ESD-TR-67-51

Technical Note

1967-12

L. F. Mondshein

VITAL
Compiler-Compiler System
Reference Manual

8 February 1967

Prepared for the Advanced Research Projects Agency
under Electronic Systems Division Contract AF 19(628)-5167 by

Lincoln Laboratory

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Lexington, Massachusetts

The work reported in this document was performed at Lincoln Laboratory, a center for research operated by Massachusetts Institute of Technology; this work was supported by the U.S. Advanced Research Projects Agency of the Department of Defense under Air Force Contract AF 19(628)-5167 (ARPA Order 691).

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LINCOLN LABORATORY

VITAL COMPILER-COMPILER SYSTEM REFERENCE MANUAL

L. F. MONDSHEIN

Group 23

TECHNICAL NOTE 1967-12

8 FEBRUARY 1967

LEXINGTON MASSACHUSETTS

ABSTRACT

This manual describes the general operation of the VITAL compiler-compiler system and the details of Production Language (PL) and Formal Semantic Language (FSL).

The Appendices contain information on the system's meta-commands, a guide to the use of PL, an example of an ALGOL compiler, and a table of symbols used in PL and FSL.

Accepted for the Air Force
Franklin C. Hudson
Chief, Lincoln Laboratory Office

PREFACE

This document is intended primarily as a reference manual for users of the VITAL system as it is presently implemented on the Lincoln Laboratory TX-2 Computer. It is not an introductory text. However, hopefully it will be helpful to other readers who wish to acquaint themselves with the operation of the system. Such readers are advised to study Chapter 1 and Appendix B, read Chapters 2 and 3 in a cursory manner, and then proceed to a careful study of Appendix C.

The author wishes to thank Dr. Jerome A. Feldman and James E. Curry for the tutelage they gave in the preparation of this material.

TABLE OF CONTENTS

---

*By James E. Curry, C.W. Adams Associates, Inc.

CHAPTER I
THE VITAL SYSTEM

1. GENERAL DESCRIPTION

Purpose

The purpose of the VITAL system is to mechanize the details of compiler-writing for a wide class of potential compilers. The term VITAL denotes Variably Initialized Translator of Algorithmic Languages.

Figure 1, explained below, presents a functional view of the system.

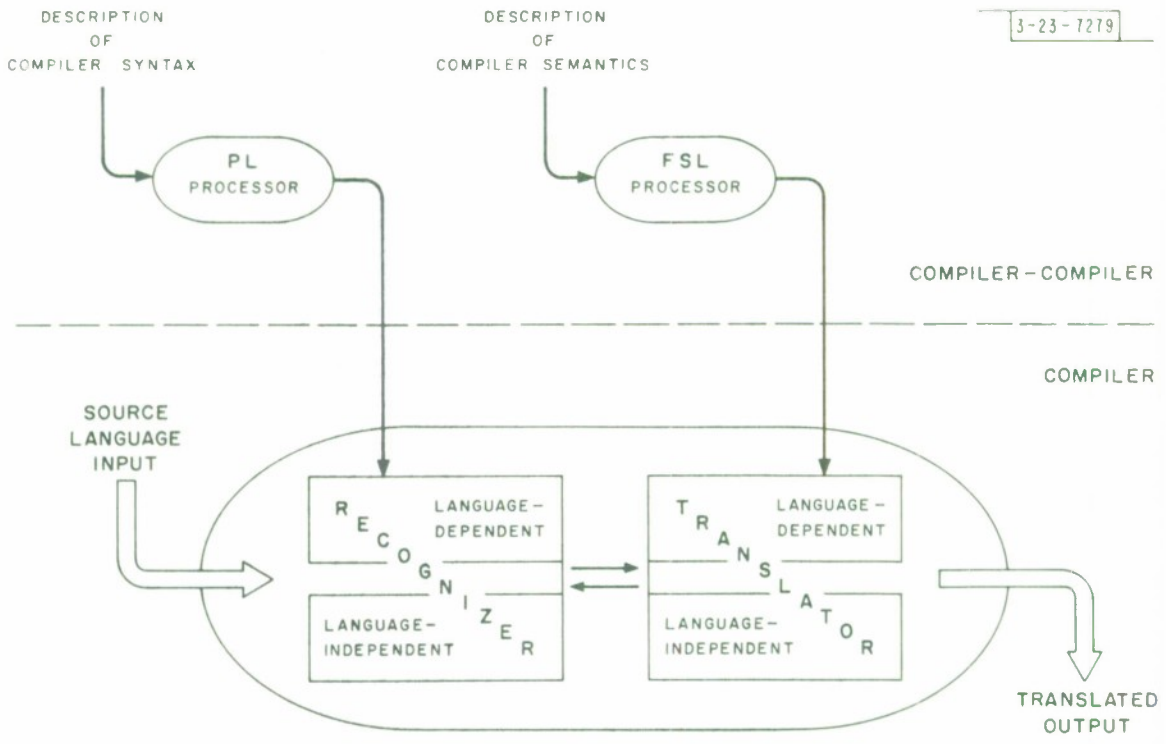

FIGURE 1

Compiler Structure

A compiler produced by VITAL consists of two units: a recognizer and a translator. The recognizer identifies syntactic features of the source text; the translator creates output code.

Each of these units is partitioned into a language-dependent and a language-independent section; the language-independent section is the same for all compilers.

Recognizer

The recognizing or parsing of input text is implemented by the production interpreter, which is driven by a table of productions. The production interpreter is language-independent; the productions are language-dependent.

The compiler writer describes the productions in a high-level language called Production Language, or PL (See Chapter 2).

Translator

The translation of input is produced by a set of semantic routines. These routines contain calls to a collection of code generator subroutines. The code generators are language-independent; the semantics are language-dependent.

The semantics are specified by the compiler writer in a high-level language known as Formal Semantic Language, or FSL (See Chapter 3).

Compiler-Compiler Construction

Both the PL and FSL processors have the same structure as the compiler illustrated in Figure 1. They were created by a hand-coding and bootstrapping operation.

First, the language-independent code generators and production interpreter were hand-coded. Then the PL processor was constructed by hand-coding its semantics and productions. The FSL processor was produced by hand-coding its semantics and writing its productions in PL.

It might be noted that the language-independent sections are identical in the PL processor, FSL processor, and all VITAL-produced compilers, with the single exception of the PL code generators. These last are a small subset of the standard collection of code generators, due to the relatively simple nature of the PL output.

2. OPERATION OF COMPILER

Productions, Semantics, Main Stack

The main stack is part of system storage. Its function is both syntactic and semantic. When the history of the stack is displayed as a tree (Appendix B, Figures 2 and 3), the result is a parse of the input text being compiled. The role of the productions is (1) to generate the successive states of the stack during a single scan through the input text and (2) to transfer control to the proper routines in the semantics whenever certain specified syntactic constructs occur in the stack.

Each stack entry consists of three consecutive machine words: the syntax word, used by the productions; and two semantic words, used by the semantics. The syntax word is discussed in the next section. Details of the semantic words are described in Chapter 3, 5.4.

The productions and semantics perform complementary functions with regard to the main stack. The productions implement the analysis of syntax by manipulating the syntax portion of the stack; the semantic

routines preserve all information needed for the generation of code, updating the semantic portion of the stack whenever necessary.

Processing of Input Text

The compiler makes a single scan through the symbols in the input text. These symbols fall into two classes: (1) those which the productions have been set up to recognize as reserved words; these words are part of the language in which the text is written (example: DO or + in FORTRAM or ALGOL); (2) all other symbols, which are referred to as identifiers.

When a reserved word is scanned, a representation of this symbol is pushed onto the main stack (in the syntax word); when an identifier is scanned, an item denoting "identifier" is entered instead. The corresponding semantic words are of no relevance at this point.

Following a scan operation, the top few syntax entries are compared with a table of expected stack configurations, i.e. expected states of the stack. This table is part of the productions. When a match is found, a collection of routines associated with the matched configuration are executed. These routines may do any of the following: modify the syntax portion of the stack; jump to semantic routines; record an error message; cause one or more symbols to be scanned; or specify the next configuration to be compared with the stack.

The process of scanning and matching continues until all the symbols in the input program have been scanned or a termination command is reached in the productions or semantics.

CHAPTER 2

PRODUCTION LANGUAGE

Production Language[3*] (PL) is the language used for specifying the productions. It is translated by the PL processor. In what follows, the term productions will refer to the PL program as well as the result of its translation; and the two will be treated as if identical.

As noted in Chapter 2 the function of the productions is (1) to generate successive states of the main stack on the basis of a single scan through the input text and (2) to transfer control to the proper routines in the semantics whenever certain specified syntactic constructs occur in the stack.

The following sections present the syntax of PL. A guide to the writing of productions is given in Appendix B.

1. PROGRAM STRUCTURE

A PL program consists of a collection of declarations followed by a number of lines each of which is called a production. The entire program is terminated by the symbol END.

2. DECLARATIONS

2.1 CHOPPER DECLARATIONS

NORM <argument>

VTERM <argument>

ITERM <argument>

Permissible argument: A sequence of Lincoln Writer codes (in octal) separated by spaces.

Description: A code appearing in a NORM declaration is treated as a normal character; a code appearing in a VTERM or ITERM declaration is treated as a visible terminator or ignored terminator, respectively. This terminology is explained below.

---

* Production language uses a slightly modified version of the formalism introduced in Reference 3.

Each VITAL symbol is composed of one or more VITAL characters. A VITAL character may be either a single Lincoln Writer character or a compound character composed of at most two "dead" (non-spacing) and two "line" (spacing) Lincoln Writer characters.

A VITAL symbol is either (1) a single Lincoln Writer character which is classified by the system as a terminator (either visible or ignored), (2) a compound character (which is automatically classified as a terminator), or (3) a string of non-terminators between two terminators. (Non-terminators are called normal characters.)

An ignored terminator, unlike a visible terminator, is not pushed onto the stack by the scan operation (see Chapter 1, section 2). For example, a space is generally treated as an ignored terminator.

Each Lincoln Writer code is classified by the system in a standard manner (see Appendix A) unless it is reclassified by a CHOPPER DECLARATION.

In Chapter 3 the term identifier will be used to refer to any VITAL symbol whose first character is not an integer.

2.2 RESERVED WORDS: RES <argument>

Permissible argument: A sequence of VITAL symbols separated by spaces (not commas).

Description: Symbols declared as reserved words become part of the language whose syntax is being specified.

Remarks: If a non-printing character or a character which is part of PL is to be declared a reserved word, it must be denoted by a special identifier to avoid confusion.

character	special identifier
↑	pHAND
=	pEQ
superscript	pSUP
subscript	pSUB
normal	pNOR
→	pAROW
‖	pDBAR
carriage return	pCR

If a word which is part of PL is to be declared a reserved word, it must be preceded by a "q". The words in this category are EXEC, STAK, UNSTK, SCAN, ERROR, HALT, NEXT, CALL, RETURN, TEST, DONE, END, RES, INT, NORM, VTERM, ITERM.

2.3 INTERNAL SYMBOLS: INT < argument >

Permissible argument: A sequence of VITAL identifiers separated by spaces (not commas).

Description: Internal symbols are symbols which are neither part of the language being specified nor part of Production Language. They are entries to be made by the compiler in the syntax portion of the stack.

2.4 CLASS NAME: < argument 1> = < argument 2 >

Permissible arguments

argument 1: any VITAL identifier (the "class name").

argument 2: a sequence of reserved words (see 2.2).

Example: ϵ BOOL = ∨ ∧ THEN pAROW

Description: A class name is simply a notational convenience; a production containing a class name is equivalent to a sequence of productions containing each of the corresponding reserved words.

2.5 CLASS NAME WITH ASSOCIATED SEMANTIC ROUTINE NUMBERS

Example: TOTH ≡ TO 116 THRU 117

Form: Same as 2.4, except that a triple equal sign is used and each reserved word is followed by an associated semantic routine number.

The use of such a class name is discussed in section 3, (5).

3. PRODUCTIONS

The syntax of a production is described most easily by example. The following production is typical.

① BE3 is a label. Labels must be followed by ||. (All labels are optional.)

② IF E THEN is one member of the table of "expected stack configurations" discussed in Chapter 1, section 2. In this example, IF and THEN are reserved words; E (denoting "expression") is an internal symbol.

The routines indicated by ③ – ⑤ are executed only if the top of the main stack matches ②.

③ The arrow indicates that the part of the stack matching ② is to be replaced by ④ (which can be empty). The absence of an arrow indicates that no alteration is to be made.

④ ICL is an internal symbol (denoting "if clause"). In the example, the top three entries in the stack are removed and ICL is pushed onto the stack in the syntax word. The contents of the corresponding semantic word are determined by semantic routine 110 (see ⑤).

Formal Semantic Language provides primitives for manipulating the semantic words of the stack once a match is found. The semantic words associated with the stack syntax words corresponding to ② are denoted by Ln, LLn ($1 \leq n \leq 7$) (recall that there are two semantic words per stack entry; Ln is the first, LLn the second). The semantic words associated with the stack syntax words corresponding to ④ are denoted by Rn, RRn ($1 \leq n \leq 4$). (L1, LL1 and R1, RR1 refer to the top of the stack.) See Chapter 3, 1.2.

⑤ EXEC 110 causes transfer of control to semantic routine 110 (see Chapter 3, 5.1). A class name may be used instead of a number. In such a case, a a transfer will be made to routine n, where n is the number associated with the reserved word actually in the stack (see 2.5).

