A VERIFIED PROGRAM VERIFIER

by

Larry Calvin Ragland

May 1973 TR-18

TR 73-18

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

CHAPTER I

INTRODUCTION AND RELATED WORK

I.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.

<u>Procedures</u>. 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.

<u>Bodies</u>. 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 semicolon. 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.

<u>Assignment</u>. leftside := expression.

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

<u>Go to</u>. GO TO identifier.

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

<u>If</u>. 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.

<u>While</u>. WHILE boolean expression DO body ELIHW.

The while statement is the usual loop control statement, again with "body" where a statement usually occurs.

<u>Case</u>. 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.

<u>Read</u>. 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 $1 \leq i \leq$ 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 $1 \leq i \leq$ 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

Operator	Priority	Operand Type
unary+,-	1	INTEGER
*,/,÷(modulo)	2	INTEGER
binary+,-	3	INTEGER
$<,\leq,>,\geq,=,\neq$	4	see below
$\neg$	5	BOOLEAN
$\wedge$	6	BOOLEAN
$\vee$	7	BOOLEAN

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;
ELIHW;
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 grammars" described by Woods [15] for dealing with natural languages. 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 <u>and</u> 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

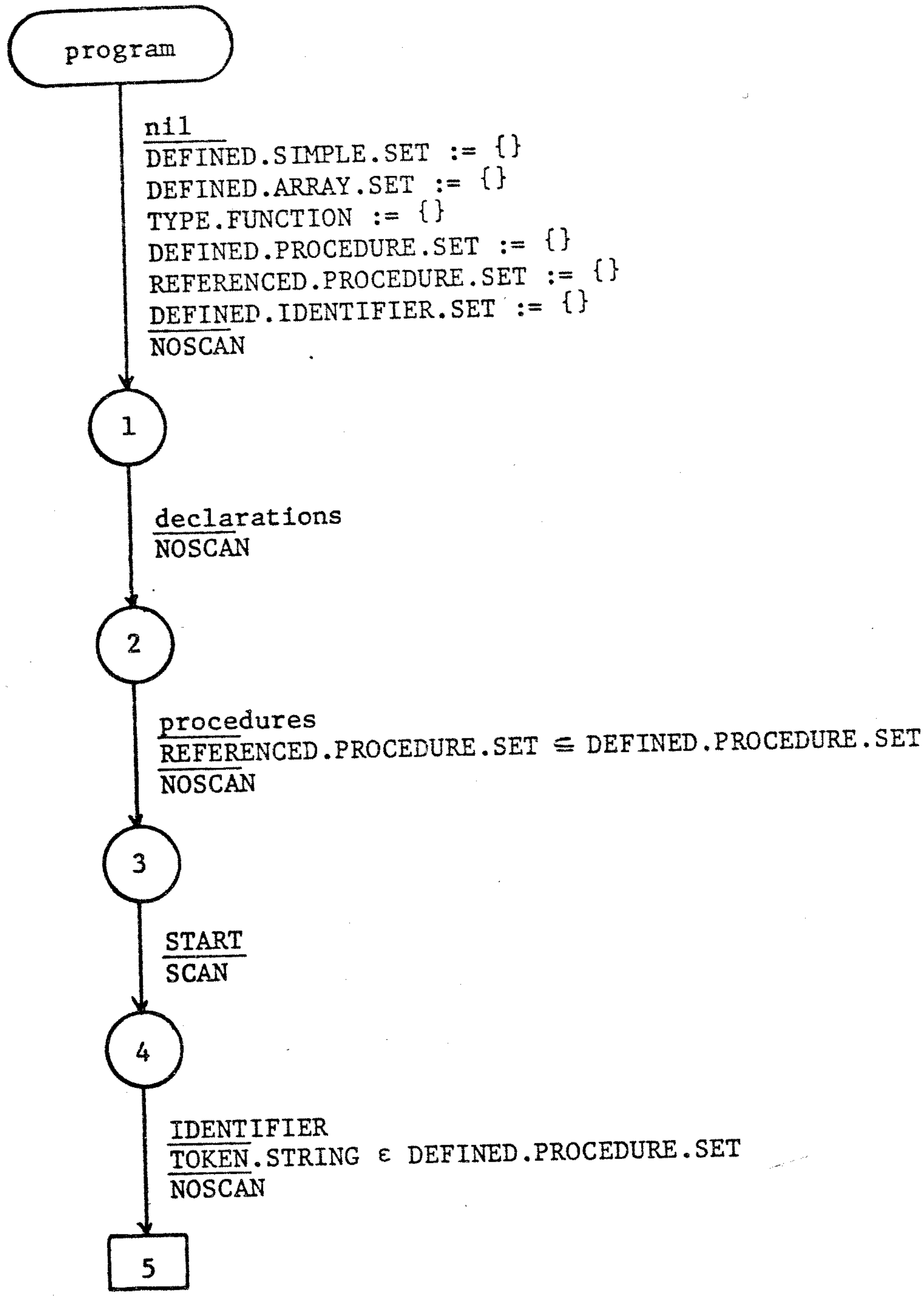

FIGURE II.1. The Opening Segment of the Nucleus Parsing Network labelled with a state name so this arc is placed on the return stack and operation proceeds to the state labelled "declarations". If a recognition state is then reached, the arc from state 1 to state 2, which is on top of the stack, will 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	Reduced Program
INTEGER I,N,SUM;	SIMPLE(I) SIMPLE(N) SIMPLE(SUM)
INTEGER ARRAY A[100];	ARRAY(A,100)
PROCEDURE ADD;	
SUM:=0;	ASSIGN(ADD:0,SUM,0)
I:=0;	ASSIGN(ADD:1,I,0)
WHILE I ≤ N DO	IF(ADD:2,(I) ≤ (N),3,7)
ENTER ABS;	ENTER(ADD:3,ABS)
SUM:=SUM+A[I];	ASSIGN(ADD:4,SUM,(SUM)+(A[I]))
I:=I+1;	ASSIGN(ADD:5,I,(I)+(1))
ELIHW;	JUMPTO(ADD:6,2)
EXIT;	EXIT(ADD:7)
	EXITPOINT(ADD)=7
PROCEDURE ABS;	

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

The Nucleus axioms are based directly on the concept of state vectors. A <u>state vector</u>, S, is a function from some <u>name space</u> NS into some <u>value space</u> VS. Each member (n,v) of S is a <u>cell</u>, 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 $S_0, S_1, S_2, \ldots$. 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

	E[0]	E[1]	E[2]	E[3]	E[4]	E[5]	E[6]	E[7]	E[8]	E[9]	E[10]	E[11]
:LOC	ADD:0	ADD:1	ADD:2	ADD:3	ABS:0	ABS:1	ABS:2	ADD:4	ADD:5	ADD:6	ADD:2	ADD:7
:LVL	-1	-1	-1	-1	0	0	0	-1	-1	-1	-1	-1
:RDHD	0	0	0	0	0	0	0	0	0	0	0	0
:WTHD	0	0	0	0	0	0	0	0	0	0	0	0
:RTNPT[0]	?	?	?	?	ADD:4	ADD:4	ADD:4	ADD:4	ADD:4	ADD:4	ADD:4	ADD:4
:RTNPT[1]	?	?	?	?	?	?	?	?	?	?	?	?
...												
:RTNPT[maxstacksize]	?	?	?	?	?	?	?	?	?	?	?	?
I	?	?	0	0	0	0	0	0	0	1	1	1
N	0	0	0	0	0	0	0	0	0	0	0	0
SUM	?	0	0	0	0	0	0	0	5	5	5	5
A[0]	-5	-5	-5	-5	-5	-5	5	5	5	5	5	5
A[1]	?	?	?	?	?	?	?	?	?	?	?	?
...												
A[100]	?	?	?	?	?	?	?	?	?	?	?	?

procedure (Axiom 17), :LVL = -1 (Axiom 18), :RDHD = 0 (Axiom 19), and :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](:LOC)=ADD:0 then by Axiom 48 we get E[1](:LOC)= ADD:1, E[1](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](:LOC)=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.

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 <u>informally</u> stated as, "<u>If</u> the initial state vector satisfies the initial assumption <u>and</u> if the program terminates normally, <u>then</u> the final state vector satisfies the desired result." Total correctness is the condition, "<u>If</u> the initial state vector satisfies the initial assumption, <u>then</u> the program terminates normally <u>and</u> the final state vector satisfies the desired result."

We will now define a series of terms which leads up to the <u>formal</u> 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 <u>execution procedure of E[i]</u> 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 <u>entry(i)</u> and <u>exit(i)</u> denote functions from one state vector index to another. They are defined so that the state vector E[entry(i)] is at the entry to the execution procedure of E[i] and if the procedure reaches its exit point, E[exit(i)] is the state vector at exit. For any given state vector sequence, we define

$$\text{entry}(i) = \begin{cases} 0 & \text{if } E[i](:LVL) = -1 \\ \max \{j \mid j \leq i \wedge \text{ENTER}(E[j-1](:LOC), \text{PNAME}(E[i](:LOC))) \\ \quad \wedge E[i](:LVL) = E[j](:LVL)\} & \text{otherwise} \end{cases}$$

$$\text{exit}(i) = \begin{cases} \min \{j \mid j \geq i \wedge \text{EXIT}(E[j](:LOC)) \wedge E[i](:LVL) = E[j](:LVL)\} \\ \quad \text{if the set is not empty} \\ \text{undefined} \quad \text{otherwise} \end{cases}$$

In the definition of 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[entry(i)] occurs before E[i] or is E[i] itself. The next term requires that E[entry(i)] is a state vector immediately following a call of the current execution procedure and the last term requires E[entry(i)] to be at the same level as E[i]. The "max" operator makes certain that E[entry(i)] is defined at the <u>most recent</u> entry to the current execution procedure at this level.

Consider now the definition of exit(i). The term "$j \geq i$" indicates that E[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 etry 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 $\{x \mid \mathrm{SIMPLE}(x) \vee \mathrm{ARRAY}(x,b)\} \cup \{:\mathrm{STEP}\}$. 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 = (X1,X2,...,Xj) then E[i](X) = (E[i](X1),E[i](X2),...,E[i](Xj)) where if Xk is array name A with bound b then E[i](Xk) = E[i](A) = E[i](A[0]),E[i](A[1]),...,E[i](A[b]). The value of E[i](X.0) is E[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[entry(i)](X),E[i](X)). For example, if B(X.0,X) is the predicate NUM $\leq$ NUM.0 where 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](NUM) $\leq$ E[entry(i)](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 H1(X.0,X),...,Hj(X.0,X), then R(X.0,X) is B(X.0,X) $\vee$ H1(X.0,X) $\vee$ ... $\vee$ Hj(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)) $\wedge$ 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,

$$E[0](A(X.0,X)) \rightarrow \exists n[NT(n) \wedge E[n](R(X.0,X))].$$

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.

instruction	successors
ASSIGN(P:Q1,N,V)	{P:Q1+1}
CASE(P:Q1,EXP,L)	{P:POINTLABELLEDWITH(W) | W ε CASELABELSET(P:Q1)} ∪ {P:L}
ENTER(P:Q1,C)	{P:Q1+1}
EXIT(P:Q1)	{ }
HALT(P:Q1)	{ }
IF(P:Q1,EXP,T,F)	{P:T,P:F}
JUMPTO(P:Q1,N)	{P:N}
READ(P:Q1,A)	{P:Q1+1}
WRITE(P:Q1,A)	{P:Q1+1}

TABLE III.1 Definition of Control Point Successors

Note the relationship between successive points and successive values of :LOC in the execution sequence of a procedure. IF E[i](:LOC) = P:Q1 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](:LOC) is a successor of P:Q1 (see Axioms 48-58). For the enter statement, the top of the return point stack in state vector E[i+1] is P:Q1+1 (Axiom 50) which means that the value of :LOC upon exit from the called procedure will be P:Q1+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:Q1. 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:Q1,...,P:Qr is a path iff P:Qi+1 is a successor of P:Qi for i = 1,...,r-1, P:Q1 and P:Qr are both tagged, and no other points in the sequence are tagged. A path will sometimes be denoted (Q1,...,Qr) 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 <u>properly tagged</u> 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 <u>infinite</u> set of executions to proofs about a <u>finite</u> 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

	Nucleus Program	Reduced Program
	INTEGER N,D,Q,R;	SIMPLE(N) SIMPLE(D) SIMPLE(Q) SIMPLE(R)
	PROCEDURE DIV;	
	ASSERT D.0 > 0;	
	ASSERT N.0 ≥ 0;	
(0)	Q:=0;	ASSIGN(DIV:0,Q,0)
	ASSERT N.0=Q*D.0+N;	
	ASSERT D=D.0;	
	ASSERT N ≥ 0;	
(1)	WHILE N ≥ D DO	IF(DIV:1,(N) ≥ (D),2,5)
(2)	N:=N-D;	ASSIGN(DIV:2,N,(N)-(D))
(3)	Q:=Q+1;	ASSIGN(DIV:3,Q,(Q)+(1))
(4)	ELIHW;	JUMPTO(DIV:4,1)
(5)	R:=N;	ASSIGN(DIV:5,R,N)
	ASSERT N.0=Q*D.0+R;	
	ASSERT 0 ≤ R < D.0;	
(6)	EXIT;	EXIT(DIV:6) EXITPOINT(DIV)=6
	START DIV	INITIALPROCEDURE=DIV

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 impliciation. Thus

$$\begin{array}{c} A \\ B \\ \hline C \\ D \end{array}$$

denotes $A \wedge B \rightarrow C \wedge D$.

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.0 > 0
0.2   N.0 ≥ 0
1.1   N.0=Q*D.0+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.0=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 alt(x,i). This function is called the <u>alteration function</u> 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 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.alt(N,3)" or since alt(N,3)=1 we may write "N.1". For an array A, the value of 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.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 alt(X,1) = 0 and for other paths the definition is 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.0 is used in a verification condition, it appears as just x (without the ".0" suffix).

The notation EXP.alt(X,i), where EXP is an expression, denotes the expression resulting from the substitution of x.alt(x,i) for each occurrence of any x ε X that appears in EXP. The value of E[i](EXP) is the value of EXP with E[i](x) substituted for each occurrence of x in 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 vcterms 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 cond(P:Qi,i) and vcterm(P:Qi,P:Q$\underline{\text{i+1}}$,i) and the alteration function alt(x,i) when P:Qi and P:Q$\underline{\text{i+1}}$ are points i and i+1 along a path. Recall that the underline 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 ASSIGN(P:Qi,N,V) and SIMPLE(N) then cond(P:Qi,i) is

TRUE

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

```
V.alt(X,i)≠U
N.alt(N,i+1)=V.alt(X,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 ε {N,:LOC,:STEP}
     then alt(y,i)+1
     else alt(y,i).
```

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 vcterm, "V.alt(X,i)≠U", states that the right side expression, V, is <u>not</u> 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 ≤ S.alt(X,i) ≤ 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.alt(X,i)≠0 appears in the verification condition.

The second line of vcterm, "N.alt(N,i+1) = V.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 :LOC and :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, :LOC, and :STEP.

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

```
0 ≤ S.4 ≤ 10
A.3[S.4]≠0
LEFT.3 = LEFT.2/A.3[S.4]
:LOC.6 = :LOC.5 +1
:STEP.6 = :STEP.5 +1
```

Definition 3.5.1b. assign (with array left side)

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

TRUE

and vcterm(P:Qi,P:Q<u>i+1</u>,i) is

```
EXP.alt(X,i)≠U
0 ≤ EXP.alt(X,i) ≤ BOUND(A)
```

```
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 ε {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 3.5.1a. 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[S]:=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 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) ε CASELABELSET(P:Qi)
:STEP.alt(:STEP,i+1)=:STEP.alt(:STEP,i) +1
```

and alt(y,i+1) =

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

CASE(P:Qi,EXP,L) indicates a case statement at location P:Qi with case expression EXP. The value L indicates the point to which control passes if the value of EXP does not match a case label. The assumption that Qi+1=POINTLABELLEDWITH(P:Qi:EXP.alt(X,i)) is an assumption about the path for which the verification condition terms are to be defined. The first term of vcterm 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:Qi. This is a result of the assumption that execution follows the current path from P:Qi to P:Qi+1. If the case expression is Y+Z, the set of labels for point P:Qi+1 is {5,10}, alt(Y,i)=2 and alt(Z,i)=3, then this term appears in a verification condition as "Y.2 + Z.3 ε {5,10}". The last term of 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 ε {:LOC,: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
```

```
:STEP.alt(:STEP,i')=:STEP.alt(:STEP,i) +1
tag[C:EXITPOINT(C)](X,alt(X,i'),X.alt(X,i"))
:LVL.alt(:LVL,i") ≥ 0
:LOC.alt(:LOC,i+1)=:RTNPT.alt(:RTNPT,i")[:LVL.alt(:LVL,i")]
:LVL.alt(:LVL,i+1)=:LVL.alt(:LVL,i") -1
:STEP.alt(:STEP,i+1)=:STEP.alt(:STEP,i") +1
```

and alt(y,i')=

```
if y ε {:LVL,:RTNPT,:LOC,:STEP}
      then alt(y,i)+1
      else alt(y,i)
```

and alt(y,i")=

```
if y ε alterableset(C)
      then alt(y,i')+1
      else alt(y,i')
```

and alt(y,i+1)=

```
if y ε {:LOC,:LVL,:STEP}
      then alt(y,i")+1
      else alt(y,i").
```

ENTER(P:Qi,C) indicates a call of procedure C at location P:Qi. The values i' and i" in the cond and vcterm refer to points which fall between P:Qi and P:Q<u>i+1</u> during execution. The notation y.alt(y,i') refers to the value of y after entry to the called procedure and y.alt(y,i") refers to the value of y 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 satck 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 "alterableset(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 <u>program</u> variables are assignment and read statements.

Definition 3.5.4a if (expression is true)

If IF(P:Qi,EXP,T,F) and Q<u>i+1</u>=T then cond(P:Qi,i) is

TRUE

and vcterm(P:Qi,P:Q<u>i+1</u>,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 ε {: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, Q<u>i+1</u>=T, is an assumption about the path.

Definition 3.5.4b if (expression is false)

If IF(P:Qi,EXP,T,F) and Q<u>i+1</u>=F then cond(P:Qi,i) is

TRUE

and vcterm(P:Qi,P:Q<u>i+1</u>,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 alt(y,i+1) =

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

Definition 3.5.5 jumpto

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

```
        TRUE
```

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

```
        :LOC.alt(:LOC,i+1)=P:N
        :STEP.alt(:STEP,i+1)=:STEP.alt(:STEP,i) +1
```

```
and alt(y,i+1) =

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

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

Definition 3.5.6 read

If READ(P:Qi,A) then cond(P:Qi,i) is

```
        TRUE
```

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

```
        ¬:REOF(:RDHD,alt(:RDHD,i) +1)→
             [A.alt(A,i+1)[0]="F"
             ∧(1 ≤ $ ≤ min(readsize,BOUND(A))→
                     A.alt(A,i+1)[$]=:RDFL(:RDHD.alt(:RDHD,i) +1,$))
             ∧(readsize+1 ≤ $ ≤ BOUND(A)→
                     A.alt(A,i+1)[$]=A.alt(A,i)[$])]
          :REOF(:RDHD.alt(:RDHD,i) +1)→
             [A.alt(A,i+1)[0]="T"
             ∧(1 ≤ $ ≤ BOUND(A)→
                     A.alt(A,i+1)[$]=A.alt(A,i)[$])]
          :RDHD.alt(:RDHD,i+1)=:RDHD.alt(:RDHD,i) +1
          :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 ε {A,:RDHD,:LOC,:STEP}
              then alt(y,i)+1
              else alt(y,i).
```

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 READ A; at location P:Qi with BOUND(A) = 90, readsize = 80, alt(A,i) = 2, and alt(:RDHD,i) = 3. Then the first two terms appear in a verification condition as

```
¬:REOF(:RDHD.3 +1)→
     [A.3[0]="F"
     ∧(1 ≤ $ ≤ 80 → A.3[$]=:RDFL(:RDHD.3 +1,$))
     ∧(81 ≤ $ ≤ 90 → A.3[$]=A.2[$])]
 :REOF(:RDHD.3 +1)→
     [A.3[0]="T"
     ∧(1 ≤ $ ≤ 90 → A.3[$] = A.2[$])]
```

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

Definition 3.5.7 write

If WRITE(P:Qi,A) then cond(P:Qi,i) is

TRUE

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

```
A.alt(A,i)[0]≠"T"→
     [¬:WEOF(:WTHD.alt(:WTHD,i) +1)
     ∧(1 ≤ $ ≤ min(writesize,BOUND(A))→
            :WTFL(:WTHD.alt(:WTHD,i) +1,$)=A.alt(A,i)[$])
     ∧(BOUND(A)+1 ≤ $ ≤ writesize→
            :WTFL(:WTHD.alt(:WTHD,i) +1,$)=" "]
A.alt(A,i)[0]="T"→
      :WEOF(:WTHD.alt(:WTHD,i) +1)
:WTHD.alt(:WTHD,i+1)=:WTHD.alt(:WTHD,i) +1
: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 ε {:WTHD,:LOC,:STEP}
     then alt(y,i)+1
     else alt(y,i).
```

WRITE(P:Qi,A) indicates a write statement at location P:Qi 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 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 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,

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

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 → 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 i 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 <u>then</u> 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:Q1,...,P:Qr with corresponding state vector sequence E[q1],...,E[qr]. For 1 ≤ i ≤ r-1, if

VC:

```
1          tag[P:0](X.0,X.0)
2          B(X.0,X.alt(X,1),X.alt(X,1'),X.alt(X,1"),...,X.alt(X,i))
--------------------------
3  <PRV>   cond(P:Qi,i)
4          vcterm(P:Qi,P:Qi+1,i)
__________________________
5          D(X.0,X.alt(X,1),X.alt(X,1'),X.alt(X,1"),...,X.alt(X,i+1))
```

then

SV:

```
1          NT(n)
2          n ≥ qr
3          tag[P:0](E[entry(q1)](X),E[entry(q1)](X))
4          B (E[entry(q1)](X),E[q1](X),E[q1'](X),E[q1"](X),...,E[qi](X))
5          c ε calls(qi,qi+1) ∧ tag[E[c](:LOC)](E[c](X),E[c](X)) →
               tag[E[exit(c)](:LOC)](E[c](X),E[exit(c)](X))
__________________________
6          D(E[entry(q1)](X),E[q1](X),E[q1'](X),E[q1"](X),...,E[qi+1](X))
7          c ε calls(qi,qi+1) → tag[E[c](: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 → SV. Condition VC consists of two implications, VC1 ∧ VC2 → VC3 and VC1 ∧ VC2 ∧ VC3 ∧ VC4 → VC5. The first implication results from the "<PRV>" 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.alt(X,k) for which specific values may be substituted. We now substitute values for symbols according to the pairing in Table III.2.

Symbol	Value
X.0	E[entry(q1)](X)
X.alt(X,k)	E[qk](X)
X.alt(X,k')	E[qk'](X)
X.alt(X,k")	E[qk"](X)

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

Condition VC is now in the following form.

```
VC:
1          tag[P:0](E[entry(q1)](X),E[entry(q1)](X))
2          B(E[entry(q1)](X),E[q1](X),E[q1'](X),E[q1"](X),...,E[qi](X))
------------------------
3  <PRV>   cond(P:Qi,i)
4          vcterm(P:Qi,P:Qi+1,i)
------------------------
5          D(E[entry(q1)](X),E[q1](X),E[q1'](X),E[q1"](X),...,E[qi+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) SV1 $\wedge$ SV2 $\wedge$ SV5 $\wedge$ VC3 $\rightarrow$ VC4 and (2) VC3 $\wedge$ VC4 $\rightarrow$ 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') SV1 $\wedge$ SV2 $\wedge$ SV3 $\wedge$ SV4 $\wedge$ SV5 $\wedge$ VC3 $\rightarrow$ VC1 $\wedge$ VC2 $\wedge$ VC3 $\wedge$ VC4. Since SV3 $\wedge$ SV4 $\equiv$ VC1 $\wedge$ VC2 and VC1 $\wedge$ VC2 $\rightarrow$ VC3 then (1') becomes (1") SV1 $\wedge$ SV2 $\wedge$ SV3 $\wedge$ SV4 $\wedge$ SV5 $\rightarrow$ VC1 $\wedge$ VC2 $\wedge$ VC3 $\wedge$ VC4.

