

AD 745751



ARPA ORDER NO.: 189-1

R-887-ARPA

April 1972

Data Reconfiguration Service Compiler: Communications Among Heterogeneous Computer Centers Using Remote Resource Sharing

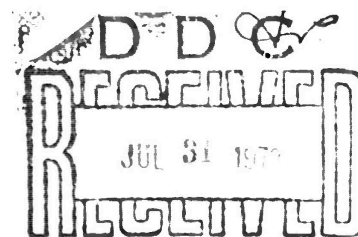
E. F. Harslem, J. Heafner and T. D. Wisniewski

A Report prepared for
ADVANCED RESEARCH PROJECTS

Reproduced by
NATIONAL TECHNICAL
INFORMATION SERVICE
U.S. Department of Commerce
Springfield, VA 22151

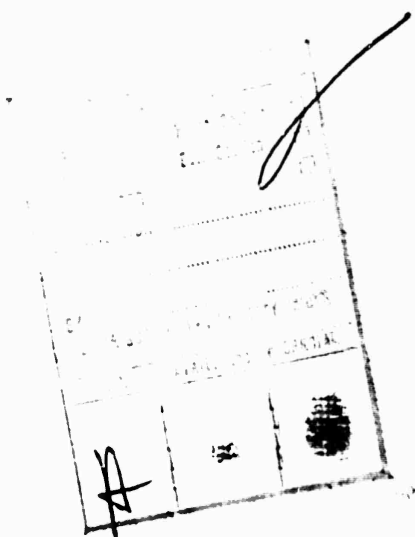


Rand
SANTA MONICA, CA 90406



APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

This research is supported by the Advanced Research Projects Agency under Contract No. DAHC15 67 C 0141. Views or conclusions contained in this study should not be interpreted as representing the official opinion or policy of Rand or of ARPA.



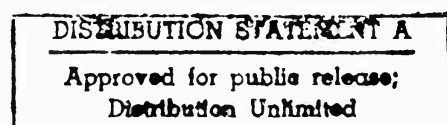
R-887-ARPA

April 1972

Data Reconfiguration Service Compiler: Communications Among Heterogeneous Computer Centers Using Remote Resource Sharing

E. F. Harslem, J. Heafner and T. D. Wisniewski

A Report prepared for
ADVANCED RESEARCH PROJECTS AGENCY



Rand
SANTA MONICA, CA 90406

Bibliographies of Selected Rand Publications

Rand maintains a number of special subject bibliographies containing abstracts of Rand publications in fields of wide current interest. The following bibliographies are available upon request:

*Aerodynamics • Arms Control • Civil Defense
Communication Satellites • Communication Systems
Communist China • Computer Simulation • Computing Technology
Decisionmaking • Game Theory • Maintenance
Middle East • Policy Sciences • Program Budgeting
SIMSCRIPT and Its Applications • Southeast Asia
Space Technology and Planning • Statistics • Systems Analysis
USSR/East Europe • Weapon Systems Acquisition
Weather Forecasting and Control*

To obtain copies of these bibliographies, and to receive information on how to obtain copies of individual publications, write to: Communications Department, Rand, 1700 Main Street, Santa Monica, California 90406.

Published by The Rand Corporation

PREFACE

This report describes an experimental service being developed in conjunction with the ARPANET for the Information Processing Techniques Office of ARPA. The work is an integral part of an overall program to explore the application of computer resources to defense-related requirements.

ARPANET is a network of computers located on the premises of approximately 20 ARPA contractors. There are plans to include several military installations. ARPANET addresses the problem of how to share heterogeneous computer resources, separated geographically, with widely varying languages and hardware. This study examines a computer program to conveniently translate one computer's messages to another, much in the same way that a translator aids communication between people speaking different languages.

This report delineates a part of the computer program, the compiler. This communication service reformats messages passing between dissimilar computers in such a way that the ARPANET appears to adapt the user's computer programs.

The report discusses both the compiler and its implementation. It is intended for specialists who want to maintain the compiler or to construct a similar service. The reader is assumed to be familiar with R-860-ARPA, *The Data Reconfiguration Service--An Experiment in Adaptable Process/Process Communication*. AD-757518

SUMMARY

This report describes the use, implementation, and maintenance procedures for the Data Reconfiguration Service (DRS) Compiler. The nature, scope, and goals of the DRS experiment are also explained. ARPANET resources are rapidly expanding, and the number of users is increasing. Of growing concern is the problem of incompatibilities between the remote user's program or terminal and the service that the user wishes to access. The DRS experiment tests and evaluates one method of resolving different communication interfaces by placing the DRS between user and server to reconfigure the data they pass to each other.

Several ARPANET sites will provide the DRS to compare and contrast its operation with the current kind of operation, which specifies standard data representations to which both user and server must conform. A goal of the experiment is to ascertain if such ARPANET "adaptability" yields a valuable mode of operation for a large spectrum of users.

The report provides an overview of the language in which data-reconfiguration definitions are expressed. Syntax is stated in a formal notation.

Another overview describes the DRS interpreter as a component of the service that performs the actual data transformations in real time. The report provides a functional description of the interpreter, and briefly describes each instruction's operation.

The study highlights the compiler's functions and operations. The compiler processes descriptions of data reconfigurations (for use by the interpreter) as instructions for reformatting the data passing between user and server. The compile process entails a lexical scan of the reconfiguration definition, a syntactic verification of the resulting lexical units, and the generation of instructions for the interpreter. The compiler does not communicate directly with the person who creates the descriptions; instead, it operates through a file system to retrieve the descriptions and emit the instruction sequence.

Because this report is a guide to maintaining the compiler, one section describes the function of each subroutine, the use of the

compiler generator, and the use and format of data structures; it also shows how to modify semantic subroutines.

Emphasis was placed on expediting compiler implementation instead of producing a fast compiler or highly efficient instructions for the interpreter. Thus, suggested improvements are included. The improvements would reflect lower maintenance, more optimized generated instructions, and smaller memory requirements for the compiler. The report also details compiler implementation, and points out pitfalls and alternate strategies.

ACKNOWLEDGMENTS

The authors would like to thank Vinton Cerf, University of California at Los Angeles, for specifying an initial interpreter, and also for his comments on this report. The authors would also like to thank the following persons for their suggestions and review of this study: R. M. Balzer, R. L. Bisbey, The Rand Corporation; and James White, University of California at Santa Barbara.

CONTENTS

PREFACE	111
SUMMARY	v
ACKNOWLEDGMENTS	vi1
FIGURES	x1

Section

I. INTRODUCTION	1
The Nature of the Experiment	1
The Scope of the Experiment	2
The Goals of the Experiment	3
II. THE DRS LANGUAGE	5
Highlights of Language Semantics	5
III. THE DRS INTERPRETER	6
Interpreter Overview	6
IV. THE COMPILER	8
Glossary	8
Compiler Functional Overview	8
Overview of Compiler Operations	9
Lexical Analysis	10
Syntax Analysis	11
Semantic Subroutines	12
Input and Output to the SMFS	19
Compiler Characteristics	19
Maintenance	20
Subroutines and the Source Language	20
Parser Generator	21
The Data Tables	22
Instruction-Sequence Table	22
Label Table	23
Literal/Identifier Table	24
Defined-Type Table (DFTYPE)	25
Path Table (LTRNTKN)	26
Modifying the Semantic Subroutines	26
Modifying a Non-Null Subroutine	27
Replacing a Null by a Non-Null Semantic Subroutine	27
Deleting a Non-Null Subroutine	28
Reflecting DRS Syntax Changes	28
Improvements	28
DRS Syntax	28
Parser Generator Output	29
Lexical Analyzer	29

Syntax Analyzer	29
Semantic Subroutines	30
Find Literal (FINDLT)	32
File Input/Output	32
V. DISCUSSION	34
Compiler Development	34
Looking Back	34
Appendix	
A. PARSER GENERATOR'S OUTPUT	37
B. INTERPRETER INSTRUCTIONS AND REPERTOIRE	47
C. DRS COMPILER LISTINGS	56
D. EXAMPLE COMPILATION	95
E. OBJECT LANGUAGE INSTRUCTION FORMATS	100
F. FLOWCHARTS OF COMPILER	102
REFERENCES	115

FIGURES

1. Data "Transformer"	2
2. Interpreter Interfaces	7
3. Interpreter Components	7
4. Functional View of the Compiler	9
5. Compiler Memory Requirements	19
6. Compiled Instruction Sequence File (DRS_OBJI_formname)	23
7. Compiled Label Table: Part of File DRS_OBJT_formname	23
8. Compiled Literals and Identifiers: Part of File DRS_OBJT_formname	24
9. Entries in the Literal/Identifier Table	25
10. Syntax Analysis Routine: Control Loop	103
11. Syntax Analysis Routine: Processing the Read State ...	104
12. Syntax Analysis Routine: Processing the Apply State ..	105
13. Syntax Analysis Routine: Processing the Look-Ahead and Push States	106
14. Lexical Analysis Routine	107
15. Lexical Analysis Routine: Verify and Index Subroutines	108
16. Semantic Routine: Control Loop	109
17. Semantic Routine: Printing the Instruction Lists	110
18. Input/Output Routine: Executing SMFS Channel Commands and Closing SMFS Files	111
19. Input/Output Routine: Opening and Writing an SMFS File	112
20. Input/Output Routine: Reading an SMFS File	113

I. INTRODUCTION

THE NATURE OF THE EXPERIMENT

The ARPANET [1-5] embodies a growing number of service centers that provide a collection of unique and valuable services as resources to an expanding remote user group. Users are frequently located either at sites with minimal computational power or at sites remote from the service they need. Collectively, they use a varied set of peripheral devices and application programs. The services, on the other hand, are generally predecessors of the ARPANET; they accommodate a more limited set of devices and program interfaces than those presented by ARPANET users. ARPANET personnel are investigating the problem of identifying and applying techniques to aid user and service communications.

Three approaches to solving these disparate communications requirements immediately come to mind:

1. Servers can tailor their software interfaces for coupling to a much larger set of users.
2. Each user can provide the necessary software interfaces to all services he wishes to access.
3. High-level data-representation protocols, to which both users and servers conform, can be defined.

The first approach is highly unattractive because of the burden and responsibilities it places on service centers. The second is likewise undesirable because it implies upgrading user equipment and modifying user programs to meet service center specifications. The inclination to date has been toward the third approach. Thus far, standards have been specified for logical message-path management and teletype-like character transmissions. At higher linguistic levels (e.g., data and file transmission, remote job entry, and interactive graphics), protocols have not been defined, partly because of the divergence of user needs at these problem-oriented levels.

An interim (and perhaps even long-term) solution to this communications dichotomy is the use of a fourth approach--the Data Reconfiguration Service (DRS) [6-7]. The DRS is a computer program, transparent to both

user and server, that couples user and server and carries out transformations on data passing between them (see Fig. 1).

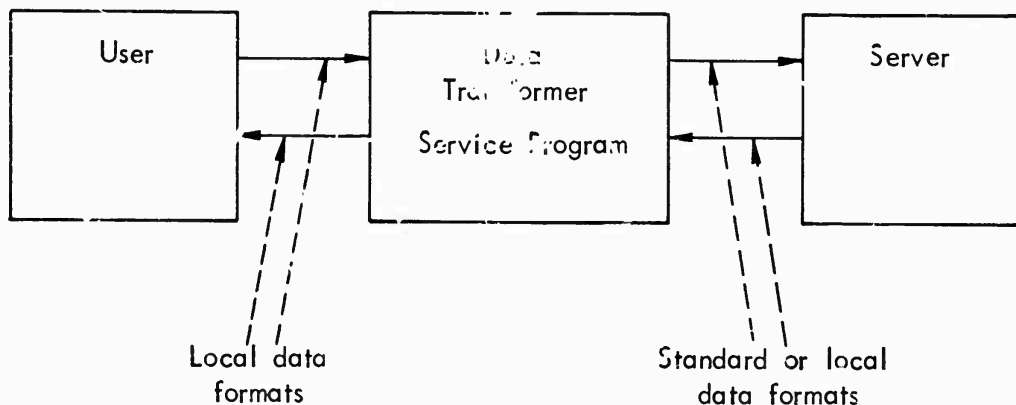


Fig. 1--Data "Transformer"

This approach offers several advantages. Because the reconfiguration definitions (called forms) are easily specified, user/server interface connections can be readily accomplished, with only minor changes made to their respective programs. For the $n \times m$ possible transformations (n users times m services), there need only be a single *adaptable* transformer in the ARPANET.

THE SCOPE OF THE EXPERIMENT

Four ARPANET sites (Rand, UCSB, UCLA, and MIT) are participating in DRS development. Specifically, Rand and UCLA are implementing DRS compilers. This report details the Rand implementation of the compiler. UCSB, UCLA, and MIT are implementing interpreters. The compilers take character-string definitions of data transformations and produce an intermediate (compiled) representation of the definition. The interpreters apply the compiled definitions to data streams passing between user and server in real time.

The Rand-implemented compiler and the UCSB interpreter will operate on UCSB's IBM 360/75 as a DRS service. The UCLA compiler and interpreter will operate on the UCLA Sigma-7. The MIT interpreter will offer the

reconfiguration service on a PDP-10, using data definitions compiled at UCLA and UCSB.

The DRS experiment is limited in scope. It is not intended as an intermediary for all ARPANET information exchange. The kinds of transformations that can be expressed easily and concisely in the DRS language include: character-set conversions, insertion and deletion of message headers and trailers (e.g., identifiers and counters), transposition of fields, data-format conversions (e.g., binary to binary-coded-decimal), expansion and compression of repeated symbol strings, and stripping or appending terminal signals.

Two kinds of uses are planned for the DRS. One is to offer a limited service to minimally configured nodes to gain some practical user experience. Another is to duplicate (in parallel) one or more existing user-server ties for purposes of comparative evaluation. Statistics of interest include declaration times of DRS data-reconfiguration definitions compared to coding time for the existing conventional implementations, and real-time data-transmission comparisons of the two operating modes.

THE GOALS OF THE EXPERIMENT

One experimental goal is to determine the viability of a mode of operation where a broad class of users can readily correspond with standard services, with minimal perturbations to the user's programs. The experiment is clearly prohibitive with respect to bandwidth and data rate for either large-volume data handling or highly interactive dialogues.

If a technically and economically aesthetic DRS results from this experiment, it could be provided as a standard service by: (1) distributing its capability to each major ARPANET service center so that both the DRS and the desired service reside at the same site, or (2) implementing a DRS interpreter in microcode on a small computer, as a unique service.

As a computer program, the DRS is expected to perform well on one-time-only data reformatting, where the original data are in one or more

formats and where writing programs to reformat the data would be time-consuming. Several examples of needed data transformations exist today, where the target data are to reside on a trillion-bit store to be shared by many installations. Other appropriate applications center around conversational-mode programs with low response-time requirements (10 to 30 characters/sec).

II. THE DRS LANGUAGE[†]

HIGHLIGHTS OF LANGUAGE SEMANTICS

A *form* is an operational definition of data reformatting performed on data passing over a unidirectional,[‡] logical ARPANET message path. Forms are specified to the DRS, then compiled and stored by the DRS. The interpreter applies a compiled form to an input data stream from the user and emits a reconfigured output stream to the server, and vice versa.

A form is an ordered collection of rules (language statements) for explicating reconfiguration operations on data streams. *Rules* specify replacement, comparison, or assignment operations on local variables in the context of the form. Rules are subdivided into an assemblage of terms. Data-stream-related terms describe the attributes (replication, length, value, and data type) of a field in the input or output stream. Rules consist of two parts: terms that describe or set conditions on the input data, and terms that format data for emission in the output stream. Each term may optionally and conditionally specify a transfer of control to the beginning of another rule. Rules are processed sequentially in the absence of explicit transfer of control.

[†]Appendix A includes the syntax of the DRS grammar. See Refs. 6 and 7 for a detailed description of DRS semantics.

[‡]In general, ARPANET connections are duplex, and a separate form is required to specify transformations on data passing in each direction.

III. THE DRS INTERPRETER

INTERPRETER OVERVIEW

The interpreter applies a pre-compiled form to a real-time data stream to effect data transformations[†] (see Fig. 2). The compiler produces the instructions, label table, literals, and identifiers. The interpreter is a stack machine driven by a Polish postfix instruction sequence. It consists of an instruction decoder; instruction execution routines (called operators) for data fetching, storing, and conversions; an assemblage of state registers for control; and a run-time stack to house instruction operands (see Fig. 3).

Run-time-stack operands are used for arithmetic expression evaluation, concatenation, and comparison; they are also used as arguments to input and output instruction routines.

The Current Input Pointer addresses the next bit to be processed in the input stream. The Rule Input Pointer addresses the bit position of the input stream corresponding to the beginning of the current rule. Two input pointers are required: the Current Input Pointer moves along as each term is processed, but the Rule Input Pointer is not advanced unless the rule correctly describes the input. The Output Pointer addresses the next available bit position for inserting data in the output stream. The Instruction Counter points to the current instruction of the pre-compiled instruction sequence. The Binary Switch is a true-false indicator set by input call and compare instructions, and checked by test and branch instructions. See Appendix B for instruction descriptions and the instruction repertoire.

[†]Private communication with James White, Computer Research Laboratory, University of California, Santa Barbara.