← S0 is a transfer indicator, causing the main stack to be compared next with the production labelled S0. If no transfer indicator appears, the stack will be compared with the succeeding production.

Both EXEC 110 and ← S0 are known as actions. Additional actions are:

UNSTK m:	remove the top m entries from the stack and save in storage
STK n:	take the n^{th} entry (counting from the top) removed by UNSTK and push it onto the stack (including both the syntax and semantic words).
STAK <argument>:	push the argument (a reserved word or internal symbol) onto the stack.
SCAN:	scan one input symbol.
SCAN n:	scan n input symbols.
ERROR n:	store error message n in output buffer (for either console or Xerox print-out).
HALT n:	halt execution of productions, store n in output buffer, return to system control.
NEXT <argument>:	same as ← <argument> (see ⑤).
TEST <argument>:	same as ← <argument>, except that the action has effect only if SIGNAL has the value TRUE (see Chapter 3, 1.20).
RETURN:	return to production which made the last CALL (see below).
CALL <argument> (where argument is a production label):	execute the productions starting at the label, and continue until the action RETURN.

4. REMARKS

Two frequently used symbols which are part of PL are SG (sigma) and I (identifier). When used in part ② of a production, SG matches any item in the stack. "I" is the symbol pushed onto the stack whenever an identifier is scanned. (See Chapter 1, section 2, OPERATION OF TRANSLATOR).

CONTENTS OF CHAPTER 3

2. ARITHMETIC AND BOOLEAN EXPRESSIONS

3. UNCONDITIONAL AND CONDITIONAL STATEMENTS

Unconditional Statements

Storage Manipulation

3.1 Stack Commands

3.2 Enter Command

3.3 Store

3.4 Tally

3.5 Storage Operations

Control

3.6 Assign

3.7 Label

3.8 Subroutine Name, Subroutine Entrance Name

3.9 Return

3.10 Exit

3.11 Jump

3.12 Call

3.13 Fault

3.14 Stop

3.15 Complete

Conditional Statements

3.16 If ... then

3.17 If not ... then

3.18 If ... then ... else

4. DECLARATIONS

4.1 Cell Delcaration

4.2 Index Declaration

4.3 Run-index Declaration

CHAPTER 3

FORMAL SEMANTIC LANGUAGE

Formal Semantic Language[1,2] (FSL) is the language used to specify the semantics. It is translated by the FSL processor. The components of FSL are discussed in the first four sections below. Section 5 discusses general features of the language. Careful attention should be given to 5.3 (code brackets) and 5.4 (semantic description).

1. PRIMARIES AND BOOLEAN PRIMARIES

The primaries and boolean primaries are the basic constituents of FSL. They include variables whose values are maintained by the system or by the user, as well as operators.

For example, CODELOC (see 1.1) is a system-maintained variable whose value is the address in which the next compiled instruction will be stored. Cells (see 1.3), which are the simplest user-maintained variables, correspond to single machine words.

In the description of the syntax of an operator, the argument is denoted by the symbol < argument >.

Primaries

1.1 SYSTEM CELL: CODELOC, STORLOC, PERSLOC, ACCUM, MARK, MAIN, SAVCELL

Description: The system cells contain pointers or bits that are useful to both the language designer and the system.

> CODELOC points to the location in which the next word of code will be stored.
>
> STORLOC points to the next free register in the run-code storage area. (The term run code refers to the code generated by the compiler.) Proper manipulation of STORLOC enables the compiler-writer to conserve storage space at the time the run-code is executed.

PERSLOC points to storage ("persistent storage") that can be loaded during compilation and utilized during the running of the compiled program. For example, non-integer constants may be stored in the PERSLOC area.

ACCUM contains information concerning the state of the run-code accumulator. For example, whenever an instruction is compiled which loads the accumulator, the βINA bit in ACCUM is set to 1 (see 4.8). When an instruction is compiled which stores from the accumulator, the βINA bit is cleared. Other bits which may be set are: βDIR, βNEG, and βCON (see 4.8).

MARK preserves information used during the compilation of (possibly recursive) subroutines. It points to a register containing the return address associated with the last subroutine call or subroutine entrance call (3.12). This register implements a push-down stack used by the RETURN command (3.9), thus permitting recursive subroutine calls.

MAIN points to the top of the main stack (see Chapter 1, section 2).

SAVCELL points to a temporary address generated by the SAVTEMP statement (see 3.5).

1.2 PRODUCTION OPERAND: L1, LL1, ..., L7, LL7, R1, RR1, ...R4, RR4

Description: L1, LL1 – L7, LL7 are the first seven semantic entries of the main stack before a production is executed. R1, RR1 – R4, RR4 are the first four semantic entries of the stack after completion of the entire production. (See Chapter 1, section 2; also Chapter 2, section 3.)

Remarks: The production operands provide a convenient way of manipulating the semantic portion of the stack. For example, the store commands (see 3.3) "L1 → R2; LL1 → RR2; L2 → R1; LL2 → RR1" permute the contents of the top two semantic entries.

1.3 CELL IDENTIFIER

Form: Any VITAL identifier (see Chapter 2, section 2.1).

Description: Cells are single storage locations within the compiler. Cells must appear in a cell declaration (4.1) before being used elsewhere.

1.4 STACK IDENTIFIER

Form: Any VITAL identifier

Description: A stack identifier is the name of a push-down stack. Before its use, the identifier must appear in a STACK or RSTAK declaration (4.5, 4.6), in which case it refers to a stack in the compiler or run-code, respectively.

The stack may be operated upon by means of the stack commands (3.1). When used as an ordinary operand the value of a stack identifier is the entry at the top of the stack.

1.5 TABLE OPERAND

Examples: SYMTABL { L1, ADDR }; JMPTABL { L3, LOC, 1}; LABLTAB {0, COL3}

Form: < argument 1> { < argument 2>, < argument 3>, < argument 4 > }

Permissible arguments

argument 1: table identifier, i.e. identifier appearing in a table declaration (4.4).

argument 2: primary or 0.

argument 3: column name (see 4.4).

argument 4: 0 or 1 ("mode"; optional if 0).

Description: The value of the table operand is the table entry found by the following procedure. The first column of the table is searched for an entry equal to argument 2 if it is a primary. The value of the table operand is taken from the row containing this entry and the column identified by the column name. If argument 2 is 0, the row used is that referenced by the last table operand search.

If the mode (argument 4) is 0 or unspecified, the table is searched upward from the bottom entry; if the mode is 1, the search starts upward from the last row referenced by a table operand search (or from the bottom entry if there has been no previous search).

If no match is found for the primary, SIGNAL (see 1.22) is set to FALSE (see 1.18) and the table operand is given the value 0.

Like most primaries, a table operand may appear either inside or outside code brackets (see 5.3).

1.6 FLAD: FLAD1, ..., FLAD4

Description: Flads are intended to facilitate forward references such as jumps to program locations not yet determined. FLADn is used as an operand, its value being the value of CODELOC at the time the assign command FLADn ← is executed (see 3.6).

Note: (1) Each flad is a stack; the value of FLADn is the entry at the top of the stack. To use flads in nested procedures, do a PUSH FLADn (see 3.1) at each successive level of nesting. The assign command will automatically pop the stack.

(2) A flad can handle forward references only; its value depends upon the corresponding assign command which first follows its use.

1.7 CHAIN: ⨍ <argument>

Permissible argument: primary

Description: The purpose of the chain operator is to permit compilation of forward and backward references such as jumps to program locations; hence, the operator appears only within code brackets (see 5.3).

The expression ⨍ < argument> is used as an operand, its value being a semantic description (5.4) of the value of CODELOC (see 1.1) at the time the assign command < argument> ← is executed (see 3.6). The system handles the problems arising from the separation in time between the appearance of ⨍ < argument> and the corresponding assign command.

The argument of the chain operator must have as its value the address of an arbitrary machine word initially containing 0. A typical example of the use of a chain is "0 → TEMP; λ TEMP → VAR; ⊏ JUMP ⨍ VAR ⊐ " (TEMP and VAR are cells; see 5.3 for a discussion of the symbols ⊏ ... ⊐ .) TEMP is called the cell through which VAR is chained. If VAR is chained through a different cell, the chain takes on a new value. The original value is restored by chaining VAR through TEMP once again (provided TEMP has not been modified).

1.8 CONTENTS: k <argument>

Permissible argument: primary

Discussion: Both the contents of a machine word and its address may be needed during compilation. This and the following operator provide a way to refer to each of these items in terms of the other. For example, if TEMP is a cell, the value of λ TEMP is the address of the machine word containing the value of TEMP. The value of kTEMP is the contents of the word is whose address is the value of TEMP (assuming that the value of TEMP is an address). The term run code used below refers to the code being compiled.

Description: Outside code brackets (see 5.3), it is assumed that the value of the argument is an address. The value of k <argument> is the present contents of this address.

Inside code brackets, it is assumed that the value of the argument is a semantic description (see 5.4) of a run-code address. The value of k <argument> is a semantic description referring to the contents of this address, i.e. to this address deferred.

Remarks: The contents operator may be nested only once. I.e. kk < argument> is permissible; kkk < argument> is not.

1.9 LOCATION: λ <argument>

Permissible argument: table identifier or primaries 1.1 – 1.8

Description: When the argument is a table identifier, the value of λ < argument> is a pointer (a 17-bit address) to the first free register in the table.

The value of λ <argument> is described separately for each of the primaries 1.1 — 1.8.

(1) System Cell. The value of λ < argument> is the address of the machine word corresponding to the system cell.

(2) Production Operand. When used outside code brackets (see 5.3), the value of λ < argument> is the address of the machine word corresponding to the production operand.

When used inside code brackets, the value of λ <argument> is a semantic description (see 5.4) of the operand (e.g. machine address, index number) containing the operand described by < argument> . Because of the use of persistent storage (see 1.1) and pointers in the compilation process, the former operand already exists. However, such an operand would not be available for λ λ < argument> , and nesting is therefore not allowed.

(3) Cell Identifier. The value of λ < argument> is the address of the machine word corresponding to the cell identifier.

(4) Stack Identifier. The value of λ < argument> is a 17-bit pointer to the top of the stack (when used outside code brackets with a stack) or a semantic word description of this pointer (when used inside code brackets with a run-stack). See 4.5, 4.6.

(5) Table Operand. The value of λ < argument> is the address of the word in the table containing the looked-up table entry (see 1.5).

(6, 7) Flads, Chains. The value of λ < argument> is a semantic description of a run-code address containing the value of the flad or chain.

(8) Contents. The location operator is the inverse of the contents operator.

1.10 TYPE: t < argument>

Permissible argument: primary

Description: It is assumed that the value of < argument> is a semantic description (see 5.4). The value of t <argument > is this description with all bits masked out except the data type bits (4.1 – 4.5).

1.11 ABSOLUTE VALUE: || <argument >

Permissible argument: production operand (or operand whose value is a semantic description (see 5.4).

Description: The address portion of a production operand sometimes contains an address in complemented form. || < argument > differs from <argument > only only in that the address is represented in non-negative form.

1.12 CONSTANT

Description: Numerical constants may be used in FSL. Outside code brackets, a constant c is interpreted as a cell with value c. Inside code brackets, c is interpreted as an operand whose value is a semantic description (see 5.4) of a run-code register containing c.

1.13 PARENTHESIZED EXPRESSION: (< argument >)

Permissible argument: arithmetic expression (see Section 2).

Remarks: Parentheses may be used whenever desired to indicate the grouping of arithmetic expressions.

1.14 CODE-BRACKETED ARITHMETIC EXPRESSION: ⊂ < argument > ⊃

Permissible argument: arithmetic expression.

Description: The value of the bracketed expression is a semantic description (see 5.4) of the result of performing the operation(s) specified by the arithmetic expression.

In addition to being a primary, a code-bracketed arithmetic expression may cause run-code to be generated (see 5.3).

1.15 STORE WITH VALUE: < argument 1 > ≡ < argument 2 >

Permissible arguments

argument 1: arithmetic expression. (This includes primaries, see section 2.)

argument 2: primary.

Description: STORE WITH VALUE is both a primary and an unconditional statement. It has the same effect as an assignment statement. In addition, it may be used as a primary having the same value as the first argument.

Example: TEMP0 ≡ TEMP1 + 1 ≡ TEMP2.

TEMP0 is stored into TEMP1; the primary TEMP0 ≡ TEMP1 has the same value

as TEMP0. This value is incremented by 1 and the result is stored in TEMP2; this result is likewise the value of TEMP0 ≡ TEMP1 + 1 ≡ TEMP2. The same effect would be obtained if parentheses were used as follows: (TEMP0 ≡ TEMP1)+ 1 ≡ TEMP2.

1.16 TAG: < argument1 > | < argument2 >

Permissible arguments

argument 1: primary.

argument 2: list of data or bit identifiers separated by commas (see 4.7, 4.8).

Example: TEMP | ΔBOOL, βINA

Description: Argument 1 is tagged with the bits specified in argument 2. The tagging operation consists of clearing the left half of the word being tagged, followed by successive unite operations with the left half of words containing the specified bits. Thus, the right half of the tagged word is preserved.

1.17 ↓ FIX: < argument >

Permissible argument: primary whose value is a semantic description.

Description: The fix operation is used only within code brackets. The operand described by the primary is made into an integer by means of truncation.

Boolean Primaries

1.18 BOOLEAN CONSTANT: TRUE, FALSE

Description: The two boolean constants are the values of the boolean primaries which follow. They may be used as operands of the store command (3.3), in a manner similar to numerical constants (see 1.12).