Since VC5 $\equiv$ SV6 then (2) becomes (2') VC3 $\wedge$ VC4 $\wedge$ VC5 $\rightarrow$ SV6 $\wedge$ SV7 which implies that (2") VC1 $\wedge$ VC2 $\wedge$ VC3 $\wedge$ VC4 $\wedge$ VC5 $\rightarrow$ SV6 $\wedge$ 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 cond and vcterm are defined in a way such that (1) SV1 $\wedge$ SV2 $\wedge$ SV5 $\wedge$ VC3 $\rightarrow$ VC4 and (2) VC3 $\wedge$ VC4 $\rightarrow$ SV7. For all except the enter statement, these conditions can be reduced even further. For these statements, calls(qi, qi+1) is empty and the cond term is "TRUE", therefore SV5, SV7, and VC3 are satisfied. Condition (2) then is also satisfied and condition (1) is reduced to SV1 $\wedge$ SV2 $\rightarrow$ VC4.

Case 1a. ASSIGN(P:Qi,N,V) and SIMPLE(N).

We must show SV1 $\wedge$ SV2 $\rightarrow$ VC4, that is, we must show NT(n) $\wedge$ n $\geq$ qr $\rightarrow$ vcterm(P:Qi,P:Qi+1,i). The vcterm with substitution of values for symbols as specified in Table III.2 is

1 E[qi](V)≠U
2 E[qi+1](N)=E[qi](V)
3 E[qi+1](:LOC)=E[qi](:LOC) +1
4 E[qi+1](:STEP)=E[qi](:STEP) +1

The conjunction NT(n) $\wedge$ 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 qi and qi+1. The first line of Axiom 48 and normal termination imply NAME(N,qi)≠U $\wedge$ E[qi](V)≠U. This implies line one of vcterm. The remainder of Axiom 48 parallels the second and third lines of vcterm. The last line of 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](EXP)≠U
2    0 ≤ E[qi](EXP) ≤ BOUND(A)
3    E[qi](V)≠U
4    E[qi+1](A[$])= IF $=E[qi](EXP)
                       THEN E[qi](V)
                       ELSE E[qi](A[$])
5    E[qi+1](:LOC)=E[qi](:LOC) +1
6    E[qi+1](:STEP)=E[qi](:STEP) +1
```

From Axiom 48 and the normal termination assumption, we get NAME(A[EXP],qi)≠U ∧ E[qi](V)≠U. ARRAY(A,B) and Axiom 11 imply that NAME(A[EXP],qi)≠U only if E[qi](EXP)≠U and 0 ≤ E[qi](EXP) ≤ 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](EXP)≠U
2    E[qi+1](:LOC)=P: Qi+1
3    E[qi](EXP) ε CASELABELSET(P:Qi)
4    E[qi+1](:STEP)=E[qi](:STEP) +1
```

Normal termination and line one of Axiom 49 imply E[qi](EXP)≠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=POINTLABELLEDWITH(P:Qi: E[qi](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

```
1    E[qi](EXP)≠U
2    E[qi+1](:LOC)=P:Qi+1
3    E[qi](EXP) ∉ CASELABELSET(P:Qi)
4    E[qi+1](:STEP)=E[qi](:STEP) +1
```

The argument presented for case 2a holds except the assumption that Qi+1=L implies a change in the third vcterm line.

Case 3. ENTER(P:Qi,C)

For the enter statement, we must prove SV1 ∧ SV2 ∧ SV5 ∧ VC3 → VC4 and VC3 ∧ VC4 → SV7. 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:Qi, and the initial assertion of the called procedure, implies vcterm(P:Qi,P:Qi+1,i) with substitution as defined earlier. The cond term with substitution is

```
1    [E[qi](:LVL) < maxstacksize
2    ∧E[qi'](:LVL)=E[qi](:LVL) +1
3    ∧E[qi'](:RTNPT[$])= IF $=E[qi](:LVL) +1
                            THEN E[qi](:LOC) +1
                            ELSE E[qi](:RTNPT[$])
4    ∧E[qi'](:LOC)=C:0
5    ∧E[qi'](:STEP)=E[qi](:STEP) +1
6    →tag[C:0](E[qi'](X),E[qi'](X))]
```

and the vcterm with substitution is

```
1    E[qi](:LVL) < maxstacksize
2    E[qi'](:LVL)=E[qi](:LVL) +1
3    E[qi'](:RTNPT[$])= IF $=E[qi](:LVL) +1
                           THEN E[qi](:LOC) +1
                           ELSE E[qi](:RTNPT[$])
4    E[qi'](:LOC)=C:0
5    E[qi'](:STEP)=E[qi](:STEP) +1
6    tag[C:EXITPOINT(C)](E[qi'](X),E[qi"](X))
7    E[qi"](:LVL) ≥ 0
8    E[qi+1](:LOC)=E[qi"](:RTNPT[E[qi"](:LVL)])
9    E[qi+1](:LVL)=E[qi"](:LVL) -1
10   E[qi+1](:STEP)=E[qi"](:STEP) +1
```

The first line of Axiom 50 together with the assumption of normal termination implies NAME(":RTNPT[:LVL+1]",i)$\neq$U and by Axioms 9 and 11 this implies E[qi](:LVL) < 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[qi'] $\geq$ 0 which is line seven of vcterm. The remainder of Axiom 51 implies lines eight and nine.

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

$\underline{\text{Case 4a}}$. IF(P:Qi,EXP,T,F) and Q$\underline{\text{i+1}}$=T

The vcterm with substitution is

1 E[qi](EXP)$\neq$U
2 E[qi](EXP)
3 E[q$\underline{\text{i+1}}$](:LOC)=P:Q$\underline{\text{i+1}}$
4 E[$\underline{\text{qi+1}}$](:STEP)=E[$\underline{\text{qi}}$](:STEP) +1

Normal termination and line one of Axiom 53 imply line one of vcterm. The remainder of Axiom 53 with the assumption that Q$\underline{\text{i+1}}$=T implies line two of vcterm. Line three comes from the assumption.

$\underline{\text{Case 4b}}$. IF(P:Qi,EXP,T,F) and Q$\underline{\text{i+1}}$=F

The vcterm with substitution is

```
1     E[qi](EXP)≠U
2     ¬E[qi](EXP)
3     E[qi+1](:LOC)=P:Qi+1
4     E[qi+1](:STEP)=E[qi](:STEP) +1
```

Same argument as for case 4a.

<u>Case 5</u>. JUMPTO(P:Qi,N)

The vcterm with substitution is

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

Line one is a direct result of Axiom 54.

<u>Case 6</u>. READ(P:Qi,A)

The vcterm with substitution is

```
1     ¬:REOF(E[qi](:RDHD) +1)→
           [E[qi+1](A[0])="F"
           ∧(1 ≤ $ ≤ min(readsize,BOUND(A))→
                   E[qi+1](A[$])=:RDFL(E[qi](:RDHD) +1,$)
           ∧(readsize+1 ≤ $ ≤ BOUND(A)→
                   E[qi+1](A[$])=E[qi](A[$])]
2     :REOF(E[qi](:RDHD) +1)→
           [E[qi+1](A[0])="T"
           ∧(1 ≤ $ ≤ BOUND(A)→
                   E[qi+1](A[$])=E[qi](A[$])]
3     E[qi+1](:RDHD)=E[qi](:RDHD) +1
4     E[qi+1](:LOC)=E[qi](:LOC) +1
5     E[qi+1](:STEP)=E[qi](:STEP) +1
```

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

<u>Case 7</u>. WRITE(P:Qi,A)

The vcterm with substitution is

```
1     E[qi](A[0])≠"T"→
           [¬:WEOF(E[qi](:WTHD)+1)
           ∧(1 ≤ $ ≤ min(writesize,BOUND(A))→
                   :WTFL(E[qi](:WTHD)+1,$)=E[qi](A[$])
           ∧(BOUND(A)+1 ≤ $ ≤ writesize→
                   :WTFL(E[qi](:WTHD)+1,$)=" "]
```

```
2    E[qi](A[0])="T"→
          :WEOF(E[qi](:WTHD)+1)
3    E[qi+1](:WTHD)=E[qi](:WTHD)+1
4    E[qi+1](:LOC)=E[qi](:LOC)+1
5    E[qi+1](:STEP)=E[qi](:STEP)+1
```

IF E[qi](A[0])≠"T" then Axiom 58 implies the vcterm. If E[qi](A[0])="T" then Axiom 57 implies the 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 <u>verification condition</u>

```
          tag[P:0](X.0,X.0)
          tag[P:Q1](X.0,X.alt(X,1))
------------------------
    <PRV> cond(P:Q1,1)
          vcterm(P:Q1,P:Q2,1)
          ...
    <PRV> cond(P:Qr-1,r-1)
          vcterm(P:Qr-1,P:Qr,r-1)
------------------------
          tag[P:Qr](X.0,X.alt(X,r))
```

is satisfied then

```
          NT(n)
          n ≥ qr
          tag[P:0](E[entry(q1)](X),E[entry(q1)](X))
          tag[P:Q1](E[entry(q1)](X),E[q1](X))
```

```
c ε calls(q1,qr) ∧ tag[E[c](:LOC)](E[c](X),E[c](X)) →
    tag[E[exit(c)](:LOC)](E[c](X),E[exit(c)](X))
______________________________________________
tag[P:Qr](E[entry(q1)](X),E[qr](X))
c ε calls(q1,qr) → 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 $1 \leq i \leq r$, we define

INTi(X.0,X.alt(X,1),X.alt(X,1'),X.alt(X,1"),...,X.alt(X,i)) to be

```
tag[P:Q1](X.0,X.alt(X,1))
vcterm(P:Q1,P:Q2,1)
...
vcterm(P:Qi-1,P:Qi,i-1)
```

Because the verification condition is satisfied, it follows from the definition of INTi that for $1 \leq i \leq r-1$,

```
tag[P:0](X.0,X.0)
INTi(X.0,X.alt(X,1),X.alt(X,1'),X.alt(X,1"),...,X.alt(X,i))
________________________
```

$$\begin{array}{ll}
\langle PRV\rangle & cond(P{:}Qi,i) \\
 & vcterm(P{:}Qi,P{:}Q\underline{i+1},i) \\
\hline
 & INT\underline{i+1}(X.0,X.alt(X,1),X.alt(X,1'),X.alt(X,1''),\ldots,X.alt(X,i+1))
\end{array}$$

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

$$\begin{array}{l}
NT(n) \\
n \geq qr \\
tag[P{:}0](E[entry(q1)](X),E[entry(q1)](X)) \\
INTi(E[entry(q1)](X),E[q1](X),E[q1'](X),E[q1''](X),\ldots,E[qi](X)) \\
c \in calls(qi,q\underline{i+1}) \wedge tag[E[c](:LOC)](E[c](X),E[c](X)) \rightarrow \\
\quad tag[E[exit(c)](:LOC)](E[c](X),E[exit(c)](X)) \\
\hline
INT\underline{i+1}(E[entry(q1)](X),E[q1](X),E[q1'](X),E[q1''](X),\ldots, \\
\quad E[q\underline{i+1}](X)) \\
c \in calls(qi,q\underline{i+1}) \rightarrow tag[E[c](:LOC)](E[c](X),E[c](X))
\end{array}$$

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

$$\begin{array}{ll}
A: & NT(n) \\
 & n \geq qr \\
 & tag[P{:}0](E[entry(q1)](X),E[entry(q1)](X)) \\
 & tag[P{:}Q1](E[entry(q1)](X),E[q1](X)) \\
 & c \in calls(q1,qr) \wedge tag[E[c](:LOC)](E[c](X),E[c](X)) \rightarrow \\
 & \quad tag[E[exit(c)](:LOC)](E[c](X),E[exit(c)](X)) \\
\hline
 & INTr(E[entry(q1)](X),E[q1](X),E[q1'](X),E[q1''](X),\ldots,E[qr](X)) \\
 & c \in calls(q1,qr) \rightarrow tag[E[c](:LOC)](E[c](X),E[c](X))
\end{array}$$

which has been labelled A for later reference. The verification condition and the definition of INTr yield

$$\begin{array}{l}
tag[P{:}0](X.0,X.0) \\
INTr(X.0,X.alt(X,1),X.alt(X,1'),X.alt(X,1''),\ldots,X.alt(X,r)) \\
\hline
tag[P{:}Qr](X.0,X.alt(X,r))
\end{array}$$

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

$$\begin{array}{l}
tag[P{:}0](E[entry(q1)](X),E[entry(q1)](X)) \\
INTr(E[entry(q1)](X),E[q1](X),E[q1'](X),E[q1''](X),\ldots,E[qr](X)) \\
\hline
tag[P{:}Qr](E[entry(q1)](X),E[qr](X))
\end{array}$$

By applying this result to A above we get

```
          NT(n)
          n ≥ qr
          tag[P:0](E[entry(ql)](X),E[entry(ql)](X))
          tag[P:Ql](E[entry(ql)](X),E[ql](X))
          c ε calls(ql,qr) ∧ tag[E[c](:LOC)](E[c](X),E[c](X)) →
              tag[E[exit(c)](:LOC)](E[c](X),E[exit(c)](X))
-------------------------
          tag[P:Qr](E[entry(ql)](X),E[qr](X))
          c ε calls(ql,qr) → tag[E[c](: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:Ql,...,P:Qr with P:Ql=INITIALPROCEDURE:0 and with corresponding state vector sequence E[ql],...,E[qr].

If the <u>verification condition</u>

```
          tag[P:0](X.0,X.0)
          :LVL.0=-1
          :RDHD.0=0
          :WTHD.0=0
          :STEP.0=0
-------------------------
    <PRV> cond(P:Ql,1)
          vcterm(P:Ql,P:Q2,1)
          ...
    <PRV> cond(P:Qr-1,r-1)
          vcterm(P:Qr-1,P:Qr,r-1)
-------------------------
          tag[P:Qr](X.0,X.alt(X,r))
```

is satisfied, then

```
          NT(n)
          n ≥ qr
          tag[P:0](E[entry(ql)](X),E[entry(ql)](X))
          c ε calls(ql,qr) ∧ tag[E[c](:LOC)](E[c](X),E[exit(c)](X)) →
              tag[E[exit(c)](:LOC)](E[c](X),E[exit(c)](X))
-------------------------
```

```
tag[P:Qr](E[entry(ql)](X),E[qr](X))
c ε calls(ql,qr) → tag[E[c](: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:Ql then tag[P:Ql] 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],...,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],...,E[x]. Then tag[P:0](E[e](X),E[e](X)) → tag[P:EXITPOINT(P)] (E[e](X),E[x](X)).

Proof:

The sequence E[e],...,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 tag[P:0](E[e](X),E[e](X)) → 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],...,E[x] does contain procedure calls and E[x] corresponds to a halt or exit of procedure P. Then tag[P:0](E[e](X),E[e](X)) → tag[P:EXITPOINT(P)] (E[e](X),E[x](X)).

Proof:

Again the sequence E[e],...,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],...,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 tag[P:0](E[e](X),E[e](X)) → 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 tag[P:0](E[e](X),E[e](X)) → tag[E[x+1](: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 tag[P:0](E[e](X),E[e](X)) → tag[E[x+1](: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 $E[0](A(X.0,X)) \wedge NT(n) \rightarrow E[n](R(X.0,X))$, 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 $E[0](A(X.0,X)) \wedge NT(n)$ 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, $E[0](A(X.0,X)) \wedge NT(n) \rightarrow E[n](R(X.0,X))$. 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, $E[0](A(X.0,X)) \wedge NT(n) \rightarrow E[n](R(X.0,X))$. 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)) $\wedge$ NT(n) $\rightarrow$ 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 <u>assumed</u> absent when normal termination is assumed; however, if normal termination is to be demonstrated, then these conditions must be <u>proved</u> 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 ≤ EXP.alt(X,i) ≤ 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
∧:LVL.alt(:LVL,i')=:LVL.alt(:LVL,i)+1
∧:RTNPT.alt(:RTNPT,i')[:LVL.alt(:LVL,i)+1]=:LOC.alt(:LOC,i)+1
∧: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'))]

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

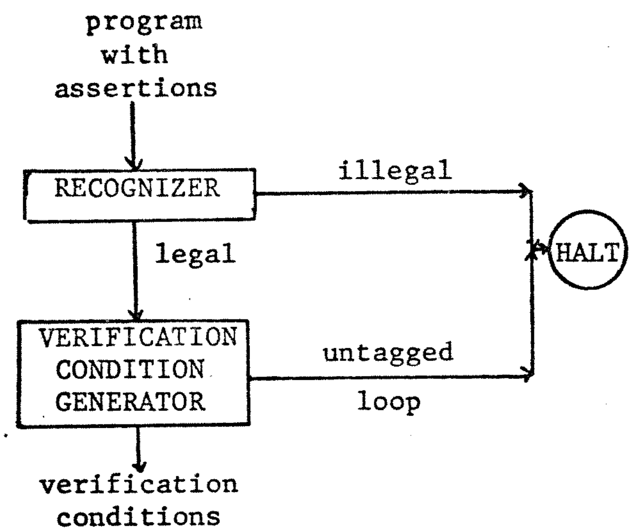

Figure IV.1. Verifier Structure

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 <u>initial assumption</u> is, "The Nucleus transition network and all system constants are prestored 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 prestored 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,PROCCALLS.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,PROCCALLS.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,PROCCALLS.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 ASRTLOCPT= -1;
(0.5)   ASSERT ASRTLOC1[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 CASELABELSETPT= -1;
(0.11)  ASSERT CASELABELFRONT[0]= 0;
(0.12)  ASSERT DEFINEDCASELABELSETTOP= -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)     ASRTSCANFLAG:=FALSE;
(10)    ENTER SCAN;
(11)    CARINSTRING:=TOKEN;
(12)    RTNSTACKTOP:=-1;
(13.1)  ASSERT ATPARSESTATE;
(13.2)  ASSERT ¬ASRTSCANFLAG;
(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,DEFINEDIDENTIFIERSET,DEFINEDIDENTIFIERSETPT,
        DEFINEDPROCEDURESET,DEFINEDPROCEDURESETPT,DESCLOC,
        EXPLIST,INITIALPROCEDURE,PROC,PROCCALLS,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 ASRTSCANFLAG 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 ∧ ATSAVEDPARSESTATE]
        ∨[¬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
(2)     THEN ENTER TRAVERSE;
(3)     ELSE ENTER NILMATCH;
(5)     IF NILMATCHFLAG
(6)     THEN ENTER TRAVERSE;
(7)     ELSE IF ARC < FIRSTARC[STATE+1] ∧ SYMBOL[ARC] < 0
(9)     THEN RTNSTACKTOP:=RTNSTACKTOP+1;
(10)    RTNSTACK[RTNSTACKTOP]:=ARC;
(11)    STATE:=-SYMBOL[ARC];
(12)    ELSE IF RECOGNITIONSTATE[ARC]
(14)    THEN ENTER STACKTEST;
(15)    IF STACKTESTFLAG
(16)    THEN ARC:=RTNSTACK[RTNSTACKTOP];
(17)    RTNSTACKTOP:=RTNSTACKTOP-1;
(18)    ENTER TRAVERSE;
(19)    ELSE IF RTNSTACKTOP < 0
(21)    THEN RECOGNITION:=TRUE;
(22)    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;
        FI;
        FI;
(25)    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;
        FI;
        FI;
        FI;
        FI;
(28.1)  ASSERT ASRTSCANFLAG.0 → ATASRTSCANSTATE;
(28.2)  ASSERT ¬ASRTSCANFLAG.0 ∧ ATSCANSTATE.0 → ATSCANSTATE ∧
        ATSAVEDPARSESTATE;
(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$\neq${ } $\wedge$ test(car(RTNSTACK.i))(REGVAL.i)" which appears as a test condition in the definition of the transition network traversal.

```
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 <u>not</u> 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
(2)            CURRENTPROC:=0;
(3.1)          ASSERT 0 ≤ P ≤ DEFINEDPROCEDURESETPT;
(3.2)          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
(4)                 CURRENTPROC:=CURRENTPROC+1;
(5)            ELIHW;
(6)            BIAS:=PROC[7*CURRENTPROC+1];
(7)            ENTER PROCVCGEN;
(8)            P:=P+1;
(9)       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],...,DEFINEDPROCEDURESET[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 $\wedge$ 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

Category	Number	Per Cent
immediate proofs	32	40
simple proofs	25	31
more difficult proofs	23	29
TOTAL	80	100

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 $\wedge$ (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 $\wedge$ V1 in RANGE(N1)

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 X SYMBOLS X TESTS X ACTIONS X SCANFLAG X STATES

where SYMBOLS = ALPHABET U STATES U {NIL}

NIL not in ALPHABET U 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) $S1 \neq S2 \vee \neg(P1(x) \wedge P2(x))$

(b) S1 in STATES $\rightarrow$ S2 not in STATES $\wedge$ Q1 not in RECOGNITIONSTATES

(7) RECOGNITIONSTATES $\subseteq$ STATES

(8) 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 = { }

stateof(x) = car(x)	first element
symbol(x) = cadr(x)	second element
test(x) = caddr(x)	third element
action(x) = cadddr(x)	fourth element
flag(x) = caddddr(x)	fifth element
nextstate(x) = cadddddr(x)	sixth element

IMPERATIVES

```
if[∃ARC in ARCS∋(stateof(ARC)=STATE.i)∧(symbol(ARC)=car(INSTRING.i))∧(test(ARC)(REGVAL.i))]
  then E[i+1]=A[INSTRING, if flag(ARC)=SCAN then cdr(INSTRING.i) else INSTRING.i,
             A[REGVAL, action(ARC)(REGVAL.i),
             A[STATE, nextstate(ARC),E[i] ]]]
  else if[∃ARC in ARCS∋(stateof(ARC)=STATE.i)∧(symbol(ARC)=NIL)∧(test(ARC)(REGVAL.i))]
       then E[i+1]=A[INSTRING, if flag(ARC)=SCAN then cdr(INSTRING.i) else INSTRING.i,
                  A[REGVAL, action(ARC)(REGVAL.i),
                  A[STATE,nextstate(ARC), E[i] ]]]
       else if[∃ARC in ARCS∋(stateof(ARC)=STATE.i)∧(symbol(ARC) in STATES)]
            then E[i+1]=A[RTNSTACK, cons(ARC,RTNSTACK.i),
                       A[STATE, symbol(ARC), E[i] ]]
            else if [STATE.i IN RECOGNITIONSTATES]
                 then if[(RTNSTACK.i≠{})∧(test(car(RTNSTACK.i))(REGVAL.i))]
                      then E[i+1]=A[RTNSTACK, cdr(RTNSTACK.i),
                                A[INSTRING, if flag(car(RTNSTACK.i))=SCAN then cdr(INSTRING.i)
                                                                         else INSTRING.i,
                                A[REGVAL, action(car(RTNSTACK.i))(REGVAL.i),
                                A[STATE, nextstate(car(RTNSTACK.i)), E[i] ]]]]
                      else if [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 L IS USED, IT IS OF THE FORM P:Q 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 BOUND.

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 OF A CASE-ELSE STATEMENT. X IS THE INTEGER VALUED CASE EXPRESSION, AND P IS THE POINT FOLLOWING THE ESAC.

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

CASELABELSET(L) = S IS PRODUCED IF POINT Q IN PROCEDURE P HAS A CASE STATEMENT. S 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 N.

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.

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

IF(L,X,T,F) IS PRODUCED WHEN POINT Q IN PROCEDURE P HAS A TWO WAY BRANCH. X IS THE EXPRESSION TO BE EVALUATED AND IF X IS TRUE CONTROL GOES TO POINT T, OTHERWISE TO POINT F. IFS ARE PRODUCED FOR A 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 FI.
- 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 ELIHW.

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

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

```
            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(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 FI.
            NULL:  JUMPTO(L.PLUS1(Q))
            RETURN: JUMPTO(L.EXITPOINT(P)).
            WHILE: JUMPTO(L.W) WHERE Q IS THE POINT BEFORE THE
                   ELIHW. AND W IS THE POINT AT THE WHILE.

