the current procedure is written in the form "\( x.0 \)" and the symbol at the beginning of the current path is simply "\( x \)". If the beginning of the current procedure and the current path is the same, the conflict is resolved by using "\( x.0 \)". This is accounted for in the definition of the alteration function at the beginning of a path. For paths which begin at the entry point of a procedure, the alteration function is defined as \( \text{alt}(X, 1) = 0 \) and for other paths the definition
is $\text{alt}(X, 1) = 0$ (zero-dot). The first behaves in normal fashion and the zero-dot behaves **numerically** the same as zero, but when $x \cdot 0$ is used in a verification condition, it appears as just $x$ (without the ".$0$" suffix).

The notation $\text{EXP}.\text{alt}(X,i)$, where $\text{EXP}$ is an expression, denotes the expression resulting from the substitution of $x.\text{alt}(x,i)$ for each occurrence of any $x \in X$ that appears in $\text{EXP}$. The value of $E[i](\text{EXP})$ is the value of $\text{EXP}$ with $E[i](x)$ substituted for each occurrence of $x$ in $\text{EXP}$.

We will define the verification conditions by defining a set of verification condition terms in definitions 3.5.1 - 3.5.7. These reflect the effect of the Nucleus statements on the state vector. These terms are combined to form the **partial correctness** verification conditions in Theorems 3.6.1-3.6.4.

Two sets of verification condition terms are defined for each statement along a path, a cond set and a vcterm set. The cond terms represent a condition which is sufficient to prove that execution proceeds "properly" to the next path point. For **partial** correctness, the only "improper" execution is entering a called procedure without satisfying its initial assertion. Thus, the partial correctness cond set is identically "TRUE" for each statement except the enter statement. The cond set for enter statements requires that the initial assertion of the called procedure is satisfied. The cond sets for **total** correctness include the above condition for enter statements and, in
addition, include any conditions which result in abnormal termination. The verification conditions require proof of the terms in the cond set from the vcterm for preceding points along the path. The vcterm set states the relationship between the state vectors which correspond to successive path points.

The verification condition terms are written in a tabular format. The lines in a column are connected by logical conjunction. The following definitions describe the partial correctness verification condition terms \( \text{cond}(P:Q_i,i) \) and \( \text{vcterm}(P:Q_i,P:Q_{i+1},i) \) and the alteration function \( \text{alt}(x,i) \) when \( P:Q_i \) and \( P:Q_{i+1} \) are points \( i \) and \( i+1 \) along a path. Recall that the underlined convention indicates that the underlined portion is to be evaluated before being concatenated onto the string.

Definition 3.5.1a assign (with simple left side)

If \( \text{ASSIGN}(P:Q_i,N,V) \) and \( \text{SIMPLE}(N) \) then \( \text{cond}(P:Q_i,i) \) is

\[
\text{TRUE}
\]

and \( \text{vcterm}(P:Q_i,P:Q_{i+1},i) \) is

\[
\begin{align*}
V.\text{alt}(X,i) \neq & U \\
N.\text{alt}(N,i+1) = & V.\text{alt}(X,i) \\
:LOC.\text{alt}(:LOC,i+1) = :LOC.\text{alt}(:LOC,i) +1 \\
:STEP.\text{alt}(:STEP,i+1) = :STEP.\text{alt}(:STEP,i) +1
\end{align*}
\]

and \( \text{alt}(y,i+1) = \\
\begin{align*}
\text{if } y \in \{N,:LOC,:STEP\} \\
\text{then } \text{alt}(y,i)+1 \\
\text{else } \text{alt}(y,i).
\end{align*}
\]
ASSIGN(P:Qi,N,V) indicates an assignment statement at location P:Qi having a left side of N and a right side of V. The cond term is "TRUE" because, with the assumption of normal termination for partial correctness, an assignment statement always executes "properly". When writing a verification condition, terms which are identically "TRUE" do not appear.

The first line of vterm, "V.alt(X,i)#U", states that the right side expression, V, is not undefined. This is a result of the partial correctness assumption of normal termination. If normal termination is assumed, then the right side cannot be undefined since this would result in abnormal termination (Axiom 48). An expression is undefined (1) if the subscript expression for any array reference has a value outside the array bounds (Axiom 11), (2) if for any integer operation, the resulting value v does not satisfy inrange(v) (Axioms 22-27), or (3) if division or modulo by zero is attempted (Axioms 26, 27). When writing a verification condition, the term "V.alt(X,i)#U" appears in the form of terms which indicate the absence of the three conditions above which would result in an undefined value for V. For each array reference in V of the form A[S] where A is an array and S is an integer expression, a term of the form $0 \leq S.alt(X,i) \leq \text{BOUND}(A)$ appears in the verification condition. For each integer operation in V of the form A op B where A and B are integer expressions and op is an integer operation, a term of the form inrange(A.alt(X,i) op B.alt(X,i)) appears in the verification condition. Since the number of terms of this form is
extremely large, and since they are needed only in special kinds of proofs, these terms are not included in the verification conditions generated by the system described in Chapter IV. For each division or modulo operation in $V$ of the form $A/B$ or $A+B$ where $A$ and $B$ are integer expressions, a term of the form $B.\text{alt}(X,i) \neq 0$ appears in the verification condition.

The second line of $\text{vcterm}$, "$N.\text{alt}(N,i+1) = V.\text{alt}(X,i)$", states that the value of $N$ at the next point along the path is the value of the right side expression at the current point. The remaining two lines indicate that $\text{:LOC}$ and $\text{:STEP}$ are incremented by one. The alteration function definition reflects the fact that the only variables whose values change at the assignment statement are the left side variable $N$, $\text{:LOC}$, and $\text{:STEP}$.

As an example, consider the assignment statement $\text{LEFT} := \text{LEFT}/A[S]$; at point $P:Qi$ with $\text{alt(LEFT,i)}=2$, $\text{alt(A,i)}=3$, $\text{alt(S,i)}=4$, $\text{alt(:LOC,i)}=5$, $\text{alt(:STEP,i)}=5$, and $\text{BOUND(A)}=10$. Then the $\text{vcterm}$ is

\[
\begin{align*}
0 & \leq S.4 \leq 10 \\
A.3[S.4] & \neq 0 \\
\text{LEFT}.3 & = \text{LEFT}.2/A.3[S.4] \\
\text{:LOC}.6 & = \text{:LOC}.5 +1 \\
\text{:STEP}.6 & = \text{:STEP}.5 +1
\end{align*}
\]

Definition 3.5.1b. assign (with array left side)

If $\text{ASSIGN}(P:Qi,A[\text{EXP}],V)$ and $\text{ARRAY}(A,B)$ then $\text{cond}(P:Qi,1)$ is

TRUE

and $\text{vcterm}(P:Qi,P:Qi+1,i)$ is

\[
\begin{align*}
\text{EXP.}\text{alt}(X,i) & \neq U \\
0 & \leq \text{EXP.}\text{alt}(X,i) \leq \text{BOUND}(A)
\end{align*}
\]
V.alt(X,i) # U
A.alt(A,i+1)[$]=IF $=EXP.alt(X,i)
    THEN V.alt(X,i)
    ELSE A.alt(A,i)[$]
:LOC.alt(:LOC,i+1)=:LOC.alt(:LOC,i) + 1
:STEP.alt(:STEP,i+1)=:STEP.alt(:STEP,i) + 1

and alt(y,i+1) =

if y ∈ {A,:LOC,:STEP}
    then alt(y,i)+1
    else alt(y,i).

ASSIGN(P:Qi,A[EXP],V) indicates an assignment statement at
point P:Qi with a left side of the form A[EXP] and a right side expression
of V. The first and third terms of vcterm state that EXP and V are
not undefined. These terms are handled as described for Definition 3.5.1.
The second term states that the array subscript is within the array
bounds. These three terms are a result of the assumption of normal
termination for partial correctness. The fourth term indicates that
element EXP.alt(X,i) of array A is assigned the value V.alt(X,i) and
the remaining elements are unchanged. For the assignment statement
A[$]:=E; at point P:Qi with alt(A,i)=2, alt(S,i)=3, and alt(E,i)=4,
the fourth term appears in a verification condition as "A.3[{$]=IF
$=S.3 THEN E.4 ELSE A.2[{$]". The last two terms of the vcterm and the
alteration function definition are as described in Definition 3.5.1a.

Definition 3.5.2a. case (case expression matches a label)
If CASE(P:Qi,EXP,L) and Qi+1=POINTLABELLEDWITH(P:Qi:EXP.alt(X,i))
then cond(P:Qi,i) is

    TRUE
and \text{vcterm}(P:Q_i,P:Q_{i+1},i) \text{ is}

\begin{align*}
\text{EXP.alt}(X,i) & \neq \text{U} \\
\text{LOC.alt}(i: \text{LOC}, i+1) & = P:Q_{i+1} \\
\text{EXP.alt}(X,i) & \in \text{CASELABELSET}(P:Q_i) \\
\text{STEP.alt}(i: \text{STEP}, i+1) & = \text{STEP.alt}(i: \text{STEP}, i) + 1
\end{align*}

and \text{alt}(y,i+1) =

\begin{align*}
\text{if } y & \in \{i: \text{LOC}, i: \text{STEP}\} \\
\text{then } \text{alt}(y,i) + 1 \\
\text{else } \text{alt}(y,i).
\end{align*}

\text{CASE}(P:Q_i, \text{EXP}, L) \text{ indicates a case statement at location } P:Q_i

with case expression \text{EXP}. The value \text{L} \text{ indicates the point to which control passes if the value of } \text{EXP} \text{ does not match a case label. The assumption that } Q_{i+1} = \text{POINTLABELLEDWITH}(P:Q_i: \text{EXP.alt}(X,i)) \text{ is an assumption about the path for which the verification condition terms are to be defined. The first term of \text{vcterm} \text{ states that the case expression is not undefined and again this results from the assumption of normal termination. The second term indicates that the location counter becomes the point whose label is matched by the case expression value. The third term indicates that the case expression value is in the set of labels for point } P:Q_i. \text{ This is a result of the assumption that execution follows the current path from } P:Q_i \text{ to } P:Q_{i+1}. \text{ If the case expression is } Y+Z, \text{ the set of labels for point } P:Q_{i+1} \text{ is } \{5,10\}, \text{ alt}(Y,i)=2 \text{ and alt}(Z,i)=3, \text{ then this term appears in a verification condition as } "Y.2 + Z.3 \in \{5,10\}". \text{ The last term of \text{vcterm} and the alteration function are as discussed earlier.}

Definition 3.5.2b case (case expression does not match a label)
If CASE(P:Qi, EXP, L) and Qi+1=L then cond(P:Qi, i) is

TRUE

and vcterm(P:Qi, P:Qi+1, i) is

EXP.alt(X, i) ≠ U
:LOC.alt(:LOC, i+1) = P:Qi+1
:EXP.alt(X, i) = CASELABELSET(P:Qi)
:STEP.alt(:STEP, i+1) = :STEP.alt(:STEP, i) +1

and alt(y, i+1) =

if y ∈ {COL, STEP}
    then alt(y, i)+1
else alt(y, i).

The assumption that Qi+1=L is again an assumption about the path. The third line of the vcterm is the result of the assumption that execution follows the path from P:Qi to P:Qi+1 and that Qi+1=L.

Definition 3.5.3. enter

If ENTER(P:Qi, C) then cond(P:Qi, i) is

[:LVL.alt(:LVL, i) < maxstacksize
 ∧ :LVL.alt(:LVL, i') = :LVL.alt(:LVL, i) +1
 ∧ :RTNPT.alt(:RTNPT, i')[§] =
     IF § = :LVL.alt(:LVL, i) +1
     THEN :LOC.alt(:LOC, i) +1
     ELSE :RTNPT.alt(:RTNPT, i)[§]
 ∧ :LOC.alt(LOC, i') = C:0
 ∧ :STEP.alt(:STEP, i') = :STEP.alt(:STEP, i) +1
 +tag[C:0](X.alt(X, i'), X.alt(X, i'))]

and vcterm(P:Qi, P:Qi+1, i) is

:LVL.alt(:LVL, i) < maxstacksize
:LVL.alt(:LVL, i') = :LVL.alt(:LVL, i) +1
:RTNPT.alt(:RTNPT, i')[§] =
     IF § = :LVL.alt(:LVL, i) +1
     THEN :LOC.alt(:LOC, i) +1
     ELSE :RTNPT.alt(:RTNPT, i)[§]
:LOC.alt(LOC, i') = C:0
and \( \text{alt}(y,i') = \)

\[
\begin{align*}
&\text{if } y \in \{\text{LVL}, \text{RTNPT}, \text{LOC}, \text{STEP}\} \\
&\quad \text{then } \text{alt}(y,i') + 1 \\
&\quad \text{else } \text{alt}(y,i)
\end{align*}
\]

and \( \text{alt}(y,i'') = \)

\[
\begin{align*}
&\text{if } y \in \text{alterables}(C) \\
&\quad \text{then } \text{alt}(y,i') + 1 \\
&\quad \text{else } \text{alt}(y,i')
\end{align*}
\]

and \( \text{alt}(y,i+1) = \)

\[
\begin{align*}
&\text{if } y \in \{\text{LOC}, \text{LVL}, \text{STEP}\} \\
&\quad \text{then } \text{alt}(y,i'') + 1 \\
&\quad \text{else } \text{alt}(y,i'')
\end{align*}
\]

\text{ENTER}(P:Qi,C) \text{ indicates a call of procedure } C \text{ at location } P:Qi. \text{ The values } i' \text{ and } i'' \text{ in the cond and vcterm refer to points which fall between } P:Qi \text{ and } P:Qi+1 \text{ during execution. The notation } y.\text{alt}(y,i') \text{ refers to the value of } y \text{ after entry to the called procedure and } y.\text{alt}(y,i'') \text{ refers to the value of } y \text{ at the exit point of the called procedure.}

The cond term for an enter statement is a single term with an implication. The lines preceding the implication reflect the entry to the called procedure and the line following the implication is the initial assertion of the called procedure. Thus, the proof of the cond term as required in a verification condition will require proof that
entry to a called procedure implies the initial assertion of the called procedure. The first line of the cond term is a result of the normal termination assumption for partial correctness and states that the return stack level before entry is less than the maximum return stack size. The second line indicates that the return stack level is incremented by one at entry and the next line indicates that the location following the location of the procedure call is placed at the top of the return stack and that all other elements are unchanged. This line appears in a verification condition in the same form as the similar line in an assignment statement with an array reference on the left side. The last two lines indicate that the location at entry is C:0 and that :STEP is incremented by one.

The first five lines of the vcterm again reflect the entry to the called procedure and are identical to the lines preceding the implication of the cond term. The next line is the final assertion of the called procedure which makes the result of procedure execution available to the verification condition. The remaining lines reflect the exit from the procedure (Axiom 51). The value of "steps(C,i)" is the number of steps in the execution of procedure C which is called at state vector i.

In the definition of the alteration function for the enter statement, reference is made to a set called "alterablesset(C)". The alterable set for a procedure is the set of variables which potentially can be altered by a call of the procedure. These include variables altered within the procedure itself and all alterable sets of procedures
called by it. The definition of the alteration function in Definitions 3.5.1 - 3.5.7 reflects the fact that the only statements which can alter program variables are assignment and read statements.

Definition 3.5.4a if (expression is true)
If IF(P:Qi, EXP, T, F) and Qi+1=T then cond(P:Qi, i) is

TRUE

and vcterms(P:Qi, P:Qi+1, i) is

EXP.alt(X, i) ≠ U
EXP.alt(X, i)
:LOC.alt(:LOC, i+1) = P:T
:STEP.alt(:STEP, i+1) = :STEP.alt(:STEP, i) + 1

and alt(y, i+1) =

if y ∈ { :LOC, :STEP}
then alt(y, i)+1
else alt(y, i).

IF(P:Qi, EXP, T, F) indicates an if statement at location P:Qi with if expression EXP. Points T and F indicate the points to which control passes if the expression evaluates true and false respectively. The assumption, Qi+1=T, is an assumption about the path.

Definition 3.5.4b if (expression is false)
If IF(P:Qi, EXP, T, F) and Qi+1=F then cond(P:Qi, i) is

TRUE

and vcterms(P:Qi, P:Qi+1, i) is

EXP.alt(X, i) ≠ U
¬EXP.alt(X, i)
:LOC.alt(:LOC, i+1) = P:F
:STEP.alt(:STEP, i+1) = :STEP.alt(:STEP, i) + 1
and \( \text{alt}(y,i+1) = \)

\[
\begin{align*}
&\text{if } y \in \{\text{LOC}, \text{STEP}\} \\
&\text{then } \text{alt}(y,i)+1 \\
&\text{else } \text{alt}(y,i).
\end{align*}
\]

Definition 3.5.5 \text{jumpto}

If \( \text{JUMPTO}(P:Qi,N) \) then cond\((P:Qi,i)\) is

\( \text{TRUE} \)

and \( \text{vcterm}(P:Qi,P:Qi+1,i) \) is

\[
\begin{align*}
\text{:LOC.\text{alt}(\text{:LOC},i+1)\text{=}P:N} \\
\text{:STEP.\text{alt}(\text{:STEP},i+1)\text{=}STEP.\text{alt}(\text{:,STEP},i) +1}
\end{align*}
\]

and \( \text{alt}(y,i+1) = \)

\[
\begin{align*}
&\text{if } y \in \{\text{LOC}, \text{STEP}\} \\
&\text{then } \text{alt}(y,i)+1 \\
&\text{else } \text{alt}(y,i).
\end{align*}
\]

\( \text{JUMPTO}(P:Qi,N) \) indicates a jump to point \text{N} at location \( P:Qi \).

Definition 3.5.6 \text{read}

If \( \text{READ}(P:Qi,A) \) then cond\((P:Qi,i)\) is

\( \text{TRUE} \)

and \( \text{vcterm}(P:Qi,P:Qi+1,i) \) is

\[
\begin{align*}
\neg:\text{REOF}(\text{:RDHD,\text{alt}(\text{:RDHD},i) +1})\to \\
&[A.\text{alt}(A,i+1)[0]='F'] \\
&\land(1 \leq $ \leq \min(\text{readsize}, \text{BOUND}(A))\to \\
&\text{A.\text{alt}(A,i+1)[$]=:RDFL(\text{:RDHD.\text{alt}(\text{:RDHD},i) +1},\$)} \\
&\land(\text{readsize}+1 \leq $ \leq \text{BOUND}(A)\to \\
&A.\text{alt}(A,i+1)[$]=A.\text{alt}(A,i)[\$]) \to \\
\text{REOF}(\text{:RDHD.\text{alt}(\text{:RDHD},i) +1})\to \\
&[A.\text{alt}(A,i+1)[0]='T'] \\
&\land(1 \leq $ \leq \text{BOUND}(A)\to \\
&A.\text{alt}(A,i+1)[$]=A.\text{alt}(A,i)[\$])\to \\
&\text{:RDHD.\text{alt}(\text{:RDHD},i+1)=:RDHD.\text{alt}(\text{:RDHD},i) +1} \\
&\text{:LOC.\text{alt}(\text{:LOC},i+1)=:LOC.\text{alt}(\text{:LOC},i) +1} \\
&\text{:STEP.\text{alt}(\text{:STEP},i+1)=:STEP.\text{alt}(\text{:,STEP},i) +1}
\end{align*}
\]
and \( alt(y,i+1) = \)

\[
\begin{align*}
& \quad \text{if } y \in \{A,:RDHD,:LOC,:STEP\} \\
& \qquad \text{then } alt(y,i)+1 \\
& \qquad \text{else } alt(y,i).
\end{align*}
\]

"READ(P:Qi,A)" indicates a read statement at location \( P:Qi \) with read array \( A \). The first term of vcterm states that if the next read record is not an end-of-file then "F" is placed in element zero of the read array and the read record is placed in consecutive elements until either the read record or the array is exhausted. The second term indicates that if the next read record is an end-of-file then "T" is placed in element zero of the read array and the rest of the array is unchanged. The last three terms reflect incrementation of :RDHD,:LOC, and :STEP.

As an example, consider read statement \( \text{READ A; at location } P:Qi \) with \( \text{BOUND(A)} = 90, \text{readsize = 80}, \text{alt(A,1)} = 2, \text{and } alt(:RDHD,1) = 3 \).

Then the first two terms appear in a verification condition as

\[
\neg:\text{REOF(:RDHD.3 +1)+} \\
[A.3[0]="F"] \\
\wedge(1 \leq $ \leq 80 \rightarrow A.3[$]=:RDFL(:RDHD.3 +1,\$)) \\
\wedge(81 \leq $ \leq 90 \rightarrow A.3[$]=A.2[\$])
\]

\[
\neg:\text{REOF(:RDHD.3 +1)+} \\
[A.3[0]="T"] \\
\wedge(1 \leq $ \leq 90 \rightarrow A.3[$] = A.2[\$])
\]

If \( \text{readsize+1} > \text{BOUND(A)} \), then the last line of the first term does not appear in the verification condition.

Definition 3.5.7 write

If \( \text{WRITE(P:Qi,A)} \) then \( \text{cond(P:Qi,i)} \) is

\( \text{TRUE} \)
and \( \text{vcterm}(P:Q_i, P:Q_{i+1}, i) \) is

\[
\begin{align*}
A.\text{alt}(A,i)[0] &= \text{"T"} \Rightarrow \\
& \text{- :WEOF(:WTHD.alt(:WTHD,i) +1)} \\
& \wedge (1 \leq $ \leq \min(\text{writesize}, \text{BOUND}(A)) \Rightarrow \\
& :\text{WTFL}(:\text{WTHD.alt}(:\text{WTHD,i} +1,$)) = A.\text{alt}(A,i)[$.]) \\
& \wedge (\text{BOUND}(A)+1 \leq $ \leq \text{writesize} \Rightarrow \\
& :\text{WTFL}(:\text{WTHD.alt}(:\text{WTHD,i} +1,$)) = \text{""} )
\end{align*}
\]

\[
A.\text{alt}(A,i)[0] = \text{"T"} \Rightarrow \\
\text{:WEOF(:WTHD.alt(:WTHD,i) +1)} \\
:\text{WTHD.alt}(:\text{WTHD,i}+1) = :\text{WTHD.alt}(:\text{WTHD,i}) +1 \\
:\text{LOC.alt}(:\text{LOC,i}+1) = :\text{LOC.alt}(:\text{LOC,i}) +1 \\
:\text{STEP.alt}(:\text{STEP,i}+1) = :\text{STEP.alt}(:\text{STEP,i}) +1
\]

and \( \text{alt}(y,i+1) = \)

\[
\begin{align*}
\text{if } y \in \{ :\text{WTHD}, :\text{LOC}, :\text{STEP} \} \\
\text{then } \text{alt}(y,i)+1 \\
\text{else } \text{alt}(y,i).
\end{align*}
\]

\( \text{WRITE}(P:Q_i,A) \) indicates a write statement at location \( P:Q_i \) with write array \( A \). The write statement is the inverse of the read statement. The first term of vcterm states that if element zero of the write array does not contain "T" then the write end-of-file predicate :WEOF is defined to be false at the next write record and the characters of the write array are assigned to consecutive write record locations until either the write record or the write array is exhausted. The rest of the write record is assigned blanks. The second term indicates that if element zero of the write array does contain "T" then the write end-of-file predicate :WEOF is defined to be true at the next write record.

III.6. Verification Conditions for Partial Correctness

The next theorem shows how the above definitions are used
in proving certain relationships between state vectors. These relationships will be used to prove that the verification conditions defined in Theorems 3.6.1 - 3.6.4 are sufficient to establish partial correctness.

In the theorems which follow, reference is made to a function \texttt{calls(i,j)}. Arguments \(i\) and \(j\) must be state vector indices such that \(i < j\) and \(E[i](:LVL) = E[j](:LVL)\). The value of \texttt{calls(i,j)} is the set of state vector indices of procedure entry points between \(i\) and \(j\) and at the next level. That is,

\[
\text{calls}(i,j) = \{ c | i < c < j, c = \text{entry}(c) \land E[i](:LVL)+1 = E[c](:LVL) \}.
\]

The format employed in the theorems is interpreted as follows. Terms written in a column are connected by logical conjunction and a solid horizontal line denotes an implication sign. The dashed horizontal lines have no formal significance and are included merely for readability. Any line preceded by "<PRV>" must be proved from the lines appearing above it.

The next theorem contains two conditions. The first is labelled VC since from it will emerge the verification condition definition, and the second is labelled SV since it states a relationship among state vectors. The theorem asserts that \(VC \Rightarrow SV\).

Condition VC is stated in terms of points along the current path and the alteration function. It is satisfied if the initial assertion of the current procedure and a property B on the first path points imply the cond term at point i and if all of these together with the vcterm at point i imply property D on the first i+1 path points.
Condition SV is stated in terms of the state vectors which correspond to the path points. The assumptions of SV are

1. normal termination,

2. the initial assertion of the current procedure is satisfied at entry,

3. property B is true for the state vectors corresponding to the first i path points, and

4. for any procedure called at path point i, the initial assertion at entry implies the final assertion at exit.

Condition SV is satisfied when these assumptions imply property D on the state vectors for the first i+1 path points and for any procedure called at point i, the initial assertion is true at entry.

More briefly stated, the theorem states that if the verification condition terms lead from property B over the first i path points to property D over the first i+1 path points then property B over the state vectors corresponding to the first i path points implies property D over the state vectors corresponding to the first i+1 path points.

The term alt(x,i') is the alteration counter for x at entry to the procedure called at point i along the path and E[qi'] is the corresponding state vector (qi'=1+qi). If point i does not call a procedure, these values do not exit. The term alt(x,i'') and state vector E[qi''] are the alteration counter and state vector at the exit point of the procedure called at point i. Again, these values may not exist.
Theorem 3.6.1

Consider path $P:Q_1, \ldots, P:Q_r$ with corresponding state vector sequence $E[q_1], \ldots, E[q_r]$. For $1 \leq i \leq r-1$, if

VC:

1. $\text{tag}[P:0](X.0,X.0)$
2. $B(X.0,X.\text{alt}(X,1),X.\text{alt}(X,1'),X.\text{alt}(X,1''), \ldots, X.\text{alt}(X,i))$

---

3. $\langle \text{PRV} \rangle$ cond($P:Q_i,i$)
4. $\text{vcterm}(P:Q_i,P:Q_i+1,i)$
5. $D(X.0,X.\text{alt}(X,1),X.\text{alt}(X,1'),X.\text{alt}(X,1''), \ldots, X.\text{alt}(X,i+1))$

then

SV:

1. $\text{NT}(n)$
2. $n \geq qr$
3. $\text{tag}[P:0](E[\text{entry}(q_1)](X),E[\text{entry}(q_1)](X))$
4. $B(E[\text{entry}(q_1)](X),E[q_1](X),E[q_1'](X),E[q_1''](X), \ldots, E[q_1](X))$
5. $c \in \text{calls}(q_i,q_i+1) \times \text{tag}[E[c](\text{:LOC})](E[c](X),E[c](X)) \Rightarrow$
   \[
   \text{tag}[E[\text{exit}(c)](\text{:LOC})](E[c](X),E[\text{exit}(c)](X))
   \]
6. $D(E[\text{entry}(q_1)](X),E[q_1](X),E[q_1'](X),E[q_1''](X), \ldots, E[q_i+1](X))$
7. $c \in \text{calls}(q_i,q_i+1) \Rightarrow \text{tag}[E[c](\text{:LOC})](E[c](X),E[c](X))$

The lines are numbered to simplify references to them during the proof. VC3 will denote line 3 of VC.

Proof of Theorem 3.6.1:

We want to prove VC $\rightarrow$ SV. Condition VC consists of two implications, VC1 $\land$ VC2 $\rightarrow$ VC3 and VC1 $\land$ VC2 $\land$ VC3 $\land$ VC4 $\rightarrow$ VC5. The first implication results from the "$\langle \text{PRV} \rangle$" at the cond term and the second results from the solid line. Condition VC is stated in terms of arbitrary values such as $X.0$ and $X.\text{alt}(X,k)$ for which specific values may be substituted. We now substitute values for symbols according to the pairing in Table III.2.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>X.0</td>
<td>E<a href="X">entry(q1)</a></td>
</tr>
<tr>
<td>X.alt(X,k)</td>
<td>E<a href="X">qk</a></td>
</tr>
<tr>
<td>X.alt(X,k')</td>
<td>E<a href="X">qk'</a></td>
</tr>
<tr>
<td>X.alt(X,k'')</td>
<td>E<a href="X">qk''</a></td>
</tr>
</tbody>
</table>

Table III.2. Pairing of Verification Condition Symbols and Values at State Vectors.

Condition VC is now in the following form.

VC:

1. \( \text{tag}[P:0](E[\text{entry(q1)}](X), E[\text{entry(q1)}](X)) \)
2. \( B(E[\text{entry(q1)}](X), E[q1](X), E[q1'](X), E[q1''](X), \ldots, E[q_i](X)) \)

\[ \text{<PRV> cond}(P:Q_i,i) \]
\[ \text{vcterm}(P:Q_i,P:Q_{i+1},i) \]

\[ D(E[\text{entry(q1)}](X), E[q1](X), E[q1'](X), E[q1''](X), \ldots, E[q_{i+1}](X)) \]

where cond and vcterm denote cond and vcterm with the substitutions defined above. Note that with the substitutions, VC1 \( \equiv \) SV3, VC2 \( \equiv \) SV4, and VC5 \( \equiv \) SV6.

We now claim that if we can show (1) \( \text{SV1} \land \text{SV2} \land \text{SV5} \land \text{VC3} \rightarrow \text{VC4} \) and (2) \( \text{VC3} \land \text{VC4} \rightarrow \text{SV7} \) then we have shown VC \( \rightarrow \) SV. This claim is supported below.

Since SV3 \( \equiv \) VC1 and SV4 \( \equiv \) VC2 then (1) becomes (1')

\( \text{SV1} \land \text{SV2} \land \text{SV3} \land \text{SV4} \land \text{SV5} \land \text{VC3} \rightarrow \text{VC1} \land \text{VC2} \land \text{VC3} \land \text{VC4} \). Since SV3 \land SV4 \( \equiv \) VC1 \land VC2 and VC1 \land VC2 \rightarrow VC3 then (1') becomes (1'') \( \text{SV1} \land \text{SV2} \land \text{SV3} \land \text{SV4} \land \text{SV5} \land \text{VC1} \land \text{VC2} \land \text{VC3} \land \text{VC4} \).

