THE NUCLEUS COMPILER

by

Eileen Victoria Josue

May 1973 TR-14

THE NUCLEUS COMPILER*

by

Eileen Victoria Josue

---

* This paper is supported in part by the National Science Foundation grant GJ-36424.

ABSTRACT

The Nucleus compiler is a two-pass compiler written in PASCAL for the programming language Nucleus. Nucleus is a language specifically designed for program verification purposes. A summary of the definition of Nucleus is presented first. A description of Pass I of the compiler (recognizer and reduced program generator) is presented next. Finally, a description of Pass II of the compiler (code generation and execution) is presented.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

This thesis describes an implementation in PASCAL (Wirth [12]) on the CDC 6600 of a compiler for the Nucleus programming language (Good and Ragland [6]), a language specifically designed to produce computer programs that run correctly at all times. Nucleus was designed with seven specific goals in mind.

1. Each Nucleus program must be provable by the inductive assertion method. Thus, the Nucleus language includes a way of stating inductive assertions within programs and contains only constructs on which the inductive assertion method may be used.

2. Nucleus should be structured to facilitate the construction of correct programs. Toward this end, Nucleus has a high level ALGOL-like syntax and includes statements that support the basic ideas of structured programming (Dijkstra [4], Wirth [11]).

3. In order to make Nucleus available on a wide range of machines, Nucleus programs must be easily compilable into almost any machine language.

4. A proof of the correctness of a semi-automatic inductive assertion verifier for Nucleus programs must be possible. Such a verifier is needed to handle the verification of non-trivial programs due to the volume of work

and detail required to verify such programs. Thus, the verifier itself must be proved correct.

5. A proof of correctness of a Nucleus compiler must be possible. Even if we use a correct verifier to obtain a proof of correctness of some program, that program will not run correctly if it is not compiled correctly. Unfortunately, this still does not guarantee that the program will always run correctly because the program has to rely on the hardware and operating system of the machine on which it is run. Nucleus, however, does not address itself to operating system and hardware correctness.

6. All syntactic and semantic aspects of Nucleus must be rigorously defined in order to attain the goals of proving a verifier and a compiler.

7. Lastly, non-trivial programs, such as compilers and verifiers for programming languages must be expressable in Nucleus. Then, by writing compilers and verifiers for other languages in Nucleus, a process of bootstrapping more and more correct software in more and more languages could be started.

The Nucleus compiler described here is one part of a four-phase project to construct a completely verified software system for Nucleus. The first phase of the project is a verification condition compiler (VCC) for Nucleus programs, Wang [9]. This VCC is written in SNOBOL IV and was debugged by conventional techniques. The second phase is another Nucleus VCC by Ragland [8]. This VCC is written in Nucleus and is being proved correct with the aid of the Wang VCC. In order to run Nucleus programs, and in particular the Ragland VCC, a Nucleus compiler is needed. Thus, the third phase of the project and the subject of this thesis is a Nucleus compiler written in PASCAL. This compiler is not proved, but has been debugged by conventional debugging techniques. The fourth phase of the project, which has not yet been initiated, is another Nucleus compiler that is written in Nucleus and proved correct. The completion of this project leaves us with both a correct VCC and a correct compiler. Thus, we can prove and compile Nucleus programs and be assured that they actually will run correctly barring operating system or hardware failure. With this VCC and compiler, we can begin a process of bootstrapping more and more proved processors for more and more languages. A more complete discussion of developing correct software systems in this manner can be found in Good [5].

Syntactically, Nucleus is formally defined by transition networks similar to those of Woods [13] and semantically by means of axioms as suggested by Burstall [2]. The transition networks define the recognizer for the language. The semantics of Nucleus consists of a mapping from Nucleus programs into sentences in the predicate calculus and a set of axioms. The sentences in the predicate calculus are called the reduced program. The axioms act as an interpreter on the reduced program. In other words, it is the reduced program that gets interpreted rather than the Nucleus program itself. A summary of the Nucleus language is given in Chapter II.

The Nucleus compiler is implemented as a two-pass compiler. Pass I accepts a Nucleus program and produces its reduced program, and Pass II generates code from the reduced program. A two-pass compiler was written for two main reasons. The first reason was to be able to check the definition of Nucleus by inspecting the reduced program produced by Pass I. The second reason was that the reduced program produced by Pass I can be used as input to a verification condition compiler just as well as for a machine code compiler. Thus, Pass I provides a basis for future verification systems as well as for the compiler described here.

Pass I, which is described in Chapter III, checks the syntax of a Nucleus program, i.e., recognizes an input string to be a Nucleus program, and transforms the program into its reduced program. In other words, the transition networks that define the syntax of Nucleus are implemented in Pass I. Within these transition networks is also the mechanism that defines the semantic mapping. Thus, this mechanism, too, is implemented, and the output of Pass I is the reduced program specified by this semantic mapping.

Pass II uses the output of Pass I to generate absolute object code for the Nucleus program. Since all syntactic errors have been detected by the end of Pass I, object code can easily be generated from the reduced program. This is because object code is only generated if a program has no syntax errors, thus eliminating any translation or load errors. The object code is generated onto a file and passed to the PASCAL operating system to be loaded into memory and executed. Thus, Pass II generates object code compatible with the PASCAL operating system.

CHAPTER II

NUCLEUS LANGUAGE SUMMARY

Method of Definition

Before explaining the implementation of Nucleus, an understanding of the language is necessary. Formally, Nucleus is defined syntactically by means of transition networks which are a modification of the "augmented transition network grammars" described by Woods[13] for dealing with natural languages and semantically by means of axioms as suggested by Burstall[2]. A complete description of the method of definition appears in Good and Ragland[6]. Since the networks are based on finite state transition diagrams, the language defined by the networks is the set of strings accepted by the network. Thus, the syntax of the language is defined by specifying its recognizer in terms of transition networks.

A transition network is actually a graph of labelled states and arcs. Of the states, one is the initial state, and a finite number of states are designated as recognition states. Arcs may be labelled in one of three ways: by an input string character, a state name, or NIL. Also, each arc has a test, a set of actions, and a SCAN flag. An arc is traversable in one of three ways:

1. Consider all character labelled arcs first. If the input pointer points to a character matching the character on the arc and the test is satisfied, the arc is traversable.

2. If no traversable character labelled arc exists, consider all NIL labelled arcs. If the test is satisfied, the arc is traversable.

3. Otherwise, consider the arc labelled with a state name (only one such arc may exist leaving any state). Stack this arc on the arc stack and proceed to the state which labels the arc. If a recognition state is then encountered, the arc at the top of the stack is reconsidered. If its test is satisfied, pop the stack and traverse the arc.

The Nucleus language recognizer consists of two networks: a scanning network and a parsing network. The following is an example of a portion of the parsing network. This is the initial section of the parser and defines the recognition of a program. "FIND character" specifies a character or NIL labelled arc and means to look for the specified character on the input string. "PHRASE statename" specifies an arc that is labelled with a state name. TEST defines the tests, if any, that must be satisfied, and DO defines the set of actions to be performed. The "SCAN-NOSCAN" "statenum" flag defines whether or not to advance the input string pointer and designates which state to proceed to next. A more detailed discussion of these transition networks can be found in Good and Ragland [6].

```
PROGRAM:
   FIND NIL
   DO   DEFINED.SIMPLE.SET :=[ ]
        DEFINED.ARRAY.SET :=[ ]
        TYPE.FUNCTION :=[ ]
        DEFINED.PROCEDURE.SET := [ ]
        REFERENCED.PROCEDURE.SET :=[ ]
        DEFINED.IDENTIFIER.SET :=[ ]
   NOSCAN 1

1: PHRASE DECLARATION.SEQUENCE
   NOSCAN 2

2: PHRASE PROCEDURE.SEQUENCE
   TEST REFERENCED.PROCEDURE.SET SUBSETOF
        DEFINED.PROCEDURE.SET
   NOSCAN 3

3: FIND START
   SCAN 4

4: FIND IDENTIFIER
   TEST TOKEN.STRING IN DEFINED.PROCEDURE.SET
   DO   SENTENCE(INITIALPROCEDURE=TOKEN.STRING)
   NOSCAN 5

5[RECOGNITION] :
```

The parser starts at the initial state PROGRAM, performs the actions, and continues to state 1. At state 1, the arc between state 1 and state 2 is stacked, and the network proceeds to the state DECLARATION.SEQUENCE (which is in a part of the network that is not shown). Once a declaration sequence has been recognized, the arc from state 1 to state 2 is popped and the network continues at state 2. This procedure is followed through state 5, the final recognition state.

The formal definition of the semantics of Nucleus consists of a mapping from Nucleus programs into sentences in the predicate calculus and a set of axioms. The predicate calculus sentences are called the reduced program. This semantic mapping is defined in the parser by the action SENTENCE(X). For example, in the previous example, the action at state 4 is a SENTENCE action. This causes the sentence "INITIALPROCEDURE=procedure name" to be entered into the reduced program.

The reduced program can be viewed as defining a program for a virtual machine which itself is defined by the Nucleus axioms. The reduced program defines two distinct parts of the virtual machine - the data memory and the instruction memory. The data memory is defined by the sentences generated during the recognition of the DECLARATION.SEQUENCE part of a Nucleus program. The instruction memory and the virtual instructions are defined by the sentences generated during the recognition of the PROCEDURE.SEQUENCE of a program. The set of virtual instructions comprise the virtual program to be interpreted by the axioms.

Within each procedure, virtual instructions are associated sequentially with virtual addresses starting at address 0. Thus, many virtual instructions may be associated with the same numeric address. For example, the virtual address of the first instruction of every procedure is 0.