POINTLABELLEDWITH(S) = Q
            A SENTENCE OF THIS FORM IS PRODUCED FOR EVERY IDENTIFIER
            THAT APPEARS AS THE LABEL OF A STATEMENT. IN THIS CASE
            S HAS THE FORM P:I WHERE P IS THE NAME OF THE PROCEDURE
            CONTAINING THE LABEL AND I IS THE LABEL ITSELF. A SENTENCE
            OF THIS FORM ALSO IS PRODUCED FOR EVERY NUMBER THAT
            APPEARS AS A CASE LABEL. IN THIS CASE S HAS THE FORM
            P:C:N WHERE P IS THE NAME OF THE PROCEDURE CONTAINING
            THE LABEL. C IS THE POINT IN THE PROCEDURE THAT HAS THE
            CASE STATEMENT IN WHICH THE LABEL IS DEFINED. AND N IS
            THE LABEL ITSELF.  IN BOTH CASES Q IS THE POINT THAT HAS
            THE LABELLED STATEMENT.

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

```
************************************************************************
*                                                                      *
*                         NUCLEUS AXIOMS                               *
*                       FILE 1380.AXIOMS                               *
*                            5/1/72                                    *
*                                                                      *
************************************************************************


P R I M I T I V E   F U N C T I O N S   .  .  .

+,-,*  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 - B * (A / B) WHERE /
        DENOTES INTEGER DIVISION AS DEFINED ABOVE.

<,≤,≥,>  DENOTE THE USUAL MATHTHEMATICAL ORDERING RELATIONS ON THE
         INTEGERS.

¬,∧,∨,→  DENOTE THE LOGICAL OPERATIONS OF NOT, AND, OR, AND IMPLIES.
         THE ASSUMED PRECEDENCE IS ¬ FIRST,..., → LAST.

BOOLOFCHAR(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 CHARACTER.CONSTANT TOKEN AND
                   RETURNS THE SECOND CHARACTER OF THAT TOKEN STRING.

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

CHAROFINT(X) = ↑(BLANK)↑  IF ABS(X) MOD 64 = 0
             = ↑A↑  IF ABS(X) MOD 64 = 1
             .
             .
             .
             = ↑(SHARP)↑  IF ABS(X) MOD 64 = 63
    THE ORDER USED ABOVE IS THE SAME AS THE ORDER OF APPEARANCE
    IN THE BASIC NUCLEUS CHARACTER SET . . .

       (BLANK) A B C D E F G H I J K L M N O P Q R S T U V W X
       Y Z 0 1 2 3 4 5 6 7 8 9 ( [ ] ) + * / ↑ ↓ - < ≤ ≥ > = ≠
       ¬ ∧ ∨ → ≡ , ; : . $ (SHARP)
```

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

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

IDENTIFIERTOKEN(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.

INTEGERVALUE(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 X = *A*
             .
             .
             .
             = 63  IF X = *(SHARP)*
     THE ORDER ABOVE CORRESPONDS WITH THE ORDER OF APPEARANCE IN
     THE BASIC NUCLEUS CHARACTER SET.

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

PLUS1(X) = DIGITS(INTEGERVALUE(X)+1)

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


I M P L E M E N T A I O N   P A R A M E T E R S   .  .  .


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.

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

READSIZE  IS THE FIXED NUMBER OF POSITIONS IN ANY INPUT RECORD.

WRITESIZE  IS THE FIXED NUMBER OF POSITIONS IN ANY OUTPUT RECORD.
```

```
D E C L A R A T I V E S   .   .   .

 1 INNAMESPACE(U)

 2 SIMPLE(X) → INNAMESPACE(X)

 3 SIMPLE(↑:LOC↑)

 4 SIMPLE(↑:LVL↑)

 5 SIMPLE(↑:RDHD↑)

 6 SIMPLE(↑:WTHD↑)

 7 ARRAY(A,B) → BOUND(A)=B

 8 0≤I≤INTEGERVALUE(BOUND(A)) → INNAMESPACE(A ↑[↑ DIGITS(I) ↑]↑)

 9 ARRAY(↑:RTNPT↑,DIGITS(MAXSTACKSIZE))


E V A L U A T I V E S   .   .   .

10 IDENTIFERTOKEN(X) → NAME(X,I)=X

11 NAME(A ↑[↑ S ↑]↑,I) = IF 0≤S.I≤INTEGERVALUE(BOUND(A))
                         THEN (A ↑[↑ DIGITS(S,I) ↑]↑) 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 INNAMESPACE(X) → X.I=IF X=U THEN U ELSE ε[I](X)

17 ↑:LOC↑.0 = INITIALPROCEDURE ↑:0↑

18 ↑:LVL↑.0 = -1

19 ↑:RUHD↑.0 = 0

20 ↑:WTHD↑.0 = 0

21 (↑+↑ X).I = X.I

22 (↑-↑ X).I = IF X.I=U THEN U
               ELSE IF INRANGE(-X.I) THEN -X.I ELSE U

23 (X ↑+↑ Y).I = 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).I = 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).I = IF X.I=U ∨ Y.I=U THEN U
```

```
                  ELSE IF INRANGE(X.I*Y.I) THEN X.I*Y.I ELSE U

26 (X ↑/↑ Y).I = IF X.I=U ∨ Y.I=U ∨ Y.I=0 THEN U
                  ELSE IF INRANGE(X.I/Y.I) THEN X.I/Y.I ELSE U

27 (X ↑÷↑ Y).I = IF X.I=U ∨ Y.I=U ∨ Y.I=0 THEN U
                  ELSE IF INRANGE(X.I MOD Y.I)
                       THEN X.I MOD Y.I ELSE U

28 (X ↑<↑ Y).I = IF X.I=U ∨ Y.I=U THEN U ELSE X.I<Y.I

29 (X ↑≤↑ Y).I = IF X.I=U ∨ Y.I=U THEN U ELSE X.I≤Y.I

30 (X ↑=↑ Y).I = IF X.I=U ∨ Y.I=U THEN U ELSE X.I=Y.I

31 (X ↑≠↑ Y).I = IF X.I=U ∨ Y.I=U THEN U ELSE X.I≠Y.I

32 (X ↑≥↑ Y).I = IF X.I=U ∨ Y.I=U THEN U ELSE X.I≥Y.I

33 (X ↑>↑ Y).I = IF X.I=U ∨ Y.I=U THEN U ELSE X.I>Y.I

34 (↑¬↑ X).I = IF X.I=U THEN U ELSE ¬X.I

35 (X ↑∧↑ Y).I = IF X.I=U ∨ Y.I=U THEN U ELSE X.I∧Y.I

36 (X ↑∨↑ Y).I = IF X.I=U ∨ Y.I=U THEN U ELSE X.I∨Y.I

37 (↑INTOFBOOL(↑ X ↑)↑).I = IF X.I=U THEN U ELSE INTOFBOOL(X.I)

38 (↑INTOFCHAR(↑ X ↑)↑).I = IF X.I=U THEN U ELSE INTOFCHAR(X.I)

39 (↑BOOLOFINT(↑ X ↑)↑).I = IF X.I=U THEN U ELSE BOOLOFINT(X.I)

40 (↑BOOLOFCHAR(↑ X ↑)↑).I = IF X.I=U THEN U ELSE BOOLOFCHAR(X.I)

41 (↑CHAROFINT(↑ X ↑)↑).I = IF X.I=U THEN U ELSE CHAROFINT(X.I)

42 (↑CHAROFBOOL(↑ X ↑)↑).I = IF X.I=U THEN U ELSE CHAROFBOOL(X.I)

43 (A ↑[↑ S ↑]↑).I = NAME(A ↑[↑ S ↑]↑,I).I

44 (↑(↑ X ↑)↑).I = X.I

45 PNAME(X ↑:↑ Y) = X

46 LOCPLUS1(X ↑:↑ Y) = X ↑:↑ PLUS1(Y)


I M P E R A T I V E S   .   .   .

47 A[N,V,S](X) = IF N=X THEN V ELSE S(X)

48 ASSIGN(↑:LOC↑.I,N,V) →
        [NAME(N,I)=U ∨ V.I=U → TERMINATION(I)]
      ∧ [¬(NAME(N,I)=U ∨ V.I=U) →
              E[I,1] = A[↑:LOC↑,LOCPLUS1(↑:LOC↑.I),
                         A[N,V.I,E[I]]]];

49 CASE(↑:LOC↑.I,X,P) →
```

```
        [X,I=U → TERMINATION(I)]
      ∧ [X,I≠U →
            E[I+1] = A[↑:LOC↑,PNAME(↑:LOC↑,I) ↑:↑
                     (IF DIGITS(X,I) IN CASELABELSET(↑:LOC↑,I)
                     THEN POINTLABELLEDWITH(↑:LOC↑,I ↑:↑ DIGITS(X,I))
                     ELSE P),E[I]]]

50 ENTER(↑:LOC↑,I,N) →
        [NAME(↑:RTNPT[:LVL+1]↑,I)=U → TERMINATION(I)]
      ∧ [NAME(↑:RTNPT[:LVL+1]↑,I)≠U →
         E[I+1] = A[↑:LOC↑,N ↑:0↑,
                  A[NAME(↑:RTNPT[:LVL+1]↑,I),LOCPLUS1(↑:LOC↑,I),
                   A[↑:LVL↑,↑:LVL+1↑,I,E[I]]]]]

51 EXIT(↑:LOC↑,I) →
        [↑:LVL↑,I<0 → TERMINATION(I)]
      ∧ [↑:LVL↑,I≥0 →
            E[I+1] = A[↑:LOC↑,I,↑:RTNPT[LVL]↑,I,
                      A[↑:LVL↑,↑:LVL-1↑,I,E[I]]]]

52 HALT(↑:LOC↑,I) → TERMINATION(I)

53 IF(↑:LOC↑,I,X,P,Q) →
        [X,I=U → TERMINATION(I)]
      ∧ [X,I≠U →
             E[I+1] = A[↑:LOC↑,PNAME(↑:LOC↑,I) ↑:↑
                      IF X,I=TRUE THEN P ELSE Q,E[I]]]

54 JUMPTO(↑:LOC↑,I,P) →
        E[I+1] = A[↑:LOC↑,PNAME(↑:LOC↑,I) ↑:↑ P,E[I]]

55 READ(↑:LOC↑,I,A) ∧ :REOF((↑:RDHD↑,I)+1) →
        E[I+1](X) = IF X=A ↑[0]↑ THEN ↑T↑
                    ELSE IF X=↑:RDHD↑ THEN (↑:RDHD↑,I)+1
                         ELSE IF X=↑:LOC↑ THEN LOCPLUS1(↑:LOC↑,I)
                              ELSE E[I](X)

56 READ(↑:LOC↑,I,A) ∧ ¬:REOF((↑:RDHD↑,I)+1) →
        E[I+1](X) = IF X=A ↑[0]↑  THEN ↑F↑
                    ELSE IF 1≤J≤MIN(INTEGERVALUE(BOUND(A)),READSIZE)
                              ∧ X=A ↑[↑ DIGITS(J) ↑]↑
                         THEN :RDFL((↑:RDHD↑,I)+1,J)
                         ELSE IF X=↑:RDHD↑ THEN (↑:RDHD↑,I)+1
                              ELSE IF X=↑:LOC↑
                                   THEN LOCPLUS1(↑:LOC↑)
                                   ELSE E[I](X)

57 WRITE(↑:LOC↑,I,A) ∧ (A ↑[0]↑),I=↑T↑ →
        [E[I+1] = A[↑:LOC↑,LOCPLUS1(↑:LOC↑,I),
                  A[↑:WTHD↑,(↑:WTHD↑,I)+1,E[I]]]]
      ∧ [:WEOF((↑:WTHD↑,I)+1)]

58 WRITE(↑:LOC↑,I,A) ∧ (A ↑[0]↑),I≠↑T↑ →
        [E[I+1] = A[↑:LOC↑,LOCPLUS1(↑:LOC↑,I),
                  A[↑:WTHD↑,(↑:WTHD↑,I)+1,E[I]]]]
      ∧ [¬:WEOF((↑:WTHD↑,I)+1)]
      ∧ [1≤J≤MIN(INTEGERVALUE(BOUND(A)),WRITESIZE) →
            :WTFL((↑:WTHD↑,I)+1,J)=(A ↑[↑ DIGITS(J) ↑]↑),I]
      ∧ [INTEGERVLAUE(BOUND(A))+1≤J≤WRITESIZE →
            :WTFL((↑:WTHD↑,I)+1,J)=↑ ↑]
```

APPENDIX D

VERIFIER LISTING

```
INTEGER
ANDTYPETOP,$POINTS TO TOP OF AND.TYPE STACK.  -1 FOR EMPTY $
ARC,$CURRENT TRANSITION NETWORK ARC DURING SCAN AND PARSE $
ASRTBACK,$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 ASRTLOC1.  -1 FOR EMPTY $
ASRTSCANPOINTER,$POINTS TO NEXT 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,$ SAVES 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 $
BINARYADDTYPETOP,$POINTS TO TOP OF BINARY.ADD.TYPE STACK.  -1 FOR EMPTY $
CARINSTRING,$NEXT INPUT SYMBOL DURING TRANSITION NETWORK TRAVERSAL $
CASEEXPRESSIONTOP,$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 $
CHARLISTPT,$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 PROCEDURE DESCRIPTION IN ARRAY PROC $
CURRFRONT,$POINTS TO AN ELEMENT OF ARRAY PATHFRONTS. PATHFRONTS[CURRFRONT] IS
        CONTROL POINT FROM WHICH PATHS ARE CURRENTLY BEING GENERATED $
DEFINEDARRAYSETPT,$POINTS TO LAST ENTRY OF DEFINED.ARRAY.SET.  -1 FOR EMPTY $
DEFINEDCASELABELSETTOP,$POINTS TO TOP OF DEFINED.CASE.LABEL.SET STACK.  -1 FOR
        EMPTY $
DEFINEDIDENTIFIERSETPT,$POINTS TO LAST ENTRY OF DEFINED.IDENTIFIER.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 $
EXPBACK,$POINTS TO END OF AS EXPRESSION DESCRIPTION IN ARRAY EXPLIST.  USED BY
        PROCEDURE EXPCHECKER $
EXPFRONT,$POINTS TO FRONT OF AN EXPRESSION DESCRIPTION IN ARRAY EXPLIST. USED BY
        PROCEDURE EXPCHECKER $
EXPLISTPT,$POINTS TO LAST ENTRY IN EXPLIST.  -1 FOR EMPTY $
EXPTYPETOP,$POINTS TO TOP OF EXP.TYPE STACK.  -1 FOR EMPTY $
F,$TEMPORARY $
FINDID1,FINDID2,$POINT TO FRONT AND BACK OF AN IDENTIFIER STRING IN ARRAY
        CHARLIST.  USED AS INPUT ARGUMENTS BY PROCEDURE FINDID $
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,$TEMPORARIES $
ID,$POINTS TO AN IDENTIFIER IN ARRAY DEFINEDIDENTIFIERSET $
IDENTSTR1,IDENTSTR2,IDENTSTR3,IDENTSTR4,$INPUT ARGUMENTS FOR PROCEDURE IDENTSTR
        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 $
IFEXPRESSIONTOP,$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 $
J.K.L.$TEMPORARIES$
LABELTABLEPT.$POINTS TO LAST ENTRY IN ARRAY LABELTABLE.  -1 FOR EMPTY $
LEFTARRAYNAME.$NETWORK 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.$NETWORK 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 CHARLIST $
LINEPT.$POINTS TO LAST ENTRY IN ARRAY LINE $
MULTIPLYTYPETOP.$POINTS TO TOP OF MULTIPLY.TYPE STACK.  -1 FOR EMPTY $
N.$TEMPORARY$
NEWCASELABEL.$NETWORK REGISTER--INTEGER VALUE OF MOST RECENT CASE LABEL $
NEXTCHARACTER.$NETWORK REGISTER--SAVES THE CURRENT INPUT SYMBOL $
NOTTYPETOP.$POINTS TO TOP OF NOT.TYPE STACK.  -1 FOR EMPTY $
NSTEP.$COUNTS NUMBER OF PATH STEPS $
P.$TEMPORARY$
PARSESTACKTOP.$POINTS TO TOP OF PARSESTACK.  -1 FOR EMPTY $
PARSESTART.$STATE NUMBER FOR INITIATION OF PARSE NETWORK $
PATHFRONTSPT.$POINTS TO LAST ENTRY IN ARRAY PATHFRONTS.  -1 FOR EMPTY $
PATHPT.$POINTS TO LAST ENTRY IN ARRAY PATH.  -1 FOR EMPTY $
POINT.$NETWORK REGISTER--CURRENT CONTROL POINT $
PRIMARYTYPETOP.$POINTS TO TOP OF PRIMARY.TYPE STACK.  -1 FOR EMPTY $
PROCCALLED.$ POINTS TO DESCRIPTION OF CALLED PROCEDURE IN ARRAY PROC--USED IN
        PROCEDURE GENENTERTERM $
PROCEDURENAME.$NETWORK REGISTER--POINTS TO IDENTIFIER IN DEFINED.IDENTIFIER.SET
        WHICH NAMES CURRENT PROCEDURE $
PROCNUM.$POINTS TO A PROCEDURE DESCRIPTION IN PROC[7*PROCNUM] THROUGH
        PROC[7*PROCNUM+6] $
R.$TEMPORARY$
REFERENCEDLABELSETPT.$POINTS TO LAST ENTRY IN REFERENCED.LABEL.SET.  -1 FOR
        EMPTY $
REFERENCEDPROCEDURESETPT.$POINTS TO LAST ENTRY IN REFERENCED.PROCEDURE.SET
        -1 FOR EMPTY $
RELATIONTYPETOP.$POINTS TO TOP OF RELATION.TYPE STACK.  -1 FOR EMPTY $
RIGHTBRACK.$USED AS INPUT ARGUMENT BY PROCEDURE BALANCE.  CONTAINS CODE FOR
        RIGHT BRACKET $
RTNSTACKTOP.$POINTS TO TOP OF RTNSTACK.  -1 FOR EMPTY $
S.$TEMPORARY$
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.$TEMPORARY$
TOKEN.$NETWORK REGISTER--INTEGER CODE OF TOKEN $
TYPE.$NETWORK REGISTER-- =1 FOR INTEGER. =2 FOR BOOLEAN. =3 FOR CHARACTER $
UNARYTYPETOP.$POINTS TO TOP OF UNARY.TYPE STACK.  -1 FOR EMPTY $
V.W.$TEMPORARIES$
WHILEEXPRESSIONTOP.$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 SYMBOLS $
DONE.$USED BY PROCEDURE LISTCALLEDPROCS TO INDICATE WHEN ALL PROCEDURES
        REACHABLE FROM CALLED PROCEDURE HAVE BEEN LISTED $
FOUND.$RETURNS RESULT FOR SEVERAL SEARCH PROCEDURES $
IDENTSTRFLAG.$RETURNS RESULT OF PROCEDURE IDENTSTR.  TRUE IF SPECIFIED STRINGS
        ARE IDENTICAL $
INRANGE.$SET BY PROCEDURE EVALINTOK. TRUE IF NUMBER IS IN RANGE(0≤NUMBER≤999999)
        $
NILMATCHFLAG.$RETURNS RESULT OF PROCEDURE NILMATCH IN TRANSITION NETWORK
        TRAVERSAL $
RECOGNITION.$BECOMES TRUE WHEN TRANSITION NETWORK RECOGNIZES A STRING $
STACKTESTFLAG.$RETURNS RESULT OF PROCEDURE STACKTEST IN TRANSITION NETWORK
        TRAVERSAL $
TESTFLAG:$RETURNS RESULT OF NETWORK TESTS $


CHARACTER
CHAR:$CONTAINS A NUCLEUS CHARACTER $


INTEGER ARRAY
ACTION[161].$ACTIONS OF TRANSITION NETWORK. ACTION[X] CONTAINS ACTION NUMBER FOR
        ARC X $
ALTNUM[249].$ALTERATION 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 $
ANDTYPE[9].$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[1999].$ASSERTION POINTERS. -1 INDICATES NO ASSERTION. OTHERWISE POINTS
        TO ARRAY ASRTLOC1 $
ASRTLOC1[499].$ASSERTION 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[ASRTLOC1[X]].....ASRTS[ASRTLOC1[X+1]-1] IS A LIST OF
        FRONT-BACK PAIRS WHICH POINT TO ASSERTION BODY STRINGS IN ARRAY
        CHARLIST $
BINARYADDTYPE[9].$NETWORK REGISTER--BINARY.ADD.TYPE STACK  =1 FOR INTEGER.
        =2 FOR BOOLEAN. =3 FOR CHARACTER $
BOUNDFUNCTION[249].$BOUNDFUNCTION[X]=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[9].$NETWORK REGISTER--CASEEXPRESSION[2*X] AND
        CASEEXPRESSION[2*X+1] POINT TO FRONT AND BACK OF CASE EXPRESSION IN
        ARRAY EXPLIST $
CASELABELFRONT[500].$POINTERS TO FRONTS OF LABEL LISTS IN ARRAY CASELABELS.
        ALWAYS 1 EXTRA ENTRY TO PROVIDE END OF LAST LIST $
CASELABELPOINTS[49].$CASELABELPOINTS[X] CONTAINS CONTROL POINT OF CASE LABEL IN
        DEFINEDCASELABELSET[X] $
CASELABELS[549].$ELEMENTS ARE CASE LABELS--CASELABELS[CASELABELFRONT[X]].....
        CASELABELS[CASELABELFRONTS[X+1]-1] IS A CASE LABEL LIST FOR A POINT $
CASELABELSET[499].$CASELABELSET[X] IS CONTROL POINT FOR LIST POINTED TO BY
        CASELABELFRONT[X] $
CASELABELSTACK[4].$CASELABELSTACK[X] POINTS TO LAST ENTRY OF LEVEL X OF
        DEFINED.CASE.LABEL.SET STACK $
```

