A NUCLEUS VERIFICATION CONDITION COMPILER

by

Yin-Yin Lee Wang

May 1973

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

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>II. THE NUCLEUS LANGUAGE</td>
<td>4</td>
</tr>
<tr>
<td>0. METHOD OF DEFINITION</td>
<td>4</td>
</tr>
<tr>
<td>1. DESCRIPTION OF NUCLEUS</td>
<td>11</td>
</tr>
<tr>
<td>2. BASIC ELEMENTS</td>
<td>12</td>
</tr>
<tr>
<td>3. PROGRAMS</td>
<td>13</td>
</tr>
<tr>
<td>4. DECLARATIONS</td>
<td>14</td>
</tr>
<tr>
<td>5. PROCEDURES</td>
<td>15</td>
</tr>
<tr>
<td>6. BODIES</td>
<td>15</td>
</tr>
<tr>
<td>7. ASSIGNMENT</td>
<td>16</td>
</tr>
<tr>
<td>8. GO TO</td>
<td>16</td>
</tr>
<tr>
<td>9. RETURN</td>
<td>16</td>
</tr>
<tr>
<td>10. NULL</td>
<td>17</td>
</tr>
<tr>
<td>11. IF</td>
<td>17</td>
</tr>
<tr>
<td>12. CASE</td>
<td>17</td>
</tr>
<tr>
<td>13. WHILE</td>
<td>18</td>
</tr>
<tr>
<td>14. ENTER</td>
<td>19</td>
</tr>
<tr>
<td>15. HALT</td>
<td>19</td>
</tr>
<tr>
<td>16. READ</td>
<td>19</td>
</tr>
<tr>
<td>17. WRITE</td>
<td>20</td>
</tr>
<tr>
<td>18. EXPRESSIONS</td>
<td>21</td>
</tr>
<tr>
<td>19. PRIMARIES</td>
<td>23</td>
</tr>
<tr>
<td>20. TRANSFER FUNCTIONS</td>
<td>24</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>II.1a.</th>
<th>Nucleus Program</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>II.1b.</td>
<td>Reduced Program for Declarations and Procedure ReadData</td>
<td>8</td>
</tr>
<tr>
<td>II.1c.</td>
<td>Reduced Program for Procedure Main</td>
<td>9</td>
</tr>
<tr>
<td>II.2.</td>
<td>The Virtual Program of the Nucleus Program</td>
<td>10</td>
</tr>
<tr>
<td>III.1.</td>
<td>Configuration Sets and Successor Relations of the Parser for Grammar G</td>
<td>28</td>
</tr>
<tr>
<td>III.2.</td>
<td>Characteristic Finite State Machine of Grammar G</td>
<td>29</td>
</tr>
<tr>
<td>III.3.</td>
<td>The Instruction Table for Procedure ReadData</td>
<td>32</td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION

This thesis describes the implementation of a verification condition compiler for Nucleus programs. This compiler, which is written in Snobo14 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 \( \forall \) and \( \exists \). 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\leq K \leq I-1, THEN \neg: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\leq K \leq I, then \neg: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\leq I \leq 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\leq K \leq I-1, THEN \neg:REOF(K);
(3)WHILE I\leq 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 (1)
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}
POINTLABELLEDWITH (READDATA: 4:1) = 5
JUMPTO (READDATA: 6, CASEJOINPOINT (READDATA: 4))
POINTLABELLEDWITH (READDATA: 4:2) = 7
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
A[0]  |  READ(A[0], 0)
A[1]  |  WRITE(A[1], 0)
A[80] |  EXIT(A[80])
C[0]  |  EXIT(C[0])
C[10] |  ASSIGN(C[10], l)
L[0]  |  EXIT(L[0])
L[80] |  EXIT(L[80])
COW   |  EXIT(COW)
I     |  EXIT(I)
LAMB  |  EXIT(LAMB)
MORECOW |  EXIT(MORECOW)
MORELAMB |  EXIT(MORELAMB)

**AXIOMS**

*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, \( \text{rdc} \) and \( \text{par} \), in conjunction with the productions. The function \( \text{rdc}(\text{<symbol>}) \) means the reduced program associated with \( \text{<symbol>} \). Consider the example

\[
\begin{align*}
\text{<program> } & \rightarrow \text{ <decseq> ; <procseq> ; <startpt> } \\
\text{rdc}(\text{<program>}) & = \text{ rdc( <decseq> )rdc( <procseq> )rdc( <startpt> )} \\
\text{<startpt> } & \rightarrow \text{ START ID } \\
\text{rdc}(\text{<startpt>}) & = \text{ INITIALPROCEDURE = ID}
\end{align*}
\]

This first production states that the reduced program of \( \text{<program>} \) consists of reduced programs of \( \text{<decseq>} \), \( \text{<procseq>} \), and \( \text{<startpt>} \). The second production then specifies the reduced program of \( \text{<startpt>} \). The function \( \text{par} \) 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,

\[ <\text{stmt}> \rightarrow (p)_{\text{HALT}} (p+1) \]

This means that if \( p \) is the point corresponding to the beginning of the \text{HALT} statement, then \( p+1 \) is the point corresponding to the end.

2. \underline{Basic Elements}

Nucleus programs are composed of characters from the set

\[
\{ \text{blank A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 ( [ ] ) \# / \div + - < \leq \geq > = \neq \sim \wedge \vee \equiv , ; . $ \#} \]

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.

\[
( [ ] ) \# / \div + - < \leq \geq > = \neq \sim \wedge \vee \equiv , ; . $ \#
\]

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

\[
\begin{align*}
\text{<decseq> } & \rightarrow \text{ <dec> } \\
\text{rdc(<decseq>) } & \equiv \text{ rdc(<dec>)} \\
\text{<decseq}_1 & \rightarrow \text{ <decseq>}_2 ; \text{ <dec> } \\
\text{rdc(<decseq>}_1 & ) \equiv \text{ rdc(<decseq>}_2 ) \text{ rdc(<dec>)} \\
\text{<dec> } & \rightarrow \text{ <simpledec> } \\
\text{rdc(<dec>) } & \equiv \text{ rdc(<simpledec>)} \\
\text{<dec> } & \rightarrow \text{ <arraydec> } \\
\text{rdc(<dec>) } & \equiv \text{ rdc(<arraydec>)} \\
\text{<simpledec> } & \rightarrow \text{ <type> ID } \\
\text{rdc(<simpledec>) } & \equiv \text{ SIMPLE(ID)} \\
\text{<simpledec> } & \rightarrow \text{ <simpledec>}_2 , \text{ ID } \\
\text{rdc(<simpledec>}_1 & ) \equiv \text{ rdc(<simpledec>}_2 ) \text{ SIMPLE(ID)} \\
\text{<arraydec> } & \rightarrow \text{ <type> ARRAY ID[INTEGER] } \\
\text{rdc(<arraydec>) } & \equiv \text{ ARRAY(ID,INTEGER)} \\
\text{<arraydec> } & \rightarrow \text{ <arraydec>}_2 , \text{ ID[INTEGER] } \\
\text{rdc(<arraydec>}_1 & ) \equiv \text{ rdc(<arraydec>}_2 ) \text{ ARRAY(ID,INTEGER)} \\
\text{<type> } & \rightarrow \text{ INTEGER } \\
\text{<type> } & \rightarrow \text{ BOOLEAN } \\
\text{<type> } & \rightarrow \text{ CHARACTER }
\end{align*}
\]

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 INTEGER between the brackets is the array upper bound.
5. **Procedures**

\[
\begin{align*}
\langle \text{procseq} \rangle & \rightarrow \langle \text{proc} \rangle \\
\text{rdc}(\langle \text{procseq} \rangle) & = \text{rdc}(\langle \text{proc} \rangle) \\
\langle \text{procseq} \rangle_1 \rightarrow \langle \text{procseq} \rangle_2, \langle \text{proc} \rangle \\
\text{rdc}(\langle \text{procseq} \rangle_1) & = \text{rdc}(\langle \text{procseq} \rangle_2) \text{rdc}(\langle \text{proc} \rangle) \\
\langle \text{proc} \rangle & \rightarrow \text{PROCEDURE ID; (o) body (p) EXIT} \\
\text{rdc}(\langle \text{proc} \rangle) & = \text{rdc}(\langle \text{body} \rangle) \text{ EXIT(ID:p) EXITPOINT(ID) = p}
\end{align*}
\]

The procedure sequence consists of one or more procedures. Each procedure has a procedure name, ID, followed by a \langle body \rangle 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, \ldots, 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 \langle proc \rangle production above. The association of the intermediate points in the sequence are shown in the subsequent productions that define \langle body \rangle. 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 \langle body \rangle, we use the notation \pi:p to refer to control points and \pi refers to the name of the procedure in which \langle body \rangle appears.