1.19 TEST: < argument1 > IS < argument2 >

Permissible arguments

argument 1: primary.

argument 2: data id, sub-data id, or bit id (see 4.7, 4.8, 4.9).

Description: This primary has the value TRUE or FALSE depending on whether the argument on the left is or is not tagged with the bits specified on the right.

1.20 CONSTANT TEST: CONST < argument >

Permissible argument: primary.

Description: The value is TRUE or FALSE, depending on whether or not the β CON bit is set in the argument. This primary is meaningful only when its argument is a production operand or other operand whose value is a semantic description (see 5.4). It can be used outside code brackets only.

1.21 RELATION

Example: < argument > = < argument > (Relations which can be used other than include ≠ , <, >, ≤, ≥)

Permissible argument: primary

Description: The primary has value TRUE if the relation is satisfied, FALSE otherwise.

1.22 BOOLEAN CELL: OK, SIGNAL

Description: The cells OK and SIGNAL are given the value TRUE or FALSE by the user and system, respectively.

If OK is set FALSE, no further code will be produced, though the input will continue to be scanned. OK cannot be reset once it is FALSE; it is intended to be used only for irrecoverable syntax errors.

The system gives SIGNAL the value TRUE or FALSE upon successful or unsuccessful completion of certain routines. For example, SIGNAL is set TRUE or FALSE depending upon whether a table search does or does not succeed in finding the desired item (see 1.5).

1.23 PARENTHESIZED BOOLEAN EXPRESSION: (< argument >)

Permissible argument: Boolean expression (see Section 2).

Description: Parentheses may be used whenever desired to indicate the grouping of boolean expressions.

2. ARITHMETIC AND BOOLEAN EXPRESSIONS

Arithmetic expressions are formed from the primaries and the operators +, - (both binary and unary), *, /, and exponentiation.

Boolean expressions are formed from the Boolean primaries and the operators $\sim$, $\wedge$, $\vee$ (inclusive or), $\oplus$ (exclusive or). The usual dominances apply but may be overridden by parentheses.

Primaries and boolean primaries are themselves considered as arithmetic and boolean expressions, respectively.

3. UNCONDITIONAL AND CONDITIONAL STATEMENTS

The statements of FSL consist of commands used for manipulating storage and modifying program flow.

Unconditional Statements

Storage Manipulation.- The following commands facilitate manipulation of storage.

3.1 STACK COMMANDS

PUSH < argument 1>

PUSH { < argument 1 > , < argument 2 > }

POP < argument 1 >

POP { < argument 1 > , < argument 2 > }

Permissible arguments

argument 1: stack identifier (see 1.4).

argument 2: primary.

Description: If argument 2 is not specified, PUSH takes the second argument as zero, while POP simply discards the top entry of the stack.

When the commands are used outside code brackets, PUSH places the value of argument 2 on top of the stack, while POP removes the top item in the stack and stores it in argument 2.

When commands are used inside code brackets, they describe run-code operations. The value of argument 2 must therefore be a semantic description (see 5.3, 5.4).

3.2 ENTER COMMAND

The ENTER command is used to enter a new row in a table.

Example: ENTER { TABL, TEMP1, TEMP2, TEMP3 }

A new row is entered in the table TABL.

TEMP1 is stored in column 1, TEMP2 in column 2, etc.

Remarks: The first argument must be a table identifier. The remaining arguments must be arithmetic expressions. The number of arithmetic expressions can be less than the number of columns in the table.

SIGNAL is set to FALSE if the table is already filled. (The number of rows in a table can be specified by the compiler writer (see 4.4).

3.3 STORE: < argument 1 > → < argument 2 >

Permissible arguments

argument 1: primary.

argument 2: primary or Boolean cell.

Remarks: The store operation is non-destructive. See 5.3 for a discussion of the influence of code brackets on the interpretation of the arguments.

3.4 TALLY

TALLY < argument 1 >

TALLY { < argument 1 > , < argument 2 > }

Permissible arguments

argument 1: primary.

argument 2: arithmetic expression.

Description: Tally adds the value of argument 2 to argument 1. If the second argument is unspecified, it is taken as 1.

3.5 STORAGE OPERATIONS

LOAD < argument >

SAVLOAD < argument >

SAVTEMP < argument >

Permissible argument: Production operand (or operand whose value is a semantic description (see 5.4).

Description: The accumulator operations are used within code brackets only.

(a) LOAD < argument > , SAVLOAD < argument >

Run-code is compiled for a store operation from the operand described by < argument > to the run-code accumulator. The βINA bit is set in ACCUM (see 1.1). Further, in the case of SAVLOAD, code is compiled for a store of the previous contents of the run-code accumulator into a temporary register; a semantic description of this register is placed in the appropriate stack entry.

(b) SAVTEMP < argument > ("save in temporary location")

The argument is a semantic description of a run-code operand, which may be either an index register, ordinary memory register, or the run-code accumulator. Run-code is compiled for a store of this operand in a temporary location. The original semantic description is replaced by a semantic description of this location.

Control.- The following commands are used to implement transfers of control such as jumps, subroutine calls, error exits, or returns to the VITAL controller.

3.6 ASSIGN: < argument > ←

Permissible argument: flad or chain (see 1.6, 1.7).

Description: The flad or chain takes the value of CODELOC at the time the ASSIGN command is executed.

The assign command can be used inside code brackets only.

3.7 LABEL: < argument > ←

Permissible argument: any VITAL identifier (referred to below as a label).

Description: Labels are intended for use outside code brackets with the JUMP command (see 3.11).

A labelled statement may be located either before or after a JUMP to it.

3.8 SUBROUTINE NAME: < argument > ←

SUBROUTINE ENTRANCE NAME: < argument > ←

Permissible argument: any VITAL identifier (referred to below as name).

Description: The name is intended for use with the CALL command (see 3.12).

3.9 RETURN: RETURN

Permissible argument: none

Description: The RETURN statement must be the logical end of a subroutine.

3.10 EXIT: EXIT

Permissible argument: none

Description: EXIT terminates execution of a semantic routine in which it appears (see 5.1).

3.11 JUMP: JUMP < argument >

Permissible argument:

Outside code brackets: label (see 3.10).

Inside code brackets: primary whose value is a semantic description (see 5.4).

Description: A JUMP is an unconditional transfer of control. (See description of flads and chains, 1.6 and 1.7.)

3.12 CALL: CALL < argument >

Permissible argument: subroutine name or subroutine entrance name (see 3.8).

Description: The CALL transfers control to the subroutine or entrance and saves in MARK a pointer to a register containing the return address. Calls may be recursive (see 1.1).

3.13 FAULT: FAULT < argument >

Permissible argument: integer

Description: Control is returned to the controller, which informs the user of the fault number and awaits further commands. This command is intended for use when irrecoverable semantic states occur.

3.14 STOP: STOP

Permissible argument: none

Description: STOP returns control to the user via the VITAL controller. It can be used inside code brackets for logical termination of the run-code.

3.15 COMPLETE: COMPLETE

Permissible argument: none

Description: COMPLETE initiates system functions needed in order to terminate compilation. Control passes to the VITAL controller.

Conditional Statements

3.16 IF ... THEN: IF < argument 1 > THEN < argument 2 > ||

Permissible arguments:

argument 1: Boolean expression.

argument 2: sequence of statements (conditional or unconditional) separated by semicolons.

Description: The sequence of statements is executed or not executed, depending on whether argument 1 has value TRUE or FALSE, respectively.

3.17 IFNOT...THEN: IFNOT < argument 1 > THEN < argument 2> ||

Permissible arguments

argument 1: Boolean expression.

argument 2: sequence of statements (conditional or unconditional) separated by semicolons.

Description: The sequence of statements is executed or not executed, depending on whether argument 1 has value FALSE or TRUE, respectively. The IFNOT statement is generally more efficient than an IF statement with a negated Boolean expression.

3.18 IF...THEN...ELSE: IF < argument 1 > THEN < argument 2 > ELSE < argument 3 > ||

Permissible arguments

argument 1: Boolean expression.

argument 2 and 3: sequence of statements (conditional or unconditional) separated by commas.

Description: If argument 1 has value TRUE, argument 2 is executed; if it has value FALSE, argument 3 is executed.

4. DECLARATIONS

4.1 CELL DECLARATION: CELL < argument >

Permissible argument: list of cell identifiers separated by commas.

Description: The CELL declaration specifies single storage locations extant at compile time. The declaration must precede all other uses of the item being declared; the same is true of all other declarations.

4.2 INDEX DECLARATION: INDEX < argument >

Permissible argument: list of VITAL symbols separated by commas (and not beginning with a number).

Description: Identifiers declared as indices may be used by the compiler in the usual manner as subscripts and in arithmetic expressions. Indices are treated in the same way as integers but may lead to more efficient code in TALLY, subscripts, etc. Eight indexes are available.

4.3 RUN-INDEX DECLARATION: RINDEX < argument >

The details are similar to 4.2, except that the index is part of the run-code rather than the compiler.

4.4 TABLE DECLARATION

Example: TABLE SYMBOL 400 ID SEMANT EXTRA

Form: TABLE < argument 1 >< argument 2>< argument 3 >

Permissible arguments

argument 1 (name of the table): any VITAL identifier.

argument 2: integer (the number of rows in the table).

argument 3: list of VITAL identifiers separated by spaces (names of the columns).

Description: The TABLE declaration specifies tables that are used at compile time. More than one table may be declared by separating each complete table description with a comma.

4.5 STACK DECLARATION: STACK < argument >

Permissible argument: list of VITAL identifiers (stack names) separated by commas.

Description: The STACK declaration specifies push-down stacks used by the compiler. The depth of the stack is 100_8, unless some other depth is indicated by placing a digit after the identifier (the two being separated by a space).

4.6 RUN-STACK DELCARATION: RSTAK < argument >

The details are similar to 4.5, except that a run-stack is part of the run-code rather than the compiler.

4.7 DATA DECLARATION: DATA < argument >

Permissible argument: list of VITAL identifiers (data identifiers) separated by commas.

Description: Data identifiers are used to tag and test (see 1.16, 1.19) operands with bit configurations at compile time. Thirty-two is the maximum number of data and sub-data identifiers which can be declared (see 4.8).

At present, VITAL provides four common data identifiers, with no need for their explicit declaration: ΔBOOL, ΔINTGR, ΔFRACT, ΔREAL (see 5.4).

4.8 SUB-DATA DECLARATION

BDATA < argument >

IDATA < argument >

FDATA < argument >

RDATA < argument >

Permissible argument: list of VITAL identifiers (sub-data identifiers) separated by commas.

Description: The system-provided DATA identifiers ΔBOOL, ΔINTGR, ΔFRACT, ΔREAL (see 4.7) influence the type of run-code that is generated (see 5.4). The user may create subdivisions of these DATA identifiers by means of the declarations BDATA, IDATA, FDATA, RDATA, respectively. When a semantic description is used to determine the proper run-code to be generated (see 5.3, 5.4), a semantic description tagged with a BDATA identifier is treated as though it were tagged with ΔBOOL (analogously for IDATA, FDATA, and RDATA identifiers).

When run-code is compiled for a (unary or binary) operation, a semantic description of the result is generated (see 1.14). This semantic description (see 5.4) is either tagged with the data-type determined by the operation and operands; or, if this data-type coincides with the data types of the operands and these operands were originally tagged with a single sub-data identifier, then the semantic description is tagged with the sub-data identifier.

5. GENERAL FEATURES

5.1 PROGRAM STRUCTURE

An FSL program consists of a collection of declarations (see Section 4) followed by a collection of semantic routines. A semantic routine consists of the following items in succession:

① # followed by an integer (this integer is used by the productions in order to call the routine; (see Chapter 2, Section 3).

② space or tab

③ sequence of statements (conditional or unconditional) separated by semi-colons.

④ ‡ (Note: If a conditional statement is immediately followed by ‡, the symbol ‖ (see 3.17, 3.18, 3.19) need not be typed.

Each declaration must terminate with a semicolon. The first and last items in the program must be BEGIN and END respectively.

5.2 COMMENTS

A comment can be inserted at any point in the program. It must be preceded by two asterisks and followed by a carriage return.

5.3 CODE BRACKETS

The routines in an FSL program describe not only the operands and operators of the translator but also those in the run-code, i.e. the code being compiled. To distinguish between these two, the latter are enclosed within code brackets: ⊂ ... ⊃ ; corresponding run-code is generated when appropriate.

Most symbols can be used both inside and outside these delimiters, but with different effects. An example is helpful in explaining the situation.

Example: ⊂ L4 + L2 ⊃ → R2

The primaries L4, L2, R2 are variables with values, whether they appear inside or outside code brackets. However, their interpretation in connection with operators is different in these two cases. Outside code brackets, the primary itself is an operand (in the same sense that an address is an operand in machine language). Inside code brackets, it is assumed that the value of the primary is a semantic description (see 5.4). This description determines the operand in the run-code being generated. For example, in "L4 + L2" above, "+" refers to a run-code addition whose operands are described by L4 and L2. On the other hand, the arrow (→) outside the code brackets refers to a compiler store operation whose operands are R2 and ⊂ L4 + L2 ⊃ (both of which are primaries; see 1.2 and 1.14.

5.4 SEMANTIC DESCRIPTION

A semantic description is a 36-bit word containing information needed by the system when generating run-code. For example, this information

includes: (1) bits describing data types (real, integer, etc.), which influence the kind of run-code that is generated; (2) bits referring to a run-code address or index number, which determine the operands of the run-code. The contents of the first word in each semantic entry of the main stack is a semantic description. (However, the second word in each semantic entry has no pre-determined format; it is unused by the system, being intended to give the user ample space to store semantic information.)