Thus, many "0" addresses may occur in the total virtual program. To distinguish one "0" address from another, the procedure name is made a part of the virtual address. Thus, every virtual address has the form "procedurename:point," where "point" is the numeric part of the address. Fig. 1 shows a Nucleus program and its corresponding reduced and virtual programs. Another example of a Nucleus program and its reduced program can be found at the end of this chapter.

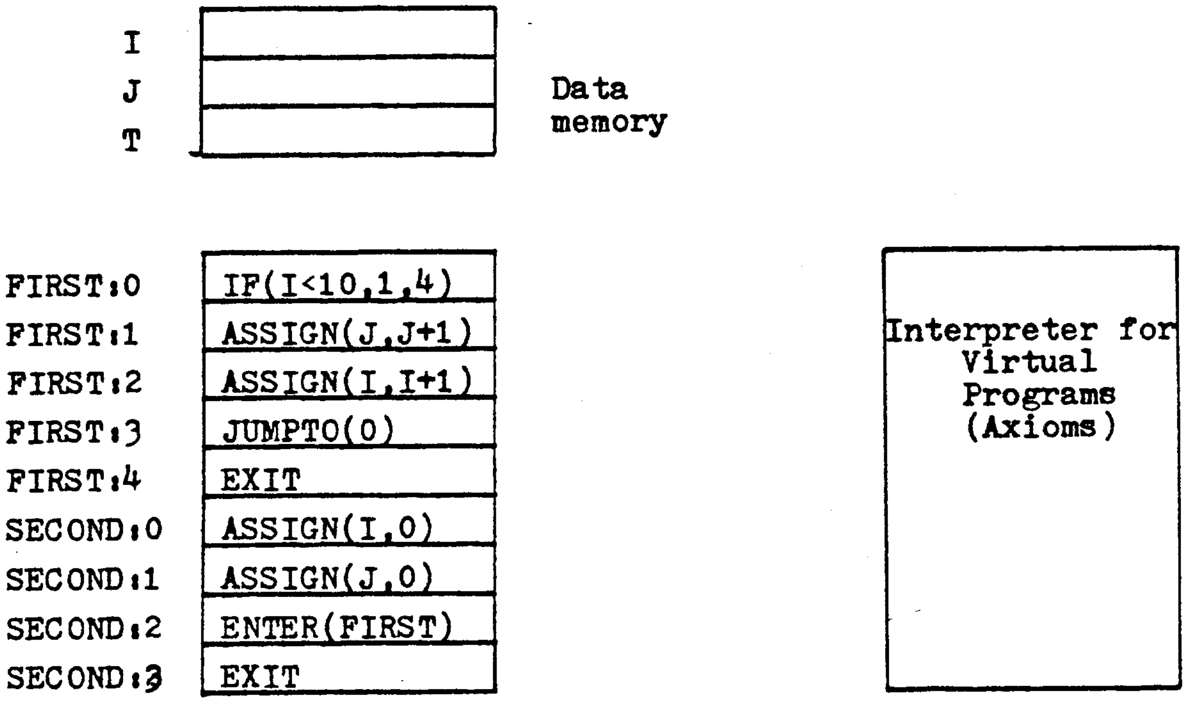
```
INTEGER I,J;
BOOLEAN T;
PROCEDURE FIRST;
     WHILE I<10 DO
          J:=J+1;
          I:=I+1;
     ELIHW;
EXIT;
PROCEDURE SECOND;
     I:=0;
     J:=0;
     ENTER FIRST;
EXIT;
START SECOND
```

a) Nucleus program

```
SIMPLE(I)
SIMPLE(J)
SIMPLE(T)
IF(FIRST:0,I<10,1,4)
ASSIGN(FIRST:1,J,J+1)
ASSIGN(FIRST:2,I,I+1)
JUMPTO(FIRST:3,0)
EXIT(FIRST:4)

ASSIGN(SECOND:0,I,0)
ASSIGN(SECOND:1,J,0)
ENTER(SECOND:2,FIRST)
EXIT(SECOND:3)
INITIALPROCEDURE=SECOND
```

b) Reduced program



FIRST:0	IF(I<10,1,4)
FIRST:1	ASSIGN(J,J+1)
FIRST:2	ASSIGN(I,I+1)
FIRST:3	JUMPTO(0)
FIRST:4	EXIT
SECOND:0	ASSIGN(I,0)
SECOND:1	ASSIGN(J,0)
SECOND:2	ENTER(FIRST)
SECOND:3	EXIT

c) Virtual program

Fig. 1. Nucleus reduced and virtual programs

Syntax and Semantics

The basic character set of Nucleus consists of 64 elements. These characters are grouped into character strings called "tokens." The scanning network reads the basic characters on the input string and groups them into tokens. These tokens are then used as the input string for the parsing network and are the basic symbols used in writing Nucleus programs. The character set consists of: {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 ([ ]) ↑ * / ↓ + - < ≤ ≥ > = ≠ ¬ ∧ ∨ → ≡ , ; : . $ #}. Blank, the letters, the digits, :, ↑, and $ are not tokens, but each member of the remainder of the set is considered as a separate token. However, : is a token provided it is not immediately followed by = ; := is a separate token.

Nucleus also has a set of reserved words, each of which is considered a token. The reserved words are:

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.

Other tokens are ASSERTION tokens, IDENTIFIER tokens, NUMBER tokens, and CHARACTERCONSTANT tokens. ASSERTION tokens consist of "ASSERT text;" where the text consists of any sequence of characters except ;. The text may contain a quoted semicolon, a semicolon immediately preceded by ↑. IDENTIFIER tokens consist of a letter followed by any number of letters or digits (no embedded blanks are allowed). The reserved words and ASSERT do not constitute IDENTIFIER tokens, however. NUMBER tokens consist of any string of digits. CHARACTERCONSTANT tokens consist of ↑c, where c is any element of the basic character set.

A Nucleus program is of the form:

declarations procedures START identifier

The parsing network recognizes the declarations and procedures that comprise a Nucleus program. While recognizing these parts, the Nucleus program is mapped into its reduced program.

The declarations define simple variables and array variables. Each variable used in the program must be declared uniquely in this section. In other words, no local variables are allowed in Nucleus. A simple declaration has the form:

type identifier, ... , identifier;

where the type is either INTEGER, BOOLEAN, or CHARACTER. When a simple variable is declared, the sentence "SIMPLE(identifier)" is produced in the reduced program.

An array declaration is of the form:

type ARRAY identifier[number] , ... , identifier[number] ;

where "number" defines the upper bound of the array. The lower bound is always assumed to be zero. Thus, only linear arrays are allowed. As each array is declared, the reduced program sentence "ARRAY(identifier,number)" is generated.

A Nucleus procedure has the form:

PROCEDURE identifier; body EXIT;

The identifier is the procedure name and must not have been declared previously as a procedure or as a variable. Procedures are recursive but allow no parameters.

The body of a procedure consists of the statements and assertions that define the procedure. A body itself consists of any number of statements and/or assertions. A body is delimited by any of the tokens EXIT, ESAC, FI, ELIHW, or NUMBER. The delimiting tokens are not included in the body.

Eleven statement types exist in Nucleus. Each statement may have one or more identifier labels and must be terminated by a semicolon. The statement types are: assignment, GO TO, RETURN, null, IF, CASE, WHILE, ENTER, HALT, READ, and WRITE. As each statement type is recognized by the parser, one or more sentences of the reduced program are generated. The statement types and their corresponding reduced program sentences can be found in Table 1.

The first argument of every sentence is the virtual address of the instruction in the virtual program. Since the reduced program sentences are generated immediately following the recognition of a statement, the sentences may not be generated in the order in which they are associated with

TABLE 1. Statements and corresponding sentences

Statement	Form	Sentence
Assignment	leftside;=expression	ASSIGN(virtual address,leftside, expression)
GO TO	GO TO identifier	JUMPTO(virtual address, virtual address of identifier)
RETURN	RETURN	JUMPTO(virtual address, virtual address of EXIT statement)
Null	NOP	JUMPTO(virtual address, virtual address + 1)
IF	a) IF expression THEN body FI	a) IF(virtual address, expression, virtual address + 1, virtual address of statement following FI)
	b) IF expression THEN body ELSE body FI	b) JUMPTO(virtual address of end of body, virtual address of statement following FI) IF(virtual address, expression, virtual address + 1, virtual address of statement following ELSE)
ENTER	ENTER identifier	ENTER(virtual address, identifier)
HALT	HALT	HALT(virtual address)
READ	READ identifier	READ(virtual address, identifier)
WRITE	WRITE identifier	WRITE(virtual address, identifier)

TABLE 1. (continued)

Statement	Form	Sentence
CASE	a) CASE expression OF alternative sequence ESAC	a) CASELABELSET(virtual address)= caselabelset CASE(virtual address, expression, virtual address of statement following ESAC) CASEJOINPOINT(virtual address)= virtual address of statement following ESAC
	b) CASE expression OF alternative sequence ELSE body ESAC	b) CASELABELSET(virtual address)= caselabelset CASE(virtual address, expression, virtual address of statement following ELSE) CASEJOINPOINT(virtual address)= virtual address of statement following ESAC
WHILE	WHILE expression DO body ELIHW	JUMPTO(virtual address of end of body, virtual address of WHILE statement) IF(virtual address of WHILE statement, expression, virtual address of WHILE + 1, virtual address of statement following ELIHW)

virtual addresses. For example, IF, WHILE, and CASE statements do not generate their sentences in the correct order. Fig. 2 is an example of an IF statement with its sentences in the order they are generated and in the order they appear in the virtual program.

```
PROCEDURE IFEX;
IF I<J THEN I:=I+1; FI;
EXIT;
```

a) Procedure

```
ASSIGN(IFEX:1,I,I+1)
IF(IFEX:0,I<J,1,2)
```

b) Actual order sentences are generated

```
IF(IFEX:0,I<J,1,2)
ASSIGN(IFEX:1,I,I+1)
```

c) Virtual program order

Fig. 2. IF statement and sentences

A detailed discussion of all the statements will not be given here. Only the CASE statement and the READ and WRITE statements will be discussed here. The other statement types are self-explanatory. A more detailed discussion of the other statements can be found in Good and Ragland [6].