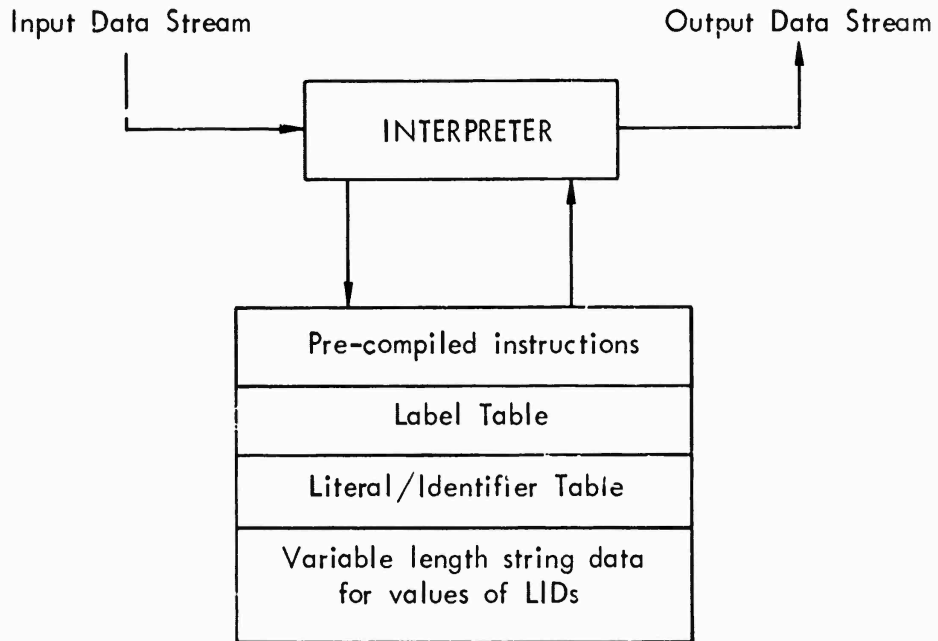


Fig. 2--Interpreter Interfaces

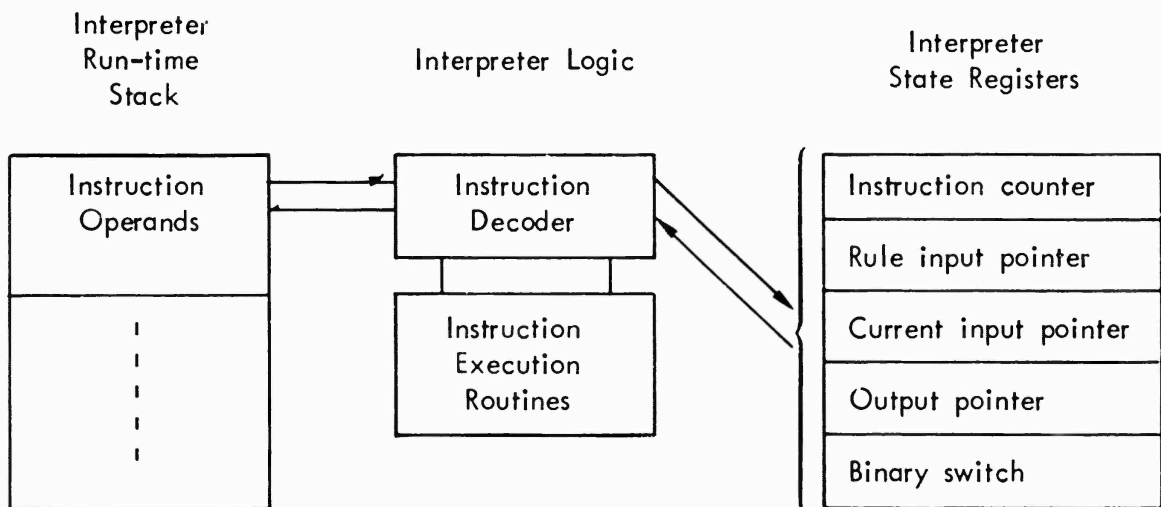


Fig. 3--Interpreter Components

IV. THE COMPILER

GLOSSARY

A *terminal* is any fundamental symbol string in the language, i.e., any string not defined in terms of other strings.

A *defined-type* is any symbol string in the language that is defined in terms of other symbol strings.

A *syntactic unit* is either a terminal or a defined-type.

The *Vocabulary Table* is a list of terminals.

A *production* is a statement in the syntactic specification of the language. Each production consists of a defined-type followed by a sequence of syntactic units.

COMPILER FUNCTIONAL OVERVIEW

The DRS compiler (a PL/1 program) accepts a *form* file as input and generates a source-diagnostic file for the user and two object files for execution by the interpreter. The compiler is logically made up of several data tables and three processes (the lexical analyzer, the syntax analyzer, and the semantic subroutines). The lexical-analyzer process scans and extracts meaningful characters, or groups of characters,[†] from the input stream (form definition). The character(s) is passed to the syntax-analyzer process to check the syntax of the input by comparing it to the syntactic units specified in a data table. If it agrees with any of the defined-types (see Appendix A), then the third process, a collection of semantic subroutines, is invoked to generate object code (see Fig. 4).

The data tables are pre-generated by a compiler generator, the LALR(k) Parser Generator,[‡] developed by the Computer Research Group at the University of Toronto [8-9]. A Backus Normal Form (BNF) [10] representation of the DRS syntax is input to the Parser Generator.

[†]The characters correspond to primitive elements of the DRS syntax, e.g., delimiters, integers, and identifiers.

[‡]The Parser Generator was written to produce XPL-coded compilers. In this instance, the XPL was hand-translated to PL/1.

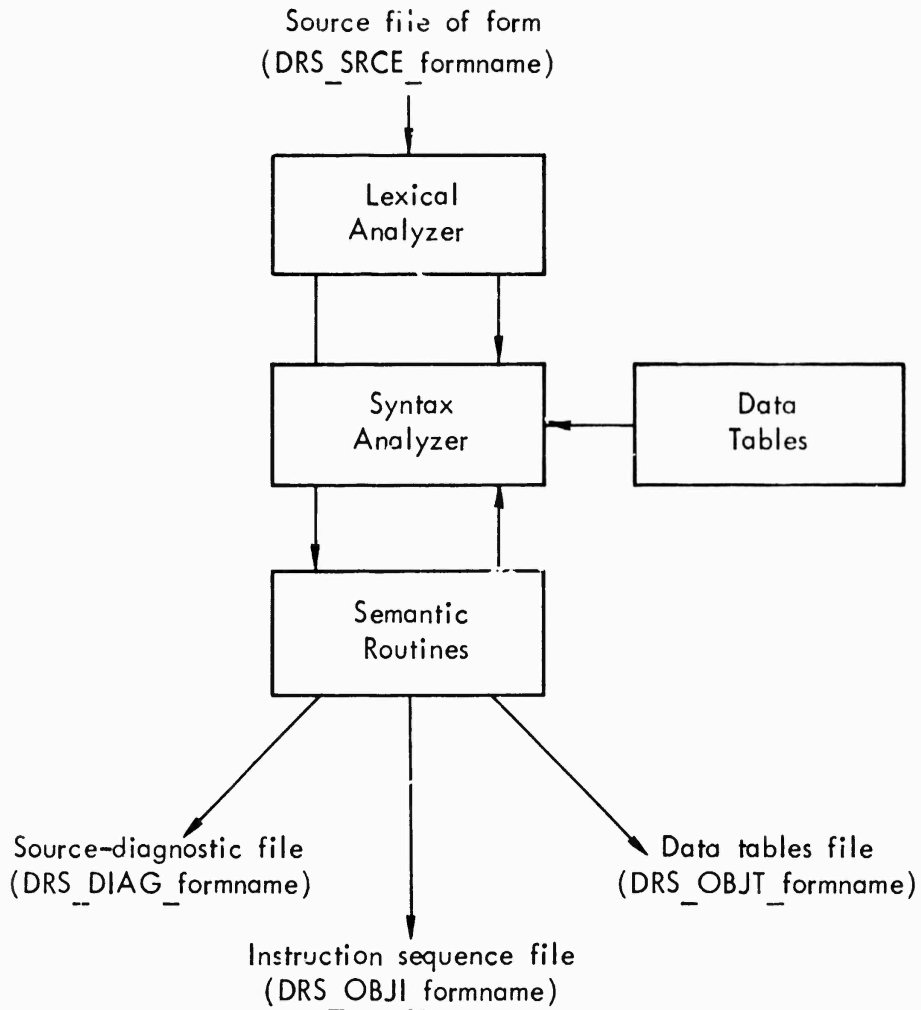


Fig. 4--Functional View of the Compiler

OVERVIEW OF COMPILER OPERATIONS

The compiler (see Appendix C) is invoked either as a job step or by being attached as an asynchronous subtask. Its source *form* input and its diagnostic and object outputs use the facilities of the Simple Minded File System (SMFS) [11], a remote ARPANET resource at UCSB.

The name of the *form* to be compiled is passed on to the compiler either in the "PARM" field of the execute card[†] for the compiler job

[†]See IBM System Reference Library, Form No. C28-6539-9.

step or as a supervisor call parameter if the compiler is attached as a subtask. The compiler concatenates the parameter (*formname*) to the string 'DRS_SRCE_' to make up the complete file name of the source form, DRS_SRCE_*formname*. The *formname* is appended to similar strings to form the output file names shown in Fig. 4. The compiler creates and writes the three output files.

The diagnostic file is always written; it contains a copy of each source *rule*. If the rule parses correctly, the compiled code is listed after the rule in a format typical of an assembly listing. If the rule does not parse, a diagnostic, written after the source rule, replaces the compiled code. (See Appendix D for an example of the diagnostic file.) If the compilation is error-free, the instruction sequence and data-table files are also written; if syntax errors are detected, these object files are purged.

LEXICAL ANALYSIS

The DRS syntax contains a set of terminal symbols. The "arbitrary number of" symbol, #, denoting the replication factor, is a terminal. Delimiters, arithmetic and concatenate operators are also terminals. Integers, alphanumeric strings, and literals are also terminal or primitive in the sense that they are fundamentally irreducible, as opposed to an arithmetic expression that might be reducible to a series of binary operations.

The Parser Generator deduces terminals from the BNF language description, and generates them to make up the Vocabulary Table.

The lexical analyzer detects terminals as it processes the input stream (form definition). By ignoring non-terminals, the lexical analyzer filters out ARPANET control characters. Upon detection of a terminal, an index[†] (rather than the terminal itself) corresponding to the entry in the Vocabulary Table is returned to the syntax analyzer. A special terminal (goal symbol)[‡] that cannot occur in the input stream

[†]The terminal type is available through the index, and the input terminal string is placed in a variable.

[‡]See the first production, GLUMP, in the syntax in Appendix A.

indicates the end of a form. The lexical analyzer translates an end-of-file from the form-definition source into the goal symbol, and passes the appropriate Vocabulary Table index to the syntax analyzer. (The goal symbol appears as " " in the syntax specification.) Literals are stripped of their delimiting double quote marks before being passed to the syntax analyzer.

SYNTAX ANALYSIS

The syntax analyzer is a "state machine," driven by initialized state tables produced by the Parser Generator. The tables guide the syntax analysis, which in turn calls upon the lexical analyzer to supply terminals. In fact, the Parser Generator produces a variety of output (see Appendix A). For example, it indicates ambiguities in the syntax and whether or not they can be resolved by looking ahead one terminal in the input stream. The most important output (for the present discussion) is a symbolic deck of XPL [12] table declarations and initialization constants. The tables are used in syntax analysis, except for the Vocabulary Table, which is placed in the lexical analyzer because it contains the terminals of the DRS language.

Analysis involves moving from one state to another, where the next state is a function of the current state and, for some states, a function of the lexical input. Each state produces a specific set of actions, e.g., requesting input or generating (compiling) code. The kinds of states include *read*, *look-ahead*, *push-down*, and *apply*.

A *read state* gets the next terminal from the lexical analyzer (the current state is pushed down on a state stack). A set of acceptable terminals is associated with each read state. Each terminal in the set leads to a next state. If the terminal read matches one of those acceptable in the present state, a transition is made to the corresponding next state. Failure to match one of the state's set is indicated by a syntax error, whereupon the current rule is ignored by skipping past the semicolon delimiter; the parse process then continues with the next rule.

When the syntax analyzer is in a *look-ahead state*, it asks the lexical analyzer for a *copy* of the next terminal (without advancing the

lexical analyzer's pointer in the form input). That is, look-ahead leaves the terminal available for subsequent look-ahead inspection or read. As in a read state, each look-ahead state has an associated set of acceptable terminals with corresponding next states. Likewise, if a terminal is matched with a member of the set, a transition is made to the corresponding state; otherwise, a syntax error occurs and processing resumes with the next rule.

A *push-down state* puts a syntactic unit on the stack. The next state is a function of only the current state. Push-down is used for productions that have empty righthand sides.

An *apply state* recognizes a defined-type and thus invokes a semantic subroutine, which in turn generates code. The next state is determined from the current state and the state stack. If a semantic error is detected, the syntax analyzer skips to the next rule to continue processing.

SEMANTIC SUBROUTINES

The semantic subroutines[†] generate the diagnostic file, the instruction sequence file, and the associated Label Table and Literal/Identifier Table file (see Fig. 4). The latter two files are accumulated internally until a complete form is recognized. A "record" of the diagnostic file, written whenever a rule is recognized, contains the source rule statement followed by either a diagnostic message or a list of the compiled instructions.

Table entries[‡] are made whenever literals or labels are encountered as identifiers. Labels are checked for uniqueness. Identifiers may have multiple references, with different values and data types for each reference. Literals are checked for uniqueness so that identical literals appear only once in the Literal/Identifier Table. (When multiple definition of a label occurs, the error is reported to the syntax analyzer.)

[†]In the program, the semantic subroutines are collectively named SMNTC.

[‡]The entries are made by the subroutines FINDID, FINDLT, and FINDLB.

The semantic subroutines generate code directly, without creating an intermediate parse tree. Because the grammar requires a look-ahead of one terminal, there is no need to try alternate productions until the successful one is found. Consequently, there is no need to back up over code generated from each unsuccessful "try." The semantic subroutines are given a parameter to indicate the recognized production. Thus, semantic actions are invoked for each recognized production--setting variables, making an entry in the Label Table or Literal/Identifier Table, or generating an instruction sequence. If any code is generated when a semantic subroutine is executed, a common exit is taken to update a location counter for the instruction sequence.

Specific semantic actions that occur upon recognition of the productions are listed below (the descriptions do not include pre- and post-processing common to each production):

GLUMP ::= FORM

An unconditional return with a code of zero is generated both in the instruction sequence and on the diagnostic file. The number of bytes of instructions is recorded in a length field preceding the instruction sequence (the interpreter uses the length to determine storage requirements). The instruction-sequence file is written along with the length field. Similar length fields precede the Label Table and the Literal/Identifier Table, which are written as shown in Fig. 4. The Label Table and the Literal/Identifier Table are written as unformatted SMFS files.

FORM ::= RULE | FORM RULE

No action is taken.

RULE ::= LABEL INPUTSTREAM OUTPUTSTREAM;

Unless the separator ":" appears first, an input/output term-flag is set to identify the next term encountered as an input term. The number-of-rules counter is incremented and the number-of-terms (within a rule) counter is cleared. The end-of-rule pseudo-instruction is generated. A second-pass compile is made (at the end of each rule) to complete the address field of AD instructions. On first-pass, these instructions are flagged with the pattern,

2130₁₆. The instruction sequence generated for the current rule is recorded[†] in the diagnostic file. The SICP instruction is generated as part of the sequence for the next rule.

`LABEL ::= INTEGER`

The label is entered in the Label Table; if the label is already defined, an error flag is set. An SICP is generated as the first instruction of the rule.

`LABEL ::= <EMPTY>`

An SICP is generated as the first instruction of a rule.

`INPUTSTREAM ::= <EMPTY> | TERMS`

Upon recognition of all input terms, an input/output term-flag is set to identify the terms that follow as output terms.

`TERMS ::= TERM | TERMS, TERM`

The Path Table (see p. 26) is cleared. Each element of the table corresponds to a defined-type and contains the number of the recognized production of that type. Semantic subroutines use the table to determine the history of the parse. Array HOLD is initialized to zeros. Each element of the array preserves indices in the Label Table or the Literal/Identifier Table. The fourth element of the array indicates whether the terms are input or output. The term counter is incremented and an end of term instruction is generated.

`TERM ::= IDENTIFIER DESCRIPTOR`

The input/output term-flag is checked. If it is on, the identifier descriptor was written on the wrong side of the input/output term delimiter ':'; thus, no code is generated. If the term occurs on the left (input) side of a rule, the instruction sequence LD x followed by STO is generated. When executed, this sequence stores the value of the identifier retrieved by the input call.

[†]The subroutine SPOCODE writes the file output.

TERM ::= IDENTIFIER

The following instruction sequence is generated:

```
NULL
LD      x
LIT
LD      x
LIC
LD      x
LIL
```

where x is an index in the Literal/Identifier Table. This sequence stacks the input/output parameters for the interpreter. The input/output term-flag is examined to determine which of the instructions (OUT, IND) to generate.

TERM ::= DESCRIPTOR | COMPARATOR

No action is taken.

IDENTIFIEE ::= IDENTIFIER

A semantic subroutine (the one corresponding to the defined-type IDENTIFIER) previously stored the identifier in the Literal/Identifier Table. This subroutine saves an index to the identifier for use by higher-level semantic subroutines.

TERM ::= IDENTIFIER

The identifier is a terminal symbol. If not already recorded, it is stored in the Literal/Identifier Table. An index in the table is saved for later use.

DESCRIPTOR ::= REP | DATYPE | VALUE | LENGTH CONTROL

No action is taken.

COMPARATOR ::= COMPAREXPR CONTROL | ASSGNEXPR CONTROL

No action is taken.

COMPAREXPR ::= CONCAT CONNECTIVE CONCAT

No action is taken.

ASSGNEXPR ::= IDENTIFIER .<=. CONCATEXPR

The instructions LD x followed by STO

are generated to store the value of the righthand side of the assignment statement in the identifier on the lefthand side. The x is an index in the Literal/Identifier Table for the identifier.

REP ::= #

The ARB operand is generated.

REP ::= ARITHEXPR

If the alternate production recognized for the defined-type PRIMARY is INTEGER, then the integer is saved for higher-level semantic subroutines; otherwise, no action is taken.

REP ::= <EMPTY>

The NULL instruction is generated.

DAType ::= B | O | X | E | A | ED | AD | SB | T(Identifier)

The allowable data types are as follows:

Type	Meaning	Code
--	Undefined	0
B	Binary	1
O	Octal	2
X	Hexadecimal	3
E	EBCDIC	4
A	Network ASCII [†]	5
ED	EBCDIC Decimal	6
	Number	
AD	Network ASCII	7
	Decimal Number	
SB	Signed Binary	8

For all but T(Identifier), the instruction IC x is generated, where x is one of the values 0 through 8. For T(Identifier), the instruction sequence LD x followed by LIT is generated, where x is an index in the Literal/Identifier Table.

VALUE ::= CONCAT

The index to the Literal/Identifier Table is saved.

VALUE ::= <EMPTY>

The NULL instruction is generated.

[†]Network ASCII is a standard 7-bit ASCII code right-justified in an 8-bit field, with a high-order bit equal to zero.

LENGTH ::= ARITH

The integer is saved if the arithmetic expression is an integer. The OUT instruction is generated if the term is an output term; otherwise, the following instruction sequence is generated:

```
INS
AD      end of rule instruction number
BF
LD      if an IDENTIFIER was specified
STO
```

LENGTH ::= <EMPTY>

The NULL instruction is generated.

CONNECTIVE ::= .LE. | .LT. | .GT. | .GE. | .EQ. | .NE.

For the syntactic unit below (left column), the code (right column) is generated:

.LE.	CLE
.LT.	CLT
.GT.	CGT
.GE.	CGE
.EQ.	CEQ
.NE.	CNE

The sequence AD followed by BF is generated.

CONCAT ::= VAL

No action is taken.