Since VC5 \( \equiv \) SV6 then (2) becomes (2') \( \text{VC3} \land \text{VC4} \land \text{VC5} \rightarrow \text{SV6} \land \text{SV7} \) which implies that (2'') \( \text{VC1} \land \text{VC2} \land \text{VC3} \land \text{VC4} \land \text{VC5} \rightarrow \text{SV6} \land \text{SV7} \). Conditions (1) and (2) imply conditions (1'') and (2'') which are clearly sufficient to prove VC \( \rightarrow \) SV.
The remainder of the proof will show that, for each Nucleus reduced program statement, the verification condition terms \texttt{cond} and \texttt{vcterm} are defined in a way such that (1) \( SV_1 \land SV_2 \land SV_5 \land VC_3 \rightarrow VC_4 \) and (2) \( VC_3 \land VC_4 \rightarrow SV_7 \). For all except the enter statement, these conditions can be reduced even further. For these statements, \texttt{calls}(q_i, q_{i+1})\) is empty and the cond term is "TRUE", therefore \( SV_5, SV_7, \) and \( VC_3 \) are satisfied. Condition (2) then is also satisfied and condition (1) is reduced to \( SV_1 \land SV_2 \rightarrow VC_4 \).

\textbf{Case 1a.} \texttt{ASSIGN}(P:Q_i,N,V) and \texttt{SIMPLE}(N).

We must show \( SV_1 \land SV_2 \rightarrow VC_4 \), that is, we must show \( NT(n) \land n \geq qr \rightarrow \texttt{vcterm}(P:Q_i,P:Q_{i+1},i) \). The \texttt{vcterm} with substitution of values for symbols as specified in Table III.2 is

\begin{align*}
1 & \quad E[q_i](V) \neq U \\
2 & \quad E[q_{i+1}](N) = E[q_i](V) \\
3 & \quad E[q_{i+1}](:LOC) = E[q_i](:LOC) + 1 \\
4 & \quad E[q_{i+1}](:STEP) = E[q_i](:STEP) + 1
\end{align*}

The conjunction \( NT(n) \land n \geq qr \) indicates that normal termination occurs and occurs no sooner than the end of the current path. This assumption of normal termination and Axiom 48 will be shown to imply the above relationship between state vectors \( q_i \) and \( q_{i+1} \). The first line of Axiom 48 and normal termination imply \( \texttt{name}(N,q_i) \neq U \land E[q_i](V) \neq U \). This implies line one of \texttt{vcterm}. The remainder of Axiom 48 parallels the second and third lines of \texttt{vcterm}. The last line of \texttt{vcterm} reflects the incrementation of \( :STEP \) which is an assertion variable for counting the steps in the execution.
Case 1b. ASSIGN(P:Qi,A[EXP],V) and ARRAY(A,B).

The vcterm with substitution is

1. \(E[qi](\text{EXP}) \neq U\)
2. \(0 \leq E[qi](\text{EXP}) \leq \text{BOUND}(A)\)
3. \(E[qi](V) \neq U\)
4. \(E[qi+1](A[\$]) = \begin{cases} E[qi](\text{EXP}) & \text{if } \$_i = E[qi](\text{EXP}) \\ \text{ELSE } E[qi](A[\$]) & \end{cases}\)
5. \(E[qi+1](\text{:LOC}) = E[qi](\text{:LOC}) + 1\)
6. \(E[qi+1](\text{:STEP}) = E[qi](\text{:STEP}) + 1\)

From Axiom 48 and the normal termination assumption, we get \(\text{NAME}(A[\text{EXP}], qi) \neq U \land E[qi](V) \neq U\). ARRAY(A,B) and Axiom 11 imply that \(\text{NAME}(A[\text{EXP}], qi) \neq U\) only if \(E[qi](\text{EXP}) \neq U\) and \(0 \leq E[qi](\text{EXP}) \leq \text{BOUND}(A)\). These are the first two lines of the vcterm. The remaining lines are handled as in case 1a.

Case 2a. CASE(P:Qi,EXP,L) and Qi+1=POINTLABELLEDWITH(P:Qi:E[qi](EXP))

The vcterm with substitution is

1. \(E[qi](\text{EXP}) \neq U\)
2. \(E[qi+1](\text{:LOC}) = P:Qi+1\)
3. \(E[qi](\text{EXP}) \in \text{CASELABELSET}(P:Qi)\)
4. \(E[qi+1](\text{:STEP}) = E[qi](\text{:STEP}) + 1\)

Normal termination and line one of Axiom 49 imply \(E[qi](\text{EXP}) \neq U\) which is the first line of the vcterm. The assumption that execution proceeds along the path from \(P:Qi\) to \(P:Qi+1\) and that \(Qi+1=\text{POINTLABELLEDWITH}(P:Qi:E[qi](\text{EXP}))\) yields the second line of vcterm, and this assumption together with Axiom 49 implies the third line of vcterm. The :STEP line results from the definition of :STEP.

Case 2b. CASE(P:Qi,EXP,L) and Qi+1=L

The vcterm with substitution is
The argument presented for case 2a holds except the assumption that $Q_{i+1} = L$ implies a change in the third vcterm line.

**Case 3. ENTER(P:Q_i,C)**

For the enter statement, we must prove $SV_1 \land SV_2 \land SV_5 \land VC_3 \Rightarrow VC_4$ and $VC_3 \land VC_4 \Rightarrow SV_7$. The first of these requires proof that normal termination, the implication of the final assertion at exit from the assumption of the initial assertion at entry for the procedure called at $P:Q_i$, and the initial assertion of the called procedure, implies $vcterm(P:Q_i,P:Q_{i+1},t)$ with substitution as defined earlier. The condition with substitution is:

1. $E[q_i](LVL) < maxstacksize$
2. $\forall E[q_i'](LVL) = E[q_i](LVL) + 1$
3. $\forall E[q_i'](RTNPT[\$]) = IF $\$ = E[q_i](LVL) + 1$ THEN $E[q_i](LOC) + 1$ ELSE $E[q_i](RTNPT[\$])$

and the vcterm with substitution is:

1. $E[q_i](LVL) < maxstacksize$
2. $E[q_i'](LVL) = E[q_i](LVL) + 1$
3. $E[q_i'](RTNPT[\$]) = IF $\$ = E[q_i](LVL) + 1$ THEN $E[q_i](LOC) + 1$ ELSE $E[q_i](RTNPT[\$])$
4. $E[q_i'](LOC) = C:0$
5. $E[q_i'](STEP) = E[q_i](STEP) + 1$
6. $tag[C:EXITPOINT(C)](E[q_i'](X),E[q_i''](X))$
7. $E[q_i''](LVL) \geq 0$
8. $E[q_i+1](LOC) = E[q_i''](RTNPT[E[q_i''](LVL)])$
9. $E[q_i+1](LVL) = E[q_i''](LVL) - 1$
10. $E[q_i+1](STEP) = E[q_i''](STEP) + 1$
The first line of Axiom 50 together with the assumption of normal termination implies \( \text{NAME}("\text{RTPN}[i+1]", i+1) \neq U \) and by Axioms 9 and 11 this implies \( E[q_i](\text{LVL}) < \text{maxstacksize} \) which is line one of cond and vcterm. The next three lines of cond and vcterm result from the last three lines of Axiom 50. We have now satisfied the terms before the implication of cond and are free to use the result. VC3 and SV5 imply line six of vcterm. The normal termination assumption and the first line of Axiom 51 imply \( E[q_i'] \geq 0 \) which is line seven of vcterm. The remainder of Axiom 51 implies lines eight and nine.

The second proof required is VC3 \( \land \) VC4 \( \land \) SV7. The set \( \text{calls}(qi, qi+1) \) has one element which is \( qi' \). Since \( E[q_i'](\text{LOC})=C:0 \) then SV7 becomes \( c \in \{qi'\} \to \text{tag}[C:0](E[q_i'](X), E[q_i'](X)) \) which is implied by VC3 \( \land \) VC4.

**Case 4a.** IF(P;Qi,EXP,T,F) and Qi+1=T

The vcterm with substitution is

1. \( E[q_i](\text{EXP}) \neq U \)
2. \( E[q_i](\text{EXP}) \)
3. \( E[q_i+1](\text{LOC})=P;Qi+1 \)
4. \( E[q_i+1](\text{STEP})=E[q_i](\text{STEP}) +1 \)

Normal termination and line one of Axiom 53 imply line one of vcterm.

The remainder of Axiom 53 with the assumption that Qi+1=T implies line two of vcterm. Line three comes from the assumption.

**Case 4b.** IF(P;Qi,EXP,T,F) and Qi+1=F

The vcterm with substitution is
1 E[q1](EXP) # U
2 ¬E[q1](EXP)
3 E[q1+1](LOC) = P: Qi+1
4 E[q1+1](STEP) = E[q1](STEP) + 1

Same argument as for case 4a.

**Case 5. JUMPTO(P: Qi, N)**

The vterm with substitution is

1 E[q1+1](LOC) = P: N.
2 E[q1+1](STEP) = E[q1](STEP) + 1

Line one is a direct result of Axiom 54.

**Case 6. READ(P: Qi, A)**

The vterm with substitution is

1 ¬: REOF(E[q1](RDHD) + 1) →
   [E[q1+1](A[0])] = "F"
   \[1 \leq S \leq \min(\text{readsize}, \text{BOUND}(A)) →
   E[q1+1](A[S]) = \text{RDFL}(E[q1](RDHD) + 1, S)\]
   \[\text{readsize} + 1 \leq S \leq \text{BOUND}(A) →
   E[q1+1](A[S]) = E[q1](A[S])\]
2 : REOF(E[q1](RDHD) + 1) →
   [E[q1+1](A[0])] = "T"
   \[1 \leq S \leq \text{BOUND}(A) →
   E[q1+1](A[S]) = E[q1](A[S])\]
3 E[q1+1](RDHD) = E[q1](RDHD) + 1
4 E[q1+1](LOC) = E[q1](LOC) + 1
5 E[q1+1](STEP) = E[q1](STEP) + 1

If ¬: REOF(E[q1](RDHD) + 1), then by Axiom 56 we get the vterm. If
: REOF(E[q1](RDHD) + 1) then Axiom 55 yields the vterm.

**Case 7. WRITE(P: Qi, A)**

The vterm with substitution is

1 E[q1](A[0]) ≠ "T" →
   [¬: WEOF(E[q1](WTHD)+1)]
   \[1 \leq S \leq \min(\text{writesize}, \text{BOUND}(A)) →
   : \text{WTFL}(E[q1](WTHD)+1, S) = E[q1](A[S])\]
   \[\text{BOUND}(A)+1 \leq S \leq \text{writesize} →
   : \text{WTFL}(E[q1](WTHD)+1, S) = " "\]
2 \text{E[q]}(A[0]) = 'T' \rightarrow \\
\text{: WEOF(E[q]}(\text{: WTHD}) + 1) \\
3 \text{E[q+1]}(\text{: WTHD}) = \text{E[q]}(\text{: WTHD}) + 1 \\
4 \text{E[q+1]}(\text{: LOC}) = \text{E[q]}(\text{: LOC}) + 1 \\
5 \text{E[q+1]}(\text{: STEP}) = \text{E[q]}(\text{: STEP}) + 1

IF \text{E[q]}(A[0]) != 'T' then Axiom 58 implies the \text{vcterm}. IF \text{E[q]}(A[0]) = 'T' then Axiom 57 implies the \text{vcterm}. This completes the proof of Theorem 3.6.1.

The next two theorems define the verification conditions for partial correctness. They state the property of the state vector sequence which is implied by the verification condition. This result will be used in proving the validity of the inductive assertion method as defined for Nucleus programs.

Theorem 3.6.2.

Consider path P:Q1,...,P:Qr with P:Q1\#INITIALPROCEDURE:0 and with corresponding state vector sequence E[q1],...,E[qr].

If the verification condition

\[ \text{tag}[P:0](X.0,X.0) \]
\[ \text{tag}[P:Q1](X.0,X.alt(X,1)) \]

\[ \text{<PRV>} \text{ cond}(P:Q1,1) \]
\[ \text{ vcterm}(P:Q1,P:Q2,1) \]
\[ ... \]
\[ \text{<PRV>} \text{ cond}(P:Qr-1,r-1) \]
\[ \text{ vcterm}(P:Qr-1,P:Qr,r-1) \]

\[ \text{tag}[P:Qr](X.0,X.alt(X,r)) \]

is satisfied then

\[ \text{NT}(n) \]
\[ n \geq qr \]
\[ \text{tag}[P:0](E[entry(q1)](X),E[entry(q1)](X)) \]
\[ \text{tag}[P:Q1](E[entry(q1)](X),E[q1](X)) \]
c ∈ calls(q1,qr) \land \text{tag}[E[c](:LOC)](E[c](X),E[c](X)) \land \text{tag}[E[exit(c)](:LOC)](E[c](X),E[exit(c)](X))
\text{tag}[P:qr](E[entry(q1)](X),E[qr](X))
\Rightarrow c ∈ calls(q1,qr) \rightarrow \text{tag}[E[c](:LOC)](E[c](X),E[c](X))

This theorem defines the verification condition for any path which starts at some point other than INITIALPROCEDURE:0. The theorem states that if the verification condition is satisfied, then whenever the corresponding path is traversed during execution, if the initial assertion of the current procedure is satisfied at entry and the assertion at the beginning of the path is satisfied and if for all called procedures along the path the initial assertion at entry implies the final assertion at exit, then when the end of the path is reached, the assertion at the end of the path is satisfied and the initial assertion of each procedure called along the path is true at its entry.

Proof of Theorem 3.6.2:

For every 1 ≤ i ≤ r, we define

\text{INT}_i(X.0,X.\text{alt}(X,1),X.\text{alt}(X,1'),X.\text{alt}(X,1''),...,X.\text{alt}(X,i)) to be

\text{tag}[P:Q1](X.0,X.\text{alt}(X,1))
\text{vcterm}(P:Q1,P:Q2,1)
...
\text{vcterm}(P:Q_i-1,P:Q_i,i-1)

Because the verification condition is satisfied, it follows from the definition of \text{INT}_i that for 1 ≤ i ≤ r-1,

\text{tag}[P:0](X.0,X.0)
\text{INT}_i(X.0,X.\text{alt}(X,1),X.\text{alt}(X,1'),X.\text{alt}(X,1''),...,X.\text{alt}(X,i))
\(<PRV>\) \quad \text{cond}(P:Q_i,1)
\quad \text{vcterm}(P:Q_i,P:Q_{i+1},i)

\hspace{2cm}\text{INT}_{i+1}(X.0,X.\text{alt}(X,1),X.\text{alt}(X,1'),X.\text{alt}(X,1''),...,X.\text{alt}(X,i+1))

Then by Theorem 3.6.1 we get for \(1 \leq i \leq r-1\),

\begin{align*}
\text{NT}(n) \\
n \geq qr \\
\text{tag}[P:0](E[\text{entry}(q_1)](X),E[\text{entry}(q_1)](X)) \\
\text{INT}_{i}(E[\text{entry}(q_1)](X),E[q_1](X),E[q'_1](X),E[q''_1](X),...,E[q_i](X)) \\
c \in \text{calls}(q_i,q_{i+1}) \wedge \text{tag}[E[c](:\text{LOC})](E[c](X),E[c](X)) \to \\
\text{tag}[E[\text{exit}(c)](:\text{LOC})](E[c](X),E[\text{exit}(c)](X)) \\
\text{INT}_{i+1}(E[\text{entry}(q_1)](X),E[q_1](X),E[q'_1](X),E[q''_1](X),...,E[q_{i+1}](X)) \\
c \in \text{calls}(q_i,q_{i+1}) \to \text{tag}[E[c](:\text{LOC})](E[c](X),E[c](X))
\end{align*}

The combination of this set of \(r-1\) state vector implications yields

\begin{align*}
\text{A:} \\
n \geq qr \\
\text{tag}[P:0](E[\text{entry}(q_1)](X),E[\text{entry}(q_1)](X)) \\
\text{tag}[P:Q_1](E[\text{entry}(q_1)](X),E[q_1](X)) \\
c \in \text{calls}(q_1,qr) \wedge \text{tag}[E[c](:\text{LOC})](E[c](X),E[c](X)) \to \\
\text{tag}[E[\text{exit}(c)](:\text{LOC})](E[c](X),E[\text{exit}(c)](X)) \\
\text{INT}_{r}(E[\text{entry}(q_1)](X),E[q_1](X),E[q'_1](X),E[q''_1](X),...,E[q_{r}](X)) \\
c \in \text{calls}(q_1,qr) \to \text{tag}[E[c](:\text{LOC})](E[c](X),E[c](X))
\end{align*}

which has been labelled A for later reference. The verification condition and the definition of \text{INT}_{r} yield

\begin{align*}
\text{tag}[P:0](X.0,X.0) \\
\text{INT}_{r}(X.0,X.\text{alt}(X,1),X.\text{alt}(X,1'),X.\text{alt}(X,1''),...,X.\text{alt}(X,r)) \\
\text{tag}[P:Q_r](X.0,X.\text{alt}(X,r))
\end{align*}

which is expressed in terms of arbitrary values. Substituting specific values in the same manner as in the proof of Theorem 3.6.1 we get

\begin{align*}
\text{tag}[P:0](E[\text{entry}(q_1)](X),E[\text{entry}(q_1)](X)) \\
\text{INT}_{r}(E[\text{entry}(q_1)](X),E[q_1](X),E[q'_1](X),E[q''_1](X),...,E[q_{r}](X)) \\
\text{tag}[P:Q_r](E[\text{entry}(q_1)](X),E[q_{r}](X))
\end{align*}

By applying this result to A above we get
NT(n)
n ≥ qr
\(\text{tag}[P:0][E[\text{entry}(q1)](X), E[\text{entry}(q1)](X)]\)
\(\text{tag}[P:q1][E[\text{entry}(q1)](X), E[q1](X)]\)
c ∈ \text{calls}(q1, qr) \land \text{tag}[E[c](\text{:LOC})](E[c](X), E[c](X)) \land
\text{tag}[E[\text{exit}(c)](\text{:LOC})](E[c](X), E[\text{exit}(c)](X))
\text{tag}[P:qr][E[\text{entry}(qr)](X), E[qr](X)]\)
c ∈ \text{calls}(q1, qr) \land \text{tag}[E[c](\text{:LOC})](E[c](X), E[c](X))

which completes the proof of Theorem 3.6.2.

Theorem 3.6.3 defines the verification condition for a path which begins at INITIALPROCEDURE:0. It is the same as Theorem 3.6.2 except that four lines are added at the front to indicate the initial values of :LVL, :RDHD, :WTHD, and :STEP.

Theorem 3.6.3

Consider path \(P:q1, ..., P:qr\) with \(P:q1=\text{INITIALPROCEDURE}:0\) and with corresponding state vector sequence \(E[q1], ..., E[qr]\).

If the verification condition

\[
\text{tag}[P:0](X.0,X.0)
\text{:LVL}.0=-1
\text{:RDHD}.0=0
\text{:WTHD}.0=0
\text{:STEP}.0=0
\]

\[
\text{<PRV> cond}(P:q1,1)
\text{vcterm}(P:q1,P:q2,1)
\ldots
\text{<PRV> cond}(P:qr-1,r-1)
\text{vcterm}(P:qr-1,P:qr,r-1)
\]
\text{tag}[P:qr](X.0,X.\text{alt}(X,r))

is satisfied, then

NT(n)
n ≥ qr
\(\text{tag}[P:0][E[\text{entry}(q1)](X), E[\text{entry}(q1)](X)]\)
\(c ∈ \text{calls}(q1, qr) \land \text{tag}[E[c](\text{:LOC})](E[c](X), E[\text{exit}(c)](X)) \land
\text{tag}[E[\text{exit}(c)](\text{:LOC})](E[c](X), E[\text{exit}(c)](X))\)
\[ \text{tag[P:Qr]}(E[\text{entry}(q1)](X), E[qr](X)) \]
\[ c \in \text{calls}(q1,qr) \rightarrow \text{tag}[E[c](:\text{LOC})](E[c](X), E[c](X)) \]

Proof of Theorem 3.6.3:

The four extra lines reflect the initial values defined in Axioms 18–20 and the definition of :STEP. Since P:0 is P:Q1 then tag[P:Q1] which appears in Theorem 3.6.2 can be treated as true and the proof of Theorem 3.6.2 then applies to Theorem 3.6.3 also.

The following three lemmas give properties needed for the proof of Theorem 3.6.4.

**Lemma 1**

Consider the finite execution sequence \(E[e], \ldots, E[x]\) of procedure P.

Assume that the partial correctness verification condition for each path in P is satisfied and that there are no procedure calls in sequence \(E[e], \ldots, E[x]\). Then \(\text{tag[P:0]}(E[e](X), E[e](X)) \rightarrow \text{tag[P:EXITPOINT(P)]}(E[e](X), E[x](X))\).

**Proof:**

The sequence \(E[e], \ldots, E[x]\) can be broken down into subsequences which correspond to paths of procedure P. Since all verification conditions are true, then by Theorems 3.6.2 and 3.6.3, if the initial assertion and the assertion at the front of the path are true, then the assertion at the end of the path is true. Chaining these implications together we get \(\text{tag[P:0]}(E[e](X), E[e](X)) \rightarrow \text{tag[P:EXITPOINT(P)]}(E[e](X), E[x](X))\).
Lemma 2

Assume the same conditions as Lemma 1 except that the sequence $E[e], \ldots, E[x]$ does contain procedure calls and $E[x]$ corresponds to a halt or exit of procedure $P$. Then $\text{tag}[P:0](E[e](X), E[e](X)) \rightarrow \text{tag}[P:EXITPOINT(P)](E[e](X), E[x](X))$.

Proof:

Again the sequence $E[e], \ldots, E[x]$ can be broken down into subsequences which correspond to paths of procedure $P$. This time, however, there are "gaps" in some subsequences. These "gaps" correspond to procedure execution sequences for the called procedures. In order to apply Theorems 3.6.2 and 3.6.3, we need to know for each procedure call, that if the initial assertion is true at entry, then the final assertion is true at exit. If the execution of the called procedure contains no procedure calls, then we can apply Lemma 1 to get the desired condition. If not, we repeat this argument on the execution sequence of the called procedure. Since $E[e], \ldots, E[x]$ is finite, the number of procedure calls must be finite and a procedure execution with no procedure calls will ultimately be found. For this procedure execution, the initial assertion at entry implies the final assertion at exit by Lemma 1. This result together with Theorems 3.6.2 and 3.6.3 and the chaining argument of Lemma 1 is used to fill the innermost gap. Repeating this process until all gaps are filled, yields $\text{tag}[P:0](E[e](X), E[e](X)) \rightarrow \text{tag}[P:EXITPOINT(P)](E[e](X), E[x](X))$. 
Lemma 3

Assume the same conditions as for Lemmas 1 and 2 except that the procedure execution sequence $E[e],...,E[x]$ does not reach a halt or exit of $P$. State vector $E[x]$ thus corresponds to a procedure call which leads to termination at a halt. Then $\text{tag}[P:0](E[e](X),E[e](X)) \rightarrow \text{tag}[E[x+1](::\text{LOC})](E[x+1](X),E[x+1](X))$.

Proof:

The sequence $E[e],...,E[x]$ can be broken down into subsequences which correspond to paths of procedure $P$; however, there will be a sequence $E[w],...,E[x]$ left over. This sequence corresponds to the front of the path (not necessarily unique) which was being executed when the procedure call was made. Sequence $E[w],...,E[x]$ corresponds to a procedure entry path. Since all verification conditions are true, the verification condition for the path (or paths) headed by this procedure entry path is true. By Theorems 3.6.2 and 3.6.3, the initial assertion of all called procedures are true at entry. Then by chaining the implications for all complete paths, as in Lemmas 1 and 2, together with the implication for the procedure entry path, we get $\text{tag}[P:0](E[e](X),E[e](X)) \rightarrow \text{tag}[E[x+1](::\text{LOC})](E[x+1](X),E[x+1](X))$. This completes the proof of the three lemmas.

The next theorem asserts the validity of the inductive assertion method as defined above for Nucleus.

Theorem 3.6.4

Consider Nucleus program NP with initial assumption $A(X,0,X)$ and
desired result $R(X.0,X)$. If NP is properly tagged and the partial
correctness verification condition for each path is satisfied, then
$E[0](A(X.0,X)) \land NT(n) \rightarrow E[n](R(X.0,X))$, that is, NP is partially
correct with respect to initial assumption $A(X.0,X)$ and $R(X.0,X)$.

Proof of Theorem 3.6.4:

Given $E[0](A(X.0,X)) \land NT(n)$ and that all partial correctness
verification conditions are satisfied, we are to prove $E[n](R(X.0,X))$.
Since the program terminates normally at state vector n, we have a
finite state vector sequence $E[0], \ldots, E[n]$ representing the program
execution. This state vector sequence may be broken down into
subsequences corresponding to executions of procedures. A procedure
execution may be any of three different types. These are (1) a
procedure execution which calls no other procedures, (2) a procedure
execution which contains procedure calls and which reaches its own
halt or exit, and (3) a procedure execution which contains procedure
calls and which does not reach its own halt or exit (a called procedure
leads to execution of a halt).

If the execution of the starting procedure contains no
procedure calls, then by Lemma 1, $E[0](A(X.0,X)) \land NT(n) \rightarrow E[n](R(X.0,X))$.
If the execution of the starting procedure does contain procedure
calls, but still reaches a halt or exit of the starting procedure, then,
by Lemma 2, $E[0](A(X.0,X)) \land NT(n) \rightarrow E[n](R(X.0,X))$. If the execution
of the starting procedure contains procedure calls and never reaches a
halt or exit of the starting procedure, then, by Lemma 3, the last
procedure called from the starting procedure is entered with its initial assertion satisfied. For this procedure, we reapply the above argument. Since the execution sequence E[0],...,E[n] is finite, we must eventually reach the procedure which executes the halt and its initial assertion is satisfied at entry. By Lemmas 1 and 2, we get E[n](R(X.0,X)) at the exit or halt of this procedure. Thus, E[0](A(X.0,X))∧NT(n) → E[n](R(X.0,X)). This concludes the proof.

3.7. Verification Conditions for Total Correctness

In this section we discuss the verification conditions necessary to prove total correctness. The proofs of total correctness differ from those for partial correctness in two ways. First, the verification condition terms must be modified, and second, each assertion must place a finite upper bound on :STEP.

Any condition which causes abnormal termination can be assumed absent when normal termination is assumed; however, if normal termination is to be demonstrated, then these conditions must be proved absent. Thus many terms which are included in the vcterm for partial correctness must be moved to the cond term for total correctness. The new cond terms will be listed with the understanding that these terms are all removed from the vcterm and that the remainder of the vcterm and the alteration counter function are unchanged. Assume P:Qi is the \(i\)-th point along a path.

Definition 3.6.1a assign

If ASSIGN(P:Qi,N,V) and SIMPLE(N) then cond(P:Qi,i) is
V.alt(X,i)#U

Definition 3.7.1b
If ASSIGN(P:Qi,A[EXP],V) and ARRAY(A,B) then cond(P:Qi,i) is

\[ EXP.alt(X,i)#U \]
\[ 0 \leq EXP.alt(X,i) \leq BOUND(A) \]
\[ V.alt(X,i)#U \]

Definition 3.7.2a,b case
If CASE(P:Qi,E,L) then cond(P:Qi,i) is

\[ E.alt(X,i)#U \]

Definition 3.7.3 enter
If ENTER(P:Qi,C) then cond(P:Qi,i) is

\[ :LVL.alt(:LVL,i) < maxstacksize \]
\[ [:LVL.alt(:LVL,i) < maxstacksize \]
\[ a:LVL.alt(:LVL,i')=LVL.alt(:LVL,i)+1 \]
\[ a:RTNPT.alt(:RTNPT,i')[:LVL.alt(:LVL,i)+1]=LOC.alt(:LOC,i)+1 \]
\[ a:LOC.alt(:LOC,i')=C:0 \]
\[ a:STEP.alt(:STEP,i')=STEP.alt(:STEP,i)+1 \]
\[ \rightarrow \text{tag}[C:0](X.alt(X,i'),X.alt(X,i'))] \]

Definition 3.7.4a,b if
If IF(P:Qi,E,T,F) then cond(P:Qi,i) is

\[ E.alt(X,i)#U \]

Definition 3.7.5 jumpto
If JUMPTO(P:Qi,N) then cond(P:Qi,i) is

TRUE
Definition 3.7.6 read
If READ(P:Qi,A) then cond(P:Qi,i) is
TRUE

Definition 3.7.7 write
If WRITE(P:Qi,A) then cond(P:Qi,i) is
TRUE

This modification of the verification condition terms makes certain that abnormal termination is proved not to occur. The requirement of a finite upper bound on :STEP at each assertion insures that for each tagged point P:Qi, there is an upper bound on qi where E[qi] corresponds to P:Qi. The maximum of all such qi's, say max qi, indicates that E[max qi] is the latest state vector which can correspond to a tagged point. The tagged points are chosen in a way such that only a finite number of execution steps may occur before another tagged point is achieved. Thus, there can be only a finite number of state vectors following E[max qi]; and, in fact, since normal termination must occur at a tagged point, E[max qi] is the upper bound on any execution.

The verification conditions modified as described above are sufficient to prove total correctness.
CHAPTER IV

A NUCLEUS VERIFIER AND ITS PROOF

IV.1. Introduction

This chapter describes the correctness proof of a Nucleus program which generates the Nucleus verification conditions defined in Chapter III. The listing of the program with assertions is in Appendix D.

The motivation for this proof is simple. In order for proofs of correctness by the inductive assertion method to be feasible for a large class of programs, it must be possible to generate the verification conditions automatically, and if the proof based on these verification conditions is to be valid, the verification conditions must be correctly generated. This Nucleus program then is proposed as the base system in a sequence of verified verifiers of increasing sophistication leading eventually to a verified system with a full set of verification services.

The proof of this verifier employs a unique combination of proof techniques. The basic technique is the inductive assertion method based on inductive assertions and verification conditions; however, a set of procedures which performs the traversal of transition networks is proved correct by proving equivalence of the procedures and the transition network definition. The equivalence proof made it unnecessary to construct inductive assertions to describe the intermediate stages of transition network traversal.
Section IV.2 outlines the approach used to construct and prove the program and describes the overall structure of the resulting system. The structure and proof of the top-level procedure of the system, VERIFY are described in Section IV.3. Sections IV.4-IV.6 discuss the recognition component, which is headed by procedure PARSE, the transition network traversal component headed by TRANSNET, and the verification condition generation component headed by procedure VCGEN. A statistical summary of the proof is presented in Section IV.7 to indicate the general magnitude of the project, and Section IV.8 is a subjective evaluation of the validity of the proof with comments about the probability of the existence of verifier errors and the implications of their existence.