6. **Bodies**

\[
\begin{align*}
\langle \text{body} \rangle & \rightarrow \text{ASSERTION} \\
\text{rdc}(\langle \text{body} \rangle) & = \phi
\end{align*}
\]
$\langle \text{body} \rangle_1 \rightarrow \langle \text{body} \rangle_2$ \textbf{ASSERTION}
\[\text{rdc}(\langle \text{body} \rangle_1) = \text{rdc}(\langle \text{body} \rangle_2)\]

$\langle \text{body} \rangle \rightarrow \langle \text{labelledstmt} \rangle$;
\[\text{rdc}(\langle \text{body} \rangle) = \text{rdc}(\langle \text{labelledstmt} \rangle)\]

$\langle \text{body} \rangle_1 \rightarrow \langle \text{body} \rangle_2 \langle \text{labelledstmt} \rangle$;
\[\text{rdc}(\langle \text{body} \rangle_1) = \text{rdc}(\langle \text{body} \rangle_2) \text{ rd}(\langle \text{labelledstmt} \rangle)\]

$\langle \text{labelledstmt} \rangle \rightarrow (q)\text{ ID} : (q) \langle \text{labelledstmt} \rangle$
\[\text{rdc}(\langle \text{labelledstmt} \rangle) = (\text{POINTLABELLEDWITH}(\pi\text{:ID}=q) \text{ rd}(\langle \text{labelledstmt} \rangle)\]

A $\langle \text{body} \rangle$ 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. \textbf{Assignments}

$\langle \text{stmt} \rangle \rightarrow (p)\langle \text{cellref} \rangle := \langle \text{exp} \rangle (p+1)$
\[\text{rdc}(\langle \text{stmt} \rangle) = \text{ASSIGN}(\pi:p,\text{par}(\langle \text{cellref} \rangle),\text{par}(\langle \text{exp} \rangle))\]

The $\langle \text{cellref} \rangle$ and $\langle \text{exp} \rangle$ must be of the same type. The function \text{par}(x) gives the fully parenthesized form of its argument $x$, thus specifying the order of applying operations in evaluating $\langle \text{cellref} \rangle$ and $\langle \text{exp} \rangle$.

8. \textbf{Go To}

$\langle \text{stmt} \rangle \rightarrow (p) \text{ GO TO ID } (p+1)$
\[\text{rdc}(\langle \text{stmt} \rangle) = \text{JUMPTO}(\pi:p,\text{POINTLABELLEDWITH}(\pi:\text{ID}))\]

ID is a label which must be within the procedure $\pi$.

9. \textbf{Return}

$\langle \text{stmt} \rangle \rightarrow (p) \text{ RETURN } (p+1)$
\[\text{rdc}(\langle \text{stmt} \rangle) = \text{JUMPTO}(\pi:p,\text{EXITPOINT}(\pi))\]

A return statement is a jump to the exit of procedure $\pi$. 
10. **Null**

\[ <\text{stmt}> \rightarrow (p)_{\text{NOP}}(p+1) \]
\[ \text{rdc}(<\text{stmt}>) = \text{JUMPTO}(\pi:p,p+1) \]

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

11. **If**

\[ <\text{stmt}> \rightarrow (q)_{\text{IF}}<\text{exp}> \text{THEN}(q+1)<\text{body}>_{1}(r)\text{ELSE}(r+1)<\text{body}>_{2}\text{FI}(s) \]
\[ \text{rdc}(<\text{stmt}>) = \text{IF}(\pi:q,\text{par}(<\text{exp}>),q+1,r+1) \]
\[ \quad \text{rdc}(<\text{body}>_{1}) \]
\[ \quad \text{JUMPTO}(\pi:r,s) \]
\[ \quad \text{rdc}(<\text{body}>_{2}) \]

\[ <\text{stmt}> \rightarrow (q)_{\text{IF}}<\text{exp}> \text{THEN}(q+1)<\text{body}> \text{FI}(r) \]
\[ \text{rdc}(<\text{stmt}>) = \text{IF}(\pi:q,\text{par}(<\text{exp}>),q+1,r) \]

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

12. **Case**

\[ <\text{stmt}> \rightarrow (p)_{\text{CASE}}<\text{exp}> \text{OF}(p+1)<\text{altseq}>_{q}(q)_{\text{ESAC}}(q+1) \]
\[ \text{rdc}(<\text{stmt}>) = \text{CASE}(\pi:p,\text{par}(<\text{exp}>),\pi:q+1) \]
\[ \quad \text{rdc}(<\text{altseq}>) \]
\[ \quad \text{CASEJOINPOINT}(\pi:p) = q+1 \]

\[ <\text{stmt}> \rightarrow (p)_{\text{CASE}}<\text{exp}> \text{OF}(p+1)<\text{altseq}>_{q}(q)_{\text{ELSE}}(q+1)<\text{body}> \text{ESAC}(r) \]
\[ \text{rdc}(<\text{stmt}>) = \text{CASE}(\pi:p,\text{par}(<\text{exp}>),\pi:q+1) \]
\[ \quad \text{rdc}(<\text{altseq}>) \]
\[ \quad \text{rdc}(<\text{body}>) \]
\[ \quad \text{CASEJOINPOINT}(\pi:p) = r \]

\[ <\text{altseq}> \rightarrow <\text{alt}> \]
\[ \text{rdc}(<\text{altseq}>) = \text{rdc}(<\text{alt}>) \]
<\text{altseq}_{1} \rightarrow <\text{altseq}_{2} <\text{alt}\
\text{rdc}(<\text{altseq}_{1}) = \text{rdc}(<\text{altseq}_{2}) \text{rdc}(<\text{alt})>

<\text{alt} > \rightarrow (p) \text{\textsc{INTEGER} : (p) <\text{body} > (q)}
\text{rdc}(<\text{alt}) = \text{\textsc{INTEGER} } \in \text{CASELABELSET}(\pi : c)\
\text{POINTLABELLEDWITH}(\pi : c : \text{\textsc{INTEGER}}) = p\n\text{rdc}(<\text{body}>)
\text{JUMPTO}(\pi : q, \text{CASEJOINPOINT}(\pi : c))\n\text{where } c \text{ is the point at the beginning of the case statement.}

<\text{alt}_{1} > \rightarrow (p) \text{\textsc{INTEGER} : (p) <\text{alt}_{2} > (q)}
\text{rdc}(<\text{alt}_{1}) = \text{\textsc{INTEGER} } \in \text{CASELABELSET}(\pi : c)\
\text{POINTLABELLEDWITH}(\pi : c : \text{\textsc{INTEGER}}) = p\n\text{rdc}(<\text{alt}_{2})
\text{where } c \text{ is the point at the beginning of the case statement.}

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

13. \textbf{While}

<\text{stmt} > \rightarrow (q) \text{\textsc{WHILE}} <\text{exp}> \text{DO}(q+1) <\text{body}> (r) \text{\textsc{ELIHW}}(r+1)
\text{rdc}(<\text{stmt}>) = \text{IF}(\pi : q, \text{par}(<\text{exp}>), q+1, r+1)\n\text{rdc}(<\text{body}>)
\text{JUMPTO}(\pi : r, q)

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

14. **Enter**

\[
\begin{align*}
\text{<stmt> } & \rightarrow (q) \text{ ENTER ID}(q+1) \\
\text{rdc(<stmt>) } & = \text{ ENTER}(\pi:q, \text{ID})
\end{align*}
\]

This is a possibly recursive call of the procedure name ID. Before entering the procedure, the point \(\pi:q+1\) 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**

\[
\begin{align*}
\text{<stmt> } & \rightarrow (q) \text{ HALT}(q+1) \\
\text{rdc(<stmt>) } & = \text{ HALT}(\pi:q)
\end{align*}
\]

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

16. **Read**

\[
\begin{align*}
\text{<stmt> } & \rightarrow (q) \text{ READ ID}(q+1) \\
\text{rdc(<stmt>) } & = \text{ READ}(\pi:q, \text{ID})
\end{align*}
\]

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 ≤ i ≤ min(upper
bound of ID, record size). The remainder of the array,
if any, is left unchanged.

