A NUCLEUS VERIFICATION CONDITION COMPILER

by

Yin-Yin Lee Wang

May 1973 TR-19

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

Technical Report No. 19
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

ABSTRACT

This report describes a verification condition compiler for the Nucleus Language. The first part shows how the Nucleus can be described by an SLR(1) grammar, and also shows the correspondence between Nucleus programs and reduced programs. The second part shows how the verification condition terms constructed. This compiler accepts Nucleus programs and free-form inductive assertions as input and then compiles verification conditions that are sufficient to imply the correctness of the program.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

This thesis describes the implementation of a verification condition compiler for Nucleus programs. This compiler, which is written in Snobol4 and runs on a CDC 6600, accepts Nucleus programs and free-form inductive assertions as input and then compiles verification conditions that are sufficient to imply the correctness of the program. The verification conditions must be proved manually.

Chapter II begins by giving a brief overview of the method used to state the formal definition of Nucleus. This method consists basically of defining a mapping from Nucleus programs into reduced programs, and then specifying axioms that define the executions of reduced programs. The remainder of Chapter II gives an SLR(1) grammar for Nucleus and, using this grammar, shows how Nucleus programs map into reduced programs. This mapping is a central issue because the reduced programs provide the basis for construction of the verification conditions.

Chapter III describes the actual operation of the verification condition compiler which consists of a recognizer and a verification condition generator. The SLR(1) parsing algorithm is reviewed, and the modifications of this algorithm that were used in the program are discussed. We then describe how the parser constructs an internal representation of the reduced program and also describe the program listing and verification conditions that are produced as output.

The verification condition compiler described here is a partial automation of the inductive assertion method of proving program correctness. The first system to automate this proof method was the program verifier of King [7]. This verifier automates the entire inductive assertion method except for the choice of intermediate assertions. The verifier accepts programs written in a simple, Algol-like source language that includes an ASSERT statement for associating the inductive assertions with various points in the program. The assertions are Algol boolean expressions extended to include the logical quantifiers ∀ and ∃. Given the program with its assertions, verifier then automatically reduces the program to a flow-chart like model to which the inductive assertion method is applied. Verification conditions are constructed automatically using backward substitution and algebraic simplification. The verification conditions then are subjected to an automatic theorem prover specifically designed for working with integers.

Good [6] describes another approach to automating proofs of correctness by the inductive assertion method. The major difference between this system and the one of King is that there is no automatic theorem prover. Proofs of the verification conditions are supplied manually through man-machine interaction. The system is composed of a non-interactive program analyzer and an interactive proof synthesizer. As in the system of King, the program analyzer accepts programs in an extremely simple Algol-like language and constructs a flow-chart like model of the program. This system, however, does not permit assertions to be included in the source program. Instead, they are entered later through the interactive proof synthesizer. The generation of verification conditions also is done by the synthesizer as well as maintaining a detailed record of the proof.

A number of other systems have been built since these first two. A more detailed summary of these other systems can be found in London [8]. This paper also describes the wide class of programs that have been proved.

CHAPTER II

THE NUCLEUS LANGUAGE

0. Method of Definition

The Nucleus language has a complete, formal definition of both syntax and semantics. In this section we present a brief overview of this method of definition. For a complete discussion of the method, see Good and Ragland [5].

The syntax of Nucleus is a set of rules for determining whether or not any given character string is a Nucleus program. The Nucleus syntax is defined in terms of transition networks modeled after those of Woods [10]. The language defined is the set of strings accepted by the network. This amounts to defining the syntax by defining a Nucleus recognizer in terms of a transition network.

The semantics of Nucleus define the execution of the program for any given input. The semantics are defined by the axiomatic method described by Burstall [1]. First, a transformation, called the semantic mapping, from Nucleus programs into sentences in the predicate calculus is defined. This set of sentences is called the reduced program. The same transition network that defines the Nucleus syntax also defines the semantic mapping. The second part of the definition of semantics is the specification of a set of axioms such that the execution of any Nucleus program can be deduced from its reduced program and the axioms.

Figure II.1 is a Nucleus program of two procedures and its corresponding reduced program. The numbers with parentheses such as (p) and (p.n) are not a part of the program. The numbers p serve effectively as labels, local to the procedure, for key points in the programs. The sentences in the reduced program are listed in the order in which they are defined. The points p are referred to in stating the reduced program. For example, the sentence IF(READDATA:1,A[0] = ↑T,3,4), has references to points 1,3 and 4. The meaning of this sentence, which is established by the axioms, is that point 1 in procedure READDATA has a two way branch. If the expression A[0] = ↑T is true at point 1, control goes next to point 3, else to point 4.

(p.n) ASSERT ...; is an assertion which is not executable, and hence, is not in part of the reduced program.

The reduced program is a set of predicate calculus sentences that describe the structure of a Nucleus program, that is, they say what statements and expressions the program contains and how these statements and expressions are related. Given these relations, the program execution can be deduced from the axioms. This can be put in less abstract terms by viewing the reduced program as a machine language program for a virtual machine whose interpreter is defined by the axioms. Figure II.2 shows the virtual program of the previous Nucleus program. The first column is the virtual address, and the second column is its content. The first block is the data memory and the second block is the instruction memory.

```
$  THIS PROGRAM IS DESIGNED TO SHOW THE MOST FEATURES OF NUCLEUS
   LANGUAGE  $

CHARACTER ARRAY A[80], C[10], L[10];
INTEGER LAMB, COW, I, MORECOW, MORELAMB;
PROCEDURE READDATA;
(0.1)ASSERT LAMB=X(1)+...+X(I-1);
(0.2)ASSERT COW=Y(1)+...+X(I-1);
(0.3)ASSERT IF 1≤K≤I-1, THEN¬:REOF(K);
(0)READ A;
(1)WRITE A;
(2)IF A[0] = ↑T (3)THEN (3)RETURN; (4)FI;
(4)CASE INTEGER(A[80]) OF
     4:  (5)LAMB := LAMB + 10  *  (INTEGER(A[1]) - 27)
                  + (INTEGER(A[2]) - 27) ;
         (6)2:  (7)COW := COW + 10  *  (INTEGER(A[3]) - 27)
                  + (INTEGER(A[4]) - 27);
         (8)ESAC;
(9.1)ASSERT :RDHD=:RDHD.0+1,:WTHD=:WTHD.0+1;
(9.2)ASSERT LAMB=X(1)+...+X(IF :REOF(:RDHD) THEN I-1 ELSE I);
(9.3)ASSERT COW =Y(1)+...+Y(IF :REOF(:RDHD) THEN I-1 ELSE I);
(9.4)ASSERT IF A[0]=↑T THEN I=FIRST K SUCH THAT :REOF(K);
(9.5)ASSERT IF A[0]≠↑T AND 1≤K≤I, then ¬:REOF(K);
(9)EXIT;
PROCEDURE MAIN;
(0)I:=1;
(1)COW := 0;
(2)LAMB := 0;
(3.1)ASSERT I=:RDHD=:WTHD;
(3.2)ASSERT 1≤I≤101;
(3.3)ASSERT LAMB=X(1)+...+X(I-1) WHERE X(K)=THE INTEGER IN COLUMN
            1-2 OF READ RECORD K IF COLUMN 80 HAS ↑D AND ZERO IF NOT;
(3.4)ASSERT COW=Y(1)+...+Y(I-1) WHERE Y(K)=THE INTEGER IN COLUMN
            3-4 OF READ RECORD K IF COLUMN 80 HAS ↑B AND ZERO OTHERWISE;
(3.5)ASSERT WRITE RECORDS 1,...,I-1 ARE COPIES OF READ RECORDS 1,...,I-1;
(3.6)ASSERT IF 1≤K≤I-1, THEN ¬:REOF(K);
(3)WHILE I≤100 DO
     (4)ENTER READDATA;
     (5)IF A[0]=↑I (6)THEN (6)GO TO S; (7)FI;
     (7)I := I + 1;
     (8)ELIHW;
(9.1)ASSERT I=MIN(101,FIRST K SUCH THAT :REOF(K));
(9.2)ASSERT LAMB=X(1)+...+X(I-1);
(9.3)ASSERT COW=Y(1)+...+Y(I-1);
S:  (9)IF LAMB<COW (10)THEN
        (10)MORECOW := COW - LAMB;
        (11)GO TO W;
        (12)ELSE (13)MORELAMB := LAMB - COW;
        (14)FI;
```

```
(14)L[0] := ↑F;
(15)L[1] := CHARACTER(MORELAMB / 10 + 27);
(16)MORELAMB := MORELAMB ↓ 10;
(17)L[2] := CHARACTER(MORELAMB + 27);
(18)WRITE L;
(19)GO TO E;
W:  (20)C[0] := ↑F;
(21)C[1] := CHARACTER(MORECOW / 10 + 27);
(22)MORECOW := MORECOW ↓ 10;
(23)C[2] := CHARACTER(MORECOW + 27);
(24)WRITE C;
E:  (25)NOP;
(26.1)ASSERT IF LAMB<COW THEN WRITE RECORD I+1 HAS COW-LAMB IN COLUMN 1-2;
(26.2)ASSERT IF COW<LAMB THEN WRITE RECORD I+1 HAS LAMB-COW IN COLUMN 1-2;
(26)EXIT;
START MAIN
```

FIGURE II.1a. Nucleus Program

```
ARRAY(A,80)
ARRAY(C,10)
ARRAY(L,10)

SIMPLE(LAMB)
SIMPLE(COW)
SIMPLE(I)
SIMPLE(MORECOW)
SIMPLE(MORELAMB)

READ(READDATA:0,A)
WRITE(READDATA:1,A)
IF(READDATA:1,A[0]=↑T,3,4)
JUMPTO(READDATA:3,EXITPOINT(READDATA))
CASE(READDATA:4,INTEGER(A[80]),9)
CASELABELSET(READDATA:4)={4,2}
ASSIGN(READDATA:5,LAMB,LAMB+10*(INTEGER(A[1])-27)+(INTEGER(A[2])-27))
POINTLABELLEDWITH(READDATA:4:1)=5
JUMPTO(READDATA:6,CASEJOINPOINT(READDATA:4))
POINTLABELLEDWITH(READDATA:4:2)=7
ASSIGN(READDATA:7,COW,COW+10*(INTEGER(A[3])-27)+(INTEGER(A[4])-27))
JUMPTO(READDATA:8,CASEJOINPOINT(READDATA:4))
JUMPTO(READDATA:8,9)
CASEJOINPOINT(READDATA:4)=9
EXIT(READDATA:9)
EXITPOINT(READDATA)=9
```

FIGURE II.1b. Reduced Program for Declarations and Procedure READDATA

```
ASSIGN(MAIN:0,I,0)
ASSIGN(MAIN:1,COW,0)
ASSIGN(MAIN:2,LAMB,0)
IF(MAIN:3,I ≤ 100,4,9)
ASSIGN(MAIN:4,I,I+1)
IF(MAIN:5,A[0]=↑T,6,7)
JUMPTO(MAIN:6,POINTLABELLEDWITH(MAIN,S))
ENTER(MAIN:7,READDATA)
JUMPTO(MAIN:8,3)
POINTLABELLEDWITH(MAIN:S)=9
IF(MAIN: 9,LAMB < COW,10,13)
ASSIGN(MAIN:10,MORECOW,COW-LAMB)
JUMPTO(MAIN:11,POINTLABELLEDWITH(MAIN:W))
JUMPTO(MAIN:12,15)
ASSIGN(MAIN:13,MORELAMB,LAMB-COW)
ASSIGN(MAIN:14,L[0],↑F)
ASSIGN(MAIN:15,L[1],CHARACTER(MORELAMB/10+27))
ASSIGN(MAIN:16,MORELAMB,MORELAMB↓10)
ASSIGN(MAIN;17,L[2],CHARACTER(MORELAMB+27))
WRITE(MAIN:18,L)
JUMPTO(MAIN:19,25)
POINTLABELLEDWITH(MAIN:W)=20
ASSIGN(MAIN:20,C[0],↑F)
ASSIGN(MAIN:21,C[1],CHARACTER(MORECOW/10+27))
ASSIGN(MAIN:22,MORECOW,MORECOW↓10)
ASSIGN(MAIN:23,C[2],CHARACTER(MORECOW+27))
WRITE(MAIN:24,C)
POINTLABELLEDWITH(MAIN:E)=25
JUMPTO(MAIN:25,26)
EXIT(MAIN:26)
EXITPOINT(MAIN)=26
INITIALPROCEDURE=MAIN
```

FIGURE II.1c. Reduced Program for Procedure MAIN

Data Memory

A[0]	
⋮	
A[80]	
C[0]	
⋮	
C[10]	
L[0]	
⋮	
L[80]	
COW	
I	
LAMB	
MORECOW	
MORELAMB	

Instruction Memory

READDATA:0	READ(READDATA:0,A)
READDATA:1	WRITE(READDATA:1,A)
READDATA:2	IF(READDATA:1,A[0]=↑T,3,4)
READDATA:3	JUMPTO(READDATA:3,9)
⋮	
READDATA:9	EXIT(READDATA:9)
MAIN:0	ASSIGN(MAIN:0,I,0)
⋮	
MAIN:26	EXIT(MAIN:26)

Interpreter

A X I O M S

FIGURE II.2. The Virtual Program of the Previous Nucleus Program

1. Description of Nucleus

In this section we present a description of Nucleus with particular emphasis on the semantic mapping from Nucleus programs into reduced programs. The reduced programs are extremely important because they are the base from which the verification conditions are generated by the program described in the next chapter. Although the formal definition of the Nucleus syntax is given by a transition network, the description given here is based on a context-free grammar. This is for two reasons. First, this provides a description of Nucleus by a more conventional method than a transition network; and second, the verification condition generator described in the next chapter is based on this grammar.

The semantic mapping from Nucleus into reduced programs is shown by using two functions, rdc and par, in conjunction with the productions. The function rdc(<symbol>) means the reduced program associated with <symbol>. Consider the example

```
<program> → <decseq>; <procseq>; <startpt>
rdc(<program>) = rdc(<decseq>)rdc(<procseq>)rdc(<startpt>)

<startpt> → START ID
rdc(<startpt>) = INITIALPROCEDURE = ID
```

This first production states that the reduced program of <program> consists of reduced programs of <decseq>, <procseq>, and <startpt>. The second production then specifies the reduced program of <startpt>. The function par applies to an expression and gives that expression fully parenthesized. This defines precisely the order of evaluations within the expression.

In specifying the semantic mapping, it is also necessary to specify the correspondence between points (virtual addresses) in the reduced program and lexical position in the Nucleus program. This is done by writing the points above the production at their proper positions. For example,

$$\texttt{<stmt>} \rightarrow {}^{(p)}\text{HALT}^{(p+1)}$$

This means that if p is the point corresponding to the beginning of the HALT statement, then p+1 is the point corresponding to the end.