```
CASEPOINT[4],$NETWORK REGISTER--CASE.POINT STACK $
COLONALTNUM[10],$ALTERATION COUNTERS FOR COLON-IDENTIFIERS(:LOC,:LVL,:RDHD,ETC)$
COLONPTS[11],$COLONPTS[X] POINTS TO FRONT OF IDENTIFIER STRING IN ARRAY
        COLONWORDSET FOR COLON-IDENTIFIER NUMBER X $
DEFINEDARRAYSET[49],$NETWORK REGISTER--IDENTIFIER NUMBERS FOR ELEMENTS OF
        DEFINED.IDENTIFIER.SET WHICH ARE IN DEFINED.ARRAY.SET $
DEFINEDCASELABELSET[49],$NETWORK REGISTER--DEFINEDCASELABELSET[X] IS LABEL AT
        CONTROL POINT CASELABELPOINTS[X] $
DEFINEDIDENTIFIERSET[499],$NETWORK REGISTER--DEFINEDIDENTIFIERSET[2*X] AND
        DEFINEDIDENTIFIERSET[2*X+1] POINT TO FRONT AND BACK OF IDENTIFIER
        NUMBER X IN ARRAY CHARLIST $
DEFINEDLABELSET[49],$NETWORK REGISTER--DEFINEDLABELSET[2*X] AND
        DEFINEDLABELSET[2*X+1] POINT TO FRONT AND BACK OF LABEL NUMBER X IN
        ARRAY CHARLIST $
DEFINEDPROCEDURESET[199],$NETWORK REGISTER--CONTAINS IDENTIFIER NUMBERS OF
        NAMES OF DEFINED PROCEDURES IN DEFINED.IDENTIFIER.SET $
DEFINEDSIMPLESET[249],$NETWORK REGISTER--CONTAINS IDENTIFIER NUMBERS OF NAMES
        OF DEFINED SIMPLE VARIABLES $
DESCLOC[1999],$ IF POINT+BIAS=X THEN DESCLOC[X] POINTS TO DESCRIPTION OF POINT
        IN ARRAY STATEMENT $
EXPLIST[49999],$EXPLIST[2*X] AND EXPLIST[2*X+1] CONTAIN A TOKEN-VALUE PAIR.
        E.G.(IDENTIFIER,ID NUMBER) OR (NUMBER,VALUE) $
EXPSTRING[1],$REPLACES ALL EXPRESSION STACKS. POINTS TO FRONT AND BACK OF
        EXPRESSION IN EXPLIST $
EXPTYPE[9],$NETWORK REGISTER--EXP.TYPE STACK  =1 FOR INTEGER, =2 FOR BOOLEAN, =3
        FOR CHARACTER $
FIRSTARC[108],$ARCS FOR TRANSITION NETWORK STATES. FIRSTARC[X] THROUGH
        FIRSTARC[X+1]-1 ARE ARCS FROM STATE X $
IDENTIFIERNAME[1],$NETWORK REGISTER--FRONT AND BACK OF IDENTIFIER STRING IN
        ARRAY CHARLIST $
IFELSEPOINT[9],$NETWORK REGISTER--IF.ELSE.POINT STACK $
IFEXPRESSION[19],$NETWORK REGISTER--IFEXPRESSION[2*X] AND IFEXPRESSION[2*X+1]
        ARE FRONT AND BACK OF EXPRESSION IN EXPLIST AND REPRESENT IF EXPRESSION
        AT LEVEL X $
IFPOINT[9],$NETWORK REGISTER--IF.POINT STACK $
LABELTABLE[74],$LABELTABLE[3*X] AND LABELTABLE[3*X+1] ARE FRONT AND BACK OF
        LABEL STRING IN CHARLIST. LABELTABLE[3*X+2] IS CONTROL POINT FOR LABEL
        OR LINK TO LABEL REFERENCES IF NOT YET DEFINED.  -1 IS LAST LINK $
LEFTSIDE[3],$NETWORK REGISTER--LEFTSIDE[0]=1 FOR SIMPLE, =2 FOR ARRAY REFERENCE.
        LEFTSIDE[1]=IDENTIFIER NUMBER.  LEFTSIDE[2] AND LEFTSIDE[3] POINT TO
        FRONT AND BACK OF SUBSCRIPT EXPRESSION IN EXPLIST FOR ARRAY REFERENCES$
MULTIPLYTYPE[9],$NETWORK REGISTER--MULTIPLY.TYPE STACK  =1 FOR INTEGER, =2 FOR
        BOOLEAN, =3 FOR CHARACTER $
NEXTSTATE[161],$NEXT STATES FOR TRANSITION NETWORK ARCS.  NEXTSTATE[X] IS NEXT
        STATE FOR ARC X $
NOTTYPE[9],$NETWORK REGISTER--NOT.TYPE STACK  =1 FOR INTEGER, =2 FOR BOOLEAN,
        =3 FOR CHARACTER $
PARSESTACK[99],$TEMPORARY STORAGE FOR THE PARSE RETURN POINT STACK WHILE SCAN
        NETWORK IS TRAVERSED $
PATH[49],$PATH[X] IS THE X-TH CONTROL POINT ALONG THE CURRENT PATH $
PATHFRONTS[19],$CONTAINS CONTROL POINTS FROM WHICH PATHS MAY BEGIN $
PRIMARYTYPE[9],$NETWORK REGISTER--PRIMARY.TYPE STACK  =1 FOR INTEGER, =2 FOR
        BOOLEAN, =3 FOR CHARACTER $
PROC[1399],$PROC[7*X],....,PROC[7*X+6] CONTAIN POINTERS TO THE DESCRIPTION OF
        PROCEDURE NUMBER X
        PROC[7*X]  =IDENTIFIER NUMBER OF PROCEDURE NAME
        PROC[7*X+1]=POINTER TO ARRAY DESCLOC FOR CONTROL POINT 0
        PROC[7*X+2]=POINTER TO ARRAY DESCLOC FOR EXIT CONTROL POINT
        PROC[7*X+3]=POINTER TO FRONT OF CALLED PROCEDURES IN ARRAY PROCCALLS
        PROC[7*X+4]=POINTER TO BACK  OF CALLED PROCEDURES IN ARRAY PROCCALLS
        PROC[7*X+5]=POINTER TO FRONT OF ALTERATION LIST IN ARRAY ALTSET
        PROC[7*X+6]=POINTER TO BACK  OF ALTERATION LIST IN ARRAY ALTSET $
```

```
PROCCALLS[799],$CONTAINS CALLED PROCEDURES. PROCCALLS[2*X] AND PROCCALLS[2*X+1]
        POINT TO FRONT AND BACK OF A PROCEDURE NAME IN ARRAY CHARLIST $
REFERENCEDLABELSET[49],$NETWORK REGISTER--REFERENCEDLABELSET[2*X] AND
        REFERENCEDLABELSET[2*X+1] POINT TO FRONT AND BACK OF A LABEL IN
        ARRAY CHARLIST $
REFERENCEDPROCEDURESET[399],$NETWORK REGISTER--REFERENCEDPROCEDURESET[2*X] AND
        REFERENCEDPROCEDURESET[2*X+1] POINT TO FRONT AND BACK OF A PROCEDURE
        NAME IN ARRAY CHARLIST $
RELATIONTYPE[9],$NETWORK REGISTER--RELATION.TYPE STACK  =1 FOR INTEGER,
        =2 FOR BOOLEAN, =3 FOR CHARACTER $
RESCODE[26],$RESCODE[X] IS INTEGER CODE FOR RESERVED WORD NUMBER X $
RESWORDPTS[27],$RESWORDPTS[X] POINTS TO BEGINNING OF RESERVED WORD NUMBER X IN
        ARRAY RESERVEDWORDSET $
RTNSTACK[99],$RETURN POINT STACK FOR TRANSITION NETWORK $
STATEMENT[9999],$STORES DESCRIPTIONS OF STATEMENTS.  THE DESCRIPTION LENGTH
        VARIES ACCORDING TO THE STATEMENT TYPE
        ASSIGNMENT(7 ELEMENTS)
        1   64  (CODE FOR :=)
        2-5 SEE LEFTSIDE
        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(1ELEMENT)
        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(2 ELEMENTS)
        1   98 (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[1],$NETWORK REGISTER--POINTS TO FRONT AND BACK OF TOKEN.STRING IN
        ARRAY CHARLIST $
TYPEFUNCTION[249],$NETWORK REGISTER--TYPEFUNCTION[X] IS TYPE OF IDENTIFIER
        NUMBER X.  =1 FOR INTEGER, =2 FOR BOOLEAN, =3 FOR CHARACTER $
UNARYADDTYPE[9],$NETWORK REGISTER--UNARY.ADD.TYPE STACK  =1 FOR INTEGER,
        =2 FOR BOOLEAN, =3 FOR CHARACTER $
WHILEEXPRESSION[9],$NETWORK REGISTER--WHILEEXPRESSION[2*X] AND
        WHILEEXPRESSION[2*X+1] POINT TO FRONT AND BACK OF EXPRESSION IN EXPLIST
        AT LEVEL X $
WHILEPOINT[4];$NETWORK REGISTER--WHILE.POINT STACK $
```

```
BOOLEAN ARRAY
ALTFLAG[249],$ALTFLAG[X] IS TRUE IF IDENTIFIER NUMBER X CAN BE ALTERED BY A CALL
         OF PROCEDURE PROCCALLED $
COLONALTFLAG[10],$COLONALTFLAG[X] IS TRUE IF COLON-IDENTIFIER NUMBER X CAN BE
         ALTERED BY A CALL OF PROCEDURE PROCCALLED $
FLAG[161],$SCAN FLAGS FOR TRANSITION NETWORK.  FLAG[X] IS TRUE IF ARC X HAS A
         SCAN $
PROCFLAG[199],$PROCFLAG[X] IS TRUE IF PROCEDURE NUMBER X CAN BE REACHED BY A
         CALL OF PROCEDURE PROCCALLED $
PROCFLAG1[199],$PROCFLAG1[X] IS TRUE IF PROCEDURE NUMBER X HAS ALREADY BEEN
         PROCESSED BY PROCEDURE LISTCALLEDPROCS $
RECOGNITIONSTATE[107]:$RECOGNITION STATES FOR TRANSITION NETWORK.
         RECOGNITIONSTATE[X]=TRUE IF STATE X IS A RECOGNITION STATE $


CHARACTER ARRAY
ASRTTRUE[4],$CONSTANT ARRAY CONTAINING STRING TRUE $
BLANKLINE[0],$CONSTANT ARRAY CONTAINING F IN ELEMENT 0 TO PRINT A BLANK LINE $
CARD[80],$CONTAINS CARD IMAGE DURING SCAN AND PARSE $
CHARLIST[99999],$CHARACTER STRING STORAGE--IDENTIFIERS, LABELS, ASSERTIONS,ETC$
COLONWORDSET[38],$COLONWORDSET[COLONPTS[X]],....,COLONWORDSET[COLONPTS[X+1]-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[10],LINE2[10],LINE3[10],LINE4[10],$ 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 $
LVLLINE[13],$CONSTANT ARRAY CONTAINING :LVL.1=:LVL+1        $
NONRECOGNITION[14],$CONSTANT ARRAY CONTAINING NONRECOGNITION      $
PATHIS[7],$CONSTANT ARRAY CONTAINING PATH IS      $
PRESET[243],$CONSTANT ARRAY CONTAINING SEVERAL STRINGS USED IN BUILDING
         VERIFICATION CONDITIONS $
RESERVEDWORDSET[119]:$RESERVEDWORDSET[RESWORDPTS[X]],....,
         RESERVEDWORDSET[RESWORDPTS[X+1]-1] IS CHARACTER STRING FOR RESERVED
         WORD NUMBER X $


PROCEDURE  ACTIONS :
ASSERT [¬ASRTSCANFLAG ^ ATSCANSTATE ^ ATSAVEDPARSESTATE] v [¬ASRTSCANFLAG ^ ATPA
RSESTATE] v [ASRTSCANFLAG ^ ATASRTSCANSTATE] :
CASE ACTION[ARC] OF
0:  NOP :
1:  ENTER ACTION1 :
2:  ENTER ACTION2 :
3:  ENTER ACTION3 :
4:  ENTER ACTION4 :
5:  ENTER ACTION5 :
6:  ENTER ACTION6 :
7:  ENTER ACTION7 :
8:  ENTER ACTION8 :
9:  ENTER ACTION9 :
10:  ENTER ACTION10 :
11:  ENTER ACTION11 :
12:  ENTER ACTION12 :
13:  ENTER ACTION13 :
14:  ENTER ACTION14 :
15:  ENTER ACTION15 :
16:  ENTER ACTION16 :
17:  ENTER ACTION17 :
```

```
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 := FIRSTARC[STATE] ;
WHILE  ARC < FIRSTARC[STATE+1] ∧ SYMBOL[ARC] > 0  DO
      IF SYMBOL[ARC] = CARINSTRING
            THEN ENTER TESTS ;
            IF TESTFLAG
                        THEN ALPHABETMATCHFLAG := TRUE ;
                              RETURN ;
                  FI ;
      FI ;
      ARC := ARC + 1 ;
ELIHW ;
ALPHABETMATCHFLAG := FALSE ;
EXIT ;

PROCEDURE  ASRTSCAN ;
ASRTSCANFLAG:=TRUE ;
STATE:=SCANSTART ;
RTNSTACKTOP:=-1 ;
CARINSTRING:=INTEGER(CHARLIST[ASRTSCANPOINTER]);
RECOGNITION := FALSE ;
ASSERT ASRTSCANFLAG;
ASSERT ATASRTSCANSTATE;
WHILE ¬RECOGNITION DO
      ENTER TRANSNET ;
ELIHW ;
ASSERT INSTRING=REMOVETOKEN(INSTRING.0);
ASSERT TOKEN=NEXTTOKEN(INSTRING.0);
ASSERT TOKENSTRING=NEXTTOKENSTRING(INSTRING.0);
EXIT ;

PROCEDURE BALANCE;
ASSERT SUBFRONT POINTS TO THE FIRST ELEMENT TO THE RIGHT OF A LEFTBRACK OF AN EX
PRESSION IN EXPLIST;
N:=1;
ASSERT N≥1;
ASSERT N IS THE NUMBER OF EXCESS LEFTBRACKS OVER RIGHTBRACKS ENCOUNTERED IN EXPL
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 v EXPLIST[2*(SUBBACK+1)]≠RIGHTBRACK  DO
     SUBBACK:=SUBBACK+1;
     IF EXPLIST[2*SUBBACK]=LEFTBRACK
          THEN N:=N+1;
     FI;
     IF EXPLIST[2*SUBBACK]=RIGHTBRACK
          THEN N:=N-1;
     FI;
ELIHW;
ASSERT SUBFRONT.0 AND SUBBACK POINT TO FRONT AND BACK OF AN EXPRESSION IN EXPLIS
T WHICH IS BALANCED WITH RESPECT TO LEFTBRACK.0 AND RIGHTBRACK.0;
EXIT ;

PROCEDURE BUILDFROMPRESET;
ASSERT LINE[0]=LINE.0[0];
ASSERT LINEFRONT.0≤LINEFRONT≤LINEBACK.0+1;
ASSERT WRITEFILE(:WTHD) U LINE[1],...,LINE[LINEPT]=WRITEFILE(:WTHD.0) U LINE.0[1]
,...,LINE.0[LINEPT.0] U PRESET.0[LINEFRONT.0],...,PRESET.0[LINEFRONT-1];
WHILE LINEFRONT≤LINEBACK  DO
     IF LINEPT=120
          THEN WRITE LINE;
               LINEPT:=0;
     FI;
     LINEPT:=LINEPT+1;
     LINE[LINEPT]:=PRESET[LINEFRONT];
     LINEFRONT:=LINEFRONT+1;
ELIHW;
ASSERT LINE[0]=LINE.0[0];
ASSERT WRITEFILE(:WTHD) U LINE[1],...,LINE[LINEPT]=WRITEFILE(:WTHD.0) U LINE.0[1]
,...,LINE.0[LINEPT.0] U PRESET.0[LINEFRONT.0],...,PRESET.0[LINEBACK.0];
EXIT ;

PROCEDURE  BUILDLINE ;
ASSERT 0≤LINEFRONT≤LINEBACK+1≤CHARLISTPT+1;
ASSERT WRITEFILE(:WTHD) U LINE[0],...,LINE[LINEPT],CHARLIST[LINEFRONT],...,CHARLIS
T[LINEBACK]=WRITEFILE(:WTHD.0) U LINE.0[0],...,LINE.0[LINEPT.0],CHARLIST.0[LINEFR
ONT.0],...,CHARLIST.0[LINEBACK.0];
WHILE LINEFRONT≤LINEBACK DO
     IF LINEPT=120
          THEN WRITE LINE ;
               LINEPT := 0 ;
     FI ;
     LINEPT:=LINEPT+1 ;
     LINE[LINEPT]:=CHARLIST[LINEFRONT] ;
     LINEFRONT:=LINEFRONT+1 ;
ELIHW ;
ASSERT WRITEFILE(:WTHD) U LINE[0],...,LINE[LINEPT]=WRITEFILE(:WTHD.0) U LINE.0[0]
,...,LINE.0[LINEPT.0],CHARLIST.0[LINEFRONT.0],...,CHARLIST.0[LINEBACK.0] ;
ASSERT LINE[0]=LINE.0[0];
EXIT ;

PROCEDURE EVALINTOK ;
ASSERT 0≤TOKENSTRING[0]≤TOKENSTRING[1]≤CHARLISTPT;
ASSERT TOKENSTRING[0]≤$$≤TOKENSTRING[1] → +0≤CHARLIST[$$]≤+9;
I:=TOKENSTRING[0];
INTVAL:=0;
ASSERT INTVAL=INTEGERVALUE(CHARLIST.0[TOKENSTRING.0[0]],...,CHARLIST.0[I-1])≤999
999;
ASSERT TOKENSTRING.0[0]≤I≤TOKENSTRING.0[1]+1;
ASSERT CHARLIST=CHARLIST.0;
ASSERT TOKENSTRING=TOKENSTRING.0;
WHILE I≤TOKENSTRING[1] DO
```

```
    V:=INTEGER(CHARLIST[I])-27;
    IF INTVAL>(999999-V)/10
         THEN INRANGE := FALSE ;
              INTVAL:=999999 ;
         RETURN ;
    FI ;
    INTVAL:=10*INTVAL+V;
    I:=I+1 ;
ELIHW ;
INRANGE:=TRUE ;
ASSERT INRANGE → INTVAL=INTEGERVALUE(CHARLIST.0[TOKENSTRING.0[0]],...,CHARLIST.0
[TOKENSTRING.0[1]])≤999999;
ASSERT ¬INRANGE → INTEGERVALUE(CHARLIST.0[TOKENSTRING.0[0]],...,CHARLIST.0[TOKEN
STRING.0[1]])>999999 ^ INTVAL=999999;
EXIT ;

PROCEDURE EXPCHECKER;
ASSERT EXPBACK=EXPBACK.0;
ASSERT EXPFRONT.0≤EXPFRONT≤EXPBACK;
ASSERT EXPLIST CONTAINS AN EXPRESSION BETWEEN EXPFRONT.0 AND EXPBACK.0;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U VERIFICATION CONDITION LINES FOR EA
CH ARRAY SUBSCRIPT WHOSE [ PRECEDES EXPFRONT AND FOR EACH DIVISOR WHOSE / OR ÷ P
RECEDES EXPFRONT IN THE EXPRESSION IN EXPLIST.0 BETWEEN EXPFRONT.0 AND EXPBACK.0
;
ASSERT LINEPT=0 ^ LINE[0]=↑F;
WHILE EXPFRONT<EXPBACK  DO
     CASE EXPLIST[2*EXPFRONT]  OF
$[$       41: ENTER INSERTPRV;
              CHAR:=↑0;
              ENTER INSERTCHAR;
              CHAR:=↑≤;
              ENTER INSERTCHAR;
              SUBFRONT:=EXPFRONT+1;
              SUBBACK:=SUBFRONT;
              LEFTBRACK:=41;  $[$
              RIGHTBRACK:=42;  $]$
              ENTER BALANCE;
              ENTER PRINTSUBEXP;
              CHAR:=↑≤;
              ENTER INSERTCHAR;
              INTVAL:=BOUNDFUNCTION(EXPLIST[2*(EXPFRONT-1)+1]);
              ENTER INTPRINT;
              ENTER PRINTLINE;
 $/ ÷$     48:59: ENTER INSERTPRV;
              SUBFRONT:=EXPFRONT+2;
              SUBBACK:=SUBFRONT;
              LEFTBRACK:=43; $($
              RIGHTBRACK:=44; $)$
              ENTER BALANCE;
              ENTER PRINTSUBEXP;
              CHAR:=↑≠;
              ENTER INSERTCHAR;
              CHAR:=↑0;
              ENTER INSERTCHAR;
              ENTER PRINTLINE;
      ESAC;
      EXPFRONT:=EXPFRONT+1;
 ELIHW;
 ASSERT LINEPT=0 ^ LINE[0]=↑F;
 ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U  VERIFICATION CONDITION LINES FOR E
 ACH ARRAY SUBSCRIPT OR DIVISOR IN THE EXPRESSION IN EXPLIST.0 BETWEEN EXPFRONT.0
  AND EXPBACK.0;
```

```
EXIT ;

PROCEDURE FINDID;
ASSERT 0≤SS≤DEFINEDIDENTIFIERSETPT → 0≤DEFINEDIDENTIFIERSET[2*SS]≤DEFINEDIDENTIF
IERSET[2*SS+1]≤CHARLISTPT;
ASSERT 0≤FINDID1≤CHARLISTPT ∧ 0≤FINDID2≤CHARLISTPT;
ASSERT DEFINEDIDENTIFIERSETPT≥-1;
IDENTSTR1:=FINDID1;
IDENTSTR2:=FINDID2;
AT:=0;
ASSERT 0≤SS≤DEFINEDIDENTIFIERSETPT → 0≤DEFINEDIDENTIFIERSET[2*SS]≤DEFINEDIDENTIF
IERSET[2*SS+1]≤CHARLISTPT;
ASSERT 0≤SS≤AT-1 → CHARLIST.0[DEFINEDIDENTIFIERSET[2*SS]]+...+CHARLIST.0[DEFINED
IDENTIFIERSET[2*SS+1]]≠CHARLIST.0[FINDID1.0]+...+CHARLIST.0[FINDID2.0];
ASSERT IDENTSTR1=FINDID1.0;
ASSERT IDENTSTR2=FINDID2.0;
ASSERT 0≤AT≤DEFINEDIDENTIFIERSETPT+1;
ASSERT DEFINEDIDENTIFIERSET=DEFINEDIDENTIFIERSET.0;
ASSERT DEFINEDIDENTIFIERSETPT=DEFINEDIDENTIFIERSETPT.0;
ASSERT CHARLIST=CHARLIST.0;
WHILE AT≤DEFINEDIDENTIFIERSETPT DO
      IDENTSTR3:=DEFINEDIDENTIFIERSET[2*AT];
      IDENTST43:=DEFINEDIDENTIFIERSET[2*AT+1];
      ENTER IDENTSTR;
      IF IDENTSTRFLAG
           THEN FOUND:=TRUE;
                RETURN;
      FI;
      AT:=AT+1;
ELIHW;
FOUND:=FALSE;
ASSERT FOUND → 0≤AT≤DEFINEDIDENTIFIERSETPT.0 ∧ CHARLIST.0[DEFINEDIDENTIFIERSET.0
[2*AT]]+...+CHARLIST.0[DEFINEDIDENTIFIERSET.0[2*AT+1]]=CHARLIST.0[FINDID1.0]+...+C
HARLIST.0[FINDID2.0];
ASSERT ¬FOUND ∧ 0≤SS≤DEFINEDIDENTIFIERSETPT.0 → CHARLIST.0[DEFINEDIDENTIFIERSET.
0[2*SS]]+...CHARLIST.0[DEFINEDIDENTIFIERSET.0[2*SS+1]]≠CHARLIST.0[FINDID1.0]+...+
CHARLIST.0[FINDID2.0];
EXIT;