The bit identifiers and their interpretations are given below.

Bits 1.1 – 2.8	If no β-bits are set, the address in 1.1 – 2.8 is taken as the run-code operand.
βDIR (direct)	Run-code operand should be a register containing the number in bits 1.1 – 2.8.
βNEG (negative)	Bits 1.1 – 2.8 refer to a run-code operand (i.e. address or index register number) the complement of whose contents is desired.
βINDX (index)	Bits 1.1 – 2.8 specify an index number.
βINA (in run-accumulator)	Operand is run-code accumulator.
βCON (constant)	Bits 1.1 – 2.8 contain a pointer to a system-created table containing all constants in the input text being compiled.
ΔBOOL (data type)	The data types are not single bits, but bit configurations (see 4.7). The run-code generated depends upon these data types, see also 4.8.

APPENDIX A

CONTROL META-COMMANDS*

1. LANGUAGES

A VITAL language consists of three types of elements: one set of productions, one set of semantics, and any number of programs. Each element consists of a symbolic directive (the source text) and one or more binary files (the results of compilation). All information for a language is maintained in the user's APEX directory.

Most of the VITAL meta commands are used to create, modify, and compile the various elements of a language. Before using these commands, the user must direct VITAL's attention to the desired language via the command LANG and then to the desired element of that language via one of the commands PROD, SEM, PROG. A language or a program under a language may be deleted via the DROP command; productions and semantics are deleted individually via the FRESH command.

COMMANDS

LANG LL

LANG switches VITAL's attention from the current language, if any, to the language LL. The current language is frozen in its current state. (If the current language is already LL, LANG LL freezes the current element of LL (productions, semantics, or a program) in its present state and removes that element from VITAL's attention.)

1. If LL is not a defined language name, or if both the productions directive and the semantics directive for LL are fresh (contain no text), VITAL will define LL as a language name, create fresh productions and semantics directives, and respond "FRESH LANG".

2. If LL is a defined language name and either the production's directive or the semantics directive contains text, VITAL will respond "OK".

* By James E. Curry, C. W. Adams Associates, Inc.

In either case, all subsequent commands until the next LANG command will refer to the language LL or one of its elements.

LANG has no effect on the state of an existing language; thus no harm is done if an existing language is accidentally referred to via LANG. If a new language is accidentally created with LANG, some superfluous items are created in the user's directory; these may (and should) be deleted by dropping the language via DROP.

LL must be an acceptable APEX name; it must begin with an alphabetic character and contain only alphanumeric characters (no periods).

The language name LL is entered in the user's directory as 9LL·; it is defined as a file group (map) which contains the names of all programs defined under LL. 9LL· is entered into the file group 9·VITAL.

The productions directive and the semantics directive for LL each consist of two files named 9LL·1D, 9LL·1L and 9LL·2D, 9LL·2L, respectively. These files are not entered into any file group.

PROD

The PROD command is legal whenever VITAL's attention is directed to a language LL or to one of its elements. PROD switches VITAL's attention to the productions of LL; the current element of LL under VITAL's attention, if any, is frozen in its present state. (If the productions of LL are currently under VITAL's attention, PROD has no effect.)

1. If the productions directive is fresh (contains no text, VITAL will respond "FRESH".

2. If the productions directive contains text, VITAL will respond "OL".

In either case, all subsequent commands until the next LANG, SEM, or PROG command will reference the productions of the language LL.

PROD may not have an argument; since there is only one set of productions for each language, it is not given a name.

PROD has no effect on the state of a language or on the user's directory; if a set of productions is accidentally referenced via a PROD command, no harm is done.

Note that the production of a language cannot be dropped via DROP except by dropping the language itself. Use FRESH command to delete productions.

SEM

The SEM command is legal whenever VITAL's attention is directed to a language LL or to one of its elements. SEM switches VITAL's attention to the semantics of LL; the current element of LL under VITAL's attention, if any, is frozen in its present state. (If the semantics of LL are currently under VITAL's attention, SEM has no effect.)

1. If the semantics directive is fresh (contains no text), VITAL will respond "FRESH".
2. If the semantics directive contains text, VITAL will respond "OK".

In either case, all subsequent commands until the next LANG, PROD, or PROG command will reference the semantics of the language LL.

SEM may not have an argument; since there is only one set of semantics for each language, it is not given a name.

SEM has no effect on the state of a language or on the user's directory; if a set of semantics is accidentally referenced via a SEM command, no harm is done.

Note that the semantics of a language cannot be dropped via DROP except by dropping the language itself. Use the FRESH command to delete semantics.

PROG PP

The PROG command is legal whenever VITAL's attention is directed to a language LL or to one of its elements, and if the productions of LL have been compiled. (See note below.) PROG PP switches VITAL's attention to

the program PP under LL; the current element of LL under VITAL's attention, if any, is frozen in its present state. (If the program PP is currently under VITAL's attention, PROG PP has no effect.)

1. If PP is not a defined program name under LL, or if the directive for PP if fresh (contains no text), VITAL will define PP as a program name under LL, create a fresh directive for PP, and respond "FRESH".

2. If PP is a defined program name under LL and its directive contains text, VITAL will respond "OK".

In either case, all subsequent commands until the next LANG, PROD, SEM, or PROG command will reference the program PP.

PROG has no effect on the state of an existing program; thus no harm is done if an existing program is accidentally referenced via PROG. If a new program name is accidentally created with PROG, some superfluous items are created in the user's directory; these may (and should) be delted by dropping the program with DROP.

PP must be an acceptable APEX name; it must begin with an alphabetic character and contain only alphanumeric characters (no periods).

The program name PP under the language LL is entered in the user's directory as 9LL·PP·; it is defined as a file group (which currently remains empty). 9LL·PP· is entered in the file group 9LL· .

The directive for PP consists of two files named 9LL·PP·3D and 9LL·PP·3L. These files are not entered into any file group.

Note: Since the productions of LL specify the chopping rules for all programs written in LL, the productions must be compiled before any program may be defined under LL. (A program may not be compiled unless the semantics of LL have also been compiled.) If the binary production files become undefined at any point, (this occurs for FRESH and READ on productions), the existing programs of LL remain defined but may not be referenced with PROG until the binary production files are redefined via compilation. (If the

chopping rules for the language are changed in the process, all existing program directives must be reconverted (see REC).)

DROP

The DROP command is legal whenever VITAL's attention is directed to a program PP under some language, or whenever VITAL's attention is directed to a language LL and not to its productions to semantics (i.e. after a LANG command but before any PROD or SEM command). The effect of DROP is to undefine the program PP or the language LL, respectively.

1. If the program PP under the language LL is currently under VITAL's attention, VITAL will ask "DROP PROG?" and await a response. If the user strikes the NO key, the DROP command will be cancelled. If the user strikes the YES key, VITAL will undefine the program name, the directive, and all binary files for PP. VITAL's attention will then be directed to the language LL but to no particular element of LL.

2. If no element of LL is currently under VITAL's attention, VITAL will ask "DROP LANG?" and await a response. If the user strikes the NO key, the DROP command will be cancelled. If the user strikes the YES key, the language name and the directives and binary files for all the elements of the language LL will be undefined. VITAL will respond "CLEAN"; no language will be under VITAL's attention. DROP may not have an argument; it always references the current language or program.

Note that DROP cannot be used to undefine the productions or semantics of a language, since these are essential to the definition of the language. The FRESH command may be used to delete existing text from the productions or semantics (as well as a program) of a language.

2. DIRECTIVES — DIRECTIVE EDITING

DIRECTIVES

Each element of a VITAL language (productions, semantics, and each program) has an associated directive. Each directive consists of two

files. One file, the dictionary, contains the character strings for all words which appear in the source text; this file provides a means of associating a 13-bit internal symbol with each word. The other file, the text file, contains the source text itself; each word is represented by its internal dictionary symbol.

DIRECTIVE EDITING

The VITAL editing meta commands are legal whenever VITAL's attention is directed to some element (productions, semantics, or a program) of some language.

The editing commands accept arguments similar to MK5's; the exceptions are: directive lines are referenced differently; the MOVE destination is separated by "←" rather than by tab; text may not be typed after the editing argument, even if only one line is to be typed.

Directive lines are referenced by their first word. If the word used to reference a line occurs only once as the first word on a directive line, that line is referenced. If the word occurs more than once, the first occurrence is the line referenced. If, however, the word is followed by "(k", k a positive octal integer, the k^{th} occurrence is the referenced line.

A directive line may also be referenced as the n^{th} line above or below a specified line by following the identifying word (or the "(k") with "+n" or "-n", n a non-negative octal integer. (The algebraic sum is taken if more than one "±n" is typed.)

There are two restrictions on referencing directive lines:

1. If a directive line begins with a space or a tab, that line may not be referenced by the first word; the "±n" facility must be used.

2. If the first word on a directive line contains a non-alphanumeric character and is more than one character in length, that word may not be used to reference the line; the "±n" facility must be used. (This inconvenience will be removed soon.) The legal forms of editing arguments are:

a. arg 1 (arg 1 ← arg 2 for MOVE)

b. arg 1 | n (arg 1 | n ← arg 2 for MOVE)

c. arg 1 → arg 2 (arg → arg 2 ← arg 3 for MOVE)

where arg 1, arg 2 and arg 3 are directive line references as described above. Type 1 specifies a single line; type 2 specifies n lines beginning with "arg 1"; and type 3 specifies the lines from "arg 1" up to but not including "arg 2". (The argument after the "←" may be present only for MOVE; it specifies the destination of the block to be moved.)

The character "#" is a special argument which specifies the first line of the directive if it occurs as "arg 1" and the (blank) line after the last line of the directive if it occurs as "arg 2" or "arg 3".

Spaces and tabs may be used indiscriminately in editing arguments; they are always ignored except as word separators.

If the word used in the line specification for "arg n" occurs more than once as the first word on a directive line, the message "ARGn kx" will be printed; "k" is the number of times the word occurs. (This is not considered an error.)

The following errors are possible on editing commands:

1. Bad format. (VITAL type "NO".)

   This error occurs when the argument is formatted improperly; e.g. multiple line specification for an INS, no "←" argument for a MOVE, illegal argument separator, etc. This error also occurs when a word is chopped into two words because of a non-alphanumeric character contained in the word.

2. Bad argument. (VITAL types "ARG n NG").

   This error occurs when a word is not found at the beginning of some line in the directive; when the k^{th} occurrence of a word is specified and there are fewer than k occurrences; when a "± n" specification specifies a line not in the directive.

3. Bad line block. (VITAL types "ARG → NG").

   This error occurs when an "arg 1 → arg 2" argument is specified if "arg 2" specifies a line above that specified by "arg 1".

4. Bad MOVE destination. (VITAL types "ARG ← NG").

   This error occurs when the line specified after the "←" is within the line block which is being moved.

Thus it is impossible to clobber a directive by faulty line block specifications.

The INS and REP commands expect text to follow. All lines typed after an INS or REP command are processed as text until the text is terminated by another meta command (a line beginning with red "←←" is typed), or by the YES or NO function keys. If the text is terminated by a red "←←" line or by the YES key, VITAL will accept the text and complete the INS or REP command. If the text is terminated by the NO key, VITAL will ask "REJ?" and await a response. If the response is the YES key, VITAL will reject the text that has been typed and cancel the INS or REP command. If the response is the NO key, VITAL will accept the text and complete the INS or REP command.

If the READIN key is struck while text is being accepted, VITAL will read paper tape and accept text from the tape. In this case, the INS or REP command is terminated by a meta command line from the tape or by the end of the tape.

If an INS or REP command is given from paper tape, the text is obviously taken from the tape also. In this case the command is terminated by the next meta command line or by the end of the tape.

If a function key other than YES, NO, or READIN is seen while text is being typed, VITAL will type "IL CHAR" and ignore the illegal function key.

If a chopping error is detected in a line of text, VITAL will type "CHOPPER" followed by the text on the line separated into words by vertical bars. The line is accepted; if it is incorrectly chopped, it may be replaced or edited later.

INS

The INS command accepts only a single line specification. The text typed after the command will be inserted before the line specified. (Text is terminated as described above.)

The command INS# is used to insert text into a fresh direction.

REP

The REP command accepts a single line or line block specification. The text typed after the command replaces the line or line block specified. (Text is terminated as described above.)

DEL

The DEL command accepts a single line or line block specification. The line or line block specified is deleted from the directive. No text is accepted after DEL.

MOVE

The MOVE command accepts a single line or line block specification followed by "←" and a single line specification. The first line or line block is moved to just before the line specified after the "←". No text is accepted after MOVE.

EDIT

The EDIT command accepts a single line or line block specification. If the line block is longer than 30_8 lines, the EDIT command will be rejected; VITAL types "NO". The line or line block specified is displayed on the user's scope with a marker box on the first character; "EDIT" is typed, and VITAL waits for the user to edit the displayed text. (Do not type anything before the "EDIT" message is typed, even though the text may be displayed before

the message appears). Editing is done as follows:

Moving the Marker

YES	Move marker one position to right.
NO	Move marker one position to left.
BEGIN	Move marker one word to right. A "word" is an alphanumeric.
READIN	Move marker one word to left. Character string.
LINE FEED UP	Move marker to beginning of line, or if at beginning of line, to beginning of previous line.
LINE FEED DOWN	Move marker to beginning of next line.