2. Basic Elements

Nucleus programs are composed of characters from the 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 ([ ]) ↑ * / ↓ + − < ≤
≥ > = ≠ ¬ ∧ ∨ ≡ , ; : . $ #}

These characters are grouped into tokens which correspond to the terminal symbols of the grammatical description of Nucleus given in the following sections.

Each of the following single characters is a token.

([ ]) ↑ * / ↓ + − < ≤ ≥ > = ≠ − ∧ ∨ , : ;

Also certain character strings are tokens. Each of the reserved words

ARRAY, BOOLEAN, CASE, CHARACTER, DO, ELIHW, ELSE,
ENTER, ESAC, EXIT, FALSE, FI, GO, HALT, IF, INTEGER,
NOP, OF, PROCEDURE, READ, RETURN, START, THEN, TO,
TRUE, WHILE, and WRITE,

is a token. Finally, the tokens INTEGERN, ID, CH, ASSERTION and :=

are defined as follows:

INTEGERN: A non-empty sequence of decimal digits.

ID: A non-empty sequence of letters and digits. The first character must be a letter.

CH: The character ↑ followed immediately by the character c where c is any element of the basic character set.

ASSERTION: An ASSERTION token has the form

ASSERT text;

where text is any sequence of characters not containing an unquoted semicolon. A quoted semicolon is one that is immediately preceded by ↑.

:= : consists of : followed immediately by =.

Nucleus allows comments to appear between any two adjacent tokens. The form of a comment is

$ text $

where text is any string not containing a $.

3. Programs

```
<program> → <decseq>; <procseq>; <startpt>
rdc(<program>) = rdc(<decseq>)rdc(<procseq>)rdc(<startpt>)

<startpt> → START ID
rdc(<startpt>) = INITIALPROCEDURE=ID
```

A Nucleus program consists of a sequence of declarations, a sequence of procedures, and a starting point. The declarations define the global data variables of the program. Since Nucleus has no concept of a local data variable, these are the only variables that can be manipulated by the procedures in the procedure sequence. The ID following START specifies the name of the procedure where execution of the program is to begin.

4. Declarations

<decseq> → <dec>
rdc(<decseq>) = rdc(<dec>)

<decseq>$_1$ → <decseq>$_2$; <dec>
rdc(<decseq>$_1$) = rdc(<decseq>$_2$)rdc(<dec>)

<dec> → <simpledec>
rdc(<dec>) = rdc(<simpledec>)

<dec> → <arraydec>
rdc(<dec>) = rdc(<arraydec>)

<simpledec> → <type> ID
rdc(<simpledec>) = SIMPLE(ID)

<simpledec>$_1$ → <simpledec>$_2$, ID
rdc(<simpledec>$_1$) = rdc(<simpledec>$_2$) SIMPLE(ID)

<arraydec> → <type> ARRAY ID[INTEGERN]
rdc(<arraydec>) = ARRAY(ID,INTEGERN)

<arraydec>$_1$ → <arraydec>$_2$, ID[INTEGERN]
rdc(<arraydec>$_1$) = rdc(<arraydec>$_2$) ARRAY(ID,INTEGERN)

<type> → INTEGER
<type> → BOOLEAN
<type> → CHARACTER

The declaration sequence consists of simple declarations and/or array declarations. Simple declarations declare simple variables of either type INTEGER, BOOLEAN, or CHARACTER. A CHARACTER variable takes on single character values. Array declarations declare arrays of type INTEGER, BOOLEAN, or CHARACTER where the lower subscript bound is assumed to be zero and the INTEGERN between the brackets is the array upper bound.

5. Procedures

```
<procseq> → <proc>
rdc(<procseq>) = rdc(<proc>)
```

$\langle procseq\rangle_1 \rightarrow \langle procseq\rangle_2$; <proc>
rdc($\langle procseq\rangle_1$) = rdc($\langle procseq\rangle_2$)rdc(<proc>)

<proc> → PROCEDURE ID; $^{(o)}$<body>$^{(p)}$ EXIT
rdc(<proc>) = rdc(<body>) EXIT(ID:p) EXITPOINT(ID) = p

The procedure sequence consists of one or more procedures. Each procedure has a procedure name, ID, followed by a <body> and EXIT. The identifier used as procedure name must not be declared previously as a simple variable, an array, or another procedure. Procedures have no parameters, but may be called recursively.

Each procedure has associated with it a sequence {0,...,p} of local control points. Control always enters a procedure at point 0 and leaves from point p. The association of these two points with the program text are shown in the <proc> production above. The association of the intermediate points in the sequence are shown in the subsequent productions that define <body>. In order to distinguish between the local control points of different procedures, the notation ID:p is used to denote point p in procedure ID. In the subsequent definition of the reduced program corresponding to <body>, we use the notation π:p to refer to control points and π refers to the name of the procedure in which <body> appears.

6. Bodies

<body> → ASSERTION
rdc(<body>) = Φ

$\langle body\rangle_1 \rightarrow \langle body\rangle_2$ ASSERTION
rdc($\langle body\rangle_1$) = rdc($\langle body\rangle_2$)

<<body> → <labelledstmt>;
rdc(<body>) = rdc(<labelledstmt>)

$\langle body\rangle_1 \rightarrow \langle body\rangle_2$ <labelledstmt>;
rdc($\langle body\rangle_1$) = rdc($\langle body\rangle_2$) rdc(<labelledstmt>)

<labelledstmt> → $^{(q)}$ID :$^{(q)}$ <labelledstmt>
rdc(<labelledstmt>) = (POINTLABELLEDWITH(π:ID)=q)
rdc(<labelledstmt>)

A <body> consists of assertions and/or statements. Note that each statement is terminated by a semicolon. A statement can be labelled by a sequence of identifiers or may be unlabelled. Labels are local to the procedure in which they appear.

7. Assignments

<stmt> → $^{(p)}$<cellref> := <exp>$^{(p+1)}$
rdc(<stmt>) = ASSIGN(π:p,par(<cellref>),par(<exp>))

The <cellref> and <exp> must be of the same type. The function par(x) gives the fully parenthesized form of its argument x, thus specifying the order of applying operations in evaluating <cellref> and <exp>.

8. Go To

<stmt> → $^{(p)}$ GO TO ID $^{(p+1)}$
rdc(<stmt>) = JUMPTO(π:p,POINTLABELLEDWITH(π:ID))

ID is a label which must be within the procedure π.

9. Return

<stmt> → $^{(p)}$ RETURN $^{(p+1)}$
rdc(<stmt>) = JUMPTO(π:p,EXITPOINT(π))

A return statement is a jump to the exit of procedure π.

10. Null

```
<stmt> → $^{(p)}$NOP$^{(p+1)}$
rdc(<stmt>) = JUMPTO(π:p,p+1)
```

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

11. If

```
<stmt> → $^{(q)}$IF <exp> THEN$^{(q+1)}$ <body>$_1$ $^{(r)}$ELSE$^{(r+1)}$ <body>$_2$ FI$^{(s)}$
rdc(<stmt>) = IF(π:q,par(<exp>),q+1,r+1)
              rdc(<body>$_1$)
              JUMPTO(π:r,s)
              rdc(<body>$_2$)

<stmt> → $^{(q)}$IF <exp> THEN$^{(q+1)}$ <body> FI$^{(r)}$
rdc(<stmt>) = IF(π:q,par(<exp>),q+1,r)
```

The if statement has two forms, either IF-THEN or IF-THEN-ELSE. In both cases <exp> must be type boolean. The if statement is a two way branch, if the value of <exp> is true, then execution goes to the body after THEN, else to the next <body>. In an IF-THEN-ELSE control flows from the end of the <body> following THEN to the end of IF.

12. Case

```
<stmt> → $^{(p)}$CASE <exp> OF$^{(p+1)}$ <altseq>$^{(q)}$ESAC$^{(q+1)}$
rdc(<stmt>) = CASE(π:p,par(<exp>),π:q+1)
              rdc(<altseq>)
              CASEJOINPOINT(π:p) = q+1

<stmt> → $^{(p)}$CASE <exp> OF$^{(p+1)}$ <altseq> $^{(q)}$ELSE$^{(q+1)}$ <body> ESAC$^{(r)}$
rdc (<stmt>) = CASE(π:p,par(<exp>),π:q+1)
               rdc(<altseq>)
               rdc(<body>)
               CASEJOINPOINT(π:p) = r

<altseq> → <alt>
rdc(<altseq>) = rdc(<alt>)
```

$<altseq>_1 \rightarrow <altseq>_2\ <alt>$
$rdc(<altseq>_1) = rdc(<altseq>_2)\ rdc(<alt>)$

$<alt> \rightarrow {}^{(p)}INTEGERN :^{(p)} <body>\ ^{(q)}$
rdc(<alt>)= INTEGERN ε CASELABELSET(π:c)
(POINTLABELLEDWITH(π:c:INTEGERN)=p)
rdc(<body>)
JUMPTO(π:q,CASEJOINPOINT(π:c))
where c is the point at the beginning of the case statement.

$<alt>_1 \rightarrow {}^{(p)}INTEGERN :^{(p)} <alt>_2\ ^{(q)}$
$rdc(<alt>_1)$ = INTEGERN ε CASELABELSET(π:c)
POINTLABELLEDWITH(π:c:INTEGERN)=p
$rdc(<alt>_2)$
where c is the point at the beginning of the case statement.

In both forms of the case statements, the <exp> following CASE must be type integer. If the value of <exp> is k and k is in the CASELABELSET(π:c) (c is the point at the beginning of the case statement), then control goes to the alternative having k as a numeric label. When execution of an alternative is complete, control jumps to the CASEJOINPOINT(π:c) at the end of the statement. In a simple case statement if the value k of <exp> is not in CASELABELSET(π:c), control goes to CASEJOINPOINT(π:c) whereas in the CASE-ELSE form control jumps to the <body> following the ELSE.

13. While

$<stmt> \rightarrow {}^{(q)}WHILE <exp> DO^{(q+1)} <body>\ ^{(r)}ELIHW^{(r+1)}$
rdc(<stmt>) = IF(π:q,par(<exp>),q+1,r+1)
rdc(<body>)
JUMPTO(π:r,q)

Beginning at point q, if the value of <exp> is true control goes to the <body> and then jumps back to the back to point q. This statement loops continuously until the value of <exp> is false, and then control goes to point r+1.

14. Enter

$\langle stmt \rangle \rightarrow {}^{(q)}\text{ENTER ID}^{(q+1)}$
rdc(<stmt>) = ENTER(π:q,ID)

This is a possibly recursive call of the procedure name ID. Before entering the procedure, the point π:q+1 is saved on the return point stack. When a procedure exits, control flows to the point on the top of the return point stack provided the stack is not empty. If the stack is empty, execution terminates. The upper bound on this stack size is an implementation parameter, and any attempt to exceed the stack limit causes program termination.

15. Halt

$\langle stmt \rangle \rightarrow {}^{(q)}\text{HALT}^{(q+1)}$
rdc(<stmt>) = HALT(π:q)

HALT causes execution of the entire Nucleus program to terminate immediately.

16. Read

$\langle stmt \rangle \rightarrow {}^{(q)}\text{READ ID}^{(q+1)}$
rdc(<stmt>) = READ(π:q,ID)

The following discussion of read and write statements is taken from Good and Ragland [5]. ID is the name of some array of type character. The read statement accesses the standard input file. This file is structured as a sequence of records numbered 1,2,.... Each of these records either is, or is not, an end-of-file record. If a record is not an end-of-file record, it consists of a sequence of n elements of the basic character set. The record size, n, is the same for all records and is an implementation parameter.

At the beginning of program execution an input file record pointer is set to zero. The execution of a read statement then proceeds as follows:

i) The input pointer is increased by 1 to a value of, say, p.

ii) If record p is an eof record, the character T is placed in ID[0] and the rest of the elements in the array are unchanged.

iii) If record p is not an eof record, the character F is placed into ID[0]. Then character i of record p is placed into ID[i] for all i such that $1 \leq i \leq$ min(upper bound of ID, record size). The remainder of the array, if any, is left unchanged.

17. <u>Write</u>

$$\langle stmt \rangle \rightarrow {}^{(q)}\text{WRITE ID}^{(q+1)}$$
$$\text{rdc}(\langle stmt \rangle) = \text{WRITE}(\pi{:}q,\text{ID})$$

ID is the name of some array of type character. The write statement accesses a standard output file whose structure is similar to the input file, the only difference being the record size. The size of the records on the output file is also an implementation parameter and need not be the same as the record size of the input file.

i) The output pointer is increased by 1 to a value of, say, q.

ii) If ID[0] contains the character T, record q becomes an eof record.

iii) If ID[0] does not contain the character T, characters 1,...,m of record q become the characters contained in ID[1],...,ID[m] where m = min(upper bound of ID, record size). The rest of the characters in the record, if any, become blanks.

18. Expressions

<exp> → <andexp>
par(<exp>) = par(<andexp>)

<exp>$_1$ → <exp>$_2$ ∨ <andexp>
par(<exp>$_1$) = (par(<exp>$_2$)) ∨ (par(<andexp>))

<andexp> → <notexp>
par(<andexp>) = par(<notexp>)

<andexp>$_1$ → <andexp>$_2$ ∧ <notexp>
par(<andexp>$_1$) = (par(<andexp>$_2$)) ∧ (par(<notexp>))

<notexp> → <relexp>
par(<notexp>) = par(<relexp>)

<notexp> → ¬<relexp>
par(<notexp>) = ¬(par(<relexp>))

<relexp> → <binadexp>
par(<relexp>) = par(<binadexp>)

<relexp> → <binadexp>$_1$<relationop><binadexp>$_2$
par(<relexp>) = (par(<binadexp>$_1$))<relationop>(par(<binadexp>$_2$))

<binadexp> → <multexp>
par(<binadexp>) = par(<multexp>)

<binadexp>$_1$ → <binadexp>$_2$<adop><multexp>
par(<binadexp>$_1$) = (par(<binadexp>$_2$))<adop>(par(<multexp>))

<multexp> → <unadexp>
par(<multexp>) = par(<unadexp>)

<multexp>$_1$ → <multexp>$_2$<multop><unadexp>
par(<multexp>$_1$) = (par(<multexp>$_2$))<multop>(par(<unadexp>))

<unadexp> → <primary>
par(<unadexp>) = par(<primary>)

<unadexp> → <adop><primary>
par(<unadexp>) = <adop>(par(<primary>))

<relationop> → <

<relationop> → ≤

<relationop> → ≥

<relationop> → >

<relationop> → =

<relationop> → ≠

<adop> → +

<adop> → -

<multop> → *

<multop> → /

<multop> → ÷

The following discussion of expressions, primaries and the transfer functions is also taken from Good and Ragland [5]. Expressions are built from primaries in the usual way. Type integer primaries are required for required for <adop> and <multop> operands. Type boolean primaries are required for logical operands, ¬, ∧, and ∨. The relational operations may be applied to operands of any type, provided both operands are of the <u>same</u> type. If operands of type boolean or character are used, the transfer function to type integer is applied automatically.