CONCAT ::= CONCAT ' | VAL

The CON instruction is generated.

VAL ::= LITERAL

The instruction LD x is generated, where x is an index in the Literal/Identifier Table.

VAL ::= ARITH

No action is taken.

ARITH ::= PRIMARY

No action is taken.

ARITH ::= ARITH OPERATOR PRIMARY

The instruction corresponding to the arithmetic operator is generated:

+ ADD
- SUB
* MUL
/ DIV

PRIMARY ::= IDENTIFIER | L(IDENTIFIER) | V(IDENTIFIER)

The instruction LD x is generated, where x is an index in the Literal/Identifier Table. An LIL is generated for L(IDENTIFIER); an LIV is generated for V(IDENTIFIER).

INTEGER ::= terminal

The value of the integer is saved and the instruction IC x is generated, where x is the value of the integer.

OPERATOR ::= + | - | * | /

No action is taken.

LITERAL ::= LITYPE LITSTRING

The literal is stored in the Literal/Identifier Table.

LITYPE ::= B | O | X | E | A | ED | AD | SB

No action is taken.

CONTROL ::= | OPTIONS

No action is taken.

OPTIONS ::= SFUR (ARITH) | SFUR (ARITH), SFUR (ARITH)

If the test is SR, FR, or UR, the RET instruction is generated; otherwise, the sequence LUL followed by BU is generated.

SFUR ::= S | SR

The instructions AD x followed by BF are generated, where x is the address of the first instruction in the next rule.

SFUR ::= F | SF

The sequence AD x followed by BT is generated.

SFUR ::= U | UR

No action is taken.

INPUT AND OUTPUT TO THE SMFS

Most input and output requests to the SMFS [11] are centralized in the input/output subroutine SMFSIO. Commands to SMFS are formatted as unaligned bit strings. UCSB's PL/1-Network interface [13] expects data as aligned array elements; however, the DRS compiler constructs the file commands in PL/1 structures. Data representation and access incompatibilities are resolved in SMFSIO by the POINT routine, through dummy dope vectors.

The input/output subroutine validates file operations. The SMFS and the ARPANET report the completion of a file transaction by returning a completion code[†] and by echoing the file command. The code is passed[‡] to the caller after receiving and checking the echo.

COMPILER CHARACTERISTICS

The compiler is a PL/1 program. Figure 5 shows the memory requirements for each module.

9000 bytes	STATIC COMPILER TABLES	}	Compiler Tables
2600	SMFSIO		
2600	SPUCODE	}	Compiler Routines
6200	PRSER		
3400	LXANLZR		
12000	SMNTC		
1100		}	SMFS File Routines
22000	PL1 Library Subroutines		
59000			
Total Bytes			

Fig. 5--Compiler Memory Requirements

[†]The completion code is returned in parameter DS in SMFSIO.

[‡]The code is passed in the variable RESPONSE.

The program consists of compiler routines, tables, PL/1 library routines, and SMFS file-interface routines. The file-handling routines, written in assembler language, add little to the total size. The static tables account for approximately 15 percent of the program size; the remainder is compiler code and library routines. Within a 65K partition, the compiler uses about 6K for dynamic storage.

Because of the simple parse process and few explicit subroutine calls, the compiler is fast. At present, there are no statistics on the compile rate.

MAINTENANCE

Subroutines and the Source Language

The compiler consists of the following routines:

PRSER	The syntax analyzer.
LXANLZR	The lexical analyzer.
SMNTC	The semantic routines.
FINDLT	Routine to seek and insert literals.
FINDLB	Routine to seek and insert labels.
FINDID	Routine to seek and insert identifiers.
SMFSIO	Routine to input/output to the SMFS.
POINT	Routine to overlay arrays onto structures for input/output.

The indentations indicate nested subroutines. The first three subroutines are the major components of the compiler (see pp. 10-12). The three FIND subroutines are called exclusively by the semantic subroutines. SMFSIO uses the SMFS. The PRSER, LXANLZR, and SMNTC use SMFSIO, although PRSER also directs the file system to open and close files. Subroutine POINT converts data representations between the PL/1-Network interface [13] and the compiler.

PL/1 F-level compiler, version 5, was used. The PL/1 character string built-in functions are necessary for the lexical analyzer. Note, for example, that the VERIFY function is not present in all PL/1 versions.

Some installations have a default source margin other than the one assumed for the compiler source code. Columns 1-72 must be used. The assumed PL/1 options are

EBCDIC	LOAD
CHAR60	NODECK
NOMACRO	FLAGW
NOSOURCE2	STMT
NOMACDCK	SIZE = 0133854
COMP	LINECNT = 057
SOURCE	OPT = 01
ATR	SORMGIN = (001, 072)
XREF	NOEXTDIC
NGEXTREF	NEST
NOLIST	OPLIST

Parser Generator

The *User's Guide* [8] describes options provided by the Generator. Appendix A is a listing from the run that generated the DRS compiler tables. Briefly, the following are the rules for constructing the BNF input.

Such specifications as IDENTIFIEE ::= IDENTIFIER are written simply as IDENTIFIEE IDENTIFIER. Successive productions are given on subsequent cards if the defined-type has alternatives. For example

OPERATOR ::= + | - | * | /

is input as

```
OPERATOR +
          -
          *
          /
```

The defined-types are terminated by a /* image. Names of the defined-types can be any continuous sequence of alphabetic characters, or the name can be delimited by the symbols '<' and '>', which allow imbedded blanks. For example, one can write either SFURIDENT or <SFUR IDENT> as the name of a defined-type. The name <EMPTY> specifically defines the null type. Finally, any symbol that does not appear on the left of a production is considered a terminal. The symbols +, -, *, / exemplify this in the DRS grammar because they appear only as alternate

productions of the defined-type OPERATOR. Note that the message below must precede the list of declarations; it indicates that the tables are acceptable (after editing from XPL to PL/1) as declarations to the DRS compiler (see Appendix A).

****NOTE**** GRAMMAR IS LALR(1)

The table declarations in Appendix A are identical to the punched cards produced by the Generator. The comment cards may be discarded. The declarations are edited to PL/1 in the following order.

- o Replace the phrase LITERALLY 'integer' by INITIAL (integer).
- o Remove the STATE-NAME array variable. (It is not used by the compiler.)
- o In the remaining array-variable declarations, replace any references to the variable declared 'LITERALLY' by the equivalent integer value.
- o In the remaining array-variable declarations, replace the attributes BIT (8) by BIN (8).
- o To save space, entries other than those containing terminal symbols can be discarded from the array-variable VOCAB.
- o The array index in XPL starts at zero, and in PL/1 at one; thus the initial XPL value should be deleted.
- o Use the contents of the vocabulary array to initialize the character-string variable, VOCAB, in the lexical analyzer. The vocabulary-array declaration may then be discarded.
- o Place the remaining array declarations in routine PRSER.

The Data Tables

Three data structures contain the compiler's output for the interpreter. They are (1) the Instruction-Sequence Table, (2) the Label Table (to resolve label references), and (3) the Literal/Identifier Table (to resolve references to literals or identifiers). The Defined-Type Table and Path Table control the semantic actions of the sub-routines. To conserve space, all arrays are declared static.

Instruction-Sequence Table

The Instruction-Sequence Table (see Fig. 6) contains the instruction sequence (see Appendix E) executed by the interpreter. It is

headed by a byte-count of the instruction-sequence length. Every instruction is 16 bits in length.

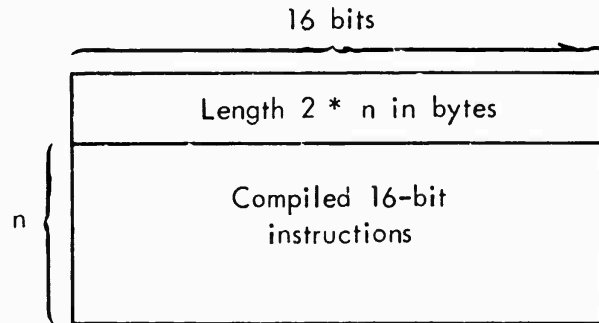


Fig. 6--Compiled Instruction Sequence File
(DRS_OBJI_formname)

Label Table

The interpreter uses the Label Table (see Fig. 7) to resolve label references made by instructions. The table is headed by a byte count of the table's length. Each entry contains a label name (an integer n , $0 \leq n \leq 9999$) and a byte offset of the instruction in the instruction sequence.

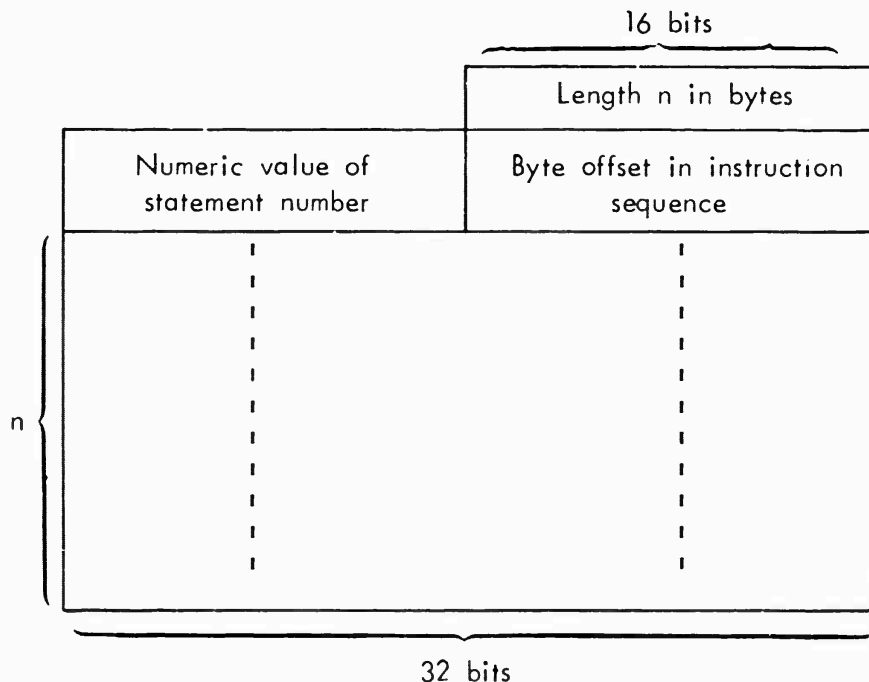
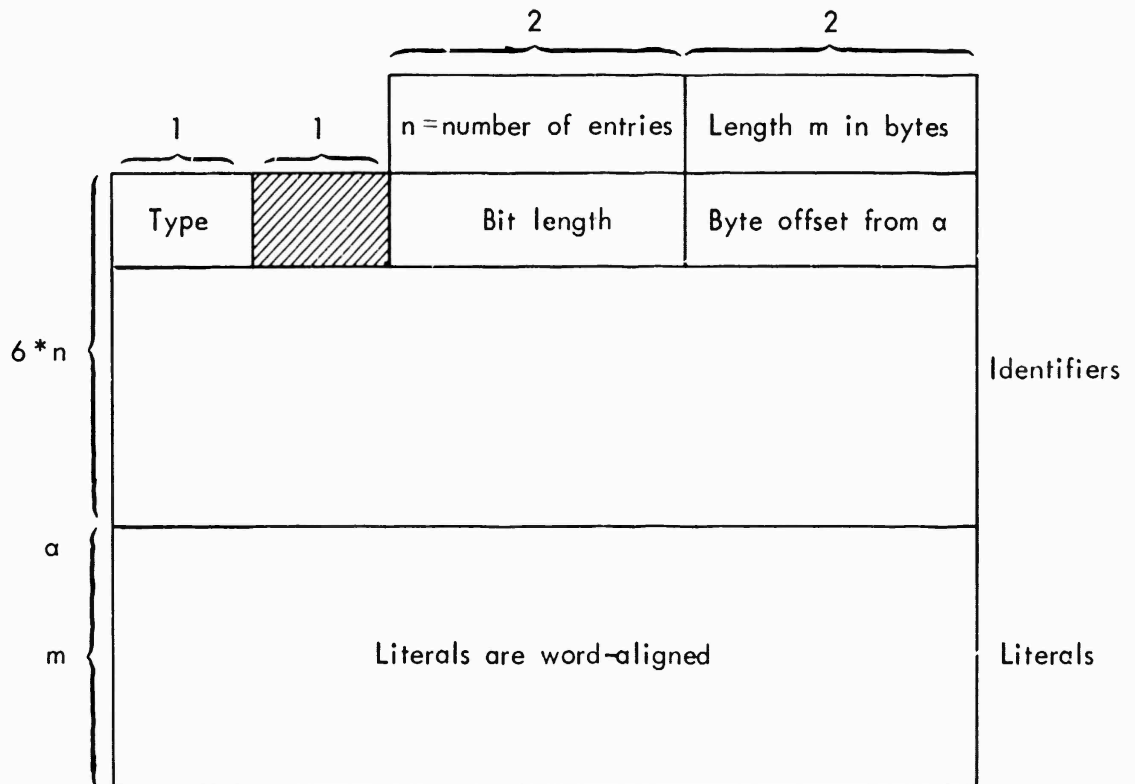


Fig. 7--Compiled Label Table: Part of File
DRS_OBJT_formname

Literal/Identifier Table

Each literal and identifier encountered in the source is entered in the Literal/Identifier Table (see Fig. 8). Literals are fully described by their entries, because their attributes are known at compile time.



Legend:

- Type 0 = undefined
- 1 = B (binary)
- 2 = ϕ (octal)
- 3 = X (hexadecimal)
- 4 = E (EBCDIC)
- 5 = A (ASCII)
- 6 = ED (EBCDIC encoded decimal)
- 7 = AD (ASCII encoded decimal)
- 8 = SB (signed binary, two's complement)

Fig. 8--Compiled Literals and Identifiers: Part of File
DRS_OBJT_formname

The type field (Fig. 9) contains a value from zero to eight that identifies the literal as binary, octal, hexadecimal, etc. The bit length of the literal is stored in the second field (Fig. 9). The byte offset is the location of the literal value (relative to the start of the literal pool).

The second half of the table (Fig. 8) is a literal pool containing each literal value in the format that conforms to its type specification.

Identifiers have null entries in the Literal/Identifier Table. The entries with undefined type (zero values) are easily recognized by the interpreter as identifier entries. The length and offset fields are updated by the interpreter as it processes the input-data stream.

Types B, ϕ , X, AC, ED, and SB point to 32-bit word-aligned data as shown below.



Types E and A point to byte-aligned symbol streams as shown below.

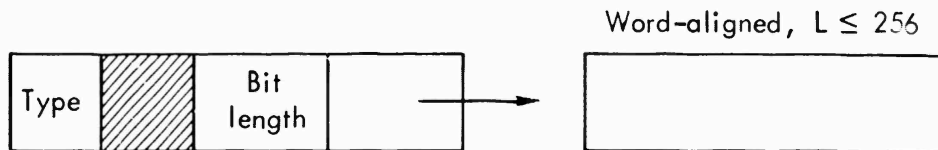


Fig. 9--Entries in the Literal/Identifier Table

Defined-Type Table (DFTYPE)

The Defined-Type Table, which is dependent on the syntax specification, records semantic actions. Each entry corresponds to a production in the DRS syntax, and has a non-zero value that is the ordinal of the defined-type for that production. For example, the specification for the defined-type FORM

FORM ::= RULE | FORM RULE

is currently the second defined-type in the DRS syntax. The production alternatives, RULE and FORM RULE, are the second and third productions in the specification. Thus, both the second and third entries in the Defined-Type Table have the value 2 because they are produced from the second defined-type, i.e., FORM.

A positive value indicates that a non-null semantic subroutine exists for the production. If the value is negative, the semantic subroutine is null. All productions are recognized, but only those with positive table values cause semantic actions.

Path Table (LTRNTKN)

The Path Table records the productions recognized while parsing a rule. Each entry corresponds to a defined-type. For example, if the third syntactic production is recognized, the second entry of the Path Table contains a 3 because the third production belongs to the second defined-type. The semantic subroutines use the table to determine the history of the parse. For example, there is a semantic subroutine for the second production of the defined-type

ARITH ::= PRIMARY | ARITH OPERATOR PRIMARY.

Recognition of the production requires that the terms ARITH, OPERATOR, and PRIMARY be previously recognized. (For each of these earlier recognitions, the semantic subroutines did generate instructions to load the run-time stack with the left and right parts of the arithmetic statement.) The term OPERATOR produces no semantic action, but the production of the defined-type OPERATOR is recorded in the Path Table. When the semantic subroutine for ARITH's second production is invoked, the table is examined to determine which OPERATOR production was previously recognized, and thus, which arithmetic instruction should be generated.

Modifying the Semantic Subroutines

Three modifications to the semantic subroutines, not involving syntax changes, are discussed below.[†] These are (1) changing a non-null

[†]Also see Appendix F.

subroutine, (2) inserting a non-null subroutine, and (3) making a non-null subroutine null. Null subroutines perform no semantic actions.

Modifying a Non-Null Subroutine

Each non-null subroutine is identified by a label of the form SROUT(x), where x corresponds to the production's BNF ordinal. When a production is recognized by the syntax analyzer (when it reaches an apply state), its ordinal is passed as an index to the semantic subroutines. SMNTC then transfers to the label subscripted by that index. For example, the grammar contains the following production:

LABEL ::= INTEGER | <EMPTY>

The Parser Generator assigns index numbers L_1 and L_j to the productions INTEGER and <EMPTY>. If the first is recognized, the syntax analyzer passes L_1 to the semantic subroutines, which determine whether the semantic subroutine is non-null. If not, SMNTC transfers to the label SROUT(L_1), to generate the code. Every non-null semantic subroutine terminates by transferring to EXIT, NOOP, or ERROR. To modify the existing semantic actions, replace the code bounded by the label and the transfer.

Replacing a Null by a Non-Null Semantic Subroutine

The semantic subroutines detect null subroutines by checking the Defined-Type Table entries. A negative entry means the subroutine is null, in which case the syntax analyzer regains control immediately after the production's number is recorded in the Path Table. To insert a non-null semantic subroutine for production L_1 , change the L_1 entry in the Defined-Type Table to the production's ordinal and insert the semantic code. The subscripted (SROUT(L_1)), precedes the code. The inserted code transfers to EXIT, NOOP, or ERROR. If the subroutine generates code, the subroutine transfers to EXIT to update the instruction counter. If no instructions are generated, the subroutine transfers to NOOP. If an error is detected, the subroutine goes to ERROR, where a return code is set for the syntax analyzer.

Deleting a Non-Null Subroutine

To make a non-null subroutine null, set a negative entry in the Defined-Type Table. Space can be saved either by converting the subroutine to comments or by deleting it.