17. Write

<stmt> → (q)\texttt{WRITE ID}^{(q+1)}
rdc(<stmt>) = WRITE(\pi:q, 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

\[
\begin{align*}
\text{<exp>} & \rightarrow \text{<andexp>} \\
\text{par}(\text{<exp>}) & = \text{par}(\text{<andexp>})
\end{align*}
\]

\[
\begin{align*}
\text{<exp>}_1 & \rightarrow \text{<exp>}_2 \lor \text{<andexp>} \\
\text{par}(\text{<exp>}_1) & = (\text{par}(\text{<exp>}_2)) \lor (\text{par}(\text{<andexp>}))
\end{align*}
\]

\[
\begin{align*}
\text{<andexp>} & \rightarrow \text{<notexp>} \\
\text{par}(\text{<andexp>}) & = \text{par}(\text{<notexp>})
\end{align*}
\]

\[
\begin{align*}
\text{<andexp>}_1 & \rightarrow \text{<andexp>}_2 \land \text{<notexp>} \\
\text{par}(\text{<andexp>}_1) & = (\text{par}(\text{<andexp>}_2)) \land (\text{par}(\text{<notexp>}))
\end{align*}
\]

\[
\begin{align*}
\text{<notexp>} & \rightarrow \text{<relexp>} \\
\text{par}(\text{<notexp>}) & = \text{par}(\text{<relexp>})
\end{align*}
\]

\[
\begin{align*}
\text{<notexp>} & \rightarrow \neg(\text{<relexp>}) \\
\text{par}(\text{<notexp>}) & = \neg(\text{par}(\text{<relexp>}))
\end{align*}
\]

\[
\begin{align*}
\text{<relexp>} & \rightarrow \text{<binadexp>} \\
\text{par}(\text{<relexp>}) & = \text{par}(\text{<binadexp>})
\end{align*}
\]

\[
\begin{align*}
\text{<relexp>} & \rightarrow \text{<binadexp>}_1 \cdot \text{<relationop>}\text{<binadexp>}_2 \\
\text{par}(\text{<relexp>}) & = (\text{par}(\text{<binadexp>}_1)) \cdot (\text{relationop})(\text{par}(\text{<binadexp>}_2))
\end{align*}
\]

\[
\begin{align*}
\text{<binadexp>} & \rightarrow \text{<multexp>} \\
\text{par}(\text{<binadexp>}) & = \text{par}(\text{<multexp>})
\end{align*}
\]

\[
\begin{align*}
\text{<binadexp>}_1 & \rightarrow \text{<binadexp>}_2 \cdot \text{<adop>}\text{<multexp>} \\
\text{par}(\text{<binadexp>}_1) & = (\text{par}(\text{<binadexp>}_2)) \cdot \text{adop}(\text{par}(\text{<multexp>}))
\end{align*}
\]
\(<\text{multexp}\> + \text{<unadexp>}
\text{par(}<\text{multexp}>\text{)} = \text{par(}<\text{unadexp}>\text{)}

\(<\text{multexp}> \rightarrow \text{<multexp>}, \text{<multtop><unadexp>}
\text{par(}<\text{multexp}>_1\text{)} = (\text{par(}<\text{multexp}>_2\text{)})\text{<multtop>par(}<\text{unadexp}>\text{)}

\(<\text{unadexp}> \rightarrow \text{<primary>}
\text{par(}<\text{unadexp}>\text{)} = \text{par(}<\text{primary}>\text{)}

\(<\text{unadexp}> \rightarrow \text{<adop><primary>}
\text{par(}<\text{unadexp}>\text{)} = \text{<adop>par(}<\text{primary}>\text{)}

\(<\text{relationop}> \rightarrow <
\text{<relationop>} \rightarrow \leq
\text{<relationop>} \rightarrow \geq
\text{<relationop>} \rightarrow >
\text{<relationop>} \rightarrow =
\text{<relationop>} \rightarrow \\
\text{<adop>} \rightarrow +
\text{<adop>} \rightarrow -
\text{<multtop>} \rightarrow *
\text{<multtop>} \rightarrow /
\text{<multtop>} \rightarrow \\)

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 \(<\text{adop}>\) and \(<\text{multtop}>\) operands. Type boolean primaries are required for logical operands, \(\neg\), \(\wedge\), and \(\vee\). The relational operations may be applied to operands of any type, provided both operands are of the same type. If operands of type boolean or
character are used, the transfer function to type integer is applied automatically.

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Priority</th>
<th>Operand Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>+,- (unary)</td>
<td>1</td>
<td>INTEGER</td>
</tr>
<tr>
<td>*, /, +</td>
<td>2</td>
<td>INTEGER</td>
</tr>
<tr>
<td>+,- (binary)</td>
<td>3</td>
<td>INTEGER</td>
</tr>
<tr>
<td>&lt;,&lt;=, =, #, &gt;=</td>
<td>4</td>
<td>explained above</td>
</tr>
<tr>
<td>¬</td>
<td>5</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>A</td>
<td>6</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>V</td>
<td>7</td>
<td>BOOLEAN</td>
</tr>
</tbody>
</table>

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

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 remaining) by zero, and array bound violation.

If the value of expression is undefined, the execution terminates.

19. **Primaries**

\(<\text{primary}> \to \text{INTEGER}\)
par(<primary>) = INTEGER

\(<\text{primary}> \to \text{TRUE}\)
par(<primary>) = TRUE

\(<\text{primary}> \to \text{FALSE}\)
par(<primary>) = FALSE

\(<\text{primary}> \to \text{CH}\)
par(<primary>) = CH

\(<\text{primary}> \to <\text{cellref}>\)
par(<primary>) = par(<cellref>)
\[ <\text{cellref}> \rightarrow \text{ID}[<\text{exp}>] \]
\[ \text{par}(<\text{cellref}>) = \text{ID}[\text{par}(<\text{exp}>)] \]

\[ <\text{cellref}> \rightarrow \text{ID} \]
\[ \text{par}(<\text{cellref}>) = \text{ID} \]

\[ <\text{primary}> \rightarrow (<\text{exp}>) \]
\[ \text{par}(<\text{primary}>) = (\text{par}(<\text{exp}>) ) \]

\[ <\text{primary}> \rightarrow \text{INTEGER} ( <\text{exp}> ) \]
\[ \text{par}(<\text{primary}>) = \text{INTEGER} (\text{par}(<\text{exp}>) ) \]

\[ <\text{primary}> \rightarrow \text{BOOLEAN} ( <\text{exp}> ) \]
\[ \text{par}(<\text{primary}>) = \text{BOOLEAN} (\text{par}(<\text{exp}>) ) \]

\[ <\text{primary}> \rightarrow \text{CHARACTER} ( <\text{exp}> ) \]
\[ \text{par}(<\text{primary}>) = \text{CHARACTER} (\text{par}(<\text{exp}>) ) \]

A primary may be a constant token such as INTEGER, 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:

\[ \text{boolofchar}(x) = \text{boolofint}(\text{intofchar}(x)) \]

\[ \text{boolofint}(x) = \text{false if abs}(x) \mod 2 = 0 \]
\[ = \text{true if abs}(x) \mod 2 = 1 \]

\[ \text{charofbool}(x) = \text{charofint}(\text{intofbool}(x)) \]

\[ \text{charofint}(x) = "\" \text{if abs}(x) \mod 64 = 0 \]
\[ = "\text{A}\" \text{if abs}(x) \mod 64 = 1 \]
\[ \vdots \]
\[ = "\#\" \text{if abs}(x) \mod 64 = 63 \]
\texttt{intofbool(x) = 0 if x = false}
\texttt{= 1 if x = true}

\texttt{intofchar(x) = 0 if x = " "}
\texttt{= 1 if x = "A"}
\texttt{::}
\texttt{= 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 written 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 accommodate 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 \( G = (\{ , a, +, - \}, \{ S, E \}, S, P) \) be a context-free grammar where \( \{ , a, +, - \} \) is the set of terminal symbols \( V_t \), \( \{ S, E \} \) is the set of non-terminal symbols \( V_n \), \( S \) is the starting symbol, and \( P \) the set of productions

\[
\begin{align*}
\#1 & & S \rightarrow E - \\
\#2 & & E \rightarrow a + E \\
\#3 & & E \rightarrow a
\end{align*}
\]

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.