IV.2. Methodology

The program construction and proof were performed in parallel following basically a top-down structured programming approach whereby the top-level procedures were written and proved before proceeding on to the next level. Each procedure is kept simple by breaking complex computations into small parts to be handled by separate procedures. The code itself is highly structured and contains no GO TO statements. The combination of these approaches results in procedures which tend toward simple structures, and this simple structure introduces a corresponding simplicity to the proof.

The program operation is straightforward and follows the general pattern illustrated in Figure IV.1. The input is a program with assertions. If the program is identified as legal, the verification
condition generation begins; otherwise, an error message is printed and execution of the system terminates. Verification condition generation proceeds through the procedures in the order that they appear in the input file. If an untagged loop is encountered, a message is printed and execution halts; otherwise, execution continues through generation of all verification conditions.

The top-level procedure is VERIFY, which establishes the overall program structure. The structure and proof of VERIFY are discussed in Section IV.3. The recognition component is headed by procedure PARSE and the verification condition generation component is headed by procedure VCGEN. The structure and the proof of these procedures are discussed in Sections IV.4 and IV.6. The proof of the recognition component includes equivalence proofs between the implementation and the definition, together with inductive assertion proofs. The
proof of the verification condition generation component is based entirely on the inductive assertion method.

The inductive assertion method verification conditions for the proof were generated by a modified version of a Snobol program [14]. This Snobol program is not verified which means that the verification conditions must be hand checked before proof. No simplification of the verification conditions is performed by the Snobol program, thus placing full responsibility for each detail of the proof process on the user.

The Snobol verification condition generator of [14] was modified by adding a feature which makes top-down proofs practical by allowing generation of verification conditions for procedures that call other procedures not yet completely specified. The system treats PENDING as an additional reserved word which can be used as a program statement. The PENDING statement is defined as a statement which changes all program variables to some unknown value in the value space. It is used to represent a section of code which is not yet written, hence the term "pending". The Snobol verifier does not generate verification conditions for any procedure containing a PENDING statement, but the initial and final assertions of these procedures are used to generate verification conditions for procedures which call the ones containing PENDING. Thus the verification conditions can be generated for top-level procedures before the lower-level procedures are written. The assumption is that the lower-level procedures eventually will be completed; and when they are, they will be proved correct with respect
to their initial and final assertions.

IV.3. VERIFY

Procedure VERIFY is the top-level procedure of the system and therefore establishes the system structure. The procedure with assertions is listed below. The numbers in parentheses at the left are not a part of the procedure. They correspond to the control points of the reduced program and are included for reference purposes.

PROCEDURE VERIFY;

(0.1) ASSERT READFILE(:RDHD)=INPUTSTRING;
(0) ENTER SETUP;
(1) ENTER PARSE;
(2) ENTER VCGEN;
(3.1) ASSERT INPUTSTRING IS A LEGAL NUCLEUS PROGRAM;
(3.2) ASSERT WRITEFILE(:WTHD)=LISTING AND VERIFICATION CONDITIONS FOR INPUTSTRING;
(3) EXIT;

Procedure SETUP initializes several variables such as stack pointers and is proved using the inductive assertion method. Procedure PARSE is the recognition component and is discussed in Section IV.4. Procedure VCGEN is the verification condition generator and is discussed in Section IV.6.

Partial correctness for VERIFY is stated in terms of an initial assumption and desired result. VERIFY is partially correct if it can be shown that for any execution which terminates normally, the desired result is satisfied. The initial assumption is, "The Nucleus transition network and all system constants are restored in their designated locations and the readfile contains the input string." The
initial assumption is stated in terms of the initial assertion of
VERIFY and the two predicates NETWORKS(FIRSTARC, RECOGNITIONSTATE, SYMBOL,
TEST, ACTION, FLAG, NEXTSTATE, PARSESTART, SCANSTART) and CONSTANTS(PRESET,
RESERVEDWORDSET, RESWORDPTS, RESCODE, LOOP, DASHES, DOTS, ASRTTRUE,
BLANKLINE, PATHIS, LINE1, LINE2, LINE3, LINE4, COLONWORDSET, COLONPTS,
LVLLINE). These two predicates are assumed at each tagged point in
addition to any additional stated assertions at that point. Since none
of the arguments are altered anywhere in the program, the predicates
need not be proved along a path. The arguments of NETWORKS are the
locations in which the Nucleus transition network is prestore and
the arguments of CONSTANTS are the locations where predefined constants
are stored.

The desired result is, "The input string contains no syntax
or untagged loop errors and its listing and verification conditions
are contained on the output file or the input string contains a
syntax or untagged loop error." The first part of the desired result
is stated in terms of the final assertion of VERIFY. Each halt
statement in the program is tagged with an assertion which declares
that a syntax or untagged loop error has occurred.

Three other assumptions were made in order to simplify the
proof. It is assumed for purposes of the proof that all array subscripts
are within the declared bounds, the return point stack does not overflow,
and there is no integer overflow. Nucleus provides run-time checks on
each of these conditions so no such error will occur without some
notification. The terms dealing with these conditions are not included in the verification conditions generated by the Snobol system for this proof. Therefore, there is a slight difference between the verification conditions used to prove the Nucleus system and the verification conditions generated by the Nucleus system.

The verification condition for the only path in procedure 

VERIFY is listed below.

0.A :LVL.0 = -1
0.B :RDHD.0 = 0
0.C :WTHD.0 = 0
0.D :STEP.0 = 0
0.1 READFILE(:RDHD.0)=INPUTSTRING

0 <PRV>TRUE
0  CURRENTPROC.1 = -1
0  STATEMENTPT.1 = -1
0  ASRTLOCPT.1 = -1
0  ASRTLOC1.1[0] = 0
0  0 <= $$ <= 1999 + ASRTLOC.1[$$] = -1
0  EXPLISTPT.1 = -1
0  EXPSTRING.1[0] = 0
0  EXPSTRING.1[1] = -1
0  CASELABELSETPT.1 = -1
0  CASELABELFRONT.1[0] = 0
0  DEFINEDCASELABELSETTOP.1 = -1
1 <PRV>READFILE(:RDHD.0)=INPUTSTRING
1 <PRV>CURRENTPROC.1 = -1
1 <PRV>STATEMENTPT.1 = -1
1 <PRV>ASRTLOCPT.1 = -1
1 <PRV>ASRTLOC1.1[0] = 0
1 <PRV>0 <= $$ <= 1999 + ASRTLOC.1[$$] = -1
1 <PRV>EXPLISTPT.1 = -1
1 <PRV>EXPSTRING.1[0] = 0
1 <PRV>EXPSTRING.1[1] = -1
1 <PRV>CASELABELSETPT.1 = -1
1 <PRV>CASELABELFRONT.1[0] = 0
1 <PRV>DEFINEDCASELABELSETTOP.1 = -1

PROGRAM(ALTSET.1,ASRTLOC.2,ASRTLOC1.2,ASRTS.1,
BOUNDFUNCTION.1,CASELABELFRONT.2,CASELABELS.1,
CASELABELSET.1,CHARLIST.1,DEFINEDIDENTIFIERSET.1
DEFINEDIDENTIFIERSETPT.1,DEFINEDPROCEDURESET.1,
DEFINEDPROCEDURESETPT.1, DESCLOC.1, EXPLIST.1,
INITIALPROCEDURE.1, PROC.1, PROCALLS.1, STATEMENT.1)
1 WRITEFILE(:WTHD.1)=LISTING
1 LINEPT.1=0 ∧ LINE.1[0]=≠F
2 <PRV> PROGRAM(ALTSET.1, ASRTLOC.2, ASRTLOC1.2, ASRTS.1,
  BOUNDFUNCTION.1, CASELABELFRONT.2, CASELABELS.1,
  CASELABELSET.1, CHARLIST.1, DEFINEDIDENTIFIERSET.1,
  DEFINEDIDENTIFIERSETPT.1, DEFINEDPROCEDURESET.1,
  DEFINEDPROCEDURESETPT.1, DESCLOC.1, EXPLIST.1,
  INITIALPROCEDURE.1, PROC.1, PROCALLS.1, STATEMENT.1)
2 <PRV> LINEPT.1=0 ∧ LINE.1[0]=≠F
2 PROGRAM(ALTSET.2, ASRTLOC.3, ASRTLOC1.3, ASRTS.2,
  BOUNDFUNCTION.2, CASELABELFRONT.3, CASELABELS.2,
  CASELABELSET.2, CHARLIST.2, DEFINEDIDENTIFIERSET.2,
  DEFINEDIDENTIFIERSETPT.2, DEFINEDPROCEDURESET.2,
  DEFINEDPROCEDURESETPT.2, DESCLOC.2, EXPLIST.2,
  INITIALPROCEDURE.2, PROC.2, PROCALLS.2, STATEMENT.2)
2 WRITEFILE(:WTHD.2)=WRITEFILE(:WTHD.1) AND
verification conditions for INPUTSTRING

3.1 INPUTSTRING IS A LEGAL NUCLEUS PROGRAM
3.2 WRITEFILE(:WTHD.2)=LISTING AND verification
conditions for INPUTSTRING

The numbers at the left refer to the statements from which the
terms result. The <PRV> lines correspond to the initial assertions of
the called procedures. Immediately below the initial assertions are
the final assertions of the same procedures. The first four lines of
the verification condition 0.A, 0.B, 0.C, and 0.D appear only on this
path since it starts at point zero of the initial procedure. They
indicate the initial values of the return stack level, the read and
write record pointers, and verifier variable :STEP which counts state
vectors. Line 0.1 is the initial assertion of VERIFY and lines 3.1
and 3.2 are the final assertion of VERIFY. PROGRAM is a predicate which
asserts that INPUTSTRING is a legal Nucleus program and that the
reduced program is represented by the values listed as its arguments.
The bodies of procedures PARSE and VCGEN used in the generation of this verification condition were represented by PENDING statements. Thus the alteration counter for each variable is increased at the procedure call for each of these two procedures.

IV.4. PARSE

The recognition component which is headed by procedure PARSE, performs the syntax check of the input string to identify it as a legal or illegal Nucleus program, and maps the legal program strings into reduced program form. This component also produces a program listing with control points inserted to aid the user in associating corresponding parts of the program and the verification conditions.

Procedure PARSE is listed below with the assertions used in its proof. Statements 0-12 initialize the traversal of the Nucleus parsing network, and then the while loop repeatedly calls TRANSNET, which traverses from one state to the next in the transition network each time it is called. ATPARSESTATE in assertion 13.1 is an assertion predicate which is true whenever the transition network is at a parse state. STATE is the current network state (either the scanning or parsing) and PARSESTATE is the current parsing network state. Each of these is set to the initial parsing network state PARSESTART (statements 0,1). The first card is read into array CARD (2) and the output line is set to empty (3,4). If the first card is not an end-of-file (5) then it is placed in the listing by procedure LIST (6).
PROCEDURE PARSE;

(0.1) ASSERT READFILE(:RDHD)=INPUTSTRING;
(0.2) ASSERT CURRENTPROC= -1;
(0.3) ASSERT STATEMENTPT= -1;
(0.4) ASSERT ASRTOCPT= -1;
(0.5) ASSERT ASRTOCL[0]= 0;
(0.6) ASSERT 0 ≤ $$≤ 1999 → ASRTLOC[$$]= -1;
(0.7) ASSERT EXPLISTPT= -1;
(0.8) ASSERT EXPSTRING[0]= 0;
(0.9) ASSERT EXPSTRING[1]= -1;
(0.10) ASSERT CASELABELSETP= -1;
(0.11) ASSERT CASELABELFRONT[0]= 0;
(0.12) ASSERT DEFINECASELABELSETP= -1;

(0) STATE:=PARSESTART;
(1) PARSESTATE:=PARSESTART;
(2) READ CARD;
(3) LINEPT:=0;
(4) LINE[0]:=↑F;
(5) IF CARD[0]#↑T
(6) THEN ENTER LIST;
FI;

(7) COL:=1;
(8) RECOGNITION:=FALSE;
(9) ASRSCANFLAG:=FALSE;
(10) ENTER SCAN;
(11) CARINSTRING:=TOKEN;
(12) RTNSTACKTOP:=−1;
(13.1) ASSERT ATPARSESTATE;
(13.2) ASSERT ¬ASRSCANFLAG;
(13.3) ASSERT LINEPT=0 ∧ LINE[0]=↑F;
(13.4) ASSERT WRITEFILE(:WTHD)=LISTING OF INPUTSTRING
THROUGH :RDHD;
(13) WHILE ¬RECOGNITION DO
(14) ENTER TRANSNET;
(15) ELIHW;
(16.1) ASSERT PROGRAM(ALTSET,ASRTLOC,ASRTLOC1,ASRTS,
BOUNDFUNCTION,CASELABELFRONT,CASELABELS,CASELABELSET,
CHARLIST,DEFINEDIDENTIFIERS,DEFINEDIDENTIFIERS,P,
DEFINEDPROCEDURESET,DEFINEDPROCEDURESETP,DESCLOC,
EXPLIST,INITIALPROCEDURE,PROC,PROCALLS,STATEMENT);
(16.2) ASSERT WRITEFILE(:WTHD)=LISTING;
(16.3) ASSERT LINEPT=0 ∧ LINE[0]=↑F;
(16) EXIT;

COL points to the current input character and is initially set to one

(7). RECOGNITION becomes TRUE when transition network traversal achieves
recognition and ASRTSCANSFLAG is TRUE when a scan is being performed on the characters in an assertion during verification condition generation. Each of these is initially set to FALSE (8,9). The scan of the first token is performed (10) and CARINSTRING is set to the code integer for this token (11). The return stack is set to its empty condition (12) and the transition network traversal loop is entered. If a syntax error is discovered, a halt statement is executed in TRANSNET, thus PARSE need test only for recognition of the input string and does not need to be concerned with error checks.

The proof of procedure PARSE is an inductive assertion method proof. Even though the proof of TRANSNET does not employ the inductive assertion method, it is assigned initial and final assertions which are used in the proof of PARSE. These assertions are shown in the next section, which describes TRANSNET and its proof. This is the point at which the equivalence proofs are joined to the inductive assertion proof.

IV.5. TRANSNET

Procedure TRANSNET is the heart of the recognition component and is listed below with its initial, final, and halt assertions. Procedure TRANSNET can be seen to be of the same structure as the axiomatization of transition network traversal contained in Appendix A. A description of the program variables and network representation codes provides the remaining detail needed to verify that TRANSNET and its four utility procedures ALPHABETMATCH, NILMATCH, STACKTEST, and TRAVERSE are equivalent to their definitions.
PROCEDURE TRANSNET;

(0.1) ASSERT [-ASRTSCANFLAG ∧ ATSCANSTATE ∧ ATSAVEPARENSTATE] ∨ [-ASRTSCANFLAG ∧ ATPARSESTATE] ∨ [ASRTSCANFLAG ∧ ATASRTSCANSTATE];

(0.2) ASSERT ¬RECOGNITION;

(0.3) ASSERT LINEPT=0 ∧ LINE[0]=F;

(0.4) ASSERT WRITEFILE(:WTHD)=LISTING OF INPUTSTRING THROUGH: RDHD;

(0) ENTER ALPHABETMATCH;
(1) IF ALPHABETMATCHFLAG THEN ENTER TRAVERSE;
(2) ELSE ENTER NILMATCH;
(3) IF NILMATCHFLAG THEN ENTER TRAVERSE;
(4) ELSE IF ARC < FIRSTARC[STATE+1] ∧ SYMBOL[ARC] < 0 THEN RTNSTACKTOP:=RTNSTACKTOP+1;
(5) RTNSTACK[RTNSTACKTOP]:=ARC;
(6) STATE:=SYMBOL[ARC];
(7) ELSE IF RECOGNITIONSTATE[ARC] THEN ENTER STACKTEST;
(8) IF STACKTESTFLAG THEN ARC:=RTNSTACK[RTNSTACKTOP];
(9) RTNSTACKTOP:=RTNSTACKTOP-1;
(10) ENTER TRAVERSE;
(11) ELSE IF RTNSTACKTOP < 0 THEN RECOGNITION:=TRUE;
(12) ELSE WRITE NONRECOGNITION;

(24.1) ASSERT INPUTSTRING NOT RECOGNIZED BY TRANSITION NETWORK;
(24.2) ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) AND NONRECOGNITION MESSAGE;

(24) HALT;
(25) FI;
(26) FI;
(27) ELSE WRITE NONRECOGNITION;

(27.1) ASSERT INPUTSTRING NOT RECOGNIZED BY TRANSITION NETWORK;
(27.2) ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) AND NONRECOGNITION MESSAGE;

(27) HALT;
(28) FI;
(29) FI;
(30) FI;
(31) FI;

(28.1) ASSERT ASRTSCANFLAG.0 → ATASRTSCANSTATE;
(28.2) ASSERT ¬ASRTSCANFLAG.0 ∧ ATSCANSTATE.0 → ATSCANSTATE ∧ ATSAVEPARENSTATE;
(28.3) ASSERT ¬ASRTSCANFLAG.0 ∧ ATPARSESTATE.0 → ATPARSESTATE;
(28.4) ASSERT LINEPT=0 ∧ LINE[0]=F;
(28.5) ASSERT WRITEFILE(:WTHD)=LISTING OF INPUTSTRING THROUGH :RDHD;

(28) EXIT;
To examine in more detail the relationship between a procedure and its definition, consider procedure STACKTEST which is shown below. The portion of the definition (Appendix A) which corresponds to procedure STACKTEST is "RTNSTACK.i#{ }∧ test(car(RTNSTACK.i))(REGVAL.i)" which appears as a test condition in the definition of the transition network traversal.

```plaintext
PROCEDURE STACKTEST;
  IF RTNSTACKTOP ≥ 0
    THEN S:=ARC;
    ARC:=RTNSTACK[RTNSTACKTOP];
    ENTER TESTS;
    ARC:=S;
    IF TESTFLAG
      THEN STACKTESTFLAG:=TRUE;
      RETURN;
    FI;
  FI;
STACKTESTFLAG:=FALSE;
EXIT;
```

Integer variable RTNSTACKTOP points to the top of the transition network return stack and a value of -1 is used to indicate an empty stack. Therefore, the first test in STACKTEST is satisfied only if the return stack is not empty. Integer variable S is a temporary used to save the value of ARC since procedure TESTS may alter ARC. Integer variable ARC is the current transition network arc (these arcs are represented as integers). RTNSTACK is an integer array and RTNSTACK[0],...RTNSTACK[RTNSTACKTOP] is the stack of arcs making up the current transition network return stack. Procedure TESTS evaluates the test associated with ARC and sets boolean variable TESTFLAG to the result. We can now see that if the return stack is not
empty, then the test on the arc at the top of the return stack is evaluated and the result is stored in TESTFLAG. The second test checks this result and if the test is true, boolean variable STACKTESTFLAG is set to "TRUE". If the test is not true or if the first test determines that the return stack is empty, then STACKTESTFLAG is set to "FALSE". Thus it can be seen that procedure STACKTEST is equivalent to the corresponding test in the definition.

Each action and test of the Nucleus definition network is implemented as a separate procedure. Each of these procedures is, as nearly as possible, a direct encoding of its definition counterpart. The action procedure correctness proofs are equivalence proofs much like those for the TRANSNET procedures and the proofs for the test procedures employ the inductive assertion method.

The recognition component also contains several other small utility procedures which support the procedures discussed above. The inductive assertion method is used in their correctness proofs.

IV.6. VCGEN

The verification condition generation component headed by procedure VCGEN, is activated when PARSE notes that TRANSNET has achieved recognition of the input string. This identifies the string as a legal Nucleus program, and since the transition network handles the generation of the reduced program, the reduced program now exists for VCGEN. Verification condition generation proceeds as long as no untagged loops are found. An untagged loop causes an error message
to be printed and execution terminates.

Procedure VCGEN is listed below with assertions.

PROCEDURE VCGEN;

(0.1) ASSERT LINEPT=0 ∧ LINE[0]=↑F;
(0) P:=0;
(1.1) ASSERT 0 ≤ P ≤ DEFINEDPROCEDURESETPT+1;
(1.2) ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) AND
VERIFICATION CONDITIONS FOR ALL PROCEDURES IN
DEFINED.PROCEDURE.SET FROM 0 THROUGH P-1;
(1.3) ASSERT LINEPT=0 ∧ LINE[0]=↑F;
(1) WHILE P ≤ DEFINEDPROCEDURESETPT DO
    CURRENTPROC:=0;
    (2) ASSERT 0 ≤ P ≤ DEFINEDPROCEDURESETPT;
    (3.1) ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0)
AND VERIFICATION CONDITIONS FOR ALL PROCEDURES
IN DEFINED.PROCEDURE.SET FROM 0 THROUGH P-1;
(3.3) ASSERT LINEPT=0 ∧ LINE[0]=↑F;
(3) WHILE DEFINEDPROCEDURESET P≠PROC[7*CURRENTPROC] DO
    CURRENTPROC:=CURRENTPROC+1;
(4) ELIHW;
(5) BIAS:=PROC[7*CURRENTPROC+1];
(6) ENTER PROCVCGEN;
(7) P:=P+1;
(8) ELIHW;
(10.1) ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) AND
VERIFICATION CONDITIONS FOR INPUTSTRING;
(10) EXIT;

Note that the assertion containing the predicate PROGRAM which was
included in the initial assertion of VCGEN for the proof of VERIFY has
now been omitted. This assertion is assumed to be included at all
tagged points in the verification condition generation procedures. Since
none of its arguments are altered in these procedures, it need not be
included except when needed in the proof of a verification condition.

The defined procedures of the reduced program are indexed by
pointers in DEFINEDPROCEDURESET[0],... , DEFINEDPROCEDURE
SETPT]. Thus, in the outer loop of VCGEN, P takes on the values
of these indices. For each index \( P \), the inner loop looks up the
pointers to the description of \( P \) in array PROC and then calls procedure
PROCVCGEN which generates all of the verification conditions for the
procedure indexed by \( P \). The inner while loop of VCGEN thus control
the generation of the verification conditions for the procedure indexed
by \( P \).

The character array LINE is used in building up strings of
characters to be printed. Integer variable LINEPT is a pointer to
the last character stored. The definition of a write statement states
that element zero of the output array is not printed, but is used to
indicate whether or not an end-of-file is to be written, where a "T"
in element zero indicates write end-of-file. Therefore, the assertion
LINEPT=0 \( \land \) LINE[0]=\( {}^* \)F, where "\( {}^* \)" is the Nucleus quote symbol, indicates
that the LINE is in its empty state and is ready to accept the
characters of an output line.

The correctness proofs for the procedures in this component
of the Nucleus system are inductive assertion method proofs. Again
the verification conditions used in the proof were generated by the
Snobol program described earlier.

IV.7. Proof Profile

The intent of this section is to portray the general magnitude
of the program and proof. The entire system is composed of 203
procedures. Nearly all of these are less than one page long including
assertions. The inductive assertion method was used in the proof of
100 of these procedures, leaving 103 equivalence proofs. The 103 equivalence proofs are broken down further into 98 action procedures, which were proved equivalent to the Nucleus network actions, and 5 transition network traversal procedures, which were proved equivalent to the transition network axiomatization of Appendix A.

Table IV.1 summarizes a subjective categorization of the difficulty of proof for the terms in ten sample verification conditions from the inductive assertion method proofs of the 100 procedures.

<table>
<thead>
<tr>
<th>Category</th>
<th>Number</th>
<th>Per Cent</th>
</tr>
</thead>
<tbody>
<tr>
<td>immediate proofs</td>
<td>32</td>
<td>40</td>
</tr>
<tr>
<td>simple proofs</td>
<td>25</td>
<td>31</td>
</tr>
<tr>
<td>more difficult proofs</td>
<td>23</td>
<td>29</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>80</strong></td>
<td><strong>100</strong></td>
</tr>
</tbody>
</table>

Table IV.1. Difficulty of Proof of Terms from Ten Verification Conditions.

mentioned above. The category "immediate" includes simple tautologies and any term which is identical to some term which can be used in its proof. The category "simple" includes proofs which involve one or two simple steps such as substitution. All other terms are categorized as "more difficult" even though many of these proofs are still quite straightforward.

If we assume that the 100 procedures whose proofs employed the inductive assertion method average five verification conditions
and that the verification conditions average eight terms to be proved (see Table IV.1) then we have 4000 terms to be proved assuming no reproofs due to modifications of the program or proof. According to Table IV.1, we can expect that approximately 70 per cent of these, or 2800, will be immediate or simple proofs and that approximately 30 per cent, or 1200, will be more difficult. Thus there are 1200 proofs which require some effort and 4000 proofs which must be recorded and saved.

These numbers all assume no modifications in the program or proof. Reproofs introduced by modifications increase these numbers significantly.

IV.8. Validity of the Proof

Although there are no known errors in the Nucleus verifier or its proof, there undoubtedly are errors in each. This section describes two likely causes of such errors and discusses methods of finding and eliminating them.

One source of errors is the lack of terminology and proofs of properties needed to support assertions about Nucleus reduced programs and their executions. Even though Nucleus has been rigorously defined, this definition has not yet been used to develop a full set of terms for the components of programs and executions. The terminology necessary for this research was developed in earlier sections (execution procedure of a state vector, entry state vector, exit state vector, execution sequence, path, correspondence between paths and
execution sequences, etc.), however, many of the properties of programs and executions were assumed without proof. For example, it is assumed that the level of the return stack after exit from a procedure is the same as the level before entry. Even though these properties seem clear, they are a potential source of errors since the use of intuition is quite tempting and leads to increasingly liberal assumptions.

The Nucleus definition provides a firm theoretical base on which techniques for proving properties of the reduced programs and their executions can be built. Using these techniques, the properties of Nucleus which are assumed, can be proved (or disproved), thus eliminating this potential error source.

The magnitude of the proof, as described in the previous section, is a second indication and probable cause of errors. The only automatic assistance used was the Snobol program which generated verification conditions and the output from it required a hand check.

Two approaches to this problem would be repeated careful checks of the proof by other people, and checks of the proof by various mechanical systems. The second of these seems most likely and most appealing. Even though the results of an unverified mechanical aid can not be accepted as proof, it is generally easier to check results by hand than to generate them by hand.
CHAPTER V

CONCLUSION

The development of a verified program verifier for Nucleus programs is organized into three steps. Each step is composed of a formulation and a set of proofs that the formulation satisfies certain requirements. Step one is the formal definition of the Nucleus syntax and semantics. The proofs required at this step are a proof of consistency for the Nucleus definition, and proofs of properties of Nucleus programs as needed for the proofs in steps two and three. These include such properties as: Each control point has exactly one reduced program instruction.

At step two of the verifier development, we define correctness of execution for Nucleus programs and formulate the inductive assertion method for proving correctness. This formulation defines a set of verification conditions. The set of proofs for step two show that the inductive assertion method as formulated is valid by showing that the verification conditions are sufficient to prove correctness.

Step three is the implementation of a generator for the verification conditions defined in step two. This verifier is written in Nucleus and operates on programs written in Nucleus. The proof at step three is required to show that the verifier correctly generates the Nucleus verification conditions.

The current state of the development of these three steps is detailed in the preceding chapters. Chapter II reviews the Nucleus
programming language and the methods used in its formal definition. This definition, in terms of transition networks and axioms, provides the basis for the formulation of the inductive assertion method for Nucleus programs and for the proof strategy used in verifying the Nucleus verifier. The consistency of the Nucleus definition and the properties of Nucleus used in this work, though carefully considered, were assumed without proof. A consistency proof for the Nucleus definition, and the formulation and application of a technique for proving properties about Nucleus are projects recommended as worthy of further attention. For example, it should be possible to associate assertions with states in transition networks and apply the inductive assertion method to prove properties of Nucleus reduced programs. The formulation of such a technique would provide added appeal to the method used to define Nucleus.

Chapter III presents the theoretical basis for the verification of Nucleus programs by the inductive assertion method. Termination of Nucleus programs is separated into two types, normal and abnormal. From two different treatments of normal termination, partial and total correctness, a set of verification conditions is defined. These verification conditions introduce input and output operations to the inductive assertion method. The sets of verification conditions are shown to be sufficient to prove partial and total correctness. Thus, stage two of the development of a verified verifier is complete.

Chapter IV describes the construction and correctness proof
of the verified program verifier. The correctness proof employs a
unique approach to program verification which combines the inductive
assertion method with equivalence proofs. This initial verified
verifier provides a starting point for a sequence of verified verifiers
of increasing sophistication, while the proof technique provides
a model for proofs of similar programs such as a Nucleus compiler.
These projects are also recommended as worthy of further attention.
The verifier together with its correctness proof represent the completion
of stage three.

Among the factors contributing to the achievement of this
verifier, undoubtedly the most important is the formal definition of
Nucleus in terms of transition networks and axioms. Other contributing
factors are the Snobol verification condition generator, the top-down
structured approach to the construction of the program, and the free-
form assertions which provided the flexibility needed to describe the
intermediate stages of development. These together with the unique
proof strategy using the inductive assertion method in combination with
equivalence proofs made the proof possible. In return, this research
offers the verifier itself as the base system in a sequence of verified
verifiers of increasing sophistication, the proof strategy which can be
employed in similar proofs such as for a Nucleus compiler, and a
proof of a moderate-sized program as a contribution to the general
pool of program correctness experience.
APPENDIX A

TRANSITION NETWORK DEFINITION

TRANSITION NETWORK = (STATES, ALPHABET, REGISTERS, TESTS, ACTIONS, ARCS, RECOGNITIONSTATES, STARTSTATE)

where:

(1) STATES is a set

(2) ALPHABET is a set such that ALPHABET \cap STATES = \{ \}

(3) REGISTERS is a set of register name/register range pairs.