The CASE statement is used for multi-way branches and has the forms stated in Table 1. The CASE expression must be of type INTEGER. The alternative sequence has the form:

numericlabels body numericlabels body ...

where numericlabels is a numeric label sequence

number: ... :number:

Each numeric label must be unique within an alternative sequence. In CASE statement a) of Table 1, the expression is evaluated first and control goes to the body labelled with the value of the expression. If no such label exists, control flows to the end of the CASE statement, i.e., execution continues at the point immediately following the ESAC token. In CASE statement b) of Table 1, the same procedure is followed except that if the expression value does not match a label, execution continues at the point immediately following the ELSE token. In both cases when control reaches the end of a body in the alternative sequence, control goes next to the statement following the ESAC token.

The form of the READ/WRITE statements can be found in Table 1. The identifier must be an array of type CHARACTER. The READ/WRITE statements access the standard input and output files, respectively. The standard files are actually a numbered sequence of records (1, 2, ...), each record being either an end-of-file (eof) record or not an eof record. Non-eof records consist of n characters of the basic character set, n being constant for all records. However, the input and output file record sizes need not be the same.

All arrays have a lower bound of zero. For the character arrays referenced in READ/WRITE statements, identifier [0] is used as an eof flag. If the character T is in identifier[0], then the record is an eof record. Otherwise, the character F is in identifier[0]. For READ statements that access non-eof records, the character i of the record is placed into identifier[i] for all i such that $1 \le i \le$ min(upper bound of identifier, record size). Any remaining array elements are left unchanged. For WRITE statements that access non-eof records, characters 1, ... , m of the output file record become the characters in identifier[1], identifier[2], ... , identifier[m], where m = min(upper bound of identifier, record size). The record is blank-filled if m is less than the record size. The record sizes for the READ/WRITE statements are implementation parameters. The implementation parameters will be discussed in Chapter IV.

From Table 1, it can be seen that many statement types are built from expressions. Expressions are built from primaries in the usual way. A primary is defined to be a constant, a simple variable, or an array reference. The operators available for expressions are given in Table 2.

Nucleus Program Example

This section gives an example of a Nucleus program and its reduced program. The numbers in parentheses to the left of the Nucleus program define the local points of each procedure and are not part of the program. These points correspond to the virtual instruction addresses of each procedure.

TABLE 2. Operators

Operator	Priority	Operand Type
unary +,-	1	INTEGER
*,/,↓ (modulo)	2	INTEGER
binary +,-	3	INTEGER
<,≤,=,≠,≥,>	4	Any type, provided operands are of the same type
¬	5	BOOLEAN
∧	6	BOOLEAN
∨	7	BOOLEAN

```
     INTEGER FIRST,LAST,MIDDLE,X,N;
     BOOLEAN FOUND;
     INTEGER ARRAY A[100] ;

     PROCEDURE BINARYSEARCH;
     $SEARCH ARRAY A FOR X$

 (0) FOUND:=FALSE;
 (1) FIRST:=0;
 (2) LAST:=N;
 (3) WHILE FIRST≤LAST ∧ ¬FOUND DO
 (4)     MIDDLE:=(FIRST+LAST)/2;
 (5)     IF X<A[MIDDLE] THEN
 (6)          LAST:=MIDDLE-1;
 (8)     ELSE IF X=A[MIDDLE]  THEN
 (9)          FOUND:=TRUE;
(11)          ELSE FIRST:=MIDDLE+1;
              FI;
         FI;
(12) ELIHW;
(13) EXIT;
     START BINARYSEARCH
```

a) Nucleus program

Fig. 3. Program example

```
     Virtual              Virtual
     address              instructions

                       SIMPLE(FIRST)
                       SIMPLE(LAST)
Data memory            SIMPLE(MIDDLE)
(no addresses)         SIMPLE(X)
                       SIMPLE(N)
                       SIMPLE(FOUND)
                       ARRAY(A,100)
BINARYSEARCH:0         ASSIGN(BINARYSEARCH:0,FOUND,FALSE)
BINARYSEARCH:1         ASSIGN(BINARYSEARCH:1,FIRST,0)
BINARYSEARCH:2         ASSIGN(BINARYSEARCH:2,LAST,N)
BINARYSEARCH:3         IF(BINARYSEARCH:3,FIRST≤LAST ∧ ¬FOUND,4,13)
BINARYSEARCH:4         ASSIGN(BINARYSEARCH:4,MIDDLE,(FIRST+LAST)/2)
BINARYSEARCH:5         IF(BINARYSEARCH:5,X<A[MIDDLE],6,8)
BINARYSEARCH:6         ASSIGN(BINARYSEARCH:6,LAST,MIDDLE-1)
BINARYSEARCH:7         JUMPTO(BINARYSEARCH:7,12)
BINARYSEARCH:8         IF(BINARYSEARCH:8,X=A[MIDDLE],9,11)
BINARYSEARCH:9         ASSIGN(BINARYSEARCH:9,FOUND,TRUE)
BINARYSEARCH:10        JUMPTO(BINARYSEARCH:10,12)
BINARYSEARCH:11        ASSIGN(BINARYSEARCH:11,FIRST,MIDDLE+1)
BINARYSEARCH:12        JUMPTO(BINARYSEARCH:12,3)
BINARYSEARCH:13        EXIT(BINARYSEARCH:13)
                       EXITPOINT(BINARYSEARCH)=13
                       INITIALPROCEDURE=BINARYSEARCH
```

b) Reduced program

Fig. 3. (continued)

CHAPTER III

PASS I: RECOGNIZER AND REDUCED PROGRAM GENERATOR

The Nucleus compiler is a two-pass compiler written in PASCAL. This chapter deals with the first pass, which determines if a character string is a Nucleus program and maps it into its reduced program. The second pass, which generates object code from the reduced program, will be discussed in the following chapter.

Recognizer

Pass I recognizes Nucleus programs by implementing the transition networks described in Chapter II. Again consider the first segment of the parsing network.

```
PROGRAM:
   FIND NIL
   DO   DEFINED.SIMPLE.SET:=[]
        DEFINED.ARRAY.SET:=[]
        TYPE.FUNCTION:=[]
        DEFINED.PROCEDURE.SET:=[]
        REFERENCED.PROCEDURE.SET:=[]
        DEFINED.IDENTIFIER.SET:=[]
   NOSCAN 1

1: PHRASE DECLARATION.SEQUENCE
   NOSCAN 2

2: PHRASE PROCEDURE.SEQUENCE
   TEST REFERENCED.PROCEDURE.SET SUBSETOF
        DEFINED.PROCEDURE.SET
   NOSCAN 3

3: FIND START
   SCAN 4
```

```
4: FIND IDENTIFIER
   TEST TOKEN.STRING IN DEFINED.PROCEDURE.SET
   DO   SENTENCE(INITIALPROCEDURE = TOKEN.STRING)
   NOSCAN 5

5[RECOGNITION]:
```

At state 1, according to the formal definition, the arc from state 1 to state 2 would be stacked, and the network would proceed to a state labelled DECLARATION.SEQUENCE. Since the PASCAL system uses a stack for procedure return points, it is unnecessary to program explicitly the stacking operation. Rather than performing a stack operation as such, each state name that appears in a "PHRASE statename" statement is implemented as a separate PASCAL procedure. Then the procedure corresponding to the state name is called where "PHRASE statename" occurs. Upon exit of the procedure, which is equivalent to encountering a recognition state, control automatically returns to the calling point. In each case, this is equivalent to having stacked the arc, proceeding to the state, encountering a recognition state, and popping the arc stack.

All sets, i.e., DEFINED.SIMPLE.SET, REFERENCED.PROCEDURE.SET, etc., are implemented as linear arrays. Because of this, appending an element to a set cannot continue indefinitely as implied by the formal definition. The size restrictions of the arrays will be discussed further on in this chapter.

If a state has no traversable arc and is not a recognition state, the input string is immediately rejected as a Nucleus program. However, when the parser detects an error, it prints an error message and continues parsing as much of the program as possible. In many cases, the parser acts as though no error occurred and continues from where it found the error. This is the case for such errors as: missing semicolon, undefined identifier, non-matching expression types, and missing EXIT statements. However, for some errors, this is not possible. In these cases, the error routine scans to the end of the statement being parsed, i.e., scans to the next semicolon, and continues. This is the case for such errors as: missing identifier, missing :=, and a missing TO in a GO TO statement. Table 3 defines each error detected by the parser and the type of recovery made. In Table 3, SCAN means to scan to the next semicolon. NOSCAN means to return to the point where the error was detected and continue through the program.