<table>
<thead>
<tr>
<th>State name</th>
<th>Configuration set</th>
<th>Successor</th>
<th>Next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{S \rightarrow .E \rightarrow}</td>
<td>_</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>{S \rightarrow .E \rightarrow}</td>
<td>_</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>\ E \rightarrow a+E</td>
<td>_</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>\ E \rightarrow a}</td>
<td>_</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>{S \rightarrow .E. \rightarrow}</td>
<td>_</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>{E \rightarrow a.+E</td>
<td>_</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>\ E \rightarrow a}</td>
<td>_</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>{E \rightarrow a.+E</td>
<td>_</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>\ E \rightarrow a+E</td>
<td>_</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>\ E \rightarrow .a}</td>
<td>_</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>{E \rightarrow a+E.}</td>
<td>_</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>{S \rightarrow .E \rightarrow}</td>
<td>_</td>
<td>#2</td>
</tr>
<tr>
<td>7</td>
<td>{ }</td>
<td>_</td>
<td>#1</td>
</tr>
</tbody>
</table>

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 \( \{-\} \) 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 grammar, 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
The instruction table shown below for procedure READDATA.

<table>
<thead>
<tr>
<th>p</th>
<th>&quot;READDATA CODE p&quot;</th>
<th>&quot;READDATA p&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>READ</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>WRITE</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>IF</td>
<td>3,A[0] = T,4</td>
</tr>
<tr>
<td>3</td>
<td>JMP</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>CASE</td>
<td>5,INTEGER(A[80]) = 4,7,INTEGER(A[80]) = 2,9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LAMB,LAMB+10*INTEGER(A[1])-27+</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER(A[2])-27</td>
</tr>
<tr>
<td>5</td>
<td>:=</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>JMP</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>:=</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>JMP</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>EXIT</td>
<td>9</td>
</tr>
</tbody>
</table>

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. **Verification Conditions**

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

\[ \begin{align*}
A & \quad \text{.........} \\
B & \quad \text{--------} \\
C & \end{align*} \]

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 preceding 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. ASSIGN\((P,q,N,V)\)

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

\[
\begin{align*}
16(PRV) & \quad 10 \neq 0 \\
16 & \quad \text{MORELAMB.}2=\text{MORELAMB.}1+10
\end{align*}
\]

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

\[
\begin{align*}
14(PRV) & \quad 0 \leq 0 \leq 10 \\
14 & \quad \text{L.}1[\$] = \text{IF } \$ = 0 \text{ THEN } +F \text{ ELSE L[\$]}
\end{align*}
\]

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. \textbf{CASE(P:q,E,f)}

The term is either

\[ E^* = \text{the element of CASELABELSET(P:q)} \]

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

\[ E^* \neq \text{any of the elements of CASELABELSET(P:q)} \]

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

\[
\begin{align*}
4 \text{(PRV)} & \quad 0 \leq 80 \leq 80 \\
4 & \quad \text{INTEGER}(A[80]) = (4)
\end{align*}
\]

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

\[
\begin{align*}
4 \text{(PRV)} & \quad 0 \leq 80 \leq 80 \\
4 & \quad \text{INTEGER}(A[80]) = (2)
\end{align*}
\]

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

\[
\begin{align*}
4 \text{(PRV)} & \quad 0 \leq 80 \leq 80 \\
4 & \quad \text{INTEGER}(A[80]) \neq (4 \times 2)
\end{align*}
\]

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

4.3. \text{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,

\text{IF(MAIN:3,I \leq 100,4,9)} in path(3 4 5 7 3) has the term

3 \quad I \leq 100

path(3 9) has the term

3 \quad \neg(I \leq 100)
4.4. \textbf{JUMPTO}(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, \textbf{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 6 9) with no terms shown for point 6.

4.5. \textbf{READ}(P,q,A)

The terms are

\[ :\text{REOF}(:\text{RDHD} . a :\text{RDHD} +1) \rightarrow A . (a +1)[0]=\top \]
\[ \land [1 \leq a \leq \text{bound}(A) \rightarrow A . (a +1)[a]=A [a] ] \]

\[ \neg :\text{REOF} (:\text{RDHD} . a :\text{RDHD} +1) \rightarrow A . (a +1)[0]=\bot \]
\[ \land [1 \leq a \leq \text{MIN}(\text{readsize}, \text{bound}(A)) \rightarrow A . (a +1)[a]=:\text{RDFL} (:\text{RDHD} . a :\text{RDHD} +1)[a] ] \]
\[ \land [(\text{readsize}+1) \leq a \leq \text{bound}(A) \rightarrow A . (a +1)[a]=A [a] ] \]

\[ :\text{RDHD} . (a :\text{RDHD} +1)=(:\text{RDHD} . a :\text{RDHD}) +1 \]

For example, \textbf{READ}(\text{READDATA}:0,A) has the term

\[ 0 :\text{REOF} (:\text{RDHD} +1) \rightarrow A . 1[0]=\top \land [1 \leq a \leq 80 \rightarrow A . 1[a]=A [a] ] \]
\[ \neg :\text{REOF} (:\text{RDHD} +1) \rightarrow A . 1[0]=\bot \]
\[ \land [1 \leq a \leq \text{MIN}(80,80) \rightarrow A . 1[a]=:\text{RDFL} (:\text{FDHD} +1,a) ] \]
\[ \land [81 \leq a \leq 80 \rightarrow A . 1[a]=A [a] ] \]

\[ :\text{RDHD} . 1=(:\text{RDHD}) +1 \]

This means that if the next read record is an end-of-file, then "\top" is placed in the element zero of the read array A, and the rest of the elements in the array A are unchanged. \textbf{REOF} is the function for read end-of-file, \textbf{RDHD} is read head, a pointer to the next record to be read, and \textbf{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
"r" 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 $81 \leq r \leq 80 \Rightarrow A.1[r] = A[r]$ 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. WRITE(P; q, a)

The terms are

\[
\begin{align*}
A.&(a_r+1)[0]=T \rightarrow \text{WEOF}(:\text{WTBD}.a,\text{WTBD}+1) \\
A.&(a_r+1)[0] \neq T \rightarrow \neg \text{WEOF}(:\text{WTBD}.a,\text{WTBD}+1) \\
&\Lambda[1 \leq r \leq \text{MIN(bound}(A),\text{writesize}) \rightarrow \\
&\quad \text{:WTFL}(:\text{WTBD}.a,\text{WTBD}+1,r) = A.\Lambda[r]] \\
&\Lambda[\text{bound}(A)+1 \leq r \leq \text{writeSize} \rightarrow \\
&\quad \text{:WTFL}(:\text{WTBD}.a+1)=T] \\
&\text{WTBD}.(a_r) = (:\text{WTBD}.a,\text{WTBD}+1)
\end{align*}
\]

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

\[
\begin{align*}
1 \rightarrow A.1[0]=T \rightarrow \text{WEOF}(:\text{WTBD}+1) \\
A.1[0] \neq T \rightarrow \neg \text{WEOF}(:\text{WTBD}+1) \\
&\Lambda[1 \leq r \leq \text{MIN}(80,132) \rightarrow \\
&\quad \text{:WTFL}(:\text{WTBD}+1,r) = A.1[r]] \\
&\Lambda[81 \leq r \leq 132 \rightarrow \\
&\quad \text{:WTFL}(:\text{WTBD}+1,r) = T] \\
&\text{WTBD}.1 = (:\text{WTBD})+1
\end{align*}
\]

For a WRITE only the alteration counter for :WTBD 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. \textbf{ENTER}(P:q,H)

The terms are

\begin{align*}
: \text{LVL.}(a: \text{LVL}+1) &= (: \text{LVL}.a: \text{LVL}) + 1 \\
(\text{PRV}) \ 0 &\leq : \text{LVL}.(a: \text{LVL}+1) \leq \text{maximum return point stack size} \\
: \text{RTNPT.}(a: \text{RTNPT}+1)[\$] &= \text{IF } \$ = : \text{LVL}.(a: \text{LVL}+1) \\
&\quad \text{THEN } P:(q+1) \text{ ELSE } : \text{RTNPT}[\$] \\
(\text{PRV}) \ I^* \ I^*+1 \\
: \text{LVL.}(a: \text{LVL}+2) &= (: \text{LVL}.(a: \text{LVL}+1)) - 1
\end{align*}