Reflecting DRS Syntax Changes

When the DRS syntax is changed, the Defined-Type Table and the Path Table must be redefined to accommodate the new specification. In addition, semantic subroutines that are syntax dependent must be updated. To minimize program changes, place any new defined-types after the current defined-types. New table entries can be appended and the current semantic subroutines need not be changed. Redefine the maximum lengths of both tables to accommodate the new entries.

If a defined-type is changed, but the number of productions remains the same, replace the old type-definition by the new one. The tables do not change. Any other changes in syntax normally require redefining the tables and updating some semantic subroutines.

IMPROVEMENTS

The constraints of this experiment favored reducing compiler implementation time at the expense of optimization. That is, rather than concentrate on the efficiency of generated code to increase the interpreter's processing rate, we wanted feedback from early use to judge the effectiveness of this mode of operation.

This subsection identifies the more obvious compiler modifications. Compiler optimizing techniques have not been examined to produce the list of improvements. The kinds of improvements enumerated below entail both reorganization and recoding. Payoffs are reduced maintenance problems, more optimized code-generation, and reduced core requirements.

DRS Syntax

1. Reduce the number of productions to decrease program size. Some defined-types of the form shown below are extraneous.

A ::= B
B ::= C

Apply the transitive law that results in a production of the form shown below.

A ::= C

2. The syntax should be factored where possible, as illustrated below. Specify

X ::= R | C | D | E | F
Y ::= B | C | D | E | G

as

X ::= Z | F
Y ::= Z | G
Z ::= B | C | D | E .

Parser Generator Output

1. To reduce maintenance, collect the tables generated by the Parser Generator into a single subroutine that can be referenced externally.

Lexical Analyzer

1. Include the VOCAB and CHRTP table in (1) above.
2. Remove the order dependencies of the terminal symbols in the VOCAB and CHRTP tables.
3. Recode the analyzer in assembler language for improved speed.

Syntax Analyzer

1. Collect the state tables and other major compiler structures in a single subroutine that can be referenced externally.
2. Place the input/output tables, initialization code, and input/output termination code in SMFSIO.

3. Collect the lexical, syntactic, and semantic-diagnostic handling in a subroutine invoked only by the syntax analyzer.

4. Recode the analyzer in assembler language.

Semantic Subroutines

1. Place the generated instruction sequence, the Label Table, and the Literal/Identifier Table in the subroutine containing the major compiler structures.

2. Evaluate any arithmetic expression that involves a sequence of constants. Currently, in an arithmetic expression of the form

$$5 + 6 + 7 - 3 ,$$

the semantic subroutines would produce the sequence

IC	5
IC	6
ADD	
IC	7
ADD	
IC	3
SUB	

which is equivalent to an IC 15. Note that the interpreter can already handle two's complement arithmetic for the 12-bit integer constant, IC. This notion could be extended to include literal operands and the concatenate operator, with the appropriate alignment and conversion code.

Note that though this improvement is rather easy to implement and often cited as a compiler optimizing technique, in practice the gain is small because such expressions are seldom generated by the user.

3. Currently, the address and branch faults (AD, BF) sequence is generated for test and branch at the end of each term. One could define a new instruction to load a branch register. This instruction, the first of each rule, would load the register with the address of the next rule. Upon encountering an end-of-term, the interpreter would then test the Binary Switch register and either continue or branch indirectly through the branch register.

4. Currently, the instruction sequence is kept in core memory until the entire form is processed. The length of the instruction sequence is calculated after the form is processed, and the length precedes the code on the output file. The length should be written as a separate file (or the file should be backspaced to write the length) to remove the artificial limit on the form's size.[†]

5. Remove the input/output tables from the semantic subroutines and place them in SMFSIO.

6. The routine TABLES is detachable from the semantic subroutines and can be replaced by a dummy routine to conserve space. TABLES lists (on the diagnostic file) the contents of the Label Table and Literal/Identifier Table.

7. If arithmetic expressions involving constants are evaluated by the compiler (see (2) above), it is possible to check the validity of the label for the branch forms shown below:

S(x)
F(x)
U(x)

When the operand x is an arithmetic expression involving constants alone, the semantic subroutines could check the computed value for an integer, $0 \leq n \leq 9999$.

8. If a routine is written to centralize error processing, (see (3), Syntax Analyzer Improvements), certain syntax errors could be corrected. For example, the term "(A .GE. B : UR(5+x)," contains a syntax error in the control field; the user omitted the second right parenthesis before the comma. The error-processing routine could force "recognition" of the missing right parenthesis. Two practical results are achieved. If the form contains only a few such errors, it does not have to be recompiled; by continuing the compilation, other errors can be detected and reported. Corrective actions can be taken where the error involves a terminal for a defined-type represented by a single production. In fact, any composite that reduces to a unique terminal

[†]The current limit is 2000 instructions. To increase the size, change the variables MXINSTS and CODE.

(e.g., a missing-rule delimiter at the end of a form, a missing comma between descriptors, or a missing colon before a control expression) can be corrected.

Some semantic errors can be flagged and temporarily ignored in order to compile as much as possible. Errors reported by the semantic subroutines are usually such that the instructions are non-executable. When such errors are detected, the compiler skips to the next rule. Instead, the error condition could be held in abeyance until either an uncorrectable syntax error is found or until the entire form is parsed. For example, such errors as a doubly defined label or a compiler table overflow can be treated this way.

Find Literal (FINDLT)

1. Literals currently begin on a full-word boundary, but could be aligned on a byte boundary because the interpreter is independent of boundary alignment.

File Input/Output

1. Add a new entry point in SMFSIO for the following (see Ref. 11 to understand the jargon).

- a. Open a duplex connection for a file, given the name. Establish the socket numbers[†] within this entry point rather than in PRSEK, where it is currently done.
- b. Issue a delete and an allocate file command for all but the source file.
- c. Issue a read command to open the source file.
- d. Attempt to get the input from the SYSIN data set if the source is not available. Write diagnostic messages accordingly.

Add a new entry point in SMFSIO to close all files. If a file error is detected, delete the object files if they exist.

[†]Socket numbers are the names of each end of ARPANET logical message paths.

2. Remove input/output dependencies in the compile: by moving the input/output tables to the subroutine containing the major compiler structures, and by executing all input/output within SMFSIO.

V. DISCUSSION

COMPILER DEVELOPMENT

A primitive version of the semantic subroutines was coded and tested using JOSS [14], a console-oriented language. JOSS is strictly algebraic and provides a limited amount of working storage.

After initial checks, the semantic routines were coded in Conversational Programming System (CPS)[†] [15], another console-oriented language. The lexical analyzer and the routines to manage semantic tables were coded in CPS and checked and then combined with the semantic subroutines and a crude syntax analyzer. The combined program taxed the storage limits of CPS, but a working version of the compiler was developed.

The CPS program was then translated to PL/1. In the PL/1 version of the compiler, the semantic subroutines and lexical analyzer were fully developed and tested. A skeleton syntax analyzer from the Parser Generator replaced the CPS-coded analyzer; the state tables and the input/output routines were added.

LOOKING BACK

Perhaps the compiler should have been coded directly in PL/1, rather than in intermediate forms in the other languages. Many of the limitations encountered in JOSS and CPS do not exist in PL/1. Sections of troublesome code could have been coded in CPS in order to debug them easily, and then recoded in PL/1 in parallel to the PL/1 program development.

Compiler writing systems, e. the Parser Generator, provide a skeleton compiler of the lexical and syntax analyzers as well as convenient input/output mechanisms for the compiler's input and the semantic output. They free the user to concentrate on the BNF syntax and the semantics. We used only the syntax analyzer skeleton with no major

[†]CPS offers a subset of PL/1 constructs.

inconvenience. However, the greatest inconvenience was that we did not use the Parser Generator for its intended purpose--generating an XPL-coded compiler. Because our compiler was PL/1-coded, we had to go through the previously described editing process, which introduced many clerical errors.

The compiler began with a simple input/output method that reads card images and prints. Input/output code and tables are scattered throughout the compiler. Closer attention to input/output from the beginning would have prevented a number of problems that were later uncovered. Some of the suggested improvements reorganize the input/output into a centralized component.

Preceding page blank

-37-

Appendix A

PARSER GENERATOR'S OUTPUT

```
//JOB LIR DD DSN=B5765.LALR,DISP=SHR
// EXEC PGM=LALR,REGION=228K
//NONTERM DD SPACE=(CYL,9),UNIT=SYSNA
//FSM DATA DD SPACE=(CYL,9),UNIT=SYSNA
//PTABLES DD SYSOUT=B,DCB=(RECFM=FB,LRECL=80,BLKSIZE=400)
//SYSPRINT DD SYSOUT=A,DCB=(RECFM=FB,LRECL=133,BLKSIZE=1995)
//SYSIN DD *
OPTIONS (BNF,AINPUT,GPOST,DETAILED,LALR,NOTRACE,GRAMMAR,NOSXREF)
GLUMP FORM
FORM RULE
FORM RULE
RULE LABEL INPUTSTREAM OUTPUTSTREAM :
LABEL INTEGER
<EMPTY>
INPUTSTREAM TERMS
<EMPTY>
TERMS TERM
TERMS , TERM
OUTPUTSTREAM SEPARATOR TERMS
<EMPTY>
TERM IDENTIFIER ( DESCRIPTOR CONTROL )
IDENTIFIER
( DESCRIPTOR CONTROL )
( COMPAREXPR CONTROL )
( ASSGNEXPR CONTROL )
IDENTIFIER IDENTIFIER
DESCRIPTOR REP , DATYPE , VALUE , LENGTH
CONTROL : OPTIONS
<EMPTY>
COMPAREXPR CONCAT CONNECTIVE CONCAT
<EMPTY>
ASSGNEXPR IDENTIFIER .<=. CONCAT
IDENTIFIER A
R
E
F
L
I
S
T
U
V
X
AD
ED
FR
SR
SR
UR
<ALPHA ALPHANUM>
REP #
ARITH
<EMPTY>
DATYPE LITYPE
T ( IDENTIFIER )
<EMPTY>
VALUE CONCAT
```



```

    <EMPTY>
LENGTH ARITH
    <EMPTY>
OPTIONS TEST
    TEST , TEST
CONCAT VAL
    CONCAT || VAL
CONNECTIVE .LE.
    .LT.
    .GE.
    .GT.
    .EQ.
    .NE.
ARITH PRIMARY
    ARITH OPERATOR PRIMARY
LITYPE B
    O
    X
    F
    A
    ED
    AD
    SR
TEST <SFUR IDENT> ( ARITH )
VAL LITYPE LITSTRING
    ARITH
PRIMARY IDENTIFIER
    L ( IDENTIFIER )
    V ( IDENTIFIER )
    INTEGER
OPERATOR +
    -
    *
    /
<SFUR IDENT> S
    F
    U
    SR
    FR
    UR
SEPRATOR :
/*

```

```

/* THESE ARE LALR PARSING TABLES */

DECLARE MAXR# LITERALLY '55': /* MAX READ # */

DECLARE MAXL# LITERALLY '90': /* MAX LOOK # */

DECLARE MAXP# LITERALLY '104': /* MAX PUSH # */

DECLARE MAXS# LITERALLY '194': /* MAX STATE # */

DECLARE START_STATE LITERALLY '56':

DECLARE TERMINAL# LITERALLY '40': /* # OF TERMINALS */

DECLARE VOCAB# LITERALLY '69':

DECLARE VOCAB(VOCAB#) CHARACTER INITIAL ('(',')','+', '*', ')', ':', '-', '/',
',', '.', ':', '#', 'A', 'R', 'F', 'F', 'L', 'O', 'S', 'T', 'U', 'V', 'X', '|', '|', 'AD', 'ED',
'FR', 'SB', 'SR', 'UR', 'I', 'L', '<=', 'EQ', 'GE', 'GT', 'LE', 'LT',
',', 'NE', '<EMPTY>', 'INTEGER', 'LITSTRING', 'ALPHA ALPHANUM', 'REP', 'VAL',
'FORM', 'RULE', 'TERM', 'TEST', 'ARITH', 'GLUMP', 'LABEL', 'TERMS', 'VALUE',
'CONCAT', 'DATYPE', 'LENGTH', 'LITYPE', 'CONTROL', 'OPTIONS', 'PRIMARY',
'OPERATOR', 'ASSGNEXPR', 'SEPRATOR', 'COMPAREXPR', 'CONNECTIVE',
'DESRIPTOR', 'IDENTIFIE', 'IDENTIFIER', 'INPUTSTREAM', '<SFUR IDENT>',
'OUTPUTSTREAM'):

DECLARE P# LITERALLY '90': /* # OF PRODUCTIONS */

DECLARE STATE_NAME(MAXR#) BIT(8) INITIAL (0,0,1,1,1,1,1,1,8,8,8,8,8,9
,15,18,20,22,30,41,43,46,47,47,47,47,47,49,50,50,51,52,52,52,52,53,55
,56,56,56,56,59,60,61,62,63,64,64,65,66,66,66,66,67,68,69):

DECLARE RSIZE LITERALLY '373': /* READ STATES INFO */

DECLARE LSIZE LITERALLY '83': /* LOOK AHEAD STATES INFO */

DECLARE ASIZE LITERALLY '54': /* APPLY PRODUCTION STATES INFO */

DECLARE READ1(RSIZE) BIT(8) INITIAL (0,38,10,11,12,13,14,15,16,17,18,19
,20,21,23,24,25,26,27,28,38,40,10,11,12,13,14,15,16,17,18,19,20,21,23
,24,25,26,27,28,38,40,11,12,13,14,15,16,17,18,19,20,21,23,24,25,26,27
,28,40,11,12,13,14,15,16,17,18,19,20,21,23,24,25,26,27,28,40,11,12,13
,14,15,16,17,18,19,20,21,23,24,25,26,27,28,40,11,12,13,14,15,16,17,18
,19,20,21,23,24,25,26,27,28,38,40,1,11,12,13,14,15,16,17,18,19,20,21,23
,24,25,26,27,28,40,11,12,13,16,18,21,23,24,26,11,12,13,14,15,16,17,18
,19,20,21,23,24,25,26,27,28,38,40,14,17,19,25,27,28,11,12,13,14,15,16
,17,18,19,20,21,23,24,25,26,27,28,38,40,14,17,19,25,27,28,1,1,1,1,1,12
,13,14,15,16,17,18,19,20,21,23,24,25,26,27,28,38,40,11,12,13,14,15,16
,17,18,19,20,21,23,24,25,26,27,28,38,40,8,29,38,8,2,3,6,7,2,3,6,7,2,3,6
,7,2,3,4,6,7,2,3,6,7,1,11,12,13,14,15,16,17,18,19,20,21,23,24,25,26,27
,28,40,8,8,8,22,31,32,33,34,35,36,22,22,22,8,39,4,4,4,4,11,12,13,14,15
,16,17,18,19,20,21,23,24,25,26,27,28,38,40,9,1,11,12,13,14,15,16,17,18
,19,20,21,23,24,25,26,27,28,40,9,11,12,13,14,15,16,17,18,19,20,21,23,24
,25,26,27,28,38,40,9,9,1,30,4,4,4,9,1,5):

DECLARE LOOK1(LSIZE) BIT(8) INITIAL (0,38,0,4,8,9,0,8,0,8,0,8,0,4,9,0
,39,0,39,0,39,0,1,0,39,0,1,0,39,0,39,0,39,0,39,0,29,38,0,8,0,2,3,6,7,8

```


/* THE FOLLOWING IS THE INPUT GRAMMAR */

```

/*      1      GLUMP ::= FORM _I_                               */
/*      2      FORM ::= RULE                                     */
/*      3          | FORM RULE                                   */
/*      4      RULE ::= LABEL INPUTSTREAM OUTPUTSTREAM ;       */
/*      5      LABEL ::= INTEGER                                 */
/*      6          |                                           */
/*      7      INPUTSTREAM ::= TERMS                             */
/*      8          |                                           */
/*      9      TERMS ::= TERM                                    */
/*     10          | TERMS , TERM                                */
/*     11      OUTPUTSTREAM ::= SEPERATOR TERMS                 */
/*     12          |                                           */
/*     13      TERM ::= IDENTIFEE ( DESCRIPTOR CONTROL )        */
/*     14          | IDENTIFF                                     */
/*     15          | ( DESCRIPTOR CONTROL )                     */
/*     16          | ( COMPAREXPR CONTROL )                     */
/*     17          | ( ASSGNFXPR CONTROL )                      */
/*     18      IDENTIFF ::= IDENTIFIER                           */
/*     19      DESCRIPTOR ::= REP , DATYPE , VALUE , LENGTH     */
/*     20      CONTROL ::= : OPTIONS                             */
/*     21          |                                           */
/*     22      COMPAREXPR ::= CONCAT CONNECTIVE CONCAT         */
/*     23          |                                           */
/*     24      ASSGNFXPR ::= IDENTIFIER .<=. CONCAT             */
/*     25      IDENTIFIER ::= A                                  */
/*     26          | R                                           */
/*     27          | F                                           */
/*     28          | F                                           */
/*     29          | L                                           */
/*     30          | O                                           */
/*     31          | S                                           */
/*     32          | T                                           */
/*     33          | U                                           */
/*     34          | V                                           */
/*     35          | X                                           */
/*     36          | AD                                           */
/*     37          | FD                                           */
/*     38          | FR                                           */
/*     39          | SR                                           */

```

```

/*      40              | SR                      */
/*      41              | UR                      */
/*      42              | <ALPHA ALPHANUM>          */

/*      43  RFP ::= #                      */
/*      44          | ARITH                      */
/*      45          |                      */

/*      46  DATYPE ::= LITYPE                      */
/*      47          | T ( IDENTIFIER )          */
/*      48          |                      */

/*      49  VALUE ::= CONCAT                      */
/*      50          |                      */

/*      51  LENGTH ::= ARITH                      */
/*      52          |                      */

/*      53  OPTIONS ::= TEST                      */
/*      54          | TEST , TEST                */

/*      55  CONCAT ::= VAL                      */
/*      56          | CONCAT || VAL              */

/*      57  CONNNECTIVE ::= .LF.                      */
/*      58          | .LT.                      */
/*      59          | .GF.                      */
/*      60          | .GT.                      */
/*      61          | .EQ.                      */
/*      62          | .NE.                      */

/*      63  ARITH ::= PRIMARY                      */
/*      64          | ARITH OPERATOR PRIMARY      */

/*      65  LITYPE ::= R                      */
/*      66          | O                      */
/*      67          | X                      */
/*      68          | F                      */
/*      69          | A                      */
/*      70          | ED                      */
/*      71          | AD                      */
/*      72          | SH                      */

/*      73  TEST ::= <SFUR IDENT> ( ARITH )          */

/*      74  VAL ::= LITYPE LITSTRING                      */
/*      75          | ARITH                      */

/*      76  PRIMARY ::= IDENTIFIER                      */
/*      77          | L ( IDENTIFIER )          */
/*      78          | V ( IDENTIFIER )          */
/*      79          | INTEGER                      */

/*      80  OPERATOR ::= +                      */
/*      81          | -                      */
/*      82          | *                      */
/*      83          | /                      */