TABLE 3. Error messages and recovery

Error Number	Error Message	Recovery
1	Identifier expected	SCAN
2	; expected	NOSCAN
3	[expected	NOSCAN
4	Number expected	SCAN
5	] expected	NOSCAN
6	Statement expected	SCAN
7	START expected	NOSCAN
8	Label previously defined	NOSCAN
9	Undefined identifier	NOSCAN
10	Undefined array	NOSCAN
11	Previously defined identifier	NOSCAN
12	Procedure expected	NOSCAN
13	EXIT expected	NOSCAN
14	Undefined label	NOSCAN
15	OF expected	SCAN
16	Type BOOLEAN expected	NOSCAN
17	Types do not match	NOSCAN
18	Unacceptable relation type	NOSCAN
19	Type INTEGER expected	NOSCAN
20	(expected	NOSCAN
21	) expected	NOSCAN
22	:= expected	SCAN

TABLE 3. (continued)

Error Number	Error Message	Recovery
23	TO expected	SCAN
24	ELIHW expected	NOSCAN
25	Character array expected	NOSCAN
26	FI or ELSE expected	NOSCAN
27	THEN expected	SCAN
28	DO expected	SCAN
29	Error in declaration part	SCAN
30	: expected	NOSCAN
31	Unacceptable primary	SCAN
32	Undefined procedure	NOSCAN
33	ELSE or ESAC expected	NOSCAN
34	Array too large	NOSCAN
35	Procedure too long	SCAN to next procedure

Reduced Program Generator

As differences in the implementation of the transition networks and their formal definition exist, so do differences exist in the actual reduced program and the formal reduced program. The first major difference has to do with the declaration of simple and array variables. Instead of generating SIMPLE(idname) or ARRAY(idname, bound) for the program variables, the parser sets up a symbol table and uses this table for object code generation. Having a symbol table makes it easier to determine how much space to allocate for variables during code generation.

The second major difference has to do with the representation of the sentences in memory. Rather than storing the actual character strings, the sentences are stored as tree structures with the sentence type (ASSIGN, IF, etc.) being the root of the tree. The tree structures are discussed in more detail further on in this chapter.

Another major difference between the formal definition of the reduced program and the actual reduced program has to do with which sentence trees are actually generated. The only sentence trees produced are those that will generate object code. For example, X:=1 generates object code, so its sentence tree would be constructed. However, some sentences do not generate code, such as one of the sentences connected with a CASE statement, i.e.,

CASEJOINPOINT(procedurename:casepoint) = point.
This statement defines where the end of a CASE statement is and neet not generate any object code.

As stated in Chapter II, the virtual addresses associated with the virtual instructions have the form "procedurename:point." However, in the reduced program produced by Pass I, the instructions are addressed sequentially starting at virtual address 0. Thus, the points associated with each procedure are all relative to the starting address of the procedure.

The reduced program actually produced by Pass I consists of four parts: the symbol table, the constant table, the procedure table, and the sentence tree table. The symbol table, constant table, and procedure table define the data memory for the virtual machine. The sentence tree table defines the virtual program for the machine. The form of the actual reduced program is found in Fig. 4.

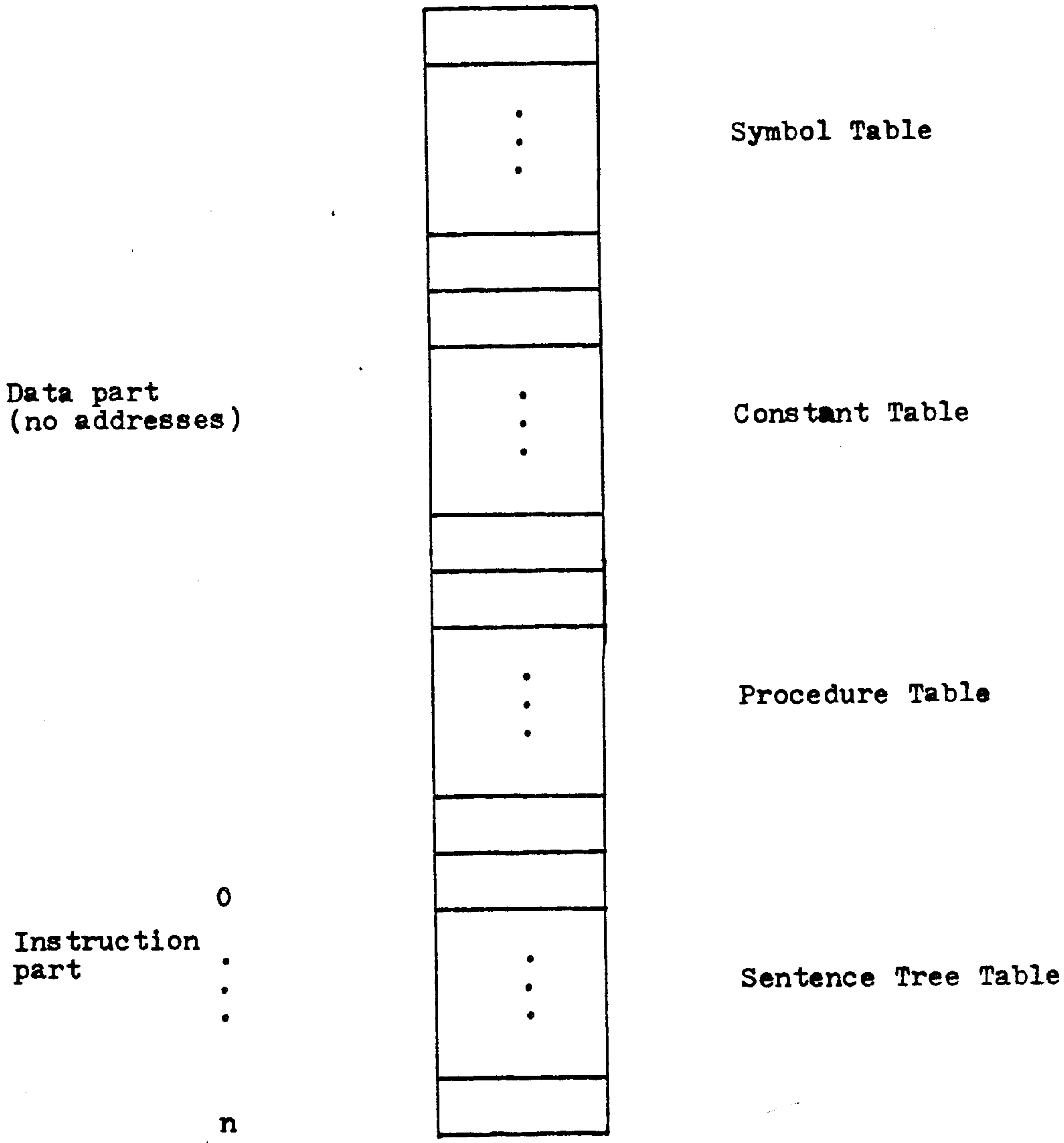
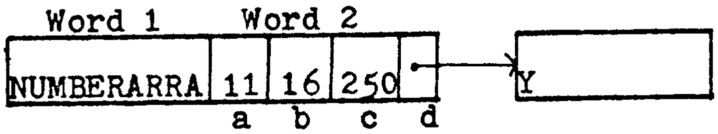


Fig. 4. Reduced program produced by Pass I

The symbol table contains all the simple identifier names and array variable names. Each element of the symbol table contains the first ten characters of the identifier name, the length of the identifier, the type of the identifier, the array length (-1 for simple variables), and a pointer to the rest of the identifier if the identifier length is greater than ten. For example, INTEGER ARRAY NUMBERARRAY[250] would be entered into the symbol table as:

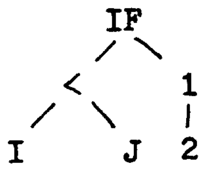


where a = the identifier length, b = the type, c = the array bound, and d = the pointer to the rest of the identifier. Since every identifier contains its type, the set TYPE.FUNCTION of the formal definition need not be implemented separately.

The constant table contains all the constants used in the Nucleus program and is self-explanatory. The procedure table is somewhat similar to the symbol table. However, part c of the procedure identifier contains the starting address of the procedure in the virtual program.

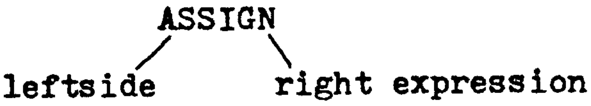
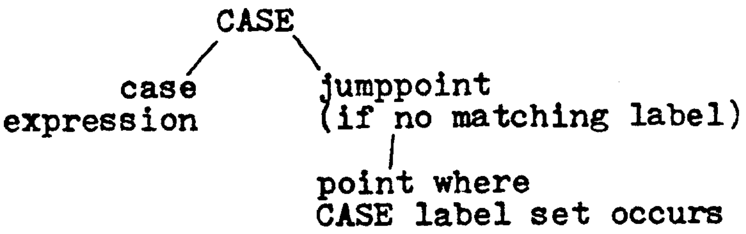
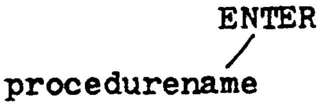
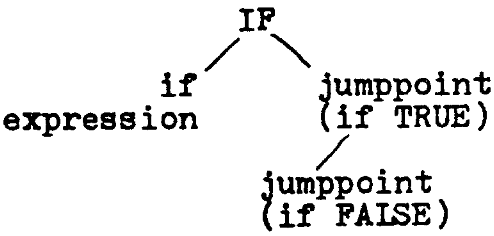
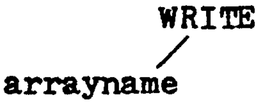
The sentence tree table contains the tree representations of the code-generating reduced program sentences. Nine types of reduced program sentences generate code. These are: ASSIGN, CASE, ENTER, EXIT, HALT, IF, JUMPTO, READ, and WRITE. Therefore, only the sentences starting with these nine words are contained in the sentence tree table.