Deleting Text

DELETE	Deletes character in marker.
WORD-EXAM	Delete character in marker and rest of word (alpha-numeric character string).
STOP	Delete rest of line, excluding carriage return.

Other characters typed are inserted into the text to the left of the character in the marker; the marker remains on the original character.

Editing is terminated by striking the RED key. VITAL types "ACC?" and waits for a response. If the YES key is struck, VITAL types "OK" and replaces the original line or line block with the text appearing on the scope. If the NO key is struck, VITAL types "REJ?" and waits for another response. If the YES key is struck this time, VITAL will cancel the original EDIT command and type OK. If the NO key is struck again, VITAL will type "EDIT" again and return to editing mode.

Do not use the HELP button to reject the text.

OTHER DIRECTIVE COMMANDS

The FRESH, REC, WRITE, and READ meta commands are legal whenever VITAL's attention is directed to some element (productions, semantics, or a program) of some language.

FRESH

The FRESH command initializes the current language element. All existing binary files for that element are undefined, and the directive is made fresh.

REC

The REC command performs the following actions: the text in the current directive is saved in character form, the directive is made fresh, and the saved text is inserted into the directive. Thus, the effect of REC is to re-chop the directive text. (REC does not undefine the existing binary files.)

There are two possible reasons for using the REC command.

1. If the chopping rules for a language are changed (e.g. a terminator is changed to a non-terminator, an ignored terminator is changed to a visible terminator), any existing program directives should be REC'd in order to re-chop them with the new chopping rules. Similarly, if it becomes necessary for the system designers to change the chopping rules for the production language or the semantics language, all productions and semantics directives must be REC'd.

2. The editing meta commands currently make no attempt to re-use the space vacated by a deleted line block; thus as a directive is edited, it grows in size. The REC command shrinks the directive back to minimal size.

WRITE FF

The WRITE command accepts as its argument a legal APEX name FF; the name must begin with an alphabetic character and contain only alphanumeric characters. WRITE creates an APEX file named FF and stores the text of the current directive into this file. The name FF may now be given as the argument to a READ command.

FF becomes a file with no directory origin. The characters are stored four per word from right to left beginning at the fourth register of the file. (See the note following the READ description.)

READ FF

The READ command accepts as its argument an APEX file name FF; the name must begin with an alphabetic character and contain only alphanumeric characters. FF must have the format of VITAL character file (i.e. created by WRITE). READ makes the current language element fresh (binary files are undefined, and the directive is made fresh), and inserts the found in FF into the fresh directive.

NOTE: The READ and WRITE commands are intended to give the user a way of transferring directives from one APEX name to another or from one language to another under the same name.

To transfer a directive from one language to another under the same APEX name, simply WRITE the directive from the proper element of the first language into the file FF, switch VITAL's attention to the same element of the second language, and READ FF.

To transfer a directive from one APEX name to another:

1. WRITE the directive into FF.
2. Exit VITAL via HELP or SCRATCH.
3. To the BT:

   5WHAT FF (note length of file)

   5WRM4F FF FRE 500 (or other MK4 area)

   (If FF is n pages long, it is 400n registers long and occupies 2n drum tracks.)
4. Log out (or abort) and log in under the second APEX name.
5. To the BT:

   5DFIL FRE 500 0 400n (FF)
6. Under VITAL, do a READ FF for the proper language element.

(This inconvenient procedure will become obsolete when READ and WRITE are implemented for MK4 drum areas.)

3. GENERAL

INPUT TO VITAL

Function keys (READIN, BEGIN, YES, NO, WORD EXAM, LINE FEED DOWN, and LINE FEED UP) are special keys in VITAL. They always terminate the current line just as carriage return does. This fact is insignificant if the function key is typed alone on a line; but if characters are typed before the function key is struck, VITAL will act as if a carriage return was struck before the function key.

A line which has been terminated with carriage return or a function key cannot be recalled. The DELETE key deletes the preceding character on the current line; if DELETE is struck n times in succession, the last n characters on the line are deleted. ("Characters" means single LW codes, not compound characters.) The STOP key (not the NO key) deletes the current line.

No distinction is made between red and black characters (except for red '↵').

The READIN key signals VITAL to read paper tape. The text on a paper tape is processed somewhat differently from keyboard text; the differences are explained below. Also, function keys on a paper tape are ignored. The entire tape is read in before any characters are processed; if the abort button is pushed while the tape is being read in, VITAL will type 'ABORTED' and return to keyboard input mode without processing the tape.

META COMMANDS IN GENERAL

When first entered (from the BT), VITAL types '↵' and awaits a meta command. If the command is one which is not followed by lines of text (unlike INS, REP), VITAL will process the command (possible soliciting a response from the user) and eventually type '↵' again. If the command

is one which expects text, VITAL must be told when to return to command mode; this is done by striking the YES key or the NO key (to accept or reject the text typed, or by following the last line of text with a line beginning with two red hands. In the former cases, VITAL will type '↵↵' and await another meta command; in the latter case, the line beginning with the red hands will be processed as the next meta command. The intention of this method of processing is to allow VITAL to maintain closer control over the user's actions; observations on the relative (aesthetic or practical) merits of the M5 and VITAL meta command philosophy are solicited.

RULES FOR META COMMANDS

A meta command will be rejected if the underlined portion of the command word (see meta command list) is not typed completely, or if the first four letters of the command word are typed incorrectly. (E.g. LI, LIS, LIST, LISTZZ will all be accepted as LIST; but L, LIZ will be rejected.)

Meta commands may be separated from arguments by any number of spaces and/or tabs; arguments which are single upper case characters do not require a separator. (E.g. BIN*, INS#.) (See the description of the editing commands for possible exceptions.)

Meta command lines are always checked for format. The last significant word on a line must be followed by carriage return or a function key (possibly with intervening spaces and/or tabs). That is, commands for which an argument is meaningless may not have one; and commands which require a single argument may have only one.

Whenever VITAL rejects a meta command line typed on the keyboard, a message to that effect will be given, and '↵↵' will be typed. All typing done before the '↵↵' is typed will be ignored. If there is doubt as to whether the last character typed was ignored, the STOP key guarantees that it will be.

Whenever an error is detected in a meta command line from paper tape, the line in error will be typed out before the error message. The incorrect

line will be ignored, '↩↩' will be typed, and input will be temporarily switched to keyboard mode. The user may then type as many lines as he wishes; when he wishes to resume processing the tape (at the line following the line in error), he strikes the READIN key again. (The tape will not be read again; no 'PETR FREE, PUSH GO' message will appear.)

All meta commands on paper tape must be preceded by two red hands. If VITAL expects a meta command (as it does when READIN was struck when VITAL expected a meta command, or when VITAL finishes processing a command not followed by text) and does not see red hands on the next line of the tape, it will type a message to that effect and treat the situation as it does other errors.

If a meta command from paper tape requires a response from the user, it is to be typed on the keyboard. VITAL will switch input mode to keyboard when necessary.

If a meta command is rejected by VITAL, nothing is changed by the rejected command.

META COMMANDS DESCRIPTION

HELP	MOVE arg
SCRATCH	EDIT arg
QUIT	LIST (arg)
LANG name	TYPE (arg)
PROD	FRESH
SEM	REC
PROG name or COMP name	BIN (mode number*)
DROP	DICT
INS arg	READ name
DEL arg	WRITE name
REP arg	KEEP name

* 1000 = no semantics
100 = scan listing
10 = type errors
1 = stack state (if errors)

Note: To obtain a combination of the above modes, simply unit the corresponding binary numbers.

STANDARD CHOP TABLE

The following table lists the Lincoln Writer codes and their corresponding chopper classification (see Chapter 2, 2.1).

0	0	n*	40	Q	n	100	⌐	t	140	α	t
1	1	↓	41	R	↓	101	Σ	↓	141	Δ	↓
2	2		42	S		102	|		142	ρ	
3	3		43	T		103	‖		143	ε	
4	4		44	U		104	/		144	h	
5	5		45	V		105	x		145	⊃	
6	6		46	W		106	#		146	β	
7	7		47	X		107	→		147	∧	
10	8		50	Y		110	<		150	λ	
11	9	n	51	Z	n	111	>	t	151	~	
12	—		52	(	t*	112	¯		152	{	
13	O		53	)	t	113	□		153	}	
14	RI		54	+	t	114	RI		154	≡	
15	BN		55	-	t	115	BN		155	≠	
16	NO		56	,	t	116	NO		156	'	
17	YES		57	.	t	117	YES		157	*	t
20	A	n	60	CR	ic*	120	n	t			
21	B	↓	61	TAB	ic	121	⊂	↓			
22	C		62	BSP		122	∨				
23	D		63	BLK		123	q				
24	E		64	SUP	t	124	γ				
25	F		65	NOR	t	125	t				
26	G		66	SUB	t	126	w				
27	H		67	RED		127	x				
30	I		70	SP	it**	130	i				
31	J		71	WX		131	y				
32	K		72	LFD		132	z				
33	L		73	LFU		133	?				
34	M		74	---		134	U				
35	N		75	---		135	∩				
36	O		76	STP		136	j				
37	P	↓	77	DEL		137	k	t			

* n = normal; t = terminate (visible); it = ignored terminator; ic = ignored control character.

** SPACE may only be declared VTERM or ITERM (see Chapter 2, 2.1).

APPENDIX B

PRODUCTION LANGUAGE GUIDE

This appendix is intended to help the reader familiarize himself with the use of Production Language. Productions are developed for the compilation of any expression satisfying the following Backus Normal Form specification:

< expression > :: = < term > | - < term > | < expression > - < term >

< term > :: = < primary > | < term > * < primary >

< primary > :: = < identifier > | (< expression >)

Identifiers are the basic symbolic units (see Chapter 2, 2.1).

For example, (- A *B - C) is such an expression.

The writing of productions need not be based on a BNF-specified syntax. With a little experience, it is quite natural to think directly in terms of Production Language. BNF is used here only as a convenience in introducing PL.

1. Writing the above BNF description in production-like form yields:

E_1||
```
    T → E
   -T → E
  E-T → E
```
T_1||
```
    P → T
  T*P → T
```
P_1||
```
    I → P
  (E)→ P
```

P, T, and E would be declared as internal symbols in the final PL program; see Chapter 2, 2.2. The symbols *, -,), and (would be declared as reserved words; see Chapter 2, 2.1.

2. Since simpler constructs must be recognized before more complex ones, the productions must be re-ordered P_1, T_1, E_1. Also where one sequence of symbols resembles the end of another, the longer sequence must be tested first if misidentification is to be avoided (consider E_1 above).

The result of this re-ordering is:

P_1|| I → P

(E) → P

T_1|| T *P → T

P → T

E_1|| E -T → E

-T → E

T → E

3. Flow of execution is rarely sequential; when a match is found and the production completed, a jump must usually be made. The jump may be unconditional, or it may depend upon the next symbol scanned. In the latter case, productions are needed to determine proper branching.

Whenever the stack is altered or a construct is recognized, an FSL routine must generally be executed to update the semantic column of the stack, modify storage, or generate code.

The productions must begin in such a manner that the first symbols scanned will be properly recognized. Also, productions should be included for detecting syntax errors, i.e. undesired formation of the input.

The following productions are the result of the above considerations and the heuristic guidance of the question, "What symbol can come next?"

B_1||	-	SCAN	⌐P_1
P_1||	(	SCAN	⌐B_1
	I → P	EXEC 1	⌐T_1

P_2 |	(E) → P	EXEC 2	← T_1
	SG	ERROR 1	
T_1 |	T * P → T	EXEC 3	SCAN ← TBR
	P → T		SCAN ← TBR
TBR |	T *		SCAN ← P_1
E_1 |	E-T SG → E SG	EXEC 4	← EBR
	- T SG → E SG	EXEC 5	← EBR
	T SG → E SG		← EBR
EBR |	E)		← P_2
	E -		SCAN ← P_1
	SG	ERROR 2	

The numbering of the EXECs and ERRORs is arbitrary.

Below is a trace of the productions executed when the formula

(-A *B -C)

is processed.

Each part of an executed production is written on a separate line; if the stack has been modified, the new version appears on the right.

Label	Production	Stack
		([initial state]
P_1	(	
	SCAN	(-

(cont)

Label	Production	Stack
	☛ B_1	
B_1	-	
	SCAN	(- I
	☛ P_1	
$P_1 + 1$	I	
	P	(- P
	EXEC 1	
	☛ T_1	
$T_1 + 1$	P	
	T	(- T
	SCAN	(- T *
	☛ TBR	
TBR	T *	
	SCAN	(- T * I
	☛ P_1	
$P_1 + 1$	I	
	P	(- T * P
	EXEC 1	
	☛ T_1	
T_1	T * P	
	T	(- T
	EXEC 3	

Label	Production	Stack
	SCAN	(- T -
	← TBR	
$E_1 + 1$	- T SG	
	E SG	(E -
	EXEC 5	
	← EBR	
EBR + 1	E -	
	SCAN	(E - I
	← P_1	
$P_1 + 1$	I	
	P	(E - P
	EXEC 1	
	← T_1	
$T_1 + 1$	P	
	T	(E - T
	SCAN	(E - T)
	← TBR	
E_1	E - T SG	
	E SG	(E)
	EXEC 4	
	← EBR	
EBR	E)	
	← P_2	

Label	Production	Stack
P_2	(E)	
	P	P
	EXEC 2	
	$\neg T_1$	
$T_1 + 1$	P	
	T	T
	SCAN	end of example.