```

/*	84	<SFUR IDENT> ::= S	*/
/*	85	F	*/
/*	86	II	*/
/*	87	SR	*/
/*	88	FR	*/
/*	89	UR	*/
/*	90	SEPARATOR ::= :	*/

```

/* THESE ARE LALR PARSING TABLES */
DECLARE MAXR# LITERALLY '55': /* MAX READ # */
DECLARE MAXL# LITERALLY '90': /* MAX LOOK # */
DECLARE MAXP# LITERALLY '104': /* MAX PUSH # */
DECLARE MAXS# LITERALLY '194': /* MAX STATE # */
DECLARE START_STATE LITERALLY '56':
DECLARE TERMINAL# LITERALLY '40': /* # OF TERMINALS */
DECLARE VOCAB# LITERALLY '69':
DECLARE VOCAB(VOCAB#) CHARACTER 'INITIAL ('','(',')','+', '*', ')', ',', ';', ':', '-', '/',
',', '.', ':', '#', '@', 'A', 'B', 'E', 'F', 'I', 'O', 'S', 'T', 'U', 'V', 'X', '|', '|', '|', 'AND', 'END',
',', 'FR', 'SH', 'SR', 'UR', '_l_', '<=.', '.EQ.', '.GF.', '.GT.', '.LF.', '.LT.',
',', 'NE.', '<EMPTY>', 'INTEGER', 'LITSTRING', '<ALPHA ALPHANUM>', 'REP', 'VAL',
',', 'FORM', 'RULE', 'TERM', 'TEST', 'ARITH', 'GLUMP', 'LABEL', 'TERMS', 'VALUE',
',', 'CONCAT', 'DATATYPE', 'LENGTH', 'LITYPE', 'CONTROL', 'OPTIONS', 'PRIMARY',
',', 'OPERATOR', 'ASSGNEXPR', 'SEPERATOR', 'COMPAREXPRI', 'CONNECTIVE',
',', 'DESCRIPTOR', 'IDENTIFIER', 'IDENTIFIER', 'INPUTSTREAM', '<SFUR IDENT>',
',', 'OUTPUTSTREAM');
DECLARE P# LITERALLY '90': /* # OF PRODUCTIONS */
DECLARE STATE_NAME(MAXR#) BIT(8) INITIAL (0,0,1,1,1,1,1,1,8,8,8,8,9
,15,18,20,22,30,41,43,46,47,47,47,47,49,50,50,51,52,52,52,53,55
,56,56,56,56,59,60,61,62,63,64,64,65,66,66,66,67,68,69);
DECLARE RSIZE LITERALLY '373': /* READ STATES INFO */
DECLARE ASIZE LITERALLY '54': /* APPLY PRODUCTION STATES INFO */
DECLARE READ1(RSIZE) BIT(8) INITIAL (0,38,10,11,12,13,14,15,16,17,18,19
,20,21,23,24,25,26,27,28,38,40,10,11,12,13,14,15,16,17,18,19,20,21,23
,24,25,26,27,28,38,40,11,12,13,14,15,16,17,18,19,20,21,23,24,25,26,27
,28,40,11,12,13,14,15,16,17,18,19,20,21,23,24,25,26,27,28,40,11,12,13
,14,15,16,17,18,19,20,21,23,24,25,26,27,28,40,11,12,13,14,15,16,17,18
,19,20,21,23,24,25,26,27,28,38,40,1,11,12,13,14,15,16,17,18,19,20,21,23
,24,25,26,27,28,40,11,12,13,16,18,21,23,24,26,11,12,13,14,15,16,17,18
,19,20,21,23,24,25,26,27,28,38,40,14,17,19,25,27,28,11,12,13,14,15,16
,17,18,19,20,21,23,24,25,26,27,28,38,40,14,17,19,25,27,28,1,1,1,1,1,12
,13,14,15,16,17,18,19,20,21,23,24,25,26,27,28,38,40,11,12,13,14,15,16
,17,18,19,20,21,23,24,25,26,27,28,38,40,8,29,38,8,2,3,6,7,2,3,6,7,2,3,6
,7,2,3,4,6,7,2,3,6,7,1,11,12,13,14,15,16,17,18,19,20,21,23,24,25,26,27
,28,40,8,8,8,22,31,32,33,34,35,36,22,22,22,8,39,4,4,4,11,12,13,14,15
,16,17,18,19,20,21,23,24,25,26,27,28,38,40,9,1,11,12,13,14,15,16,17,18
,19,20,21,23,24,25,26,27,28,40,9,1,12,13,14,15,16,17,18,19,20,21,23,24
,25,26,27,28,38,40,9,9,1,30,4,4,4,9,1,5);
DECLARE LOOK1(LSIZE) BIT(8) INITIAL (0,38,0,4,8,9,0,8,0,8,0,8,0, ,9,0
,39,0,39,0,39,0,1,0,29,0,1,0,39,0,39,0,39,0,39,0,29,38,0,8,0,2,3,6,7,8
,0,8,0,2,3,6,7,0,4,9,0,5,9,0,8,0,8, ,22,0,22,0,22,0,9,0,9,0,9,0,9,0,1,0
,30,0,9,0);
/* PUSH STATES ARE BUILT-IN TO THE INDEX TABLES */
DECLARE APPLY1(ASIZE) BI (8) INITIAL (0,0,0,20,0,0,0,43,0,0,8,0,0,3,0
,44,46,47,0,0,0,27,2,8,43,4,5,6,0,0,0,0,0,0,45,18,10,0,0,2,3,7,12,0,9,0
,11,0,17,0,41,0,0,0,0);
DECLARE READ2(RSIZE) BIT(8) INITIAL (0,109,147,62,63,64,132,65,66,135
,136,137,67,68,69,70,142,71,144,145,183,146,147,129,130,131,132,65,134
,135,136,137,67,139,140,141,142,143,144,145,183,146,129,130,131,132,133
,134,135,136,137,138,139,140,141,142,143,144,145,146,129,130,131,132
,133,134,135,136,137,138,139,140,141,142,143,144,145,146,129,130,131
,132,133,134,135,136,137,138,139,140,141,142,143,144,145,146,129,130
,131,132,65,134,135,136,137,67,139,140,141,142,143,144, ,5,183,146,57
,129,130,131,132,133,134,135,136,137,138,139,140,141,142,143,144,145
,146,173,169,172,170,15,171,175, , -176,62,63,64,132,65,66,135,136,137
,67,68,69,70,142,71,144,145,183,1 ,189,188,190,192,191,193,129,130,13

```


Appendix B

INTERPRETER INSTRUCTIONS AND REPERTOIRE

INSTRUCTION DESCRIPTIONS

Literal or Identifier Reference (LD)

A LD, which points to either an entry for an identifier (variable) or a literal in the Literal/Identifier Table, is an operand in the instruction sequence. The instruction decoder pushes a LD, unmodified, onto the stack.

Integer Constant (IC)

The IC operand is a 12-bit 2's complement constant in the instruction sequence. The IC is included for efficient handling of (absolute) numbers without the indirect addressing associated with a literal reference. It is pushed on the stack unchanged.

Address Constant (AD)

The AD operand is a 12-bit positive integer that addresses an instruction in the instruction sequence. It is used only as an operand of a branch operator.

Arbitrary Replication (ARB)

The ARB operand, which indicates an indefinite replication factor[†] in an input term, is a constant in the instruction sequence.

Null Value (NULL)

The NULL operand in the instruction sequence indicates an omitted field in a term. It occurs only for terms that collect data from the input stream or emit data in the output stream.

[†]The arbitrary replication is denoted by the pound sign, #, in the DRS syntax.

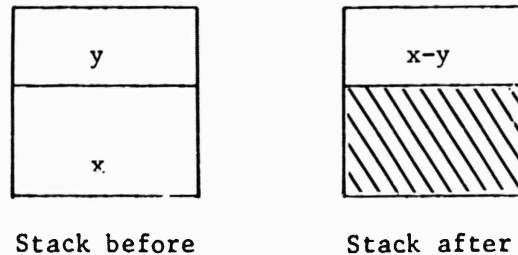
Store (STO)

A value is stored in the Literal/Identifier Table. The first two stack entries describe the location and value, respectively. Both elements are removed from the stack upon execution.

Binary Operators (ADD, SUB, MUL, DIV, CON)

The binary operators compute $x <op> y$, delete both x and y from the stack,[†] and push the result[‡] back onto the top of the stack:

Example: $x-y$



Binary operators have no effect on the Binary Switch register. All operators except concatenate (CON) expect x and y to describe type B, O, X, AD, ED, or SB.[‡] The result is always a 32-bit type-B element. The concatenate operator expects both types x and y to be identical.

Compare (CEQ, CNE, CLE, CLT, CGE, CGT)

The compare operators (e.g., .EQ., .LT., etc.) test the values described by the first two stack entries. The second element of the stack is compared to the first. The form fails for Boolean comparators where types differ. For CEQ and CNE, the data must have identical type and length attributes. For identical types, B, O, AD, ED, and X cause

[†]The stack may actually contain instruction operands that describe data (rather than the data themselves). For convenience of illustration, the data rather than their descriptors are shown on the stack. For detailed formats of instructions and tables, see Sec. IV and Appendix E.

[‡]B, O, X, AD, ED, and SB represent binary, octal, hexadecimal, ASCII decimals, EBCDIC decimals, and signed binary, respectively.

binary right-justified comparison operations. Types A and E[†] cause left-justified string comparison operations. Prior to the comparison, the shorter string is right-padded with blanks.

Branch (BT, BF, BU)

The branch operators check the Binary Switch register and either increment the Instruction Counter register by one or replace it by the value described by the first stack operand. The top stack operand addresses a new Instruction Counter value in the instruction sequence. The top stack operand is removed.

Input Call (INS, IND)

The input call operators retrieve data from the input stream. They require four stack operands as shown below.

length descriptor	binary number or null
value descriptor	LD or null
data-type descriptor	binary code or null
replication descriptor	binary number, arbitrary indicator, or null

If the value-descriptor parameter is null, the input routine extracts as much data as needed from the input stream (of the required data type) to satisfy the length-descriptor and replication-descriptor requirements. If the value descriptor is not null, the input-stream data is compared to the described value. The Binary Switch is set to true if the four stack operands correctly describe the input. The stack operands are deleted. For an INS, the string obtained from the input is described by the top operand of the stack. For an IND, the stack is left empty. If the conditions are not satisfied, the stack operands are deleted and the Binary Switch is set to false.

[†] A and E represent ASCII and EBCDIC, respectively.

Upon successful application of the input-call operator, the Current Input Pointer register is advanced over the data extracted from the input stream.

Output Call (OUT)

The output-call operator emits data in the output stream. The four stack operands are the same as those described for the input call. The value is converted if the output type and the value descriptor differ. The value expression is transformed to the desired output type and fitted in the field specified by the length expression. See Ref. 6 for truncation and padding rules. The Binary Switch is unaffected.

Move Pointer (SCRIP, SRCP)

These operators replace the contents of the Current Input Pointer by the contents of the Rule Input Pointer and vice versa. There are no stack operands.

Return Value (RET)

The return-value operator returns, to the originating user, the value described by the first stack operand.

Look Up Label (LUL)

The Label Table is searched for the entry referenced by the stack operand (which is a type-B value). If located, the stack entry is replaced by the relative address in the instruction sequence of the label (in the form of an AD operand); if the label is not found, the rule fails.

Load Identifier Value, Length, Type, Contents (LIV, LIL, LIT, LIC)

The top stack operand is a LD to a defined identifier. These operands extract the indicated attribute (value, length, type, contents) of the identifier and replace the first stack operand by the extracted value.

End of Term, Rule (EOT, EOR)

These pseudo operands are used to debug real-time data-reconfiguration failures. If a form fails, the interpreter scans forward in the form's instruction sequence and reports to the originating user, over the control connection, the rule and term on which the form failed. These operands carry a sequence number so that the failure may be coupled to the particular term in the specific rule that failed.

INSTRUCTION REPERTOIRE

Data Descriptors

<u>Meaning</u>	<u>Mnemonic</u>	<u>Stack^a</u>	<u>Comment</u>
Literal or Identifier Reference	LD	→ LDx	This set of operands, processed by subsequent operators, is pushed unchanged into the stack.
Integer Constant	IC	→ ICx	
Address Constant	AD	→ ADx	
Arbitrary Replication	ARB	→ ARB	
Null Value	NULL	→ NULL	

Data Storing

<u>Meaning</u>	<u>Mnemonic</u>	<u>Stack</u>	<u>Comment</u>
Store	STO	x,y →	y is stored in x

Binary Operators

<u>Meaning</u>	<u>Mnemonic</u>	<u>Stack</u>	<u>Comment</u>
Add	ADD	x,y → y+x	
Subtract	SUB	x,y → y-x	
Multiply	MUL	x,y → y*x	

^aWe use | to indicate the top of stack. Hence, |A,B → |C means A and B were the first two operands on the stack prior to instruction execution and C was first on the stack with A and B removed after the instruction execution.

INSTRUCTION REPERTOIRE (cont.)

Binary Operators (cont.)

<u>Meaning</u>	<u>Mnemonic</u>	<u>Stack</u>	<u>Comment</u>
Divide	DIV	x,y → y/x	
Concatenate	CON	x,y → y x	

Comparison

<u>Meaning</u>	<u>Mnemonic</u>	<u>Stack</u>	<u>Comment</u>
Equal	CEQ	x,y →	For each, the comparison y comp x is made and the binary switch set accordingly.
Not Equal	CNE	x,y →	
Less than or Equal	CLE	x,y →	
Less than	CLT	x,y →	
Greater than or Equal	CGE	x,y →	
Greater than	CGT	x,y →	

Branching

<u>Meaning</u>	<u>Mnemonic</u>	<u>Stack</u>	<u>Comment</u>
True	ET	ADx →	The instruction counter is set to ADx if the branch is taken.

INSTRUCTION REPERTOIRE (cont.)

Branching (cont.)

<u>Meaning</u>	<u>Mnemonic</u>	<u>Stack</u>	<u>Comment</u>
False	BF	ADx +	
Unconditional	BU	ADx +	

I/O of Data Stream

<u>Meaning</u>	<u>Mnemonic</u>	<u>Stack</u>	<u>Comment</u>
Input and Save	INS	Len,Val,Type,Rep + LDx	The stack has a pointer to value found in input stream.
Input and Discard	IND	Len,Val,Type,Rep +	The binary switch is set for both input calls.
Emit	OUT	Len,Val,Type,Rep +	

Input Pointer Register Manipulation

<u>Meaning</u>	<u>Mnemonic</u>	<u>Stack</u>	<u>Comment</u>
Current + Rule	SCRp	None	The rule input pointer is set to the current input pointer.
Rule + Current	SRCP	None	The current input pointer is reset to the rule input pointer.

INSTRUCTION REPERTOIRE (cont.)

Form Execution Termination

<u>Meaning</u>	<u>Mnemonic</u>	<u>Stack</u>	<u>Comment</u>
Return Value	RET	x →	x is returned to the form initiator.

Label Table Search

<u>Meaning</u>	<u>Mnemonic</u>	<u>Stack</u>	<u>Comment</u>
Find Label	LUL	x → ADx	The address of the numeric label x is put on the stack.

Data Loading

<u>Meaning</u>	<u>Mnemonic</u>	<u>Stack</u>	<u>Comment</u>
Load Contents	LIC	LDx → LIC(LDx)	The contents of x are put on the stack.
Load Value	LIV	LDx → LIV(LDx)	Value used to convert from decimal character to binary.
Load Length	LIL	LDx → LIL(LDx)	
Load Type	LIT	LDx → LIT(LDx)	

Debugging Aids

<u>Meaning</u>	<u>Mnemonic</u>	<u>Stack</u>	<u>Comment</u>
Term End	EOT	None	Used by the interpreter to indicate locations of form errors.
Rule End	EOR	None	