PROCEDURE  FINDLABEL ;
ASSERT 0≤FINDLABEL1≤FINDLABEL2≤CHARLISTPT;
ASSERT LABELTABLEPT>-1;
ASSERT 0≤SS≤LABELTABLEPT → 0≤LABELTABLE[3*SS]+LABELTABLE[3*SS+1]≤CHARLISTPT;
IDENTSTR1 := FINDLABEL1 ;
IDENTSTR2 := FINDLABEL2 ;
AT:=0;
ASSERT 0≤SS≤AT-1 → CHARLIST.0[LABELTABLE.0[3*SS]]+...+CHARLIST.0[LABELTABLE.0[3*
SS+1]]≠CHARLIST.0[FINDLABEL1.0]+...+CHARLIST.0[FINDLABEL2.0];
ASSERT 0≤AT≤LABELTABLEPT.0+1;
ASSERT IDENTSTR1=FINDLABEL1.0 ;
ASSERT IDENTSTR2=FINDLABEL2.0 ;
ASSERT LABELTABLE=LABELTABLE.0;
ASSERT LABELTABLEPT=LABELTABLEPT.0;
ASSERT FINDLABEL1=FINDLABEL1.0;
ASSERT FINDLABEL2=FINDLABEL2.0;
ASSERT CHARLIST=CHARLIST.0 ;
WHILE AT≤LABELTABLEPT DO
      IDENTSTR3:=LABELTABLE[3*AT];
      IDENTSTR4:=LABELTABLE[3*AT+1];
      ENTER IDENTSTR ;
      IF IDENTSTRFLAG
           THEN FOUND := TRUE ;
```

```
                RETURN ;
     FI ;
     AT:=AT+1;
ELIHW ;
FOUND := FALSE ;
ASSERT  FOUND → 0≤AT≤LABELTABLEPT.0 ∧ CHARLIST.0[LABELTABLE.0[3*AT]],...,CHARLIS
T.0[LABELTABLE.0[3*AT+1]]=CHARLIST.0[FINDLABEL1.0],...,CHARLIST.0[FINDLABEL2.0];
ASSERT ¬FOUND ∧ 0≤$$≤LABELTABLEPT.0 → CHARLIST.0[LABELTABLE.0[3*$$]],...,CHARLIS
T.0[LABELTABLE.0[3*$$+1]]≠CHARLIST.0[FINDLABEL1.0],...,CHARLIST.0[FINDLABEL2.0];
EXIT ;

PROCEDURE  FINDRESWORD ;
ASSERT 0≤TOKENSTRING[0]≤TOKENSTRING[1]≤CHARLISTPT ;
ASSERT CONSTANTS(PRESET,RESERVEDWORDSET,RESWORDPTS,RESCODE,LOOP,DASHES,DOTS,ASRT
TRUE,BLANKLINE,PATHIS,LINE1,LINE2,LINE3,LINE4,COLONWORDSET,COLONPTS,LVLLINE);
AT:=0 ;
ASSERT 0≤$$≤AT-1 → RESERVEDWORDSET.0[RESWORDPTS.0[$$]],...,RESERVEDWORDSET.0[RES
WORDPTS.0[$$+1]-1]≠CHARLIST.0[TOKENSTRING.0[0]],...,CHARLIST.0[TOKENSTRING.0[1]]
;
ASSERT 0≤AT≤27;
ASSERT CONSTANTS(PRESET,RESERVEDWORDSET,RESWORDPTS,RESCODE,LOOP,DASHES,DOTS,ASRT
TRUE,BLANKLINE,PATHIS,LINE1,LINE2,LINE3,LINE4,COLONWORDSET,COLONPTS,LVLLINE);
ASSERT TOKENSTRING=TOKENSTRING.0 ;
ASSERT CHARLIST=CHARLIST.0 ;
WHILE AT ≤ 26 DO
     IF RESWORDPTS[AT+1]-RESWORDPTS[AT]-1=TOKENSTRING[1]-TOKENSTRING[0]
          THEN J:=0 ;
ASSERT 0≤$$≤AT-1 → RESERVEDWORDSET.0[RESWORDPTS.0[$$]],...,RESERVEDWORDSET.0[RES
WORDPTS.0[$$+1]-1]≠CHARLIST.0[TOKENSTRING.0[0]],...,CHARLIST.0[TOKENSTRING.0[1]]
;
ASSERT RESERVEDWORDSET.0[RESWORDPTS.0[AT]],...,RESERVEDWORDSET.0[RESWORDPTS.0[AT
]+J-1]=CHARLIST.0[TOKENSTRING.0[0]],...,CHARLIST.0[TOKENSTRING.0[0]+J-1];
ASSERT RESWORDPTS.0[AT+1]-RESWORDPTS.0[AT]-1=TOKENSTRING.0[1]-TOKENSTRING.0[0];
ASSERT 0≤J≤TOKENSTRING.0[1]-TOKENSTRING.0[0]+1;
ASSERT 0≤AT≤26;
ASSERT CONSTANTS(PRESET,RESERVEDWORDSET,RESWORDPTS,RESCODE,LOOP,DASHES,DOTS,ASRT
TRUE,BLANKLINE,PATHIS,LINE1,LINE2,LINE3,LINE4,COLONWORDSET,COLONPTS,LVLLINE);
ASSERT TOKENSTRING=TOKENSTRING.0 ;
ASSERT CHARLIST=CHARLIST.0 ;
                WHILE J≤TOKENSTRING[1]-TOKENSTRING[0] ∧
                      CHARLIST[TOKENSTRING[0]+J]=
                      RESERVEDWORDSET[RESWORDPTS[AT]+J] DO
                     J:=J+1 ;
                ELIHW ;
                IF J>TOKENSTRING[1]-TOKENSTRING[0]
                     THEN FOUND:=TRUE ;
                          RETURN ;
                FI ;
     FI ;
     AT:=AT+1 ;
ELIHW ;
FOUND:=FALSE ;
ASSERT FOUND → RESERVEDWORDSET.0[RESWORDPTS.0[AT]],...,RESERVEDWORDSET.0[RESWORD
PTS.0[AT+1]-1]=CHARLIST.0[TOKENSTRING.0[0]],...,CHARLIST.0[TOKENSTRING.0[1]] ∧ 0
≤AT≤26;
ASSERT ¬FOUND ∧ 0≤$$≤26 → RESERVEDWORDSET.0[RESWORDPTS.0[$$],...,RESERVEDWORDSET
.0[RESWORDPTS.0[$$+1]-1]≠CHARLIST.0[TOKENSTRING.0[0]],...,CHARLIST.0[TOKENSTRING
.0[1]];
EXIT ;

PROCEDURE  FINISHPATH ;
ASSERT ISPATHFRONT(PATH,PATHPT,BRANCH,CURRENTPROC);
```

```
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT PATH[0],...,PATH[PATHPT] CONTAINS NO UNTAGGED LOOPS;
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT);
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
IF STATEMENT[DESCLOC[PATH[PATHPT]+BIAS]]=85      $IF$
 v STATEMENT[DESCLOC[PATH[PATHPT]+BIAS]]=72      $CASE$
      THEN BRANCH[PATHPT]:=1;
      ELSE BRANCH[PATHPT]:=0;
FI;
ENTER GENNEXTPATHPT ;
ENTER LOOPCHECK ;
ENTER GENTERM ;
PATHPT := PATHPT + 1 ;
IF  STATEMENT[PATH[PATHPT]+BIAS]=80      $EXIT$
  v STATEMENT[PATH[PATHPT]+BIAS]=84      $HALT$
  v ASRTLOC[PATH[PATHPT]+BIAS]≥0     $HAVE ASSERTION$
      THEN ENTER WRITESTEPLINE];
           WRITE DASHES;
           ENTER WRITEASRTS;
           WRITE BLANKLINE;
           ENTER PRINTPATH;
      ELSE ENTER FINISHPATH;
FI ;
ASSERT THE CURRENTPROC $PATHS IN PATH,PATHPT,AND BRANCH IS THE IMMEDIATE SUCCESS
OR OF THE CURRENTPROC PATHFRONT IN PATH.0, PATHPT.0, AND BRANCH.0;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U VERIFICATION CONDITION TERMS FOR PA
TH[PATHPT.0],...,PATH[PATHPT];
ASSERT LINEPT=0 ^ LINE[0]=+F;
EXIT ;

PROCEDURE  FROMPOINTVCGEN ;
ASSERT LINEPT=0 ^ LINE[0]=+F;
ENTER RESETALT ;
NSTEP:=0;
PATH[0]:=CURRFRONT;
PATHPT := 0 ;
ENTER FINISHPATH ;
ENTER GENSUCCESSORS ;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U VERIFICATION CONDITIONS IN PROCEDUR
E CURRENTPROC FROM POINT CURRFRONT;
ASSERT LINEPT=0 ^ LINE[0]=+F;
EXIT ;

PROCEDURE  GENASSIGNTERM ;
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT);
ASSERT STMT=DESCLOC[PATH[PATHPT]+BIAS];
ASSERT STATEMENT[STMT]=64;
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
IF STATEMENT[STMT+1]=2   $ARRAYREF ON LEFT$
     THEN ENTER INSERTPRV;
          CHAR:=+0;
          ENTER INSERTCHAR;
          CHAR:=+≤;
          ENTER INSERTCHAR;
          SUBFRONT:=STATEMENT[STMT+3];
          SUBBACK:=STATEMENT[STMT+4];
          ENTER PRINTSUBEXP;
          CHAR:=+≤;
          ENTER INSERTCHAR;
          INTVAL:=BOUNDFUNCTION(STATEMENT[STMT+2]);
          ENTER INTPRINT;
```

```
            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 INTPRINT;
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;
ALTNUM[STATEMENT[STMT+2]]:=ALTNUM[STATEMENT[STMT+2]]+1;
COLONALTNUM[1]:=COLONALTNUM[1]+1;      $:LOC$
NSTEP:=NSTEP+1;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U VERIFICATION CONDITION TERM FOR STA
TEMENT AT PATH[PATHPT];
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT+1);
ASSERT LINEPT=0 ^ LINE[0]=↑F;
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
EXIT ;

PROCEDURE GENCASETERM;
ASSERT LINEPT=0 ^ LINE[0]=↑F;
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT);
ASSERT STMT=DESCLOC[PATH[PATHPT]+BIAS];
ASSERT STATEMENT[STMT]=72;
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]+1≥BRANCH[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:=↑);
```

```
            ENTER INSERTCHAR;
      ELSE CHAR:=+F;
            ENTER INSERTCHAR;
            CHAR:=+(;
            ENTER INSERTCHAR;
            FRONTLABEL:=CASELABELFRONT[CASELABELSET[STATEMENT[STMT+4]]];
             BACKLABEL:=CASELABELFRONT[CASELABELSET[STATEMENT[STMT+5]+1]]-1;
            ENTER PRINTCASELABELS;
            CHAR:=+);
            ENTER INSERTCHAR;
FI;
ENTER PRINTLINE;
COLONALTNUM[1]:=COLONALTNUM[1]+1;
NSTEP:=NSTEP+1;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U VERIFICATION CONDITION TERM FOR STA
TEMENT AT PATH[PATHPT];
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT+1);
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
EXIT ;

PROCEDURE  GENENTERTERM ;
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT);
ASSERT STMT=DESCLOC[PATH[PATHPT]+BIAS];
ASSERT STATEMENT[STMT]=78;
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
WRITE LVLLINE;
ENTER WRITERTNPTLINE;
ENTER WRITESTEPLINE1;
FINDID1:=STATEMENT[STMT+1];
FINDID2:=STATEMENT[STMT+2];
ENTER FINDID;
PROCCALLED:=0;
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT);
ASSERT AT IS IDENTIFIER NUMBER OF CALLED PROCEDURE;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U LVLLINE U RTNPTLINE U STEPLINE1;
ASSERT NSTEP=0;
WHILE PROC[7*PROCCALLED]≠AT  DO
      PROCCALLED:=PROCCALLED+1;
ELIHW;
COLONALTNUM[ 1]:=COLONALTNUM[ 1]+1;  $:LOC$
COLONALTNUM[ 2]:=COLONALTNUM[ 2]+1;  $:LVL$
COLONALTNUM[ 5]:=COLONALTNUM[ 5]+1;  $:RTNPT$
COLONALTNUM[10]:=COLONALTNUM[10]+1;  $:STEP$
IF ASRTLOC[PROC[7*PROCCALLED+1]]>0
     THEN G:=ASRTLOC1[ASRTLOC[PROC[7*PROCCALLED+1]]];
          ASRTFLAG:=1;
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT ALTERATION COUNTERS IN ALTNUM AND COLONALTNUM HAVE BEEN ACCUMULATED THROU
GH ENTRY OF CALLED PROCEDURE;
ASSERT ASRTFLAG=1 ^ IN INITIAL ASSERTION OF ENTER STATEMENT;
ASSERT PROCEDURE NUMBER PROCCALLED IS THE CALLED PROCEDURE;
ASSERT POINT 0 OF PROCCALLED IS TAGGED WITH ASSERTIONS;
ASSERT ASRTLOC1[ASRTLOC[PROC[7*PROCCALLED+1]]]≤G≤ASRTLOC1[ASRTLOC[PROC[7*PROCCAL
LED+1]]+1];
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U LVLLINE U RTNPTLINE U STEPLINE1 U S
UBSCRIPTED POINT 0 ASSERTIONS THROUGH ASSERTION G-1;
ASSERT NSTEP=0;
          WHILE G<ASRTLOC1[ASRTLOC[PROC[7*PROCCALLED+1]]+1]  DO
                ASRTFRONT:=ASRTS[2*G];
```

```
                ASRTSCANPOINTER:=ASRTFRONT;
                ASRTBACK:=ASRTS[2*G+1];
                ENTER INSERTPRV;
                ENTER WRITENEXTASRT;
                G:=G+1;
            ELIHW;
FI ;
ENTER UPDATEALTNUM;
IF ASRTLOC[PROC[7*PROCCALLED+2]]≥0
     THEN G:=ASRTLOC1[ASRTLOC[PROC[7*PROCCALLED+2]]];
          ASRTFLAG:=2;
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT ALTERATION COUNTERS IN ALTNUM AND COLONALTNUM HAVE BEEN ACCUMULATED THROU
GH EXECUTION OF CALLED PROCEDURE;
ASSERT ASRTFLAG=2 ^ IN FINAL ASSERTION OF ENTER STATEMENT;
ASSERT PROCEDURE NUMBER PROCALLED IS THE CALLED PROCEDURE;
ASSERT EXIT POINT OF PROCCALLED IS TAGGED WITH ASSERTIONS;
ASSERT ASRTLOC1[ASRTLOC[PROC[7*PROCCALLED+2]]]≤G≤ASRTLOC1[ASRTLOC[PROC[7*PROCCAL
LED+2]]+1];
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U LVLLINE U RTNPTLINE U STEPLINE1 U S
UBSCRIPTED POINT 0 ASSERTIONS U SUBSCRIPTED EXIT POINT ASSERTIONS THROUGH ASSERT
ION G-1;
ASSERT NSTEP=0;
            WHILE G<ASRTLOC1[ASRTLOC[PROC[7*PROCCALLED+2]]+1]  DO
                ASRTFRONT:=ASRTS[2*G];
                ASRTSCANPOINTER:=ASRTFRONT;
                ASRTBACK:=ASRTS[2*G+1];
                ENTER WRITENEXTASRT;
                G:=G+1;
            ELIHW;
FI;
ENTER WRITESTEPLINE2;
COLONALTNUM[ 1]:=COLONALTNUM[ 1]+1;
COLONALTNUM[ 2]:=COLONALTNUM[ 2]+1;
COLONALTNUM[ 5]:=COLONALTNUM[ 5]+1;
COLONALTNUM[10]:=COLONALTNUM[10]+1;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U VERIFICATION CONDITION TERM FOR STA
TEMENT AT PATH[PATHPT];
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT+1);
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT NSTEP=0;
EXIT ;

PROCEDURE GENIFTERM;
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT);
ASSERT STMT=DESCLOC[PATH[PATHPT]+BIAS];
ASSERT STATEMENT[STMT]=85;
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
EXPFRONT:=STATEMENT[STMT+1];
EXPBACK:=STATEMENT[STMT+2];
ENTER EXPCHECKER;
IF BRANCH[PATHPT]=2
     THEN CHAR:=+¬;
          ENTER INSERTCHAR;
          CHAR:=+(;
          ENTER INSERTCHAR;
FI;
SUBFRONT:=STATEMENT[STMT+1];
SUBBACK:=STATEMENT[STMT+2];
ENTER PRINTSUBEXP;
IF BRANCH[PATHPT]=2
```

```
     THEN CHAR:=+);
          ENTER INSERTCHAR;
FI;
ENTER PRINTLINE;
COLONALTNUM[1]:=COLONALTNUM[1]+1;
NSTEP:=NSTEP+1;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U VERIFICATION CONDITION TERM FOR STA
TEMENT AT PATH[PATHPT];
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT+1);
ASSERT LINEPT=0 ∧ LINE[0]=+F;
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
EXIT ;

PROCEDURE  GENNEXTPATHPT ;
ASSERT PATH[PATHPT] IS NOT HALT OR EXIT;
CASE STATEMENT[DESCLOC[PATH[PATHPT]+BIAS]]  OF
     98:91:78:64:      $WRITE,READ,ENTER,ASSIGN$
          PATH[PATHPT+1]:=PATH[PATHPT]+1;
     95:      $JUMP$
          PATH[PATHPT+1]:=STATEMENT[DESCLOC[PATH[PATHPT]+BIAS]+1];
     85:      $2+$
          IF BRANCH[PATHPT]=1
             THEN PATH[PATHPT+1]:=STATEMENT[DESCLOC[PATH[PATHPT]+BIAS]+3];
             ELSE PATH[PATHPT+1]:=STATEMENT[DESCLOC[PATH[PATHPT]+BIAS]+4];
          FI ;
     72:      $CASE$
          IF BRANCH[PATHPT]<STATEMENT[DESCLOC[PATH[PATHPT]+BIAS]+5]
                          -STATEMENT[DESCLOC[PATH[PATHPT]+BIAS]+4]+2
             THEN PATH[PATHPT+1]:=CASELABELSET[STATEMENT[DESCLOC[PATH[PATHPT]+
                  BIAS]+4]+BRANCH[PATHPT]-1];
             ELSE PATH[PATHPT+1]:=STATEMENT[DESCLOC[PATH[PATHPT]+BIAS]+6];
          FI ;
ESAC ;
ASSERT PATH[PATHPT+1] IS NEXTPOINT FOR PATH.0[0],...,PATH.0[PATHPT.0];
ASSERT 0≤$S≤PATHPT.0 ⊃ PATH[$S]=PATH.0[$S];
EXIT ;

PROCEDURE GENREADTERM;
ASSERT LINEPT=0 ∧ LINE[0]=+F;
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT);
ASSERT STMT=DESCLOC[PATH[PATHPT]+BIAS];
ASSERT STATEMENT[STMT]=91;
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
ID:=STATEMENT[STMT+1];
B:=BOUNDFUNCTION[ID];
ENTER WRITEREADLINE1;
ENTER WRITEREADLINE2;
ENTER WRITEREADLINE3;
COLONALTNUM[1]:=COLONALTNUM[1]+1;
COLONALTNUM[3]:=COLONALTNUM[3]+1;
NSTEP:=NSTEP+1;
ALTNUM[ID]:=ALTNUM[ID]+1;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U VERIFICATION CONDITION TERM FOR STA
TEMENT AT PATH[PATHPT];
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT+1);
ASSERT LINEPT=0 ∧ LINE[0]=+F;
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
EXIT ;

PROCEDURE GENSUCCESSORS ;
ASSERT LINEPT=0 ∧ LINE[0]=+F;
ASSERT IMMEDIATE SUCCESSOR OF $PATH$ THROUGH PATH,PATHPT,BRANCH IS THE IMMEDIATE
```

```
 SUCCESSOR OF THE $PATH$ THROUGH PATH.0,PATHPT.0,BRANCH.0;
ASSERT PATH[0],...,PATH[PATHPT] CONTAINS NO UNTAGGED LOOPS;
WHILE PATHPT>0 DO
PATHPT := PATHPT-1 ;
ENTER ISANOTHERBRANCH ;
IF ANOTHERBRANCH
     THEN ENTER REGEN ;
          ENTER FINISHPATH ;
     ENTER GENSUCCESSORS ;
          RETURN;
FI ;
ELIHW ;
ASSERT WRITEFILE(:WTHD) = WRITEFILE(:WTHD.0) U VERIFICATION CONDITIONS IN CURREN
TPROC FROM PATH[0] FOLLOWING $PATH$ PATH.0,PATHPT.0,BRANCH.0;
ASSERT LINEPT=0 ^ LINE[0]=+F;
EXIT ;

PROCEDURE  GENTERM ;
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT);
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
IF PATHPT=0
     THEN WRITE BLANKLINE ;
          WRITE BLANKLINE ;
          ID:=PROC[7*CURRENTPROC];
          ENTER PRINTID;
          ENTER PRINTLINE;
          WRITE BLANKLINE;
          IF CURRENTPROC=INITIALPROCEDURE ^ CURRFRONT=0
               THEN WRITE LINE1 ;
                    WRITE LINE2 ;
                    WRITE LINE3 ;
                    WRITE LINE4 ;
          FI ;
          ENTER WRITEASRTS;
WRITE DOTS ;
FI ;
STMT:=DESCLOC[PATH[PATHPT]+BIAS];
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT);
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
ASSERT PATHPT.0=0 → WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U VERIFICATION CONDITION
 THROUGH DOTS;
ASSERT PATHPT.0≠0 → :WTHD=:WTHD.0;
ASSERT STMT=DESCLOC[PATH[PATHPT]+BIAS];
CASE STATEMENT[STMT]  OF
     64: ENTER GENASSIGNTERM ;
     95:  COLONALTNUM[1]:=COLONALTNUM[1]+1;  $:LOC$
          NSTEP:=NSTEP+1;
     78: ENTER GENENTERTERM ;
     91: ENTER GENREADTERM ;
     98: ENTER GENWRITETERM ;
     85: ENTER GENIFTERM ;
     72: ENTER GENCASETERM ;
ESAC ;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U VERIFICATION CONDITION TERM FOR STA
TEMENT AT PATH[PATHPT];
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT+1);
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
EXIT ;
```

```
PROCEDURE GENWRITETERM;
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT);
ASSERT STMT=DESCLOC(PATH[PATHPT]+BIAS);
ASSERT STATEMENT[STMT]=98;
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
ID:=STATEMENT[STMT+1];
B:=BOUNDFUNCTION(ID);
ENTER WRITEWRITELINE1;
ENTER WRITEWRITELINE2;
ENTER WRITEWRITELINE3;
COLONALTNUM[1]:=COLONALTNUM[1]+1;
COLONALTNUM[4]:=COLONALTNUM[4]+1;
NSTEP:=NSTEP+1;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U VERIFICATION CONDITION TERM FOR STA
TEMENT AT PATH(PATHPT);
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT+1);
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
EXIT ;

PROCEDURE IDENTSTR ;
IF IDENTSTR1=IDENTSTR3 ^ IDENTSTR2=IDENTSTR4
      THEN IDENTSTRFLAG := TRUE ;
           RETURN ;
FI ;
IF IDENTSTR2-IDENTSTR1 ≠ IDENTSTR4-IDENTSTR3 v
   IDENTSTR1>IDENTSTR2 v IDENTSTR3>IDENTSTR4
      THEN IDENTSTRFLAG := FALSE ;
           RETURN ;
FI ;
I := 0 ;
ASSERT CHARLIST.0[IDENTSTR1.0],....,CHARLIST.0[IDENTSTR1.0+I-1]=CHARLIST.0[IDENTS
TR3.0],....,CHARLIST.0[IDENTSTR3.0+I-1];
ASSERT 0≤I≤IDENTSTR2-IDENTSTR1+1 ;
ASSERT IDENTSTR4-IDENTSTR3=IDENTSTR2-IDENTSTR1 ;
ASSERT IDENTSTR1.0≤IDENTSTR2.0;
ASSERT IDENTSTR3.0≤IDENTSTR4.0;
ASSERT CHARLIST=CHARLIST.0;
ASSERT IDENTSTR1=IDENTSTR1.0;
ASSERT IDENTSTR2=IDENTSTR2.0;
ASSERT IDENTSTR3=IDENTSTR3.0;
ASSERT IDENTSTR4=IDENTSTR4.0;
WHILE I ≤ IDENTSTR2-IDENTSTR1 DO
      IF CHARLIST[IDENTSTR1 + I] ≠ CHARLIST[IDENTSTR3 + I]
           THEN IDENTSTRFLAG := FALSE ;
                RETURN ;
      FI;
      I:=I+1;
ELIHW;
IDENTSTRFLAG:=TRUE;
ASSERT IDENTSTRFLAG IFF CHARLIST.0[IDENTSTR1.0],....,CHARLIST.0[IDENTSTR2.0]=CHAR
LIST.0[IDENTSTR3.0],....,CHARLIST.0[IDENTSTR4.0];
EXIT;