(N1,R1),(N2,R2) in REGISTERS \land (N1,R1) \neq (N2,R2) \rightarrow N1 \neq N2

REGISTER NAMES = \{N such that (N,R) in REGISTERS\}

REGISTER RANGES = \{R such that (N,R) in REGISTERS\}

RANGE : REGISTER NAMES \rightarrow REGISTER RANGES such that

RANGE(N) = R iff (N,R) in REGISTERS

REGISTER VALUES = \{V such that V is a set of pairs

(N1,V1) in V \rightarrow [N1 in REGISTER NAMES \land V1 in RANGE(N1)

\land V2 \neq V1 \rightarrow (N1,V2) not in V]\}

(4) TESTS \subseteq \{P such that P : REGISTER VALUES \rightarrow \{TRUE,FALSE\}\}

(5) ACTIONS \subseteq \{DO such that DO : REGISTER VALUES \rightarrow REGISTER VALUES\}

(6) ARCS \subseteq STATES \times SYMBOLS \times TESTS \times ACTIONS \times SCANFLAG \times STATES

where

SYMBOLS = ALPHABET \cup STATES \cup \{NIL\}

NIL not in ALPHABET \cup STATES

SCANFLAG = \{SCAN, NOSCAN\}

let arc1 = (Q1,S1,P1,D1,F1,Q2), arc2 = (Q1,S2,P2,D2,F2,Q3),

arc1 \neq arc2, arc1 in ARCS, arc2 in ARCS.
then (a) \( S_1 \neq S_2 \lor \neg(P_1(x) \land P_2(x)) \)

(b) \( S_1 \text{ in } \text{STATES} \rightarrow S_2 \text{ not in } \text{STATES} \land Q_1 \text{ not in } \text{RECOGNITIONSTATES} \)

(7) \text{RECOGNITIONSTATES} \subseteq \text{STATES}

(8) \text{STARTSTATE IN STATES}
INPUTSTRING = A0, A1, A2, ..., An where each Ai in ALPHABET

DECLARATIVES

NAMESPACE = \{STATE, REGVAL, INSTRING, RTNSTACK\}

EVALUATIVES

STATE.0 = STARTSTATE

REGVAL.0 in REGISTER VALUES

INSTRING.0 = INPUTSTRING

RTNSTACK.0 = \{ \}

\begin{align*}
\text{stateof}(x) &= \text{car}(x) & \text{first element} \\
\text{symbol}(x) &= \text{cadr}(x) & \text{second element} \\
\text{test}(x) &= \text{caddr}(x) & \text{third element} \\
\text{action}(x) &= \text{caddrr}(x) & \text{fourth element} \\
\text{flag}(x) &= \text{caddddr}(x) & \text{fifth element} \\
\text{nextstate}(x) &= \text{cadddddr}(x) & \text{sixth element}
\end{align*}
IMPERATIVES

if [\exists \text{ARC} \in \text{ARCS} : \text{stateof}(\text{ARC}) = \text{STATE}.i, \text{symbol}(\text{ARC}) = \text{car}(\text{INSTRING}.i), \text{test}(\text{ARC})(\text{REGVAL}.i)]
then $E[i+1] = \text{A}[\text{INSTRING}, \text{if flag}(\text{ARC}) = \text{SCAN} \text{ then cdr}(\text{INSTRING}.i) \text{ else INSTRING}.i, \text{A[REGVAL, action(ARC)(REGVAL}.i), A[STATE, nextstate(ARC), E[i]]]]$
else if [\exists \text{ARC} \in \text{ARCS} : \text{stateof}(\text{ARC}) = \text{STATE}.i, \text{symbol}(\text{ARC}) = \text{NIL}, \text{test}(\text{ARC})(\text{REGVAL}.i)]
then $E[i+1] = \text{A}[\text{INSTRING}, \text{if flag}(\text{ARC}) = \text{SCAN} \text{ then cdr}(\text{INSTRING}.i) \text{ else INSTRING}.i, \text{A[REGVAL, action(ARC)(REGVAL}.i), A[STATE, nextstate(ARC), E[i]]]]$
else if [\exists \text{ARC} \in \text{ARCS} : \text{stateof}(\text{ARC}) = \text{STATE}.i, \text{symbol}(\text{ARC}) \in \text{STATES}]
then $E[i+1] = \text{A}[\text{RTNSTACK}, \text{cons}(\text{ARC}, \text{RTNSTACK}.i), \text{A[STATE, symbol(ARC), E[i]]]]$
else if [\text{STATE}.i \text{ IN RECOGNITIONSTATES}]
then if [\text{RTNSTACK}.i ≠ {} \text{ and test(car(RTNSTACK).i))(REGVAL}.i)]
then $E[i+1] = \text{A}[\text{RTNSTACK}, \text{cdr}(\text{RTNSTACK}.i), \text{A[INSTRING, if flag(car(RTNSTACK).i)) = SCAN then cdr(INSTRING}.i) \text{ else INSTRING}.i, \text{A[REGVAL, action(car(RTNSTACK).i))(REGVAL}.i), A[STATE, nextstate(car(RTNSTACK).i), E[i]]]]$
else if [\text{RTNSTACK}.i = {}]
then recognition(i+1)
else non-recognition(i+1)
else non-recognition(i+1)
APPENDIX B

REDUCED PROGRAM COMPONENTS

ALL ARGUMENTS TO THE FUNCTIONS DESCRIBED BELOW ARE CHARACTER
STRINGS.  ALSO WHENEVER THE ARGUMENT C IS USED IT IS OF THE FORM
PIO WHERE P IS A STRING THAT IS THE NAME OF SOME PROCEDURE AND
Q IS A STRING OF DIGITS.

ARRAY(A+B) IS PRODUCED FOR EACH ARRAY DEFINED IN THE
DECLARATIONS.  A IS THE ARRAY NAME AND B IS THE
SUBSCRIPT UPPER LIMIT.

ASSIGN(L+N:V) IS PRODUCED WHEN POINT Q IN PROCEDURE P HAS
AN ASSIGNMENT STATEMENT WITH LEFT SIDE N AND
RIGHT SIDE V.

CASE(L+X+P) IS PRODUCED WHEN POINT Q IN PROCEDURE P HAS EITHER
A CASE STATEMENT OR A CASE-ELSE STATEMENT.  L IS
THE INTEGER VALUED CASE EXPRESSION AND P IS THE
POINT FOLLOWING THE ESAC.

CASEJOINP(L) = J IS PRODUCED IF POINT Q IN PROCEDURE P HAS
A CASE STATEMENT.  J IS THE POINT FOLLOWING THE ESAC.

CASELABELSET(L) = X IS PRODUCED IF POINT Q IN PROCEDURE P HAS
A CASE STATEMENT.  X IS THE SET OF NUMERIC CASE LABELS
THAT ARE DEFINED FOR THAT CASE STATEMENT.

ENTER(L+N) IS PRODUCED IF POINT Q IN PROCEDURE P HAS AN ENTER
STATEMENT WITH A PROCEDURE NAMED X.

EXIT(L) IS PRODUCED IF POINT Q IS THE EXIT OF PROCEDURE P.

EXITPOINT(P) = Q IS PRODUCED IF POINT Q IS THE EXIT OF
PROCEDURE P.

MALT(L) IS PRODUCED WHEN POINT Q IN PROCEDURE P HAS A MALT.

IF(L+X+T+F) IS PRODUCED WHEN POINT Q IN PROCEDURE P HAS A TWO
WAY branching.  L IS THE EXPRESSION TO BE EVALUATED
AND IF A IS TRUE CONTROL GOES TO POINT T OTHERWISE
TO POINT F.  IFX ARE PRODUCED FOR EACH NUMBER OF STATEMENTS
IF-THEN: IF(L+X+T+F) WHERE T IS THE POINT AT THE THEN
AND F IS THE POINT AFTER THE IF.
IF-THEN-ELSE: IF(L+X+T+F) WHERE T IS THE POINT AT THE
THEN AND F IS THE POINT FOLLOWING THE ELSE.
WHILE: IF(L+X+T+F) WHERE T IS THE POINT FOLLOWING THE DO AND F IS THE POINT FOLLOWING ELSE.

INITIALPROCEDURE = P IS PRODUCED FOR THE PROCEDURE P WHERE
EXECUTION OF THE PROGRAM IS TO BEGIN.

JUMPTO(L+W) IS PRODUCED WHEN POINT Q IN PROCEDURE P HAS A JUMP
TO POINT W.  JUMPS ARE PRODUCED FOR A NUMBER OF STATEMENTS.

102
CASE AND CASE-ELSE: JUMPTO(L·CASEJOINPOINT(P;C)) FOR EVERY POINT Q THAT IS AT THE END OF A BODY IN THE ALTERNATIVE SEQUENCE OF THE CASE STATEMENT AT POINT C IN PROCEDURE P.

GO-TO: JUMPTO(L·POINTLABELLEDWITH(TM(P));I) WHERE I IS THE IDENTIFIER APPEARING IN THE GO-TO.

IF-THEN-ELSE: JUMPTO(L·E) WHERE Q IS THE POINT BEFORE THE ELSE, AND E IS THE POINT FOLLOWING THE IF.

NULL: JUMPTO(L·PLUS(I)(Q))

RETURN: JUMPTO(L·EXITPOINT(P)).

WHILE: JUMPTO(L·*W) WHERE Q IS THE POINT BEFORE THE ELIM, AND W IS THE POINT AT THE WHILE.

POINTLABELLEDWITH(S) = 0

READ(L·A) IS PRODUCED WHEN POINT Q IN PROCEDURE P HAS A READ STATEMENT WITH ARRAY A.

SIMPLE(I) IS PRODUCED FOR EVERY IDENTIFIER I THAT APPEARS IN DECLARATION OF SIMPLE VARIABLES.

WRITE(L·A) IS PRODUCED WHEN POINT Q IN PROCEDURE P HAS A WRITE STATEMENT WITH ARRAY A.
APPENDIX C

THE NUCLEUS AXIOMS

PRIMITIVE FUNCTIONS

**** DENOTE THE USUAL MATHEMATICAL OPERATIONS OF INTEGER ADDITION

**** SUBTRACTION, AND MULTIPLICATION.

/ DENOTES TRUNCATED INTEGER DIVISION: THE INTEGER PART AFTER

PERFORMING REAL DIVISION.

MOD IS DEFINED AS A MOD B = A - R * (A / B) WHERE /

DENOTES INTEGER DIVISION AS DEFINED ABOVE.

<<>> DENOTE THE USUAL MATHEMATICAL ORDERING RELATIONS ON THE

INTEGERS.

>>> DENOTE THE LOGICAL OPERATIONS OF NOT AND OR AND IMPLIES.

THE ASSUMED PRECEDENCE IS - FIRST, *= LAST.

BOOLOFCHA(X) = BOOLOFINT(INTOFCHAR(X))

BOOLOFINT(X) = FALSE IF ABS(X) MOD 2 = 0

= TRUE IF ABS(X) MOD 2 = 1

CHARACTERCONSTANTTOKEN(X)

= TRUE IF THE CHARACTER STRING X WOULD BE RECOGNIZED AS A

CHARACTER CONSTANT BY THE SCANNING NETWORK;

= FALSE OTHERWISE.

CHARACTERVALUE(X) IS APPLIED TO CHARACTERCONSTANT TOKEN AND

RETURNS THE SECOND CHARACTER OF THAT TOKEN STRING.

CHAROFBOOL(X) = CHAROFINT(INTOFBOOL(X))

CHAROFINT(X) = 'B', 'L', 'A', 'K' IF ABS(X) MOD 64 = 0

= 'A' IF ABS(X) MOD 64 = 1

= 'S', 'M', 'A', 'P', 'P' IF ABS(X) MOD 64 = 63

THE ORDER USED ABOVE IS THE SAME AS THE ORDER OF APPEARANCE

IN THE BASIC NUCLEUS CHARACTER SET....


Y Z 0 1 2 3 4 5 6 7 8 9 (1 1) * * * * < $ $ $ (SHARP)

104.
DIGITS(X) = THE STRING OF DECIMAL DIGITS WITH NO EXCESS
LEADING ZEROS THAT REPRESENTS THE INTEGER X.

FASETOKEN(x) = TRUE IF THE STRING OF CHARACTERS X WOULD BE RECOGNIZED
BY THE SCANNING NETWORK AS A FALSE TOKEN.
= FALSE OTHERWISE.

IDENTIFIER_TOKEN(x) = TRUE IF THE STRING OF CHARACTERS X WOULD BE RECOGNIZED
BY THE SCANNING NETWORK AS AN IDENTIFIER TOKEN.
= FALSE OTHERWISE.

IN DENOTES THE SET MEMBERSHIP OPERATOR USUALLY DENOTED BY
EPSILON.

INTEGER_VALUE(x) = THE INTEGER VALUE OF THE DECIMAL DIGIT
STRING X.

INTOFBOOL(x) = 0 IF X = FALSE
= 1 IF X = TRUE.

INTOFCHAR(x) = 0 IF X = *(BLANK)*
= 1 IF 'A' = X
= 2 IF X = *(SHARP)*
= 63 IF X = G (THE ORDER ABOVE CORRESPONDS WITH THE ORDER OF APPEARANCE IN
THE BASIC NUCLEUS CHARACTER SET.)

NUMBER_TOKEN(x) = TRUE IF THE STRING X WOULD BE RECOGNIZED BY THE SCANNING
NETWORK AS A NUMBER TOKEN.
= FALSE OTHERWISE.

PLUS1(x) = DIGITS(INTEGER_VALUE(x) + 1)

TRUETOKEN(x) = TRUE IF THE STRING X WOULD BE RECOGNIZED BY THE SCANNING
NETWORK AS A TRUE TOKEN.
= FALSE OTHERWISE.

IMPLEMENTATION PARAMETERS

INRANGE(x) = TRUE IF THE INTEGER X IS IN THE SET OF INTEGERS
REPRESENTABLE ON THE MACHINE SUPPORTING A
GIVEN IMPLEMENTATION OF NUCLEUS.
= FALSE OTHERWISE.

MAXSTACKSIZEx IS A INTEGER DEFINING THE MAXIMUM SIZE OF THE
RETURN POINT STACK USED IN CALLING PROCEDURES.

READSIZEx IS THE FIXED NUMBER OF POSITIONS IN ANY INPUT RECORD.
WRITESIZEx IS THE FIXED NUMBER OF POSITIONS IN ANY OUTPUT RECORD.
DECLARATIVES

1 INAMESPACE(U)
2 SIMPLE(X) = INAMESPACE(X)
3 SIMPLE(*:LOC*)
4 SIMPLE(*:LVL*)
5 SIMPLE(*:ARCH*)
6 SIMPLE(*:TH*)
7 ARRAY(A+) = OOL::D(I)+8
8 O::ISINTEGERVALUE(HOUND(A)) = INAMESPACE(A +{DIGITS(I) +1}+)
9 ARRAY(*:RTNP+::DIGITS(MAX STACK SIZE))

EVALUATIVES

10 IDENTIFIETOKEN(X) = NAME(X+I)=X
11 NAME(A +{S+}+I) = IF CSS::ISINTEGERVALUE(HOUND(A)) THEN (A +{DIGITS(S+I) +1}+) ELSE U
12 NUMBERTOKEN(X) = X+I=IF INRANGE(INTEGERVALUE(X)) THEN INTEGERVALUE(X) ELSE U
13 TRUETOKEN(X) = X+I=TRUE
14 FALSETOKEN(X) = X+I=FALSE
15 CHARACTERCONSTANTTOKEN(X) = X+I=CHARACTERVALUE(X)
16 INAMESPACE(X) = X+I=IF X+U THEN U ELSE L(I)(X)
17 *:LOC*.+0 = INITIALPROCEDURE +10*
18 *:LVL*.+0 = -1
19 *:RUMD*.+0 = 0
20 *:THM*+0 = 0
21 (<<< X).+1 = X+I
22 (<<< X).+1 = IF X+I=U THEN U ELSE IF INRANGE(-X+I) THEN -X+I ELSE U
23 (X <<< Y).+1 = IF X+I=U + Y+I=U THEN U ELSE IF INRANGE(X+I-Y+I) THEN X+I-Y+I ELSE U
24 (X <<< Y).+1 = IF X+I=U + Y+I=U THEN U ELSE IF INRANGE(X+I-Y+I) THEN X+I-Y+I ELSE U
25 (X <<< Y).+1 = IF X+I=U + Y+I=U THEN U
ELSE IF INRANGE(X1+Y1) THEN X1*Y1 ELSE U

26 (X ++ Y).1 = IF X1 =U v Y1 =U v Y1 =U THEN U
ELSE IF INRANGE(X1/Y1) THEN X1/Y1 ELSE U

27 (X ++ Y).1 = IF X1 =U v Y1 =U v Y1 =U THEN U
ELSE IF INRANGE(X1 MOD Y1)
THEN X1 MOD Y1 ELSE U

28 (X ++ Y).1 = IF X1 =U v Y1 =U THEN U ELSE X1*Y1

29 (X ++ Y).1 = IF X1 =U v Y1 =U THEN U ELSE X1*Y1

30 (X ++ Y).1 = IF X1 =U v Y1 =U THEN U ELSE X1*Y1

31 (X ++ Y).1 = IF X1 =U v Y1 =U THEN U ELSE X1*Y1

32 (X ++ Y).1 = IF X1 =U v Y1 =U THEN U ELSE X1*Y1

33 (X ++ Y).1 = IF X1 =U v Y1 =U THEN U ELSE X1*Y1

34 (X ++ X).1 = IF X1 =U THEN U ELSE X1

35 (X ++ Y).1 = IF X1 =U v Y1 =U THEN U ELSE X1*Y1

36 (X ++ Y).1 = IF X1 =U v Y1 =U THEN U ELSE X1*Y1

37 (+INTOFBOOL(+ X ++)).1 = IF X1 =U THEN U ELSE INTOFBOOL(X1)

38 (+INTOFCHAR(+ X ++)).1 = IF X1 =U THEN U ELSE INTOFCHAR(X1)

39 (+BOOLOFINT(+ X ++)).1 = IF X1 =U THEN U ELSE BOOLOFINT(X1)

40 (+BOOLOFCHAR(+ X ++)).1 = IF X1 =U THEN U ELSE BOOLOFCHAR(X1)

41 (+CHAROFINT(+ X ++)).1 = IF X1 =U THEN U ELSE CHAROFINT(X1)

42 (+CHAROFBOOL(+ X ++)).1 = IF X1 =U THEN U ELSE CHAROFBOOL(X1)

43 (A +( S ++)).1 = NAME(A +( S ++)).1

44 (X ++ X).1 = X

45 PNAME(A +( S ++)).1 = X

46 LOCPLUS1(X ++ Y) = X ++ PLUS1(Y)

IMPERATIVES

47 A(N ++ S)(X) = IF N =X THEN V ELSE S(X)

48 ASS1(X): LOC(x + N + Y)
= [NAME(N1) =U v Y1 =U v TERMINATION(I)]
= [NAME(N1) =U v Y1 =U v TERMINATION(I)]
= E1[X] = A[(+LOC1+LOCPLUS1(+LOC1)+
A(N1+S1+E1[I1]))]

49 CASE (+LOC1+X+P) =
[X] = U  
TERMINATION(I)

[NAME] = RTNPT(:LVL+1)+1 = U  
TERMINATION(I)

E(I+1) = AI:*LOC*PNAME(:LOC*1) ++
( IF DIGITS(X,1) IN CASELABEL=ELSE(+:LOC*1) 
THEN POINTLABEL=WITH(:LOC*1 ++ DIGITS(X,1)) 
ELSE P) ++E(I+1)

50 ENTER(*:LOC*1+1) =
[NAME] = RTNPT(:LVL+1)+1 = U  
TERMINATION(I)

[NAME] = RTNPT(:LVL+1)+1 = U  
E(I+1) = AI:*LOC*PNAME(*:LOC*1) ++
A:*:(LVL++LVL+1++E(I+1))

51 EXIT(*:LOC*1) =
[NAME] = LVL+1 = U  
TERMINATION(I)

[NAME] = LVL+2 = U  
E(I+1) = AI:*LOC*1++RTNPT(:LVL+1)+1  
A:*:(LVL++LVL+1++E(I+1))

52 HALT(*:LOC*1) =
TERMINATION(I)

53 IF(*:LOC*1*X*P*0) =
[X] = U  
TERMINATION(I)

[X] = U  
E(I+1) = AI:*LOC*PNAME(:LOC*1) ++
( IF X[]=TRUE THEN P ELSE Q=E(I+1) )

54 JUMPETO(*:LOC*1+P) =
E(I+1) = AI:*LOC*PNAME(*:LOC*1) ++
A:*:P=E(I+)

55 READ(*:LOC*1+A) :=EOF(*:RDMD*1)+1 =
E(I+1)(x) = IF x=A ++(0) THEN ++
ELSE IF x=*:RDMD*1 THEN ++RDMD*1+1
ELSE IF x=*:LOC* THEN LOCPLUS(+:LOC*1)
ELSE E(I+1)(x)

56 READ(*:LOC*1+A) :=EOF(*:RDMD*1)+1 =
E(I+1)(x) = IF x=A ++(0) THEN ++
ELSE IF 1SJSMIN(INTEGERV(BOUND(A))++READSIZE)
A=x ++ DIGITS(J) ++
THEN IRDFL(*:RDMD*1)+1
ELSE IF x=*:RDMD* THEN ++RDMD*1+1
ELSE IF x=*:LOC* THEN LOCPLUS(+:LOC*1)
ELSE E(I+1)(x)

57 WRITE(*:LOC*1+A) = (A ++(0)++) = ++
E(I+1) = AI:*LOC*LOCPLUS(+:LOC*1)  
A:*:*THU*1++.WTHU+1*++E(I+1))

58 WRITE(*:LOC*1+A) = (A ++(0)++) = ++
E(I+1) = AI:*LOC*LOCPLUS(+:LOC*1)  
A:*:*THU*1++.WTHU+1*++E(I+1))

59 WRITE(*:LOC*1+A) = (A ++(0)++) = ++
E(I+1) = AI:*LOC*LOCPLUS(+:LOC*1)  
A:*:*THU*1++.WTHU+1*++E(I+1))

60 [NAME] = THU*1.++ J=+  
1SJSMIN(INTEGERV(BOUND(A))++READSIZE) =
I=WFL(*:THU*1++) = (A ++(0)++(U)++(THU*1++))
A:*:*THU*1++.WTHU+1*++E(I+1)
APPENDIX D

VERIFIER LISTING

INTEGER
ANDTYPEP, $POINTS TO TOP OF AND.TYPE STACK. -1 FOR EMPTY $.
ARC, $CURRENT TRANSITION NETWORK ARC DURING SCAN AND PARSE $.
ASRTRACK, $POINTS TO LAST CHARACTER IN ASSERTION BODY IN ARRAY CHARLIST $.
ASRTFRONT, $POINTS TO FIRST CHARACTER IN ASSERTION BODY IN ARRAY CHARLIST $.
ASRTLOCPT, $POINTS TO LAST ENTRY IN ARRAY ASRTLOC$.
ASHTSCAPINTER, $POINTS TO NEAT CHARACTER IN ASSERTION. USED DURING SCAN OF
ASSERTION $.
AT, $USED IN SEVERAL SEARCH PROCEDURES. RETURNS pointer TO LOCATION OF OBJECT IF
FOUND $.
B, $TEMPORARY $.
BACKLABEL, $POINTS TO LAST LABEL OF A CASE LABEL LIST IN ARRAY CASELABELS $.
BEFORETOKEN, $SAVE PREVIOUS TOKEN DURING SCAN OF ASSERTIONS. SET TO 0 AT
BEGINNING OF ASSERTION $.
BIAS, $BIAS = PROC(7 + CURRENTPROC + 1) WHICH POINTS TO THE CURRENT CONTROL POINT 0 IN
ARRAY DESCLOC $.
BINARYADDTYPEP, $POINTS TO TOP OF BINARY.ADD.TYPE STACK. -1 FOR EMPTY $.
CARD; STRING, $NEXT INPUT SYMBOI DURING TRANSITION NETWORK TRAVERSAL $.
CASEEXPRESSTP, $POINTS TO TOP OF CASE.EXPRESSION STACK. -1 FOR EMPTY $.
CASELABELSETPT, $POINTS TO LAST ENTRY IN CASELABELSET. -1 FOR EMPTY $.
CASEPOINTTOP, $POINTS TO TOP OF CASE.POINT STACK. -1 FOR EMPTY $.
CHARLIST, $POINTS TO LAST ENTRY IN ARRAY CHARLIST. -1 FOR EMPTY $.
COL, $POINTS TO A COLUMN OF ARRAY CARD. CARD(COL) IS NEXT SCAN INPUT SYMBOL $.
CURRENTPROC, $POINTS TO A PROCEDE DESCRIPTION IN ARRAY PROC $.
CURRFRONTC, $POINTS TO AN ELEMENT OF ARRAY PATHFRONTS. PATHFRONTS(CURRFRONTC) IS
CONTROL POINT FROM WHICH PATHS ARE CURRENTLY BEING GENERATED $.
DEFINEDARRAYSETPT, $POINTS TO LAST ENTRY OF DEFINED.ARRAY.SET. -1 FOR EMPTY $.
DEFINEDCASELABELSETPT, $POINTS TO TOP OF DEFINED.CASELABEL.SET STACK. -1 FOR
EMPTY $.
DEFINEDIDENTIFIERSETPT, $POINTS TO LAST ENTRY OF DEFINED.IDENTITY.SET. -1 FOR
EMPTY $.
DEFINEDLABELSETPT, $POINTS TO LAST ENTRY OF DEFINED.LABEL.SET. -1 FOR EMPTY $.
DEFINEDPROCEDURESETPT, $POINTS TO LAST ENTRY OF DEFINED.PROCEDURE.SET. -1 FOR
EMPTY $.
DEFINEDSIMPLESETPT, $POINTS TO LAST ENTRY OF DEFINED.SIMPLE.SET. -1 FOR EMPTY $.
EXPPACK, $POINTS TO END OF AS EXPRESSION DESCRIPTION IN ARRAY EXPLIST. USED BY
PROCEDURE EXPCHECKER $.
EXPLIST, $POINTS TO FRONT OF AN EXPRESSION DESCRIPTION IN ARRAY EXPLIST. USED BY
PROCEDURE EXPCHECKER $.
EXPLISTTP, $POINTS TO LAST ENTRY IN EXPLIST. -1 FOR EMPTY $.
EXPTYPEP, $POINTS TO TOP OF EXP.TYPE STACK. -1 FOR EMPTY $.
F, $TEMPORARY $.
FINDI, $FINDI$ POINT TO FRONT AND BACK OF AN IDENTIFIER STRING IN ARRAY
CHARLIST. USED AS INPUT ARGUMENTS BY PROCEDURE FINDI $.
FINDLABEL1, $FINDLABEL2$ POINT TO FRONT AND BACK OF LABEL STRING IN ARRAY CHARLIST
USED AS INPUT ARGUMENTS BY PROCEDURE FINDLABEL $.
FRONTLABEL, $POINTS TO FIRST LABEL OF A LIST IN ARRAY CASELABELS. USED AS INPUT
ARGUMENT BY PROCEDURE PRINTCASELABELS $.
G, $H, $I, $TEMPORARY $.
IDENTIFIER$1, $IDENTIFIER$2, $IDENTIFIER$3, $IDENTIFIER$4, $INPUT ARGUMENTS FOR PROCEDURE IDENTIFIER
POINT TO FRONT AND BACK OF STRING 1 AND FRONT AND BACK OF STRING 2 IN
ARRAY CHARLIST $.
IFELSEPOINTTOP, $POINTS TO TOP OF IF.ELSE.POINT STACK. -1 FOR EMPTY $.
IFLEXPRESSIONTOP, $POINTS TO TOP OF IF.EXPRESSION STACK. -1 FOR EMPTY $.
IFPOINTTOP$POINTS TO TOP OF IF-POINT STACK. -1 FOR EMPTY$
INITIALPROCEDURE$POINTS TO THE IDENTIFIER IN DEFINED. IDENTIFIER$SET WHICH
NAMES THE STARTING PROCEDURE$
INTVAL$CONTAINS AN INTEGER VALUE$
JL$STEMPORARIES
LABELTABLEFT$POINTS TO LAST ENTRY IN ARRAY LABELTABLE. -1 FOR EMPTY$
LEFTARRAYNAME$SNETWORK REGISTER--POINTS TO THE IDENTIFIER IN DEFINED. IDENTIFIER$
SET WHICH NAMES THE ARRAY REFERENCED ON THE LEFT OF AN ASSIGNMENT$
LEFTBRACK$USED AS INPUT ARGUMENT BY PROCEDURE BALANCE. CONTAINS CODE OF LEFT
BRACKET$
LEFTTYPE$SNETWORK REGISTER-- =1 FOR INTEGER, =2 FOR BOOLEAN, =3 FOR CHARACTER$
LINEBACK$LINEFRONT$INPUT ARGUMENTS TO PROCEDURE BUILDLINE. POINT TO FRONT AND
BACK OF A CHARACTER STRING IN ARRAY CHARRLIST$
LINEPT$POINTS TO LAST ENTRY IN ARRAY LINE$
MULTIPLYTEPTOP$POINTS TO TOP OF MULTIPLY.TYPE STACK. -1 FOR EMPTY$
N$STEMPORARIES
NEWCASELABEL$SNETWORK REGISTER--INTEGER VALUE OF MOST RECENT CASE LABEL$
NEXTCHARACTER$SNETWORK REGISTER--SAVES THE CURRENT INPUT SYMBOL$
NOTYPEPTOP$POINTS TO TOP OF NOT.TYPE STACK. -1 FOR EMPTY$
NSTEPS$COUNTS NUMBER OF PATH STEPS$
P$STEMPORARIES
PARSESTACKTOP$POINTS TO TOP OF PARSESTACK. -1 FOR EMPTY$
PARSESTART$STATE NUMBER FOR INITIATION OF PARSE NETWORK$
PATHFRONT$SPOINT$POINTS TO LAST ENTRY IN ARRAY PATHFRONT. -1 FOR EMPTY$
PATHPT$SPOINTS TO LAST ENTRY IN ARRAY PATH$. -1 FOR EMPTY$
POINT$SNETWORK REGISTER--CURRENT CONTROL POINT$
PRIMARYTYPEPTOP$POINTS TO TOP OF PRIMARY.TYPE STACK. -1 FOR EMPTY$
PROCALLED$POINTS TO DESCRIPTION OF CALLED PROCEDURE IN ARRAY PROC--USED IN
PROCEDURE GENERATION$
PROCEDURENAME$SNETWORK REGISTER--POINTS TO IDENTIFIER IN DEFINED. IDENTIFIER$SET
WHICH NAMES CURRENT PROCEDURE$
PROCNUM$POINTS TO A PROCEDURE DESCRIPTION IN PROC[1$PROCNUM$+1] THROUGH
PROC[1$PROCNUM$+6]$
Q$STEMPORARIES
REFERENCELABELSEPTOP$POINTS TO LAST ENTRY IN REFERENCED.LABEL.SET. -1 FOR
EMPTY$
REFERENCEPROCEDURESEPTOP$POINTS TO LAST ENTRY IN REFERENCED.PROCEDURE.SET
-1 FOR EMPTY$
RELATIONTYPEPTOP$POINTS TO TOP OF RELATION.TYPE STACK. -1 FOR EMPTY$
RIGHTBACK$USED AS INPUT ARGUMENT BY PROCEDURE BALANCE. CONTAINS CODE FOR
RIGHT BRACKET$
RTNSTACKTOP$POINTS TO TOP OF RTNSTACK. -1 FOR EMPTY$
S$STEMPORARIES
SCANSTART$STATE NUMBER FOR INITIATION OF SCAN NETWORK$
STATE$CURRENT STATE DURING TRANSITION NETWORK TRAVERSAL$
STATEMENTPT$POINTS TO LAST ENTRY IN ARRAY STATEMENT. -1 FOR EMPTY$
SUBBACK$SUBFRONT$INPUT ARGUMENTS FOR PROCEDURE PRINTSUBEXP. FRONT AND BACK OF
EXPRESSION IN ARRAY EXPLIST$
T$STEMPORARIES
TOKEN$SNETWORK REGISTER--INTEGER CODE OF TOKEN$
TYPE$SNETWORK REGISTER-- =1 FOR INTEGER, =2 FOR BOOLEAN, =3 FOR CHARACTER$
UNARYTYPEPTOP$POINTS TO TOP OF UNARY.TYPE STACK. -1 FOR EMPTY$
VARIABLE$STEMPORARIES
WHILEAPPRESSIONTOP$POINTS TO TOP OF WHILE.EXPRESSION STACK. -1 FOR EMPTY$
WHILEPOINTTOP$POINTS TO TOP OF WHILE.POINT STACK. -1 FOR EMPTY$