However, not all the information called for is stored in the sentence trees. As was noted in the previous chapter, the argument "procedurename:point" defines the virtual address for the virtual instruction. At the time the sentence tree is built, the tree is stored in the sentence tree table at the location corresponding to the point the statement occurs in the procedure. In Fig. 2, the sentence IF(IFEX:0,I<J,1,2) is produced when the whole IF statement has been parsed. However when the tree is built, it is automatically stored in the correct position in the table, i.e., it is stored at address 0. Thus, the argument "procedurename:point" need not be kept in the tree itself. The nine sentence types and their corresponding sentences can be found in Table 4.



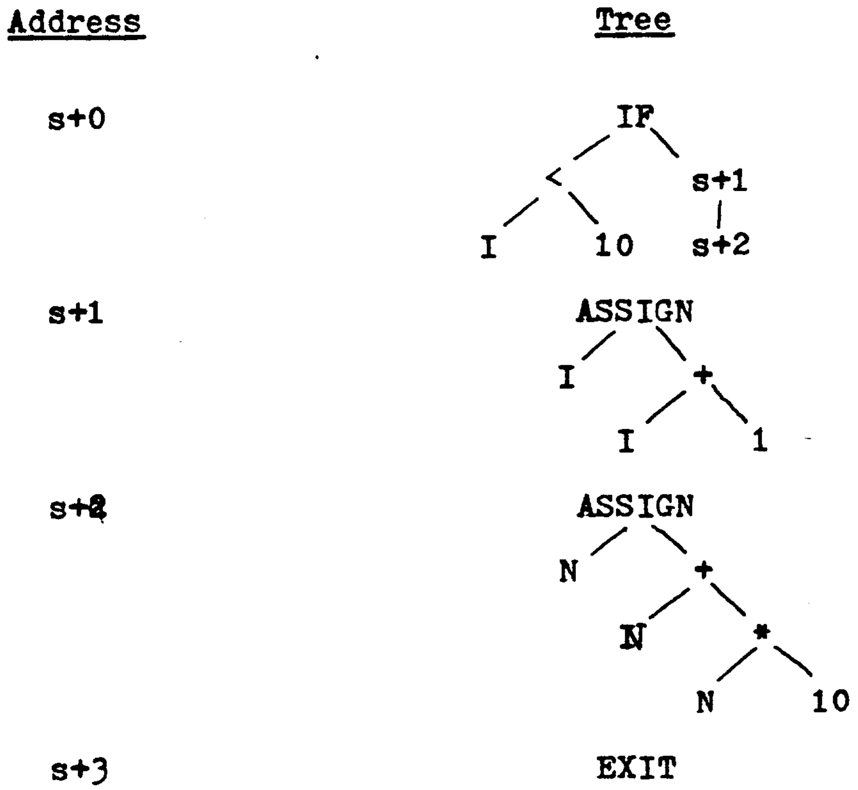
The points mentioned in Table 4 do not correspond directly to the point numbers in the formal definition of the reduced program. However, all of the points are relative to the procedure starting address in the virtual program. Thus, in the formal reduced program sentences, the argument "POINT" actually gets stored as (procedure starting address) + POINT. Fig. 5 gives an example of this.

TABLE 4. Sentence trees

Sentence Type	Tree
ASSIGN	
CASE	
ENTER	
EXIT	EXIT
HALT	HALT
IF	
JUMPTO	
READ	
WRITE	

```
PROCEDURE POINTEXAMPLE;
IF I<10 THEN
   I:=I+1;
FI;
N:=N+(N*10);
EXIT;
```

a) Program



b) Virtual program segment
(procedure starting address = s)

Fig. 5. Point example

As can be seen in Fig. 5 b), the expressions are also represented as trees. These expression trees are built as expressions are being parsed. When the sentence tree is generated, the expression tree is stored in the sentence tree at the appropriate node. Note that all trees are strict binary trees, making traversal of the trees simple and fast.

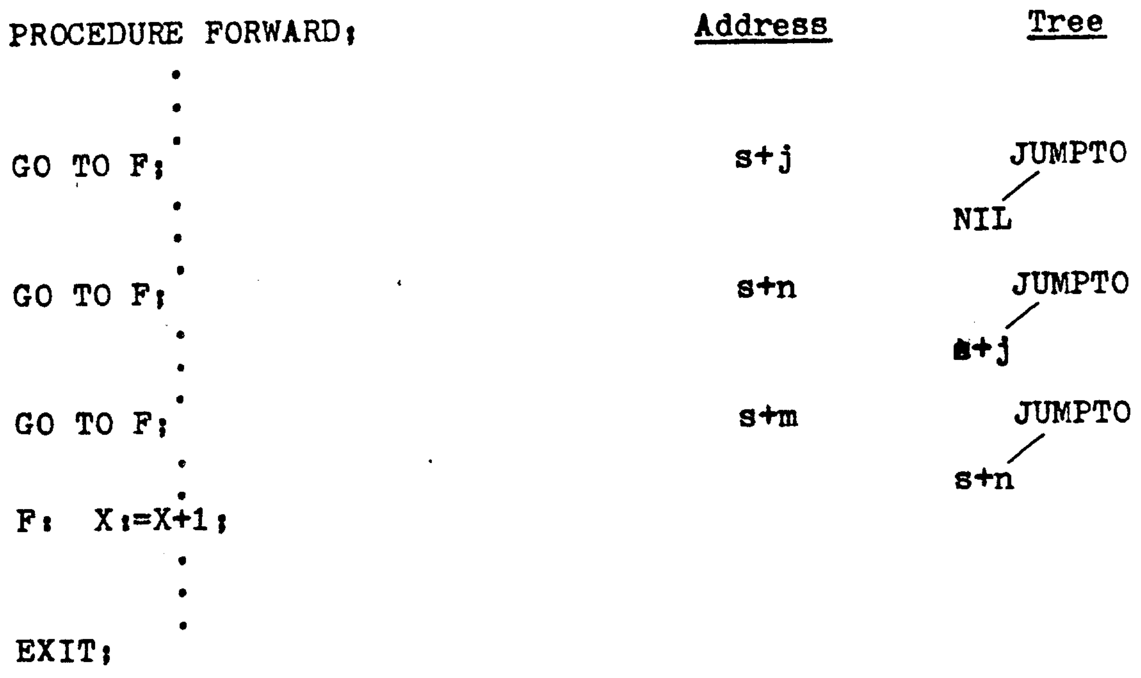
Label Handling

The treatment of labels is somewhat bothersome because forward referencing is allowed and the labels are not declared. However, many of the usual forward referencing problems do not exist in Nucleus mainly due to the fact that GO TO statements cannot jump across procedure boundaries.

Labels are handled in the following manner. As each procedure is parsed, two local tables are built up, a declared label table and a referenced label table. An entry is made into the declared label table whenever a label is encountered (LABEL: statement). The entry contains the label name and its virtual address. An entry is made into the referenced label table only if the label name in the GO TO statement is not in the declared label table and is not already in the referenced label table. In other words, the only time a label is entered into the referenced label table is when a label is being forward referenced for the first time. After a procedure declaration has been parsed, a check is made to see if every referenced label has been declared.

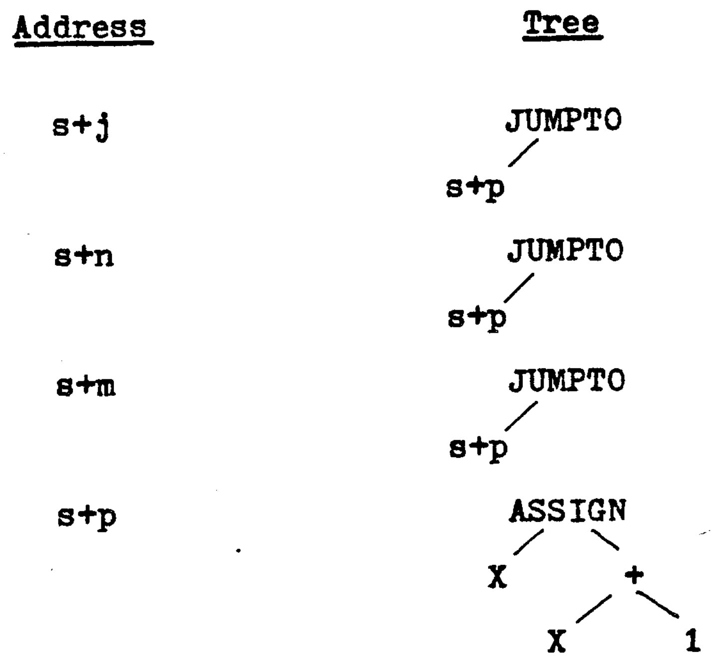
A conventional back-chaining scheme is used to handle forward references. When an undeclared label is entered into the referenced label table, the virtual address of the GO TO statement is entered as the virtual address of the label. Then, every succeeding forward reference to this same label chains itself to the last reference to the label. When the label is finally declared the chain is followed back to the head of the chain, filling in the correct label address at each link. Fig. 6 shows the chain for a forward reference before label declaration and the corrected addresses afterward. (Assume the starting address is s.)