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 \(: \text{RDHD}, : \text{WTHD}, : \text{LVL}, : \text{RTNPT}, \) and \(: \text{STEP} \) 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,

\textbf{ENTER}(\text{MAIN:}4, \text{READDATA}) \text{ has term}

4
4(\text{PRV})
4: \text{LVL.}1= (: \text{LVL}) + 1
4(\text{PRV})
0 \leq : \text{LVL.}1 \leq 511
4: \text{RTNPT.}1[\$] = \text{IF } \$ = : \text{LVL.}1 \text{ THEN } \text{MAIN:}5 \text{ ELSE } : \text{RTNPT}[\$]
4(\text{PRV})
\text{LAMB} = X(1) + \ldots + X(I-1)
4(\text{PRV})
\text{COW} = Y(1) + \ldots + Y(I-1)
4(\text{PRV})
\text{IF } 1 \leq k \leq I-1, \text{ THEN } : \text{REOF}(K)
4
(: \text{RDHD.}1) = (: \text{RDHD}) + 1, (: \text{WTHD.}1) = (: \text{WTHD}) + 1
4
(: \text{LAMB.}1) = X(1) + \ldots + X(\text{IF } : \text{REOF}(: : \text{RDHD.}1)) \text{ THEN } I-1 \text{ ELSE } 1
4
(: \text{COW.}1) = Y(1) + \ldots + Y(\text{IF } : \text{REOF}(: : \text{RDHD.}1)) \text{ THEN } I-1 \text{ ELSE } 1
4
\text{IF } (A.1)[0] = T \text{ THEN } I = \text{FIRST} k \text{ SUCH THAT } : \text{REOF}(K)
4
\text{IF } (A.1)[0] \neq T \text{ AND } 1 \leq K \leq I, \text{ THEN } \neg : \text{REOF}(K)
4: \text{LVL.}2= (: \text{LVL.}1) - 1

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

```
1  OUTPUT(*JECHE++, *OUTPUT++, ++)
2  JECHE = 1
3  x = *OPRINT*
4  ERROR = 0
5  II = 0
6  IFLR = *IFLV* = 0
7  ELSELEVE = *ELSELEVEN*
8  IFLEVE = *IFLEVEN*
9  STACK = *OOP*
10 PRO0 = ARRAY(*0:143*)
11 P = TRIM(INPUT)
12 P LFN(3) = L = 15
13 L = 15
14 PRO1 LPARSET(L) = PARSET(L) P
15 PRO2 PARSET(L) LEN(1) * w
16 PRO3 LEN(3) * SST = :S(ENDP)
17 S(L) = SST
18 PRO4 PARSET(L) LEN(1) * w
19 S(*INAD+ L) = L
20 PRO5 C = 0
21 PRO6 PARSET(L) + + = 1
22 PRO7 PARSET(L) HREAK(*++) + LT + + =
23 PRO8 LPARCEL(L) = LT
24 PRO9 C = C + 1
25 PRO10 SST = +NUCLEUS VERIFICATION CONDITION GENERATOR +
26 END
27 REINIT = +*INTEGER+
28 RESPOolean = +*BOOLAN*
29 RESCARe = +*CHARACTER*
30 RESWAY = +*ARRAY*
31 RESPROCEDURE = +*PROCEDURE+
32 RESEXIT = +*EXIT*
33 RESO = +*O*
34 RESI = +*I*
35 RESTHEN = +*THEN*
36 RESELSE = +*ELSE*
37 RESWHILE = +*WHILE*
38 RESIF = +*IF*
39 RESRETURN = +*RETURN*
40 RESMDP = +*MDP*
41 RESELIM = +*ELIM*
42 RESREUSE = +*TRUE*
```