BOOLEAN
ALPHABETMATCHFLAG$RETURNS RESULT OF PROCEDURE ALPHABETMATCH IN TRANSITION
NETWORK TRAVERSAL$
ANOTHERBRANCH$RETURNS RESULT OF PROCEDURE ISANOTHERBRANCH--TRUE IF MORE
BRANCHES FROM PATH[PATHPT]
ASRTFLAG$ = 0 IF NOT IN AN ENTER STATEMENT + 1 IF IN INITIAL ASSERTION OF CALLED
PROCEDURE OF ENTER STATEMENT + 2 IF IN FINAL ASSERTION OF CALLED
PROCEDURE OF ENTER STATEMENT

ASRTSCANFLAG$ TRUE IF SCANNING AN ASSERTION, USED BY SCANNER TO KNOW WHERE TO
GET INPUT SYMBOYS

DONE$ = SUSED BY PROCEDURE LSTCALLEDPROCS TO INDICATE WHEN ALL PROCEDURES
REACHABLE FROM CALLED PROCEDURE HAVE BEEN LISTED

FOUND$ = SRETURNS RESULT FOR SEVERAL SEARCH PROCEDURES

IDENTSTRFLAG$ = SRETURNS RESULT OF PROCEDURE IDENTSTR, TRUE IF SPECIFIED STRINGS
ARE IDENTICAL

INRANGE$ = SSET BY PROCEDURE EVALINTOK, TRUE IF NUMBER IS IN RANGE (0$NUMBER$99999)

NILMATCH$ = SRETURNS RESULT OF PROCEDURE NILMATCH IN TRANSITION NETWORK
TRAVERSAL

RECOGNITION$ = SBECOMES TRUE WHEN TRANSITION NETWORK RECOGNIZES A STRING

STACKTESTFLAG$ = SRETURNS RESULT OF PROCEDURE STACKTEST IN TRANSITION NETWORK
TRAVERSAL

TESTFLAG$ = SRETURNS RESULT OF NETWORK TESTS

CHARACTER
CHAR$ = CONTAINS A NUCLEUS CHARACTER

INTEGER ARRAY
ACTION(161$) SACTIONS OF TRANSITION NETWORK, ACTION(x) CONTAINS ACTION NUMBER FOR
ARC x

ALTNUM(249$) SALTERATION COUNTERS, ALTNUM(x) IS ALTERATION COUNTER FOR IDENTIFIER
NUMBER x IN DEFINED. IDENTIFIER$ SET

ALTSET(1000$) $CONTAINS ALTERATION SETS FOR PROCEDURES, STORES POINTERS TO
IDENTIFIERS IN DEFINED. IDENTIFIER$ SET

ANDTYPEP(1$) $NETWORK REGISTER--AND.TYPE STACK = 1 FOR INTEGER, = 2 FOR BOOLEAN, = 3
FOR CHARACTER

ARRAYNAME(1$) $NETWORK REGISTER--POINTS TO FRONT AND BACK OF ARRAY$NAME IN ARRAY
CHARLIST

ASRTLOC(11499) $ASASSERTION POINTERS, -1 INDICATES NO ASSERTION, OTHERWISE POINTS
TO ARRAY ASRTLOC$.

ASRTLOC(499$) $ASASSERTION POINTERS, POINTS TO FRONT OF FIRST ASSERTION AT POINT.
POINTS TO ARRAY ASRTS$.

ALWAYS 1 EXTRA ENTRY TO PROVIDE END OF LAST LIST

ASRTS(3999$) $ASRTS$ ASRTLOC$1$ $IS A LIST OF
FRONT-BACK PAIRS WHICH POINT TO ASSERTION BODY STRINGS IN ARRAY
CHARLIST

BINARYADDTYPE$1$ $NETWORK REGISTER--BINARY. ADD. TYPE STACK = 1 FOR INTEGER,
= 2 FOR BOOLEAN, = 3 FOR CHARACTER

BOUNDFUNCTION(249$) $NETWORK REGISTER--BOUND OF IDENTIFIER X OF DEFINED. IDENTIFIER
SET FOR ARRAY VARIABLES

BRANCH$ = 49$ $BRANCH$ X = BRANCH LAST TAKEN FROM POINT PATH$X$.

= 0 IF NOT A BRANCH

STATEMENT AT PATH$X$

CASEEXPRESSION$ = NETWORK REGISTER--CASEEXPRESSION(2$X$) AND
CASEEXPRESSION$ = (2$X$) POINT TO FRONT AND BACK OF CASE EXPRESSION IN
ARRAY EXPLICIT

CASELABEL$ = POINTERS TO FRONTS OF LABEL LISTS IN ARRAY CASELABELS.

ALWAYS 1 EXTRA ENTRY TO PROVIDE END OF LAST LIST

CASELABEL$ = CONTAINS CONTROL POINT OF CASE LABEL IN
DEFINECASELABELSET$X$

CASELABEL$ = ELEMENTS ARE CASE LABELS--CASELABELS$CASELABELFRONT$X$--
CASELABELFRONT$X$ = A CASE LABEL LIST FOR A POINT

CASELABELSET$ = CASE LABEL SET$X$ IS CONTROL POINT FOR LIST POINTED TO BY
CASELABELFRONT$X$

CASELABELSTACK$ = CASELABELSTACK$X$ POINTS TO LAST ENTRY OF LEVEL X OF
DEFINECASELABELSET$X$

STACK$ = POINTERS TO FRONTS OF LABEL LISTS IN ARRAY CASELABELS.
CASEPOINT(4) = NETWORK REGISTER -- CASE POINT STACK

COLUMNS(101) = ALTERNATION COUNTERS FOR COLUMNS -- IDENTIFIERS (LOC, LVL, MD, ETC)

COLUMNS(11) = SCOLUMNS(1) POIN TS TO FRONT OF IDENTIFIER STRING IN ARRAY

DEFINEARRAYSET(4) = NETWORK REGISTER -- IDENTIFIER NUMBERS FOR ELEMENTS OF DEFINED. IDENTIFIER_SET WHICH ARE IN DEFINE. ARRAY_SET

DEFINECASELABELSET(4) = NETWORK REGISTER -- DEFINE CASELABELSET(1) IS LABEL AT CONTROL POINT. CASELABEL POINTS(1) IS

DEFINEIDENTIFIERSET(4) = NETWORK REGISTER -- DEFINE IDENTIFIER SET(2X1) AND DEFINE IDENTIFIER SET(2X1+1) POINT TO FRONT AND BACK OF IDENTIFIER

DEFINELABELSET(4) = NETWORK REGISTER -- DEFINE LABELSET(2X1) AND DEFINE LABELSET(2X1+1) POINT TO FRONT AND BACK OF LABEL NUMBER X IN ARRAY CHA P LIST

DEFINEPROCEDURESET(19) = NETWORK REGISTER -- CONTAINS IDENTIFIER NUMBERS OF NAMES OF DEFINED PROCEDURES IN DEFINE. IDENTIFIER_SET

DEFINEDSIMPLESET(2X9) = NETWORK REGISTER -- CONTAINS IDENTIFIER NUMBERS OF NAMES OF DEFINED SIMPLE VARIABLES

DESCLOC(4999) = IF PRINT = 1 THEN DESCLOC(X) POINTS TO DESCRIPTION OF POINT IN ARRAY STATEMENT

EXPSTI(4999) = EXPSTI(2X1) AND EXPSTI(2X1+1) CONTAIN A TOKEN-VALUE PAIR. E.G., IDENTIFIER, ID NUMBER OR (NUMBER, VALUE)

EXPSRTI(11) = REPLACES ALL EXPRESSION STACKS, POINTS TO FRONT AND BACK OF EXPRESSION IN EXPSTI

EXPTYPE(19) = NETWORK REGISTER -- EXP. TYPE STACK = 1 FOR INTEGER, = 2 FOR BOOLEAN, = 3 FOR CHARACTER

FIRSTARC(1100) = ARCS FOR TRANSITION NETWORK STATES. FIRSTARC(1) THROUGH FIRSTARC(1100) ARE ARCS FROM STATE X

IDENTIFIERNAME(1) = NETWORK REGISTER -- FRONT AND BACK OF IDENTIFIER STRING IN ARRAY CHA P LIST

IFELSEPOINT(19) = NETWORK REGISTER -- IF. ELSE. POINT STACK

IFEXPRESSION(19) = NETWORK REGISTER -- IFEXPRESSION(2X1) AND IFEXPRESSION(2X1+1) ARE FRONT AND BACK OF EXPRESSION IN IFXPR. REPRESENT IF EXPRESSION AT LEVEL X

IFPOINT(19) = NETWORK REGISTER -- IF. POINT STACK

LABELTABLE(13*1) AND LABELTABLE(3X1) ARE FRONT AND BACK OF LABEL STRING IN CHA P LIST. LABELTABLE(13X1+1) IS CONTROL POINT FOR LABEL OR LINK TO LABEL REFERENCES IF NOT YET DEFINED. -1 IS LAST-LINK

LEFTSIDE(3) = NETWORK REGISTER -- LEFTSIDE(0) = 1 FOR SIMPLE, = 2 FOR ARRAY REFERENCE. LEFTSIDE(1) = IDENTIFIER NUMBER, LEFTSIDE(2) AND LEFTSIDE(3) POINT TO FRONT AND BACK OF SUBSCRIPT EXPRESSION IN EXPSTI FOR ARRAY REFERENCES

MULTIPLYTYPE(19) = NETWORK REGISTER -- MULTIPLY-TYPE STACK = 1 FOR INTEGER, = 2 FOR BOOLEAN, = 3 FOR CHARACTER

NEXTSTATE(11) = NEXT STATES FOR TRANSITION NETWORK ARCS. NEXTSTATE(1) IS NEXT STATE FOR ARC X

NOTTYPE(19) = NETWORK REGISTER -- NOT. TYPE STACK = 1 FOR INTEGER, = 2 FOR BOOLEAN, = 3 FOR CHARACTER

PARSESTACK(9) = TEMPORARY STORAGE FOR THE PARSE RETURN POINT STACK WHILE SCAN NETWORK IS INVERTED

PATH(4) = PATH(X) IS THE X-TH CONTROL POINT ALONG THE CURRENT PATH

PATH-functions(1) = CONTAINS CONTROL POINTS FROM WHICH PATHS MAY BEGIN

PRIMARYTYPE(19) = NETWORK REGISTER -- PRIMARY-TYPE STACK = 1 FOR INTEGER, = 2 FOR BOOLEAN, = 3 FOR CHARACTER

PROC(1399) = PROC(1)....PROC(7X6) CONTAIN POINTERS TO THE DESCRIPTION OF

PROCEDURE NUMBER X

PROC(7X1) = IDENTIFIER NUMBER OF PROCEDURE NAME

PROC(7X2) = POINTER TO ARRAY DESCLOC FOR CONTROL POINT

PROC(7X3) = POINTER TO ARRAY DESCLOC FOR EXIT CONTROL

PROC(7X3) = POINTER TO FRONT OF CALLED PROCEDURES IN ARRAY PROCALLS

PROC(7X5) = POINTER TO FRONT OF ALTERATION LIST IN ARRAY ALISET

PROC(7X6) = POINTER TO BACK OF ALTERATION LIST IN ARRAY ALISET
PROCCALLS[799] contains CALLED PROCEDURES, PROCCALLS[2*X] and PROCCALLS[2*X+1] point to front and back of a procedure in array CHARLIST.

REFERENCESLABELSET[2*X], NETWORK REGISTER--REFERENCESLABELSET[2*X] and REFERENCESLABELSET[2*X+1] point to front and back of a label in array CHARLIST.

REFERENCESPROCEDURESET[999], NETWORK REGISTER--REFERENCESPROCEDURESET[2*X] and REFERENCESPROCEDURESET[2*X+1] point to front and back of a procedure name in array CHARLIST.

RELATIONTYPE[999], NETWORK REGISTER--RELATIONTYPE[1] stack #1 for integer,

#2 for BOOLEAN, #3 for CHARACTER.

RESCODE[2*X], RESCODE[2*X+1] is INTEGER CODE FOR RESERVED WORD NUMBER X.

RESWORDPTS[271], RESWORDPTS[2*X] points to beginning of reserved word number X in array RESERVEDWROSET.

RTNSTACK[999], RTNSTACK POINT STACK FOR TRANSITION NETWORK.

STATEMENT[999], STORES DESCRIPTIONS OF STATEMENTS. THE DESCRIPTION LENGTH VARIES ACCORDING TO THE STATEMENT TYPE.

ASSIGNMENT[7 ELEMENTS]

1 64 (CODE FOR =)
2 5 SEE LIFETIME
6 7 POINT TO FRONT AND BACK OF RIGHT SIDE IN ARRAY EXPLIST

CASE[7 ELEMENTS]

1 72 (CODE FOR CASE)
2 3 POINT TO FRONT AND BACK OF CASE EXPRESSION IN ARRAY EXPLIST
4 CASEJOINPOINT (JOIN POINT LINK UNTIL JOIN POINT DEFINED)
5 6 FRONT AND BACK OF CASE LABELS IN ARRAY CASELABELSET
7 CONTROL POINT FOR CASE EXPRESSION # ANY CASE LABEL

ENTER (3 ELEMENTS)

1 78 (CODE FOR ENTER)
2 3 POINT TO FRONT AND BACK OF STRING IN CHARLIST WHICH NAMES CALLED PROCEDURE

EXIT[1 ELEMENT]

1 80 (CODE FOR EXIT)

HALT[1 ELEMENT]

1 84 (CODE FOR HALT)

IF[5 ELEMENTS]

1 85 (CODE FOR IF)
2 3 FRONT AND BACK OF IF EXPRESSION IN EXPLIST
4 NEXT CONTROL POINT FOR EXPRESSION = TRUE
5 NEXT CONTROL POINT FOR EXPRESSION = FALSE

JUMP[2 ELEMENTS]

1 95 (CODE FOR TO)
2 NEXT CONTROL POINT

READ[2 ELEMENTS]

1 91 (CODE FOR READ)
2 IDENTIFIER NUMBER OF READ ARRAY

WRITE[12 ELEMENTS]

1 96 (CODE FOR WRITE)
2 IDENTIFIER NUMBER OF WRITE ARRAY

SYMBOL[161], ARC SYMBOLS FOR TRANSITION NETWORK. SYMBOL[X] IS CHARACTER ON ARC NUMBER X.

TEST[161], TESTS FOR TRANSITION NETWORK. TEST[X] CONTAINS TEST NUMBER FOR ARC NUMBER X.

TOKENSTRING[111], NETWORK REGISTER--POINTS TO FRONT AND BACK OF TOKEN STRING IN ARRAY CHARLIST.

TYPEFUNCTION[2*X], NETWORK REGISTER--TYPEFUNCTION[X] IS TYPE OF IDENTIFIER NUMBER X. = 1 FOR INTEGER, = 2 FOR BOOLEAN, = 3 FOR CHARACTER.

UNARYADDTOYPE[111], NETWORK REGISTER--UNARYADDTOYPE[X] IS TYPE OF EXPRESSION AT LEVEL X.

BOOLEAN ARRAY
ALTFLAG[291]*ALTFLAG[1] TRUE IF IDENTIFIER NUMBER X CAN BE ALTERED BY A CALL OF PROCEDURE PROCALLED $
COLONALTFLAG[10]*COLONALTFLAG[1] TRUE IF COLON-IDENTIFIER NUMBER X CAN BE ALTERED BY A CALL OF PROCEDURE PROCALLED $
FLAG[161]*SCAN FLAGS FOR TRANSITION NETWORK. FLAG[1] TRUE IF ARC X HAS A SCAN $
PROCFLAG[19]*PROCFLAG[1] TRUE IF PROCEDURE NUMBER X CAN BE REACHED BY A CALL OF PROCEDURE PROCALLED $
PROCFLAG[111]*PROCFLAG[1] TRUE IF PROCEDURE NUMBER X HAS ALREADY BEEN PROCESSED BY PROCEDURE LISTCALLEDPROCS $
RECOGNITIONSTATE[1107]*RECOGNITION STATES FOR TRANSITION NETWORK. RECOGNITIONSTATE[1] TRUE IF STATE X IS A RECOGNITION STATE $

CHARACTER ARRAY
ASRTTRUE[41]*CONSTANT ARRAY CONTAINING STRING TRUE $
BLANKLINE[10]*CONSTANT ARRAY CONTAINING F IN ELEMENT 0 TO PRINT A BLANK LINE $
CAPT[40]*CONTAINS CARD IMAGE DURING SCAN AND PARSE $
CHARLIST[699]**CHARACTER STRING STORAGE--IDENTIFIERS, LABELS, ASSERTIONS, ETC $
COLONWORDSET[3]*COLONWORDSET(COLONPTS[1]).....COLONWORDSET(COLONPTS[1]) IS CHARACTER STRING FOR COLON-IDENTIFIER NUMBER X $
DASHES[10]*CONSTANT STRING OF 10 DASHES $
DOTS[10]*CONSTANT STRING OF 10 DOTS $
LINE[120]*ARRAY IN WHICH PRINT LINES ARE ASSEMBLED $
LINE1[110]*LINE2[110]*LINE3[110]*LINE4[110]* 4 CONSTANT ARRAYS CONTAINING FIRST 4 LINES OF VERIFICATION CONDITIONS FROM POINT 0 OF INITIAL PROCEDURE $
LOOP[4]*CONSTANT ARRAY CONTAINING LOOP TO PRINT WHEN UNTAGGED LOOP IS FOUND $ 
LVL1[13]*LVL2[14]*LVL3[15]*LVL4[16]*LVL5[17]*LVL6[18]*LVL7[19]*LVL8[20]*LVL9[21]*LVL10[22] CONSTANT ARRAY CONTAINING NONRECOGNITION $ 
PATHS[17]*CONSTANT ARRAY CONTAINING PATH IS $ 
PRESET[2]*3 CONSTANT ARRAY CONTAINING SEVERAL STRINGS USED IN BUILDING VERIFICATION CONDITIONS $ 
RESERVEDWORDSET[11]*RESERVEDWORDSET(RESCONPTS[1])..... RESERVEDWORDSET(RESCONPTS[1]) IS CHARACTER STRING FOR RESERVED WORD NUMBER X $

PROCEDURE ACTIONS 
ASRST (AASRTSCANFLAG A ATSCANSTATE A ATSAVEDPARSESTATE) V (AASRTSCANFLAG A ATPA RSESTATE) V (AASRTSCANFLAG A ATASRTSCANSTATE) 
CASE ACTION(ARC) OF 
0: NOP 1 
1: ENTER ACTION1 1 
2: ENTER ACTION2 1 
3: ENTER ACTION3 1 
4: ENTER ACTION4 1 
5: ENTER ACTION5 1 
6: ENTER ACTION6 1 
7: ENTER ACTION7 1 
8: ENTER ACTION8 1 
9: ENTER ACTION9 1 
10: ENTER ACTION10 1 
11: ENTER ACTION11 1 
12: ENTER ACTION12 1 
13: ENTER ACTION13 1 
14: ENTER ACTION14 1 
15: ENTER ACTION15 1 
16: ENTER ACTION16 1 
17: ENTER ACTION17 1
18: ENTER ACTION18
19: ENTER ACTION19
20: ENTER ACTION20
21: ENTER ACTION21
22: ENTER ACTION22
23: ENTER ACTION23
24: ENTER ACTION24
25: ENTER ACTION25
26: ENTER ACTION26
27: ENTER ACTION27
28: ENTER ACTION28
29: ENTER ACTION29
30: ENTER ACTION30
31: ENTER ACTION31
32: ENTER ACTION32
33: ENTER ACTION33
34: ENTER ACTION34
35: ENTER ACTION35
36: ENTER ACTION36
37: ENTER ACTION37
38: ENTER ACTION38
39: ENTER ACTION39
40: ENTER ACTION40
41: ENTER ACTION41
42: ENTER ACTION42
43: ENTER ACTION43
44: ENTER ACTION44
45: ENTER ACTION45
46: ENTER ACTION46
47: ENTER ACTION47
48: ENTER ACTION48
49: ENTER ACTION49
50: ENTER ACTION50
51: ENTER ACTION51
52: ENTER ACTION52
53: ENTER ACTION53
54: ENTER ACTION54
55: ENTER ACTION55
56: ENTER ACTION56
57: ENTER ACTION57
58: ENTER ACTION58
59: ENTER ACTION59
60: ENTER ACTION60
61: ENTER ACTION61
62: ENTER ACTION62
63: ENTER ACTION63
64: ENTER ACTION64
65: ENTER ACTION65
66: ENTER ACTION66
67: ENTER ACTION67
68: ENTER ACTION68
69: ENTER ACTION69
70: ENTER ACTION70
71: ENTER ACTION71
72: ENTER ACTION72
73: ENTER ACTION73
74: ENTER ACTION74
75: ENTER ACTION75
76: ENTER ACTION76
77: ENTER ACTION77
78: ENTER ACTION78
79: ENTER ACTION79
80: ENTER ACTION80
81: ENTER ACTION81
82: ENTER ACTION82
83: ENTER ACTION83
84: ENTER ACTION84
85: ENTER ACTION85
86: ENTER ACTION86
87: ENTER ACTION87
88: ENTER ACTION88
89: ENTER ACTION89
90: ENTER ACTION90
91: ENTER ACTION91
92: ENTER ACTION92
93: ENTER ACTION93
94: ENTER ACTION94
95: ENTER ACTION95
96: ENTER ACTION96
97: ENTER ACTION97
98: ENTER ACTION98

ESAC
EXIT

PROCEDURE ALPHABETMATCH
ARC := FIRSTANSI(STATE)
WHILE ARC < FIRSTANSI(STATE) + 1 AND SYMBOL(ARC) > 0 DO
    IF SYMBOL(ARC) = CHARSTRING
        THEN ENTER TESTS
        IF TESTFLAG
            THEN ALPHABETMATCHFLAG := TRUE
        RETURN
    FI
    ARC := ARC + 1
    ELIM
    ALPHABETMATCHFLAG := FALSE
EXIT

PROCEDURE ASRTSCAN
ASRTSCANTAG := TRUE
STATE := SCANSTART
RTNSTACKTOP := -1
CHARSTRING := INTEGER(CHARLIST(ASRTSCANPER))
RECOGNITION := FALSE
ASSERT ASRTSCANTAG
ASSERT ARRAY(STATE)
WHILE NOT RECOGNITION DO
    ENTER TRANSNET
    ELIM
    ASSERT CHARSTRING = REMOVETOKEN(STRING, 0)
    ASSERT TOKEN = NEXTOKEN(STRING, 0)
    ASSERT TOKENSTRING = NEXTOKENSTRING(STRING, 0)
EXIT

PROCEDURE BALANCE
ASSERT SUBFRONT POINTS TO THE FIRST ELEMENT TO THE RIGHT OF A LEFTBRACK OF AN EXPRESSION IN EXPLIST
N := 1
ASSERT N > 1
ASSERT N IS THE NUMBER OF EXCESS LEFTBRACKS OVER RIGHTBRACKS ENCOUNTERED IN EXP
IST BETWEEN SUBFRONT.0 AND SUBBACK
ASSERT N HAS NEVER BEEN LESS THAN 1 BETWEEN SUBFRONT.0 AND SUBBACK
ASSERT SUBFRONT = SUBFRONT.0
WHILE N>1 • EXPList2*(SUBBACK+1))•RIGHTBACK DO
    SUBBACK:=SUBBACK+1
    IF EXPList2•SUBBACK=LEFTBACK
      THEN N:=N+1
    FI
  IF EXPList2•SUBBACK=RIGHTBACK
    THEN N:=N+1
  FI
ELIH

ASSERT SUBFRONT.0 AND SUBBACK POINT TO FRONT AND BACK OF AN EXPRESSION IN EXPList
T WHICH IS BALANCED WITH RESPECT TO LEFTBACK.0 AND RIGHTBACK.01

EXIT 1

PROCEDURE BUILDFROMPRESET1:
  ASSERT LINE(0)•LINE(01)
  ASSERT LINEFRONT.0•LINEFRONT.1•LINEBACK.0•LINEBACK.1
  ASSERT WRITEFILE(:WHO)•U•LINE(1)•EXPListLINEPT=WRITEFILE(:WHO)•U•LINE(0)
  ***LINE(0)•OILINEPT.0)•U•PRESET•OILINEFRONT.0)•***PRESET•OILINEBACK.0
  WHILE LINEFRONTSLINEBACK DO
    IF LINEPT=120
      THEN WRITE LINE
          LINEPT:=0
    FI
    LINEPT:=LINEPT+1
    LINEFRONT:=LINEFRONT+1
ELIH
  ASSERT LINE(0)•LINE(01)
  ASSERT WRITEFILE(:WHO)•U•LINE(1)•EXPListLINEPT=WRITEFILE(:WHO)•U•LINE(0)
  ***LINE(0)•OILINEPT.0)•U•PRESET•OILINEFRONT.0)•***PRESET•OILINEBACK.0
  EXIT 1

PROCEDURE BUILDLINE1:
  ASSERT OILINEFRONTSLINEBACK=1•CHARLISTPT+1
  ASSERT WRITEFILE(:WHO)•U•LINE(0)
  ***EXPListLINEPT=WRITEFILE(:WHO)•U•LINE(0)
  ***LINE(0)•OILINEPT.0)•U•PRESET•OILINEFRONT.0)•***PRESET•OILINEBACK.0
  WHILE LINEFRONTSLINEBACK DO
    IF LINEPT=120
      THEN WRITE LINE
          LINEPT:=01
    FI
    LINEPT:=LINEPT+1
    LINEFRONT:=LINEFRONT+1
ELIH
  ASSERT WRITEFILE(:WHO)•U•LINE(0)
  ***EXPListLINEPT=WRITEFILE(:WHO)•U•LINE(0)
  ***LINE(0)•OILINEPT.0)•U•PRESET•OILINEFRONT.0)•***PRESET•OILINEBACK.0
  ASSERT LINE(0)=LINE(01)
  EXIT 1

PROCEDURE EVALINTOK1:
  ASSERT OITOKSTRING(0)=TOKENSTRING(1)•CHARLISTPT1
  ASSERT TOKENSTRING(0)•OITOKSTRING(11)•CHARLISTPT1
  IF TOKENSTRING(0)
    INTVAL:=01
    ASSERT INTVAL=INTEGERVALUE(CHARLIST(0)•OITOKSTRING.01)
    ***CHARLIST.01):.011):.999
    ASSERT TOKENSTRING.01=OITOKSTRING.01
    ASSERT CHARLIST=CHARLIST.01
    ASSERT TOKENSTRING=TOKENSTRING.01
  WHILE 1STOKENSTRING(11) DO
V1:=INTEGER(CHARLIST[1]) - 271
IF INTVAL=(999999-V1)/10
THEN INRANGE := FALSE 1
INTVAL:=999999
RETURN 1
FI :  
INTVAL:=10*INTVAL+V1
I:=I+1
ELIHV:
INRANGE:=TRUE 1
ASSERT INRANGE = INTVAL=INTEGERVALUE(CHARLIST,0(TOKENSTRING,0(11))1...CHARLIST,0(TOKENSTRING,0(11)))<999999
ASSERT INRANGE = INTEGERVALUE(CHARLIST,0(TOKENSTRING,0(11))1...CHARLIST,0(TOKENSTRING,0(11)))>=999999 ^ INTVAL=999999
EXIT 1:

PROCEDURE EXPCHECKER:
ASSERT EXPBACK=EXPBACK,01
ASSERT EXPFRONT=SEXPFRONT=EXPBACK1
ASSERT EXPFRONT CONTAINS AN EXPRESSION BETWEEN EXPFRONT,0 AND EXPBACK,01
ASSERT WHITEFILE(1,WTMD)=WHITEFILE(1,WTMD,0) U VERIFICATION CONDITION LINES FOR EACH ARRAY SUBSCRIPT WHOSE I PRECEDES EXPFRONT AND FOR EACH DIVISOR WHOSE / OR * OR + RECEDES EXPFRONT IN THE EXPRESSION IN EXPFRONT,0 BETWEEN EXPFRONT,0 AND EXPBACK,01