CASE labels are the only other labels allowed in Nucleus. These are handled in a different manner than statement labels. When the alternative sequence of a CASE statement is being parsed, a CASE label table is built for that CASE statement (CASE statements may be nested), each entry having the CASE label number and its virtual machine address. The table is sorted smallest to largest on label numbers and placed at the point following the alternative sequence. The virtual machine location counter is then incremented by one plus the number of labels in the CASE statement, and the next statement is parsed. Thus, the instruction part of the virtual machine includes the sentence tree table with embedded CASE label tables where necessary.



a) Program

b) Before label declaration



c) After label declaration

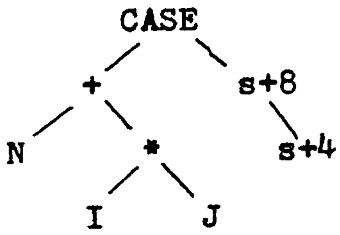
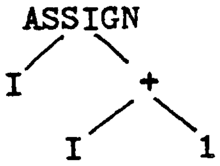
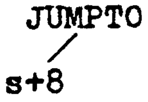
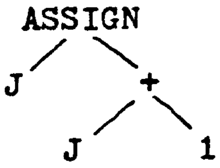
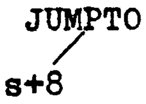
Fig. 6. Forward referencing

The CASE label table is handled this way mainly for code generation purposes and will be discussed more thoroughly in the following chapter. Fig. 7 is an example of a CASE statement and the sentences and tables generated by it. (Assume the procedure starting address is s.)

```
PROCEDURE CASEX;
CASE N+(I*J) OF
   0:  I:=I+1;
 1:2:  J:=J+1;
ESAC;
EXIT;
```

a) Program

Fig. 7. CASE example

Address	Contents
s+0	
s+1	
s+2	
s+3	
s+4	
s+5	0, s+1
s+6	1, s+2
s+7	2, s+2
s+8	EXIT

b) Virtual program segment

Fig. 7. (continued)

Implementation Restrictions

In any programming language implementation, restrictions have to be made in accordance with the machine on which the language is being implemented. Execution implementation parameters, such as integer range, will be discussed in the next chapter. The restrictions discussed here mainly deal with fixed table sizes. In the programming language PASCAL, all tables must be of some fixed length, i.e., no tables can grow dynamically. Therefore, all the tables mentioned in this chapter must have bounds. Table 5 gives these bounds.

Reduced Program Example

Fig. 8 is the reduced program produced by Pass I for the Nucleus program example in Fig. 3. Note the differences in the formal reduced program and the actual reduced program. Especially note that the virtual program sentence trees are in the correct order.

TABLE 5. Restrictions

Entity	Limit
Symbol table	500 identifiers
Constant table	250 constants
Procedure table	50 procedures
Sentence tree table	250 sentences per procedure (including CASE label tables)
CASE label table	50 CASE labels per CASE statement
Declared label table	250 labels per procedure
Referenced label table	100 labels per procedure
Identifier length	60 characters
Nesting depth	25 nested statements

Symbol Table	FIRST LAST MIDDLE X N A	Data Part (no addresses)
Constant Table	100 0 2 1	
Procedures	BINARYSEARCH	

a) Data memory

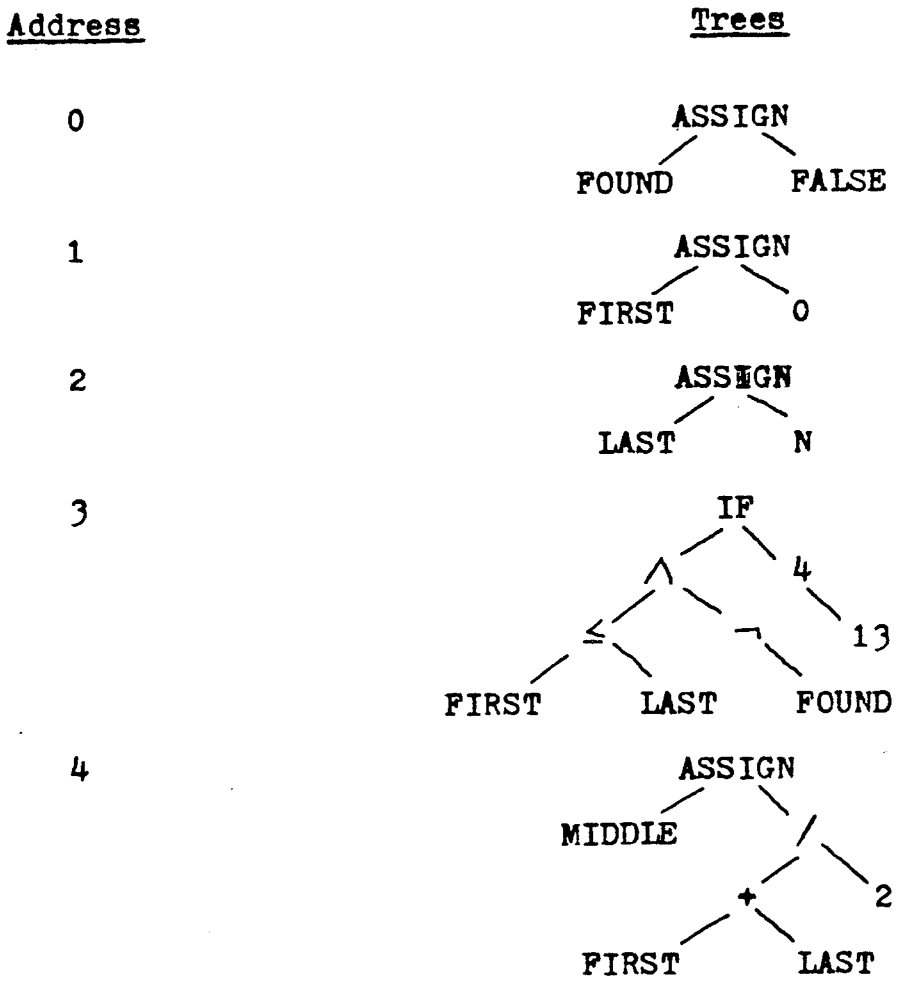
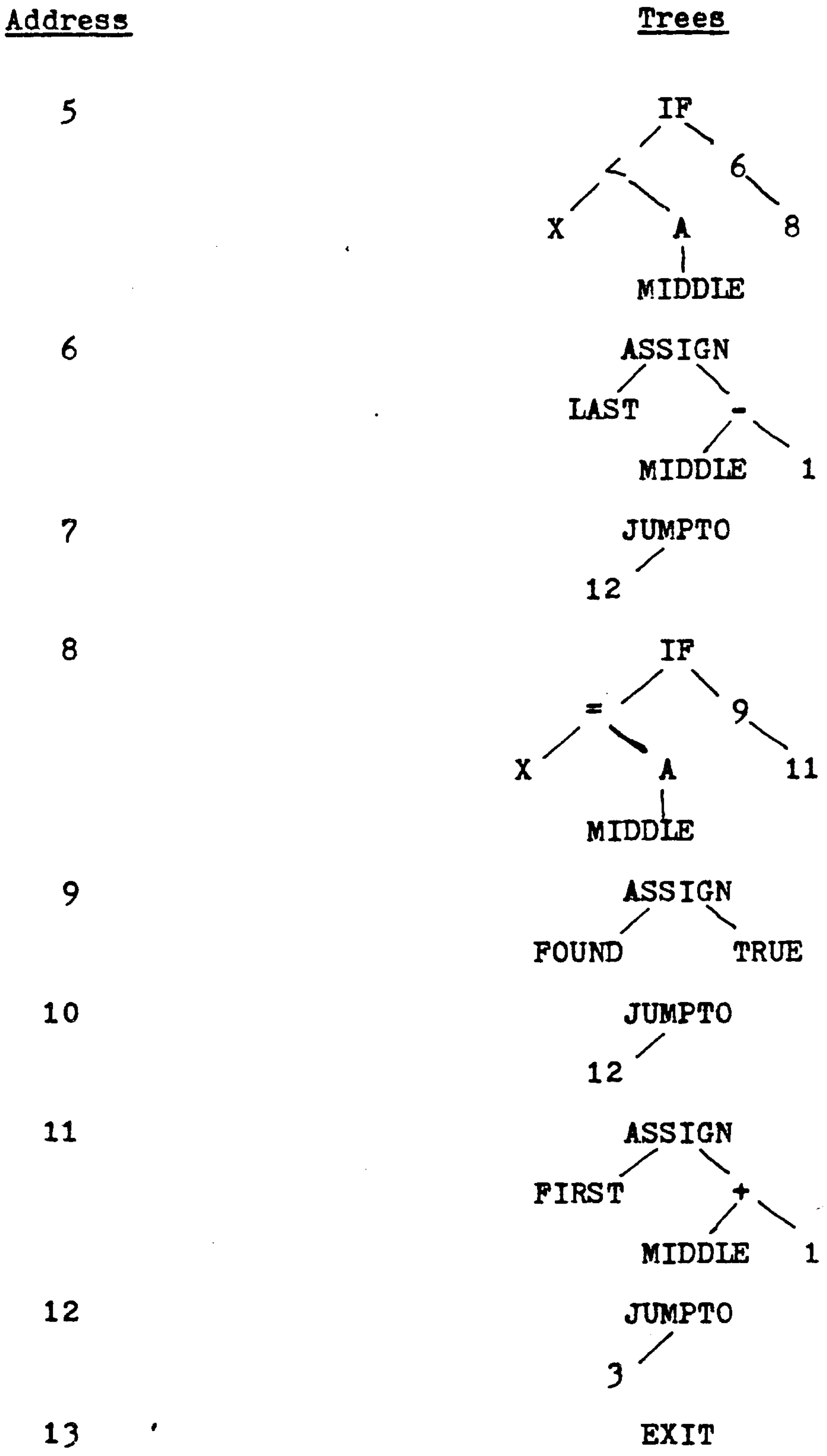


Fig. 8. Reduced program example



b) Instruction memory

Fig. 8. (continued)

CHAPTER IV

PASS II: CODE GENERATION AND EXECUTION

Pass II of the Nucleus compiler generates absolute object code from the reduced program produced by Pass I. Then the PASCAL operating system is used to load the program into memory and execute it. In order to facilitate the use of the PASCAL operating system, much of the structure of Pass II is based on the code generation parts of the CDC 6600 PASCAL compiler. Thus, following the PASCAL system conventions, beginning at absolute address 6001 (octal), memory is allocated as shown in Fig. 9.