PROCEDURE  INPUT ;
ASSERT ¬ RECOGNITION ;
ASSERT [¬ASRTSCANFLAG ^ ATSCANSTATE ^ ATSAVEDPARSESTATE] v [¬ASRTSCANFLAG ^ ATPA
RSESTATE] v [ASRTSCANFLAG ^ ATASRTSCANSTATE] ;
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT WRITEFILE(:WTHD)=LISTING OF INPUTSTRING THROUGH :RDHD ;
IF 19≤STATE ^ STATE≤108                    $ I.E., IN PARSE NETWORK. $
```

```
     THEN ENTER SAVEPARSESTATUS ;
          ENTER SCAN ;
          CARINSTRING := TOKEN ;
          ENTER 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]) ;
                                        ENTER LIST;
                    FI ;
               ELSE CARINSTRING := INTEGER(CARD[COL]) ;
                    COL := COL + 1 ;
          FI ;
               ELSE IF ASRTSCANPOINTER≤ASRTBACK
                         THEN CARINSTRING:=INTEGER(CHARLIST[ASRTSCANPOINTER]);
                              ASRTSCANPOINTER:=ASRTSCANPOINTER+1;
                         ELSE CARINSTRING:=99;
     FI ;
FI ;
FI ;
ASSERT ¬ RECOGNITION ;
ASSERT CARINSTRING=CAR(INSTRING.0);
ASSERT INSTRING=CDR(INSTRING.0);
ASSERT LINEPT=0 ^ LINE[0]=↑F;
ASSERT WRITEFILE(:WTHD)=LISTING OF INPUTSTRING THROUGH :RDHD ;
EXIT ;

PROCEDURE  INSERTCHAR ;
ASSERT LINE[0]=↑F;
CHARLIST[CHARLISTPT]:=CHAR ;
LINEFRONT:=CHARLISTPT ;
LINEBACK:=CHARLISTPT ;
ENTER BUILDLINE ;
ASSERT WRITEFILE(:WTHD) U LINE[0],...,LINE[LINEPT]=WRITEFILE(:WTHD.0) U LINE.0[0]
,...,LINE.0[LINEPT.0] U CHAR.0 ;
ASSERT LINE[0]=↑F;
ASSERT 0≤$$≤CHARLISTPT-1 → CHARLIST[$$]=CHARLIST.0[$$];
EXIT ;

PROCEDURE INSERTFIRSTCHAR;
CHARLISTPT:=CHARLISTPT+1;
CHARLIST[CHARLISTPT]:=CHARACTER(NEXTCHARACTER);
TOKENSTRING[0]:=CHARLISTPT;
TOKENSTRING[1]:=CHARLISTPT;
ASSERT TOKENSTRING=CHARACTER(NEXTCHARACTER);
ASSERT CHARLISTPT=CHARLISTPT.0+1;
EXIT ;

PROCEDURE INSERTPRV;
ASSERT LINEPT=0 ^ LINE[0]=↑F;
LINEFRONT:=33;
LINEBACK:=38;
ENTER BUILDFROMPRESET;
ASSERT LINEPT=6;
ASSERT LINE[0],...,LINE[6]=↑F<PRV> ↑ ;
ASSERT :WTHD=:WTHD.0;
EXIT ;

PROCEDURE  INTPRINT ;
ASSERT INTVAL≥0;
```

```
ASSERT LINE[0]=+F;
IF INTVAL=0
     THEN CHAR:=+0 ;
          ENTER INSERTCHAR ;
     ELSE
          LINEFRONT:=CHARLISTPT ;
          LINEBACK:=CHARLISTPT-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 :WTHD=:WTHD.0;
          WHILE INTVAL≠0 DO
               LINEBACK:=LINEBACK+1 ;
               CHARLIST[LINEBACK]:=CHARACTER(INTVAL+10+27);
               INTVAL:=INTVAL/10 ;
          ELIHW ;
ASSERT CHARLIST[LINEFRONT],...,CHARLIST[LINEBACK] CONTAIN RIGHTMOST (LINEBACK-LIN
EFRONT+1) DIGITS OF INTVAL.0 IN REVERSE ORDER;
ASSERT WRITEFILE(:WTHD) U LINE[1],...,LINE[LINEPT]=WRITEFILE(:WTHD.0) U LINE.0[1]
,...,LINE.0[LINEPT.0] U DIGITS OF INTVAL.0 EXCEPT THE RIGHTMOST (LINEBACK-LINEFRO
NT+1) DIGITS;
ASSERT LINEBACK≥LINEFRONT-1;
ASSERT LINE[0]=LINE.0[0];
          WHILE LINEBACK≥LINEFRONT  DO
               CHAR:=CHARLIST[LINEBACK];
               ENTER INSERTCHAR;
               LINEBACK:=LINEBACK-1;
          ELIHW;
FI ;
ASSERT WRITEFILE(:WTHD) U LINE[0],...,LINE[LINEPT]=WRITEFILE(:WTHD.0) U LINE.0[0]
,...,LINE.0[LINEPT.0] U DIGITS OF INTVAL.0;
ASSERT LINE[0]=+F;
EXIT ;

PROCEDURE  ISANOTHERBRANCH ;
IF BRANCH[PATHPT]=0
     THEN ANOTHERBRANCH:=FALSE;
     ELSE IF STATEMENT[DESCLOC[PATH[PATHPT]+BIAS]]≠72  $CASE$
               THEN IF BRANCH[PATHPT]=1
                    THEN ANOTHERBRANCH:=TRUE;
                    ELSE ANOTHERBRANCH:=FALSE;
                    FI;
               ELSE IF STATEMENT[DESCLOC[PATH[PATHPT]+BIAS]+5]
                     - STATEMENT[DESCLOC[PATH[PATHPT]+BIAS]+4]+2 >BRANCH[PATHPT]
                         THEN ANOTHERBRANCH := TRUE ;
                         ELSE ANOTHERBRANCH := FALSE ;
                    FI ;
          FI;
FI;
ASSERT ANOTHERBRANCH IFF THERE IS ANOTHER BRANCHPOINT FROM PATH.0[PATHPT.0];
EXIT ;

PROCEDURE ISCOLONID;
AT:=1;
ASSERT 1≤AT≤11;
ASSERT 1≤$$≤AT-1 → TOKEN.STRING ≠ COLON IDENTIFIER $$;
WHILE AT≤10  DO
     IF COLONPTS[AT+1]-COLONPTS[AT]=TOKENSTRING[1]-TOKENSTRING[0]
          THEN I:=0;
ASSERT 1≤AT≤10;
```

```
ASSERT 1≤SS≤AT-1 → TOKEN.STRING ≠ COLON IDENTIFIER SS;
ASSERT 0≤I≤LENGTH OF TOKEN.STRING+1;
ASSERT 0≤SS≤I-1 → LETTER SS+1 OF TOKEN.STRING = LETTER SS+1 OF COLON IDENTIFIER
 -AT;
              WHILE I≤TOKENSTRING[1]-TOKENSTRING[0] ∧
                   CHARLIST[TOKENSTRING[0]+I]=COLONWORDSET[COLONPTS[AT]+I]  DO
                   I:=I+1;
              ELIHW;
              IF I>TOKENSTRING[1]-TOKENSTRING[0]
                   THEN FOUND:=TRUE;
                        AT:=-AT;
                        RETURN;
              FI;
     FI;
     AT:=AT+1;
ELIHW;
FOUND:=FALSE;
ASSERT  FOUND → TOKEN.STRING IS COLON IDENTIFIER AT;
ASSERT ¬FOUND → TOKEN.STRING IS NOT A COLON IDENTIFIER;
EXIT ;

PROCEDURE  LABELREFCHECK ;
ASSERT 0≤TOKENSTRING[0]≤TOKENSTRING[1]≤CHARLISTPT;
ASSERT REFERENCEDLABELSETPT>=1;
ASSERT 0≤SS≤2*REFERENCEDLABELSETPT+1 → 0≤REFERENCEDLABELSET[SS]≤CHARLISTPT;
IDENTSTR1:=TOKENSTRING[0];
IDENTSTR2:=TOKENSTRING[1];
AT:=0;
ASSERT 0≤SS≤AT-1 → CHARLIST.0[REFERENCEDLABELSET.0[2*SS]],...,CHARLIST.0[REFEREN
CEDLABELSET.0[2*SS+1]]≠CHARLIST.0[TOKENSTRING.0[0]],...,CHARLIST.0[TOKENSTRING.0
[1]];
ASSERT 0≤AT≤REFERENCEDLABELSETPT.0+1;
ASSERT IDENTSTR1=TOKENSTRING.0[0];
ASSERT IDENTSTR2=TOKENSTRING.0[1];
ASSERT REFERENCEDLABELSET=REFERENCEDLABELSET.0;
ASSERT REFERENCEDLABELSETPT=REFERENCEDLABELSETPT.0 ;
ASSERT CHARLIST=CHARLIST.0 ;
ASSERT TOKENSTRING=TOKENSTRING.0 ;
WHILE AT≤REFERENCEDLABELSETPT DO
     IDENTSTR3:=REFERENCEDLABELSET[2*AT];
     IDENTSTR4:=REFERENCEDLABELSET[2*AT+1];
     ENTER IDENTSTR ;
     IF IDENTSTRFLAG
          THEN FOUND := TRUE ;
               RETURN ;
     FI ;
     AT:=AT+1;
ELIHW ;
FOUND := FALSE ;
ASSERT  FOUND → 0≤AT≤REFERENCEDLABELSETPT.0 ∧ CHARLIST.0[REFERENCEDLABELSET.0[2*
AT]],...,CHARLIST.0[REFERENCEDLABELSET.0[2*AT+1]]=CHARLIST.0[TOKENSTRING.0[0]],.
..,CHARLIST.0[TOKENSTRING.0[1]];
ASSERT ¬FOUND ∧ 0≤SS≤REFERENCEDLABELSETPT.0 → CHARLIST.0[REFERENCEDLABELSET.0[2*
SS]],...,CHARLIST.0[REFERENCEDLABELSET.0[2*SS+1]]≠CHARLIST.0[TOKENSTRING.0[0]],.
..,CHARLIST.0[TOKENSTRING.0[1]];
EXIT ;

PROCEDURE LIST;
     $PRINTS CURRENT CONTROL POINT FOLLOWED BY CARD IMAGE $
ASSERT ATSAVEDPARSESTATE ;
ASSERT LINEPT=0 ∧ LINE[0]=↑F;
ASSERT CARD[0]≠↑T;
```

```
ASSERT WRITEFILE(:WTHD)=LISTING OF INPUTSTRING UP TO :RDHD;
IF PARSESTATE≥41 ∨ PARSESTATE=36
      THEN INTVAL:=POINT;
           ENTER INTPRINT;
           CHAR:=+ ;
           IF POINT<10
                THEN ENTER INSERTCHAR;
           FI;
           ENTER INSERTCHAR;
           ENTER INSERTCHAR;
      ELSE CHAR:=+ ;
           ENTER INSERTCHAR;
           ENTER INSERTCHAR;
           ENTER INSERTCHAR;
           ENTER INSERTCHAR;
FI;
I:=1;
ASSERT 1≤I≤81;
ASSERT CARD[0]≠+T;
ASSERT LINE[0]=+F;
ASSERT LINE[1],...,LINE[LINEPT]= LISTING LABEL;
ASSERT LINE[LINEPT+1],...,LINE[LINEPT+I-1]=CARD[1],...,CARD[I-1];
ASSERT WRITEFILE(:WTHD.0)=LISTING OF INPUTSTRING UP TO :RDHD;
WHILE I≤80  DO
      LINE[I+LINEPT]:=CARD[I];
      I:=I+1;
ELIHW;
LINEPT:=LINEPT+80;
ENTER PRINTLINE;
ASSERT WRITEFILE(:WTHD)=LISTING OF INPUTSTRING THROUGH :RDHD ;
ASSERT LINEPT=0 ∧ LINE[0]=+F;
EXIT;

PROCEDURE LISTCALLEDPROCS;
I:=0;
ASSERT 0≤I≤200;
ASSERT 0≤SS≤I-1 → ¬PROCFLAG[SS] ∧ ¬PROCFLAG1[SS];
ASSERT PROCCALLED=PROCCALLED.0;
WHILE I≤199  DO
      PROCFLAG[I]:=FALSE;
      PROCFLAG1[I]:=FALSE;
      I:=I+1;
ELIHW;
PROCFLAG[PROCCALLED]:=TRUE;
DONE:=FALSE;
ASSERT DONE → PROCFLAG[SS] IFF PROCEDURE SS CAN BE REACHED BY A CALL TO PROCEDUR
E PROCCALLED.0;
ASSERT PROCFLAG[PROCCALLED.0];
ASSERT PROCFLAG[SS] → PROCEDURE SS CAN BE REACHED BY A CALL TO PROCEDURE PROCCAL
LED.0;
ASSERT PROCFLAG1[SS] → PROCFLAG[SS] ∧ PROCFLAG[*] IF PROCEDURE * CAN BE REACHED
BY A CALL TO PROCEDURE SS;
ASSERT PROCCALLED=PROCCALLED.0;
WHILE ¬DONE  DO
      DONE := TRUE;
      PROCNUM:=0;
 ASSERT 0≤PROCNUM≤200;
 ASSERT DONE ∧ (0≤SS≤PROCNUM-1) ∧ PROCFLAG[SS] → PROCFLAG1[SS];
 ASSERT PROCFLAG[PROCCALLED.0];
 ASSERT PROCCALLED=PROCCALLED.0;
 ASSERT PROCFLAG[SS] → PROCEDURE SS CAN BE REACHED BY A CALL TO PROCEDURE PROCCAL
 LED.0;
```

```
ASSERT PROCFLAG1[$$] → PROCFLAG[$$] ∧ PROCFLAG[*] IF PROCEDURE * CAN BE REACHED
BY A CALL TO PROCEDURE $$;
      WHILE PROCNUM≤199  DO
            IF PROCFLAG[PROCNUM] ∧ ¬PROCFLAG1[PROCNUM]
                  THEN ENTER LISTPROCCALLS;
                        PROCFLAG1[PROCNUM]:=TRUE;
            FI;
            PROCNUM:=PROCNUM+1;
      ELIHW;
ELIHW;
ASSERT PROCFLAG[$$] IFF PROCEDURE $$ CAN BE REACHED BY A CALL TO PROCEDURE PROCC
ALLED.0;
EXIT ;

PROCEDURE LISTPROCCALLS;
I:=PROC[7*PROCNUM+3];
ASSERT PROC[7*PROCNUM.0+3]≤I≤PROC[7*PROCNUM.0+4];
ASSERT PROCNUM=PROCNUM.0;
ASSERT DONE → PROCFLAG=PROCFLAG.0;
ASSERT (PROC[7*PROCNUM.0+3]≤$$≤I-1) ∧ (CALLED PROCEDURE $$ = PROCEDURE *) → PROC
FLAG[*];
WHILE I<PROC[7*PROCNUM+4]  DO
      FINDID1:=PROCCALLS[2*I];
      FINDID2:=PROCCALLS[2*I+1];
      ENTER FINDID;
      K:=0;
ASSERT PROCNUM=PROCNUM.0;
ASSERT PROC[7*PROCNUM.0+3]≤I<PROC[7*PROCNUM.0+4];
ASSERT DONE → PROCFLAG=PROCFLAG.0;
ASSERT (PROC[7*PROCNUM.0+3]≤$$≤I-1) ∧ (CALLED PROCEDURE $$ = PROCEDURE *) → PROC
FLAG[*];
ASSERT CALLED PROCEDURE I = IDENTIFIER AT;
      WHILE PROC[7*K]≠AT  DO
            K:=K+1;
      ELIHW;
      IF ¬PROCFLAG[K]
            THEN DONE:=FALSE;
                  PROCFLAG[K]:=TRUE;
      FI;
      I:=I+1;
ELIHW;
ASSERT DONE → PROCFLAG=PROCFLAG.0;
ASSERT PROCFLAG[$$] IF PROCEDURE $$ CAN BE REACHED BY A CALL TO PROCEDURE PROCNU
M.0;
EXIT ;

PROCEDURE  LOOPCHECK ;
ASSERT PATH[0],...,PATH[PATHPT] CONTAINS NO UNTAGGED LOOPS;
I:=1 ;
ASSERT 1≤I≤PATHPT.0+1;
ASSERT PATH  =PATH.0;
ASSERT PATHPT=PATHPT.0;
ASSERT 1≤$$≤I-1 → PATH.0[$$]≠PATH.0[PATHPT.0+1];
ASSERT :WTHD=:WTHD.0;
WHILE I≤PATHPT DO
      IF PATH[I]=PATH[PATHPT+1]
            THEN WRITE LOOP ;
ASSERT PATH.0[0],...,PATH.0[PATHPT.0+1] CONTAINS AN UNTAGGED LOOP;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U LOOP;
                  HALT ;
            ELSE I:=I+1 ;
      FI ;
```

```
ELIHW ;
ASSERT PATH.0[0],...,PATH.0[PATHPT.0+1] CONTAINS NO UNTAGGED LOOPS;
ASSERT :WTHD=:WTHD.0;
EXIT ;

PROCEDURE  NILMATCH ;
WHILE  ARC < FIRSTARC[STATE+1] ^ SYMBOL[ARC] = 0  DO
      ENTER TESTS ;
            IF TESTFLAG
            THEN NILMATCHFLAG := TRUE ;
                  RETURN ;
      FI ;
      ARC := ARC + 1 ;
ELIHW ;
NILMATCHFLAG := FALSE ;
EXIT ;

PROCEDURE  PARSE ;
ASSERT READFILE(:RDHD) = INPUTSTRING ;
ASSERT CURRENTPROC=-1;
ASSERT STATEMENTPT=-1;
ASSERT ASRTLOCPT=-1;
ASSERT ASRTLOC1[0]=0;
ASSERT 0≤$$≤1999 → ASRTLOC[$$]=-1;
ASSERT EXPLISTPT=-1;
ASSERT EXPSTRING[0]=0;
ASSERT EXPSTRING[1]=-1;
ASSERT CASELABELSETPT=-1;
ASSERT CASELABELFRONT[0]=0;
ASSERT DEFINEDCASELABELSETTOP=-1;
STATE := PARSESTART ;
PARSESTATE:=PARSESTART;
READ CARD ;
LINEPT:=0;
LINE[0]:=↑F;
IF CARD[0]≠↑T
      THEN ENTER LIST;
FI;
COL := 1 ;
RECOGNITION := FALSE ;
ASRTSCANFLAG:=FALSE;
ENTER SCAN ;
CARINSTRING := TOKEN ;
RTNSTACKTOP := -1 ;
ASSERT ATPARSESTATE ;
ASSERT ¬ASRTSCANFLAG ;
ASSERT LINEPT=0 ^ LINE[0]=↑F;
ASSERT WRITEFILE(:WTHD)=LISTING OF INPUTSTRING THROUGH :RDHD ;
WHILE  ¬ RECOGNITION  DO
      ENTER TRANSNET ;
ELIHW ;
ASSERT PROGRAM(ALTSET,ASRTLOC,ASRTLOC1,ASRTS,BOUNDFUNCTION,CASELABELFRONT,CASELA
BELS,CASELABELSET,CHARLIST,DEFINEDIDENTIFIERSET,DEFINEDIDENTIFIERSETPT,DEFINEDPR
OCEDURESET,DEFINEDPROCEDURESETPT,DESCLOC,EXPLIST,INITIALPROCEDURE,PROC,PROCCALLS
,STATEMENT);
ASSERT WRITEFILE(:WTHD)=LISTING ;
ASSERT LINEPT=0 ^ LINE[0]=↑F;
EXIT ;

PROCEDURE PRINTCASELABELS;
ASSERT LINE[0]=↑F;
IF FRONTLABEL>BACKLABEL
```

```
      THEN RETURN;
FI;
INTVAL:=CASELABELS[FRONTLABEL];
ENTER INTPRINT;
ASSERT FRONTLABEL.0≤FRONTLABEL≤BACKLABEL.0=BACKLABEL;
ASSERT WRITEFILE(:WTHD) U LINE[1],...,LINE[LINEPT]=WRITEFILE(:WTHD.0) U LINE.0[1]
,...,LINE.0[LINEPT.0] U CASELABELS[FRONTLABEL.0] v,...,v CASELABELS[FRONTLABEL]
;
ASSERT LINE[0]=LINE.0[0];
WHILE FRONTLABEL<BACKLABEL  DO
      FRONTLABEL:=FRONTLABEL+1;
      CHAR:=↑ ;
      ENTER INSERTCHAR;
      CHAR:=↑v;
      ENTER INSERTCHAR;
      CHAR:=↑ ;
      ENTER INSERTCHAR;
      INTVAL:=CASELABELS[FRONTLABEL];
      ENTER INTPRINT;
ELIHW;
ASSERT WRITEFILE(:WTHD) U LINE[1],...,LINE[LINEPT]=WRITEFILE(:WTHD.0) U LINE.0[1]
,...,LINE.0[LINEPT.0] U CASELABELS[FRONTLABEL.0] v ... vCASELABELS[BACKLABEL.0];
ASSERT LINE[0]=LINE.0[0];
EXIT ;

PROCEDURE PRINTID;
ASSERT LINE[0]=↑F;
LINEFRONT:=DEFINEDIDENTIFIERSET[2*ID];
LINEBACK:=DEFINEDIDENTIFIERSET[2*ID+1];
            ENTER BUILDLINE ;
ASSERT WRITEFILE(:WTHD) U LINE[1],...,LINE[LINEPT]=WRITEFILE(:WTHD.0) U LINE.0[1]
,...,LINE.0[LINEPT.0] U DEFINED.IDENTIFIER ID.0;
ASSERT LINE[0]=↑F;
EXIT ;

PROCEDURE PRINTIDSUB;
ASSERT LINE[0]=↑F;
ENTER PRINTID;
IF ALTNUM[ID] > 0
      THEN CHAR:=↑.;
            ENTER INSERTCHAR;
            INTVAL:=ALTNUM[ID];
            ENTER INTPRINT;
FI;
ASSERT WRITEFILE(:WTHD) U LINE[1],...,LINE[LINEPT]=WRITEFILE(:WTHD.0) U LINE.0[1]
,...,LINE.0[LINEPT.0] U IDENTIFIER ID SUBSCRIPTED WITH ITS ALTERATION COUNTER;
ASSERT LINE[0]=LINE.0[0];
EXIT ;

PROCEDURE PRINTIDSUB1;
ASSERT LINE[0]=↑F;
ASSERT ALTNUM[ID]≥0;
ENTER PRINTID;
CHAR:=↑.;
ENTER INSERTCHAR;
INTVAL:=ALTNUM[ID]+1;
ENTER INTPRINT;
ASSERT WRITEFILE(:WTHD) U LINE[1],...,LINE[LINEPT]=WRITEFILE(:WTHD.0) U LINE.0[1]
,...,LINE.0[LINEPT.0] U IDENTIFIER ID SUBSCRIPTED WITH ITS ALTERATION COUNTER+1;
ASSERT LINE[0]=LINE.0[0];
EXIT ;
```