ASSERT LINETH=0 ^ LINE(0)=*F1
WHILE EXPFRONT EXPBACK = DO
CASE EXPLIST(2*EXPFRONT,0) OF
$S
   41: ENTER INSERTPRV1
      CHAR:='*'
      ENTER INSERTCHAR1
      CHAR:='*'
      ENTER INSERTCHAR1
      SUBFRONT=EXPFRONT+1
      SUBBACK=SUBFRONT1
      LEFTBACK=41 $S
      RIGHTBACK=42 $S
      ENTER BALANCE1
      ENTER PRINTSUBEXP1
      CHAR:=*'
      ENTER INSERTCHAR1
      CHAR:='/'
      ENTER INSERTCHAR1
      INRANGE=BOUNOFUNCTION(EXPLIST(2*(EXPFRONT-1)+1))1
      ENTER INPRINT1
      ENTER PRINTLINE1
      $S
      48159: ENTER INSERTPRV1
      SUBFRONT=EXPFRONT+21
      SUBBACK=SUBFRONT1
      LEFTBACK=41 $S
      RIGHTBACK=42 $S
      ENTER BALANCE1
      ENTER PRINTSUBEXP1
      CHAR:='*
      ENTER INSERTCHAR1
      CHAR:='*
      ENTER INSERTCHAR1
      ENTER PRINTLINE1
      $S
      48159: ENTER INSERTPRV1
      SUBFRONT=EXPFRONT+21
      SUBBACK=SUBFRONT1
      LEFTBACK=41 $S
      RIGHTBACK=42 $S
      ENTER BALANCE1
      ENTER PRINTSUBEXP1
      CHAR:='*
      ENTER INSERTCHAR1
      CHAR:='*
      ENTER INSERTCHAR1
      ENTER PRINTLINE1
      $S
      EXIT
      EXPFRONT=EXPFRONT+11
      ELIHV:
      ASSERT LINETH=0 ^ LINE(0)=*F1
      ASSERT WHITEFILE(1,WTMD)=WHITEFILE(1,WTMD,0) U VERIFICATION CONDITION LINES FOR EACH ARRAY SUBSCRIPT OR DIVISOR IN THE EXPRESSION IN EXPFRONT,0 BETWEEN EXPFRONT,0 AND EXPBACK,01
EXIT 1

PROCEDURE FINDID1
ASSERT 0$5$5$DEFINEDIDENTIFIERSETP1 = 0$DEFINEDIDENTIFIERSETP1(2$5$5$5$1)$DEFINEDIDENTIFI
ERSET(2$5$5$5$1)$CHARLISTP1
ASSERT 0$FINDID2$CHARLISTP1 = 0$FINDID2$CHARLISTP1
ASSERT 0$DEFINID2$IDENTIFIERSETP1 = 11
IDENTSTR1 = FINDID1
IDENTSTR2 = FINDID2
AT = 01
ASSERT 0$5$5$5$DEFINEDIDENTIFIERSETP1 = 0$DEFINEDIDENTIFIERSETP1(2$5$5$5$1)$DEFINEDIDENTIFI
ERSET(2$5$5$5$1)$CHARLISTP1
ASSERT 0$5$5$5$AT = 1$CHARLIST,0$DEFINEDIDENTIFIERSETP1(2$5$5$1)$CHARLIST,0$DEFINED
IDENTIFIERSETP1(2$5$5$1)$CHARLIST,0$FINDID1,0)10$CHARLIST,0$FINDID2,011
ASSERT IDENTSTR1 = FINDID1
ASSERT IDENTSTR2 = FINDID2
ASSERT 0$5$5$5$DEFINEDIDENTIFIERSETP1 = 11
ASSERT 0$DEFINID2$IDENTIFIERSETP1 = 0$DEFINEDIDENTIFIERSETP1
ASSERT CHARLIST = CHARLIST,0
WHILE AT$DEFINEDIDENTIFIERSETP1 DO
IDENTSTR2 = IDENTIFIERSETP1(2$AT)1
IDENTSTR1 = IDENTIFIERSETP1(2$AT + 1)1
ENTEND IDENTSTR1
IF IDENTSTRFLAG THEN FOUND = TRUE
RETURN
FI
AT = AT + 11
ELSE IF FOUND = FALSE
ASSERT FOUND = 0$5$5$5$DEFINEDIDENTIFIERSETP1,0$CHARLIST,0$DEFINEDIDENTIFIERSETP1,0
(2$AT)10$CHARLIST,0$DEFINEDIDENTIFIERSETP1,0(2$AT + 1)10$CHARLIST,0$FINDID1,010$CHAR
LIST,0$FINDID2,011
ASSERT = FOUND = 0$5$5$5$DEFINEDIDENTIFIERSETP1,0$CHARLIST,0$DEFINEDIDENTIFIERSETP1,
0(2$5$5$1)$CHARLIST,0$DEFINEDIDENTIFIERSETP1,0(2$5$5$1)$CHARLIST,0$FINDID1,010$CHAR
LIST,0$FINDID2,011
EXIT 1

PROCEDURE FINDLABEL
ASSERT 0$FINDLABEL2$FINDLABEL2$CHARLISTP1
ASSERT LABELTABLEP1 = 11
ASSERT 0$5$5$LABELTABLEP1 = 0$LABELTABLEP1(3$5$5$1)$LABELTABLEP1(3$5$5$1)$CHARLISTP1
IDENTSTR1 = FINDLABEL1
IDENTSTR2 = 0$LABELTABLEP1
AT = 01
ASSERT 0$5$5$AT = 1$CHARLIST,0$LABELTABLEP1(3$5$5$1)$CHARLIST,0$LABELTABLEP1(3$5
5$1)$CHARLIST,0$FINDLABEL1,0)10$CHARLIST,0$FINDLABEL2,011
ASSERT 0$AT$LABELTABLEP1 = 11
ASSERT IDENTSTR1 = FINDLABEL1
ASSERT IDENTSTR2 = LABELTABLE
ASSERT LABELTABLE = LABELTABLE,01
ASSERT LABELTABLE = LABELTABLE,01
ASSERT FINDLABEL1 = LABELTABLE,01
ASSERT LABELTABLE = LABELTABLE,01
ASSERT CHARLIST = CHARLIST,01
WHILE AT$LABELTABLEP1 DO
IDENTSTR1 = LABELTABLE,01
IDENTSTR4 = LABELTABLE,01
ENTEND IDENTSTR1
IF IDENTSTRFLAG THEN FOUND = TRUE

RETURN 1

AT:=AT+1

END

EXIT

PROCEDURE FINDRESERVED
ASSERT 0 TOKENSTRING(0)TOKENSTRING(1)=CHARLIST(0)
ASSERT CONSTANTS(0)RESERVEDWORDSET+RESERVEDWORDPTS+RESCODE+LOOP+DASHES+DOTS+ASRT
TRUE+BLANKLINE+PATHISLINE1+LINE2+LINE3+LINE4+COLONWORDSET+COLONPTS+LYLLINE)

AT:=0

ASSERT 0 AT-1 RESERVEDWORDSET+RESERVEDWORDPTS+RESCODE+LOOP+DASHES+DOTS+ASRT
TRUE+BLANKLINE+PATHISLINE1+LINE2+LINE3+LINE4+COLONWORDSET+COLONPTS+LYLLINE)

ASSERT TOKENSTRING(0)

ASSERT CHARLIST=CHARLIST(0)

WHILE AT <= 26 DO

IF RESERVEDWORDPTS(AT)=RESERVEDWORDPTS(0)END

THEN J:=0

ASSERT 0 AT-1 RESERVEDWORDSET+RESERVEDWORDPTS+RESCODE+LOOP+DASHES+DOTS+ASRT
TRUE+BLANKLINE+PATHISLINE1+LINE2+LINE3+LINE4+COLONWORDSET+COLONPTS+LYLLINE)

ASSERT TOKENSTRING(0)

ASSERT CHARLIST=CHARLIST(0)

WHILE J TOKENSTRING(0) J=CHARLIST(TOKENSTRING(0)+J)

RESERVEDWORDSET/RESERVEDWORDPTS(AT)=J DO

J:=J+1

END

ELIHU

IF J TOKENSTRING(1)=TOKENSTRING(0)

THEN FOUND:=TRUE

RETURN 1

FI

AT:=AT+1

FLIHU

FOUND:=FALSE

ASSERT FOUND=RESERVEDWORDSET+RESERVEDWORDPTS+AT=1 RESERVEDWORDSET+RESERVEDWORDPTS+AT=1

CHARLIST(0)TOKENSTRING(0)CHARLIST(0)TOKENSTRING(0)

SAT:=26

ASSERT -FOUND=0 AT-1 RESERVEDWORDSET+RESERVEDWORDPTS+RESERVEDWORDSET+RESERVEDWORDPTS+AT=1

CHARLIST(0)TOKENSTRING(0)CHARLIST(0)TOKENSTRING(0)

EXIT

PROCEDURE FINISHPATH

ASSERT ISPATHFLONT(PATH,PATHT,P:BRANCH+CURRENTPROC)
ASSERT LINEPT=0  A LINE[0]='#F1
ASSERT PATH[0]...=PATH[PATHPT] CONTAINS NO UNTAGGED LOOPS
ASSERT ALTLIST(ALTNUM+COLONALTNUM+PATH+PATHPT)
ASSERT NSTEP=STEPS SINCE (STEP PRINT)
IF STATEMENTDESCLOC(PATH[PATHPT]+BIAS)]=A5  SIFS
  STATEMENTDESCLOC(PATH[PATHPT]+BIAS)]=72  SCASES
  THEN BRANCH(PATHPT)=11
ELSE BRANCH(PATHPT)=01
FI
ENTER GENNEXTPATHPT 1
ENTER LOOPCHECK 1
ENTER GENTERM 1
PATHPT := PATHPT + 1
IF STATEMENT(PATH(PATHPT)+BIAS)=A0  SEXITS
  STATEMENT(PATH(PATHPT)+BIAS)=A4  SMALS
  ASHTLOC(PATH(PATHPT)+BIAS)=0  SHAVE ASSERTIONS
  THEN ENTER WRITESTEP(LINE11
    WRITE DASHES1
    ENTER WRITEASSERTS1
    WRITE BLANKLINE1
    ENTER PRINTPATH1
  ELSE ENTER FINISHPATH1
FI
ASSERT THE CURRENTPROC PATHS IN PATH+PATHPT+AND BRANCH IS THE IMMEDIATE SUCCESS
OR OF THE CURRENTPROC PATHFRONT IN PATH+PATHPT,0+ AND BRANCH+01
ASSERT WRITEFILE(0+T=D)+WRITEFILE(0+WTHD.0) U VERIFICATION CONDITION TERMS FOR PA
THE(PATHPT,0)...PATH(PATHPT1)
ASSERT LINEPT=0  a LINE[0]='#F1
EXIT 1

PROCEDURE FROMPOINTVCCEN 1
ASSERT LINEPT=0  a LINE[0]='#F1
ENTER HESETALT 1
NSTEP=01
PATH[0] := CURRFRT1
PATHPT := 01
ENTER FINISHPATH 1
ENTER GENSUCCESSORS 1
ASSERT WRITEFILE(0+WTHD)+WRITEFILE(0+WTHD.0) U VERIFICATION CONDITIONS IN PROCEDURE
E CURRENTPROC PATH POINT CURRFRT1
ASSERT LINEPT=0  a LINE[0]='#F1
EXIT 1

PROCEDURE GENASSIGNTERM 1
ASSERT ALTLIST(ALTNUM+COLONALTNUM+PATH+PATHPT)1
ASSERT STMDESCLOC(PATH[PATHPT]+BIAS)11
ASSERT STATEMENT(STM)=#41
ASSERT LINEPT=0  a LINE[0]='#F1
ASSERT NSTEP=STEPS SINCE (STEP PRINT)
IF STATEMENT(STM)=2  SARRAY REF ON LEFT$1
THEN ENTER INSERTPRV1
  CHAR=01
  ENTER INSERTCHAR1
  CHAR=#51
  ENTER INSERTCHAR1
  SUBFRONT:=STATEMENT(STM+3)
  SUBBACK:=STATEMENT(STM+4)
  ENTER PRINTSUBEXP1
  CHAR=#51
  ENTER INSERTCHAR1
  INVAL=#BOUNDFUNCTION(STATEMENT(STM+2))
  ENTER INPUTRN1
ENTER PRINTLINE
EXPFRONT:=STATEMENT(STMT+3)
EXPBACK:=STATEMENT(STMT+4)
ENTER EXPCHECKER

FI
EXPFRONT:=STATEMENT(STMT+5)
EXPBACK:=STATEMENT(STMT+6)
ENTER EXPCHECKER
ID:=STATEMENT(STMT+2)
ENTER PRINTID
CHAR:=**
ENTER INSERTCHAR
INTVAL:=ALTNUM[ID]+1
ENTER INPUT:
IF STATEMENT(STMT+1)=2
THEN CHAR:=**(;
ENTER INSERTCHAR
SUBFRONT:=STATEMENT(STMT+3)
SUBBACK:=STATEMENT(STMT+4)
ENTER PRINTSUBEXP:
CHAR:=**;
ENTER INSERTCHAR

FI

CHAR:=**
ENTER INSERTCHAR
SUBFRONT:=STATEMENT(STMT+5)
SUBBACK:=STATEMENT(STMT+6)
ENTER PRINTSUBEXP
ENTER PRINTLINE
ENTER PRINTLINE
ALTNUM[STATEMENT(STMT+2)]:=ALTNUM[STATEMENT(STMT+2)]+1
COLONALTNUM[1]:=COLONALTNUM[1]+1
$LOC$=NSTEP+11
ASSERT WRITEFILE(\WTHD)\=WRITEFILE(\WTHD,0) U VERIFICATION CONDITION TERM FOR STATEMENT AT \PATH(\PATHPT)1
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATN,PATNPTH+11
ASSERT LINES:=0 ^ LINE(0)==+1
ASSERT NSTEP=STEPS SINCE \STEP PRINT:
EXIT 1

PROCEDURE GENCASETERM:
ASSERT LINES:=0 ^ LINE(0)==+1
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATN,PATNPTH+11
ASSERT STM(T=DECLOC(PAT(PATNPTH+1,BIAS)
ASSERT STATEMENT(STMT)=?1
ASSERT NSTEP=STEPS SINCE \STEP PRINT:
EXPFRONT:=STATEMENT(STMT+1)
EXPBACK:=STATEMENT(STMT+2)
ENTER EXPCHECKER
SUBFRONT:=STATEMENT(STMT+1)
SUBBACK:=STATEMENT(STMT+2)
ENTER PRINTSUBEXP:
IF STATEMENT(STMT+5)=STATEMENT(STMT+4)+12BRANCH(\PATHPT)
THEN CHAR:=**
ENTER INSERTCHAR
CHAR:=**
ENTER INSERTCHAR
FRONTLABEL:=CASELABELFRONT[CASELABELSET[STATEMENT(STMT+4)BRANCH(\PATHPT)=1]]
BACKLABEL:=CASELABELFRONT[CASELABELSET[STATEMENT(STMT+4)BRANCH(\PATHPT)]=1]
ENTER PRINTCASELABELS
CHAR:=**}
ASRTSCANPOINTER := ASRTFRONT
ASRTBACK := ASRTS(2*G+11)
ENTER INSERTPRI
ENTER WRITENEXTASRT
G:=G+11
ELIMI
FI
ENTER UPDATEALTNUMI
IF ASRTLLOC(PROC1[PROC1#PROCALLED+2])#0
THEN G:=ASRTLLOCI(ASRTLLOCI(PROC1#PROCALLED+2))
ASRTFLAG:=21
ASSERT LINEPT0 := LINE(0)#*F:
ASSERT ALTERATION COUNTERS IN ALTNUM AND COLONALTNUM HAVE BEEN ACCUMULATED THPOU
GM EXECUTION OF CALLED PROCEDURE
ASSERT ASRTFLAG=0 AND IN FINAL ASSERTION OF ENTER STATEMENT
ASSERT PROCEDURE NUMBER PROCALLED IS THE CALLED PROCEDURE
ASSERT EXIT POINT OF PROCALLED IS TAGGED WITH ASSERTIONS
ASSERT ASRATIONALLOC[PROC1#PROCALLED+2])#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASRTLOCI(ASRTLLOCI(PROC1#PROCALLED+2))#0 ASSERT WRITEFILE[:(WTHD)]:=WRITEFILE[:(WTHD, 0)] U (7) LINE U RTNPTLINE U STPLINE U SUBSCRIPTED EXIT POINT ASSERTIONS THROUGH ASSERT
ION G:=11
ASSERT NSTEP=01
WHILE G<ASRTLLOCI(PROC1[PROC1#PROCALLED+2])#0 DO
ASRTFRONT := ASRTS[2*G]
ASRTSCANPOINTER := ASRTFRONT
ASRTBACK := ASRTS[2*G+11]
ENTER WRITENEXTASRT
G:=G+11
ELIMI
FI
ENTER WRITESTEPLINE2
ASSERT WRITEFILE[:(WTHD)]:=WRITEFILE[:(WTHD, 0)] U VERIFICATION CONDITION TERM FOR STATEMENT AT PATH(PATHPT1)
ASSERT ALTLIST[ALTNUM#COLONALTNUM#PATH#PATHPT1]
ASSERT LINEPT0 := LINE(0)#*F:
ASSERT NSTEP=01
EXIT 1
PROCEDURE GENIFTERM1
ASSERT LINEPT0 := LINE(0)#*F:
ASSERT ALTLIST[ALTNUM#COLONALTNUM#PATH#PATHPT1]
ASSERT STM1=DESCLOC[PATH(PATP+1)+BIAS1]
ASSERT STATEMENT[STM1]=R51
ASSERT NSTEP=STEPS SINCE :STEP PRNT1
EXPFONT1=STATEMENT[STM1]
EXPFACK1=STATEMENTS[STM1]
ENTER EXPCHECKER!
IF BRANCH(PATHPT1)=2
THEN Ch0:=++1
ENTER INSERTCHAR!
Ch0:=++1
ENTER INSERTCHAR!
FI
SUBFRONT:=STATEMENT[STM1]
SUBBACK:=STATEMENT[STM1]
Enter PRINTSUMCHA!
IF BRANCH(PATHPT1)=2

THEN CHAR:=++
ENTER INSERCCHAR

FI
ENTER PRINTLINE
NSTEP:=NSTEP+1
ASSERT writefile(\wthd)\=writefile(\wthd,0) U verification condition term for statement at path\(\text{PATHPT}\)
ASSERT ALTLIST(ALTNUM\+\COLONALTNUM\+,PATH\+PATHPT\+1)
ASSERT LINEDPT\0 \+ LINE[10]\=\+\+F1
ASSERT NSTEP\=STEPS SINCE \STEP PRINT
EXIT

procedure gennextpathpt
ASSERT PATH[PATHPT] IS NOT MALT OR EXIT
CASE statementidescloc[PATH[PATHPT]BASIS]+1 OF
85: if branch[PATHPT]=1
FI
72: $cases
THEN $PATH[PATHPT]+1:=cases8=else(statementidescloc[PATH[PATHPT]BASIS]++2
$PATH[PATHPT]+1:=branch[PATHPT]+1)
FI
Else
$assert path[PATHPT] IS NEXTPOINT FOR PATH[0(0)]...PATH[0(PATHPT)0]
$assert 03$PATHPT,0 := PATH[3$PATHPT[0]]
EXIT

procedure genreadterm
ASSERT LINEPT=0 \+ LINE[10]=++F1
ASSERT ALTLIST(ALTNUM\+\COLONALTNUM\+,PATH\+PATHPT)
ASSERT statidescloc[PATH[PATHPT]BASIS]++1
ASSERT statement[stmt]++=1
$assume: BOUNDFUNCTION[1D]++
ENTER WHITEHEADLINE11
ENTER WHITEHEADLINE21
ENTER WHITEHEADLINE31
NSTEP:=NSTEP+1
ASSERT writefile(\wthd)\=writefile(\wthd,0) U verification condition term for statement at path\(\text{PATHPT}\)
ASSERT ALTLIST(ALTNUM\+\COLONALTNUM\+,PATH\+PATHPT\+1)
ASSERT LINEDPT\0 \+ LINE[10]\=\+\+F1
ASSERT NSTEP\=STEPS SINCE \STEP PRINT
EXIT

procedure gensuccessors
ASSERT LINEPT=0 \+ LINE[10]=++F1
ASSERT IMMEDIATE SUCCESSOR OF $PATH$ THROUGH PATH\+PATHPT\+BRANCH IS THE IMMEDIATE
SUCCESSOR OF THE PATHS THROUGH PATH.0*PATHPT.0*BRANCH.0
ASSERT PATH(0)*PATH(PATHPT) CONTAINS NO UNTAGGED LOOPS
WHILE PATHPT>0 DO
PATHPT := PATHPT-1
ENTER (ANOTHERBRANCH 1
IF ANOTHERBRANCH THEN ENTER REGEN 1
ENTER FINISHPATH 1
ENTER GENSUCCESSORS 1
RETURN 1
FI 1
ELIHW 1
ASSERT WRITEFILE(1:WTHD) = WRITEFILE(1:WTHD.0) U VERIFICATION CONDITIONS IN CURRENT
TPROC FROM PATH(0) FOLLOWING SPATHS PATH.0*PATHPT.0*BRANCH.0
ASSERT LINSPT=0 A LINE(0)=+FI 1
EXIT 1
PROCEDURE GENTERM 1
ASSERT LINSPT=0 A LINE(0)=+FI 1
ASSERT ALIST[ALSTNUM+COLONALTNUM+PATH+PATHPT] 1
ASSERT NSTEP=STEPS SINE :STEP PRINT 1
IF PATHPT=0 THEN WRITE BLANKLINE 1
WRITE BLANKLINE 1
WRITE BLANKLINE 1
ID=PROC(?CURRENTPROC) 1
ENTER PRINTIN 1
ENTER PRINTLINE 1
WRITE BLANKLINE 1
IF CURRENTPROC=INITIALPROCEDURE A CURRFRONT=0 THEN WRITE LINE1 1
WRITE LINE2 1
WRITE LINE3 1
WRITE LINE4 1
FI 1
ENTER WRITEASRST 1
WRITE DOTS 1
FI 1
STMT:=DESCLOC(PATH(PATHPT)+BIAS) 1
ASSERT LINSPT=0 A LINE(0)=+FI 1
ASSERT ALIST[ALSTNUM+COLONALTNUM+PATH+PATHPT] 1
ASSERT NSTEP=STEPS SINE :STEP PRINT 1
ASSERT PATHPT.0=0 A WRITEFILE(1:WTHD) WRITEFILE(1:WTHD.0) U VERIFICATION CONDITION
THROUGH DOTS 1
ASSERT PATHPT.0#0 A :WTHD:=:WTHD.0
ASSERT STMT:=DESCLOC(PATH(PATHPT)+BIAS) 1
CASE STATEMENT(STMT) OF
641 ENTER GENASSIGNTERM 1
951 COLONALTNUM(1)=COLONALTNUM(1)+11 SILOC$ NSTEP=NSTEP+1
781 ENTER GENENTRYTERM 1
911 ENTER GENHEADTERM 1
981 ENTER GENWRITETERM 1
851 ENTER GENTERM 1
721 ENTER GENCASETERM 1
ESAC 1
ASSERT WRITEFILE(1:WTHD) WRITEFILE(1:WTHD.0) U VERIFICATION CONDITION TERM FOR STATEMENT AT PATH(PATHPT) 1
ASSERT ALIST[ALSTNUM+COLONALTNUM+PATH+PATHPT+1] 1
ASSERT LINSPT=0 A LINE(0)=+FI 1
ASSERT NSTEP=STEPS SINE :STEP PRINT 1
EXIT 1
PROCEDURE GENWRITETERM1
ASSERT LINEPT=0 & LINE[0]=#I
ASSERT ALTLIST[ALTNUM+COLONALTNUM+PATH+PATHPT]=1
ASSERT STM1=DESCLOC[PATH[PATHPT+1]+BIAS]
ASSERT STATEMENT[STM1]=#81
ASSERT NSTEP=STEPS SINCE :STEP PRINT1
ID1=STATEMENT[STM1+1]
B=HOUDFUNCTION(ID1)
ENTER WRITEWRITELINE1
ENTER WRITEWRITELINE2
ENTER WRITEWRITELINE3
NSTEP=NSTEP+1
ASSERT WRITEFILE(1;THD)=WRITEFILE(1;THD,#0) U VERIFICATION CONDITION TERM FOR STATEMENT AT PATH[PATHPT+1]
ASSERT ALTLIST[ALTNUM+COLONALTNUM+PATH+PATHPT+1]=1
ASSERT LINEPT=0 & LINE[0]=#1
ASSERT NSTEP=STEPS SINCE :STEP PRINT1
EXIT 1

PROCEDURE IDENTSTR1
IF IDENTSTR1=IDENTSTR3 & IDENTSTR2=IDENTSTR4
THEN IDENTSTRFLAG := TRUE 1
RETURN 1
FI 1
IF IDENTSTR2=IDENTSTR1 & IDENTSTR4=IDENTSTR3
THEN IDENTSTRFLAG := FALSE 1
RETURN 1
FI 1
I := 0 1
ASSERT CHARLIST[0;IDENTSTR1,0]...CHARLIST[0;IDENTSTR1,0+1]=CHARLIST[0;IDENTSTR3,0]...CHARLIST[0;IDENTSTR3,0+1]
ASSERT IDENTSTR4=IDENTSTR3=IDENTSTR2=IDENTSTR1 1
ASSERT IDENTSTR1=IDENTSTR3=IDENTSTR2=IDENTSTR4 1
ASSERT IDENTSTR1=IDENTSTR3=IDENTSTR4=IDENTSTR2 1
ASSERT IDENTSTR1=IDENTSTR3=IDENTSTR4=IDENTSTR4 1
WHILE I < IDENTSTR2=IDENTSTR1 DO
IF CHARLIST[IDENTSTR1+I] # CHARLIST[IDENTSTR3+I]
THEN IDENTSTRFLAG := FALSE 1
RETURN 1
FI 1
I := 1+1
EXIT 1

PROCEDURE INPUT
ASSRT ~ RECOGNITION 1
ASSRT (~ASRTSCANFLAG & ATSCANSTATE & ATSAVEDPARSESTATE) V (~ASRTSCANFLAG & ATPA
RESTATE) V (~ASRTSCANFLAG & ATASRTSCANSSTATE) 1
ASSRT LINEPT=0 & LINE[0]=#1
ASSERT WRITEFILE(1;THD)=LISTING OF INPUTSTRING THROUGH :ROMO 1
IF 19STATE = STATES10A
$ 1.E. IN PAUSE NETWORK. $
THEN ENTER SAVEPARSESTATUS
ENTER SCAN
CARINSTRING := TOKEN
END RESTOREPARSESTATUS
ELSE IF -ASRTSCANFLAG THEN IF COL=81
THEN COL := 1
READ CARD
IF CARD(0) = +T
THEN CARINSTRING := 99
ELSE CARINSTRING := INTEGER(CARD(1))
END LIST
FI
ELSE CARINSTRING := INTEGER(CARD(COL))
COL := COL + 1
FI
ELSE IF ASRTSCANINTERFASRTBACK
THEN CARINSTRING:=INTEGER(CHARLIST{ASRTSCANINTER}1)
ASRTSCANINTER:=ASRTSCANINTER+1
ELSE CARINSTRING:=991
FI
FI
FI
ASSERT = RECOGNITION
ASSERT CA[N]STRING=CAR(INSTRING,0)
ASSERT INSTRING=CDR(INSTRING,0)
ASSERT LINEPT=0 AND LINE=0=+FI
ASSERT WRITEFILE:=(WHO)=LISTING OF INPUTSTRING THROUGH :ROMD
EXIT

PROCEDURE INSERTCHAR
ASSERT LINE=0=+FI
CHARLIST(CHARLSTPT1)=CHAR 1
LINEFRONT:=CHARLSTPT 1
LINEBACK:=CHARLSTPT 1
ENTER =ULC LINE 1
ASSERT WRITEFILE:=(WHO)=U LINE(0)...LINE(LINEPT)=WRITEFILE(,WHO)=U LINE.0(0)
...8LINE=6LINEPT=0=U CHAR.0 1
ASSERT LINE(0)=+FI
ASSERT $557 CHARLSTPT-1 = CHARLIST($57)=CHARLST.0($57)
EXIT

PROCEDURE INSERTFIRSTCHAR
CHARLIST(CHARLSTPT1)=CHARLIST(CHARLSTPT1)
CHARLIST(CHARLSTPT1)=CHARACTER(NEXTCHARACTER)
TOKENSTRING(0)=CHARLSTPT1
TOKENSTRING(1)=CHARLSTPT1
ASSERT TOKENSTRING=CHARACTER(NEXTCHARACTER)
ASSERT CHARLSTPT=CHARLSTPT,0+1
EXIT

PROCEDURE INSERTPRV
ASSERT LINE=PT=0 AND LINE(0)=+FI
LINEFRONT=331
LINEBACK=331
ENTER =ULC=FROMPRESET
ASSERT LINE(0)...LINE(6)=+F<PRV> + 1
ASSERT :WHO=WHO=0,01
EXIT

PROCEDURE INTPRINT
ASSERT INTVAL=201
ASSERT LINE(0)=FI
IF INTVAL=0
    THEN CHAR:=+0 1
    ENTER INSERTCHAR 1
ELSE
    LINEFRONT:=CHARLISTPT 1
    LINEBACK:=CHARLISTPT-1 1
    ASSERT CHARLIST(LineFront)....CHARLIST(LineBack) CONTAIN RIGHTMOST (LINEBACK=LIN
    EFRONT+1) DIGITS OF INTVAL.0 IN REVERSE ORDER!
    ASSERT INTVAL=INTVAL.0/10**((LINEBACK-LINEFRONT+1))
    ASSERT LINE=LINE.0
    ASSERT LinePT=LinePT.0
    ASSERT INT=INT+0 0 1
    WHILE INTVAL=0 DO
        LINEBACK:=LINEBACK+1 1
        CHARLIST(LineBack)::$CHARACTER(INTVAL*10+27)$
        INTVAL:=INTVAL/10 1
    ELIN=
    ASSERT CHARLIST(LineFront)....CHARLIST(LineBack) CONTAIN RIGHTMOST (LINEBACK=LIN
    EFRONT+1) DIGITS OF INTVAL.0 IN REVERSE ORDER!
    ASSERT WRITEFILE($INT=INT+0 0 1$ U LINE(0)...LINE(LinePT)=WRITEFILE($INT=INT+0 0 1$ U LINE(0)
    ...LINE(LinePT)=WRITEFILE($INT=INT+0 0 1$ U LINE(0)
    IF INT=INT+0 0 1
    ASSERT LINE(0)=FI
    EXIT 1
PROCEDURE ISANOTHERBRANCH 1
    IF BRANCH(PATHPT)=0
        THEN ANOTHERBRANCH=FALSE 1
    ELSE IF STATEMENT(DESCLOC(PATH(PATHPT)+BIAS1))#2 SCASES
        THEN IF BRANCH(PATHPT)=1
            THEN ANOTHERBRANCH=TRUE 1
            ELSE ANOTHERBRANCH=FALSE 1
        FI 1
    ELSE IF STATEMENT(DESCLOC(PATH(PATHPT)+BIAS1)+5)
        = STATEMENT(DESCLOC(PATH(PATHPT)+BIAS1)+4)*2 >BRANCH(PATHPTH)
        THEN ANOTHERBRANCH := TRUE 1
        ELSE ANOTHERBRANCH := FALSE 1
    FI 1
FI 1
    ASSERT ANOTHERBRANCH IFF THERE IS ANOTHER BRANCH POINT FROM PATH.0(PATHPT.0)
    EXIT 1
PROCEDURE ISCOLONID1
    AT:=11
    ASSERT 1<ATS:=11
    ASSERT 1<ATS:=1 < TOKEN.STRING * COLON IDENTIFIER $$$
    WHILE AT<10 DO
        IF COLONPTS(AT)=1-COLONPTS(AT)=TOKENSTRING(AT)=TOKENSTRING(0)
            THEN I:=10
            ASSERT I=SATS101
ASSERT $SS$<AT-1 = TOKEN.STRING \# COLON IDENTIFIER $SS$
ASSERT $SS$LENGTH OF TOKEN.STRING = 1
ASSERT $SS$1 = LETTER $SS$+1 OF TOKEN.STRING = LETTER $SS$+1 OF COLON IDENTIFIER