46
**INSERT ABSOLUTE OVERLAY GENERATION HERE**

```
47

54  *ESFALSE = *FALSE*
55  RES TART = *STANT*
56  *ESF I = *FI*
57  *ESESAC = *LSAC*
58  RES HALT = *HALT*
59  *ESCH = *CH*
60  *ESCASE = *CASE*
61  RESNP = *CF*
62  PTA IN = *GO*
63  PTA RETURN = *RETURN*
64  PTA WHIL E = *WHILE*
65  PTA IF = *IF*
66  PTA CASE = *CASE*
67  PTA IF THEN = *IF THEN*
68  PTA READ = *READ*
69  PTA WRITE = *WRITE*
70  PTA NOP = *NOP*
71  PTA ELSE = *ELSE*
72  PTA EXIT = *EXIT*
73  PTA Halt = *Halt*

74  DEFINE(TOKENS(X)+*TOK*)
75  IDENT(CARD)
76  IDENT(Y+*EOF+*)
77  TK00 SCARD LEN(133)
78  OUTPUT = SCARD
79  SCARD = INPUT
80  I = *FOR*
81  TK04 SCARD LEN(90) * W =
82  OUTPUT = W
83  TK05 SCARD LEN(90) * W =
84  OUTPUT = + +
85  TK06 SCARD = + + SCARD
86  TK01 CARD = CARD SCARD
87  SCARD =
88  IDENT(Y+*EOF+*)
89  CARD RTAB(L) LEN(L) . B
90  W +
91  CARD = TP13(CARD) +
92  X *NOPRINT*
93  TK02 OUTPUT = CARD
94  TK04 KEEPBLANK =
95  TK04 CARD LEN(L) . B
96  B +
97  KEEPBLANK = KEEPBLANK +
98  CARD =
99  CARD *S* =
100  TK03 SCARD = SCARD KEEPBLANK *S*
101  TK03 CARD AHEAD(*S*) . V *S* =
102  SCARD = SCARD V *S*
103  TK03 OUTPUT = SCARD CARD
104  SCARD =
105  TK04 CARD = INPUT
106  IDENT(Y+CARD)
107  IDENT(Y+*EOF+*)
108  CARD NELIMITED
109  ACMD = CARD
110  TK05 CARD LEN(L) . W
111  NELIMITED
112  W *X*
113  W *X*
114  CARD LEN(N) . WORD
115  IDENT(Y+*EOF+*)
116  TOKEN = +CH+
117  TK06 CARD AHEAD(0EL) . WORD
118  IDENT(Y+*WORD+*ASSERT+*)
119  IDENT(Y+*EXIT+* WORD+*)
120  WORD LETTER
121  WORD LEN(L) . W
122  W LETTER
123  TOKEN = +10+

124  IF (TK01)
125  IF (TK04)
126  IF (TK05)
127  IF (TK06)
128  IF (TK02)
129  IF (TK04)
130  IF (TK04)
131  IF (TK04)
132  IF (TK03)
133  IF (TK03)
134  IF (TK04)
135  IF (TK07)
136  IF (TK07)
137  IF (TK07)
138  IF (RETURN)
139  IF (TK0A)
140  IF (RETURN)
141  IF (RETURN)
```

```
EXPC = $1(CHARACTER* WORD) WORD
EXPC = $1(CHARACTER ARRAY* WORD) WORD
EXPC = $1(CHARACTER* WORD) WORD
IS(INTERNAL, IF(EFPXP))

EXP1 = EXP2 * EXP3
EXP1 = EXP2 * EXP3
EXP1 = EXP2 + EXP3
IS(EXPC)

EXP10 = $1(PNAME *TENTEN*) WORD * +
EXP10 = $1(PNAME *TENTEN*) WORD * +
IS(EXPC)

EXP11 = $1(PNAME *TENTEN*) WINDO +
EXP11 = $1(PNAME *TENTEN*) WINDO +
IS(EXPC)

EXP12 = $1(PNAME *TENTEN*) WINDO +
EXP12 = $1(PNAME *TENTEN*) WINDO +
IS(EXP10)

ERXP = ERMN + 1
SCARD LEN4) = SCARD = EWYX ERMO SCARD + <WYN TYPE> + 
IS(RETURN)

DEINT = DEFINE (+INTERNAL(x)++INT) + 
IS(DEFPT)

INT1 = W = 0
* W = W
IS(NOPINT+)
IS(RETURN)

INT1 = W = 0
* W = W
IS(NOPINT+)
IS(RETURN)

DEFP = DEFINE (+POINTN(x)++PT+) + 
IS(DEFPT)+

PT = TOKEN +PHCDECIME+
TOKEN +TOKEN
IS(COMP)

INT1 = $PNAME *TOKEN
INT1 = $PNAME *TOKEN
IS(P11)

TKMN +SAN
TKMN +SAN
IS(P11)

*TP +TOKEN
*TP +TOKEN
IS(P11)

** TP +TOKEN
** TP +TOKEN
IS(P15)

/* TP +TOKEN
/* TP +TOKEN
IS(P15)

PT2 = TOKEN
PT2 = TOKEN
IS(P15)

PT2 = TOKEN
PT2 = TOKEN
IS(P15)

PT2 = TOKEN
PT2 = TOKEN
IS(P15)

PT2 = TOKEN
PT2 = TOKEN
IS(P15)

PT2 = TOKEN
PT2 = TOKEN
IS(P15)

PT2 = TOKEN
PT2 = TOKEN
IS(P15)

PT2 = TOKEN
PT2 = TOKEN
IS(RETURN)(P2)

PT2 = TOKEN
PT2 = TOKEN
IS(RETURN)(P2)

PT2 = TOKEN
PT2 = TOKEN
IS(RETURN)(P2)

PT2 = TOKEN
PT2 = TOKEN
IS(RETURN)(P2)

PT2 = TOKEN
PT2 = TOKEN
IS(RETURN)(P2)

PT2 = TOKEN
PT2 = TOKEN
IS(RETURN)(P2)

PT2 = TOKEN
PT2 = TOKEN
IS(RETURN)(P2)

PT2 = TOKEN
PT2 = TOKEN
IS(RETURN)(P2)

PT2 = TOKEN
PT2 = TOKEN
IS(RETURN)(P2)

PT2 = TOKEN
PT2 = TOKEN
IS(RETURN)(P2)

PT2 = TOKEN
PT2 = TOKEN
IS(RETURN)(P2)

PT2 = TOKEN
PT2 = TOKEN
IS(RETURN)(P2)

PT2 = TOKEN
PT2 = TOKEN
IS(RETURN)(P2)

PT2 = TOKEN
PT2 = TOKEN
IS(RETURN)(P2)

PT2 = TOKEN
PT2 = TOKEN
IS(RETURN)(P2)

PT2 = TOKEN
PT2 = TOKEN
IS(RETURN)(P2)

PT2 = TOKEN
PT2 = TOKEN
IS(RETURN)(P2)

PT2 = TOKEN
PT2 = TOKEN
IS(RETURN)(P2)

PT2 = TOKEN
PT2 = TOKEN
IS(RETURN)(P2)

PT2 = TOKEN
PT2 = TOKEN
IS(RETURN)(P2)

PT2 = TOKEN
PT2 = TOKEN
IS(RETURN)(P2)

PT2 = TOKEN
PT2 = TOKEN
IS(RETURN)(P2)
* DEFINE VARIABLE COUNTER FOR ASSERTION

DEFCTR DEFCNT((COUNTH(X)+*CTR*))

CH = TYPELIST

CH1 = T PREAD(1+), WHD + = IF(RETURN)

$(*ID + WIRD) +(*CTR*) = 0

* DEFINE VARIATION CONDITION ON RIGHT HAND SIDE OF I=

DEFHNS DEFINE(NEWHSIDE(X)++NR*)

NR =

NR1 = IDENT(RW++)

iW LEN(1) * I

iW DELIMITER = $NR2

iW LETTER

K + ASSERT*

iW IWDH * uIW I* +

iW IDTH * uIW I* +

iW ILVL + uIW I* +

iW STEP + uIW I* +

iW TRNPT + uIW I* +

iW INFL: iWFL :PROF :WEOF :LCC + uIW I* + = $NR34

NR13 = NEXTW =

NR14 = NEXTW * 0*

iW IDLY((1+ID + 1W + CTR))

iW EV((1+ID + 1W + CTR) + 0)

K + ASSERT+

NRW = NRW I* + $(*ID + 1W + CTR) +

NR15 = IDENT(UPAM)

NR2 = NRW ++ (1W + W)

NR3 = + I*

NR4 = NRW ++ I*

NR5 = NRW ++ I*

NR6 = NRW ++ I*

NR7 = NRW ++ I*

NR8 = IDENT(UPAM)

NR9 = GET(MIDAM-UPAM)

NR10 = OUTPUT = B + OS55 MIDAM ++ UPAM

NR11 = UPAM =

NR12 = MIDAM =

NR13 = OUTPUT = HP ++ * ARRAY OVERFLOW+

NR14 = OUTPUT = +ZERO DEVI504+

NR100 = X + ASSERT+

NR30 = MOD = NR*

NR50 = MOD PREAD(1++$++*

NR1 = MOD LEN(1) LEN(1) 0 = 0

NR40 = MOD MOD = MOD

NR41 = MOD 0 +

NR42 = MOD PREAD(++$++$++) 0

NR50 = OUTPUT = B SAVE MOD $0*

NR51 = OUTPUT = B SAVE MOD $0*

NR52 = OUTPUT = B SAVE MOD $0*
544  EN118  WRN = WRN + INCLVL
545  EN118  WW = \$(P * CALLENTE+)  \$ (RETURN)
546  EN118  WW BREAK(+) + V + + = 伪
547  P V
548  INCLVL = 1
549  WW = \$(V * CALLENTE+)  WW
550  V = \$(V + ENTER+)
551  EN109  V BREAK(+) + AW + + = 伪
552  L = L
553  LW = LW AW + +
554  DEF#0  DEFINE (*PATHNASSERT(X)+*GOCALL+)
555  GOCALL  III = 0
556  PT#0  PATHCN = NP
557  ZZZ = 0
558  PT#1  PATH = PATH NP + +
559  PATH + + - + + =
560  NP + + =
561  PT#2  $PNAME + CODE+ NP) + IF+
562  $PNAME + CODE+ NP) + JMP+