```
PROCEDURE  PRINTLINE ;
ASSERT 0≤$$≤LINEPT.0 → LINE[$$]=LINE.0[$$] ;
ASSERT LINEPT.0+1≤$$≤LINEPT → LINE[$$]=BLANK ;
WHILE LINEPT<120 DO
     LINEPT:=LINEPT+1 ;
     LINE[LINEPT]:=↑ ;
ELIHW ;
WRITE LINE ;
LINEPT:=0;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U LINE.0[0],...,LINE.0[LINEPT.0] ;
ASSERT LINEPT=0;
ASSERT LINE[0]=LINE.0[0];
EXIT ;

PROCEDURE  PRINTPATH ;
ASSERT LINEPT=0 ^ LINE[0]=↑F;
ASSERT PATH[0],...,PATH[PATHPT] IS A $PATH$;
WRITE PATHIS ;
I:=0 ;
ASSERT WRITEFILE(:WTHD) U LINE[1],...,LINE[LINEPT]=WRITEFILE(:WTHD.0) U PATHIS U
PATH[0],...,PATH[I-1];
ASSERT LINE[0]=↑F;
ASSERT 0≤I≤PATHPT+1;
ASSERT PATH  =PATH.0;
ASSERT PATHPT=PATHPT.0;
ASSERT 0≤LINEPT≤120;
WHILE I≤PATHPT DO
      CHAR:=↑  ;
      ENTER INSERTCHAR ;
      INTVAL:=PATH[I] ;
      ENTER INTPRINT ;
      I:=I+1 ;
ELIHW ;
ENTER PRINTLINE ;
ASSERT LINEPT=0 ^ LINE[0]=↑F;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U PATHIS U PATH.0[0],...,PATH.0[PATHPT
.0];
EXIT ;

PROCEDURE PRINTRDHDSUB;
ASSERT LINE[0]=↑F;
IF COLONALTNUM[3]>0
      THEN CHAR:=↑.;
            ENTER INSERTCHAR;
            INTVAL:=COLONALTNUM[3];
            ENTER INTPRINT;
FI;
ASSERT WRITEFILE(:WTHD) U LINE[1],...,LINE[LINEPT]=WRITEFILE(:WTHD.0) U LINE.0[1]
,...,LINE.0[LINEPT.0] U ALTERATION COUNTER OF :RDHD;
ASSERT LINE[0]=LINE.0[0];
EXIT ;

PROCEDURE PRINTSUBEXP;
ASSERT SUBBACK=SUBBACK.0;
ASSERT SUBFRONT.0≤SUBFRONT≤SUBBACK.0+1;
ASSERT WRITEFILE(:WTHD) U LINE[1],...,LINE[LINEPT]=WRITEFILE(:WTHD.0) U LINE.0[1]
,...,LINE.0[LINEPT.0] U SUBSCRIPTED EXPRESSION IN EXPLIST FROM SUBFRONT.0 TO SUBF
RONT-1;
ASSERT LINE[0]=↑F;
WHILE SUBFRONT≤SUBBACK  DO
      CASE EXPLIST[2*SUBFRONT]  OF
$CHARACTER.CONSTANT$  65: CHAR:=↑↑ ;
```

```
            ENTER INSERTCHAR;
            CHAR:=CHARACTER(EXPLIST[2*SUBFRONT+1]);
            ENTER INSERTCHAR;
$NUMBERS$ 66: INTVAL:=EXPLIST[2*SUBFRONT+1];
            ENTER INTPRINT;
$IDENTIFIERS 67: ID:=EXPLIST[2*SUBFRONT+1];
            ENTER PRINTID;
            IF ALTNUM[ID]>0
                THEN CHAR:=+.;
                    ENTER INSERTCHAR;
                    INTVAL:=ALTNUM[ID];
                    ENTER INTPRINT;
            FI;
$BOOLEAN$ 71: AT:=1;
            ENTER RESWORDPRINT;
$CHARACTER$ 74: AT:=3;
            ENTER RESWORDPRINT;
$FALSE$ 81: AT:=10;
            ENTER RESWORDPRINT;
$INTEGER$ 87: AT:=15;
            ENTER RESWORDPRINT;
$TRUE$ 96: AT:=24;
            ENTER RESWORDPRINT;
$SINGLE CHARACTERS$ ELSE CHAR:=CHARACTER(EXPLIST[2*SUBFRONT]);
            ENTER INSERTCHAR;
        ESAC;
        SUBFRONT:=SUBFRONT+1;
ELIHW;
ASSERT WRITEFILE(:WTHD) U LINE[1],...,LINE[LINEPT]=WRITEFILE(:WTHD.0) U LINE.0[1]
,...,LINE.0[LINEPT.0] U SUBSCRIPTED EXPRESSION IN EXPLIST FROM SUBFRONT.0 THROUGH
 SUBBACK.0;
ASSERT LINE[0]=+F;
EXIT ;

PROCEDURE PRINTWTHDSUB;
ASSERT LINE[0]=+F;
IF COLONALTNUM[4]>0
     THEN CHAR:=+.;
          ENTER INSERTCHAR;
          INTVAL:=COLONALTNUM[4];
          ENTER INTPRINT;
FI;
ASSERT WRITEFILE(:WTHD) U LINE[1],...,LINE[LINEPT]=WRITEFILE(:WTHD.0) U LINE.0[1]
,...,LINE.0[LINEPT.0] U ALTERATION COUNTER OF :WTHD;
ASSERT LINE[0]=LINE.0[0];
EXIT ;

PROCEDURE  PROCVCGEN ;
ASSERT LINEPT=0 ^ LINE[0]=+F;
CURRFRONT := 0 ;
ENTER FROMPOINTVCGEN;
CURRFRONT:=1;
ASSERT 1≤CURRFRONT≤PROC[7*CURRENTPROC+2]+1;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U VERIFICATION CONDITIONS FOR PATHS I
N PROCEDURE CURRENTPROC STARTING FROM POINTS UP TO CURRFRONT;
ASSERT LINEPT=0 ^ LINE[0]=+F;
WHILE CURRFRONT<PROC[7*CURRENTPROC+2]  DO
     IF STATEMENT[DESCLOC[CURRFRONT+BIAS]]≠84 $HALT$ ^ASRTLOC[CURRFRONT+BIAS]≠-1
          THEN ENTER FROMPOINTVCGEN;
     FI;
     CURRFRONT := CURRFRONT + 1 ;
ELIHW ;
```

```
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U VERIFICATION CONDITIONS FOR PROCEDU
RE CURRENTPROC;
ASSERT LINEPT=0 ^ LINE[0]=+F;
EXIT ;

PROCEDURE REGEN ;
ASSERT THERE IS ANOTHER $BRANCH$ FROM PATH[PATHPT];
ASSERT LINEPT=0 ^ LINE[0]=+F;
R:=PATHPT;
PATHPT:=0 ;
NSTEP:=0;
ENTER RESETALT ;
ASSERT R=PATHPT.0;
ASSERT 0≤PATHPT≤PATHPT.0;
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT);
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U VERIFICATION CONDITION TERMS FOR PA
TH[0],...,PATH[PATHPT-1];
ASSERT PATH  =PATH.0;
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
ASSERT LINEPT=0 ^ LINE[0]=+F;
WHILE PATHPT<R  DO
      ENTER GENTERM ;
      PATHPT:=PATHPT+1 ;
ELIHW ;
BRANCH[PATHPT]:=BRANCH[PATHPT]+1 ;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U VERIFICATION CONDITION TERMS FOR PA
TH.0[0],...,PATH.0[PATHPT.0-1];
ASSERT PATH  =PATH.0;
ASSERT PATHPT=PATHPT.0;
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT);
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
ASSERT LINEPT=0 ^ LINE[0]=+F;
EXIT ;

PROCEDURE  RESETALT ;
I:=0;
ASSERT 0≤I≤250;
ASSERT 0≤$$≤I-1 → ALTNUM[$$]=0;
WHILE I≤249  DO
      ALTNUM[I]:=0;
      I:=I+1;
ELIHW;
I:=1;
ASSERT 0≤$$≤249 → ALTNUM[$$]=0;
ASSERT 1≤I≤11;
ASSERT 1≤$$≤I-1 → COLONALTNUM[$$]=0;
WHILE I≤10  DO
      COLONALTNUM[I]:=0;
      I:=I+1;
ELIHW;
ASSERT 0≤$$≤249 → ALTNUM[$$]=0;
ASSERT 1≤$$≤10  → COLONALTNUM[$$]=0;
EXIT ;

PROCEDURE RESETFLAGS;
I:=0;
ASSERT 0≤I≤250;
ASSERT 0≤$$≤I-1 → ¬ALTFLAG[$$];
WHILE I≤249  DO
      ALTFLAG[I]:=FALSE;
      I:=I+1;
ELIHW;
```

```
I:=2;
ASSERT 0≤SS≤249 ⇒ ¬ALTFLAG[SS];
ASSERT 2≤I≤10;
ASSERT 2≤SS≤I-1 ⇒ ¬COLONALTFLAG[SS];
WHILE I≤9  DO
      COLONALTFLAG[I]:=FALSE;
      I:=I+1;
ELIHW;
COLONALTFLAG[ 1]:=TRUE;
COLONALTFLAG[10]:=TRUE;
ASSERT ALL ALTERATION FLAGS IN ALTFLAG AND COLONALTFLAG ARE FALSE EXCEPT COLONAL
TFLAG[1] AND COLONALTFLAG[10] WHICH ARE ALWAYS TRUE;
EXIT ;

PROCEDURE  RESTOREPARSESTATUS ;
ASSERT -1≤PARSESTACKTOP;
STATE := PARSESTATE ;
I := 0 ;
ASSERT 0≤I≤PARSESTACKTOP.0+1;
ASSERT 0≤SS≤I-1 ⇒ RTNSTACK[SS]=PARSESTACK.0[SS] ;
ASSERT STATE = PARSESTATE.0 ;
ASSERT PARSESTACKTOP=PARSESTACKTOP.0;
WHILE  I ≤ PARSESTACKTOP  DO
      RTNSTACK[I] := PARSESTACK[I] ;
      I := I + 1 ;
ELIHW ;
RTNSTACKTOP := PARSESTACKTOP ;
ASSERT STATE = PARSESTATE.0 ;
ASSERT 0≤SS≤PARSESTACKTOP.0 ⇒ RTNSTACK[SS]=PARSESTACK.0[SS] ;
ASSERT RTNSTACKTOP = PARSESTACKTOP.0 ;
EXIT ;

PROCEDURE RESWORDPRINT;
ASSERT LINE[0]=↑F;
I:=RESWORDPTS[AT];
ASSERT RESWORDPTS[AT]≤I≤RESWORDPTS[AT+1];
ASSERT WRITEFILE(:WTHD) U LINE[1]+...+LINE[LINEPT]=WRITEFILE(:WTHD.0) U LINE.0[1]
+...+LINE.0[LINEPT.0] U THE FIRST I-RESWORDPTS[AT] LETTERS OF RESERVED WORD AT;
ASSERT LINE[0]=↑F;
WHILE I<RESWORDPTS[AT+1]  DO
      CHAR:=RESERVEDWORDSET[I];
      ENTER INSERTCHAR;
      I:=I+1;
ELIHW;
ASSERT WRITEFILE(:WTHD) U LINE[1]+...+LINE[LINEPT]=WRITEFILE(:WTHD.0) U LINE.0[1]
+...+LINE.0[LINEPT.0] U RESERVED WORD AT;
ASSERT LINE[0]=↑F;
EXIT ;

PROCEDURE  SAVEPARSESTATUS ;
ASSERT -1≤RTNSTACKTOP;
PARSESTATE := STATE ;
I := 0 ;
ASSERT PARSESTATE = STATE.0 ;
ASSERT 0≤I≤RTNSTACKTOP.0+1;
ASSERT 0≤SS≤I-1 ⇒ PARSESTACK[SS]=RTNSTACK.0[SS] ;
ASSERT RTNSTACKTOP=RTNSTACKTOP.0;
WHILE  I ≤ RTNSTACKTOP  DO
      PARSESTACK[I] := RTNSTACK[I] ;
      I := I + 1 ;
ELIHW ;
PARSESTACKTOP := RTNSTACKTOP ;
```

```
ASSERT PARSESTATE = STATE.0 ;
ASSERT 0≤SS≤RTNSTACKTOP.0 ⇒ PARSESTACK[SS]=RTNSTACK.0(SS) ;
ASSERT PARSESTACKTOP = RTNSTACKTOP.0 ;
EXIT ;

PROCEDURE  SCAN ;
ASSERT ¬ RECOGNITION ;
ASSERT INPUTSTRING = CARD[0],CARD[COL],...,CARD[80],READFILE(:RDHD);
ASSERT ¬ASRTSCANFLAG ;
ASSERT LINEPT=0 ∧ LINE[0]=+F;
ASSERT WRITEFILE(:WTHD)=LISTING OF INPUTSTRING THROUGH :RDHD ;
ASSERT ATSAVEDPARSESTATE ;
STATE := SCANSTART ;
RTNSTACKTOP := -1 ;
IF CARD[0] = +T
      THEN CARINSTRING := 99 ;
      ELSE CARINSTRING := INTEGER(CARD[COL]) ;
FI ;
ASSERT ATSCANSTATE ;
ASSERT ¬ASRTSCANFLAG ;
ASSERT LINEPT=0 ∧ LINE[0]=+F;
ASSERT WRITEFILE(:WTHD)=LISTING OF INPUTSTRING THROUGH :RDHD ;
ASSERT ATSAVEDPARSESTATE ;
WHILE  ¬ RECOGNITION  DO
      ENTER TRANSNET ;
ELIHW ;
RECOGNITION := FALSE ;
ASSERT INSTRING=REMOVETOKEN(INSTRING.0);
ASSERT TOKEN=NEXTTOKEN(INSTRING.0);
ASSERT TOKENSTRING=NEXTTOKENSTRING(INSTRING.0);
ASSERT ¬ RECOGNITION ;
ASSERT LINEPT=0 ∧ LINE[0]=+F;
ASSERT WRITEFILE(:WTHD)=LISTING OF INPUTSTRING THROUGH :RDHD ;
EXIT ;

PROCEDURE SETALTIDS;
PROCNUM:=0;
ASSERT 0≤PROCNUM≤200;
ASSERT 0≤SS≤PROCNUM-1 ∧ PROCFLAG[SS] ⇒ ALTERATION FLAG IS TRUE FOR ALL IDENTIFIE
RS IN ALTERATION SET OF PROCEDURE SS;
WHILE PROCNUM≤199  DO
      IF PROCFLAG[PROCNUM]
            THEN I:=PROC[7*PROCNUM+5];
ASSERT 0≤PROCNUM≤199;
ASSERT 0≤SS≤PROCNUM-1 ∧ PROCFLAG[SS] ⇒ ALTERATION FLAG IS TRUE FOR ALL IDENTIFIE
RS IN ALTERATION SET OF PROCEDURE SS;
ASSERT PROC[7*PROCNUM+5]≤I≤PROC[7*PROCNUM+6]+1;
ASSERT ALTERATION FLAG IS TRUE FOR ALL IDENTIFIERS IN POSITIONS PROC[7*PROCNUM+5
] THROUGH I-1 OF ALTERATION SET OF PROCEDURE PROCNUM;
                  WHILE I≤PROC[7*PROCNUM+6]  DO
                        IF ALTSET[I]≥0
                              THEN ALTFLAG[ALTSET[I]]:=TRUE;
                              ELSE COLONALTFLAG[-ALTSET[I]]:=TRUE;
                        FI;
                        I:=I+1;
                  ELIHW;
      FI;
      PROCNUM:=PROCNUM+1;
ELIHW;
ASSERT PROCFLAG[SS] ⇒ ALTERATION FLAG IS TRUE FOR ALL IDENTIFIERS IN ALTERATION
SET OF PROCEDURE SS;
EXIT ;
```

```
PROCEDURE SETUP;
    $ INITIALIZES SEVERAL VARIABLES USED IN ACTIONS $
I:=0;
ASSERT 0≤I≤2000;
ASSERT 0≤$$≤I-1 ↱ ASRTLOC[$$]=-1;
WHILE I≤1999  DO
      ASRTLOC[I]:=-1;
      I:=I+1;
ELIHW;
CURRENTPROC:=-1;
STATEMENTPT:=-1;
ASRTLOCPT:=-1;
ASRTLOC1[0]:=0;
EXPLISTPT:=-1;
EXPSTRING[0]:=0;
EXPSTRING[1]:=-1;
CASELABELSETPT:=-1;
CASELABELFRONT[0]:=0;
DEFINEDCASELABELSETTOP:=-1;
ASSERT CURRENTPROC=-1;
ASSERT STATEMENTPT=-1;
ASSERT ASRTLOCPT=-1;
ASSERT ASRTLOC1[0]=0;
ASSERT 0≤$$≤1999 ↱ ASRTLOC[$$]=-1;
ASSERT EXPLISTPT=-1;
ASSERT EXPSTRING[0]=0;
ASSERT EXPSTRING[1]=-1;
ASSERT CASELABELSETPT=-1;
ASSERT CASELABELFRONT[0]=0;
ASSERT DEFINEDCASELABELSETTOP=-1;
EXIT ;

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 ;

PROCEDURE SUBSCRIPT;
ASSERT ASRTFLAG =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 ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT);
ASSERT AT IS IDENTIFIER NUMBER OR COLON IDENTIFIER NUMBER;
ASSERT CHARLIST[ASRTSCANPOINTER],...,CHARLIST[ASRTBACK] IS PORTION OF ASSERTION F
OLLOWING IDENTIFIER AT;
ASSERT LINE[0]=↑F;
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
IF AT≥0
      THEN INTVAL:=ALTNUM[AT];
      ELSE INTVAL:=COLONALTNUM[-AT];
FI;
ASSERT ASRTFLAG=ASRTFLAG.0;
ASSERT ALTNUM=ALTNUM.0;
```

```
ASSERT COLONALTNUM=COLONALTNUM.0;
ASSERT PATH=PATH.0;
ASSERT PATHPT=PATHPT.0;
ASSERT AT=AT.0;
ASSERT CHARLIST=CHARLIST.0;
ASSERT ASRTSCANPOINTER=ASRTSCANPOINTER.0;
ASSERT ASRTBACK=ASRTBACK.0;
ASSERT LINE=LINE.0;
ASSERT LINEPT=LINEPT.0;
ASSERT NSTEP=NSTEP.0;
ASSERT INTVAL= IF AT≥0 THEN ALTNUM[AT] ELSE COLONALTNUM[-AT];
ASSERT :WTHD=:WTHD.0;
IF ASRTSCANPOINTER+1≤ASRTBACK ∧ CHARLIST[ASRTSCANPOINTER]=↑. ∧
     CHARLIST[ASRTSCANPOINTER+1]=↑0
     THEN ASRTSCANPOINTER:=ASRTSCANPOINTER+2;
          IF ASRTFLAG=0 ∧ PATH[0]>0
               THEN CHAR:=↑.;
                    ENTER INSERTCHAR;
                    CHAR:=↑0;
                    ENTER INSERTCHAR;
                    RETURN;
          FI;
          IF ASRTFLAG=2
               THEN IF AT≥0
                         THEN IF ALTFLAG[AT]
                                   THEN INTVAL:=INTVAL-1;
                              FI;
                         ELSE IF COLONALTFLAG[-AT]
                                   THEN INTVAL:=INTVAL-1;
                              FI;
                    FI;
          FI;
FI;
ASSERT INTVAL=ALTERATION COUNTER FOR IDENTIFIER AT.0;
ASSERT :WTHD=:WTHD.0;
ASSERT LINE=LINE.0;
ASSERT LINEPT=LINEPT.0;
ASSERT IF ASRTSCANPOINTER.0+1≤ASRTBACK ∧ CHARLIST.0[ASRTSCANPOINTER.0]=↑. ∧ CHAR
LIST.0[ASRTSCANPOINTER.0+1]=↑0 THEN ASRTSCANPOINTER=ASRTSCANPOINTER.0+2 ELSE ASR
TSCANPOINTER=ASRTSCANPOINTER.0;
IF INTVAL>0
     THEN CHAR:=↑.;
          ENTER INSERTCHAR;
          ENTER INTPRINT;
FI;
ASSERT WRITEFILE(:WTHD) U LINE[1]+...+LINE[LINEPT]=WRITEFILE(:WTHD.0) U LINE.0[1]
+...+LINE.0[LINEPT.0] U SUBSCRIPT OF IDENTIFIER AT.0;
ASSERT IF ASRTSCANPOINTER.0+1≤ASRTBACK ∧ CHARLIST.0[ASRTSCANPOINTER.0]=↑. ∧ CHAR
LIST.0[ASRTSCANPOINTER.0+1]=↑0 THEN ASRTSCANPOINTER=ASRTSCANPOINTER.0+2 ELSE ASR
TSCANPOINTER=ASRTSCANPOINTER.0;
ASSERT LINE[0]=LINE.0[0];
EXIT ;

PROCEDURE  TESTS ;
ASSERT (¬ASRTSCANFLAG ∧ ATSCANSTATE ∧ ATSAVEDPARSESTATE) ∨ (¬ASRTSCANFLAG ∧ ATPA
RSESTATE) ∨ (ASRTSCANFLAG ∧ ATASRTSCANSTATE) ;
CASE TEST[ARC] OF
0:  ARCTEST := TRUE ;
1:  ENTER TEST1 ;
2:  ENTER TEST2 ;
3:  ENTER TEST3 ;
4:  ENTER TEST4 ;
```

```
5:  ENTER TEST5 ;
6:  ENTER TEST6 ;
7:  ENTER TEST7 ;
8:  ENTER TEST8 ;
9:  ENTER TEST9 ;
10:  ENTER TEST10 ;
11:  ENTER TEST11 ;
12:  ENTER TEST12 ;
13:  ENTER TEST13 ;
14:  ENTER TEST14 ;
15:  ENTER TEST15 ;
16:  ENTER TEST16 ;
17:  ENTER TEST17 ;
18:  ENTER TEST18 ;
19:  ENTER TEST19 ;
20:  ENTER TEST20 ;
21:  ENTER TEST21 ;
22:  ENTER TEST22 ;
23:  ENTER TEST23 ;
24:  ENTER TEST24 ;
25:  ENTER TEST25 ;
26:  ENTER TEST26 ;
27:  ENTER TEST27 ;
28:  ENTER TEST28 ;
29:  ENTER TEST29 ;
30:  ENTER TEST30 ;
31:  ENTER TEST31 ;
32:  ENTER TEST32 ;
33:  ENTER TEST33 ;
34:  ENTER TEST34 ;
35:  ENTER TEST35 ;
36:  ENTER TEST36 ;
37:  ENTER TEST37 ;
ESAC ;
ASSERT TESTFLAG IFF ARCTEST(ARC.0);
ASSERT ARC=ARC.0 ;
ASSERT STATE=STATE.0 ;
ASSERT RTNSTACK=RTNSTACK.0 ;
ASSERT RTNSTACKTOP=RTNSTACKTOP.0 ;
ASSERT S=S.0 ;
ASSERT CARINSTRING=CARINSTRING.0 ;
ASSERT :WTHD=:WTHD.0;
EXIT ;

PROCEDURE  TRANSNET ;
ASSERT [¬ASRTSCANFLAG ∧ ATSCANSTATE ∧ ATSAVEDPARSESTATE] ∨ [¬ASRTSCANFLAG ∧ ATPA
PSESTATE] ∨ [ASRTSCANFLAG ∧ ATASRTSCANSTATE] ;
ASSERT ¬ RECOGNITION ;
ASSERT ¬ASRTSCANFLAG ⊃ LINEPT=0 ∧ LINE[0]=+F;
ASSERT ¬ASRTSCANFLAG ⊃ WRITEFILE(:WTHD)=LISTING OF INPUTSTRING THROUGH :RDHD;
ENTER ALPHABETMATCH ;
IF ALPHABETMATCHFLAG
     THEN ENTER TRAVERSE ;
     ELSE ENTER NILMATCH ;
          IF NILMATCHFLAG
               THEN ENTER TRAVERSE ;
               ELSE IF ARC < FIRSTARC[STATE+1] ∧ SYMBOL[ARC] < 0
                         THEN RTNSTACKTOP := RTNSTACKTOP + 1 ;
                              RTNSTACK[RTNSTACKTOP] := ARC ;
                              STATE := -SYMBOL[ARC] ;
                         ELSE IF RECOGNITIONSTATE[ARC]
                                   THEN ENTER STACKTEST ;
```