WHILE $I<\text{TOKENSTRING}(11)=\text{TOKENSTRING}(0) \land \text{CHARLIST}(\text{TOKENSTRING}(0)+1)=\text{COLONWORDSET}(\text{COLONPTS}(AT)+1) DO
    $I++$

ELIMWH
IF $I>\text{TOKENSTRING}(11)=\text{TOKENSTRING}(0)
THEN $\text{FOUND}:=\text{TRUE}$
$AT:=AT$
RETURN

FI

$AT:=AT+1$

ELIMWH
$\text{FOUND}:=\text{FALSE}$
ASSERT $\text{FOUND} = \text{TOKEN.STRING} \# \text{COLON IDENTIFIER} \# \text{AT}$
ASSERT $\neg\text{FOUND} = \text{TOKEN.STRING} \# \text{IS NOT A COLON IDENTIFIER}$

EXIT

PROCEDURE LabeLRefCheck: !
ASSERT $0<\text{TOKENSTRING}(0)$=\text{TOKENSTRING}(1)$=\text{CHARLISTPT}$!
ASSERT $0<\text{REFERENCEDLABELSETPT}=$ II
ASSERT $0<\text{REFERENCEDLABELSETPT}+1 = 0<\text{REFERENCEDLABELSET}($$)$=\text{CHARLISTPT}$ !
IDENTST$1=\text{TOKENSTRING}(0)$!
IDENTST$2=\text{TOKENSTRING}(11)$
AT$:=0$
ASSERT $0<\text{AT}$$=1 = \text{CHARLIST}0(\text{REFERENCEDLABELSET}0(2++$$))$$\ldots$$\text{CHARLIST}0(\text{TOKENSTRING}0(11))$$\ldots$$\text{CHARLIST}0(\text{TOKENSTRING}0(0))$$\ldots$$\text{CHARLIST}0(\text{TOKENSTRING}0(11))$

ASSERT $0<\text{AT}$$=0$$=\text{TOKENSTRING}0(0)$ II
ASSERT IDENTST$1=\text{TOKENSTRING}0(0)$ I
ASSERT IDENTST$2=\text{TOKENSTRING}0(11)$ I
ASSERT $\text{REFERENCEDLABELSET}0=\text{REFERENCEDLABELSET}0(0)$ I
ASSERT $\text{REFERENCEDLABELSETPT}=\text{REFERENCEDLABELSETPT0}$ I
ASSERT $\text{CHARLIST}0=\text{CHARLIST}0(0)$ I
ASSERT $\text{TOKENSTRING}0=\text{TOKENSTRING}0(0)$ I
WHILE AT$<$\text{REFERENCEDLABELSETPT} DO
IDENTST$3=\text{REFERENCEDLABELSET}(2++AT)$ I
IDENTST$4=\text{REFERENCEDLABELSET}(2++AT+1)$ I
ENTER IDENTST$4$
IF IDENTST$1$=FLAG$;
THEN $\text{FOUND}:=\text{TRUE}$ I
RETURN I

FI

$I:=AT+1$

ELWH
$\text{FOUND}:=\text{FALSE}$ I
ASSERT $\text{FOUND} = 0<\text{AT}$$=\text{REFERENCEDLABELSETPT}0$$=\text{CHARLIST}0(\text{REFERENCEDLABELSET}0(2++$$)$$\ldots$$\text{CHARLIST}0(\text{TOKENSTRING}0(11))$$\ldots$$\text{CHARLIST}0(\text{TOKENSTRING}0(0))$$\ldots$$\text{CHARLIST}0(\text{TOKENSTRING}0(11))$

ASSERT $\neg\text{FOUND} = 0<\text{AT}$$=\text{REFERENCEDLABELSETPT}0$$=\text{CHARLIST}0(\text{REFERENCEDLABELSET}0(2++$$)$$\ldots$$\text{CHARLIST}0(\text{TOKENSTRING}0(11))$$\ldots$$\text{CHARLIST}0(\text{TOKENSTRING}0(11))$

EXIT

PROCEDURE LISTI:
SPWNTS CURRENT CONTROL POINT FOLLOWED BY CARD IMAGE
$\text{ASSERT} ATSAVEDPARSESTATE$ I
$\text{ASSERT LINEPT}0 = LINE(0) = F1$
$\text{ASSERT CANO(0) ++}$
ASSERT WRITEFILE(:WTHD)=LISTING OF INPUTSTRING UP TO :RDHD1
IF PARSLSTATE=41 = PARSESTATE=36
    THEN INTERVAL=POINTER
    ENTER INPRINT1
    CHAR=1
    IF POINT<10
    THEN ENTER INSERTCHAR1
    FI
    ENTER INSERTCHAR1
    ELSE
    CHAR=1
    ENTER INSERTCHAR1
    ENTER INSERTCHAR1
    ENTER INSERTCHAR1
    ENTER INSERTCHAR1
    FII
II=1
ASSERT IS[1]=1
ASSERT CARD[0]=0
ASSERT LINE[0]=41
ASSERT LINE[1]=...LINE[LINEPT]= LISTING LABEL1
ASSERT LINE[LINEPT+1]=...LINE[LINEPT+1]=CARD[1]=...CARD[1]=1
ASSERT WRITEFILE():WTHD=0)=LISTING OF INPUTSTRING UP TO :RDHD1
WHILE IS=0 DO
    LINE[1+LINEPT]=CARD[1]
II=1+1
ELIMII
LINEPT=LINEPT+801
ENTER PRINTLINE
ASSERT WRITEFILE():WTHD=LISTING OF INPUTSTRING THROUGH :RDHD1
ASSERT LINEPT=0 = LINE[0]=*F1
EXIT1

PROCEDURE LISTCALLEDPROCS1
II=01
ASSERT 01<2001
ASSERT 05$1$=1 = -PROCFLAG($) = -PROCFLAG($$)$
ASSERT PROCALLED=PROCALLED+01
WHILE II<999 DO
    PROCFLAG[1]=FALSE1
    PROCFLAG[1]=FALSE1
    II=1+1
ELIMII
PROCFLAG(PROCALLED)=TRUE1
DONE=FALSE1
ASSERT DONE = PROCFLAG($) IFF PROCEDURE $S$ CAN BE REACHED BY A CALL TO PROCEDURE PROCALLED.01
ASSERT PROCFLAG(PROCALLED=01)
ASSERT PROCFLAG($) = PROCEDURE $S$ CAN BE REACHED BY A CALL TO PROCEDURE PROCALLED.01
ASSERT PROCFLAG($) = PROCFLAG($) = PROCFLAG($) IF PROCEDURE $S$ CAN BE REACHED
BY A CALL TO PROCEDURE $S$1
ASSERT PROCALLED=PROCALLED+01
WHILE -DONE DO
    DONE = TRUE1
    PROCNUM=01
ASSERT 0<PROCNUM<=2001
ASSERT DONE = 05$5$=PROCNUM-1 = PROCFLAG($) = PROCFLAG($)1
ASSERT PROCFLAG(PROCALLED=01)
ASSERT PROCALLED=PROCALLED.01
ASSERT PROCFLAG($) = PROCEDURE $S$ CAN BE REACHED BY A CALL TO PROCEDURE PROC Cald.01
ASSERT PROCFLAG(SS) A PROCFLAG(*1) IF PROCEDURE * CAN BE REACHED
BY A CALL TO PROCEDURE SS
WHILE PROCNUM<199 DO
IF PROCFLAG(PROCNUM) = PROCFLAG(*1)
THEN ENTER LISTPROCCALLS
PROCFLAG[PROCNUM] = TRUE
FI
PROCNUM = PROCNUM+1
ELSE
ASSERT PROCFLAG(SS) IFF PROCEDURE SS CAN BE REACHED BY A CALL TO PROCEDURE PROCCALLED,0,1
EXIT 1
PROCEDURE LISTPROCCALLS:
I = PROC(7*PROCNUM+4)
ASSERT PROC(7*PROCNUM+4+1) <= PROC(7*PROCNUM+4+4)
ASSERT DONE = PROCFLAG*PROCFLAG,0,1
ASSERT (PROC(7*PROCNUM+4+3) <= PROC(7*PROCNUM+4+11) A (CALLED PROCEDURE SS = PROCEDURE *) A PROC
FLAG(*1)
WHILE I <= PROC(7*PROCNUM+4+4) DO
FINDI1 = PROCFLAG(2*I+1)
FINDI2 = PROCFLAG(2*I+1)
ENTER FINDI1
K = FINDI1
ASSERT PROCNUM = PROCNUM,0,1
ASSERT PROC(7*PROCNUM+4+3) <= PROC(7*PROCNUM+4+11)
ASSERT DONE = PROCFLAG*PROCFLAG,0,1
ASSERT (PROC(7*PROCNUM+4+3) <= PROC(7*PROCNUM+4+11) A (CALLED PROCEDURE SS = PROCEDURE *) A PROC
FLAG(*1)
ASSERT CALLED PROCEDURE I = IDENTIFIER AT I
WHILE PROC(7*K)#AT DO
K = K+1
ELIMINATE
IF -PROCFLAG[K]
THEN DONE = FALSE
PROCFLAG(K) = TRUE
FI
I = I+1
ELSE
ASSERT DONE = PROCFLAG*PROCFLAG,0,1
ASSERT PROCFLAG(SS) IFF PROCEDURE SS CAN BE REACHED BY A CALL TO PROCEDURE PROCNUM
M,0,1
EXIT 1
PROCEDURE LOOPCHECK:
ASSERT PATH(0) = PATH(PATHP1) CONTAINS NO UNTAGGED LOOPS
I = 1
ASSERT S1 = PATHP1,0+1
ASSERT PATH = PATHP1,0+1
ASSERT PATHP1 = PATHP1,0+1
ASSERT S1 <= PATH(0) = PATH(0) = PATH(0) = PATH(0) = PATH(0) = PATH(0) = PATH(0) = PATH(0) = PATH(0)
WHILE S1 = PATHP1 DO
IF PATH([I] = PATH(PATHP1) THEN WRITE LOOP 1
ASSERT PATH([I] = PATH(PATHP1) CONTAINS AN UNTAGGED LOOP
ASSERT WRITEFILE(:WTHO) = WRITEFILE(:WTHO) U LOOP1
HALT 1
ELSE I = I+1
FI
FI
ELIMW 1
ASSRT PATH.0(0) ... PATH.0(PATHPT.0+1) CONTAINS NO UNTAGGED LOOPS 1
ASSRT :PTHD = :PTHD.01
EXIT 1

PROCEDURE NILMATCH 1
WHILE ARC = FIRSTHC(STATE+1) I SYMBOL(ARC) = 0 DO ENTER TESTS 1 IF TESTSFLAG
THEN NILMATCHFLAG := TRUE 1 RETURN 1
FI 1
ARC := ARC + 1 1
ELIMW 1
NILMATCHFLAG := FALSE 1
EXIT 1

PROCEDURE PARSE 1
ASSRT READFILE(:RDMD) = INPUTSTRING 1
ASSRT CURRENTPROC = -1 1
ASSRT STATE = ENTPT = -1 1
ASSRT ASRTLOCOPT = -1 1
ASSRT ASRTLOC1[0] = 0 1
ASSRT ASRTLOC1[0] = -1 1
ASSRT ASRTLOC1[0] = 0 1
ASSRT ASRTLOC1[0] = -1 1
ASSRT CASELABELSETPT = -1 1
ASSRT CASELABELSETTOP = -1 1
STATE := PARSESTART 1
PARSESTATE := PARSESTART 1
READ CARD 1
LINEPT := 0 1
LINE(0) := *F 1
IF CARD(0) <= T THEN ENTER LIST 1
FI 1
COL := 1 1
RECOGNITION := FALSE 1
ASRTSCANFLAG := FALSE 1
ENTLP CAN 1
CARSTRING := TOKEN 1
PRINSTACKTOP := -1 1
ASSRT ATPARSESTATE 1
ASSRT ASRTSCANFLAG := 1
ASSRT LINEPT := LINE(0) = *F 1
ASSRT WRITEFILE(:PTHD) = LISTING OF INPUTSTRING THROUGH :RDMD 1
WHILE - RECOGNITION DO ENTER TRANSNET 1
ELIMW 1
ASSRT PROGRAMALSET ASRTLOC ASRTLOC1 ASRTSOUNDFUNCTION CASELABELFRONT CASELABELS
CASELABELSET CHARLIST DEFINEDIDENTIFIERSET DEFINEDIDENTIFIERSETDEFINEDPR
PROCEDURESET DEFINEDPROCEDURESET DESLOC EXPLIST INITIALPROCEDURE PROC PROCalls
STATE = ENTPT 1
ASSRT WRITEFILE (:PTHD) = LISTING 1
ASSRT LINEPT := 0 LINE(0) = *F 1
EXIT 1

PROCEDURE PRINTCASELABELS 1
ASSRT LINE(0) = *F 1
IF FRONTE LABEL = BACKLABEL
THEN RETURN

IVAL=CASELABELS(FRONTLABEL)
ENTER INPRINT

ASSERT FRONTLABEL=BACKLABEL

ASSERT WRITEFILE(\TBD) U LINE(1) \LINE(\LINEPT)=WRITEFILE(\TBD) U LINE(0)

**LINE (0) LINEPT) U CASELABELS(FRONTLABEL) **CASELABELS(FRONTLABEL)

\

ASSERT LINE(1)\LINE(0)11
WHILE FRONTLABEL=BACKLABEL DO
FRONLABEL=FRONLABEL+1
CHAR=1
ENTER INSERTCHAR
CHAR=+1
ENTER INSERTCHAR
CHAR=1
ENTER INSERTCHAR
IVAL=CASELABELS(FRONTLABEL)
ENTER INPRINT

ELIMNI

ASSERT WRITEFILE(\TBD) U LINE(1) \LINE(\LINEPT)=WRITEFILE(\TBD) U LINE(0)

**LINE (0) LINEPT) U CASELABELS(FRONTLABEL) **CASELABELS(FRONTLABEL)

ASSERT LINE(0)\LINE(0)11
EXIT 1

PROCEDURE PRINTID1

ASSERT LINE(0)\F1
LINEFRONT=DEFINEIDENTIFIERSET(\ID)
LINEBACK=DEFINEIDENTIFIERSET(\ID)
ENTER GUILDLINE

ASSERT WRITEFILE(\TBD) U LINE(1) \LINE(\LINEPT)=WRITEFILE(\TBD) U LINE(0)

**LINE (0) LINEPT) U DEFINEIDENTIFIER ID 01

ASSERT LINE(1)\F1
EXIT 1

PROCEDURE PRINTIDUS91

ASSERT LINE(1)\F1
ENTER PRINTID
IF ALTNUM(ID) = 0 THEN CHAR=+1
ENTER INSERTCHAR
IVAL=ALTNUM(ID)
ENTER INPRINT

F11

ASSERT WRITEFILE(\TBD) U LINE(1) \LINE(\LINEPT)=WRITEFILE(\TBD) U LINE(0)

**LINE (0) LINEPT) U IDENTIFIER ID SUBSCRIPTED  WITH ITS ALTERATION COUNTER

ASSERT LINE(0)\LINE(0)11

EXIT 1

PROCEDURE PRINTIDUSX1

ASSERT LINE(1)\F1
ASSERT ALTNUM(ID)=0
ENTER PRINTID
CHAR=+1
ENTER INSERTCHAR
IVAL=ALTNUM(ID)+1
ENTER INPRINT

ASSERT WRITEFILE(\TBD) U LINE(1) \LINE(\LINEPT)=WRITEFILE(\TBD) U LINE(0)

**LINE (0) LINEPT) U IDENTIFIER ID SUBSCRIPTED  WITH ITS ALTERATION COUNTER

ASSERT LINE(0)\LINE(0)11

EXIT 1
PROCEDURE PRINTLINE 1
   ASSERT 0 $$SS$$LINEPT.0 = LINE.0 $$SS$$ 1
   ASSERT LINEPT.0 $$SS$$SS$$LINEPT = LINE.0 = BLANK 1
   WHILE LINEPT < 120 DO
      LINEPT := LINEPT + 1
      LINE(LINEPT) := + 1
   ENDWH
   WRITE LINE 1
   LINEPT := 01
   ASSERT WRITEFILE (:WTHD.0) = WRITEFILE (:WTHD.0) U LINE.0(0) $$SS$$ LINE.0(LINEPT.0) 1
   ASSERT LINEPT = 01
   ASSERT LINE(0) = LINE.0(0)1
   EXIT 1

PROCEDURE PRINTPATH 1
   ASSERT LINEPT = 0 + LINE(0) = F1
   ASSERT Path(0) $$SS$$ Path(PathPT) IS A PATH1
   WRITE PATHS 1
   I := 01
   ASSERT WRITEFILE (:WTHD.0) U LINE(1) $$SS$$ LINE(LINEPT) = WRITEFILE (:WTHD.0) U PATHS U
   Path(0) $$SS$$ Path(1)1
   ASSERT LINE(0) = F1
   ASSERT 0 $$SS$$ PathPT + 1
   ASSERT Path = Path.01
   ASSERT PathPT = PathPT.01
   ASSERT 0 $$SS$$ LinePT = 1201
   WHILE ISPANTPT DO
      CHAR := + 1
      ENTER INSERTCHAR 1
      ENTER INPRINT 1
      I := 11
   ENDWH
   ENTER PRINTLINE 1
   ASSERT LINEPT = 0 + LINE(0) = F1
   ASSERT WRITEFILE (:WTHD.0) = WRITEFILE (:WTHD.0) U PATHS U PATH.0(0) $$SS$$ PATH.0(PATHPT + 01)
   EXIT 1

PROCEDURE PRINTARCHSUB 1
   ASSERT LINE(0) = F1
   IF COLONALTNUM(3) > 0 THEN CHAR := + 1
      ENTER INSERTCHAR 1
      INTVAL := COLONALTNUM(3)
      ENTER INPRINT 1
   FI1
   ASSERT WRITEFILE (:WTHD.0) U LINE(1) $$SS$$ LINE(LINEPT) = WRITEFILE (:WTHD.0) U LINE.0(11)
   $$SS$$ LINE(LINEPT + 1) U ALTERATION COUNTER OF :ARCH1
   ASSERT LINE(0) = LINE.0(1)1
   EXIT 1

PROCEDURE PRINTSUBEXPI
   ASSERT SUBBACK = SUBBACK.01
   ASSERT SUBFRONT.0 $$SS$$ SUBFRONT $$SS$$ SUBBACK.0 = 11
   ASSERT WRITEFILE (:WTHD.0) U LINE(1) $$SS$$ LINE(LINEPT) = WRITEFILE (:WTHD.0) U LINE.0(11)
   $$SS$$ LINE(LINEPT) U SUBSCRIPTED EXPRESSION IN EXPLIST FROM SUBFRONT.0 TO SUBF
   RQINTPT = 11
   ASSERT LINE(0) = F1
   WHILE SUBFRONT < SUBBACK DO
      CASE EXPLIST(2 $$SS$$ SUBFRONT) OF
         $CHARACTER,CONSTANTS 65: CHAR := + 1
ENTER INSERTCHAR
CHARI=CHARACTER(EXPLIST(2*SUBFRONT+1))
ENTER INSERTCHAR
$NUMBERS 66: INTVAL=EXPLIST(2*SUBFRONT+1)
ENTER INPRINT
$IDENTIFIERS 67: ID=EXPLIST(2*SUBFRONT+1)
ENTER PRINTID
IF ALTNUM(ID)>0
THEN CHAR=+1
ENTER INSERTCHAR
INTVAL=ALTNUM(ID)
ENTER INPRINT
FI
$BOOLEANS 71: AT=+1
ENTER RESWORDPRINT
$CHARACTERS 74: AT=+1
ENTER RESWORDPRINT
$FALSE 91: AT=+1
ENTER RESWORDPRINT
$INTEGERS 77: AT=+1
ENTER RESWORDPRINT
$TRUE 96: AT=+1
ENTER RESWORDPRINT
$SINGLE CHARACTERS ELSE CHAR=CHARACTER(EXPLIST(2*SUBFRONT))
ENTER INSERTCHAR
ESAC
SUBFRONT=SUBFRONT+1
ELIMINATE
ASSERT WRITEFILE(1:WTMD) U LINE(1)...LINE(LINEPT)=WRITEFILE(1:WTMD.0) U LINE.0(1)
...LINE.0(LINEPT.0) U SUBSCRIPTED EXPRESSION IN EXPLIST FROM SUBFRONT.0 THROUGH
SUBFRONT.0
ASSERT LINE.0()==F1
EXIT 1
PROCEDURE PRINTWTMDSU1
ASSERT LINE.0()==F1
IF COLONALTNUM(4)>0
THEN CHAR=+1
ENTER INSERTCHAR
INTVAL=COLONALTNUM(4)
ENTER INPRINT
FI
ASSERT WRITEFILE(1:WTMD) U LINE(1)...LINE(LINEPT)=WRITEFILE(1:WTMD.0) U LINE.0(1)
...LINE.0(LINEPT.0) U ALTERATION COUNTER OF :WTMD1
ASSERT LINE.0==LINE.0(0)
EXIT 1
PROCEDURE PROCVCVGEN 1
ASSERT LINE.0==LINE.0()==F1
CURRFONT = 0
ENTER FROMPOINTVCVGEN
CURRFONT = 1
ASSERT CURRFONT<PROCTYPE*CURRENTPROC+21+1
ASSERT WRITEFILE(1:WTMD)=WRITEFILE(1:WTMD.0) U VERIFICATION CONDITIONS FOR PATHS 1
N PROCEDURE CURRENTPROC STARTING FROM POINTS UP TO CURRFONT
ASSERT LINE.0==LINE.0()==F1
WHILE CURRFONT<PROCUTE*CURRENTPROC+21 DO
IF STATEMENTDESCLOC(CURRFONT+BIAS)%64 SHALT ASRTLOC(CURRFONT+BIAS)%1
THEN ENTER FROMPOINTVCVGEN
FI
CURRFONT = CURRFONT + 1
ELIMINATE
ASSERT WRITEFILE(1:THD)=WRITEFILE(1:THD,0) U VERIFICATION CONDITIONS FOR PROCEDURE
CURRENTPROC1
ASSERT LINEPT=0 \& LINE(0)=*F1
EXIT 1

PROCEDURE REGEN 1
ASSERT THERE IS ANOTHER $BRANCH$ FROM PATH(PATHPT)1
ASSERT LINEPT=0 \& LINE(0)=*F1
R:=PATHPT1
PATHPT:=0 1
NSTEP:=01
ENTER RESETALT 1
ASSERT R=PATHPT,01
ASSERT 0=PATH-PTPATHPT,01
ASSERT ALTLIST(NUM-ColonaltNum-PATH-PATHPT)1
ASSERT WRITEFILE(1:THD)=WRITEFILE(1:THD,0) U VERIFICATION CONDITION TERMS FOR PA
TH(0),...PATH(PATHPT=1)1
ASSERT PATH =PATHPT,01
ASSERT NSTEP=STEPS SINCE STEP PRINT1
ASSERT LINEPT=0 \& LINE(0)=*F1
WHILE PATHPT<RE 00
ENTER GENTERM 1
PATHPT:=PATHPT+1 1
ELEWH 1
BRANCH(PATHPT):=EPANCH(PATHPT)+1 1
ASSERT WRITEFILE(1:THD)=WRITEFILE(1:THD,0) U VERIFICATION CONDITION TERMS FOR PA
TH(0),...PATH(0)(PATHPT,0)=1
ASSERT PATH =PATHPT,01
ASSERT ALTLIST(NUM-ColonaltNum-PATH-PATHPT)1
ASSERT NSTEP=STEPS SINCE STEP PRINT1
ASSERT LINEPT=0 \& LINE(0)=*F1
EXIT 1

PROCEDURE RESETALT 1
I:=01
ASSERT 051=250 1
ASSERT 0=SS$SS$-1 = ALTNUM(SS$)=01
WHILE I249 DO
ALTNUM(I1)=01
I:=I+1 1
ELEWH 1
I:=I+1
ASSERT 055=249 = ALTNUM(SS$)=01
ASSERT 15111 1
ASSERT SS$SS$-1 = ColonaltNum(SS$)=01
WHILE 1510 DO
ColonaltNum(I1)=01
I:=I+1 1
ELEWH 1
ASSERT 055=249 = ALTNUM(SS$)=01
ASSERT 155510 = ColonaltNum(SS$)=01
EXIT 1

PROCEDURE RESETFLAGSI
I:=01
ASSERT 051=250 1
ASSERT 0=SS$SS$-1 = ALFLAG(SS$)
WHILE 1249 DO
ALFLAG(I1)=FALSE 1
I:=I+1 1
ELEWH 1
II = 21
ASSERT 0$$<249 < ALTFLAG($$)
ASSERT 2$$=01
ASSERT 2$$=0-1 < COLONALFLAG($$)
WHILE 1 $$Do
   COLONALFLAG(1) := FALSE
   I := I + 1
ELIMW
   COLONALFLAG(1) := TRUE
   COLONALFLAG(10) := TRUE
   ASSERT ALL ALTERATION FLAGS IN ALTFLAG AND COLONALFLAG ARE FALSE EXCEPT COLONAL
   TFLAG(1) AND COLONALFLAG(10) WHICH ARE ALWAYS TRUE
EXIT

PROCEDURE RESTOREPARSESTATUS
   ASSERT 1$$=PARSESTACKTOP
   STATE := PARSESTATE
   I := 0
   ASSERT 0$$=PARSESTACKTOP0$$
   ASSERT 0$$=RTNSTACK($$) = PARSESTACK 0$$
   ASSERT STATE = PARSESTATE 0
   ASSERT PARSESTACKTOP = PARSESTACKTOP0
   WHILE 1 $$PARSESTACK TOP $$DO
      RTNSTACK(1) := PARSESTACK(1)
      I := I + 1
ELIMW
   RTNSTACKTOP := PARSESTACKTOP
   ASSERT STATE = PARSESTATE0
   ASSERT 0$$=PARSESTACKTOP0$$
   ASSERT RTNSTACK($$) = PARSESTACK0$$
   ASSERT RTNSTACKTOP = PARSESTACKTOP0
EXIT

PROCEDURE RESWORDPRINT
   ASSERT LINE(0) = +F1
   I := RESWORDPTS(11)
   ASSERT RESWORDPTS(A1) {RESWORDOPTS(A1)}
   ASSERT WRITEFILE(1) $$U LINE(1) $$LINE(1) := "WRITEFILE(1) $$U LINE(0) $$LINE(1) := "LINE(0) $$LINE(1) := "LINE(0) $$LINE(1) := "LINE(0) $$LINE(1) := "LINE(0)
   ASSERT LINE(0) = +F1
   WRITEFILE(1) $$U LINE(1) $$LINE(1) := "WRITEFILE(1) $$U LINE(0) $$LINE(1) := "LINE(0) $$LINE(1) := "LINE(0) $$LINE(1) := "LINE(0)
   ASSERT LINE(0) = +F1
EXIT

PROCEDURE SAVEPARSESTATUS
   ASSERT 1$$=RTNSTACKTOP
   PARSESTATE := STATE
   I := 0
   ASSERT PARSESTATE = STATE 0
   ASSERT 0$$=RTNSTACKTOP0$$
   ASSERT RTNSTACKTOP := RTNSTACK 0$$
   ASSERT RTNSTACKTOP = RTNSTACKTOP 0
   WHILE 1 $$RTNSTACK TOP $$DO
      PARSESTACK(1) := RTNSTACK(1)
      I := I + 1
ELIMW
   RTNSTACKTOP := RTNSTACKTOP
ASSERT PARSERSTATE = STATE.0
ASSERT 0:$$=RTNSTACK.TOP,0 = PARSESTACK(55)$$=RTNSTACK.0(55)$$
ASSERT PARSESTACKTOP = RTNSTACKTOP,0
EXIT

PROCEDURE SCAN
ASSERT ~ RECOGNITION
ASSERT INPUTSTRING = CARD(0)*CARD(COL)$$..$$CARD(HD)*READFILE:(ROMO)
ASSERT ASPTRSCANFLAG
ASSERT LINEP=0 & LINE(0)$$=$$F1
ASSERT WRITEFILE(1:WH0)$$=$$LISTING OF INPUTSTRING THROUGH :ROMO
ASSERT ATSAVEDPARSESTATE
STATE := SCANSTART
RTNSTACKTOP := -1
IF CARD(0)$$=$$T
THEN CARINSTRING := 99
ELSE CARINSTRING := INTEGER(CARD(COL))
FI
ASSERT ATSCANSTATE
ASSERT ~ ASPTRSCANFLAG
ASSERT LINEP=0 & LINE(0)$$=$$F1
ASSERT WRITEFILE(1:WH0)$$=$$LISTING OF INPUTSTRING THROUGH :ROMO
ASSERT ATSAVEDPARSESTATE
WHILE ~ RECOGNITION DO
ENTER TRANSET
ELIM
RECOGNITION := FALSE
ASSERT INSTRING$$=$$EMOVETOKEN(INSTRING,0)
ASSERT TOKENSTRING$$=$$EATTOKENSTRING(INSTRING,0)
ASSERT ~ RECOGNITION
ASSERT LINEP=0 & LINE(0)$$=$$F1
ASSERT WRITEFILE(1:WH0)$$=$$LISTING OF INPUTSTRING THROUGH :ROMO
EXIT