563  $PNAME + CODE+ NP) + CASE+
564  $PNAME + CODE+ NP) + HALT+
565  NP = NP + 1
566  $PNAME + CODE+ NP) + EXIT+
567  IDENT($PNAME + AS+ NP))
568  PT#22  PATH = PATH NP + +
569  PT#23  IDENT(PASSIF)
570  PT#12  PASSIF =
571  CALL = ASSFRTNS(X)
572  PT#65  NP = PATHCN
573  ZZZ = 0
574  GT(111110)
575  PT#6  NP = NP + 1
576  $PNAME + CODE+ NP) + EXIT+
577  IDENT($PNAME + AS+ NP))
578  PT#11  PATH = PATH NP + +
579  PATH + + - + + =
580  NP + + =
581  $PNAME + CODE+ NP) + EXIT+
582  IDENT($PNAME + AS+ NP))
583  PT#30  L1(111111)
584  PT#3  W = $PNAME NP)
585  w = FAKE(++) + Tw + + =
586  w = FAKE(++) + Tw + + =
587  \# BREAK(++) * Fw + + =
588  III = III + 1
589  $IFLVH III) = Tw + +
590  $IFLVH III) = Tw + +
591  $IFLVH III) = Tw + +
592  $IFLVH III) = Tw + +
593  ZZZ = ZZZ + 1
594  PASSIF = PASSIF +
595  $IFLVH ZZZ) BREAK(++) + NP)
596  $IFLVH ZZZ) BREAK(++) + NP)
597  III = GT(11111111 II)
598  III = 0
599  PT#57  $IFLVH III) BREAK(++) + + =
600  $IFLVH III) BREAK(++) + + =
601  PT#54  IDENT($IFLVH III))
602  PT#54  IDENT($IFLVH III))
603  PT#50  LH(223111)
604  PT#60  LH(223111)
605  III = III + 1
606  $IFEGCASE+ NP) =
607  w = \$IPNAME NP)
608  w = \$IPNAME NP)
609  BREAK(++) + Tw + + + Tw + + +
610  DEF#3  CALL = COUNTR(X)
611  PT#3  CALL = COUNTR(X)
612  w\# = 0
613  w\# = 0
# APPENDIX B

## NUCLEUS PARSE TABLE

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td><code>C001EC560001DECO2D031PLEDEC003ARRAYDEC04TYPE0S1INTEGER0H00DHOLFAV067CHARACTER08B</code></td>
</tr>
<tr>
<td>002</td>
<td><code>C027DECSEC41EC</code></td>
</tr>
<tr>
<td>003</td>
<td><code>C030DECSEC9PLEDEC149</code></td>
</tr>
<tr>
<td>004</td>
<td><code>C040DECSEC41EC</code></td>
</tr>
<tr>
<td>005</td>
<td><code>C057DECSEC41EC</code></td>
</tr>
<tr>
<td>006</td>
<td><code>C064DECSEC41EC</code></td>
</tr>
<tr>
<td>007</td>
<td><code>C071DECSEC41EC</code></td>
</tr>
<tr>
<td>008</td>
<td><code>C088DECSEC41EC</code></td>
</tr>
<tr>
<td>009</td>
<td><code>C095DECSEC41EC</code></td>
</tr>
<tr>
<td>010</td>
<td><code>C102DECSEC41EC</code></td>
</tr>
<tr>
<td>011</td>
<td><code>C119DECSEC41EC</code></td>
</tr>
<tr>
<td>012</td>
<td><code>C126DECSEC41EC</code></td>
</tr>
<tr>
<td>013</td>
<td><code>C133DECSEC41EC</code></td>
</tr>
<tr>
<td>014</td>
<td><code>C140DECSEC41EC</code></td>
</tr>
<tr>
<td>015</td>
<td><code>C157DECSEC41EC</code></td>
</tr>
<tr>
<td>016</td>
<td><code>C164DECSEC41EC</code></td>
</tr>
<tr>
<td>017</td>
<td><code>C171DECSEC41EC</code></td>
</tr>
<tr>
<td>018</td>
<td><code>C188DECSEC41EC</code></td>
</tr>
<tr>
<td>019</td>
<td><code>C195DECSEC41EC</code></td>
</tr>
<tr>
<td>020</td>
<td><code>C202DECSEC41EC</code></td>
</tr>
<tr>
<td>021</td>
<td><code>C219DECSEC41EC</code></td>
</tr>
<tr>
<td>022</td>
<td><code>C226DECSEC41EC</code></td>
</tr>
<tr>
<td>023</td>
<td><code>C233DECSEC41EC</code></td>
</tr>
<tr>
<td>024</td>
<td><code>C240DECSEC41EC</code></td>
</tr>
<tr>
<td>025</td>
<td><code>C257DECSEC41EC</code></td>
</tr>
<tr>
<td>026</td>
<td><code>C264DECSEC41EC</code></td>
</tr>
<tr>
<td>027</td>
<td><code>C271DECSEC41EC</code></td>
</tr>
<tr>
<td>028</td>
<td><code>C288DECSEC41EC</code></td>
</tr>
<tr>
<td>029</td>
<td><code>C295DECSEC41EC</code></td>
</tr>
<tr>
<td>030</td>
<td><code>C302DECSEC41EC</code></td>
</tr>
<tr>
<td>031</td>
<td><code>C319DECSEC41EC</code></td>
</tr>
<tr>
<td>032</td>
<td><code>C326DECSEC41EC</code></td>
</tr>
<tr>
<td>033</td>
<td><code>C333DECSEC41EC</code></td>
</tr>
<tr>
<td>034</td>
<td><code>C340DECSEC41EC</code></td>
</tr>
<tr>
<td>035</td>
<td><code>C347DECSEC41EC</code></td>
</tr>
<tr>
<td>036</td>
<td><code>C354DECSEC41EC</code></td>
</tr>
<tr>
<td>037</td>
<td><code>C361DECSEC41EC</code></td>
</tr>
<tr>
<td>038</td>
<td><code>C368DECSEC41EC</code></td>
</tr>
<tr>
<td>039</td>
<td><code>C375DECSEC41EC</code></td>
</tr>
<tr>
<td>040</td>
<td><code>C382DECSEC41EC</code></td>
</tr>
<tr>
<td>041</td>
<td><code>C389DECSEC41EC</code></td>
</tr>
<tr>
<td>042</td>
<td><code>C396DECSEC41EC</code></td>
</tr>
<tr>
<td>043</td>
<td><code>C403DECSEC41EC</code></td>
</tr>
<tr>
<td>044</td>
<td><code>C410DECSEC41EC</code></td>
</tr>
<tr>
<td>045</td>
<td><code>C417DECSEC41EC</code></td>
</tr>
<tr>
<td>046</td>
<td><code>C424DECSEC41EC</code></td>
</tr>
<tr>
<td>047</td>
<td><code>C431DECSEC41EC</code></td>
</tr>
<tr>
<td>048</td>
<td><code>C438DECSEC41EC</code></td>
</tr>
<tr>
<td>049</td>
<td><code>C445DECSEC41EC</code></td>
</tr>
<tr>
<td>050</td>
<td><code>C452DECSEC41EC</code></td>
</tr>
<tr>
<td>051</td>
<td><code>C459DECSEC41EC</code></td>
</tr>
</tbody>
</table>

59
APPENDIX C

A SAMPLE PROGRAM OF NUCLEUS LANGUAGE

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

CHARACTER array A(10); C1(10); L(10)
INTEGER LAMH; COW; 1; MORECOW; MORELAMB; MORES
PROCEDURE READATA;
ASSERT LAMH=X(1)....X(I-1)
ASSERT C1=WY(1)....WY(I-1)
ASSERT IF ISKSI=1 THEN ~IHEOF(K)
END AT1

WHILE AT1
IF A(10) = \#IF THEN RETURN F11
CASE INTEGER(A(10)) OF
4: LAMH := LAMH + 10 * (INTEGER(A(11)) - 27) + (INTEGER(A(12)) - 27)
END CASE

ASSERT A(M)H=1;DHU=2;I1;IWTH)=;IWH;D.4=11
ASSERT LAMH=X(1)....X(I-1) IF IHEOF(IWHD) THEN I-I ELSE I11
ASSERT COW=2(I)....2(Y(I-1) IF IHEOF(IWHD) THEN I-I ELSE I11
ASSERT IF A(10)\#I THEN =I11;ST K SUCH THAT :IHEOF(K)
ASSERT IF A(11)\#I AND ISKSI THEN ~IHEOF(K)
EXIT
PROCEDURE MAIN1
I=11
COW := 01
ASSERT I = INT0D = IWHD 1
ASSERT LI S(I111)
ASSERT LAMH=X(1)....X(I-1) WHERE X(K)=THE INTEGER IN COLUMN 1-2 OF READ
RECORD K
IF COLUMN AO HAS \#0 AND ZERO IF NOT
ASSERT COW=WY(1)....WY(I-1) WHERE Y(K)=THE INTEGER IN COLUMN 3-4 OF HEAD RECORD K