The operators that are available are given in the table below:

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

The division operator / gives the integer part of the quotient and the modulo operator $\div$ gives the remainder, $(a\div b=a-(a/b)*b)$.

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. An expression also becomes undefined upon division (or remaindering) by zero, and array bound violation. If the value of expression is undefined, the execution terminates.

19. Primaries

```
<primary> → INTEGERN
par(<primary>) = INTEGERN

<primary> → TRUE
par(<primary>) = TRUE

<primary> → FALSE
par(<primary>) = FALSE

<primary> → CH
par(<primary>) = CH

<primary> → <cellref>
par(<primary>) = par(<cellref>)
```

```
<cellref> → ID[<exp>]
par(<cellref>) = ID[par(<exp>)]

<cellref> → ID
par(<cellref>) = ID

<primary> → (<exp>)
par(<primary>) = ( par(<exp>) )

<primary> → INTEGER (  <exp>  )
par(<primary>) = INTEGER (  par(<exp>)  )

<primary> → BOOLEAN (  <exp>  )
par(<primary>) = BOOLEAN ( par(<exp>) )

<primary> → CHARACTER (  <exp>  )
par(<primary>) = CHARACTER ( par(<exp>) )
```

A primary may be a constant token such as INTEGERN, TRUE, FALSE, or CH, may be a single variable or an array reference. In an array reference, ID[<exp>], type integer is required for the <exp>. If the value of <exp> falls outside the array bounds, the value of array reference is undefined. A primary also may be the application of a type transfer function.

20. Transfer Functions

The type transfer functions INTEGER, BOOLEAN, and CHARACTER are defined by the functions below:

```
boolofchar(x) = boolofint(intofchar(x))

boolofint(x) = false if abs(x) mod 2 = 0
             = true if abs(x) mod 2 = 1

charofbool(x) = charofint(intofbool(x))

charofint(x) = " " if abs(x) mod 64 = 0
             = "A" if abs(x) mod 64 = 1
                :
             = "#" if abs(x) mod 64 = 63
```

```
intofbool(x) = 0 if x = false
             = 1 if x = true

intofchar(x) = 0 if x = " "
             = 1 if x = "A"
                 .
                 .
                 .
             = 63 if x = "#"
             (The order in charofint and intofchar is the
              same as that shown in the basic character
              set in Section 2 of this chapter).
```

CHAPTER III

THE VERIFICATION CONDITION COMPILER

0. Introduction

This chapter describes the verification condition compiler for Nucleus that was writtin in SNOBOL4. The compiler, which is given in Appendix A, consists of two parts, a table-driven parser for an SLR(1) grammar and a verification condition generator. The parser not only checks for the syntactic legality of a Nucleus program, but also is extended to include actions that transform the Nucleus program into an internal representation of its reduced program. The verification condition generator then constructs verification conditions from the reduced program. There were two primary reasons for using a table driven parser. First, the verification condition compiler was being written at the same time that Nucleus was being defined. With the table driven method, modification of the compiler to accomodate syntactic changes in Nucleus was quite straightforward. Second, most of the development of the Nucleus definition was done in terms of its syntax being defined by an SLR(1) grammar. The decision to define the Nucleus syntax in terms of transition networks was made quite late in the development process, and at that point it was not deemed necessary to rewrite the verification condition compiler in terms of transition networks.

Since the compiler uses a table driven parser, the program input consists of two parts, (i) the parse table, followed by

(ii) the Nucleus program. A description of the Nucleus parse table is given in Appendix B. This is the table derived from the SLR(1) grammar given in Chapter II. The output of the compiler also consists of two parts. The first is a listing of the Nucleus program showing the correspondence between points in the reduced program and position in the Nucleus program. If the Nucleus program is syntactically correct, then the second part of the output is the list of verification conditions for the Nucleus program. If the program is not syntactically correct, verification conditions are not constructed, and the output is just the listing of Nucleus program with points as described above and the error messages.

1. Parsing Method

The parsing of Nucleus programs by the verification condition compiler is based on a table-driven parser for SLR(1) grammars as discussed by DeRemer [2]. The basic ideas of this approach are reviewed with the following example. Let G = $(\{\vdash, a, +, \dashv\}, \{S, E\}, S, P)$ be a context-free grammar where $\{\vdash, a, +, \dashv\}$ is the set of terminal symbols Vt, {S,E} is the set of non-terminal symbols Vn, S is the starting symbol, and P the set of productions

#1	$S \rightarrow \vdash E \dashv$
#2	$E \rightarrow a + E$
#3	$E \rightarrow a$

To show that grammar G is a SLR(1) grammar, we begin by attempting to construct a parser for G. This requires the computation of configuration sets. Each member of a configuration set is a production in P with a special marker "." in its right part. Each configuration set represents a possible "state of the parse." If the parser is in a state corresponding to a set having a marker before the symbol s, and if the next symbol to be read is an s, then the parser will read the s and enter a state corresponding to the s-successor of the original state. A special symbol "#" in the successor indicates that a reduction should be made. Figure III.1 shows the configuration sets and successor relations of the parser for grammar G.

State name	Configuration set	Successor	Next state
0	$\{S \rightarrow .\vdash E \dashv\}$	$\vdash$	1
1	$\{S \rightarrow \vdash .E \dashv$	E	2
	$E \rightarrow .a+E$	a	3
	$E \rightarrow .a\}$	a	3
2	$\{S \rightarrow \vdash E. \dashv\}$	$\dashv$	6
3	$\{E \rightarrow a.+E$	+	4
	$E \rightarrow a.\}$	#3	7
4	$\{E \rightarrow a+.E$	E	5
	$E \rightarrow .a+E$	a	3
	$E \rightarrow .a\}$	a	3
5	$\{E \rightarrow a+E.\}$	#2	7
6	$\{S \rightarrow \vdash E \dashv.\}$	#1	7
7	$\{ \}$		

FIGURE III.1. Configuration Sets and Successor Relations of the Parser for Grammar G.

From the configuration sets and their successor relations, we can abstract the essential structure and get a characteristic finite state machine (CFSM). For each configuration set there is a corresponding state in the CFSM; the empty configuration set corresponds to the final state. The transitions of the CFSM correspond to the successor relations. Figure III.2 shows the CFSM for grammar G.

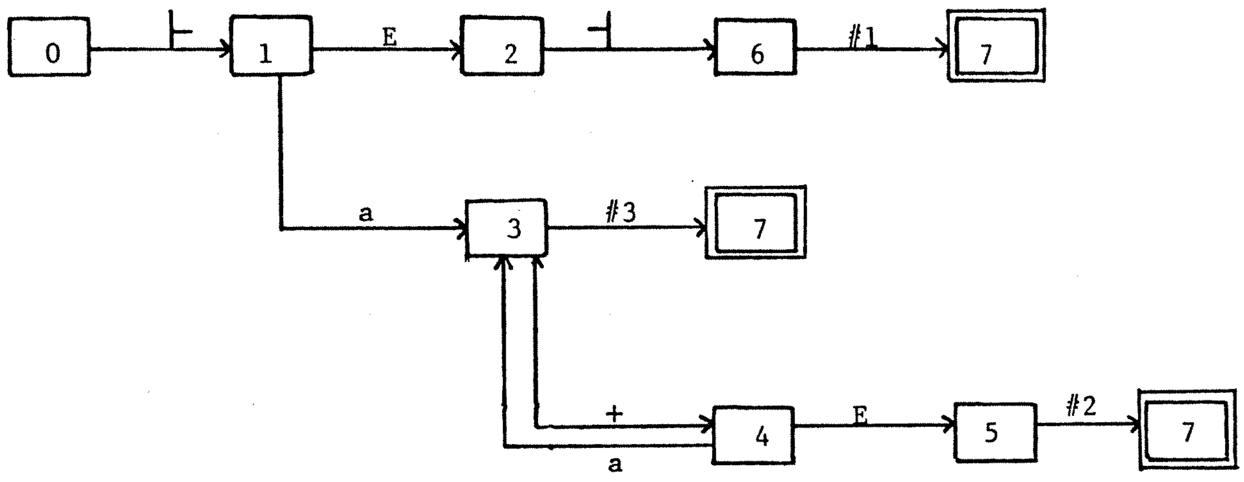

FIGURE III.2. Characteristic Finite State Machine of Grammar G

In the CFSM any state with transitions only under symbols in Vn union Vt is called a read state. Any state with one transition under one of the special # symbols and zero or one transition under a nonterminal symbol is called a reduce state. States having two or more # transitions or having one or more # transition and one or more transitions under terminal symbols are called inadequate states. In Figure III.2, states 5 and 6 are reduce states, state 3 is an inadequate state, and states 0, 1, 2, and 4 are read states. If the machine has no inadequate states, a simple algorithm can be used to parse the grammar. But if the CFSM enters an inadequate state, we do not know whether to stop and make a reduction or to allow the CFSM to continue reading. The notion of a SLR(1) grammar arises from a particularly simple solution to the indecisiveness associated with inadequate states. A context-free grammar is said to be SLR(1) if and only if each of the inadequate states of its CFSM has mutually disjoint simple 1-look-ahead sets associated with its terminal and # transitions. Grammar G is SLR(1) since the inadequate state 3 of its CFSM has the disjoint simple 1-look-ahead sets: {+} for the + transition and {$\dashv$} for the # transition. Intuitively, a 1-look-ahead set is the set of all terminal symbols that could possibly occur next.

The parsing algorithm used by the Nucleus verification condition compiler is based on the algorithm for SLR(1) grammars given by DeRemer [2]. It has been extended to use a scanner which groups the basic character string of the Nucleus program into tokens, to include error detection and recovery, and to include actions for building the reduced program. The parser starts by giving the stack the initial state of CFSM and will take Nucleus tokens as input symbols.

The algorithm:

0) If the top of stack is an inadequate state go to 2.
   If the top of stack is a reduce state go to 3.
   If the top of stack is a read state go to 1.

1) Read the next token from the input string by calling the scanner. Store on the stack the token read followed by the name of the state entered subsequently, if a transition can be made. Then do the actions associated with the transition, produce any error messages dealing with context sensitive features of the language, and return to 0. If no transition is possible, a syntactic error exists and a message is given. Then the recovery routine adjusts the stack and input string so that syntactic error detection can be

carried out for the rest of the program, and the algorithm returns to 0.

2) Call the scanner to look one token ahead. If the token is in the 1-look-ahead set of a transition under a symbol of the grammer, then go to 1. If the token is in the 1-look-ahead set of a transition under the special symbol #, go to 3. If neither, then a syntactic error exists. Perform the recovery routine and return to 0.

3) Let $A \rightarrow W$ be the production in the # transition, and let $|W|$ denote the length of W. Pop the top $2*|W|$ items off the stack. If $A = S$ (S is the starting symbol of productions) then the parse is complete so stop, otherwise return to the state whose name is on the top of the stack, and store A followed by the name of the state entered subsequently. Go to 0.

2. Reduced Program

The reduced program is represented internally by means of indirect referencing. The symbol table is stored in such a way that "ID X" has content "X" for variable X; "X BOUND" has the upper bound of array X; and "type X" has X, where type is "INTEGER", "INTEGER ARRAY", "BOOLEAN", "BOOLEAN ARRAY", "CHARACTER", or "CHARACTER ARRAY". In addition to the symbol table, an instruction table is constructed for each procedure. This table is stored in cells "_pname_ CODE _p_" and "_pname_ _p_" where pname is the procedure name and _p_ ranges over the set of virtual address for that procedure. For example, consider the instruction table shown below for procedure READDATA.

p	"READDATA CODE p"	"READDATA p"
0	READ	A
1	WRITE	A
2	IF	3,A[0] = ↑T,4
3	JMP	9
4	CASE	5,INTEGER(A[80]) = 4,7,INTEGER(A[80])=2,9
5	:=	LAMB,LAMB+10*(INTEGER(A[1])-27)+ INTEGER(A[2])-27
6	JMP	9
7	:=	COW,COW+10*(INTEGER(A[3])-27)+ INTEGER(A[4])-27
8	JMP	9
9	EXIT	9

FIGURE III.3. The Instruction Table for Procedure READDATA

One can observe that this table is quite similar to the one in Figure II.2. Most of the differences are rather minor such as the use of := rather than ASSIGN, JMP rather than JUMPTO, and a different order for the arguments in the IF sentence. A major difference is the CASE sentence. In the table above

```
4     CASE                    5,INTEGER(A[80])=4,7,INTEGER(A[80])=2,9
```

means that at point 4 if INTEGER(A[80])=4, go to point 5; if INTEGER(A[80])=2, then go to point 7; else go to 9. This records all the necessary information contained in the

```
CASE(READDATA:4,INTEGER(A[80]),9)
CASELABELSET(READDATA:4)={4,2}
POINTLABELLEDWITH(READDATA:4:1)=5
POINTLABELLEDWITH(READDATA:4:2)=7
```

of the reduced program in Figure II.1b.

3. Program Listing

The first part of the output is a listing of the Nucleus program containing numbers in parentheses that correspond to points in the reduced program. The appearance of "(q)" in the listing of procedure P means that control point P:q is associated with that position in the program. The symbols "(q.n)" preceeding an assertion mean that this is the nth assertion associated with point q in the current procedure. For example, the listing for the sample program in Appendix C is shown in Appendix D. (0),...,(9) are points corresponding to the reduced program for procedure READDATA. (0.1), (0.2), and (0.3) indicate that their succeeding assertions are associated with point 0, and similarly assertions (0.1),...,(9.5) are associated with point 9.

If any syntax errors occur in the Nucleus program, then the output will also contain error messages as shown in the example below.

```
ERR1 (0)READ <UNDEF VAR> A;
```

This means that variable A is not declared, it is an undefined variable, <UNDEF VAR>. "ERR1" means that upon completing that line, a total of one error has been detected within the program.

There are only seven error messages defined as follows.

<MTDEF VAR>	means that the next variable name is multiply defined.
<UNDEF VAR>	means that the next variable name is undefined.
<MTDEF LAB>	means that the next label name has been used previously as a label in the same procedure.
<ERR SYNTX>	means that the next token can not legally appear next.

<WRON TYPE> means that the next identifier or expression is not of required type.

<UNDEFINED LABEL NAME> means that the following label is referenced but not defined.