```
                                                IF STACKTESTFLAG
                                                    THEN ARC := RTNSTACK[RTNSTACKTOP] ;
                                                         RTNSTACKTOP := RTNSTACKTOP-1 ;
                                                         ENTER TRAVERSE ;
                                                    ELSE IF PTNSTACKTOP < 0
                                                            THEN RECOGNITION := TRUE;
                                                          . ELSE WRITE NONRECOGNITION;
ASSERT INPUTSTRING NOT RECOGNIZED BY TRANSITION NETWORK;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U NONRECOGNITION MESSAGE;
                                                                   HALT ;
                                                         FI ;
                                                FI ;
                                           ELSE WRITE NONRECOGNITION ;
ASSERT INPUTSTRING NOT RECOGNIZED BY TRANSITION NETWORK;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U NONRECOGNITION MESSAGE;
                                                HALT ;
                                  FI ;
                        FI ;
              FI ;
FI ;
ASSERT ASRTSCANFLAG.0 → ATASRTSCANSTATE ;
ASSERT ¬ASRTSCANFLAG.0 ∧ ATSCANSTATE.0 → ATSCANSTATE ∧ ATSAVEDPARSESTATE ;
ASSERT ¬ASRTSCANFLAG.0 ∧ ATPARSESTATE.0 → ATPARSESTATE ;
ASSERT LINEPT=0 ∧ LINE[0]=+F;
ASSERT ¬ASRTSCANFLAG.0 → WRITEFILE(:WTHD)=LISTING OF INPUTSTRING THROUGH :RDHD;
ASSERT ¬ASRTSCANFLAG.0 → :WTHD=:WTHD.0;
EXIT ;

PROCEDURE  TRAVERSE ;
IF FLAG[ARC]
     THEN ENTER INPUT;
FI;
ENTER ACTIONS ;
STATE := NEXTSTATE[ARC] ;
EXIT ;

PROCEDURE UPDATEALTNUM;
ENTER RESETFLAGS;
ENTER LISTCALLEDPROCS;
ENTER SETALTIDS;
I:=0;
ASSERT 0≤I≤250;
ASSERT ALTERATION FLAGS IN ALTFLAG AND COLONALTFLAG ARE TRUE IFF THE CORRESPONDI
NG IDENTIFIER CAN BE ALTERED BY A CALL TO PROCEDURE PROCALLED.0;
ASSERT PROCCALLED=PROCCALLED.0;
ASSERT ALTERATION COUNTERS IN ALTNUM HAVE BEEN UPDATED FOR IDENTIFIERS 0 THROUGH
 I-1;
WHILE I≤249  DO
      IF ALTFLAG[I]
           THEN ALTNUM[I]:=ALTNUM[I]+1;
      FI;
      I:=I+1;
ELIHW;
COLONALTNUM[1]:=COLONALTNUM[1]+1;
IF COLONALTFLAG[3]
     THEN COLONALTNUM[3]:=COLONALTNUM[3]+1;
FI;
IF COLONALTFLAG[4]
     THEN COLONALTNUM[4]:=COLONALTNUM[4]+1;
FI;
COLONALTNUM[10]:=COLONALTNUM[10]+1;
ASSERT ALTERATION COUNTERS IN ALTNUM AND COLONALTNUM HAVE BEEN UPDATED FOR A CAL
```

```
L TO PROCEDURE PROCCALLED.0;

EXIT ;

PROCEDURE  UPDATEALTSET ;
ASSERT 0≤CURRENTPROC≤200;
ASSERT 0≤PROC[7*CURRENTPROC+5]≤PROC[7*CURRENTPROC+6]≤1000;
I:=PROC[7*CURRENTPROC+5];
ASSERT PROC[7*CURRENTPROC+5]≤$$≤I-1 → ALTSET.0[$$]≠AT.0;
ASSERT PROC.0[7*CURRENTPROC.0+5]≤I≤PROC.0[7*CURRENTPROC.0+6]+1;
ASSERT ALTSET=ALTSET.0;
ASSERT AT=AT.0 ;
ASSERT PROC=PROC.0 ;
ASSERT CURRENTPROC=CURRENTPROC.0 ;
WHILE I≤PROC[7*CURRENTPROC+6] DO
      IF ALTSET[I]=AT
          THEN RETURN ;
      FI ;
      I := I + 1 ;
ELIHW ;
PROC[7*CURRENTPROC+6]:=PROC[7*CURRENTPROC+6]+1;
ALTSET[PROC[7*CURRENTPROC+6]]:=AT;
ASSERT $$≠7*CURRENTPROC.0+6 → PROC[$$]=PROC.0[$$];
ASSERT THERE EXISTS A UNIQUE VALUE OF $$ SUCH THAT PROC.0[7*CURRENTPROC.0+5]≤$$≤
PROC.0[7*CURRENTPROC.0+6] ∧ ALTSET[$$]=AT.0;
EXIT ;

PROCEDURE  VCGEN ;
ASSERT LINEPT=0 ∧ LINE[0]=+F;
P:=0;
ASSERT 0≤P≤DEFINEDPROCEDURESETPT+1;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U VERIFICATION CONDITIONS FOR ALL PRO
CEDURES IN DEFINED.PROCEDURE.SET FROM 0 THROUGH P-1;
ASSERT LINEPT=0 ∧ LINE[0]=+F;
WHILE P≤DEFINEDPROCEDURESETPT  DO
     CURRENTPROC:=0;
ASSERT 0≤P≤DEFINEDPROCEDURESETPT;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U VERIFICATION CONDITIONS FOR ALL PRO
CEDURES IN DEFINED.PROCEDURE.SET FROM 0 THROUGH P-1;
ASSERT LINEPT=0 ∧ LINE[0]=+F;
     WHILE DEFINEDPROCEDURESET[P]≠PROC[7*CURRENTPROC]  DO
          CURRENTPROC:=CURRENTPROC+1;
     ELIHW;
     BIAS:=PROC[7*CURRENTPROC+1];
               ENTER PROCVCGEN ;
     P:=P+1;
ELIHW ;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U VERIFICATION CONDITIONS FOR INPUTST
RING;
EXIT ;

PROCEDURE VERIFY ;
$ TOP LEVEL OF THE VERIFICATION SYSTEM.  $
ASSERT READFILE(:RDHD)=INPUTSTRING;
ENTER SETUP;
ENTER PARSE  ;
ENTER VCGEN ;
ASSERT INPUTSTRING IS A LEGAL NUCLEUS PROGRAM;
ASSERT WRITEFILE(:WTHD)=LISTING U VERIFICATION CONDITIONS FOR INPUTSTRING;
EXIT ;

PROCEDURE WRITEASRTS;
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT);
```

```
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT NOT IN ENTER STATEMENT;
IF ASRTLOC[PATH[PATHPT]+BIAS]<0
     THEN WRITE ASRTTRUE ;
     ELSE W:=ASRTLOC1[ASRTLOC[PATH[PATHPT]+BIAS]];
          ASRTFLAG:=0;
ASSERT ASRTLOC[PATH[PATHPT]+BIAS]≥0;
ASSERT ASRTLOC1[ASRTLOC[PATH[PATHPT]+BIAS]]≤W≤ASRTLOC1[ASRTLOC[PATH[PATHPT]+BIAS
]+1];
ASSERT ASRTFLAG=0 ^ NOT IN ENTER STATEMENT;
ASSERT ASRTLOC =ASRTLOC.0;
ASSERT PATH  =PATH.0;
ASSERT PATHPT=PATHPT.0;
ASSERT ASRTLOC1=ASRTLOC1.0;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U SUBSCRIPTED ASSERTIONS FOR PATH[PAT
HPT] FROM ASRTLOC1[ASRTLOC[PATH[PATHPT]+BIAS]] THROUGH W-1;
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT);
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
          WHILE W<ASRTLOC1[ASRTLOC[PATH[PATHPT]+BIAS]+1]  DO
               ASRTFRONT:=ASRTS[2*W];
               ASRTSCANPOINTER:=ASRTFRONT ;
               ASRTBACK:=ASRTS[2*W+1];
               ENTER WRITENEXTASRT ;
               W:=W+1;
          ELIHW ;
FI ;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U SUBSCRIPTED ASSERTIONS FOR PATH.0[P
ATHPT.0];
ASSERT LINEPT=0 ^ LINE[0]=+F;
EXIT ;

PROCEDURE  WRITENEXTASRT ;
ASSERT LINE[0]=+F;
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT);
ASSERT ASRTFLAG =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≤ASRTBACK;
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
TOKEN:=0;
ASSERT ASRTFLAG =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.0;
ASSERT ASRTFRONT.0≤ASRTFRONT≤ASRTBACK+1;
ASSERT WRITEFILE(:WTHD) U LINE[1],...,LINE[LINEPT]=WRITEFILE(:WTHD.0) U LINE.0[1]
,...,LINE.0[LINEPT.0] U SUBSCRIPTED ASSERTION
            CHARLIST[ASRTFRONT.0],...,CHARLIST[ASRTFRONT-1];
ASSERT BEFORETOKEN = PREVIOUS VALUE OF TOKEN;
ASSERT LINE[0]=+F;
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT);
ASSERT PATH=PATH.0;
ASSERT PATHPT=PATHPT.0;
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
WHILE ASRTFRONT≤ASRTBACK  DO
     BEFORETOKEN:=TOKEN;
     ENTER ASRTSCAN ;
     LINEFRONT:=ASRTFRONT;
          LINEBACK:=ASRTSCANPOINTER-1 ;
          ENTER BUILDLINE ;
     IF TOKEN=67      $ID$
          THEN IF BEFORETOKEN=39      $:$
```

```
                    THEN ENTER ISCOLONID;
                         IF ¬FOUND
                              THEN FINDID1:=TOKENSTRING[0];
                                   FINDID2:=TOKENSTRING[1];
                                   ENTER FINDID;
                         FI;
                    ELSE
          FINDID1:=TOKENSTRING[0] ;
          FINDID2:=TOKENSTRING[1] ;
          ENTER FINDID ;
          FI ;
          IF FOUND
                    THEN ENTER SUBSCRIPT;
               FI;
     FI;
          ASRTFRONT:=ASRTSCANPOINTER ;
ELIHW ;
          ENTER PRINTLINE ;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U LINEL0[1],...,LINE.0[LINEPT.0] U SU
BSCRIPTED ASSERTION IN CHARLIST[ASRTFRONT.0],...,CHARLIST[ASRTBACK.0];
ASSERT LINEPT=0 ^ LINE[0]=+F;
EXIT ;

PROCEDURE WRITEREADLINE1;
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT);
ASSERT STMT=DESCLOC[PATH[PATHPT]+BIAS];
ASSERT STATEMENT[STMT]=9];
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
ASSERT ID=STATEMENT[STMT+1];
ASSERT B=BOUNDFUNCTION[ID];
LINEFRONT:=68;
LINEBACK:=78;  $:REOF(:RDHDS
ENTER BUILDFROMPRESET;
ENTER PRINTRDHDSUB;
LINEFRONT:=50;
LINEBACK:=85;
ENTER BUILDFROMPRESET;
ENTER PRINTIDSUB1;
LINEFRONT:=86;
LINEBACK:=99;
ENTER BUILDFROMPRESET;
INTVAL:=8;
ENTER INTPRINT;
LINEFRONT:=83;
LINEBACK:=85;
ENTER BUILDFROMPRESET;
ENTER PRINTIDSUB1;
LINEFRONT:=100;
LINEBACK:=103;
ENTER BUILDFROMPRESET;
ENTER PRINTIDSUB;
LINEFRONT:=104;
LINEBACK:=107;
ENTER BUILDFROMPRESET;
ENTER PRINTLINE;
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U READLINE1;
EXIT ;

PROCEDURE WRITEREADLINE2;
ASSERT LINEPT=0 ^ LINE[0]=+F;
```

```
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT);
ASSERT STMT=DESCLOC[PATH[PATHPT]+BIAS];
ASSERT STATEMENT[STMT]=91;
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
ASSERT ID=STATEMENT[STMT+1];
ASSERT B=BOUNDFUNCTION[ID];
LINEFRONT:=67;
LINEBACK:=78;
ENTER BUILDFROMPRESET;
ENTER PRINTRDHDSUB;
LINEFRONT:=80;
LINEBACK:=85;
ENTER BUILDFROMPRESET;
ENTER PRINTIDSUB1;
LINEFRONT:=108;
LINEBACK:=121;
ENTER BUILDFROMPRESET;
IF B≤80
     THEN INTVAL:=B;
     ELSE INTVAL:=80;
FI;
ENTER INTPRINT;
LINEFRONT:=82;
LINEBACK:=85;
ENTER BUILDFROMPRESET;
ENTER PRINTIDSUB1;
LINEFRONT:=122;
LINEBACK:=136;
ENTER BUILDFROMPRESET;
ENTER PRINTRDHDSUB;
LINEFRONT:=137;
LINEBACK:=142;
ENTER BUILDFROMPRESET;
IF B≥81
     THEN LINEFRONT:=143;
          LINEBACK:=151;
          ENTER BUILDFROMPRESET;
          INTVAL:=B;
          ENTER INTPRINT;
          LINEFRONT:=83;
          LINEBACK:=85;
          ENTER BUILDFROMPRESET;
          ENTER PRINTIDSUB1;
          LINEFRONT:=100;
          LINEBACK:=103;
          ENTER BUILDFROMPRESET;
          ENTER PRINTIDSUB;
          LINEFRONT:=104;
          LINEBACK:=107;
          ENTER BUILDFROMPRESET;
FI;
ENTER PRINTLINE;
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U READLINE2;
EXIT ;

PROCEDURE WRITEREADLINE3;
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT);
ASSERT STMT=DESCLOC[PATH[PATHPT]+BIAS];
ASSERT STATEMENT[STMT]=91;
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
```

```
ASSERT ID=STATEMENT[STMT+1];
ASSERT B=BOUNDFUNCTION[ID];
LINEFRONT:=75;
LINEBACK:=79;
ENTER BUILDFROMPRESET;
INTVAL:=COLONALTNUM[3]+1;
ENTER INTPRINT;
CHAR:=+=;
ENTER INSERTCHAR;
LINEFRONT:=75;
LINEBACK:=78;
ENTER BUILDFROMPRESET;
ENTER PRINTRDHDSUB;
LINEFRONT:=80;
LINEBACK:=81;
ENTER BUILDFROMPRESET;
ENTER PRINTLINE;
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U READLINE3;
EXIT ;

PROCEDURE WRITERTNPTLINE;
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT);
ASSERT LINEPT=0 ^ LINE[0]=+F;
LINEFRONT:=27;
LINEBACK:=32;  $:RTNPT$
ENTER BUILDFROMPRESET;
CHAR:=+.;
ENTER INSERTCHAR;
INTVAL:=COLONALTNUM[5]+1;
ENTER INTPRINT;
LINEFRONT:=0;
LINEBACK:=21;  $MIDDLE OF :RTNPT LINE $
ENTER BUILDFROMPRESET;
LINEFRONT:=DEFINEDIDENTIFIERSET[2*PROC[7*CURRENTPROC]];
LINEBACK :=DEFINEDIDENTIFIERSET[2*PROC[7*CURRENTPROC]+1];
ENTER BUILDLINE;
CHAR:=+:;
ENTER INSERTCHAR;
INTVAL:=PATH[PATHPT+1];
ENTER INTPRINT;
LINEFRONT:=21;
LINEBACK:=32;  $MORE OF :RTNPT LINE $
ENTER BUILDFROMPRESET;
IF COLONALTNUM[5]>0
     THEN CHAR:=+.;
           ENTER INSERTCHAR;
           INTVAL:=COLONALTNUM[5];
           ENTER INTPRINT;
FI;
LINEFRONT:=0;
LINEBACK:=2;  $END OF :RTNPT LINE $
ENTER BUILDFROMPRESET;
ENTER PRINTLINE;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U RTNPTLINE;
ASSERT LINEPT=0 ^ LINE[0]=+F;
EXIT ;

PROCEDURE WRITESTEPLINE1;
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT);
```

```
LINEFRONT:=40;
LINEBACK:=45; $:STEP$
ENTER BUILDFROMPRESET;
INTVAL:=COLONALTNUM[10]+1;
ENTER INTPRINT;
LINEFRONT:=39;
LINEBACK:=44; $=:STEP$
ENTER BUILDFROMPRESET;
IF COLONALTNUM[10]>0
      THEN CHAR:=+.;
            ENTER INSERTCHAR;
            INTVAL:=COLONALTNUM[10];
            ENTER INTPRINT;
FI;
CHAR:=++;
ENTER INSERTCHAR;
INTVAL:=NSTEP;
ENTER INTPRINT;
ENTER PRINTLINE;
NSTEP:=0;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U STEPLINE1;
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT NSTEP=0;
EXIT ;

PROCEDURE WRITESTEPLINE2;
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT);
LINEFRONT:=40;
LINEBACK:=45; $:STEP.$
ENTER BUILDFROMPRESET;
INTVAL:=COLONALTNUM[10]+1;
ENTER INTPRINT;
LINEFRONT:=39;
LINEBACK:=45; $=:STEP.$
ENTER BUILDFROMPRESET;
INTVAL:=COLONALTNUM[10];
ENTER INTPRINT;
LINEFRONT:=46;
LINEBACK:=66; $END OF :STEP LINE$
ENTER BUILDFROMPRESET;
ID:=PROC[7,PROCCALLED];
ENTER PRINTID;
ENTER PRINTLINE;
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD.0) U STEPLINE2;
EXIT ;

PROCEDURE WRITEWRITELINE1;
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT);
ASSERT STMT=DESCLOC[PATH[PATHPT]+BIAS];
ASSERT STATEMENT[STMT]=98;
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
ASSERT B=BOUNDFUNCTION[ID];
ASSERT ID=STATEMENT[STMT+1];
ENTER PRINTIDSUB;
LINEFRONT:=152;
LINEBACK:=171;
ENTER BUILDFROMPRESET;
ENTER PRINTWTHDSUB;
LINEFRONT:=80;
```

```
LINEBACK:=82;

ENTER BUILDFROMPRESET;
$$  ENTER PRINTLINE;
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD,0) U WRITELINE1;
EXIT ;

PROCEDURE WRITEWRITELINE2;
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT);
ASSERT STMT=DESCLOC(PATH[PATHPT],BIAS];
ASSERT STATEMENT[STMT]=98;.
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
ASSERT ID=STATEMENT[STMT+1];
ASSERT B=BOUNDFUNCTION(ID];
ENTER PRINTIDSUB;
LINEFRONT:=172;
LINEBACK:=192;
ENTER BUILDFROMPRESET;
ENTER PRINTWTHDSUB;
LINEFRONT:=194;
LINEBACK:=204;
ENTER BUILDFROMPRESET;
IF B≤120
      THEN INTVAL:=B;
      ELSE INTVAL:=120;
FI;
ENTER INTPRINT;
LINEFRONT:=205;
LINEBACK:=218;
ENTER BUILDFROMPRESET;
ENTER PRINTWTHDSUB;
LINEFRONT:=219;
LINEBACK:=224;
ENTER BUILDFROMPRESET;
ENTER PRINTIDSUB;
LINEFRONT:=104;
LINEBACK:=107;
ENTER BUILDFROMPRESET;
IF B≥121
      THEN LINEFRONT:=225;
            LINEBACK:=234;
            ENTER BUILDFROMPRESET;
            INTVAL:=B;
            ENTER INTPRINT;
            LINEFRONT:=205;
LINEBACK:=218;
            ENTER BUILDFROMPRESET;
            ENTER PRINTWTHDSUB;
            LINEFRONT:=219;
            LINEBACK:=224;
            ENTER BUILDFROMPRESET;
            LINEFRONT:=235;
            LINEBACK:=237;
            ENTER BUILDFROMPRESET;
FI;
$$  ENTER PRINTLINE;
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD,0) U WRITELINE2;
EXIT ;

PROCEDURE WRITEWRITELINE3;
```

```
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT ALTLIST(ALTNUM,COLONALTNUM,PATH,PATHPT);
ASSERT STMT=DESCLOC[PATH[PATHPT]+BIAS];
ASSERT STATEMENT[STMT]=98;
ASSERT NSTEP=STEPS SINCE :STEP PRINT;
ASSERT ID=STATEMENT[STMT+1];
ASSERT B=BOUNDFUNCTION[ID];
LINEFRONT:=188;
LINEBACK:=193;
ENTER BUILDFROMPRESET;
INTVAL:=COLONALTNUM[4]+1;
ENTER INTPRINT;
LINEFRONT:=238;
LINEBACK:=243;
ENTER BUILDFROMPRESET;
ENTER PRINTWTHDSUB;
LINEFRONT:=80;
LINEBACK:=81;
ENTER BUILDFROMPRESET;
SS  ENTER PRINTLINE;
ASSERT LINEPT=0 ^ LINE[0]=+F;
ASSERT WRITEFILE(:WTHD)=WRITEFILE(:WTHD,0) U WRITELINE3;
EXIT ;

START VERIFY
```

BIBLIOGRAPHY

[1] Burstall, R. M., Formal Description of Program Structure in First Order Logic, Machine Intelligence 5, Meltzer, B. and Michie, D. (Eds.) American Elsevier, 1970.

[2] Cooper, D. C., Programs for Mechanical Program Verification, Machine Intelligence 6, Meltzer, B. and Michie, D. (Eds.), American Elsevier, 1971.

[3] Elspas, B., Levitt, K. N., Waldinger, R. J., and Waksman, A., An Assessment of Techniques for Proving Program Correctness, Computing Surveys 4, 2(June, 1972).

[4] Floyd, R. W., Assigning Meanings to Programs, Proceedings of a Symposium in Applied Mathematics, Vol. 19 -- Mathematical Aspects of Computer Science, Schwartz, J. T. (Ed.), 1967.

[5] Good, D. I., Toward a Man-Machine System for Proving Program Correctness, Ph.D. Thesis, University of Wisconsin, 1970.

[6] Good, D. I., Developing Correct Software, Proceedings of the First Texas Symposium on Computer Systems, 1972.

[7] Good, D. I., and Ragland, L. C., Nucleus--A Language of Provable Programs, Program Test Methods, Hetzel, W. C. (Ed.), Prentice-Hall, 1972.

[8] Hoare, C. A. R., An Axiomatic Basis for Computer Programming, Comm. ACM, 12, 10(October, 1969).

[9] King, J. C., A Program Verifier, Ph.D. Thesis, Carnegie-Mellon University, 1969.

[10] London, R. L., The Current State of Proving Programs Correct, Proceedings of ACM Annual Conference, ACM, 1972.

[11] Naur, P., Proof of Algorithms by General Snapshots, BIT, 6 (1966).

[12] Scott, D., Outline of a Mathematical Theory of Computation, Proceedings of the Fourth Annual Princeton Conference on Information Sciences and Systems, 1970.

[13] Snowdon, R. A., PEARL: An Interactive System for the Preparation and Validation of Structured Programs, Program Test Methods, Hetzel, W. C. (Ed.) Prentice-Hall, 1972 and SIGPLAN Notices 7, 3(March, 1972).

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

[15] Woods, W. A., Transition Network Grammars for Natural Language Analysis, Comm. ACM 13, 10(October, 1970).