The successive states of the stack consitute a parse of the expression. This succession of states can be readily represented by the following tree-like structure. The first character scanned is the lowest in the tree. Successive horizontal levels correspond to successive stack states. (See Figure 2, next page.)

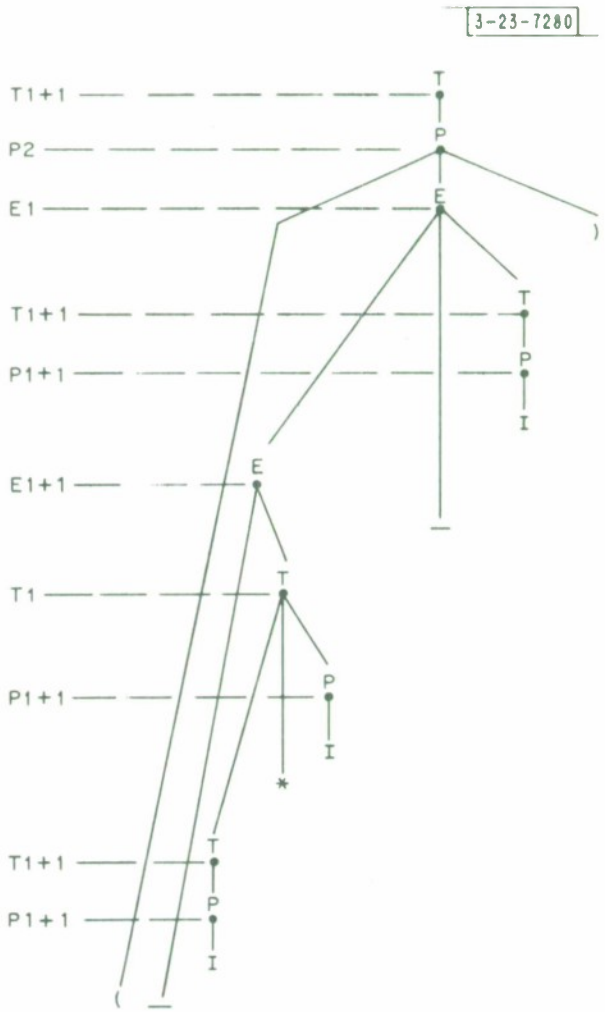

FIGURE 2

Figure 2 may be drawn in more conventional form to clearly exhibit the parsed structure of the example:

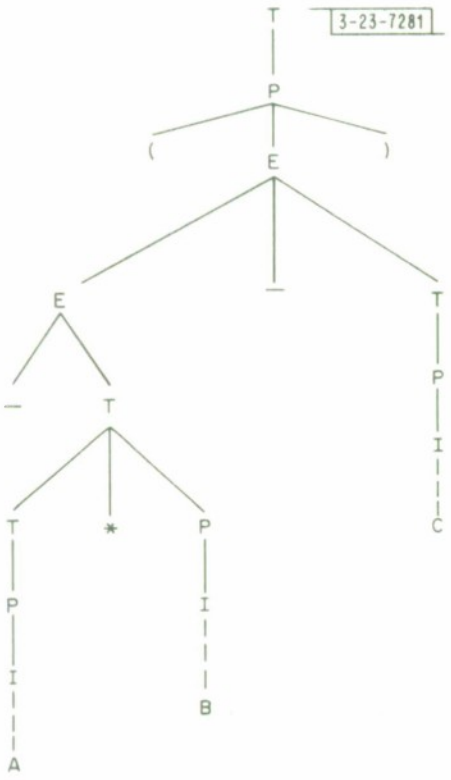

FIGURE 3

APPENDIX C

ALGOL

Included for reference are the productions and semantics of ALGOL as implemented in VITAL. Readers of this section may find Appendix D of assistance.

```
TUE 06 DEC 66 1646.08  JAF
RES ⊅HAND ⊅EQ ⊅SUP ⊅SUB ⊅NOR ⊅AROW ⊅DBAR
RES BEGIN ⫷END COMMENT PROCEDURE ⫷RETURN
RES ARRAY BOOLEAN INTEGER FIXED REAL LABEL
RES ∨ ∧ ¬ SAVE RESTORE
RES + - ≠ ≯ < > ≤ ≥ ∧ ∨ ⊃ × / ~
RES . , ELSE IF THEN GOTO GO TO THRU FOR THROUGH WHILE UNTIL
RES { } ( ) REF REFERENCE
RES STEP DO
INT ∤ PDEC ARID UN PID P BASE E T FORC FORW
INT BP BT CEXC ICL S PCALL Δ PA
⫷STP ≡ BOOLEAN 10 INTEGER 11 FIXED 12 REAL 13 LABEL 33
⫷IDL ≡ BOOLEAN 10 INTEGER 11 FIXED 12 REAL 13 ARID 14 PID 15 SAVE 35 LABEL 33
⫷UNOPE ≡ ∨ 53 ∧ 54 ⊅DBAR 55 ¬ 56 - 57 + 31 ~ 72
⫷NXT = + - ⊅EQ ≠ ≯ < > ≤ ≥ ∧ ∨ ⊅AROW
⫷TD ≡ × 60 / 61
⫷PM ≡ + 62 - 63
⫷RL ≡ ⊅EQ 64 ≠ 65 ≯ 65 > 66 < 67 ≥ 70 ≤ 71
⫷END = , ⫷END ELSE
⫷START = IF FOR ( - ~ ∨ ∧ ¬ ⊅DBAR +
⫷REFR = REF REFERENCE
⫷BOOL = ∧ ∨ THEN ⊅AROW ~
⫷TOTH ≡ TO 116 THRU 117 THROUGH 117
⫷TOTH1 ≡ TO 125 THRU 115 THROUGH 115
⫷WHUN ≡ WHILE 121 UNTIL 122
```