<UNDEFINED PROCEDURE NAME> means that the following procedure name is referenced but not defined.

The first five error messages are inserted to the Nucleus program as shown in the example above. The undefined label name is listed at the end of procedure because it is not possible to tell if a label is undefined or not until the end of the procedure is reached. For the same reason, undefined procedures are listed at the end of entire Nucleus program.

If any error occurs in the Nucleus program, then construction of the reduced program is stopped, and verification conditions are not generated. If there are no errors, verification conditions are constructed as described in the next section.

4. <u>Verification Conditions</u>

The second part of the output of the compiler is a list of verification conditions that are sufficient to imply the partial correctness of the Nucleus program. These verification conditions are sufficient to prove that each assertion included in the program is true whenever that assertion is reached during program execution, provided the initial assertion is satisfied when execution begins. Thus, if the initial assertion and all the verification conditions are satisfied, then the final assertion of the program will be satisfied if it terminates. The verification conditions are constructed for each procedure in the order in which they appear in the program. Then within each procedure one verification condition is constructed for each possible path of control between points that are tagged with assertions. In order for there to be a finite number of these paths, every possible loop must have at least one point tagged with an assertion.

The verification condition for each path is constructed to be consistent with the form described by Ragland [9]. Each verification condition has the form

```
A
..........
B
----------
C
```

which means "if A and B, then C." The A part is the set of assertions tagged to the point at the beginning of the path, the B part consists of statements that are true as a result of execution following that path, and the C part is formed from the assertions tagged to the point at the end of the path. To show that the verification condition is satisfied, it must be shown that C is provable from A and B.

The assertions are free-form and may consist of any arbitrary string of characters. These strings are interpreted as referring to program variables. A program variable is any identifier that is declared (in the declarations of the program) to be either a simple variable or an array, or any one of the special strings ":STEP", ":RDHD",

":WTHD", ":LVL", or ":RTNPT". The appearance of a program variable in an assertion is interpreted as referring to the current value of the variable. A substring of the form "variable.0" refers to the value of the variable at the time the procedure in which it appears is entered.

A verification condition is built by making a forward traversal of the path, which has a set of assertions at its beginning and another set at the end. In most cases the A part of the verification condition consists of precisely the assertions at the beginning of the path, the exception being for paths that start at the entry point of a procedure. First, "variable.0" is changed to "variable". This is because the value of the variable at the time the procedure entered is also the current value of the variable at the time the path begins. Second, if there is no assertion at the point zero, then initial assertion is assumed to be "true". Third, if the procedure happens to be the beginning of the execution of the program, then the following four statements

```
:STEP=0
:RDHD=0
:WTHD=0
:LVL=-1
```

are included. These give the initial values for each of these system variables when the program starts.

The B part of the verification condition is constructed from the program operations at the successive points along the path. For each operation, one or more terms are constructed. The key to these constructions is an alteration counter that is kept for each variable as the path is traversed. At a given point on the path the alteration counter of program variable X equals the number of times that the value of X has been altered in traversing the path up to that point. In the verification conditions, the notation X.0 refers to the value of X upon entering the procedure, just X refers to the value of X at the beginning of the path, and X.k for $k \geq 1$ refers to the value of X after it has been altered k times in traversing the path. The construction of the various terms for the B part is discussed in more detail below.

Some of the terms in the B part are labelled with "(PRV)". In proving partial correctness these terms may be used to prove the C part of the verification condition just as the unlabelled B terms are. However, if each of the labelled terms is itself proved from the lines preceeding it in the verification condition, these proofs are sufficient to imply that the program will never terminate due to an array subscript violation, divide or modulo by zero or a run time stack overflow (the stack size used is 511).

The C part of the verification condition is constructed from the assertions at the end of the path. It consists of the assertions with the alteration counter tagged to each program variable and also :RDHD, :WTHD, :LVL, :RTNPT, and :STEP. For example, if variable X is altered k times, it is changed to (X.k). If it is not altered, it is left unchanged. Similar changes are made for any other program variable except for :STEP. :STEP is changed to (:STEP+n) where n is the number of points on the path. If ".0" appears after a variable then "variable.0" is left as it is, except for the paths starting at the beginning of the procedure in which case ".0" is omitted. This is because the value at the beginning of the procedure is the same as the value at the beginning of the path.

We now explain how each of the terms in the B part of the verification condition is constructed for each of the possible elements in the reduced program. The notation a_X denotes the current value of the alteration counter of variable X, and if V is an expression, V^* is the result of substituting $X.a_X$ for every occurrence of each altered variable X in V. For example, if V is the expression (S+T)*(S+T) and S has been altered once and T is unaltered the V^* is (S.1+T)(S.1+T).

4.1. <u>ASSIGN</u>(P:q,N.V)

If N is a simple variable, then the term is

$$N.(a_A + 1) = V^*$$

and the alteration counter for N is increased by one. All other counters remain unchanged.

If N is an array reference A[E] where E is an expression, then the term is

$$A.(a_A+1)[\$] = \text{IF } \$=E^* \text{ THEN } V^* \text{ ELSE } A.a_A[\$]$$

and the alteration counter for A is increased by one. All other counters remain unchanged.

Consider, for example, path(9 13 14 15 16 17 18 26) of procedure MAIN which is shown in Appendix D. Point 16 has

ASSIGN(MAIN:16,MORELAMB,MORELAMB↓10), and the terms are

```
16(PRV)    10 ≠ 0
16         MORELAMB.2=MORELAMB.1↓10
```

The first term means that the expression on the right side of the statement has a defined value provided the divisor of the modulo operation is not zero. The second term states that the value of MORELAMB at the next point along the path is the value of the right side expression at the current point. After point 16 the alteration counter of MORELAMB equals 2 because it has been changed twice.

In the same path at point 14 has ASSIGN(MAIN:14,L[0],↑F), and its terms are

```
14(PRV)    0 ≤ 0 ≤ 10
14         L.1[$] = IF $ = 0 THEN ↑F ELSE L[$]
```

The line with "(PRV)" means that the value of expression which is the subscript of array L must be within the declared bounds of the array. The second line means that in the array only the value of element 0 is changed to ↑F while the rest of the elements in the array are unchanged.

4.2. CASE(P:q,E,f)

The term is either

E* = the element of CASELABELSET(P:q)

that is next on the current path if the next point on the path is in CASELABELSET(P:q), or

E* ≠ any of the elements of CASELABELSET(P:q)

if the next point on the path is P:f. For example, consider the case statement CASE(READDATA:4,INTEGER(A[80]),9) in procedure READDATA shown in Appendix D.

For the path(0 1 2 4 5 9), the terms are

```
4(PRV)    0 ≤ 80 ≤ 80
4         INTEGER(A[80])=(4)
```

For the path(0 1 2 4 7 9), the terms are

```
4(PRV)    0 ≤ 80 ≤ 80
4         INTEGER(A[80])=(2)
```

And for the Path(0 1 2 4 9), the terms are

```
4(PRV)    0 ≤ 80 ≤ 80
4         INTEGER(A[80]) ≠ (4 ∨ 2)
```

The value of case expression is defined to be integer number 4, 2, or any other value. The elements of CASELABELSET(READDATA:4) are 4 and 2. Hence for the first two paths, next points on the path are READDATA:5 and READDATA:7 respectively. For the third path, the value of expression is not in the CASELABELSET(READDATA:4), hence the next point on the path is READDATA:9.

4.3. IF(P:q,E,t,f)

The term is either E^* or $\neg E^*$, depending on whether the next point on the path is P:t or P:f respectively. For example, IF(MAIN:3,I ≤ 100,4,9) in path(3 4 5 7 3) has the term

```
3     I ≤ 100
```

path(3 9) has the term

```
3     ¬(I ≤ 100)
```

4.4. <u>JUMPTO</u>(P:q,r)

A JUMPTO function simply indicates which point comes next on the path and does no operation on the variables. Thus no terms are shown in the verification condition. For example, JUMPTO(MAIN:6,9) is on the path(3 4 5 9), but there is no term for it. Path(3 4 5 9) actually refers to path(3 4 5 <u>6</u> 9) with no terms shown for point 6.

4.5. <u>READ</u>(P:q,A)

The terms are

$$\begin{aligned} &:\text{REOF}(:\text{RDHD}.a_{:\text{RDHD}}+1) \rightarrow A.(a_A+1)[0]=\uparrow T \\ &\qquad \wedge[1 \leq \$ \leq \text{bound}(A) \rightarrow A.(a_A+1)[\$]=A.a_A[\$] \end{aligned}$$

$$\begin{aligned} &\neg:\text{REOF}(:\text{RDHD}.a_{\text{RDHD}}+1) \rightarrow A.(a_A+1)[0]=\uparrow F \\ &\qquad \wedge[1 \leq \$ \leq \text{MIN}(\text{readsize},\text{bound}(A) \rightarrow \\ &\qquad\qquad A.(a_A+1)[\$]=:\text{RDFL}(:\text{RDHD}.a_{:\text{RDHD}}+1,\$)] \\ &\qquad \wedge[(\text{readsize}+1) \leq \$ \leq \text{bound}(A) \rightarrow \\ &\qquad\qquad A.(a_A+1)[\$]=A.a_A[\$]] \end{aligned}$$

$$:\text{RDHD}.(a_{:\text{RDHD}}+1)=(:\text{RDHD}.a_{:\text{RDHD}})+1$$

For example, READ(READDATA:0,A) has the term

$$\begin{aligned} 0 \quad &:\text{REOF}(:\text{RDHD}+1) \rightarrow A.1[0]=\uparrow T \wedge [1 \leq \$ \leq 80 \rightarrow A.1[\$]=A[\$]] \\ &\neg:\text{REOF}(:\text{RDHD}+1) \rightarrow A.1[0]=\uparrow F \\ &\qquad \wedge[1 \leq \$ \leq \text{MIN}(80,80) \rightarrow A.1[\$]=:\text{RDFL}(:\text{FDHD}+1,\$)] \\ &\qquad \wedge[81 \leq \$ \leq 80 \rightarrow A.1[\$]=A[\$]] \\ &:\text{RDHD}.1=(:\text{RDHD})+1 \end{aligned}$$

This means that if the next read record is an end-of-file, then "T" is placed in the element zero of the read array A, and the rest of the elements in the array A are unchanged. :REOF is the function for read end-of-file, :RDHD is read head, a pointer to the next record to be read, and :RDFL is the read file itself which consists of a sequence of records. If the next read record is not an end-of-file then

"F" is placed in element zero of the array and the rest of the record is placed in the consecutive elements up to the minimum number of array bound and 80, the read record size. In this array A, if its upper bound happens to be 80 we get $81 \leq \$ \leq 80 \rightarrow A.1[\$]=A[\$]$ which is satisfied trivially. If the array upper bound is less than 80, then it means the elements between upper bound of the array and 80 are unchanged. A read statement also causes the alteration counter for the array to be increased by one as well as the counter for :RDHD.

4.6. <u>WRITE</u>(P:q,A)

The terms are

$$\begin{array}{l}
A.(a_A+1)[0]=\uparrow T \rightarrow \ :WEOF(:WTHD.a_{:WTHD}+1) \\
A.(a_A+1)[0]\neq\uparrow T \rightarrow \neg :WEOF(:WTHD.a_{:WTHD}+1) \\
\qquad\qquad \wedge[1 \leq \$ \leq \ MIN(bound(A),writesize) \rightarrow \\
\qquad\qquad\qquad :WTFL(:WTHD.a_{:WTHD}+1,\$)=A.a_A[\$]] \\
\qquad\qquad \wedge[(bound(A)+1 \leq \$ \leq writesize) \rightarrow \\
\qquad\qquad\qquad :WTFL(:WTHD.a+1)=\uparrow \ ] \\
:WTHD.(a_{:WTHD}+1)=(:WTHD.a_{:WTHD})+1
\end{array}$$

For example, WRITE(READDATA:1,A) has the term

$$\begin{array}{ll}
1 & A.1[0]=\uparrow T \rightarrow \ :WEOF(:WTHD+1) \\
 & A.1[0]\neq\uparrow T \rightarrow \neg :WEOF(:WTHD+1) \\
 & \qquad \wedge[1 \leq \$ \leq MIN(80,132) \rightarrow \ :WTFL(:WTHD+1,R)=A.1[\$]] \\
 & \qquad \wedge[81 \leq \$ \leq 132 \rightarrow \ :WTFL(:WTHD+1,\$)=\uparrow \ ] \\
 & :WTHD.1=(:WTHD)+1
\end{array}$$

For a WRITE only the alteration counter for :WTHD is increased. The above term means that if the element zero of array A is a "T", then make the current write record an end-of-file. If it is not a "T", then all elements of the current record up to the minimum of array the upper bound and the write record size, 132, are made equal to the elements of the array. The elements beyond the bound become blanks in the write file, :WTFL.

4.7. ENTER(P:q,H)

The terms are

$:LVL.(a_{:LVL}+1)=(:LVL.a_{:LVL})+1$

(PRV) $0 \leq :LVL.(a_{:LVL}+1) \leq$ maximum return point stack size

$:RTNPT.(a_{:RTNPT}+1)[\$] =$ IF $\$=:LVL.(a_{:LVL}+1)$
THEN P:(q+1) ELSE :RTNPT[$]

(PRV)I^*

0^{*+1}

$:LVL.(a_{:LVL}+2)=(:LVL.(a_{:LVL}+1))-1$

where I is the initial assertion of the called procedure and 0 is the final assertion of it. I^* is the I with its variables, and :RDHD, :WTHD, :LVL, :RTNPT, and :STEP tagged with current alteration counters, and 0^{*+1} is 0^* with the alterable variables of procedure H having their alteration counters increased by one. For example,

ENTER(MAIN:4,READDATA) has term

```
4          :LVL.1=(:LVL)+1
4(PRV)     0 ≤ :LVL.1 ≤ 511
4          :RTNPT.1[$]= IF $=:LVL.1 THEN MAIN:5 ELSE :RTNPT[$]
4(PRV)     LAMB=X(1)+...+X(I-1)
4(PRV)     COW=Y(1)+...+Y(I-1)
4(PRV)     IF 1 ≤ k ≤ I-1, THEN :REOF(K)
4          (:RDHD.1)=(:RDHD)+1,(:WTHD.1)=(:WTHD)+1
4          (LAMB.1)=X(1)+...+X(IF :REOF((:RDHD.1)) THEN I-1 ELSE I)
4          (COW.1)=Y(1)+...+X(IF :REOF((:RDHD.1)) THEN I-1 ELSE I)
4          IF (A.1)[0]= T THEN I=FIRST k SUCH THAT :REOF(K)
4          IF (A.1)[0]≠ T AND 1 ≤ K ≤ I, THEN ¬:REOF(K)
4          :LVL.2=(:LVL.1)-1
```

The first three lines mean that the new return point stack level :LVL.1 is within the bound of the :RTNPT array, which is 511. If the element of :RTNPT is :LVL then it changes to the value of the next point of the path which is MAIN:5, the rest of element in :RTNPT is unchanged.