Data Storage

Variables are stored beginning at address 6001 as shown in Fig. 9. Simple variables are allocated one word per variable. Array variables are allocated linearly one array element per word. Thus, for the declaration INTEGER ARRAY[49], fifty words of memory would be allocated, with indexing of the array beginning at 0. The variables are allocated in the order in which they are declared in the Nucleus program.

Constants are stored immediately following the variable storage in memory, one constant per word. The constant table itself is actually loaded into memory at the specified address. Fig. 10 shows the storage allocation of the data memory for the example program shown in Fig. 8.

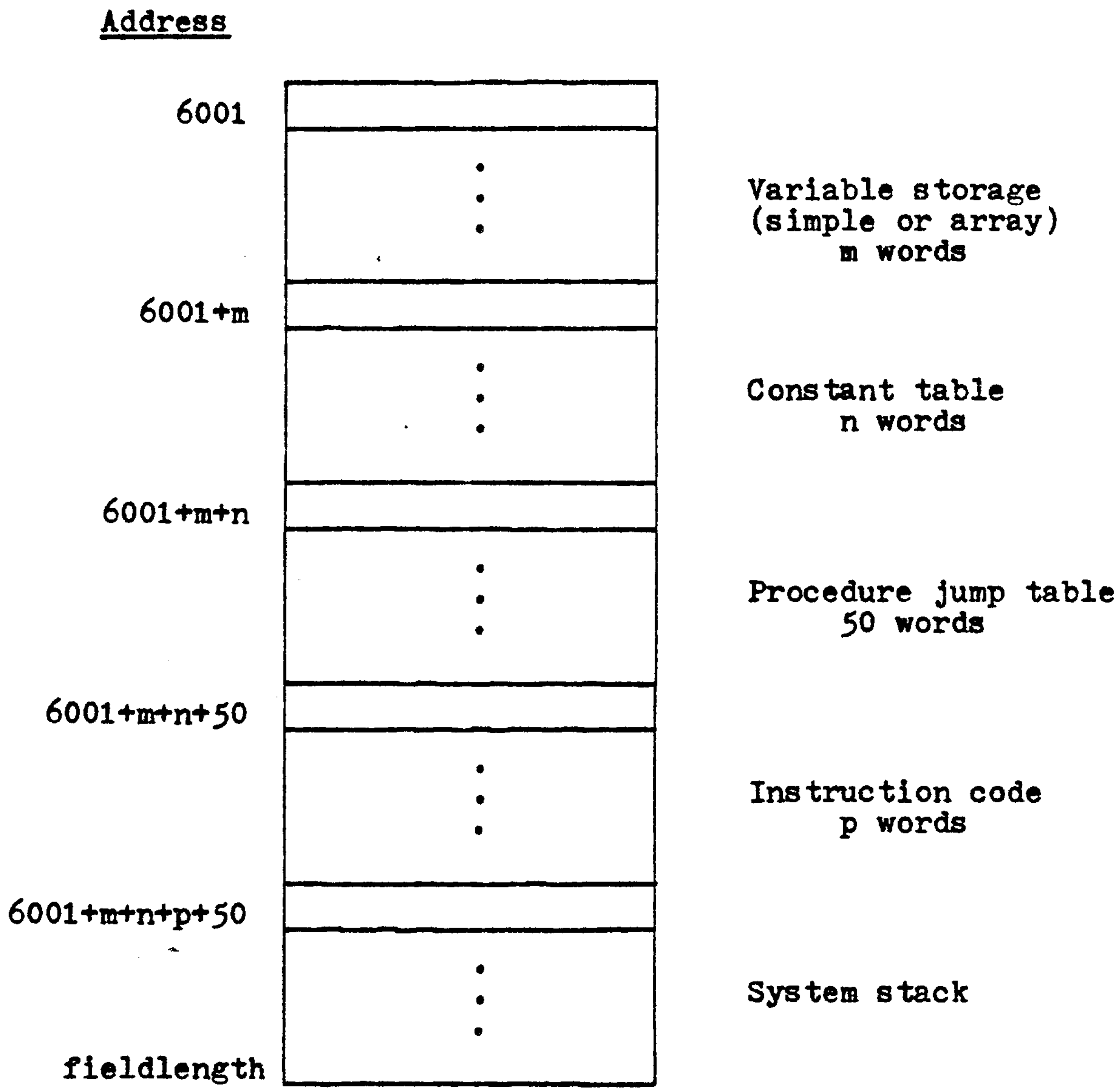


Fig. 9. Run-time memory allocation

Symbol Table	FIRST LAST MIDDLE X N A
Constant Table	100 0 2 1
Procedures	BINARYSEARCH

a) Data memory

Address		Variable name
6001		FIRST
6002		LAST
6003		MIDDLE
6004		X
6005		N
6006		FOUND
6007		A
	⋮	101 words for ARRAY A
6154	100	
6155	0	
6156	2	
6157	1	
6160	jump instruction	BINARYSEARCH

b) Data storage

Fig. 10. Data storage for Fig. 8

Code Generation

Pass II of the compiler generates code for the virtual instructions produced by Pass I by starting at virtual address 0 and continuing through the virtual instruction memory until the last virtual address has been encountered. As discussed in Chapter III, each sentence is stored as a tree structure, with the type of the sentence being the root of the tree. To generate code for each sentence, the sentence type must be determined (see Table 4). When the sentence type has been determined, the appropriate code generation routine is called.

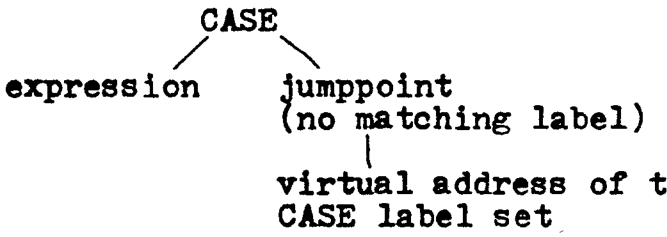
Before code is generated for a virtual instruction, the instruction counter (IC), i.e., the starting address of the virtual instruction object code is tagged on to the root of the sentence tree. Thus, when a reference to a virtual address is encountered in the virtual program, the compiler can immediately determine the absolute address of the object code generated for the referenced virtual instruction. Thus, no problems will occur in a backward reference to a virtual address. For forward references to virtual instructions, a conventional back-chaining scheme is used to insert the absolute address needed.

The code generation routines use a system stack for holding the return point addresses for procedure calls and for holding any temporary values needed while evaluating expressions. The bottom of the stack is located at the absolute address immediately following the end of the absolute program code. Thus, the stack limit depends on how much memory is left after the program is loaded. A stack pointer is maintained in register B6. The maintenance of the stack will be discussed in more detail in the section on expression evaluation and the section on procedure exit code and procedure entry code.

The code generation routines for HALT, JUMPTO, ASSIGN, and IF statements are fairly straight forward and are not discussed further.

CASE Statements

The CASE statement is the most difficult statement to evaluate in Nucleus. As stated in Chapter III, the tree structure of a CASE statement is as follows:



First, code is generated for the evaluation of the CASE expression. Then a jump instruction to a system binary search routine is generated. Before generating the jump instruction, however, code is generated to pass as parameters to the search routine the absolute address of the CASE label table, its length, and the absolute address of the fail point if no matching label exists. A binary search on the CASE labels is possible since the table was sorted smallest to largest before being placed in the virtual program. However, the absolute addresses of the CASE label table and the fail point are not known at this point. Thus, the address of the instructions that store the parameters is saved, and code is generated for all the sentences of the alternative sequence. The last virtual instruction for each body of the alternative sequence is a jump to the end of the CASE statement. Since the absolute address of the CASE statement end has not yet been determined, a back-chaining scheme is used to handle the jump statements. When the CASE label table is encountered, i.e., the end of the alternative sequence is found, the address of the table is placed in the parameter instruction generated at the end of the CASE expression code. Since the fail point for every CASE statement is always the statement following the alternative sequence (the statement following the ELSE, if one exists, or the statement following the ESAC, if no ELSE exists), the absolute address of the fail point can readily be determined at this time. The absolute address of the fail point is the absolute address of the CASE label table plus the length of the table. This address can then be placed in the parameter instruction generated at the end of the CASE expression code.

The object code for the CASE label table is then generated. Each entry of the CASE label table is set up as a jump instruction to the absolute address of the corresponding alternative sequence body. Since the code for all the statements of the alternative sequence has been generated by this time, the jump instructions are easily generated. The jump instruction is placed in the top half of the word (upper 30 bits) and its corresponding label is placed in the lower half of the word. Thus, a restriction is imposed that each numeric label must not exceed $2^{30}-1$.

After the CASE label table jump instructions have all been generated, code is generated for the statements for the ELSE part of the CASE statement, if it has one. Then the address of the end of the CASE statement is placed in the jump instruction at the end of each body of the alternative sequence (the CASE label chain is followed back to the head).

Thus, the object code generated for a CASE statement evaluates the statement according to the following algorithm.

1. Evaluate the CASE expression, leaving its value on the system stack.

2. Jump to the binary search routine, passing the appropriate parameters.

3. Perform the search for the matching label.

4. If the search fails, jump to the absolute address of the fail point given as one of the parameters to the search routine.