Appendix C

DRS COMPILER LISTINGS

COMPOOL: PROCEDURE:

```

/* VARIABLE CONTROLS TRACING OPTION.  TRACE=1 ==> TRACE */
/*                                     TRACE=0 ==> NO TRACE */
DCL TRACE BIN FIXED (31) STATIC FXT INITIAL (0);

/*  THESE ARE LALR PARSING TABLES  */
DCL MAXR# BIN FIXED (15) STATIC FXT INIT (55);
DCL MAXL# BIN FIXED (15) STATIC FXT INIT (90);
DCL MAXP# BIN FIXED (15) STATIC FXT INIT (104);
DCL START_STATE BIN FIXED (15) STATIC FXT INIT (56);
DCL READ1(373) BIN FIXED (15) STATIC FXT INIT
    (38,10,11,12,13,14,15,16,17,18,19,00000170
,20,21,23,24,25,26,27,28,38,40,10,11,12,13,14,15,16,17,18,19,20,21,23 00000180
,24,25,26,27,28,38,40,11,12,13,14,15,16,17,18,19,20,21,23,24,25,26,27 00000190
,28,40,11,12,13,14,15,16,17,18,19,20,21,23,24,25,26,27,28,40,11,12,13 00000200
,14,15,16,17,18,19,20,21,23,24,25,26,27,28,40,11,12,13,14,15,16,17,18 00000210
,19,20,21,23,24,25,26,27,28,38,40,1,11,12,13,14,15,16,17,18,19,20,21,23 00000220
,24,25,26,27,28,40,11,12,13,16,18,21,23,24,26,1,12,13,14,15,16,17,18 00000230
,19,20,21,23,24,25,26,27,28,38,40,14,17,19,25,27,28,11,12,13,14,15,16 00000240
,17,18,19,20,21,23,24,25,26,27,28,38,40,14,17,19,25,27,28,1,1,1,11,12 00000250
,13,14,15,16,17,18,19,20,21,23,24,25,26,27,28,38,40,11,12,13,14,15,16 00000260
,17,18,19,20,21,23,24,25,26,27,28,38,40,8,29,38,8,2,3,6,7,2,3,6,7,2,3,6 00000270
,7,2,3,4,6,7,2,3,6,7,1,11,12,13,14,15,16,17,18,19,20,21,23,24,25,26,27 00000280
,28,40,8,8,8,22,31,32,33,34,35,36,22,22,22,8,39,4,4,4,4,11,12,13,14,15 00000290
,16,17,18,19,20,21,23,24,25,26,27,28,38,40,9,1,11,12,13,14,15,16,17,18 00000300
,19,20,21,23,24,25,26,27,28,40,9,11,12,13,14,15,16,17,18,19,20,21,23,24 00000310
,25,26,27,28,38,40,9,9,1,30,4,4,4,9,1,5);
DCL LOOK1(83) BIN FIXED (15) STATIC FXT INIT
    (38,0,4,8,9,0,8,0,8,0,8,0,4,9,0 00000340
,39,0,39,0,39,0,1,0,39,0,1,0,39,0,39,0,39,0,29,38,0,8,0,2,3,6,7,8 00000350
,0,8,0,2,3,6,7,0,4,9,0,5,0,0,8,0,8,0,22,0,22,0,22,0,9,0,9,0,9,0,9,0,1,0 00000360
,30,0,9,0);
DCL APPLY1(54) BIN FIXED (15) STATIC FXT INIT
    (0,0,20,0,0,0,43,0,0,8,0,0,3,0 00000390
,44,46,47,0,0,0,27,2,8,43,4,5,6,0,0,0,0,0,0,45,18,10,0,0,2,3,7,12,0,9,0 00000400
,11,0,17,0,41,0,0,0,0);
DCL READ2(373) BIN FIXED (15) STATIC FXT INIT
    (109,147,62,63,64,132,65,66,135 00000430
,136,137,67,68,69,70,142,71,144,145,183,146,147,129,130,131,132,65,134 00000440
,135,136,137,67,139,140,141,142,143,144,145,183,146,129,130,131,132,133 00000450
,134,135,136,137,138,139,140,141,142,143,144,145,146,129,130,131,132 00000460
,133,134,135,136,137,138,139,140,141,142,143,144,145,146,129,130,131 00000470
,132,133,134,135,136,137,138,139,140,141,142,143,144,145,146,129,130 00000480
,131,132,65,134,135,136,137,67,139,140,1,1,142,143,144,145,183,146,57 00000490
,129,130,131,132,133,134,135,136,137,138,139,140,141,142,143,144,145 00000500
,146,173,169,172,170,15,171,175,174,176,62,63,64,132,65,66,135,136,137 00000510
,67,68,69,70,142,71,144,145,183,146,189,188,190,192,191,193,129,130,131 00000520
,132,65,134,135,136,137,67,139,140,141,142,143,144,145,183,146,189,188 00000530
,190,192,191,193,4,6,5,62,63,64,132,65,66,135,136,137,67,68,69,70,142 00000540
,71,144,145,183,146,62,63,64,132,65,66,135,136,137,67,68,69,70,142,71 00000550
,144,145,183,146,59,105,109,11,184,186,185,187,184,186,185,187,184,186 00000560
,185,187,184,186,177,185,187,184,186,185,187,57,129,130,131,132,133,134 00000570

```

```

,135,136,137,138,139,140,141,142,143,144,145,146,8,8,61,17,165,163,164 00000580
,161,162,166,17,17,17,60,178,121,120,119,117,129,130,131,132,65,134,13500000590
,136,137,67,139,140,141,142,143,144,145,183,146,13,57,129,130,131,132 00000600
,133,134,135,136,137,138,139,140,141,142,143,144,145,146,13,62,63,64 00000610
,132,65,66,135,136,137,67,68,69,70,142,71,144,145,183,146,13,13,58,18 00000620
,181,182,151,194,7,108); 00000630
DCL LOOK2(83) BIN FIXED (15) STATIC EXT INIT 00000640
(1,91,92,93,92,2,94,3,95,9,96,10 00000650
,97,97,12,173,129,169,130,172,131,14,133,170,134,16,138,171,139,175,1400000660
,174,141,176,143,20,20,98,21,157,22,22,22,22,148,179,148,23,24,24,24,2400000670
,179,155,155,26,99,99,27,28,111,29,115,32,126,33,128,34,153,42,100,44 00000680
,101,46,102,47,103,48,118,49,180,53,104); 00000690
DCL APPLY2(54) BIN FIXED (15) STATIC EXT INIT 00000700
(0,72,107,106,78,90,80,79,55,11400000710
,113,88,87,86,38,39,40,37,85,84,122,89,122,122,50,51,52,180,19,35,30 00000720
,123,124,81,82,83,31,45,74,75,25,77,76,150,36,158,73,160,159,168,167,4100000730
,54,43); 00000740
00000750
/* PUSH STATES ARE BUILT-IN TO THE INDEX TABLES */ 00000760
00000770
DCL INDEX1(194) BIN FIXED (15) STATIC EXT INIT 00000780
(1,2,22,42,60,78,96,115,134,14300000790
,162,168,187,193,194,195,196,215,234,235,237,238,242,246,250,255,259 00000800
,278,279,280,281,288,289,290,291,292,293,294,295,296,297,316,317,336 00000810
,337,356,357,358,359,360,361,362,363,364,365,1,3,7,9,11,13,16,18,20,22 00000820
,24,26,28,30,32,34,36,39,41,47,49,54,57,60,62,64,66,68,70,72,74,76,78 00000830
,80,82,110,127,149,149,152,154,156,110,112,125,125,125,125,116,1,2,2,3 00000840
,5,5,6,6,7,7,9,9,10,10,10,10,10,12,13,15,15,19,19,20,21,21,21,21,21 00000850
,21,21,21,21,21,21,21,21,21,21,21,21,21,29,29,29,30,30,30,31,31,32,32,33 00000860
,33,34,34,38,38,38,38,38,38,39,44,44,44,44,44,44,46,48,48,50 00000870
,50,50,50,52,52,52,52,53,53,53,53,53,53,54); 00000880
DCL INDEX2(194) BIN FIXED (15) STATIC EXT INIT 00000890
(1,20,20,1,18,18,19,19,9,19,6 00000900
,19,6,1,1,1,19,19,1,2,1,4,4,4,5,4,19,1,1,1,7,1,1,1,1,1,1,1,1,19,1,19 00000910
,1,19,1,1,1,1,1,1,1,1,1,1,2,4,2,2,2,3,2,2,2,2,2,2,2,2,2,2,2,3,2,6,2,5,3,300000920
,2,2,2,2,2,2,2,2,2,2,2,2,1,2,2,3,9,10,12,20,27,42,44,46,47,53,1,0,1,3,00000930
,0,0,0,0,2,1,0,4,0,3,3,3,3,0,6,1,0,2,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0 00000940
,0,0,0,0,0,0,0,3,0,0,0,0,0,0,0,2,0,2,0,0,0,0,0,0,0,0,0,0,2,0,0,0,0,0,3,1,0 00000950
,0,3,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 00000960
00000970
00000980
DCL TERMINAL# BIN FIXED (15) STATIC EXT INIT (40); 00000990
DCL VOCAB# BIN FIXED (15) STATIC EXT INIT (69); 00001000
DCL P# BIN FIXED (15) STATIC EXT INIT (90); 00001010
END COMPOOL; 00001020

```

```

PRSER: PROCEDURE (PRM) OPTIONS(MAIN):
/* VARIABLES HOLD PARM FROM EXEC CARD */
DCL PRM CHAR (100) VAR:
DCL PARM CHAR (100) VAR FXT STATIC:
/* TABLES OUTPUT FROM PARSER GENERATOR */
DCL MAXR# BIN FIXED (15) STATIC EXT:
DCL AX_# BIN FIXED (15) STATIC EXT:
DCL MAXP# BIN FIXED (15) STATIC EXT:
DCL START_STATE BIN FIXED (15) STATIC EXT:
DCL READ1(373) BIN FIXED (15) STATIC EXT:
DCL LOOK1(83) BIN FIXED (15) STATIC EXT:
DCL APPLY1(54) BIN FIXED (15) STATIC EXT:
DCL READ2(373) BIN FIXED (15) STATIC EXT:
DCL LOOK2(83) BIN FIXED (15) STATIC EXT:
DCL APPLY2(54) BIN FIXED (15) STATIC EXT:
DCL INDEX1(194) BIN FIXED (15) STATIC EXT:
DCL INDEX2(194) BIN FIXED (15) STATIC EXT:
/* ROUTINES INVOKED BY PRSER */
DCL DISPLAY ENTRY (CHAR (72) VAR):
DCL LXANLZ ENTRY RETURNS(BIN FIXED (15)):
DCL SMNTC ENTRY (BIN FIXED) RETURNS (BIN FIXED (15)):
DCL SMFCIO ENTRY ((2) BIN FIXED (31), BIN FIXED (31),
(2) BIN FIXED (31)):
DCL SMFCIO ENTRY ((2) BIN FIXED (31)):
DCL SMFLIO ENTRY :
/* TABLES FOR SMFS */
DCL CMPLT1(2) BIN FIXED (31) ALIGNED STATIC EXT INITIAL (1,1):
DCL CMPLT2(2) BIN FIXED (31) ALIGNED STATIC EXT INITIAL (2,2):
DCL CMPLT3(2) BIN FIXED (31) ALIGNED STATIC EXT INITIAL (3,3):
DCL CMPLT4(2) BIN FIXED (31) ALIGNED STATIC EXT INITIAL (4,4):
DCL LCLSKT1 BIN FIXED (31) ALIGNED STATIC INITIAL (4098):
DCL LCLSKT2 BIN FIXED (31) ALIGNED STATIC INITIAL (4102):
DCL LCLSKT3 BIN FIXED (31) ALIGNED STATIC INITIAL (4106):
DCL LCLSKT4 BIN FIXED (31) ALIGNED STATIC INITIAL (4110):
DCL WRKSPS1(2) BIN FIXED (31) ALIGNED STATIC EXT INITIAL ((2)0) :
DCL WRKSPS2(2) BIN FIXED (31) ALIGNED STATIC EXT INITIAL ((2)0):
DCL WRKSPS3(2) BIN FIXED (31) ALIGNED STATIC EXT INITIAL ((2)0):
DCL WRKSPS4(2) BIN FIXED (31) ALIGNED STATIC EXT INITIAL ((2)0):
DCL 1 CR8DIAG STATIC UNALIGNED EXT,
2 OPCD BIT (8) INITIAL ('00000010'B),
2 FLGS BIT (16) INITIAL ('0000000001000000'B),
2 NLNG BIT (8) INITIAL ('00100100'B),

```

2 FNAM CHAR (36) INITIAL (''),	00000580
2 FLNG BIN FIXED (31) INITIAL (88000);	00000590
	00000600
DCL 1 CR8OBJI STATIC UNALIGNED FXT,	00000610
2 OPCD BIT (8) INITIAL ('00000010'B),	00000620
2 FLGS BIT (16) INITIAL (0),	00000630
2 NLNG BIT (8) INITIAL ('00100100'B),	00000640
2 FNAM CHAR (36) INITIAL (''),	00000650
2 FLNG BIN FIXED (31) INITIAL (3300);	00000660
	00000670
DCL 1 CR8OBJT STATIC UNALIGNED FXT,	00000680
2 OPCD BIT (8) INITIAL ('00000010'B),	00000690
2 FLGS BIT (16) INITIAL (0),	00000700
2 NLNG BIT (8) INITIAL ('00100100'B),	00000710
2 FNAM CHAR (36) INITIAL (''),	00000720
2 FLNG BIN FIXED (31) INITIAL (48000);	00000730
	00000740
DCL 1 RDSRCH STATIC UNALIGNED FXT,	00000750
2 OPCD BIT (8) INITIAL ('00000101'B),	00000760
2 FLGS BIT (16) INITIAL (0),	00000770
2 NLNG BIT (8) INITIAL ('00100100'B),	00000780
2 FNAM CHAR (36) INITIAL (''),	00000790
2 DLNG BIN FIXED (31) INITIAL (0);	00000800
	00000810
DCL 1 WRTORBT1 STATIC FXT,	00000820
2 OPCD BIT (8) UNALIGNED INITIAL ('00000011'B),	00000830
2 FLGS BIT (16) UNALIGNED INITIAL (0),	00000840
2 NLNG BIT (8) UNALIGNED INITIAL ('00100100'B),	00000850
2 FNAM CHAR (36) UNALIGNED INITIAL (''),	00000860
2 DLNG BIN FIXED (31) ALIGNED INITIAL (0),	00000870
2 BYTLBT BIN FIXED (15) ALIGNED INITIAL (0),	00000880
2 LBTN3(0:199),	00000890
3 LRLVLU BIN (15) FIXED ALIGNED INITIAL ((200)0),	00000900
3 LRLQST BIN (15) FIXED ALIGNED INITIAL ((200)0);	00000910
	00000920
DCL 1 WRTORBTI STATIC FXT,	00000930
2 OPCD BIT (8) UNALIGNED INITIAL ('00000011'B),	00000940
2 FLGS BIT (16) UNALIGNED INITIAL (0),	00000950
2 NLNG BIT (8) UNALIGNED INITIAL ('00100100'B),	00000960
2 FNAM CHAR (36) UNALIGNED INITIAL (''),	00000970
2 DLNG BIN FIXED (31) ALIGNED INITIAL (0),	00000980
2 BYTLCDT BIN FIXED (15) ALIGNED INITIAL (0),	00000990
2 CODE(0:1999) BIN FIXED (15) ALIGNED INITIAL ((2000)0);	00010000
	00010100
DCL 1 DLTFILF STATIC UNALIGNED FXT,	00010120
2 OPCD BIT (8) INITIAL ('00000111'B),	00010130
2 FLGS BIT (16) INITIAL (0),	00010140
2 NLNG BIT (8) INITIAL ('00100100'B),	00010150
2 FNAM CHAR (36) INITIAL (''),	00010160
2 FLNG BIT (32) INITIAL (0);	00010170
	00010180
	00010190
	00011100
	00011110
/* SHARED VARIABLES */	00011120
	00011130
DCL STACKS17 BIN FIXED STATIC FXT INITIAL (40);	00011140

```

DCL STATE_STACK(40) BIN FIXED (15) STATIC INITIAL ((40)0);
DCL RCDVLU CHAR(256) VAR FXT STATIC INITIAL ('');
DCL SYMBOL(0:1) CHAR (256) VAR STATIC FXT INIT ((2)''');
DCL (IC,ICO) BIN FIXED STATIC FXT;
DCL TRACE BIN FIXED (31) STATIC FXT;

/* LOCAL VARIABLES */

DCL STATE BIN STATIC FIXED INITIAL(0);
DCL READIT BIN STATIC FIXED INITIAL(1);
DCL SP BIN STATIC FIXED INITIAL (0);
DCL MP BIN STATIC FIXED;
DCL TOKEN BIN STATIC FIXED;
DCL (I,J) BIN FIXED STATIC;
DCL ERROR BIN FIXED STATIC INITIAL (0);

/* INITIALIZE AT THIS POINT*/

/* MAKE PARM FROM EXEC CAR' AVAILABLE TO ALL ROUTINES */

PARM = PRM;

/* INITIALIZE FILES */

RDSRCF.FNAM = 'DRS SRCF '||PARM;
CR8DIAG.FNAM = 'DRS DIAG '||PARM;
CR8OBJI.FNAM = 'DRS OBJI '||PARM;
CR8OBJT.FNAM = 'DRS OBJT '||PARM;
CALL SMFCIO(CMPLT1,LCLSKT1,WRKSPS1);
IF CMPLT1(1):CMPLT1(2) = 0 THEN GO TO ABORT;
CALL SMFCIO(CMPLT2,LCLSKT2,WRKSPS2);
CALL SMFCIO(CMPLT3,LCLSKT3,WRKSPS3);
CALL SMFCIO(CMPLT4,LCLSKT4,WRKSPS4);
DLTFILF.FNAM = CR8DIAG.FNAM;
CALL SMFLIO(CMPLT2,DLTFILF);
DLTFILF.FNAM = CR8OBJI.FNAM;
CALL SMFLIO(CMPLT3,DLTFILF);
DLTFILF.FNAM = CR8OBJT.FNAM;
CALL SMFLIO(CMPLT4,DLTFILF);
CALL SMFLIO(CMPLT1,RDSRCF);
CALL SMFLIO(CMPLT2,CR8DIAG);
CALL SMFLIO(CMPLT3,CR8OBJI);
CALL SMFLIO(CMPLT4,CR8OBJT);
WRTOBJI.FNAM = CR8OBJI.FNAM;
WRTOBJT.FNAM = CR8OBJT.FNAM;

/* START COMPILE LOOP */

READIT = .
RT:
SP = 0;
STATE = START_STATE;

DO WHILE ('')B);

```

IF STATE <= MAXR# THEN	00001720
DN;	00001730
SP = SP+1;	00001740
IF SP >= STACKS17 THEN GO TO ENDIT;	00001750
STATE_STACK(SP) = STATE;	00001760
I = INDEX1(STATE);	00001770
IF READIT = 1 THEN	00001780
DN;	00001790
SYMBOL(0) = SYMBOL(1);	00001800
TOKEN = LXANLZ;	00001810
SYMBOL(1) = BCDVLU;	00001820
READIT = 0;	00001830
IF TOKEN < 0 THEN GO TO LXLAERR;	00001840
END;	00001850
DN I = 1 TO I+INDEX2(STATE)-1;	00001860
IF READ1(I) = TOKEN THEN	00001870
DN;	00001880
STATE = READ2(I);	00001890
SYMBOL(0) = SYMBOL(1);	00001900
READIT = 1;	00001910
GO TO COMP;	00001920
END;	00001930
END;	00001940
LXLAERR:	00001950
CALL DISPLAY(	00001960
'** ERROR IN TEXT BEGINNING '' BCDVLU '', **');	00001970
SMNTCFERR:	00001980
CALL DISPLAY(	00001990
'** THE RULE CONTAINING THE ERROR IS IGNORED **');	00002000
ERROR = 1;	00002010
ICO = IC;	00002020
SKIP:	00002030
READIT = 1;	00002040
SKIPMORE:	00002050
IF TOKEN <= 0 TOKEN = 29 THEN GO TO CLOSEOUT;	00002060
SYMBOL(1) = BCDVLU;	00002070
IF READIT = 0 THEN GO TO RESTART;	00002080
IF TOKEN = 5 THEN READIT = 0;	00002090
TOKEN = LXANLZ;	00002100
GO TO SKIPMORE;	00002110
END;	00002120
ELSE	00002130
/*TEST FOR APPLY STATE*/	00002140
IF STATE > MAXP# THEN	00002150
DN;	00002160
MP = SP-INDEX2(STATE);	00002170
BCDVLU = SYMBOL(0);	00002180
IF SMNTC(STATE-MAXP#) < 0 THEN GO TO SMNTCFERR;	00002190
SP = MP;	00002200
I = INDEX1(STATE);	00002210
J = STATE_STACK(SP);	00002220
DN WHILE (APPLY1(I) /= 0);	00002230
IF J = APPLY1(I) THEN GO TO TOP_MATCH;	00002240
I = I+1;	00002250
END;	00002260
TOP_MATCH:	00002270
IF APPLY2(I) = 0 THEN GO TO CLOSEOUT;	00002280


```

STATE = APPLY2(I);
END:
ELSE
/*TEST FOR LOOK STATE*/
IF STATE <= MAXL# THEN
DO:
I = INDEX1(STATE);
IF READIT = 1 THEN
DO:
SYMBOL(0) = SYMBOL(1);
TOKEN = LXLNLZ;
SYMBOL(1) = BCDVLU;
READIT = 0;
IF TOKEN < 0 THEN GO TO LXLAFRR;
END:
DO WHILE (LOOK1(I) /= 0);
IF LOOK1(I) = TOKEN THEN GO TO LOOK:
I = I+1;
END:
LOOK:
STATE = LOOK2(I);
END:
ELSE
/*MUST BE PUSH STATE*/
DO:
SP = SP+1;
IF SP >= STACKSIZ THEN GO TO ENDIT:
STATE_STACK(SP) = INDEX2(STATE);
STATE = INDEX1(STATE);
END:
END:

/* CLOSE OUT COMPILER */

ENDIT:
CALL DISPLAY('STATE STACK OVERFLOW. MAX IS'||STACKSIZ);
CLOSEOUT:
CALL DISPLAY('COMPILATION TERMINATED');
IF ERROR /= 0 THEN
DO:
CALL DISPLAY(
'THE INSTRUCTION, LABEL, AND IDENTIFIER/LITERAL TABLES WILL NOT');
CALL DISPLAY(
'RE WRITTEN BECAUSE OF THE PREVIOUSLY NOTED ERRORS. ');
DLTFILE.FNAM = CR80BJI.FNAM;
CALL SMFLIO(CMPLT3,DLTFILE);
DLTFILE.FNAM = CR80BJT.FNAM;
CALL SMFLIO(CMPLT4,DLTFILE);
END:
ELSE
DO:
CALL SMFCIO(CMPLT3);
CALL SMFCIO(CMPLT4);
END:
CALL SMFCIO(CMPLT2);
RETURN:

```

00002290
00002300
00002310
00002320
00002330
00002340
00002350
00002360
00002370
00002380
00002390
00002400
00002410
00002420
00002430
00002440
00002450
00002460
00002470
00002480
00002490
00002500
00002510
00002520
00002530
00002540
00002550
00002560
00002570
00002580
00002590
00002600
00002610
00002620
00002630
00002640
00002650
00002660
00002670
00002680
00002690
00002700
00002710
00002720
00002730
00002740
00002750
00002760
00002770
00002780
00002790
00002800
00002810
00002820
00002830
00002840
00002850

ABORT:

CALL DISPLAY('NO INPUT AVAILABLE. CAN'T COMPILE');
END PRSR:

00002860

00002870

00002880

```

LXANLZ : PROCEDURE RETURNS( BINARY FIXED (15)):                                00000010
/* SHARED VARIABLES */                                                         00000020
DCL INPUT CHAR (84) VAR ALIGNED STATIC EXT;                                  00000030
DCL TRACE BIN FIXED (31) STATIC EXT :                                         00000040
DCL TERMINAL# BIN FIXED (15) STATIC EXT;                                       00000050
DCL RCDVLU CHAR(256) VAR STATIC EXT INITIAL ('');                             00000060
DCL LINESZ BIN FIXED (15) STATIC EXT;                                           00000070
DCL LINE CHAR (332) VAR ALIGNED STATIC EXT;                                    00000080
/* ROUTINES INVOKED BY LXANLZ */                                              00000090
DCL DISPLAY ^ TRY (CHAR (72) VAR):                                             00000100
DCL READ ENTRY RETURNS(BIN FIXED (15)):                                         00000110
DCL INDX ENTRY (CHAR(256) VAR) RETURNS(BIN FIXED (15)):                       00000120
DCL VREF ENTRY (CHAR(256) VAR) RETURNS(BIN FIXED (15)):                       00000130
/* LOCAL VARIABLES */                                                         00000140
DCL CHRTP CHAR (50) STATIC                                                      00000150
INITIAL (' (+*):-/,:#".1ABCDEFGHIJKLMNPQRSTUVWXYZ0123456789');               00000160
DCL NDX(5) BIN FIXED STATIC INITIAL                                           00000170
(1,85,113,117,145);                                                            00000180
DCL LGTH BIN FIXED (15) STATIC INITIAL (0);                                     00000190
DCL VOCAB CHAR (144) STATIC ALIGNED EXT INITIAL                               00000200
('(' + * ) : - / , : # A B F F L O S T                                     00000210
U V X || AD ED ER SB SR UR _|_ .<=..EQ..GF..GT..LF..LT..N0000220
F.);                                                                            00000230
DECLARE RESULT BIN FIXED STATIC :                                             00000240
DECLARE (TEMP1,TEMP2) BIN FIXED STATIC :                                       00000250
DCL RCDLMTR CHAR(2) STATIC INIT ('/*');                                       00000260
DCL ECMDLTR CHAR(2) STATIC INIT ('/*');                                       00000270
DCL LTOMLTR CHAR(1) STATIC INIT ('');                                         00000280
DCL BLNK84 CHAR (84) STATIC INITIAL (' ');                                    00000290
/* BEGIN LEXICAL ANALYSIS */                                                  00000300
RESTART:                                                                        00000310
TEMP1 = VREF(' ');                                                             00000320
IF TEMP1 = 0 THEN GO TO EOF;                                                    00000330
RESULT = INDEX(CHRTP,SUBSTR(INPUT,1,1));                                       00000340
IF RESULT=0 THEN GO TO RESTART;                                                 00000350
IF RESULT<15 THEN GO TO LITRL;                                                 00000360
IF RESULT<=40 THEN TEMP1=15;                                                   00000370
ELSE TEMP1 = 41;                                                                00000380
TEMP2 = VREF(SUBSTR(CHRTP,TEMP1));                                              00000390
RESULT=38;                                                                      00000400
IF TEMP1=41 THEN GO TO FORM;                                                    00000410
RESULT=40;                                                                      00000420
IF LGTH>2 THEN GO TO FORM;                                                      00000430
TEMP2 = NDX(LGTH);                                                             00000440
TEMP1 = INDEX(SUBSTR(VOCAB,TEMP2,NDX(LGTH+1)-TEMP2),RCDVLU);                 00000450
IF TEMP1 /= 0 THEN RESULT = (TEMP1+TEMP2+2)/4;                                00000460

```

GO TO FORM:	00000580
LITRL:	00000590
IF RESULT=12 THEN GO TO LTSTR:	00000600
IF SUBSTR(INPUT,1,2) = RCMDLMTR THEN GO TO COMMENT:	00000610
LNTH = 1:	00000620
IF RESULT = 13 THEN LNTH = 4:	00000630
IF RESULT = 14 THEN LNTH = 2:	00000640
TFMP2 = NOX(LNTH):	00000650
TEMP1 = INDEX(SUBSTR(VOCAB,TEMP2,NOX(LNTH+1)-TFMP2),	00000660
SUBSTR(INPUT,1,LNTH)):	00000670
IF TEMP1 = 0 THEN RESULT = (TFMP1+TFMP2+2)/4:	00000680
ELSE	00000690
DO:	00000700
RESULT = -1:	00000710
LNTH = 1:	00000720
END:	00000730
TEMP1 = INDX(SUBSTR(INPUT,1,LNTH)):	00000740
FORM:	00000750
LINE = LINE RCVDLU:	00000760
GO TO COVER:	00000770
COMMENT:	00000780
RESULT = -1:	00000790
INPUT = SUBSTR(INPUT,3):	00000800
TEMP1 = INDX(RCMDLMTR):	00000810
LINE = LINE RCMDLMTR RCVDLU:	00000820
IF TEMP1 = 0 THEN GO TO COVER:	00000830
RESULT = 100:	00000840
GO TO COVER:	00000850
LTSTR:	00000860
RESULT = -1:	00000870
INPUT = SUBSTR(INPUT,2):	00000880
TEMP1 = INDX(LTDLMT):	00000890
LINE = LINE LTDLMT RCVDLU:	00000900
IF TEMP1 = 0 THEN GO TO COVER:	00000910
RESULT=39:	00000920
RCVDLU = SUBSTR(RCVDLU,1,LNTH-1):	00000930
COVER:	00000940
LINE = LINE ' ':	00000950
IF LENGTH(LINE) > LINESZ THEN	00000960
DO:	00000970
CALL DISPLAY(SUBSTR(LINE,1,LINESZ)):	00000980
LINE = SUBSTR(LINE,LINESZ+1):	00000990
GO TO COVER:	00010000
END:	00010010
IF RESULT = 100 THEN GO TO RESTART:	00010020
IF RESULT = 5 RESULT = 29 THEN	00010030
DO:	00010040
CALL DISPLAY(LINE):	00010050
LINE = '':	00010060
END:	00010070
RETURN(RESULT):	00010080
END:	00010090
RESULT = 29:	00010100
RCVDLU = '':	00010110
LINE = LINE '/* END OF FORM */':	00010120
GO TO COVER:	00010130
	00010140

VRFY: PROCEDURE(TEXT) RETURNS(BIN FIXED (15));	00001150
DCL TEXT CHAR (256) VAR;	00001160
DCL (TEMP1,TEMP2,TEMP3) BIN FIXED (15) STATIC;	00001170
	00001180
	00001190
/* GIVE A TARGET STRING TO SEARCH FOR OR VERIFY */	00001200
/* ON RETURN: INPUT CONTAINS THE STRING THAT FOLLOWS THE */	00001210
/* TARGET--INCLUDING THE TARGET (OR STRING NOT IN */	00001220
/* THE VERIFY LIST. BCDVLU CONTAINS THE SCANNED */	00001230
/* STRING--NOT INCLUDING THE NON-VERIFIED */	00001240
/* SYMBOL. THE FUNCTION VALUE WILL BE ZERO IF THE */	00001250
/* INPUT RAN OUT OR THE POSITION OF THE TARGET OR */	00001260
/* NON-VERIFIED STRING, 1, IF IT DID NOT. */	00001270
/* FUNCTION VALUE WILL BE ZERO IF THE INPUT RAN OUT */	00001280
/* OR THE POSITION OF THE TARGET OR NON-VERIFIED */	00001290
/* STRING. */	00001300
	00001310
TEMP3 = 1;	00001320
GO TO SCAN;	00001330
INDX: ENTRY(TEXT) RETURNS(BIN FIXED (15));	00001340
TEMP3 = 2;	00001350
SCAN:	00001360
BCDVLU = '';	00001370
MORE:	00001380
IF TEMP3 = 1 THEN TEMP2 = VERIFY(INPUT,TEXT);	00001390
ELSE TEMP2 = INDEX(INPUT,TEXT);	00001400
IF TEMP2 = 0 THEN	00001410
DO;	00001420
BCDVLU = BCDVLU INPUT;	00001430
INPUT = BLNK84;	00001440
IF READ ^= 0 THEN GO TO MORE;	00001450
END;	00001460
ELSE	00001470
DO;	00001480
IF TEMP3 = 2 THEN TEMP2 = TEMP2+LENGTH(TEXT);	00001490
IF TEMP2 > 1 THEN	00001500
DO;	00001510
BCDVLU = BCDVLU SUBSTR(INPUT,1,TEMP2-1);	00001520
INPUT = SUBSTR(INPUT,TEMP2);	00001530
TEMP2 = 1;	00001540
END;	00001550
END;	00001560
LENGTH = LENGTH(BCDVLU);	00001570
RETURN(TEMP2);	00001580
END VRFY;	00001590
END LXANLZ;	00001600

```
SMNTC: PROCEDURE(COMPNT) RETURNS (BIN FIXED (15));
    DCL COMPNT BIN FIXED (15) ;

/* PARAMETERS FOR TABLE SIZES */

DCL LBLFLR BIN FIXED (15) STATIC FXT INIT (0);
DCL LBLCLNG BIN FIXED (15) STATIC FXT INIT (9999);
DCL IDLNGTH BIN FIXED (15) STATIC FXT INIT (4);
DCL MXSTKSZ BIN FIXED (15) STATIC FXT INIT (15);
DCL MXINSTS BIN FIXED (15) STATIC FXT INIT (2000);
DCL MXLRLS BIN FIXED (15) STATIC FXT INIT (200);
DCL MXNDXS BIN FIXED (15) STATIC E INIT (256);
DCL MXIDS BIN FIXED (15) STATIC FXT INIT (512);
DCL MXLITS BIN FIXED (15) STATIC FXT INIT (2000);

/* SHARED VARIABLES */

DCL TRACE BIN FIXED (31) STATIC FXT;
DCL TERMINAL# BIN FIXED (15) STATIC FXT;
DCL VOCAB# BIN FIXED (15) STATIC FXT;
DCL P# BIN FIXED (15) STATIC FXT;
DCL ICO BIN FIXED STATIC FXT INITIAL (-1);
DCL IC BIN FIXED STATIC FXT INITIAL (-1);
DCL NMLRS BIN FIXED STATIC FXT INITIAL (-1);
DCL NMIDS BIN FIXED STATIC FXT INITIAL (-1);
DCL LTEND BIN FIXED STATIC FXT INITIAL (0);
DCL NMNDXS BIN FIXED STATIC FXT INITIAL (-1);
DCL RCDVLU CHAR(256) VAR FXT;

/* ROUTINES INVOKED BY SMNTC */

DCL SPUCODE ENTRY;
DCL SMFLIO ENTRY;
DCL SMFSIO ENTRY;
DCL DISPLAY ENTRY (CHAR(72) VAR);
DCL FINDLT ENTRY (CHAR(256) VAR, BIN FIXED (15))
    RETURNS(BIN FIXED (15));
DCL FINDID ENTRY (CHAR(4) VAR) RETURNS(BIN FIXED (15));
DCL FINDLR ENTRY (BIN FIXED (15)) RETURNS(BIN FIXED (15));
DCL GNRTR ENTRY (BIN FIXED (15)) RETURNS (BIN FIXED (15));
DCL GETIDNT ENTRY RETURNS (BIN FIXED (15));

/* VARIABLES FOR SMFS */

DCL CMPLT3(2) BIN FIXED (31) ALIGNED STATIC FXT;
DCL CMPLT4(2) BIN FIXED (31) ALIGNED STATIC FXT;
DCL 1 WRTORJI STATIC FXT,
    2 OPCD BIT (8) UNALIGNED,
    2 FLGS BIT (16) UNALIGNED,
    2 NLNG BIT (8) UNALIGNED,
    2 FNAM CHAR (36) UNALIGNED,
    2 DLNG BIN FIXED (31) ALIGNED,
```

```

2 BYTLCNT BIN FIXED (15) ALIGNED,
2 CODE(0:1999) BIN FIXED (15) ALIGNED:
DCL 1 WRTORBJT1 STATIC EXT,
2 OPCODE BIT (8) UNALIGNED,
2 FLGS BIT (16) UNALIGNED,
2 NLNG BIT (8) UNALIGNED,
2 FNAM CHAR (36) UNALIGNED,
2 DLNG BIN FIXED (31) ALIGNED,
2 BYTLBT BIN FIXED (15) ALIGNED,
2 LBTN3(0:199),
    3 LBLVLU BIN FIXED (15) ALIGNED INITIAL ((200)0),
    3 LBLOFST BIN FIXED (15) ALIGNED INITIAL ((200)0);
DCL 1 WRTORBJT2 STATIC EXT,
2 DUMMY BIT (8) ALIGNED INITIAL (0),
2 OPCODE BIT (8) UNALIGNED INITIAL ('000000011'B),
2 FLGS BIT (16) UNALIGNED INITIAL ('0010000000000000'B),
2 DLNG BIT (32) UNALIGNED INITIAL (0),
2 BYTLDT BIN FIXED (15) ALIGNED,
2 BYTLTT BIN FIXED (15) ALIGNED,
2 IDTN3(0:511),
    3 IDTYPE BIN FIXED (15) ALIGNED INITIAL ((512)0),
    3 IDLNG BIN FIXED (15) ALIGNED INITIAL ((512)0),
    3 IDOFST BIN FIXED (15) ALIGNED INITIAL ((512)0);
DCL 1 WRTORBJT3 STATIC EXT,
2 DUMMY BIT (8) ALIGNED INITIAL (0),
2 OPCODE BIT (8) UNALIGNED INITIAL ('000000011'B),
2 FLGS BIT (16) UNALIGNED INITIAL ('0010000000000000'B),
2 DLNG BIT (32) UNALIGNED INITIAL (0),
2 LTRLS CHAR (2000) ALIGNED INITIAL ('');
/* LOCAL VARIABLES */
DCL REPLY BIN FIXED (15) STATIC INITIAL (0);
DCL INPUTRM BIN FIXED (15) STATIC INIT (1);
DCL SICP BIN FIXED (15) STATIC INIT (0);
DCL PROCTN BIN FIXED (15) STATIC INIT (0);
DCL ALTRN8 BIN FIXED (15) STATIC INIT (0);
DCL RCNTR BIN FIXED (15) STATIC INIT (0);
DCL TCNTR BIN FIXED (15) STATIC INIT (0);
DCL ICL BIN FIXED (15) STATIC INIT (-1);
DCL IDNAME(0:255) CHAR(4) UNALIGNED STATIC EXT INIT ((256)(4)''');
DCL IDSTK(16) BIN FIXED (15) STATIC INITIAL ((16)0);
DCL IDSP BIN FIXED (15) STATIC INITIAL (0);
DCL LABEL BIN FIXED (15) STATIC INITIAL (0);
DCL (1,TEMP) BIN FIXED (15) STATIC;
DCL TEMP32 BIN FIXED (31) STATIC;
DCL SROUT(80) LABEL;
DCL ALT BIN FIXED (15) STATIC INITIAL (0);
DCL LTRNTKN(30) BIN FIXED (15) STATIC EXT INIT ((30)0);
DCL DFTYPE (96) BIN FIXED (15) STATIC EXT INIT
(+1,
-2,-2,
+3,