Line 4-6 require a proof that the initial assumption of procedure READDATA is satisfied on the current values of the program variables. The alteration counter for all the alterable variables that can be altered by procedure READDATA are all increased by one at this time. These are variables which are either the left side of assignment, the array name of a read statement and :RDHD, :WTHD for write statements, or :LVL and :RTNPT for enter statements. The alterable variables for procedure READDATA are LAMB, COW, A, :RDHD, and :WTHD. Line 7-11 are the final assertion of READDATA with "X.0" changed to "$X.a_X$" for program variables X. For program variables not followed by ".0", X is changed to $X.a_X+1$ if X is one of the alterable variables of the procedure, and is unchanged otherwise. Line 12 means that after the enter, the next level of return point is the current level minus one.

CHAPTER IV

CONCLUSION

This report describes a verification condition compiler for the Nucleus language. We have shown how Nucleus can be described by an SLR(1) grammar, and also shown the correspondence between Nucleus programs and reduced programs.

The verification condition compiler itself consists of a table-driven SLR(1) parser that recognizes the Nucleus program and builds an internal representation of the corresponding reduced program. Path forward verification conditions are then constructed from the reduced program. These are simply printed as part of the compiler output and must be proved manually.

This verification condition compiler makes it possible to prove the correctness of programs of moderate size. For example, this compiler was used to help prove the correctness of another verification condition compiler written by Ragland [9]. The Ragland compiler consists of about 200 Nucleus procedures each approximately one page in size. A proof of a program of this size would not have been possible without the kind of automatic help provided by the compiler described here.