5. If the search succeeds, execute the jump instruction in the top half of the word containing the label.

Procedure Entry and Exit

As seen in Fig. 9, the procedure table is stored as a jump table. This means that each entry of the procedure table is actually a jump to the procedure entry point. The jump table is built when the absolute address of each procedure is determined. When a procedure is called, a jump is made to the location in the jump table corresponding to the procedure name. The jump instruction found in the jump table is then executed. The procedure jump table is implemented in this manner so that the Nucleus program code will be compatible with the PASCAL operating system.

A procedure is called by an ENTER statement. Thus, when an ENTER statement is encountered in the virtual program, code must be generated to save the return address on the stack and to jump to the procedure being called. Code is generated that does the following:

1. Push the return address on the top of the stack.

2. Jump to the jump table location corresponding to the procedure being called. The jump instruction in the jump table is then executed.

Thus, when a procedure is entered, the correct return address will be stored on the top of the stack.

When the compiler encounters an EXIT statement in the virtual program, code must be generated for a jump back to the procedure from which the procedure being exited was called. When executed, the code generated here would correspond to the following algorithm. Recall that register B6 points to the top of the system stack.

1. If the system stack is empty, execution halts.
2. Otherwise, pop the return address off the stack, i.e., set B6 to B6-1.
3. Jump to the return address.

Since an EXIT statement marks the end of a procedure, the next virtual instruction is either the first instruction of the next procedure or the end of the virtual program. The EXIT code generation routine checks whether or not the end of the virtual program has been encountered. If not, the absolute address of the next instruction is stored as the absolute address of the next procedure in the procedure jump table.

Read and Write Statements

The Nucleus input routine reads characters from the standard input unit (the card reader) and stores the characters into the array specified by the READ statement. The Nucleus output routine writes characters from the specified array on to the standard output unit (the line printer). Whenever a READ or a WRITE instruction is encountered in the virtual program, object code is generated to save the location of the array variable appearing in the READ or WRITE statement and its size. The parameter that is sent to the input/output (I/O) routine is the minimum of the array size and file record size. Then an instruction to jump to the system I/O routine is generated. The system I/O routines are a modification of the standard I/O routines the PASCAL operating system uses for its standard I/O functions GET(INPUT) and PUT(OUTPUT). See Wirth[12] and Burger[1] for a discussion of the PASCAL standard I/O functions.

Expression Evaluation

ASSIGN, IF, and CASE statements all must evaluate expressions before the actual assignment or test can be made. Thus, the code generation routines for these three statements call the expression code generation routine.

Since expressions in the reduced program are set up as tree structures, object code is generated as an expression tree is traversed. An endorder traversal is made on the expression tree (traverse the left subtree, traverse the right subtree, visit the root of the tree). When generating code for the evaluation of expressions, storage is needed for temporary values. These temporary values are stored on the system stack. All expression operators are either unary or binary. Each operator assumes that the values of its arguments are either the top one or two elements of the stack (depending on the type of the operator). Binary operators pop the top two values off the stack, compute the value, and push the computed value back onto the stack. (Popping the system stack is equivalent to setting B6 to B6-1 while pushing the stack is equivalent to setting B6 to B6+1.)

In order to understand expression evaluation, consider the following example tree structure of an expression.

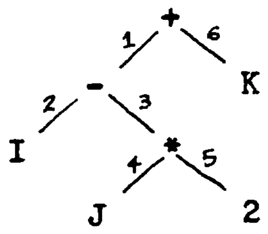


Fig. 11. Tree structure for I-(J*2)+K

The tree traversal is handled by calling the expression code generation routine recursively until a terminal node is encountered. Then the value of the terminal node is pushed on the system stack. Code generation for Fig. 11 would occur in the following way:

Arcs 1 and 2 would be followed to the terminal node containing I. Code to retrieve the value of I and push it on the stack would be generated. Arcs 3 and 4 would be followed to the terminal node containing J. Code to retrieve the value of J and push it on the stack would be generated. Arc 5 would then be followed to the terminal node containing the value 2. Code to push this value on the stack would then be generated. Now, since both the left subtree and the right subtree of the multiplication operator have been traversed, code is generated to pop the top two elements off the stack, perform the multiplication on these two elements, and push the value back onto the stack. Then code would be generated for the subtraction operation in the same manner. Then arc 6 would be followed to the terminal node containing K. Object code to push the value of K on the stack would be generated. Finally, code would be generated that would evaluate the addition operation.

Clearly, an expression tree is structured such that the terminal nodes correspond to primaries. Most of the non-terminal nodes correspond to the operators in Table 2. However, a non-terminal node may also be either an array name or the name of one of the six type-transfer functions available in Nucleus. If an array name is a non-terminal node, its left subtree is the expression that computes the index of the array element being referenced. When an array element is referenced, code is generated to check to be sure that the index value is within the array bounds. If a non-terminal node contains a type-transfer function, its left subtree is the expression whose type is to be changed.

The six type-transfer functions can be found in Table 6. They may be directly called by the user program or called by the compiler during automatic type transfers. An automatic type transfer occurs only on the arguments of the relational operators. The relational operators automatically change all of its arguments to type INTEGER. (The Nucleus syntax requires that both arguments must be of the same type.) The type-transfer functions are implemented as Nucleus run-time system functions and work as shown in Table 6. The values of the functions INTOFCHAR(X) and CHAROFINT(X) correspond to the order of the Nucleus basic character set.

When a call to a type-transfer function occurs while evaluating an expression, code is generated that passes the value to be type-changed and the return point to the appropriate type-transfer function.

Implementation Parameters

As mentioned previously, the Nucleus language is defined in terms of several implementation parameters. These parameters define the integer size, the I/O record sizes and the stack size. They are all dependent on the machine on which Nucleus is implemented. Thus, the parameters must correspond to the word size and I/O record sizes of the CDC 6600. MAXSTACKSIZE defines the size of the system stack. Since the system stack is located immediately after the user

TABLE 6. Type transfer functions

```
INTOFBOOL(X) = 0  if X = FALSE
             = 1  if X = TRUE

INTOFCHAR(X) = 0  if X = blank
             = 1  if X = A
               .
               .
               .
             = 63 if X = #

BOOLOFINT(X) = FALSE if X mod 2 = 0
             = TRUE  if X mod 2 = 1

BOOLOFCHAR(X) = BOOLOFINT(INTOFCHAR(X))

CHAROFINT(X) = blank if X mod 64 = 0
             = A     if X mod 64 = 1
               .
               .
               .
             = #     if X mod 64 = 63

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

program in memory, the stack size is dependent on the field length (FL) requested by the user program and the actual field length of the user program (PL). Table 7 gives the values of these implementation parameters.

Execution

All Nucleus programs are executed by the PASCAL operating system. Thus, the code generated by the procedures described previously is kept on a file to be loaded into memory and executed. Before execution occurs, however, the last absolute address used, the first absolute address of the object code, the address of the procedure jump table, and the address at which to start execution must be stored in the appropriate locations in order to be picked up by the PASCAL operating system, and the program is loaded and executed.

TABLE 7. Implementation parameters

<u>Parameter</u>	<u>Value</u>
INRANGE(X) (integer range)	$-2^{48}+1$ X $2^{48}-1$
READSIZE (input record size)	80 characters
WRITESIZE (output record size)	136 characters
MAXSTACKSIZE	FL-PL

CHAPTER V

SUMMARY

The Nucleus compiler is implemented as a two-pass compiler. The first pass, discussed in Chapter III, accepts a Nucleus program and produces its reduced program as output. The second pass, discussed in Chapter IV, generates absolute object code from the reduced program. A complete listing of the Nucleus compiler can be found in Appendix A. Appendix B contains the control cards and compiler options available to Nucleus users for this implementation. More options may be desired in the future. If so, a description of the option flag and its use can be found within the compiler listing of Appendix A.

The implementation of the Nucleus compiler in two passes clearly points out the idea of a virtual machine, as discussed in Chapter II. The reduced program produced by Pass I can be thought of as a set of virtual instructions written in a virtual instruction language - the virtual instruction language being defined by the nine sentence types that can be constructed by Pass I. The reduced program is then used as an "input language" to the second pass of the compiler. Pass II then interprets the sentences and generates the appropriate object code. Not only can the reduced program be used as input to Pass II of this compiler, but it can be used as input to a verification condition compiler such as the VCC by Ragland[8] or as input to other Nucleus verification systems.

The Nucleus implementation not only reflects the virtual machine idea but also follows the definition of Nucleus described in Chapter II as closely as possible. Only those constructs that are restricted by the machine, such as the identifier tables and expression stacks, are not implemented as explicitly stated in the formal definition of Nucleus. The implementation restrictions imposed on each pass are stated at the end of the chapters which discuss Pass I and Pass II.

Modifications to the compiler may be desirable at some future date, particularly in the code generation routines. If more efficient code is desired, the first place to start would be in the code generation routines for the evaluation of expressions. Since the major statement types, such as ASSIGN, IF, WHILE, and CASE rely heavily on the expression code generation routines, more efficient code could then be generated for the over-all program.

Another desirable modification may be to allow the input/output (I/O) files to be files other than just the CDC 6600 standard I/O files. The system I/O routines would then have to be adjusted accordingly to accept input and output from these other files.

It is now possible to run Nucleus programs on the CDC 6600. More importantly, the four-phase project discussed in Chapter I is one step closer to completion with the completion of this compiler. The next step now is to write a Nucleus compiler in Nucleus, prove it correct using the verification condition compiler by Ragland[8], and compile it on this compiler. Then, a correct VCC and a correct compiler would be available from which to begin a process of bootstrapping more and more proved processors for more and more languages.