IF COLUMN AO HAS \#B AND ZERO OTHERWISE
ASSERT WRITE RECORDS 1....I-1 ARE COPIES OF HEAD RECORDS 1....I-1
ASSERT IF ISKSI=1 THEN ~IHEOF(K)
WHILE I=1 TO 0 DO
ENTER READDATA;
IF A(10)=\#IF THEN GO TO S1 F11
I := I + 1
L=H11
ASSERT LAMH=WY(1)....WY(I-1) SUCH THAT :IHEOF(K)
ASSERT LAMH=X(1)....X(I-1) SUCH THAT :IHEOF(K)
S1: IF LAMH+COW THEN
MORECOW := COW + LAMH
IF COW TO W1
ELSE MORELAMB := LAMH - COW
FI
L(10) := \#B
L(11) := CHARACTEH(MORELAMB / 10 + 27)
MORELAMB := MORELAMB + 1
L(12) := CHARACTEH(MORELAMB + 27)
WRITE L1
GO TO F11
W1: C10 := \#F1
C(11) := CHARACTEH(MORECOW / 10 + 27)
MORECOW := MORECOW + 1
C(12) := CHARACTEH(MORECOW + 27)
WHITE C1
F1: END:
ASSERT IF LAMH+COW THEN WRITE RECORD 1+1 HAS COW=LAMH IN COLUMN 1-21
ASSERT IF COW=LAMH THEN WRITE RECORD 1+1 HAS LAMH-COW IN COLUMN 1-21
EXIT
START MAIN1

62
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 1

$ This program is designed to show the most features of the nucleus language $

CHARACTER A.I=0.1* C(10), L(I)!

INTEGER LAM, COW = 1, MORELAMH!

PROCEDURE READDATA:
(0.1) ASSEMB LAMBDA(X(1))*X(I-1)!
(0.2) ASSEMB COW = COW X(I)*X(I-1)!
(0.3) ASSEMB IF 15 + 1 THEN =MOREFOF K!
(0.4) READ A!
(1) WRITE A!
(2) IF A(0) = 0 THEN RETURN (4) FI!
(4) CASE INTEGER (A(I)-COW) OF
: (5) LAM := LAM + 10 * ([INTEGR(A(I))-27] + [INTEGR(A(I))-27])!
(6)12: (7) COW := COW + 10 * ([INTEGR(A(I))-27] + [INTEGR(A(I))-27])!
(7) ELSE:
(4.1) ASSEMB LAMBDA = LAMBDA + A(I) * X(I)!
(4.2) ASSEMB LAMBDA(X(I)*X(I-1) IF MOREFOF (K) THEN 1-1 ELSE 1)!
(4.3) ASSEMB COW = XY(I)*XY(I-1) IF MOREFOF (K) THEN 1-1 ELSE 1)!
(4.4) ASSEMB IF A(0) = 1 THEN =FIRST K SUCH THAT MOREFOF K!
(4.5) ASSEMB IF A(0) = 0 THEN =MOREFOF K)
(4) EXIT!

PROCEDURE MAIN:
(0) I := 1!
(1) I := 0!
(2) LAM := 0!
(3) I := A(I) = T (6) THEN (6) GO TO S1 (7) FI!
(7) := 1 + 1!
(8) FI!
(9) ASSEMB: LAMBDA(I) = FIRST K SUCH THAT MOREFOF K!
(10) ASSEMB: LAMBDA(X(I)*X(I-1)!
(11) ASSEMB: COW = XY(I)*XY(I-1)!
(12) IF LAM + COW <= 10 THEN
(13) LAM := LAM + COW - LAM!
(14) FI!
(15) NI := 10!
(16) MORELAMH := MORELAM + 10!
(17) L(I) := CHARACTER (MORELAMH / 10 + 27)!
(18) MORELAMH := MORELAM + 101!
(19) =WRITE L(I)!
(20) GO TO E1!
(21) COW := COW + COW!
(22) MORECOW := MORECOW + 101!
(23) LAM := LAM + LAM!
(24) =WRITE C!
(25) =NO

(26) ASSEMB IF LAM + COW THEN WHITE RECORD 1+1 HAS COW LAM!
(27) ASSEMB IF COW LAM THEN WHITE RECORD 1+1 HAS LAM COW!
(28) START MAIN
READDATA
0.1  LAMH=X(1)+...+X(I-1)
0.2  COW=Y(1)+...+X(I-1)
0.3  IF 15<51 THEN ~=REOF(K)

***************
0  "REOF(I:WTHD+1) = A.I(0)+T A [15<500 A A.I(5)]
  =:REOF(I:WTHD+1) = A.I(0)+F A [15<9MIN(HO+40) A A.I(5)]
  =:RDFL(I:RDHD+1.5)
  :=RDHD+1(I:RDHD+1)

1  A.I(0)+T :=REOF(I:WTHD+1) A [15<5MIN(RO+132) =:WTFL(I:WTHD+1.5)]
   :=WTFL(I:WTHD+1.5)++
   :=WTHD+1(I:WTHD+1)

2(PRV)  050500
   :=(A.I(0)+T)

4(PRV)  050500
   INTEGER(A.I(0)+2)

7(PRV)  050500
7(PRV)  050500

7(CW)  :=CW+10*(INTEGER(A.I(3)-27)*(INTEGER(A.I(4))-27)

**************
9.1  :=RDHD+1(I:RDHD+1.1) :=WTHD+1

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

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

9.4  IF (A.I(0)+T THEN I=FIRST K SUCH THAT :=REOF(K)

9.5  IF (A.I(0)+T AND 15<51 THEN ~=REOF(K)

----------

READDATA
0.1  LAMH=X(1)+...+X(I-1)
0.2  COW=Y(1)+...+X(I-1)
0.3  IF 15<51 THEN ~=REOF(K)

***************
0  "REOF(I:WTHD+1) = A.I(0)+T A [15<500 A A.I(5)]
  =:REOF(I:WTHD+1) = A.I(0)+F A [15<9MIN(HO+40) A A.I(5)]
  =:RDFL(I:RDHD+1.5)
  :=RDHD+1(I:RDHD+1)

1  A.I(0)+T :=REOF(I:WTHD+1) A [15<5MIN(RO+132) =:WTFL(I:WTHD+1.5)]
   :=WTFL(I:WTHD+1.5)++
   :=WTHD+1(I:WTHD+1)

2(PRV)  050500
   :=(A.I(0)+T)

4(PRV)  050500
   INTEGER(A.I(0)+2)

----------

9.1  :=WTHD+1(I:WTHD+1.1) :=WTHD+1

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

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

4.4  IF (A.I(0)+T THEN I=FIRST K SUCH THAT :=REOF(K)

4.5  IF (A.I(0)+T AND 15<51 THEN ~=REOF(K)
MAIN
3.1 I = :16UH0 = IDMD
3.2 1515101
3.3 LAMB=X[1]*...*X[1-1] WHERE X(K)=THE INTEGER IN COLUMN 1-2 OF READ RECORD K IF C
OLUMN HD HAS +D AND ZER0 IF NOT
3.4 COW=Y[1]*...*Y[1-1] WHERE Y(K)=THE INTEGER IN COLUMN 3-4 OF READ RECORD K IF CO
LUMN HD HAS +H AND ZER0 OTHERWISE
3.5 WHILE RECORDS 1,...,I-1 ARE COPIES OF READ RECORDS 1,...,I-1
3.6 IF I5KS[I-1] THEN ~:HEOF(K)

***************
151500

4(PRV)
05:LVL1:151501
4(PRV)
;WHTP.T.1512: IF $ = :LVL1 THEN MAINS~ ELSE :RTNPT($)
4(PRV)
(LAMB=X[1]*...*X[1-1])
4(PRV)
COW=Y[1]*...*Y[1-1])
4(PRV)
IF I5KS[1]=1 THEN ~:HEOF(K)
4(PRV)
IF (WHTH.1)=10 THEN :FSTK SUCH THAT :HEOF(K)
4(PRV)
IF (A[1][1])0=1 THEN ~:HEOF(K)
4(PRV)
:LVL2=1515101

5(PRV)
050:00
5
$A[1][1]=1
7
1.1=1

***************
3.1 (1.1) = (16UH0).1 = (16UH0.1)
3.2 1515101
3.3 (LAMB.1)=X[1]*...*X[1.1]-1) WHERE X(K)=THE INTEGER IN COLUMN 1-2 OF READ REC
ORD K IF COLUMN HD HAS +D AND ZER0 IF NOT
3.4 (COW.1)=Y[1]*...*Y[1.1]-1) WHERE Y(K)=THE INTEGER IN COLUMN 3-4 OF READ RECORD
 K IF COLUMN HD HAS +H AND ZER0 OTHERWISE
3.5 WHILE RECORDS 1,...,(1.1)-1 ARE COPIES OF READ RECORDS 1,...,(1.1)-1
3.6 IF I5KS[1.1]-1 THEN ~:HEOF(K)

MAIN
3.1 I = :16UH0 = IDMD
3.2 1515101
3.3 LAMB=X[1]*...*X[1] WHERE X(K)=THE INTEGER IN COLUMN 1-2 OF READ RECORD K IF C
OLUMN HD HAS +D AND ZER0 IF NOT
3.4 COW=Y[1]*...*Y[1] WHERE Y(K)=THE INTEGER IN COLUMN 3-4 OF READ RECORD K IF CO
LUMN HD HAS +H AND ZER0 OTHERWISE
3.5 WHILE RECORDS 1,...,I-1 ARE COPIES OF READ RECORDS 1,...,I-1
3.6 IF I5KS[I-1] THEN ~:HEOF(K)

***************
151500

9.1 I=MIN(10),FSTK SUCH THAT :HEOF(K)
9.2 LAMB=X[1]*...*X[1-1]
9.3 COW=Y[1]*...*Y[1-1]
BIBLIOGRAPHY