APPENDIX A

THE VERIFICATION CONDITION COMPILER PROGRAM

```
             OUTPUT(↑EJECT↑,↑OUTPUT↑,↑↑)
             EJECT = 1
             X = ↑NOPRINT↑
             EROR = 0
             II = 0
             IFLVR = ↑IFLVR↑
             ELSELEVER = ↑ELSELEVER↑
             IFLEVER = ↑IFLEVER↑
             STACK = ↑000↑
             PARSET = ARRAY(↑0:143↑)
PR00         P = TRIM(INPUT)
             P LEN(3) . L =                                  :S(ENDP)
             L ↑199↑
             PARSET[L] = PARSET[L] P
PR01         PARSET[L] LEN(1) . W                            :S(PR03)
             W ↑ ↑
PR02         PARSET[L] BREAK(↑01↑) . V LEN(3) . SST =        :F(PR00)
             $(V L) =  SST
             PARSET[L] LEN(1) . W                            :F(PR02)
             W ↑ ↑
             $(↑INAD↑ L) = L
PR03          C = 0
             PARSET[L] ↑ ↑ =
             PARSET[L]    BREAK(↑#↑) . LT ↑#↑ =
             $(↑REDLEFT↑ L) =  LT                            :F(PR05)
PR04         PARSET[L]   ↑ ↑ =                               :(PR04)
             C = C + 1                                       :(PR00)
PR05         $(↑REDUCE↑ L) =  C
ENDP         DIGIT = ANY(↑0123456789↑)
             LETTER = ANY(↑ASDFGHJKLQWERTYUIOPZXCVBNM↑)
             DEL = ↑: ;,.+-*/↓()[]≤≥<>=≠∧∨¬#$↑  ≠↑≠
             DELIMITER = ANY(↑: +-*/↓()[]∨∧¬≥≤><=≠;,#.≡$↑) ∨ ≠↑≠
             OUTPUT = ↑ NUCLEUS VERIFICATION CONDITION GENERATOR     ↑
                  ↑        VERSION  I   ↑ DATE
             RESINTEGER =  ↑INTEGER↑
             RESPOOLEAN = ↑BOOLEAN↑
             RESCHARACTER = ↑CHARACTER↑
             RESARRAY = ↑ARRAY↑
             RESPROCEDURE = ↑PROCEDURE↑
             RESEXIT = ↑EXIT↑
             RESGO = ↑GO↑
             RESTO = ↑TO↑
             RESIF = ↑IF↑
             RESTHEN = ↑THEN↑
             RESELSE = ↑ELSE↑
             RESWHILE = ↑WHILE↑
             RESDO = ↑DO↑
             RESENTER = ↑ENTER↑
             RESWRITE = ↑WRITE↑
             RESREAD = ↑READ↑
             RESRETURN = ↑RETURN↑
             RESNOP = ↑NOP↑
             RESELIHW = ↑ELIHW↑
             RESTRUE = ↑TRUE↑
```

```
                RESFALSE = ↑FALSE↑
                RESSTART = ↑START↑
                RESFI = ↑FI↑
                RESESAC = ↑ESAC↑
                RESHALT = ↑HALT↑
                RESCH = ↑CH↑
                RESCASE = ↑CASE↑
                RESOF = ↑OF↑
                PTNGO = ↑GO↑
                PTNRETURN = ↑RETURN↑
                PTNWHILE = ↑WHILE↑
                PTNIF = ↑IF↑
                PTNCASE = ↑CASE↑
                PTNENTER = ↑ENTER↑
                PTNREAD = ↑READ↑
                PTNWRITE = ↑WRITE↑
                PTNNOP = ↑NOP↑
                PTNELSE = ↑ELSE↑
                PTNEXIT = ↑EXIT↑
                PTNHALT = ↑HALT↑
*****          INSERT ABSOLUTE OVERLAY GENERATION HERE
*              SCANNER
DEFTK           DEFINE(↑TOKENS(X)↑,↑TOK↑)                         :(DEFSYM)
TOK             IDENT(CARD)                                       :F(TOK4)
                IDENT(I,↑EOR↑)                                    :S(RETURN)
TOK0            SCARD LEN(133)                                    :S(TOK04)
TOK01           OUTPUT = SCARD
                SCARD = INPUT                                     :S(TOK1)
                I = ↑EOR↑                                         :(TOK4)
TOK04           SCARD LEN(90) . W =
                OUTPUT = W
TOK05           SCARD LEN(90) . W =                               :F(TOK06)
                OUTPUT = ↑      ↑    W                            :(TOK05)
TOK06           SCARD = ↑       ↑ SCARD                           :(TOK01)
TOK1            CARD = CARD SCARD
                SCARD = ↑       ↑
                IDENT(I,↑EOR↑)                                    :S(TOK4)
                CARD RTAB(1) LEN(1) . B
                B ↑ ↑                                             :F(TOK2)
                CARD = TRIM(CARD)    ↑ ↑
TOK2            X ↑NOPRINT↑                                       :S(TOK4)
                OUTPUT = CARD
TOK4            KEEPBLANK =
TOK44           CARD LEN(1) . B                                   :F(TOK4A)
                B ↑ ↑                                             :F(TOK4A)
                KEEPBLANK = KEEPBLANK   ↑ ↑
                CARD ↑ ↑ =                                        :(TOK44)
TOK3            CARD ↑$↑ =
                SCARD = SCARD KEEPBLANK ↑$↑
TOK31           CARD BREAK(↑$↑) . V ↑$↑ =                         :F(TOK33)
                SCARD = SCARD V ↑$↑                               :(TOK)
TOK33           OUTPUT = SCARD  CARD
                SCARD =
                CARD =  INPUT                                     :(TOK31)
TOK4A           IDENT(CARD)                                       :F(TOK4B)
                IDENT(I,↑EOR↑)                                    :F(TOK0)
TOK4B           CARD DELIMITER                                    :S(TOK5)
                WORD = CARD                                       :(TOK7M)
TOK5            CARD LEN(1) . W
                W DELIMITER                                       :F(TOK7)
                W ↑$↑                                             :S(TOK3)
                W ≠↑≠                                             :F(TOK8)
                CARD LEN(2) . WORD                                :S(TOK6)
                IDENT(I,↑EOR↑)                                    :S(TOK8C)F(TOK0)
TOK6            TOKEN = ↑CH↑                                      :(RETURN)
TOK7            CARD BREAK(DEL) . WORD
TOK7M           IDENT(WORD,↑ASSERT↑)                              :S(TOK7A)
                IDENT($(↑RES↑ WORD))                              :F(TOK7R)
TOK7H           WORD LETTER                                       :F(TOK7D)
                WORD LEN(1) . W
                W LETTER                                          :F(TOK7H)
                TOKEN = ↑ID↑                                      :(RETURN)
```

```
TOK7B    WORD BREAK(+ASDFGHJKLZXCVBNMQWERTYUIOP+) . W
         WORD = W                                      :(TOK7D)
TOK7R    TOKEN = WORD                                  :(RETURN)
TOK7D    TOKEN = +INTEGERN+                            :(RETURN)
TOK7A    CARD BREAK(+;+) . WORD                        :S(TOK7AA)
         IDENT(I,+EOR+)                                :F(TOK7Z)
         WORD = CARD
         TOKEN = +ASSERTION+                           :(RETURN)
TOK7Z    W = INPUT                                     :S(TOK7ZZ)
         I = +EOR+
TOK7ZZ   CARD = CARD W                                 :(TOK7A)
TOK7AA   WORD = WORD +:+
         WORD ≠+;≠                                     :S(TOK7AG)
TOK7AB   TOKEN = +ASSERTION+                           :(RETURN)
TOK7AC   IDENT(I,+EOR+)                                :F(TOK0)S(TOK7AB)
TOK7AG   CARD WORD LEN(1)                              :F(TOK7AC)
         TCARD = CARD
         TCARD WORD =
         TCARD BREAK(+;+) . W2                         :F(TOKA2)
TOKA3    WORD = WORD W2 +;+                            :(TOK7AB)
TOKA2    IDENT(I,+EOR+)                                :F(TOK0)S(TOKA3)
TOK8     W +:+                                         :F(TOK8A)
         CARD LEN(2) . W                               :F(TOK0)
         W +:=+                                        :S(TOK8A)
         W = +:+
TOK8A    WORD = W
         TOKEN = W                                     :(RETURN)
TOK8C    WORD = ≠+≠
         TOKEN = WORD                                  :(RETURN)
*        BUILD THE SYMBOL TABLE FOR SEMENTIC ROUTINE
DEFSYM   DEFINE(+SMBTABLE(X)+,+SYM+)                   :(DEFCONT)
SYM      IDENT(PROHP)                                  :F(RETURN)
         BTOKEN +[+                                    :S(SYMBND)
         TOKEN +ID+                                    :S(SYMTB)
         TOKEN  +ARRAY+                                :S(SYMTYA)
         IDENT($(+RES+ TOKEN))                         :F(SYMTYS)S(RETURN)
SYMTYA   TYPE = TYPE + ARRAY+                          :(RETURN)
SYMTYS   TYPE = TOKEN                                  :(RETURN)
SYMBND   $(ARRAYNAME + BOUND+) = WORD                  :(RETURN)
SYMTB    IDENT($(+ID + WORD))                          :F(ERDEC)
         $(+ID + WORD) = WORD
         ARRAYNAME = WORD
         $(TYPE WORD) = WORD
         TYPELIST = TYPELIST WORD + +                  :(RETURN)
ERDEC    EROR = EROR + 1
         SCARD LEN(4) =
         SCARD = +ERR+  EROR SCARD  + <MTDEF VAR> +    :(RETURN)
*        DEFINE CONTROL POINTS
DEFCONT  DEFINE(+CONTRLAS(X)+,+CONAS+)                 :(DEFCON1)
CONAS    KEYASRT PROHP                                 :S(CONAS1)
         ASP = 1
CONAS1   KEEP = KEEPBLANK +(+ PROHP +.+ ASP +)+ WORD
         KEYASRT = PROHP
         W = WORD
         W +ASSERT+ = + +
CONAS2   W +  + = + +                                  :S(CONAS2)
         P = PROHP +.+ ASP +          +
         P LEN(10) . WW
         P = WW W
         $(PNAME +AS+ PROHP) = $(PNAME +AS+ PROHP)  P
         ASP = ASP + 1                                 :(RETURN)
DEFCON1  DEFINE(+CONTRL(X)+,+CON1+)                    :(DEFCHK)
CON1     KEEP = KEEPBLANK +(+ PROHP +)+ WORD
         KEYASRT =
         +FI THEN+ TOKEN                               :S(RETURN)
CONED    PROHP = PROHP + 1                             :(RETURN)
*        DEFINE CHECHING IDENTIFIER DEFINED OR NOT
DEFCHK   DEFINE(+CHECKID(X)+,+CHK+)                    :(DEFEXP)
CHK      IDENT(PROHP)                                  :S(RETURN)
         +:+  TOKEN                                    :S(CHK3)
         +ID+  TOKEN                                   :F(RETURN)
         +TO+ BTOKEN                                   :S(CHK4)
         BTOKEN +PROCEDURE+                            :S(CHK1)
```

```
EXP9C     $(↑CHARACTER↑   WORD) WORD                 :S(RETURN)
EXPCC     $(↑CHARACTER ARRAY↑   WORD) WORD           :S(RETURN)F(ERFXP)
EXP9      EXPSNT =
          EXPSNB =
          EXPSNC =                                   :(RETURN)
EXP10     $(PNAME ↑ENTER↑)  WORD ↑ ↑                 :S(EXPCC)
          $(PNAME ↑ENTER↑) = $(PNAME ↑ENTER↑)  WORD ↑ ↑   :(EXPCC)
EXP11     $(PNAME ↑ENTER↑) ↑:WTHD ↑                  :S(EXPCC)
          $(PNAME ↑ENTER↑) = $(PNAME ↑ENTER↑) ↑:WTHD ↑    :(EXPCC)
EXP12     $(PNAME ↑ENTER↑) ↑:RDHD ↑                  :S(EXP10)
          $(PNAME ↑ENTER↑) = $(PNAME ↑ENTER↑) ↑:RDHD ↑    :(EXP10)
EREXP     EROR = EROR + 1
          SCARD LEN(4) =
          SCARD = ↑ERR↑  EROR SCARD  ↑ <WRON TYPE> ↑      :(RETURN)
DEFINT    DEFINE(↑INTERNAL(X)↑,↑INT0↑)               :(DEFPT)
INT0      W = 0
          X ↑NOPRINT↑                                :S(RETURN)
INT1      OUTPUT =  $(PNAME ↑AS↑ W)
          OUTPUT = $(PNAME ↑CASE↑ W)
          OUTPUT = ↑ ↑ W  ↑     ↑ $(PNAME ↑CODE↑ W) ↑   ↑ $(PNAME W)
INT9      W = LT(W,$(PNAME ↑LASTP↑))  W + 1          :S(INT1)F(RETURN)
*        DECFINE    ARRAY FOR THE HEAD POINS
DEFPT     DEFINE(↑POINTS(X)↑,↑PT↑)                   :(DEFCTR)
PT        TOKEN ↑PROCEDURE↑                          :S(CONPR)
          IDENT(PROHP)                               :S(RETURN)
          TOKEN ↑ESAC↑                               :S(PT11)
          KSUCER ↑ALT↑                               :S(PT11)
          ↑TO↑ TOKEN                                 :S(RETURN)
          TOKEN ↑START↑                              :S(RETURN)
          TOKEN ↑ASSERTION↑                          :S(CONASP)
          ↑GO RETURN↑ TOKEN                          :S(PT15)
          IDENT($(↑PTN↑ TOKEN))                      :F(PT1)
          ↑START↑  BTOKEN                            :S(PT14)
          ↑:↑ TOKEN                                  :S(PT5)
          TOKEN ↑ELIHW↑                              :S(PT7)
          ↑OF↑ TOKEN                                 :S(PT91)
          ↑THEN  DO FI : ↑ TOKEN                     :S(PT3)
          ↑:↑  BTOKEN                                :S(PT95)
          ↑OF↑ BTOKEN                                :S(PT90)
          ↑: THEN ELSE DO ASSERTION ↑ BTOKEN         :S(PT8)
          TOKEN ↑:=↑                                 :S(PT61)
          IDENT(CODE)                                :S(RETURN)F(PT2)
CONPR     PROHP = 0                                  :(RETURN)
          BHP = ↑HHP↑                                :(RETURN)
CONASP    CALL = CONTRLAS(X)
PT7       CODE =
          CALL = CONTRL(X)
          BHP = PROHP - 1
          $(PNAME ↑CODE↑ BHP) = ↑JMP↑
          $(PNAME BHP) = $(IFLEVER II)               :(PT45)
PT1       CODE = TOKEN
          CALL = CONTRL(X)
           BHP = PROHP - 1
          $(PNAME ↑CODE↑ BHP) = TOKEN
          ↑CASE↑ TOKEN                               :S(PT10)
          ↑IF WHILE ↑ TOKEN                          :S(PT4)
          TOKEN ↑ELSE↑                               :S(PT44)
          TOKEN ↑EXIT↑                               :F(RETURN)
          $(PNAME ↑LASTP↑) = BHP
PT16      KEEPGORTN BREAK(↑ ↑) . W ↑ ↑ =             :F(PT18)
          $(PNAME ↑CODE↑ W) ↑GO↑ = ↑JMP↑             :F(PT17)
          $(PNAME W) BREAK(↑,↑) . V =
          $(PNAME W) = $(↑PTLB↑ V) ↑,↑               :(PT16)
PT17      $(PNAME ↑CODE↑ W)               = ↑JMP↑
          $(PNAME W) = $(PNAME ↑LASTP↑) ↑,↑          :(PT16)
PT18      IDENT(FORWORDLB)                           :S(CHK6)
          EROR = EROR + 1
          OUTPUT = ↑ERR  <UNDEFINED LABEL ↑ FORWORDLB ↑> ↑
CHK6      MULABL =
          LABLEV =
          FORWORDLB =
*           DEFINE VARIABLE COUNTER FOR ASSERTIION
DEFCTR       DEFINE(↑COUNTR(X)↑,↑CTR↑)                      :(DEFNRS)
CTR          WT = TYPELIST
CTR1         WT BREAK(↑ ↑) . WORD ↑ ↑ =                     :F(RETURN)
             $(↑ID ↑ WORD ↑CTR↑) = 0                        :(CTR1)
*           DEFINE VARIFICATION CONDITION ON RIGHT HAND SIDE OF :=
DEFNRS       DEFINE(↑NEWRSIDE(X)↑,↑NR↑)                     :(DEFBRE)  .
NR           NRW =
NR1          IDENT(RW,↑,↑)                                  :S(NR100)
             RW LEN(1) . IW
             IW DELIMITER                                   :S(NR2)
             RW BREAK(↑.,+-*/+[]()<≤>≥=≠∧∨¬→:;$ ≡↑)  . IW =
NR12         IW LETTER                                      :F(NR34)
             X ↑ASSERT↑                                     :F(NR13)
             ↑:RDHD ↑    BIW IW ↑ ↑                         :S(NR101)
             ↑:WTHD ↑    BIW IW ↑ ↑                         :S(NR102)
             ↑:LVL ↑     BIW IW ↑ ↑                         :S(NR103)
             ↑:STEP ↑    BIW IW ↑ ↑                         :S(NR104)
             ↑:RTNPT ↑   BIW IW ↑ ↑                         :S(NR105)
             ↑:RDFL :WTFL :REOF :WEOF :LOC ↑  BIW IW ↑ ↑    :S(NR34)
NR13         NEXTW =
             RW LEN(2) . NEXTW
             NEXTW  ↑.0↑                                    :S(NR40)
             IDENT($(↑ID ↑ IW ↑CTR↑))                       :S(NR34)
             EQ($(↑ID ↑ IW ↑CTR↑),0)                        :S(NR34)
             X ↑ASSERT↑                                     :S(NR15)
             NRW = NRW IW ↑.↑ $(↑ID ↑ IW ↑CTR↑)
NR14         BIW = IW
             IDENT(UPAM)                                    :F(NR7)S(NR1)
NR15         NRW = NRW ↑(↑ IW ↑.↑ $(↑ID ↑ IW ↑CTR↑) ↑)↑     :(NR14)
NR2          ↑/+↑  BIW                                      :S(NR4)
             ↑[↑  IW                                        :S(NR5)
             ↑]↑  IW                                        :S(NR80)
             BIW ≠↑≠                                        :S(NR3)
             X ↑ASSERT↑                                     :F(NR3)
             ↑$↑ IW                                         :S(NR45)
NR3          RW IW =
NR34         BIW = IW
             NRW = NRW IW
NR39         IDENT(UPAM)                                    :F(NR6)S(NR1)
NR40         RW NEXTW =
             Y ↑Y↑                                          :S(NR42)
             NRW = NRW IW ↑.0↑
NR41         BIW = IW                                       :(NR39)
NR42         TW = $(↑ID ↑ IW ↑CTR↑) - 1
             NRW = NRW IW ↑.↑ TW
             NRW ↑.0↑ =                                     :(NR41)
NR4          IDENT(IW,↑0↑)                                  :S(NRE)F(NR3)
NR45         RW ↑$↑ =
             RW BREAK(↑$↑) . IW ↑$↑ =
             NRW = NRW ↑$↑ IW ↑$↑                           :(NR1)
NR5          UPAM = $(BIW ↑ BOUND↑)                         :(NR3)
NR6          MIDAM = MIDAM IW
             MIDAM ↑[↑ =                                    :(NR1)
NR7          MIDAM = MIDAM IW ↑.↑ $(↑ID ↑ IW ↑CTR↑)         :(NR1)
NR80         IDENT(UPAM)                                    :S(NR3)
NR8          MIDAM LETTER                                   :S(NR9)
             MIDAM  DELIMITER                               :S(NR9)
             GT(MIDAM,UPAM)                                 :S(NR10)
NR9          X ↑ASSERT↑                                     :S(NR91)
             OUTPUT = B ↑0≤↑ MIDAM ↑≤↑ UPAM
NR91         UPAM =
             MIDAM =                                        :(NR3)
NR10         OUTPUT = BP ↑    ** ARRAY OVERFLOW↑            :(NR3)
NRE          OUTPUT = ↑ZERO DEVISOR↑                        :(NR3)
NR100        X ↑ASSERT↑                                     :S(RETURN)
NRMD         MOD = NRW
NR50         MOD BREAK(↑/+↑) =                              :F(RETURN)
             MOD LEN(1) LEN(1) . Q = Q
             SAVEMOD = MOD
             Q ↑(↑                                          :S(NR530)
             MOD BREAK(↑)+-$/+↑) . SAVEMOD =
             OUTPUT = B  SAVEMOD ↑≠0↑                       :(NR50)
```

```
        NRTN = NRTN + INCLVL                                  :(RETURN)
ENT118  WW = $(P ↑CALLENTER↑)
ENT108  WW BREAK(↑ ↑) . V ↑ ↑ =                               :F(ENT102)
        WW V                                                  :S(ENT108)
        P V                                                   :S(ENT108)
        INCLVL = 1
        WW = $(V ↑CALLENTER↑) WW
        V = $(V ↑ENTER↑)
ENT109  V BREAK(↑ ↑) . AW ↑ ↑ =                               :F(ENT108)
        LW AW                                                 :S(ENT109)
        LW = LW AW ↑ ↑                                        :(ENT109)
DEFGO   DEFINE(↑PATHNASSERT(X)↑,↑GOCALL↑)                     :(DEFASN)
GOCALL  III = 0
        NP = 0
PTH0    PATHBGN = NP
        ZZZ = 0
PTH1    PATH = PATH NP ↑ ↑
        PATH ↑ ¬↑ = ↑¬ ↑
        NP ↑¬↑ =
PTH2    $(PNAME ↑CODE↑ NP) ↑IF↑                               :S(PTH30)
        $(PNAME ↑CODE↑ NP) ↑JMP↑                              :S(PTH4)
        $(PNAME ↑CODE↑ NP) ↑CASE↑                             :S(PTH70)
        $(PNAME ↑CODE↑ NP) ↑HALT↑                             :S(PTH23)
        NP = NP + 1
        $(PNAME ↑CODE↑ NP)  ↑EXIT↑                            :S(PTH22)
        IDENT($(PNAME ↑AS↑ NP))                               :S(PTH1)
PTH22   PATH = PATH NP ↑ ↑
PTH23   IDENT(PASSIF)                                         :F(PTH57)
PTH12   PASSIF =
        CALL = ASSERTNS(X)
PTH60   NP = PATHBGN
        ZZZ = 0
        GT(III,0)                                             :S(PTH1)
PTH6    NP = NP + 1
        $(PNAME ↑CODE↑ NP)     ↑EXIT↑                         :S(RETURN)
        IDENT($(PNAME ↑AS↑ NP))                               :S(PTH6)F(PTH0)
PTH11   PATH = PATH NP ↑ ↑
  .     PATH ↑ ¬↑ = ↑¬ ↑
        NP ↑¬↑ =
        $(PNAME ↑CODE↑ NP) ↑EXIT↑                             :S(PTH57)
        IDENT($(PNAME ↑AS↑ NP))                               :S(PTH2)F(PTH57)
PTH30   LT(ZZZ,III)                                           :S(PTH55)
PTH3    W = $(PNAME NP)
        W BREAK(↑,↑) . TW  ↑,↑ =
        W BREAK(↑,↑)    ↑,↑ =
        W BREAK(↑,↑) . FW ↑,↑ =
        III = III + 1
        $(IFLVR III) = TW ↑ ↑          ↑¬↑ FW ↑ ↑    :(PTH55)
PTH55   ZZZ = ZZZ + 1
         PASSIF = ↑PASSIF↑
        $(IFLVR ZZZ) BREAK(↑ ↑) . NP                          :(PTH11)
PTH56   III = GT(III,1) III - 1                               :S(PTH57)
        III = 0                                               :(PTH12)
PTH57   $(IFLVR III) BREAK(↑ ↑) ↑ ↑ =                         :(PTH54)
PTH54   IDENT($(IFLVR III))                                   :S(PTH56)F(PTH12)
PTH4    $(PNAME NP) BREAK(↑,↑) . NP                           :(PTH11)
PTH70   LT(ZZZ,III)                                           :S(PTH55)
        III = III + 1
        $(↑NEGCASE↑ NP) =
        W = $(PNAME NP)
PTH7    W BREAK(↑,↑) . TW ↑,↑ =
        W BREAK(↑,↑) . P ↑,↑ =                                :F(PTH71)
        $(↑NEGCASE↑ NP) = $(↑NEGCASE↑ NP) P
PTH71   $(IFLVR III)  = $(IFLVR III) TW ↑ ↑
        IDENT(W)                                              :F(PTH7)
PTH73   $(IFLVR III) ↑ ↑ TW ↑ ↑ = ↑ ¬↑     TW ↑ ↑  :(PTH55)
DEFASN  DEFINE(↑ASSERTNS(X)↑,↑PTHEE↑)                         :(GOES)
PTHEE   CALL = COUNTR(X)
        PHD = 0
        WHD = 0
```

```
ASN56     NRW ↑.0↑ =                                              :S(ASN56)
ASN55     NRW LEN(90)                                             :S(ASN68B)
          OUTPUT = NRW
ASN57     OUTPUT =
          IDENT(W)                                                :F(ASNA)
          BGNZERO =                                               :(ASN0)
ASN533    OUTPUT = HP ↑.1            TRUE↑                        :(ASNHH)
ASN53     OUTPUT = HP ↑.1            TRUE↑                        :(ASN0)
ASN68B    CALL = BRKLEN(X)                                        :(ASN57)
ASN7      IDENT(NEG)                                              :S(ASN0)
          RW = $(↑NEGCASE↑ HP) ↑).↑
          NEG =
          RW ↑=↑ = ↑≠↑                                            :(ASN30)
ASN71     RW = $(PNAME ↑CASE↑ HP)  ↑).↑
          RW LEN(11) . B
          B ↑     ↑ = ↑(PRV)↑
          CALL = NEWRSIDE(X)
          P LEN(11) . B
ASN73     OUTPUT = NRW
          OUTPUT =                                                :(ASN011)
ASN8      $(PNAME HP) BREAK(↑,↑) . CWORD
          WRALT = $(↑ID ↑ CWORD ↑CTR↑)
          BWRALT = WRALT + 1
          EQ(WRALT,0)                                             :S(ASN8C)
          WRALT = ↑.↑ WRALT                                       :(ASN8H)
ASN8C     WRALT =
ASN8H     CHUND = $(CWORD ↑ BOUND↑) + 1
          $(PNAME ↑CODE↑ HP) ↑WRITE↑                              :S(ASN88)
          EQ(RHD,0)                                               :F(ASN8E)
          CALT =                                                  :(ASN8G)
ASN8E     CALT = ↑.↑ RHD
ASN8G     RHD = RHD + 1
          $(↑ID ↑ CWORD ↑CTR↑) = BWRALT
          OUTPUT = L  ↑:REOF(:RDHD↑   CALT ↑+1) ↱ ↑
                                      CWORD ≠.≠  BWRALT   ≠(0)=↑T≠
     .               ↑ ∧ (1≤$≤↑  $(CWORD ↑ BOUND↑) ↑ ↱ ↑   CWORD ↑.↑
     .                  BWRALT ↑($)=↑  CWORD      WRALT ↑($))     ↑
     .    OUTPUT = ↑                 ↑ ↑¬:REOF(:RDHD↑   CALT ↑+1) ↱ ↑
                 CWORD ↑.↑ BWRALT  ≠(0)=↑F ∧ ≠   ≠(1≤$≤MIN(≠
     .               $(CWORD ↑ BOUND↑) ↑,80) ↱ ↑   CWORD ↑.↑
     .              BWRALT ↑($)=:RDFL(:RDHD↑   CALT ↑+1,$))↑
     .    OUTPUT = ↑                          ∧ (↑ CHUND ↑≤$≤=0 ↱ ↑
                    CWORD ↑.↑  BWRALT  ↑($)=↑ CWORD      WRALT ↑($))↑
     .    P = ↑              :RDHD.↑  RHD  ↑=(:RDHD↑  CALT ↑)+1↑
          OUTPUT = P                                              :(ASNP)
ASN88     EQ(WHD,0)                                               :F(ASN8J)
          CALT =                                                  :(ASN8K)
ASN8J     CALT = ↑.↑ WHD
ASN8K     WHD = WHD + 1
          OUTPUT = L CWORD WRALT ≠(0)=↑T ↱ :WEOF(:WTHD≠ CALT ↑+1)↑
          OUTPUT = ↑                ↑  CWORD      WRALT ↑(0)≠↑ ≠↑T ↱ ¬≠
                ↑:WEOF(:WTHD↑   CALT ↑+1)↑       ↑ ∧ (1≤$≤MIN(↑
     .             $(CWORD ↑ BOUND↑)    ↑,132)↑  ↑ ↱ :WTFL(:WTHD↑
     .            CALT  ↑+1,$)=↑ CWORD      WRALT ↑($))↑
     .
```

```
         HHTOKEN = HTOKEN
         HTOKEN = TOKEN
         HWORD = WORD
         HKEEP = KEEP
GOE      X +NOPRINT+                                 :S(GOES)
         OUTPUT = STACK
         OUTPUT = +                                  + CARD     :(GOES)
*        RECOVERY ROUTINE  FROM ERROR SYNTAX
ER       TRY =
         V =
         TOKEN +EXIT+                                :S(GOR5)
         IDENT($(PTN TOKEN))                         :F(GOR6)
         CARD BREAK(+;+) . V +;+ =                   :S(GOR0)
         CARD +;+ =                                  :S(GOR0)
         V = CARD
         CARD =                                      :(GOR1)
GOR5     IDENT(FORWORDLB)                            :S(GOR6)
         OUTPUT = +ERR <UNDEFINED LABEL + FORWORDLB +> +
         FORWORDLB =                                 :(GOR6)
GOR0     V = V  +;+
GOR1     IDENT(I.+EOR+)                              :S(TOKS2)
GOR6     EROR = EROR + 1
         SCARD LEN(4) =
         SCARD = +ERR+  EROR SCARD  + <ERR SYNTX> +    V
         IDENT(PROHP)                                :S(GOR2)
         STACK LEN(7) . C
         C +BODY+                                    :S(GOR4)
GOR7     STACK LEN(3) . P LEN(4) . C = C             :F(GOR4)
         C +BODY+                                    :S(GOR8)
         STACK BREAK(+01+) =                         :S(GOR7)F(GOR4)
GOR8     STACK = P STACK                             :(GOES)
GOR4     STACK = +032BODY028;022ID017PROCEDURE009;001DECSEQ000+ :(GOES)
GOR2     IDENT($(+PTN+ TOKEN))                       :F(GOR3)
         STACK = +009;001DECSEQ000+                  :(GOES)
GOR3     PROHP = 0                                   :(GOR4)
*        INADEQUATE STATE
GON      TRY +TRY+                                   :S(GON1)
         CALL = TOKENS(X)
GON1     IDENT($(TOKEN L))                           :F(GOC)
         TRY = +TRY+
*        REDUSE STATE
GOP      C = $(+REDUCE+ L)
         LEFT = $(+REDLEFT+ L)
GOK      STACK LEN(3) =
         STACK BREAK(+01+) =
         C = GT(C.1) C - 1                           :S(GOK)
GOL1     IDENT(LEFT.+PROGRAM+)                       :S(GOMM)
         STACK LEN(3) . L
         IDENT($(LEFT L))                            :S(ER)
         +ALTSEQ+ LEFT                               :F(GOM)
         KSUCER = LEFT
GOM      STACK = $(LEFT L) LEFT STACK                :(GOE)
GOMM     OUTPUT = SCARD
         OUTPUT =
*        PRINT OUT SEMANTIC ERRORS
TOKS2    IDENT(KEEPPRONAME)                          :S(TOKS3)
         OUTPUT = +UNDEFINED PROCEDURE NAME + KEEPPRONAME      :(END)
TOKS3    GT(EROR.0)                                  :S(END)
*        VARIFICATION START
PTH000   PROCEDURENAME BREAK(+ +) . PNAME + + =            :F(END)
         EJECT = 1
         OUTPUT = + NUCLEUS VERIFICATION CONDITION GENERATOR    +
               +        VERSION  I  + DATE
*
         CALL = PATHNASSERT(X)                       :(PTH000)
END
```