Bo‖	BEGIN→	∤ BEGIN	EXEC 1	SCAN	⊅DS
	SG		ERROR 0		⊅Q
DS‖	⫷STP			SCAN 2	⊅D1
	ARRAY →	REAL ARRAY		SCAN	⊅D1
	PROCEDURE			SCAN	⊅D1
	SG				⊅S1
DS1‖	⫷STP			SCAN 2	⊅D1
	ARRAY →	REAL ARRAY		SCAN	⊅D1
	PROCEDURE			SCAN	⊅D1
	SG		ERROR 10		⊅Q
D1‖	⫷STP I SG		EXEC 16		⊅IDL
	PDEC ⫷STP ARRAY I →	PDEC ARID I	EXEC ⫷STP	SCAN	⊅IDL
	PDEC ⫷STP PROCEDURE I →	PDEC PID I	EXEC ⫷STP	SCAN	⊅IDL
	⫷STP ARRAY I →		EXEC ⫷STP EXEC 20	SCAN	⊅DA1
	⫷STP PROCEDURE I →	PDEC	EXEC ⫷STP EXEC 23	SCAN	⊅DP1
	PROCEDURE I →	PDEC	EXEC 24	SCAN	⊅DP1
	SG		EXEC 26		⊅S0
DA1‖	{ →			SCAN 4	⊅DA2
	SG		ERROR 13		⊅Q
DA2‖	I - I , →		EXEC 21	SCAN 4	⊅DA2
	I - I } →		EXEC 22	SCAN	⊅DA3

```
TUE 06 DEC 66 1646.08  JAF
                SG
DA3‖            ; →
                SG
DP1‖            PDEC { →                PDEC
                PDEC ; →                PDEC
                SG
IDL‖            ‹IDL I . →              ‹IDL
                PDEC ‹IDL I . →         PDEC
                ‹IDL I ; →
                PDEC ‹IDL I } →         PDEC
                SAVE I }→               UN
                SG
REFF‖           'REFR →
                ; →
                SG
RE1‖            I . →
                I ; →
                SG
S0‖             SG
S1‖             1
                ‹START
                ‹END
                BEGIN
                GO
                GOTO →          GO
                ‹WHUN
                SAVE
                RESTORE →       UN
                COMMENT →
                'RETURN
                SG
RTN‖            'RETURN ‹END →  UN ‹END
                SG
SID‖            I 'HAND →
                1 SG
CM‖             ‹END
                SG →
G1‖             GO TO →         GO
                GO I →          UN
                GO SG
SV1‖            SAVE { →        SAVE
                SG
EX0‖            SG
EX1‖            1
                (
```

```
ERROR 14                                'Q
EXEC 17         SCAN                    'DS
EXEC 17                                 'DS
EXEC 4          SCAN                    'DS
EXEC 27                                 'S0
ERROR 2                                 'Q
EXEC ‹IDL       SCAN 2                  'IDL
EXEC ‹IDL EXEC 17   SCAN                'DS1
EXEC ‹IDL EXEC 17   SCAN                'DS
EXEC ‹IDL       SCAN                    'REFF
EXEC 36         SCAN                    'S9
ERROR 3                                 'Q
                SCAN 2                  'RE1
                SCAN                    'REFF
EXEC 26                                 'S1
EXEC 25         SCAN 2                  'RE1
EXEC 25 EXEC 26 SCAN                    'DS
ERROR 11                                'Q
                SCAN                    'S1
                SCAN                    'SID
                                        'EX0
                                        'S9
EXEC 2          SCAN                    'DS
                SCAN                    'G1
                SCAN                    'G1
EXEC 107                                'EX0
                SCAN                    'SV1
EXEC 37         SCAN                    'S9
                SCAN                    'CM
                SCAN                    'RTN
ERROR 4                                 'Q
EXEC 5                                  'S9
                                        'EX1
EXEC 50                                 'S0
                                        'ID
                                        'S9
                SCAN                    'CM
                SCAN                    'G1
EXEC 51         SCAN                    'S9
                                        'EX1
                SCAN 2                  'IDL
ERROR 5                                 'Q
                SCAN                    'EX1
                SCAN                    'ID
                                        'EX0
```

```
TUE 06 DEC 66 1646.08  JAF
            ‹UNOPE                                                      ▾EX0
            SG                                   ERROR 6                ▾Q
ID‖         I { →            PID {                          SCAN        ▾PP
            I ▸SUB →         ARID                EXEC 30                ▾EX0
P2‖         P ▸SUB →         BASE                EXEC 4                 ▾EX0
            PID I , →        PID PA Δ            EXEC 41                ▾EX0
            PID I } →        PID PA SG           EXEC 41 EXEC 45        ▾PARG
            Δ I , →          Δ PA Δ              EXEC 41                ▾EX0
            Δ I } →          Δ PA SG             EXEC 41 EXEC 45        ▾PARG
            I SG →           P SG                EXEC 40                ▾P2
            E ‹PM P SG →     E ‹PM T SG                                 ▾T2
            ‹UNOPE P SG →    P SG                EXEC ‹UNOPE            ▾P2
            ▸ARDW P SG                                                  ▾STR
T1‖         T ‹TD P SG →     T SG                EXEC ‹TD               ▾T2
            P SG →           T SG                                       ▾T2
T2‖         T ‹TD                                                       ▾EX0
E1‖         E ‹PM T SG →     E SG                EXEC ‹PM               ▾E2
            T SG →           E SG                                       ▾E2
E2‖         E ‹PM                                                       ▾EX0
            ( E ) →          P                   EXEC 4     SCAN        ▾P2
            E ‹RL E SG →     BP SG               EXEC ‹RL               ▾BP2
            ‹BOOL E ‹NXT →   ‹BOOL BP ‹NXT                              ▾BP2
            E ‹NXT                                                      ▾EX0
            ARID E , →       ARID                EXEC 76                ▾EX0
            ARID E ▸NOR →    P                   EXEC 77    SCAN        ▾P2
            BASE E ▸NOR →    P                   EXEC 75    SCAN        ▾P2
            STEP E ‹TDTH1 →  ‹TDTH1              EXEC ‹TDTH1            ▾EX0
            ‹TDTH E DO →     FORC                EXEC ‹TDTH             ▾S0
            E SG →           BP SG                                      ▾BP2
BP2‖        ¬ BP SG →        BP SG               EXEC 72                ▾BP2
BT1‖        BT ∧ BP SG →     BT SG               EXEC 73                ▾BT2
            BP SG →          BT SG                                      ▾BT2
BT2‖        BT ∧                                                        ▾EX0
BE1‖        E ∨ BT SG →      E SG                EXEC 74                ▾BE2
            BT SG→           E SG                                       ▾BE2
BE2‖        GD E ⊃ →         GD E                           SCAN 3      ▾CGT
            E ⊃ →            CEXC                EXEC 4                 ▾EX0
            E ▸ARDW                                         SCAN 2      ▾ID
            ( E ) →          BP                  EXEC 4     SCAN        ▾BP2
            IF E THEN →      ICL                 EXEC 110               ▾S0
            PID E , →        PID PA Δ            EXEC 42                ▾EX0
            PID E } →        PID PA SG           EXEC 42 EXEC 45        ▾PARG
            Δ E , →          Δ PA Δ              EXEC 42                ▾EX0
            Δ E } →          Δ PA SG             EXEC 42 EXEC 45        ▾PARG
            CEXC E , →       CEXC E                                     ▾EX0
```

```
TUE 06 DEC 66 1646·08  JAF
                CEXC E E SG →   E SG
                E ∨
                ‹RETURN E SG →  UN SG
                STEP E ‹WHUN →  ‹WHUN
                ‹WHUN E DO →    FORC
                SG
CGT‖            GO E I , I →    UN
                SG
PARG‖           SG →
PARG1‖          ∆ PA →
                PID PA →        PCALL
                SG
PC‖             PCALL ‹END →    UN ‹END
                PCALL SG →      P SG
PP‖             PID { } →       PCALL
                PID { SG →      PID SG
STR‖            FOR E ›AROW P STEP →   STEP
                FOR E ›AROW P ‹WHUN →  ‹WHUN
                E ›AROW P ‹END →   UN ‹END
                E ›AROW P SG →  P SG
S9‖             ICL UN ELSE
                ICL UN ‹END →   S ‹END
                UN SG →         S SG
S8‖             BEGIN S ; →     BEGIN
                PDEC S ; →
                BEGIN S ‹END →  UN
                BEGIN ‹END →    UN
                ICL UN ELSE S ‹END →   S ‹END
                FORC S ‹END →   UN ‹END
                SG
S6‖             ∤ UN →
                SG
Q‖              SG
END
```

```
EXEC 100                        ›E2
                                ›EX0
EXEC 34                         ›S9
EXEC 120                        ›EX0
EXEC ‹WHUN                      ›S0
ERROR 7                         ›Q
EXEC 52         SCAN            ›S9
ERROR 15                        ›Q
                                ›PARG1
EXEC 46                         ›PARG1
EXEC 46 EXEC 47   SCAN          ›PC
ERROR 100                       ›Q
EXEC 103                        ›S9
EXEC 104                        ›P2
EXEC 44         SCAN            ›PC
EXEC 43                         ›EX1
EXEC 114                        ›EX0
EXEC 124                        ›EX0
EXEC 101                        ›S9
EXEC 102                        ›P2
EXEC 111                        ›S0
EXEC 112                        ›S8
                                ›S8
                                ›S0
EXEC 3          SCAN            ›DS
EXEC 6                          ›S6
EXEC 6                          ›S6
EXEC 113                        ›S8
EXEC 123                        ›S9
ERROR 12                        ›Q
EXEC 7                          ›Q
                SCAN            ›S9
HALT                            ›Q
```

```
TUE 06 DEC 66 1646.30  JAF
BEGIN
CELL            JBUG,SIMPLE,ARYPTR, T0,T1,T2,T3,T4,T5,
                AMULT,LEV,LP,PTYPE,ARYELM,MOVSW,PARYSW,
                SUBTMP,ARYSW,SIGN,LABLOC,MOVARY,FLTEMP,
                REFC,RESTOR,POWR,NDXR,SCNT,RT1,RT2,RT2I,
                DTYPE,TREFR,SYMBLOC,BOTREF,MAXSTR;
CELL            PWR22,PWR23,PWR24,PWR32,PWR33,PWR34,
                PWR42,PWR43,PWR44;
TABLE           SYMBOL 400      ID SEMANT EXTRA;
TABLE           LABEL 200       ID VALUE LEVEL;
TABLE           REFR 20         ID REF;
DATA            PROC ,DLABL;
BIT             ARRAY, FORMAL;
STACK           BLOCK,DSTAK;
STACK           PCNT 20, XCNT 20;
RSTAK           SAVE 200,XSTAK,PSTAK;
INDEX           P,Q;
RINDEX          I,J;
**INIT BEGIN
1#              CODELOC←LABLOC; TALLY (CODELOC,100);
                (PERSLOC|ΔINTGR)→ARYPTR;
                ARYPTR+1→SIGN;
                SIGN+1→AMULT;
                AMULT+1→RT1;
                RT1+1→RT2I; (RT2I|0)→RT2;
                (RT2+1|ΔINTGR)→SUBTMP;
                SUBTMP+1→PERSLOC;
                λREFR→BOTREF;
                λSYMBOL→SYMBLOC;
                STORLOC→MAXSTR;
                0≡LEV→SCNT;
                FALSE≡ MOVSW≡ PARYSW → ARYSW;
                0→▷PERSLOC; PERSLOC→NDXR; TALLY PERSLOC;
                0→▷PERSLOC; PERSLOC→MOVARY; TALLY PERSLOC;
                JUMP BENT ⫷
                **BLOCK BEGIN, STORLOC, SYMBLOC
2# BENT#        PUSH(BLOCK,STORLOC); PUSH(BLOCK,λSYMBOL);TALLY LEV;
                ⫷
                **SKIP DEC
3#              IF PARYSW THEN
                                ⊠POP(PSTAK,ARYPTR)⊠
                ⫸
                FALSE→PARYSW;
                ⊠RETURN; FLAD4 ≠ ⊠ CALL BOUT ⫷
                **XFER FCTN
```

```
TUE 06 DEC 66 1646.30  JAF
48              L2→R I E
                **PROCEDURE RETURN
58              IF PARYSW THEN
                                [POP (PSTAK,ARYPTR)]
                ||
                [RETURN]E
                **END BLOCK
68              CALL BOUT; EXIT;
BOUT=           POP(BLOCK, ↑SYMBOL); TALLY (LEV,-1);
                IF STORLOC > MAXSTR THEN
                                STORLOC→MAXSTR
                ||
                POP (BLOCK,STORLOC); RETURN E
                **END FINI
78              CALL BOUT; MAXSTR→▷ARYPTR;
                [STOP];
                IF ARYSW THEN
                                (PERSLOC|▵INTGR)→T5; TALLY PERSLOC;
                                (SUBTMP|▵INTGR)→SUBTMP;
                                [NDXR=
                                POP(XSTAK,T5);
                                POP(XSTAK,RT2I);
                                ▷RT2I→I; 0→AMULT;
                                RT2I-I→SUBTMP];
                                CODELOC→LP;
                                [IFNOT I > 0 THEN
                                    RT2I+▷RT2I+I+AMULT∧377777→▷T5; RETURN
                                ||;
                                (XSTAK-@SUBTMP)J×(▷RT2I)J + AMULT→AMULT;
                                POP XSTAK; TALLY (I,-2); JUMP LP];
                ||
                IF MOVSW THEN
                                [MOVARY=];
                                AMULT→TJ;
                                (SUBTMP|▵INTGR)→SUBTMP;
                                (RT2I|▵INTGR)→RT2I;
                                [POP(XSTAK,TJ);ARYPTR→▷TJ;
                                PSTAKJ→SUBTMP; ▷SUBTMP→RT2I;
                                RT2I/1000000→J;
                                ARYPTR+J+1→ARYPTR];
                                CODELOC→LP;
                                [(▷SUBTMP)J→(▷TJ)J;
                                IF J > 0 THEN
                                                TALLY(J,-1);
                                                JUMP LP
```

```
                                    ||
                                    RETURN
                                    ⊡;
                    ||
                    COMPLETE &
**10- 13 TYPESESTP, EIDL DECL
10#                 ΔBOOL→DTYPE; JUMP SCHK &
11#                 ΔINTGR→DTYPE; JUMP SCHK &
12#                 ΔFRACT→DTYPE; JUMP SCHK &
13#                 ΔREAL→DTYPE;
SCHK⇐               IF SIMPLE THEN CALL ENT || ; EXIT;
ENT⇐                ENTER (SYMBOL.L2, (STORLOC|DTYPE).T0) ;
                    TALLY STORLOC; RETURN
                    &
                    **ARID EIDL FORMAL PAR
14#                 ( 0|ARRAY)→T0; CALL ENT &
                    **PRID EIDL FORMAL PAR
**PROC 10· AS FORMAL PARAM
15#                 (0|DLABL, PROC)→T0 ;
                    CALL FORMLAB&
16#                 TRUE→SIMPLE; 0→T0 &
**END SIMPLES
17#                 FALSE→SIMPLE &
**DECL NEW ARRAY
20#                 ( 0|ARRAY)→T0; (STORLOC|ΔINTGR)→T1 ;
                    L1→L2;
                    CALL ENT; 0→P ;
                    ⊡ ARYPTR≡T1→1;
                    1→AMULT ⊡ &
**BOUND PAIR
21#                 CALL BPAIR &
**LAST BOUND PAIR
22#                 IF (P=0) ∧ (L4 IS BCON) THEN
                                    IF (L4|0)≠0 THEN ⊡T1-L4→T1⊡ ELSE EXIT ||;||;
                    TRUE→ARYSW;
                    CALL BPAIR; (P+P|BDIR, ΔINTGR, BCON)→T0;
                    ⊡1→0₁; T0∨(AMULT×1000000)→ARYPTR; AMULT+1+1→ARYPTR⊡;
L22⇐                IF P>1 THEN
                                    ⊡TALLY (1,-2); 0₁×2₁→0₁⊡
                                    TALLY(P,-1); JUMP L22
                    ||;
                    EXIT;
BPAIR⇐              ⊡L4→1₁ ; (L2-L4+1≡0₁)×AMULT→AMULT;
                    TALLY(1,2)⊡; TALLY P; RETURN
                    &
```

```
TUE 06 DEC 66 1646.30  JAF
                **FUNCTION NAME
23#             (0|DTYPE)→T0; JUMP PENT ‖
**PROC NAME. MAKE BLOCK ENTRY
24#             (0| 0)→T0;
PENT#           T0→PTYPE; L1→T2; CALL PLABL; PUSH FLAD4;
                ⊂JUMP FLAD4; T1≠⊃;
                T0→R1; 0→REFC; ↑SYMBOL→SYMBLOC;
                JUMP BENT ‖
                **MARK REFER PARAM
25#             ENTER (REFR.L2, 0);
                TALLY REFC ‖
                **INITIALIZE PARAMA FROM STACK
26#             ⊂0→ i ⊃; ↑SYMBOL-3→P;
PINIT#          (1P|↓INTGR)→T1;
                IF (2P≡T2) IS DLABL THEN
                                ⊂P (↑PSTAKi)→T1⊃; JUMP L26
                ‖
                ↑(REFR(0P.REF.0))→T0;
                IF SIGNAL THEN
                                REFC-1→REFC;
                                IF T2 IS ARRAY THEN JUMP L26A ‖
                                (1P∨400000)→1P;
                                ⊂P (↑PSTAKi)→T1⊃;
                ELSE
                                IF T2 IS ARRAY THEN
                                                IFNOT PARYSW THEN
                                                                ⊂ARYPTR→SIGN⊃
                                                ‖
                                                ⊂ PUSH(XSTAK.↑T1);
                                                CALL ⓘMOVARY⊃
                                                TRUE≡MOVSW→PARYSW;
                                ‖
                                ⊂P (PSTAKi)→T1⊃;
L26A#           ‖
L26#            ⊂TALLY(1,-1) ⊃ ; TALLY (P,-3);
                IF P≥ SYMBLOC THEN JUMP PINIT ‖
                ⊂↑PSTAK-1 → ↑PSTAK⊃;
                IF PARYSW THEN
                                ⊂PUSH(PSTAK.SIGN)⊃
                ‖
                IFNOT REFC=0 THEN FAULT ‖
                BOTREF → ↑REFR ‖
27#             JUMP BENT ‖
                **PRIMARY IS ARRAY, PUSH PTR
30#             SYMBOL (L2, SEMANT, 0)→R1;
```

```
                IFNOT SYMBOL { 0,EXTRA, 0} IS ARRAY THEN
                                FAULT 30
                ||
                PUSH XCNT;
                &
31#             FAULT 31&
32#             FAULT 32&
                **LABEL SPECIF
                **LABEL ID AS FORMAL PARAM
33#             (0|DLABL)→ T0;
                CALL FORMLAB; EXIT;
FORMLAB=        ENTER (LABEL, L2, (LABLOC∨4000000), T0);
                ENTER (SYMBOL, L2, (STORLOC|∆INTGR), T0);
                CODELOC→T1; LABLOC→CODELOC;
                [JUMP *STORLOC]
                T1→CODELOC; TALLY LABLOC; TALLY STORLOC;
                RETURN &
                **PROCEDURE RETURN WITH ARG
34#             IF PTYPE ≠ 1 L2 THEN FAULT 34||;
                IF PARYSW THEN
                                [POP (PSTAK,ARYPTR)]
                ||
                [LOAD L2; RETURN]
                &
                **SAVE PARAM
35#             CALL SPAR &
                **LAST SAVE PARAM
36#             CALL SPAR; (SCNT|∆INTGR,BDIR,BCON)→SCNT;
                [PUSH (SAVE,SCNT)];
                0→SCNT ; EXIT ;
SPAR=           SYMBOL(L2,SEMANT, 0)→T0;
                IFNOT SIGNAL THEN FAULT 36 ||
                [PUSH(SAVE, T0); PUSH(SAVE,∧T0) ];
                TALLY SCNT; RETURN &
                **RESTORE PARAM
37#             [POP (SAVE,1) ];
                CODELOC → LP;
                [IFNOT != 0 THEN
                                POP(SAVE,RT1); POP (SAVE,* RT1);
                                TALLY( 1, -2); JUMP LP
                || ]
                &
                **ID→ PARIMARY
40#             IFNOT L2 IS BCON THEN
                SYMBOL(L2,SEMANT,0)→R2;
```

```
TUE 06 DEC 66 1646.30  JAF
                IFNOT SIGNAL THEN FAULT 5 ‖ £
**PID I . →PID PA Δ
418             TALLY PCNT;
                IFNOT L2 IS BCON THEN
                                SYMBOL (L2,SEMANT)→T2;
                                IFNOT SIGNAL THEN
                                                L2→T2; CALL LABL;
                                                (T0|ΔINTGR)→R2;
                                                TRUE→RR2;
                                ELSE
                                                T2→R2; FALSE→RR2
                                ‖;
                                EXIT;
                ELSE
                                IF L2 IS BDIR THEN
                                                (STORLOC|'L2)→T2; TALLY STORLOC;
                                                ⊡L2→T2⊡; T2→R2;
                                ELSE
                                                L2→R2
                                ‖
                                FALSE→RR2;
                ‖ £
**PID E . → PID PA Δ
428             TALLY PCNT;
                IF L2 IS BINA THEN
                                (STORLOC|'L2)→R2; TALLY STORLOC;
                                ⊡L2→R2⊡;
                                FALSE→RR2;
                                EXIT
                ‖
                L2→R2;
                FALSE → RR2;
                £
**PID ( SG → PID SG
438             0→T0; L3→T2; CALL PLABL;
                PUSH(PCNT,T1); PUSH PCNT
                £
448             0→T0; L3→T2; CALL PLABL;
                ⊡CALL ⓘT1⊡; ↑ (↑T1) → R1;
                EXIT;
PLABL=          ↑LABEL(T2,VALUE,0)→T1;
                IF SIGNAL THEN
                                IFNOT LABEL(0,LEVEL,0) IS PROC THEN
                                                FAULT 44
                                ‖
```

```
                ELSE
                                ENTER (LABEL,T2,T0,PROC);
                                ↑LABEL-2→T1
                ||
                RETURN ||
**Δ PA }
45#             POP(PCNT,P); P+CODELOC→P;
                POP (PCNT,T0); PUSH(PCNT,P+1);
                ⊠CALL ⓘT0⊡; ↑(P,T0)→PTYPE;
                ||
** Δ I
46#             IF LL1 THEN
                                P→CODELOC;
                                ⊠JUMP ⓘL1⊡;
                                CODELOC→P
                ELSE
                                (L1|0)→Op; TALLY (P,-1);
                || ||
**END OF PROCEDURE STUFF
47#             POP(PCNT,CODELOC); PTYPE→R1;
                ||
                **PCALL AS STATEMENT
50#             L2→T2; CALL LABL;
                ⊠T0#⊡||
                **GO TO LABEL
51#             L1→ T2; CALL LABL;
                ⊠JUMPⓘ T0 ⊡; EXIT;
LABL#           ↑LABEL(T2,VALUE,0)→T0;
                IFNOT SIGNAL THEN
                                ENTER(LABEL, T2, 0,LEV);
                                ↑LABEL - 2 → T0
                ||
                RETURN ||
                **CONDITIONAL GO TO
52#             L3→T2; CALL LABL; T0→T1;
                L2→T2; CALL LABL;
                ⊠IF L4 THEN JUMP ⓘT1 ELSE JUMP ⓘT0‖⊡ ||
                **KOUTENTS
53#             ⊠ ↑L2 ⊡→R2 ||
                **LOC
54#             ⊠ ↑L2 ⊡→R2 ||
                **ABS VAL
55#             ⊠ ‖L2⊡→R2 ||
                **FIX
56#             ⊠ ⌊ L2 ⊡→R2 ||
```

```
                **NEG
57#             ⊂ - L2 ⊃→R2 ⊂
                **MUL
60#             ⊂ L4 ×L2 ⊃→R2 ⊂
                **DIV
61#             ⊂ L4 /L2 ⊃→R2 ⊂
                **PLUS
62#             ⊂ L4 +L2 ⊃→R2 ⊂
                **MINUS
63#             ⊂ L4 -L2 ⊃→R2 ⊂
                **RELATIONS
64#             ⊂ L4 =L2 ⊃→R2 ⊂
65#             ⊂ L4 ≠ L2 ⊃→R2 ⊂
66#             ⊂ L4 > L2 ⊃→R2 ⊂
67#             ⊂ L4 < L2 ⊃→R2 ⊂
70#             ⊂ L4 ≥ L2 ⊃→R2 ⊂
71#             ⊂ L4 ≤ L2 ⊃→R2 ⊂
                **NOT
72#             ⊂ ~ L2 ⊃→R2 ⊂
                **AND
73#             ⊂ L4 ∧ L2 ⊃→R2 ⊂
                **OR
74#             ⊂ L4 ∨ L2 ⊃→R2 ⊂
                **EXPENENTIATE
75#             (L3|ΔINTGR)→T0; (L2|ΔINTGR)→T1;
                ⊂PUSH(XSTAK,↑T0); PUSH(XSTAK,↑T1)⊃;
                IF L3 IS ΔINTGR THEN
                                IF L2 IS ΔINTGR THEN
                                                ⊂CALL PWR22⊃; ΔINTGR→R1;
                                ELSE IF L2 IS ΔFRACT THEN
                                                ⊂CALL PWR23⊃; ΔREAL→R1;
                                ELSE IF L2 IS ΔREAL THEN
                                                ⊂CALL PWR24⊃; ΔREAL →R1;
                                ELSE FAULT 751
                                ⫴⫴⫴
                ELSE IF L3 IS ΔFRACT THEN
                                IF L2 IS ΔINTGR THEN
                                                ⊂CALL PWR32⊃;
                                ELSE IF L2 IS ΔFRACT THEN
                                                ⊂CALL PWR33⊃;
                                ELSE IF L2 IS ΔREAL THEN
                                                ⊂CALL PWR34⊃;
                                ELSE FAULT 752
                                ⫴⫴⫴
                                ΔREAL→R1;
```

```
                ELSE IF L3 IS ΔREAL THEN
                                IF L2 IS ΔINTGR THEN
                                                ⊠CALL PWR42⊠;
                                ELSE IF L2 IS ΔFRACT THEN
                                                ⊠CALL PWR43⊠;
                                ELSE IF L2 IS ΔREAL THEN
                                                ⊠CALL PWR44⊠;
                                ELSE FAULT 753
                                ⫴⫴⫴
                                ΔREAL→R1;
                ELSE FAULT 754
                ⫴⫴⫴
                (0|R1,BINA)≡ACCUM→R1⫷
                **SUBSCRIPT
76#             CALL ARYSUB ⫷
                **LAST SUBSCRIPT
77#             IF XCNT = 0 THEN
                                ⊠(ΔL3)L2⊠→R1; POP XCNT;EXIT
                ⫴
                CALL ARYSUB; POP XCNT;
                (L3|ΔINTGR)→T1;
                (STORLOC|ΔINTGR)→ARYELM; TALLY STORLOC;
                ⊠PUSH (XSTAK, T1)⊠;
                PUSH(XSTAK,↑ARYELM);
                CALL ⓘNDXR⊠;
                (ARYELM|↑L3)→R1; R1+400000→R1;
                EXIT;
ARYSUB⫧         ⊠PUSH (XSTAK,L2)⊠; TALLY XCNT; RETURN ⫷
                **CONDITIONAL EXPR
100#            ↑L2 → T2; ↑L3 → T3;
                ⊠SAVLOAD L4⊠;
                IF (T2≡T0) = T3 THEN
                                ⊠IF L4 THEN LOAD L3 ELSE LOAD L2⫴⫴⊠;
                                JUMP L100
                ELSE IF (T2∨T3) IS ΔBOOL THEN
                                FAULT 100
                ELSE
                                ⊠IF L4 THEN
                                                LOAD ↑L3
                                 ELSE
                                                LOAD ↑L2
                                ⫴⫴⊠
                ΔREAL→T0
                ⫴⫴;
L100⫧           (0|T0,BINA)≡ACCUM→R2⫷
```

```
TUE 06 DEC 66 1646.30  JAF
                **STORE
101#            ☒ L4→L2 ☒ ₤
                **NESTED STORE
102#            ☒ L4≡ L2 ☒ → R2 ₤
                **PCALL AS STATEMENT
103#            IFNOT L2=0 THEN FAULT 103‖₤
                **PCALL AS PARAM
104#            IF L2=0 THEN FAULT 104‖
                (0|↑L2,BINA)≡ACCUM→R2₤
105#            FAULT ↑ ₤
106#            FAULT ↑ ₤
                ** WHILE
107#            PUSH (DSTAK, CODELOC) ₤
                **IF CLAUSE
110#            PUSH FLAD1; ☒ IFNOT L2 THEN JUMP FLAD1 ‖ ☒ ₤
                **ELSE
111#            PUSH FLAD2; ☒JUMP FLAD2; FLAD1↵☒ ₤
                **END CONDITIONAL W- 0 ELSE
112#            ☒FLAD1↵ ☒ ₤
                **END CONDITIONAL WITH ELSE
113#            ☒FLAD2 ↵ ☒ ₤
                **FOR E→P STEP
114#            PUSH FLAD3; ☒ L4→L2; JUMP FLAD3; ☒;
                L2→T4; PUSH(DSTAK,CODELOC) ₤
                **STEP EXP THRU
115#            ↑ L2 →JBUG; (SIGN|JBUG) → SIGN;
                ☒L2+T4→T4; FLAD3 ↵;
                IF L2 < 0 THEN
                                LOAD -1
                ELSE
                                LOAD 1
                ‖☒;
                (0|&INTGR,BINA)≡ACCUM→JBUG;
                ☒JBUG→SIGN☒;
                ₤
                **TO EXP DO
116#            PUSH FLAD3;
                ☒IF T4 = L2 THEN JUMP FLAD3 ‖ ☒ ₤
                **THRU EXP DO
117#            PUSH FLAD3;
                ☒IFNOT (T4-L2) × SIGN ≤ 0 THEN JUMP FLAD3‖☒₤
                **STEP E WHEN OF UNTIL
120#            ☒L2+T4→T4☒ ₤
                **WHILE E DO
121#            PUSH FLAD3;
```

```
TUE 06 DEC 66 1646.30  JAF
                       IFNOT L2 THEN JUMP FLAD3
                       **UNTIL E DO
122#                   PUSH FLAD3;
                       IF L2 THEN JUMP FLAD3
                       **END OF FOR STMNT
123#                   POP(DSTAK,T0);
                       JUMP T0; FLAD3=
                       **FOR E-P WHILE OR UNTIL
124#                   L4-L2; PUSH(DSTAK,CODELOC);
                       
                       **STEP EXP TO
125#                   L2+T4-T4; FLAD3=
END
```