PROCEDURE SETALTIDS;
PROCNUM:=001
ASSERT 0$$=$$PROCNUM$$=$$001
ASSERT 0$$=$$PROCNUM$$=$$PROCNUM-1$$=$$PROCFLAG(55)$$=$$ALTERATION FLAG IS TRUE FOR ALL IDENTIFIERS IN ALTERATION SET OF PROCEDURE $$55$$ WHILE PROCNUM$$=001$$ DO
IF PROCFLAG(1:PROCNUM-1)
THEN I:=PROC(1:PROCNUM+51)
ASSERT 0$$=$$PROCNUM$$=001$$
ASSERT 0$$=$$PROCNUM$$=$$PROCNUM-1$$=$$PROCFLAG(55)$$=$$ALTERATION FLAG IS TRUE FOR ALL IDENTIFIERS IN ALTERATION SET OF PROCEDURE $$55$$ WHILE I$$=$$PROC(1:PROCNUM+51) DO
IF ALTSET(1)$$=0$$
THEN ALTFLAG(ALTSET(1))$$=$$TRUE
ELSE COLONALFLAG(ALTSET(1))$$=$$TRUE
FI
ELSE I:=I+1
FI
PROCNUM:=PROCNUM+1
ELIM
ASSERT PROCFLAG(55)$$=$$ALTERATION FLAG IS TRUE FOR ALL IDENTIFIERS IN ALTERATION SET OF PROCEDURE $$55$$
EXIT


PROCEDURE SETUP:
$ INITIALIZES SEVERAL VARIABLES USED IN ACTIONS $
I:=01
ASSEPT 0;1520001
ASSEPT 0;555=1 ASRTLOC(5S)=11
WHILE IS;999 DO
ASRTLOC(I):=11
1:=1
1:=1
ELIF1
CURRENTPRO:-11
STATEMENTPTI=-11
ASRTLOCPT:-11
ASRTLOC(I):=01
EXPSTPT:=-11
EXPSTRING(O):=01
EXPSTRING(I):=-11
CASELABELSETPT:=-11
CASELABELFRONT(O):=01
DEFINEDCASELABELSETTOP:=-11
ASSEPT CURRENTPROC=-11
ASSEPT STATEMENTPT=-11
ASSEPT ASRTLOCPT=-11
ASSEPT ASRTLOC(I):=01
ASSEPT ASRTLOC(5S)=11
ASSEPT EXPSTPT=-11
ASSEPT EXPSTRING(0)=01
ASSEPT EXPSTRING(I):-11
ASSEPT CASELABELSETPT=-11
ASSEPT CASELABELFRONT(O):=01
ASSEPT DEFINEDCASELABELSETTOP=-11
EXIT

PROCEDURE STACKTEST 1
IF RNSTACKTOP = 0 
THEN $ := ARC 
ARC := RNSTACK(RNSTACKTOP) 1
ENTER TESTS 1
ARC := $ -1
IF TESTFLAG 
THEN STACKTESTFLAG := TRUE 1
RETURN 1
FI 1
STACKTESTFLAG := FALSE 1
EXIT 1

PROCEDURE SUBSCRIPT:
ASSEPT A$=TFLAG = 0 IF NOT IN ENTER STATEMENT, #1 IF IN INITIAL ASSERTION OF ENTER STATEMENT AND #2 IF IN FINAL ASSERTION OF ENTER STATEMENT
ASSEPT ALNLIST(ALNNUM, COLONALNUM+PATHEPPTH)
ASSEPT AT IS IDENTIFIER NUMBER OR COLON IDENTIFIER NUMBER
ASSEPT CHARLIST(ASRTSCAPINTER)...CHARLIST(ASRTBACK) IS PORTION OF ASSERTION FOLLOWING IDENTIFIER AT
ASSEPT LINE(I):=T
ASSEPT N$STEP=STEPS SINCE STEP PRINT
IF AT:0 THEN INTVAL:=ALNNUM(AT)
ELSE INTVAL:=COLONALNUM(-AT)
FI 1
ASSEPT ASRTFLAG=ASRTFLAG 01
ASSEPT ALNUM=ALNUM 01
ASSERT COLONALTNUM=COLONALTNUM.01
ASSERT PATH=PATH.01
ASSERT PATHPT=PATHPT.01
ASSERT AT=AT.01
ASSERT CHARLIST=CHARLIST.01
ASSERT ASRTSCANPOINTER=ASRTSCANPOINTER.01
ASSERT ASRTBACK=ASRTBACK.01
ASSERT LINED=LINE.01
ASSERT INIVAL=INIVAL.01
ASSERT NSTEP=NSTEP.01
ASSERT INIVAL=IF AT20 THEN ALTNUM(AT) ELSE COLONALTNUM(-AT)
ASSERT WTMD=WTMD.01
IF ASRTSCANPOINTER+1=ASRTBACK THEN CHARLIST(ASRTSCANPOINTER)=1
IF ASRTSCANPOINTER+1=ASRTSCANPOINTER THEN CHARLIST(ASRTSCANPOINTER)=0
IF ASRTFLAG=0 THEN PATH(0)=0
THEN CHAR=0
1
ENTER INSERTCHAR
RETURN
IF ASRTFLAG=2 THEN IF AT20 THEN IF ALTFLAG(AT) THEN INIVAL=INIVAL-1
ELSE IF COLONALTFLAG(AT) THEN INIVAL=INIVAL-1
EXIT
ASSERT INIVAL=ALTERATION COUNTER FOR IDENTIFIER AT.01
ASSERT :WTMD:WTMD.01
ASSERT LINED=LINE.01
ASSERT INIVAL=INIVAL.01
ASSERT IF ASRTSCANPOINTER+1=ASRTBACK THEN CHARLIST(ASRTSCANPOINTER)=1
THEN ASRTSCANPOINTER=ASRTSCANPOINTER+1 ELSE ASRTSCANPOINTER=
ASRTSCANPOINTER+1
IF INIVAL=0 THEN CHAR=0
ENTER INSERTCHAR
EXIT
ASSERT WRITEFILE(:WTMD): LINE(1): WRITEFILE(:WTMD,0): LINE.0(1)
ASSERT IF ASRTSCANPOINTER+1=ASRTBACK THEN CHARLIST(ASRTSCANPOINTER)=1
THEN ASRTSCANPOINTER=ASRTSCANPOINTER+1 ELSE ASRTSCANPOINTER=
ASSRTSCANPOINTER+1
EXIT
PROCEDURE TESTS 1
ASSERT [-ASRTSCANFLAG ATSCANSTATE ATSAVEDPARSESTATE] I [-ASRTSCANFLAG ATPA
SESTATE] I [ASRTSCANFLAG ATASCANSTATE] 1
CASE TEST(AVAC) U F
0: AWCTEST := TRUE 1
11 ENTER TEST1 1
21 ENTER TEST2 1
31 ENTER TEST3 1
41 ENTER TEST4 1
5: ENTER TEST 5
6: ENTER TEST 6
7: ENTER TEST 7
8: ENTER TEST 8
9: ENTER TEST 9
10: ENTER TEST 10
11: ENTER TEST 11
12: ENTER TEST 12
13: ENTER TEST 13
14: ENTER TEST 14
15: ENTER TEST 15
16: ENTER TEST 16
17: ENTER TEST 17
18: ENTER TEST 18
19: ENTER TEST 19
20: ENTER TEST 20
21: ENTER TEST 21
22: ENTER TEST 22
23: ENTER TEST 23
24: ENTER TEST 24
25: ENTER TEST 25
26: ENTER TEST 26
27: ENTER TEST 27
28: ENTER TEST 28
29: ENTER TEST 29
30: ENTER TEST 30
31: ENTER TEST 31
32: ENTER TEST 32
33: ENTER TEST 33
34: ENTER TEST 34
35: ENTER TEST 35
36: ENTER TEST 36
37: ENTER TEST 37

ESAC
ASSERT TESTFLAG IFF ARCTEST(ARC, 0) 1
ASSERT ARC = ARC, 0 1
ASSERT STATE = STATE, 0 1
ASSERT TST = TST, 0 1
ASSERT TINSTACKTOP = TINSTACKTOP, 0 1
ASSERT S = S, 0 1
ASSERT CARINSTRING = CARINSTRING, 0 1
ASSERT i = i, 0 1
EXIT 1

PROCEDURE TRANSEND
ASSERT (¬ASRTSCANFLAG ∧ ATASCANSTATE ∧ ATASAVEDPARSESTATE) ∨ (¬ASRTSCANFLAG ∧ ATAPARSESTATE) ∨ (ASRTSCANFLAG ∧ ATASRTSCANSTATE) 1
ASSERT ¬RECOGNITION 1
ASSERT ¬ASRTSCANFLAG ∧ LINEPT ≠ 0 ∧ LINE(0) ≠ F1
ASSERT ¬ASRTSCANFLAG ∧ WRITEFILE(1, WIP, D) ≠ LISTING OF INPUTSTRING THROUGHIRDMDP
ENTER ALPHAEMATCH 1
IF ALPHAEMATCHFLAG
THEN ENTER TRAVERSE 1
ELSE ENTER NILMATCH 1
IF NILMATCHFLAG
THEN ENTER TRAVERSE 1
ELSE IF ARC < FISTARCTESTSTATE = 1) ∧ SYMBOL(ARC) < 0
THEN TINSTACKTOP := TINSTACKTOP + 1 1
TINSTACK(TINSTACKTOP) := ARC 1
STATE := ¬SYMBOL(ARC) 1
ELSE IF RECOGNITIONSTATE(ARC)
THEN ENTER STACKTEST 1
IF STACKTESTFLAG
   THEN ARC := RTINSTACK(RTINSTACKTOP) 1
   RTINSTACKTOP := RTINSTACKTOP+1 1
   ENTER TRAVERSE 1
   ELSE IF PTINSTACKTOP < 0
      THEN RECOGNITION := TRUE 1
      ELSE WRITE NONRECOGNITION 1
      FI 1
   ASSERT INPUTSTRING NOT RECOGNIZED BY TRANSITION NETWORK 1
   ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD,0) U NONRECOGNITION MESSAGE 1
   HALT 1
   FI 1
   ELSE WRITE NONRECOGNITION 1
   ASSERT INPUTSTRING NOT RECOGNIZED BY TRANSITION NETWORK 1
   ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD,0) U NONRECOGNITION MESSAGE 1
   HALT 1
   FI 1

FI 1

ASSERT ASRTSCANFLAG,0 = ATASRTSCANSTATE 1
ASSERT ASRTSCANFLAG,0 = ATSCANSTATE,0 = ATSCANSTATE 1
ASSERT ASRTSCANFLAG,0 = ATPARSESTATE,0 = ATPARSESTATE 1
ASSERT LINEHOT=0 = LINE(0)+FI 1
ASSERT ASRTSCANFLAG,0 = WRITEFILE(:WTHD) LISTING OF INPUTSTRING THROUGH :ROMDI 1
ASSERT ASRTSCANFLAG,0 = WRITEFILE(:WTHD) EXIT 1

PROCEDURE TRAVERSE 1
   IF FLAG(ARC) 1
      THEN ENTER INPUT 1
      FI 1
   ENTER ACTIONS 1
   STATE := NEXTSTATE(ARC) 1
   EXIT 1

PROCEDURE UPDATEALTNUM 1
   ENTER WSETFLAGS 1
   ENTER LISTCALLEDPROCST 1
   ENTER SETALTDOS 1
   IF #G1
      ASSERT OS12501 1
      ASSERT ALTERATION FLAGS IN ALTFLAG AND COLONALTFLAG ARE TRUE IFF THE CORRESPONDING IDENTIFIER CAN BE ALTERED BY A CALL TO PROCEDURE PROCALLED.01 1
      ASSERT PROCALLED=PROCALLED.01 1
      ASSERT ALTERATION COUNTERS IN ALTNUM HAVE BEEN UPDATED FOR IDENTIFIERS 0 THROUGH I-1 1
      WHILE IS249 00
         IF ALTFLAG(I) 1
            THEN ALTNUM(I):=ALTNUM(I)+11
            FII 1
            I:=I+11
         ELIMI
            COLONALTNUM(I):=COLONALTNUM(I)+11
         IF COLONALTFLAG(I) 1
            THEN COLONALTNUM(3):=COLONALTNUM(3)+11
            FII 1
         IF COLONALTFLAG(I) 1
            THEN COLONALTNUM(4):=COLONALTNUM(4)+11
            FII 1
         COLONALTNUM(10):=COLONALTNUM(10)+11
      ASSERT ALTERATION COUNTERS IN ALTNUM AND COLONALTNUM HAVE BEEN UPDATED FOR A CAL
L TO PROCEDURE PROCALLED.01

EXIT 1

PROCEDURE UPDATEALTSET 1
ASSERT 0<CURRENTPROC<20011
ASSERT 0<PROC17<CURRENTPROC+515PROC17>CURRENTPROC+61510001
II=PROC17>CURRENTPROC+5
ASSERT PROC17>CURRENTPROC+5151011 = ALTSET.01
ASSERT PROC17>CURRENTPROC+5151011 PROC17>CURRENTPROC+6111
ASSERT ALTSET=ALTSET.01
ASSERT AT=AT.01
ASSERT PROC=PROC.01
ASSERT CURRENTPROC=CURRENTPROC.01
WHILE 1<PROC17<CURRENTPROC+61 DO
IF ALTSET.11=AT
THEN RETURN 1
I = I + 11
ELIH
I
PROC17>CURRENTPROC+61 = PROC17>CURRENTPROC+6111
ALTSET=PROC17>CURRENTPROC+6111
ASSERT THER EXISTS A UNIQUE VALUE OF $5 SUCH THAT PROC17>CURRENTPROC+51515
PROC17>CURRENTPROC+61 AND ALTSET.11=AT.01
EXIT 1

PROCEDURE VCGEN 1
ASSERT LINEP=0 AND LINE.11=F1
P1=P1
ASSERT $P=DEFINEDPROCEDURESETPT+11
ASSERT WRITEFILE(:WTHD).01 WRITEFILE(:WTHD.0) U VERIFICATION CONDITIONS FOR ALL PRO
CEDURES IN DEFINEDPROCEDURESETPT FROM P TO P+11
ASSERT LINEP=0 AND LINE.11=F1
WHILE PS=DEFINEDPROCEDURESETPT DO
CURRENTPROC=P1
ASSERT $P=DEFINEDPROCEDURESETPT
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U VERIFICATION CONDITIONS FOR ALL PRO
CEDURES IN DEFINEDPROCEDURESETPT FROM 0 THROUGH P+11
ASSERT LINEP=0 AND LINE.11=F1
WHILE DEFINEDPROCEDURESETPT(P)PROC17>CURRENTPROC DO
CURRENTPROC=CURRENTPROC+11
ELIH
BIAS=PROC17>CURRENTPROC+11
ENTER PROCVCGEN 1
P1=P1

ELIH 1
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U VERIFICATION CONDITIONS FOR INPUTST
RING1
EXIT 1

PROCEDURE VERIFY 1
$ TOP LEVEL OF THE VERIFICATION SYSTEM. U
ASSERT HEADFILE(:RMD).01=INPUTSTRING1
ENTER SETUP1
ENTER PARSE 1
ENTER VCGEN 1
ASSERT INPUTSTRING IS A LEGAL NUCLEUS PROGRAM1
ASSERT WRITEFILE(:WTHD).01=LISTING U VERIFICATION CONDITIONS FOR INPUTSTRING1
EXIT 1

PROCEDURE WRITEASRT51
ASSERT ALTLISTALTNUM+COLONALTNUM+PATHPATHPT)U
ASSERT NSTEM=STEPS SINCE :STEP PRINT1
ASSERT LINEPT=O ^ LINE[O]="FI"
ASSERT NOT IN ENTER STATEMENT
IF ASRTLOC(PATH=PATHPT)+BIAS1=0
THEN WRITE ASRTTRUE 1
ELSE W:=ASRTLOC[ASRTLOC(PATH=PATHPT)+BIAS1]
ASHTFLAG:=0
ASSERT ASRTLOC(PATH=PATHPT)+BIAS1=0
ASSERT ASRTLOC(PATH=PATHPT)+BIAS1=SWASRTLOC1[ASRTLOC(PATH=PATHPT)+BIAS1]
I=1
ASSERT ASHTFLAG=0 ^ NOT IN ENTER STATEMENT
ASSERT ASRTLOC=ASRTLOC.01
ASSERT PATH=PATHPT.01
ASSERT ASRTLOC=ASRTLOC.01
ASSERT WRITEFILE(:WTM0)=WRITEFILE(:WTMD.0) U SUBSCRIPTED ASSERTIONS FOR PATH/PATHPT
FOM ASRTLOC[ASRTLOC(PATH=PATHPT)+BIAS1] THROUGH W-1
ASSERT LINEPT=0 ^ LINE[O]="FI"
ASSERT ALTL[ALTNM=COLONALTNUM=PATH=PATHPT]1
ASSERT NSTEM=STEPS SINCE :STEP PRINT1
DO
WHILE ASRTLOC[ASRTLOC(PATH=PATHPT)+BIAS1]=1
ASRTFRONT:=ASRTS[2*W]
ASRTSCANPOINTER:=ASRTFRONT+1
ASRTBACK:=ASHIS[2*W+1]
ENTER WRITEEXTASRT 1
W:=W+1
ELIM 1
FI 1
ASSERT WRITEFILE(:WTMD.0)=WRITEFILE(:WTM0) U SUBSCRIPTED ASSERTIONS FOR PATH.01 PATHPT.01
ASSERT LINEPT=0 ^ LINE[O]="FI"
EXIT 1
PROCEDURE WRITEEXTASRT 1
ASSERT LINE[O]="FI"
ASSERT ALTL[ALTNM=COLONALTNUM=PATH=PATHPT]1
ASSERT ASRTFRONT=0 IF NOT IN ENTER STATEMENT: =1 IF IN INITIAL ASSERTION OF ENTE
R STATEMENT, AND =2 IF IN FINAL ASSERTION OF ENTER STATEMENT
ASSERT ASRTFRONT=ASRTBACK1
ASSERT NSTEM=STEPS SINCE :STEP PRINT1
TOKEI=O1
ASSERT ASHTFLAG =0 IF NOT IN ENTER STATEMENT: =1 IF IN INITIAL ASSERTION OF ENTE
R STATEMENT, AND =2 IF IN FINAL ASSERTION OF ENTER STATEMENT
ASSERT ASRTBACK=ASRTBACK.01
ASSERT ASRTFRONT:=ASRTFRONT+ASRTBACK+1
ASSERT WRITEFILE(:WTMD.0) U WRITEFILE(:WTMD.0) U LINE[O]1,...LINE[LINEPT]=WRITEFILE(:WTMD.0) U LINE[O]1
====LINE.0 LINEPT=0) U SUBSCRIPTED ASSERTION
CHARLIST[ASRTFRONT.0]...CHARLIST[ASRTFRONT]1
ASSERT BEFORETOKEN = PREVIOUS VALUE OF TOKEN
ASSERT LINE[O]="FI"
ASSERT ALTL[ALTNM=COLONALTNUM=PATH=PATHPT]1
ASSERT PATH=PATHPT.01
ASSERT ASRTFRONT=PATHPT.01
ASSERT NSTEM=STEPS SINCE :STEP PRINT1
WHILE ASRTFRONT=ASRTBACK DO
BEFORETOKEN:=TOKEN1
ENTER ASRTSCAN 1
LINEFRONT:=ASRTFRONT1
LINEBACK:=ASRTSCANNPOINTER-1
ENTER BUILDLINE 1
IF TOKEN=67 $103
THEN IF BEFORETOKEN=39 $IS
ASSET ALTIST (ALTNUM+COLONALTNUM+PATH+PATHPT)
ASSET SIMT=DESCLOC(PATHPATHPT)+BIAS
ASSET STATEMENT(SIMT)=9
ASSET NSTEP=STEPS SINCE :STEP PRINT
ASSET ID=STATEMENT(SIMT)+1
ASSET B=SOUNDFUNCTION(ID)
LINEFROMT=671
LINEBACK=781
ENTER :BUILDFROMPRESET
ENTER PRINTFROMSUB1
LINEFROMT=621
LINEBACK=351
ENTER :BUILDFROMPRESET
ENTER PRINTFROMSUB1
LINEFROMT=1051
LINEBACK=1221
ENTER :BUILDFROMPRESET
ENTER PRINTFROMSUB1
LINEFROMT=1371
LINEBACK=1021
ENTER :BUILDFROMPRESET
IF B#0
THEN INTVAL=B1
ELSE INTVAL=B0
FI
ENTER INPRINT
LINEFROMT=521
LINEBACK=351
ENTER :BUILDFROMPRESET
ENTER PRINTFROMSUB1
LINEFROMT=1371
LINEBACK=1021
ENTER :BUILDFROMPRESET
IF B#0
THEN LINEFROMT=1431
LINEBACK=1511
ENTER :BUILDFROMPRESET
INTVAL=B1
ENTER INPRINT
LINEFROMT=521
LINEBACK=351
ENTER :BUILDFROMPRESET
ENTER PRINTFROMSUB1
LINEFROMT=1001
LINEBACK=1331
ENTER :BUILDFROMPRESET
ENTER PRINTFROMSUB1
LINEFROMT=1341
LINEBACK=1071
ENTER :BUILDFROMPRESET
FI
ENTER PRINTLINE
ASSET LINEPT=0 ^ LINE10==F1
ASSET WITEFILE (:WTHD)=WITEFILE (:WTHD,0) U READLINE?
EXIT 1

PROCEDURE WRITEREADLINE1
ASSET LINEPT=0 ^ LINE10==F1
ASSET ALTIST (ALTNUM+COLONALTNUM+PATH+PATHPT)
ASSET STATEMENT(SIMT)=9
ASSET NSTEP=STEPS SINCE :STEP PRINT
ASSERT ID=STATEMENT(SMT=11)
ASSERT P=BOUND_FUNCTION(ID)
LINEPONT:=751
LINEBACK:=791
ENTER =UVDFFROMRESET
ENTER INPUT
CHAR:=**1
ENTER INSERTCHAR
LINEPONT:=751
LINEBACK:=781
ENTER =UVDFFROMRESET
ENTER PRINTROMSUB1
LINEPONT:=01
LINEBACK:=11
ENTER =UVDFFROMRESET
ENTER PRINTLINE
ASSERT LINEP=0 & LINE[0]=**1
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U READLINE31
EXIT 1

PROCEDURE WRITENPTLINE
ASSERT ALIST(ALTNUM+COLONALTNUM+PATH+PATHPT)1
ASSERT LINEPONT=0 & LINE[0]=**1
LINEBACK:=271
LINEPONT:=271
ENTER =UVDFFROMRESET
CHAR:=**1
ENTER INSERTCHAR
ENTER INPUT
LINEPONT:=01
LINEBACK:=211
ENTER =UVDFFROMRESET
LINEPONT:=211
ENTER INSERTCHAR
INTVAL:=PATH(PATH+PT+11)
ENTER INPUT
LINEPONT:=211
LINEBACK:=321
ENTER =UVDFFROMRESET
IF COLONALTNUM[5] > 0 THEN CHAR:=**1,
    ENTER INSERTCHAR
    INTVAL:=COLONALTNUM[5]
    ENTER INPUT
FIN
LINEPONT:=01
LINEBACK:=211
ENTER =UVDFFROMRESET
ENTER PRINTLINE
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U RTNPTLINE
ASSERT LINEP=0 & LINE[0]=**1
EXIT 1

PROCEDURE WRITESTEPLINE
ASSERT LINEP:=0 & LINE[0]=**1
ASSERT NSTEP=STEPS SINCE STEP PRINT1
ASSERT ALIST(ALTNUM+COLONALTNUM+PATH+PATHPT)!!
LINEFRONT: = 401
LINEBACK: = 451 $: STEPS
ENTER \#ULDFROMRESET;
INTVAL:= COLONALTNUM(10) + 11
ENTER INPRINT;
LINEFRON: = 591
LINEBACK: = 451 $: STEPS
ENTER \#ULDFROMRESET;
IF COLONALTNUM(10) > 0
THEN CHAR: = + 1
   ENTER INSERTCHAR;
   INTVAL:= COLONALTNUM(10)
   ENTER INPRINT;
FI;
CHAR: = + 1
ENTER INSERTCHAR;
INTVAL:= STEPS;
ENTER INPRINT;
ENTER \#PRINTLINE;
NSTEP:= 01
ASSERT WRITEFILE(\#WTHD) = WRITEFILE(\#WTHD, 0) \& STEPLINE11
ASSERT LINEPT=0 \& LINE(0)= + F1
ASSERT NSTEP = 01
EXIT 1

PROCEDURE WRITE\#STEPLINE21:
ASSE:STEPT=0 \& LINE(0)= + F1
ASSERT ALTLIST(ALTNUM+COLONALTNUM+PATH+PATHPT) 11
LINEFRONT:= 411
LINEBACK:= 51 $: STEPS
ENTER \#ULDFROMRESET;
INTVAL:= COLONALTNUM(10) + 11
ENTER INPRINT;
LINEFRONT:= 591
LINEBACK:= 51 $: STEPS
ENTER \#ULDFROMRESET;
INTVAL:= COLONALTNUM(10)
ENTER INPRINT;
LINEFRONT:= 591
LINEBACK:= 561 END OF $STEP LINES
ENTER \#ULDFROMRESET;
ID:= PROC? \& PROCalled11
ENTER PRINT11;
ENTER PRINTLINE;
ASSERT LINEPT=0 \& LINE(0)= + F1
ASSERT WRITEFILE(\#WTHD) = WRITEFILE(\#WTHD, 0) \& STEPLINE21
EXIT 1

PROCEDURE WRITE\#PRINTLINE11:
ASSE:STEPT=0 \& LINE(0)= + F1
ASSERT ALTLIST(ALTNUM+COLONALTNUM+PATH+PATHPT) 11
ASSERT STM=DESCLUCI\#PATH(PAT=PT + BIAS) 11
ASSERT STATEMENT(STM) = 911
ASSERT $STEP=STEPS SINCE $STEP PRINT1
ASSERT BOUNDFUNCTION(ID) 11
ASSERT ID=STATEMENT(STM+1) 11
ENTER \#PRINT10SUB11
LINEFRONT:= 1521
LINEBACK:= 1711
ENTER \#ULDFROMRESET;
ENTER PRINT\#WTHD\#SUB11
LINEFRONT:= 501
LINEBACK=821
ENTER BUILDFROMPRESET1
% ENTER PRINTLINE1
ASSEPT LINES=0 A LINE(0) E+F1
ASSEPT WRITEFILE(1:WTHD)+WRITEFILE(1:WTHD,0) U WRITELINE11
EXIT 1

PROCEDURE WRITEWRITELINE21
ASSEPT LINES=0 A LINE(0) E+F1
ASSEPT ALT#LIST(ALTNUM+COLONALTNUM+PATH+PATHPT1)
ASSEPT STATEMENT(STMT)=941
ASSEPT NSTEP=STEPS SINCE 1STEP PRINT1
ASSEPT ID=STATEMENT(STMT1)
ASSEPT B=BOUNDFUNCTION101
ENTER PRINTDOSUB1
LINEFRONT=1721
LINEBACK=1721
ENTER BUILDFROMPRESET1
ENTER PRINTATMODLBL1
LINEFRONT=1941
LINEBACK=2041
ENTER BUILDFROMPRESET1
IF BS120
THEN INTERVAL=81
ELSE INTERVAL=1201
FII
ENTER INPRINT1
LINEFRONT=2051
LINEBACK=2191
ENTER BUILDFROMPRESET1
ENTER PRINTATMODLBL1
LINEFRONT=2191
LINEBACK=2241
ENTER BUILDFROMPRESET1
ENTER PRINTDOSUB1
LINEFRONT=1041
LINEBACK=1071
ENTER BUILDFROMPRESET1
IF BS121
THEN LINEFRONT=2251
LINEBACK=2341
ENTER BUILDFROMPRESET1
INTERVAL=81
ENTER INPRINT1
LINEFRONT=2051
LINEBACK=2141
ENTER BUILDFROMPRESET1
ENTER PRINTATMODLBL1
LINEFRONT=2141
LINEBACK=2241
ENTER BUILDFROMPRESET1
LINEFRONT=2351
LINEBACK=2371
ENTER BUILDFROMPRESET1
FII
% ENTER PRINTLINE1
ASSEPT LINES=0 A LINE(0) E+F1
ASSEPT WRITEFILE(1:WTHD)+WRITEFILE(1:WTHD,0) U WRITELINE21
EXIT 1

PROCEDURE WRITEWRITELINE31
DECLARE LINEPT: INTEGER
DECLARE LINE:path
DECLARE ALTNUM, COLONALTNUM,path: integer
DECLARE SMT: integer
DECLARE STATEMENT(SMT): string
DECLARE BIAS: integer
DECLARE ID: integer
DECLARE BOUNDFUNCTION(ID): integer

ASSERT LINEPT=0 AND LINE(0)=+1
ASSERT ALTLIST(ALTNUM, COLONA LIST, path, P A THPT):
ASSERT SMT=DESCLOC(path, PATHPT) + BIAS
ASSERT STATEMENT(SMT)=0
ASSERT NSTEP=STEP SINCE :STEP PRINT1
ASSERT ID=STATEMENT(SMT+1)
ASSERT B=BOUNDFUNCTION(ID+1)
LINEFRONT:=1001
LINERACK:=1931
ENTER BUILDFROMPRESET
INTVAL:=COLONLTRIM(4)+1
ENTER INTPRINT
LINEFRONT:=2381
LINERACK:=2431
ENTER BUILDFROMPRESET
ENTER PRINT=TMOSUM
LINEFRONT:=801
LINERACK:=811
ENTER BUILDFROMPRESET
%% ENTER PRINTLINE
ASSERT LINEPT=0 AND LINE(0)=+1
ASSERT WRITEFILE(1, TMD)=WRITEFILE(1, TMD, 0) OR WRITELINE31
EXIT 1

START VERIFY
BIBLIOGRAPHY


[14] Wang, Y. Y., A Verification Condition Generator for Nucleus Programs, M. S. Thesis, University of Texas at Austin, [n.d.].