APPENDIX B

NUCLEUS PARSE TABLE

```
000DECSEQ001DEC002SIMPLEDEC003ARRAYDEC004TYPE005INTEGER006BOOLEAN007CHARACTER008
001:009
002 DECSEQ→DEC 199
003.010 DEC→SIMPLEDEC 199
004.011 DEC→ARRAYDEC 199
005ID012ARRAY013
006 TYPE→INTEGER 199
007 TYPE→BOOLEAN 199
008 TYPE→CHARACTER 199
009PROCSEQ014DEC015PROC016SIMPLEDEC003ARRAYDEC004PROCEDURE017TYPE005INTEGER006BO
009OLEAN007CHARACTER008
010ID018
011ID019
012 SIMPLEDEC→TYPE ID 199
013ID020
014:021
015 DECSEQ→DECSEQ : DEC 199
016 PROCSEQ→PROC 199
017ID022
018 SIMPLEDEC→SIMPLEDEC , ID 199
019(023
020(024
021STARTPT025PROC026START027PROCEDURE017
022:028
023INTEGERN029
024INTEGERN030
025 PROGRAM→DECSEQ : PROCSEQ : STARTPT 199
026 PROCSEQ→PROCSEQ : PROC 199
027ID031
028BODY032ASSERTION033LABELLEDSTMT034STMT035ID036CELLREF037GO038IF039WHILE040CAS
028E041ENTER042READ043WRITE044RETURN045NOP046HALT047
029]048
030]049
031 STARTPT→START ID 199
032EXIT050ASSERTION051LABELLEDSTMT052STMT035ID036CELLREF037GO038IF039WHILE040CAS
032E041ENTER042READ043WRITE044RETURN045NOP046HALT047
033 BODY→ASSERTION 199
034:053
035 LABELLEDSTMT→STMT 199
036:054(055 CELLREF→ID 199
037:=056
038TO057
039EXP058ANDEXP059NOTEXP060RELEXP061-062BINADEXP063MULTEXP064UNADEXP065PRIMARY06
0396ADOP067INTEGERN068TRUE069FALSE070CH071CELLREF072(073INTEGER074BOOLEAN075CHAR
039ACTER076+077-078ID079
040EXP080ANDEXP059NOTEXP060RELEXP061-062BINADEXP063MULTEXP064UNADEXP065PRIMARY06
0406ADOP067INTEGERN068TRUE069FALSE070CH071CELLREF072(073INTEGER074BOOLEAN075CHAR
040ACTER076+077-078ID079
041EXP081ANDEXP059NOTEXP060RELEXP061-062BINADEXP063MULTEXP064UNADEXP065PRIMARY06
0416ADOP067INTEGERN068TRUE069FALSE070CH071CELLREF072(073INTEGER074BOOLEAN075CHAR
041ACTER076+077-078ID079
042ID082
043ID083
044ID084
045 STMT→RETURN 199
046 STMT→NOP 199
047 STMT→HALT 199
048 ARRAYDEC→ARRAYDEC , ID [ INTEGERN ] 199
049 ARRAYDEC→TYPE ARRAY ID [ INTEGERN ] 199
050 PROC→PROCEDURE ID : BODY EXIT 199
051 BODY→BODY ASSERTION 199
```

```
098 RELATIONOP→> 199
099 RELATIONOP→≥ 199
100 RELATIONOP→= 199
101 RELATIONOP→≠ 199
102UNADEXP111→PRIMARY066ADOP067INTEGERN068TRUE069FALSE070CH071CELLREF072(073INTEG
102ER074BOOLEAN075CHARACTER076+077-078ID079
103 MULTOP→* 199
104 MULTOP→/ 199
105 MULTOP→÷ 199
106 UNADEXP→ADOP PRIMARY 199
107)120∨091
108EXP121ANDEXP059NOTEXP060RELEXP061-062BINADEXP063MULTEXP064UNADEXP065PRIMARY06
1086ADOP057INTEGERN068TRUE069FALSE070CH071CELLREF072(073INTEGER074BOOLEAN075CHAR
108ACTER076+077-078ID079
109EXP122ANDEXP059NOTEXP060RELEXP061-062BINADEXP063MULTEXP064UNADEXP065PRIMARY06
1096ADOP057INTEGERN068TRUE069FALSE070CH071CELLREF072(073INTEGER074BOOLEAN075CHAR
109ACTER076+077-078ID079
110EXP123ANDEXP059NOTEXP060RELEXP061-062BINADEXP063MULTEXP064UNADEXP065PRIMARY06
1106ADOP057INTEGERN068TRUE069FALSE070CH071CELLREF072(073INTEGER074BOOLEAN075CHAR
110ACTER076+077-078ID079
111BODY124ASSERTION033LABELLEDSTMT034STMT035ID036CELLREF037GO038IF039WHILE040CAS
111E041ENTER042READ043WRITE044RETURN045NOP046HALT047
112ALTSEQ125ALT126INTEGERN127
113 CELLREF→ID [ EXP ] 199
114ELSE128FI129ASSERTION051LABELLEDSTMT052STMT035ID036CELLREF037GO038IF039WHILE0
11440CASE041ENTER042READ043WRITE044RETURN045NOP046HALT047
115∧092 EXP→EXP ∨ ANDEXP 199
116 ANDEXP→ANDEXP ∧ NOTEXP 199
117ADOP095+077-078 RELEXP→BINADEXP RELATIONOP BINADEXP 199
118MULTOP102*103/104÷105 BINADEXP→BINADEXP ADOP MULTEXP 199
119 MULTEXP→MULTEXP MULTOP UNADEXP 199
120 PRIMARY→( EXP ) 199
121)130∨091
122)131∨091
123)132∨091
124ELIHW133ASSERTION051LABELLEDSTMT052STMT035ID036CELLREF037GO038IF039WHILE040CA
124SE041ENTER042READ043WRITE044RETURN045NOP046HALT047
125ELSE134ESAC135ALT136INTEGERN127
126 ALTSEQ→ALT 199
127:137
128BODY138ASSERTION033LABELLEDSTMT034STMT035ID036CELLREF037GO038IF039WHILE040CAS
128E041ENTER042READ043WRITE044RETURN045NOP046HALT047
129 STMT→IF EXP THEN BODY FI 199
130 PRIMARY→INTEGER ( EXP ) 199
131 PRIMARY→BOOLEAN ( EXP ) 199
132 PRIMARY→CHARACTER ( EXP ) 199
133 STMT→WHILE EXP DO BODY ELIHW 199
134BODY139ASSERTION033LABELLEDSTMT034STMT035ID036CELLREF037GO038IF039WHILE040CAS
134E041ENTER042READ043WRITE044RETURN045NOP046HALT047
135 STMT→CASE EXP OF ALTSEQ ESAC 199
136 ALTSEQ→ALTSEQ ALT 199
137BODY140ALT141ASSERTION033LABELLEDSTMT034INTEGERN127STMT035ID036CELLREF037GO03
1378IF039WHILE040CASE041ENTER042READ043WRITE044RETURN045NOP046HALT047
138FI142ASSERTION051LABELLEDSTMT052STMT035ID036CELLREF037GO038IF039WHILE040CASE0
13841ENTER042READ043WRITE044RETURN045NOP046HALT047
139ESAC143ASSERTION051LABELLEDSTMT052STMT035ID036CELLREF037GO038IF039WHILE040CAS
139E041ENTER042READ043WRITE044RETURN045NOP046HALT047
140ASSERTION051LABELLEDSTMT052STMT035ID036CELLREF037GO038IF039WHILE040CASE041ENT
140ER042READ043WRITE044RETURN045NOP046HALT047 ALT→INTEGERN : BODY 199
141 ALT→INTEGERN : ALT 199
142 STMT→IF EXP THEN BODY ELSE BODY FI 199
143 STMT→CASE EXP OF ALTSEQ ELSE BODY ESAC 199
 199
```

APPENDIX C

A SAMPLE PROGRAM OF NUCLEUS LANGUAGE

```
$  THIS PROGRAM IS DESIGNED TO SHOW THE MOST FEATURES OF THE NUCLEUS LANGUAGE  $
CHARACTER ARRAY A[80], C[10], L[10];
INTEGER LAMB, COW, I, MORECOW, MORELAMB;
PROCEDURE READDATA;
ASSERT LAMB=X(1)+...+X(I-1);
ASSERT COW=Y(1)+...+X(I-1);
ASSERT IF 1≤K≤I-1, THEN ¬:REOF(K);
READ A;
WRITE A;
IF A[0] = ↑T THEN RETURN; FI;
CASE INTEGER(A[80]) OF
     4: LAMB := LAMB + 10  * (INTEGER(A[1]) - 27)
                + (INTEGER(A[2]) - 27) ;
     2: COW := COW + 10  * (INTEGER(A[3]) - 27)
                + (INTEGER(A[4]) - 27);
     ESAC;
ASSERT :RDHD=:RDHD.0+1,:WTHD=:WTHD.0+1;
ASSERT LAMB=X(1)+...+X(IF :REOF(:RDHD) THEN I-1 ELSE I);
ASSERT COW =Y(1)+...+Y(IF :REOF(:RDHD) THEN I-1 ELSE I);
ASSERT IF A[0]=↑T THEN I=FIRST K SUCH THAT :REOF(K);
ASSERT IF A[0]≠↑T  AND 1≤K≤I, THEN  ¬:REOF(K);
EXIT;
PROCEDURE MAIN;
I:=1;
COW := 0;
LAMB := 0;
ASSERT I = :RDHD = :WTHD ;
ASSERT 1≤I≤101;
ASSERT LAMB=X(1)+...+X(I-1) WHERE X(K)=THE INTEGER IN COLUMN 1-2 OF READ
RECORD K       IF COLUMN 80 HAS   ↑D AND ZERO IF NOT;
ASSERT COW=Y(1)+...+Y(I-1) WHERE Y(K)=THE INTEGER IN COLUMN 3-4 OF READ RECORD K
               IF COLUMN 80 HAS   ↑B AND ZERO OTHERWISE;
ASSERT WRITE RECORDS 1,...,I-1 ARE COPIES OF READ RECORDS 1,...,I-1;
ASSERT IF 1≤K≤I-1,THEN ¬:REOF(K);
WHILE I≤100 DO
     ENTER READDATA;
     IF A[0]=↑T THEN GO TO S; FI;
     I := I + 1;
     ELIHW;
ASSERT I=MIN(101,FIRST K SUCH THAT :REOF(K));
ASSERT LAMB=X(1)+...+X(I-1);
ASSERT COW=Y(1)+...+Y(I-1);
S: IF LAMB<COW THEN
       MORECOW := COW - LAMB;
      GO TO W;
       ELSE MORELAMB := LAMB - COW;
       FI;
L[0] := ↑F;
L[1] := CHARACTER(MORELAMB / 10 + 27);
MORELAMB := MORELAMB + 1;
L[2] := CHARACTER(MORELAMB + 27);
WRITE L;
GO TO E;
W: C[0] := ↑F;
C[1] := CHARACTER(MORECOW  / 10 + 27);
MORECOW := MORECOW + 10;
C[2] := CHARACTER(MORECOW  + 27);
WRITE C;
E: NOP;
ASSERT IF LAMB<COW THEN WRITE RECORD I+1 HAS COW-LAMB IN COLUMN 1-2;
ASSERT IF COW<LAMB THEN WRITE RECORD I+1 HAS LAMH-COW IN COLUMN 1-2;

EXIT;
START MAIN
```