APPENDIX D

SYMBOLS

The following two tables list the symbols used in Production Language and Formal Semantic Language, as well as the sections of the manual which explain these symbols.

TABLE 1

PRODUCTION LANGUAGE: SYMBOLS

END	1 *	EXEC	1
NORM	2.1	UNSTK	1
VTERM	2.1	STK	1
ITERM	2.1	STAK	1
RES	2.2	SCAN	1
p...	2.2	ERROR	1
q...	2.2	HALT	1
INT	2.3	NEXT	1
≡	2.5	TEST	1
||	3	RETURN	1
→	1	CALL	1
←	1	I	1
		SG	1

* Numbers refer to sections of Chapter 2.

TABLE 2
FORMAL SEMANTIC LANGUAGE: SYMBOLS

CODELOC	1.1 *
STORLOC	1.1
PERSLOC	1.1
ACCUM	1.1
MARK	1.1
MAIN	1.1
SAVCELL	1.1
Li	1.2
LLi	1.2
Ri	1.2
RRi	1.2
... { ...,... }	1.5
... { ...,...,... }	1,5
FLADi	1.6
ⓘ	1.7
k	1.8
λ	1.9
t	1.10
‖	1.11
constant	1.12
(	1.13, 1.23
)	1.13, 1.23
⊂ ... ⊃	1.14
≡	1.15
... | ...	1.16
↓	1.17
TRUE	1.18
FALSE	1.18
... IS ...	1.19
CONST	1.20
=	1.21
≠	1.21
<	1.21
>	1.21
≤	1.21
≥	1.21
OK	1.22
SIGNAL	1.22
+	2
-	2
*	2
/	2
~	2
∧	2
∨	2
⊕	2
PUSH	3.1
POP	3.1
ENTER { ... }	3.2
→	3.3
TALLY	3.4
LOAD	3.5
SAVLOAD	3.5
SAVTEMP	3.5
☛	3.6, 3.7
☛̄	3.8
☛̲	3.8
RETURN	3.9

* Numbers refer to sections of Chapter 3.

(Cont.)

(Cont.)

REFERENCES

1. Feldman, J. A., A Formal Semantics for Computer Oriented Languages, Computation Center, Carnegie Institute of Technology, 1964.

2. Feldman, J. A., A Formal Semantics for Computer Languages and its Application in a Compiler-Compiler. Comm. of the ACM, Vol. 9, No. 1, January 1966.

3. Floyd, R. W., A Descriptive Language for Symbol Manipulation. JACM, October 1961.

DOCUMENT CONTROL DATA – R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
Lincoln Laboratory, M.I.T.

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b. GROUP
None

3. REPORT TITLE
VITAL Compiler-Compiler System Reference Manual

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Technical Note

5. AUTHOR(S) *(Last name, first name, initial)*
Mondshein, Lee F.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
8 February 1967	84	3

8a. CONTRACT OR GRANT NO.
AF 19 (628)-5167

b. PROJECT NO.
ARPA Order 691

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)
Technical Note 1967-12

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
ESD-TR-67-51

10. AVAILABILITY/LIMITATION NOTICES
Distribution of this document is unlimited.

11. SUPPLEMENTARY NOTES
None

12. SPONSORING MILITARY ACTIVITY
Advanced Research Projects Agency,
Department of Defense

13. ABSTRACT

This manual describes the general operation of the VITAL compiler-compiler system and the details of Production Language (PL) and Formal Semantic Language (FSL).

The appendices contain information on the system's meta-commands, a guide to the use of PL, an example of an ALGOL compiler, and a table of symbols used in PL and FSL.

14. KEY WORDS

VITAL
ALGOL
compiler-compiler system
computer languages
algorithmic language