APPENDIX D

A SAMPLE OUTPUT OF THE VERIFICATION CONDITION COMPILER PROGRAM - THE NUCLEUS PROGRAM CONTAINING NUMBERS IN PARENTHESES AND VERIFICATION CONDITIONS

```
NUCLEUS VERIFICATION CONDITION GENERATOR            VERSION  I

     $  THIS PROGRAM IS DESIGNED TO SHOW THE MOST FEATURES OF THE NUCLEUS LANGUAGE  $
     CHARACTER ARRAY A[80], C[10], L[10];
     INTEGER LAMB, COW, I, MORECOW, MORELAMB;
     PROCEDURE READDATA;
     (0.1)ASSERT LAMB=X(1)+...+X(I-1);
     (0.2)ASSERT COW=Y(1)+...+X(I-1);
     (0.3)ASSERT IF 1≤K≤I-1, THEN ¬:REOF(K);
     (0)READ A;
     (1)WRITE A;
     (2)IF A[0] = ↑T (3)THEN (3)RETURN; (4)FI;
     (4)CASE INTEGER(A[80]) OF
          4: (5)LAMB := LAMB + 10  * (INTEGER(A[1]) - 27)
                     + (INTEGER(A[2]) - 27) ;
         (6)2: (7)COW := COW + 10  * (INTEGER(A[3]) - 27)
                     + (INTEGER(A[4]) - 27);
         (8)ESAC;
      (9.1)ASSERT :RDHD=:RDHD.0+1,:WTHD=:WTHD.0+1;
      (9.2)ASSERT LAMB=X(1)+...+X(IF :REOF(:RDHD) THEN I-1 ELSE I);
      (9.3)ASSERT COW =Y(1)+...+Y(IF :REOF(:RDHD) THEN I-1 ELSE I);
      (9.4)ASSERT IF A[0]=↑T THEN I=FIRST K SUCH THAT :REOF(K);
      (9.5)ASSERT IF A[0]≠↑T  AND 1≤K≤I, THEN  ¬:REOF(K);
      (9)EXIT;
      PROCEDURE MAIN;
      (0)I:=1;
      (1)COW := 0;
      (2)LAMB := 0;
      (3.1)ASSERT I = :RDHD = :WTHD ;
      (3.2)ASSERT 1≤I<101;
      (3.3)ASSERT LAMB=X(1)+...+X(I-1) WHERE X(K)=THE INTEGER IN COLUMN 1-2 OF READ RECORD
      K       IF COLUMN 80 HAS   ↑D AND ZERO IF NOT;
      (3.4)ASSERT COW=Y(1)+...+Y(I-1) WHERE Y(K)=THE INTEGER IN COLUMN 3-4 OF READ RECORD K
                    IF COLUMN 80 HAS   ↑B AND ZERO OTHERWISE;
      (3.5)ASSERT WRITE RECORDS 1,...,I-1 ARE COPIES OF READ RECORDS 1,...,I-1;
      (3.6)ASSERT IF 1≤K≤I-1,THEN ¬:REOF(K);
      (3)WHILE I≤100 DO
           (4)ENTER READDATA;
           (5)IF A[0]=↑T (6)THEN (6)GO TO S; (7)FI;
           (7)I := I + 1;
           (8)ELIHW;
      (9.1)ASSERT I=MIN(101,FIRST K SUCH THAT :REOF(K));
      (9.2)ASSERT LAMB=X(1)+...+X(I-1);
      (9.3)ASSERT COW=Y(1)+...+Y(I-1);
      S: (9)IF LAMB<COW (10)THEN
              (10)MORECOW := COW - LAMB;
             (11)GO TO W;
              (12)ELSE (13)MORELAMB := LAMB - COW;
              (14)FI;
      (14)L[0] := ↑F;
      (15)L[1] := CHARACTER(MORELAMB / 10 + 27);
      (16)MORELAMB := MORELAMB + 10;
      (17)L[2] := CHARACTER(MORELAMB + 27);
      (18)WRITE L;
      (19)GO TO E;
      W: (20)C[0] := ↑F;
      (21)C[1] := CHARACTER(MORECOW  / 10 + 27);
      (22)MORECOW := MORECOW + 10;
      (23)C[2] := CHARACTER(MORECOW  + 27);
      (24)WRITE C;
      E: (25)NOP;

      (26.1)ASSERT IF LAMB<COW THEN WRITE RECORD I+1 HAS COW-LAMB IN COLUMN 1-2;
      (26.2)ASSERT IF COW<LAMB THEN WRITE RECORD I+1 HAS LAMB-COW IN COLUMN 1-2;
      (26)EXIT;
      START MAIN
```

```
READDATA
0.1          LAMB=X(1)+...+X(I-1)
0.2          COW=Y(1)+...+X(I-1)
0.3          IF 1≤K≤I-1, THEN ¬:REOF(K)
...................
0            :REOF(:RDHD+1) → A.1[0]=↑T ∧ [1≤$≤80 → A.1[$]=A[$]]
             ¬:REOF(:RDHD+1) → A.1[0]=↑F ∧ [1≤$≤MIN(80,80) → A.1[$]=:RDFL(:RDHD+1,$)]
                 ∧ [81≤$≤80 → A.1[$]=A[$]]
             :RDHD.1=(:RDHD)+1

1            A.1[0]=↑T → :WEOF(:WTHD+1)
             A.1[0]≠↑T → ¬:WEOF(:WTHD+1) ∧ [1≤$≤MIN(80,132) → :WTFL(:WTHD+1,$)=A.1[$]]
                 ∧ [81≤$<132 → :WTFL(:WTHD+1,$)=↑ ]
             :WTHD.1=(:WTHD)+1

2(PRV)       0≤0≤80
2            ¬(A.1[0]=↑T)

4(PRV)       0≤80≤80
4            INTEGER(A.1[80])=(2)

7(PRV)       0≤3≤80
7(PRV)       0≤4≤80
7            COW.1=COW+10*(INTEGER(A.1[3])-27)+(INTEGER(A.1[4])-27)

-------------------
9.1          (:RDHD.1)=:RDHD+1,(:WTHD.1)=:WTHD+1

9.2          LAMB=X(1)+...+X(IF :REOF((:RDHD.1)) THEN I-1 ELSE I)

9.3          (COW.1) =Y(1)+...+Y(IF :REOF((:RDHD.1)) THEN I-1 ELSE I)

9.4          IF (A.1)[0]=↑T THEN I=FIRST K SUCH THAT :REOF(K)

9.5          IF (A.1)[0]≠↑T AND 1≤K≤I, THEN ¬:REOF(K)
```

```
READDATA
0.1          LAMB=X(1)+...+X(I-1)
0.2          COW=Y(1)+...+X(I-1)
0.3          IF 1≤K≤I-1, THEN ¬:REOF(K)
...................
0            :REOF(:RDHD+1) → A.1[0]=↑T ∧ [1≤$≤80 → A.1[$]=A[$]]
             ¬:REOF(:RDHD+1) → A.1[0]=↑F ∧ [1≤$≤MIN(80,80) → A.1[$]=:RDFL(:RDHD+1,$)]
                 ∧ [81≤$≤80 → A.1[$]=A[$]]
             :RDHD.1=(:RDHD)+1


1            A.1[0]=↑T → :WEOF(:WTHD+1)
             A.1[0]≠↑T → ¬:WEOF(:WTHD+1) ∧ [1≤$≤MIN(80,132) → :WTFL(:WTHD+1,$)=A.1[$]]
                 ∧ [81≤$≤132 → :WTFL(:WTHD+1,$)=↑ ]
             :WTHD.1=(:WTHD)+1

2(PRV)       0≤0≤80
2            ¬(A.1[0]=↑T)

4(PRV)       0≤80≤80
4            INTEGER(A.1[80])≠(4 ∨ 2)

-------------------
9.1          (:RDHD.1)=:RDHD+1,(:WTHD.1)=:WTHD+1

9.2          LAMB=X(1)+...+X(IF :REOF((:RDHD.1)) THEN I-1 ELSE I)

9.3          COW =Y(1)+...+Y(IF :REOF((:RDHD.1)) THEN I-1 ELSE I)

9.4          IF (A.1)[0]=↑T THEN I=FIRST K SUCH THAT :REOF(K)

9.5          IF (A.1)[0]≠↑T AND 1≤K≤I, THEN ¬:REOF(K)
```

```
MAIN
3.1         I = :RDHD = :WTHD
3.2         1≤I≤101
3.3         LAMB=X(1)+...+X(I-1) WHERE X(K)=THE INTEGER IN COLUMN 1-2 OF READ RECORD K IF C
            OLUMN 80 HAS +D AND ZERO IF NOT
3.4         COW=Y(1)+...+Y(I-1) WHERE Y(K)=THE INTEGER IN COLUMN 3-4 OF READ RECORD K IF CO
            LUMN 80 HAS +B AND ZERO OTHERWISE
3.5         WRITE RECORDS 1,...,I-1 ARE COPIES OF READ RECORDS 1,...,I-1
3.6         IF 1≤K≤I-1,THEN ¬:REOF(K)
..................
3           I≤100

4           :LVL.1=(:LVL)+1
4(PRV)      0≤:LVL.1≤511
4           :RTNPT.1[$]= IF $=:LVL.1 THEN MAIN:5¬ ELSE :RTNPT[$]
4(PRV)      LAMB=X(1)+...+X(I-1)
4(PRV)      COW=Y(1)+...+X(I-1)
4(PRV)      IF 1≤K≤I-1, THEN ¬:REOF(K)
4           (:RDHD.1)=:(:RDHD)+1,(:WTHD.1)=:(:WTHD)+1
4           (LAMB.1)=X(1)+...+X(IF :REOF((:RDHD.1)) THEN I-1 ELSE I)
4           (COW.1) =Y(1)+...+Y(IF :REOF((:RDHD.1)) THEN I-1 ELSE I)
4           IF (A.1)[O]=+T THEN I=FIRST K SUCH THAT :REOF(K)
4           IF (A.1)[O]≠+T AND 1≤K≤I, THEN ¬:REOF(K)
4           :LVL.2=(:LVL.1)-1

5(PRV)      0≤O≤80
5           ¬(A.1[O]=+T)

7           I.1=I+1

------------------
3.1         (I.1) = (:RDHD.1) = (:WTHD.1)

3.2         1≤(I.1)≤101

3.3         (LAMB.1)=X(1)+...+X((I.1)-1) WHERE X(K)=THE INTEGER IN COLUMN 1-2 OF READ RECOR
            D K IF COLUMN 80 HAS +D AND ZERO IF NOT

3.4         (COW.1)=Y(1)+...+Y((I.1)-1) WHERE Y(K)=THE INTEGER IN COLUMN 3-4 OF READ RECORD
             K IF COLUMN 80 HAS +B AND ZERO OTHERWISE

3.5         WRITE RECORDS 1,...,(I.1)-1 ARE COPIES OF READ RECORDS 1,...,(I.1)-1

3.6         IF 1≤K≤(I.1)-1,THEN ¬:REOF(K)



MAIN
3.1         I = :RDHD = :WTHD
3.2         1≤I≤101
3.3         LAMB=X(1)+...+X(I-1) WHERE X(K)=THE INTEGER IN COLUMN 1-2 OF READ RECORD K IF C
            OLUMN 80 HAS +D AND ZERO IF NOT
3.4         COW=Y(1)+...+Y(I-1) WHERE Y(K)=THE INTEGER IN COLUMN 3-4 OF READ RECORD K IF CO
            LUMN 80 HAS +B AND ZERO OTHERWISE
3.5         WRITE RECORDS 1,...,I-1 ARE COPIES OF READ RECORDS 1,...,I-1
3.6         IF 1≤K≤I-1,THEN ¬:REOF(K)
..................
3           ¬(I<100)

------------------
9.1         I=MIN(101,FIRST K SUCH THAT :REOF(K))

9.2         LAMB=X(1)+...+X(I-1)

9.3         COW=Y(1)+...+Y(I-1)
```

BIBLIOGRAPHY

[1] Burstall, R. M., Formal Description of Program Structure and Semantics in First Order Logic, Machine Intelligence 5 (Meltzer and Michie, Ed.), American Elsevier Publishing Co., New York 1970.

[2] DeRemer, F. L., Simple LR(K) Grammars, University of California, Santa Cruz, Comm. of the ACM Volum3 14, Number 7, July 1971.

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

[4] Good, D. I., and London, R. L., Interval Arithmetic for the Burroughs B 5500: Four Algol Procedures and Proofs of their Correctness, Computer Sciences Technical Report No. 26, University of Wisconsin, June, 1968.

[5] Good, D. I., and Ragland, L. C., Nucleus-A Language of Provable Programs, University of Texas, In proceedings of SIGPLAN Symposium on Computer Test Methods, Prentice-Hall, 1972.

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

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

[8] London, R. L., Current State of Proving Programs Correct, Stanford University and University of Wisconsin, Proceedings of ACM Annual Conference, ACM 1972.

[9] Ragland, L. C., A Verified Program Verifier, Ph.D. Thesis University of Texas at Austin, 1973.

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