```

+4,+4,	00001150
+5,+5,	00001160
+6,+6,	00001170
+7,+7,	00001180
+8,+8,+8,-8,-8,	00001190
-9,	00001200
+10,	00001210
-11,-11,	00001220
12,-12,	00001230
+13,	00001240
+14,+14,+14,+14,+14,+14,+14,+14,+14,+14,+14,+14,+14,+14,+14,	00001250
+14,+14,	00001260
+15,-15,+15,	00001270
+16,+16,+16,	00001280
-17,+17,	00001290
-18,+18,	00001300
-19,-19,	00001310
-20,+20,	00001320
-21,-21,-21,-21,-21,-21,	00001330
-22,+22,	00001340
-23,-23,-23,-23,-23,-23,-23,-23,	00001350
+24,	00001360
+25,-25,	00001370
+26,+26,+26,+26,	00001380
-27,-27,-27,-27,	00001390
+28,+28,-28,+28,+28,-28,	00001400
+29):	00001410
DCL COMPRS(6) BIN FIXED (15) STATIC INITIAL	00001420
(2232,2233,2235,2234,2230,2231);	00001430
DCL HOLD(10) BIN FIXED STATIC INITIAL	00001440
(0,0,0,4,0,0,0,0,0,0);	00001450
REPLY = 0;	00001460
TEMP = DTYPE(COMPNT);	00001470
LTRNTKN(ABS(TEMP)) = COMPNT;	00001480
IF TEMP <= 0 THEN RETURN(0);	00001490
IC1 = IC0;	00001500
IF TEMP = 14 THEN GO TO SROUT(25);	00001510
IF TEMP = 28 THEN GO TO SROUT(84);	00001520
GO TO SROUT(COMPNT);	00001530
EXIT:	00001540
IC0=IC1;	00001550
NOOP:	00001560
RETURN(REPLY);	00001570
ERROR:	00001580
REPLY = -3;	00001590
IC0 = IC;	00001600
IF RCNTR > 0 THEN RCNTR = 0;	00001610
IF TCNTR > 0 THEN TCNTR = TCNTR-1;	00001620
RETURN(REPLY);	00001630
	00001640
	00001650
/* DEFINED TYPE 1	*/00001660
/* 1 GLUMP ::= FORM _I_	*/00001670
	00001680
SROUT(1):	00001690
CALL TABLES:	00001700
DO I = 0 TO NMIDS:	00001710


```

IF IDTYPE(I) = 0 THEN IDOFST(I) = 0;
END;
BYTLCDT = (IC+1)*2;
WRTOBJ1.DLNG = (8*BYTLCDT)+16;
CALL SMFLIO(CMPLT3,WRTOBJ1);
BYTLLBT = 4*(NMLBS+1);
WRTOBJT1.DLNG = (8*BYTLLBT)+16;
CALL SMFLIO(CMPLT4,WRTOBJT1);
BYTLINT = 6*(NMIDS+1);
TEMP32 = (8*BYTLINT)+32;
WRTOBJT2.DLNG = UNSPEC(TEMP32);
CALL SMFSIO(CMPLT4,WRTOBJT2);
BYTLLTT = LTEND+1;
TEMP32 = 8*BYTLLTT;
WRTOBJT3.DLNG = UNSPEC(TEMP32);
CALL SMFSIO(CMPLT4,WRTOBJT3);
GO TO EXIT;

/* DEFINED TYPE 2
/*      2      FORM ::= RULE
/*      3      | FORM RULE

/* SROUT(2): */
/* SROUT(3): */

/* DEFINED TYPE 3
/*      4      RULE ::= LABEL INPUTSTREAM OUTPUTSTREAM ;

SROUT(4):
    INPUTRM = 1;
    TCNTR = 0;
    CALL GNRTR(7000+RCNTR);
    RCNTR = RCNTR+1;
    TEMP = 3000+IC1+1;
    DO I = IC+1 TO IC1;
    IF CODE(I) = 2130 THEN CODE(I) = TEMP;
    END;
    ICO = I
    CALL SP...DE;
    LTRNTKN(4) = 0;
    LTRNTKN(5) = 0;
    LTRNTKN(6) = 0;
    GO TO EXIT;

/* DEFINED TYPE 4
/*      5      LABEL ::= INTEGER
/*      6      |

SROUT(5):
    HOLD(2) = BINARY(BCDVLU);
    TEMP = NMLBS;
    LABEL=FINDLB(HOLD(2));
    IF TEMP = NMLBS THEN
    DO;

```

```

CALL DISPLAY(PCDVLUI)' IS A DOUBLY DEFINED LABEL';
GO TO ERROR;
END;
IF LABEL >= MXLRLS THEN GO TO ERROR;
IF HOLD(2) < LRLFLR | HOLD(2) > LRLCLNG THEN
DO;
CALL DISPLAY(
'LABEL' || PCDVLUI' IS NOT >= ' || LRLFLR || ' OR <= ' || LRLCLNG);
GO TO ERROR;
END;

SROUT(6):
IF INPUTRM = 1 THEN CALL GNRTR(2241);
GO TO EXIT;

/* DEFINED TYPE 5
/*      7  INPUTSTREAM ::= TERMS
/*      8      |
*/00002450
*/00002460
*/00002470
00002480
SROUT(7):
00002490
SROUT(8):
00002500
GO TO SROUT(12);
00002510
00002520
00002530
/* DEFINED TYPE 6
/*      9  TERMS ::= TERM
/*     10      | TERMS , TERM
*/00002540
*/00002550
*/00002560
00002570
SROUT(9):
00002580
SROUT(10):
00002590
CALL GNRTR(6000+TCNTR);
00002600
TCNTR = TCNTR+1;
00002610
IDSP = 0;
00002620
DO I = 7 TO VOCAB#-TERMINAL#;
00002630
LTRNTKN(I)=0;
00002640
END;
00002650
DO I = 1 TO 10;
00002660
HOLD(I) = 0;
00002670
END;
00002680
GO TO EXIT;
00002690
00002700
00002710
/* DEFINED TYPE 7
*/00002720
/*     11  OUTPUTSTREAM ::= SEPERATOR TERMS
*/00002730
/*     12      |
*/00002740
00002750
SROUT(11):
00002760
SROUT(12):
00002770
INPUTRM = 0;
00002780
GO TO EXIT;
00002790
00002800
00002810
/* DEFINED TYPE 8
*/00002820
/*     13  TERM ::= IDENTIFF ( DESCRIPTOR CONTROL )
*/00002830
/*     14      | IDENTIFF
*/00002840
/*     15      | ( DESCRIPTOR CONTROL )
*/00002850

```

```

/*          16          | ( COMPAREEXPR CONTROL )          */00002860
/*          17          | ( ASSGNEXPR CONTROL )          */00002870
SROUT(13):
  GO TO SROUT(15);
SROUT(14):
  CALL GNRTR(5000);
  CALL GNRTR(IDSTK(1));
  CALL GNRTR(2112);
  CALL GNRTR(IDSTK(1));
  CALL GNRTR(IDSTK(1));
  CALL GNRTR(2111);
  IF INPUTRM = 0 THEN CALL GNRTR(2260);
  ELSE
    CALL GNRTR(2251);
SROUT(15):
  CALL GNRTR(2130);
  CALL GNRTR(2221);
  GO TO SROUT(17);
/* SROUT(16): */
/* SROUT(17): */
/* DEFINED TYPE 9
/*          18          IDENTIFIE ::= IDENTIFIER
/* SROUT(18): */
/* DEFINED TYPE 10
/*          19          DESCRIPTOR ::= REP , DATYPE , VALUE , LENGTH
SROUT(19):
  IF INPUTRM = 0 THEN CALL GNRTR(2260);
  ELSE
    DO;
      IF LTRNTKN(9) = 0 THEN CALL GNRTR(2251);
    ELSE
      DO;
        CALL GNRTR(2250);
        CALL GNRTR(5000+(IC1-4)+1);
        CALL GNRTR(2221);
        CALL GNRTR(IDSTK(1));
        CALL GNRTR(2200);
      END;
    END;
  GO TO EXIT;
/* DEFINED TYPE 11
/*          20          CONTROL ::= : OPTIONS
/*          21          ;
/* SROUT(20): */

```

```

/* SROUT(21): */
00003430
00003440
00003450
/* DEFINED TYPE 12
00003460
/*      22  COMPAREXPR ::= CONCAT CONNECTIVE CONCAT
00003470
/*      23
00003480
00003490
SROUT(22):
00003500
CALL GNRTR(CMPRS(LTRNTKN(21)-56));
00003510
GO TO EXIT:
00003520
00003530
/* SROUT(23): */
00003540
00003550
00003560
/* DEFINED TYPE 13
00003570
/*      24  ASSGNEXPR ::= IDENTIFIER .<=. CONCAT
00003580
SROUT(24):
00003590
CALL GNRTR(IDSTK(1));
00003600
CALL GNRTR(2200);
00003610
GO TO EXIT:
00003620
00003630
00003640
00003650
/* DEFINED TYPE 14
00003660
/*      25  IDENTIFIER ::= A
00003670
/*      26
00003680
/*      27
00003690
/*      28
00003700
/*      29
00003710
/*      30
00003720
/*      31
00003730
/*      32
00003740
/*      33
00003750
/*      34
00003760
/*      35
00003770
/*      36
00003780
/*      37
00003790
/*      38
00003800
/*      39
00003810
/*      40
00003820
/*      41
00003830
/*      42
00003840
/*      <ALPHA ALPHANUM>
00003850
SROUT(25):
00003860
/* SROUT(26): */
00003870
/* SROUT(27): */
00003880
/* SROUT(28): */
00003890
/* SROUT(29): */
00003900
/* SROUT(30): */
00003910
/* SROUT(31): */
00003920
/* SROUT(32): */
00003930
/* SROUT(33): */
00003940
/* SROUT(34): */
00003950
/* SROUT(35): */
00003960
/* SROUT(36): */
00003970
/* SROUT(37): */
00003980
/* SROUT(38): */
00003990

```

```

/* SROUT(39): */
/* SROUT(40): *
/* SROUT(41): */
/* SROUT(42): */
TEMP = LENGTH(RCDVLU);
IF TEMP > IDLENGTH THEN
DN:
CALL DISPLAY('IDENTIFIER:');
CALL DISPLAY(RCDVLU);
CALL DISPLAY(
'HAS' || TEMP || ' CHARACTERS. MAX IS 4');
GO TO ERROR;
END;
IDSP = IDSP+1;
IF IDSP > MXSTKSZ THEN
DN:
CALL DISPLAY('IDSTACK OVERFLOW. MAX IS' || MXSTKSZ);
IDSP = 0;
GO TO ERROR;
END;
IDSTK(IDSP) = FINDID(RCDVLU);
IF IDSTK(IDSP) >= MXIDS THEN GO TO ERROR;
IF IDOFST(IDSTK(IDSP)) >= MXNDXS THEN GO TO ERROR;
GO TO EXIT;

/* DEFINED TYPE 15
/* 43 REP ::= #
/* 44 | ARITH
/* 45 |

SROUT(43):
CALL GNRTR(4000);
GO TO EXIT;

/* SROUT(44): */

SROUT(45):
CALL GNRTR(5000);
GO TO EXIT;

/* DEFINED TYPE 16
/* 46 DATYPE ::= LITYPE
/* 47 | T ( IDENTIFIER )
/* 48 |

SROUT(46):
CALL GNRTR(1000+LTRNTKN(23)-64);
GO TO EXIT;

SROUT(47):
CALL GNRTR(1+TK(IDSP));
CALL GNRTR(2112);
GO TO EXIT;

SROUT(48):

```

```

00004000
00004010
00004020
00004030
00004040
00004050
00004060
00004070
00004080
00004090
00004100
00004110
00004120
00004130
00004140
00004150
00004160
00004170
00004180
00004190
00004200
00004210
00004220
0000
00004230
00004250
*/00004260
*/00004270
*/00004280
*/00004290
00004300
00004310
00004320
00004330
00004340
00004350
00004360
00004370
00004380
00004390
00004400
00004410
*/00004420
*/00004430
*/00004440
*/00004450
00004460
00004470
00004480
00004490
00004500
00004510
00004520
00004530
00004540
00004550
00004560

```

```

LTRNTKN(23) = 65:
GO TO SROUT(46):

/* DEFINED TYPE 17
/*      49      VALUE ::= CONCAT
/*      50      |
/* SROUT(49): */

SROUT(50):
CALL GNRTR(5000);
GO TO EXIT:

/* DEFINED TYPE 18
/*      51      LENGTH ::= ARITH
/*      52      |
/* SROUT(51): */

SROUT(52):
IF LTRNTKN(16) = 48 THEN CALL GNRTR(1032);
GO TO EXIT:

/* DEFINED TYPE 19
/*      53      OPTIONS ::= TEST
/*      54      | TEST , TEST
/* SROUT(53): */
/* SROUT(54): */

/* DEFINED TYPE 20
/*      55      CONCAT ::= VAL
/*      56      | CONCAT || VAL
/* SROUT(55): */

SROUT(56):
CALL GNRTR(2040);
GO TO EXIT:

/* DEFINED TYPE 21
/*      57      CONNECTIVE ::= .LF.
/*      58      | .LT.
/*      59      | .GF.
/*      60      | .GT.
/*      61      | .EQ.
/*      62      | .NE.
/* SROUT(57): */
/* SROUT(58): */
/* SROUT(59): */
/* SROUT(60): */

```

```

00004570
00004580
00004590
00004600
*/00004610
*/00004620
*/00004630
00004640
00004650
00004660
00004670
00004680
00004690
00004700
00004710
*/00004720
*/00004730
*/00004740
00004750
00004760
00004770
00004780
00004790
00004800
00004810
00004820
*/00004830
*/00004840
*/00004850
00004860
00004870
00004880
00004890
00004900
*/00004910
*/00004920
*/00004930
00004940
00004950
00004960
00004970
00004980
00004990
00005000
00005010
*/00005020
*/00005030
*/00005040
*/00005050
*/00005060
*/00005070
*/00005080
00005090
00005100
00005110
00005120
00005130

```

```

/* SROUT(61): */
/* SROUT(62): */

/* DEFINED TYPE 22
/*      63  ARITH ::= PRIMARY
/*      64      | ARITH OPERATOR PRIMARY

/* SROUT(63): */

SROUT(64):
  CALL GNRTR(2000+(LTRNTKN(27)-80)*10);
  GO TO EXIT;

/* DEFINED TYPE 23
/*      65  LITYPE ::= R
/*      66      | O
/*      67      | X
/*      68      | E
/*      69      | A
/*      70      | ED
/*      71      | AD
/*      72      | SB

/* SROUT(65): */
/* SROUT(66): */
/* SROUT(67): */
/* SROUT(68): */
/* SROUT(69): */
/* SROUT(70): */
/* SROUT(71): */
/* SROUT(72): */

/* DEFINED TYPE 24
/*      73  TEST ::= <SFUR IDENT> ( ARITH )

SROUT(73):
  IF LTRNTKN(28) >= 87 THEN CALL GNRTR(2210);
  ELSE
    DO:
      CALL GNRTR(2120);
      CALL GNRTR(2222);
    END:
  IF LTRNTKN(28) = 86 & LTRNTKN(28) = 89 THEN
    JNDP(ALT) = 3000+IC1+1;
  GO TO EXIT;

/* DEFINED TYPE 25
/*      74  VAL ::= LITYPE LITSTRING
/*      75      | ARITH

SROUT(74):
  TEMP = FINDLR(RCDVLU,LTRNTKN(23)-64);
  IF TEMP < 0 THEN GO TO ERROR;

```

```

CALL GNRTR(TEMP);
GO TO EXIT;

/* DEFINED TYPE 26
/*      76      PRIMARY ::= IDENTIFIER
/*      77              | L ( IDENTIFIER )
/*      78              | V ( IDENTIFIER )
/*      79              | INTEGER

/* SROUT(75): */

SROUT(76):
CALL GNRTR(IDSTK(IDSP));
GO TO EXIT;

SROUT(77):
CALL GNRTR(IDSTK(IDSP));
CALL GNRTR(2111);
GO TO EXIT;

SROUT(78):
CALL GNRTR(IDSTK(IDSP));
CALL GNRTR(2110);
GO TO EXIT;

SROUT(79):
HOLD(1) = BINARY(RCDVLU);
CALL GNRTR(1000+HOLD(1));
GO TO EXIT;

/* DEFINED TYPE 27
/*      80      OPERATOR ::= +
/*      81              | -
/*      82              | *
/*      83              | /

/* SROUT(80): */
/* SROUT(81): */
/* SROUT(82): */
/* SROUT(83): */

/* DEFINED TYPE 28
/*      84      <SFUR IDENT> ::= S
/*      85              | F
/*      86              | U
/*      87              | SR
/*      88              | FR
/*      89              | UR

SROUT(84):
/* SROUT(85): */
/* SROUT(86): */
/* SROUT(87): */
/* SROUT(88): */

```

```

00005710
00005720
00005730
00005740
*/00005750
*/00005760
*/00005770
*/00005780
*/00005790
00005800
00005810
00005820
00005830
00005840
00005850
00005860
00005870
00005880
00005890
00005900
00005910
00005920
00005930
00005940
00005950
00005960
00005970
00005980
00005990
00006000
00006010
00006020
*/00006030
*/00006040
*/00006050
*/00006060
*/00006070
00006080
00006090
00006100
00006110
00006120
00006130
00006140
*/00006150
*/00006160
*/00006170
*/00006180
*/00006190
*/00006200
*/00006210
00006220
00006230
00006240
00006250
00006260
00006270

```



```

/* SROUT(89): */
ALT = IC1+1;
CALL GNRTR(2130);
IF LTRNTKN(28) = 84 | LTRNTKN(28) = 87 THEN CALL GNRTR(2221);
IF LTRNTKN(28) = 85 | LTRNTKN(28) = 88 THEN CALL GNRTR(2220);
GO TO EXIT;

/* DEFINED TYPE 29
/* 90 SEPERATOR ::= :

SROUT(90):
CALL GNRTR(2240);
GO TO EXIT;

GNRTR: PROCEDURE(INSTRCTN) RETURNS(BIN FIXED (15));
DCL INSTRCTN BIN FIXED (15);
DCL SAID BIN FIXED (15) ALIGNED STATIC INITIAL (0);

/* ROUTINE TO POST AN INSTRUCTION */

I = IC1+1;
IF I >= MXINSTS THEN
  UN;
IF SAID = 0 THEN
  CALL DISPLAY(
    'INSTRUCTION STACK OVERFLOW. MAX IS '||MXINSTS);
  SAID = 1;
  REPLY = -3;
  I = IC;
  RETURN;
END;
IC1 = I;
CODE(IC1) = INSTRCTN;
END GNRTR;

FINDLB: PROCEDURE(L) RETURNS(BIN FIXED (15));
DCL L BIN FIXED (15);
DCL SAID BIN FIXED (15) ALIGNED STATIC INITIAL (0);

/* ROUTINE TO FIND A LABEL */

IF NMLBS >= 0 THEN
  DO I = 0 TO NMLBS;
  IF LBLVLU(I) = L THEN RETURN(I);
  END;
  I = NMLBS+1;
  IF I >= MXLBS THEN
    UN;
  IF SAID = 0 THEN
    CALL DISPLAY('LABEL TABLE OVERFLOW. MAX IS '||MXLBS);
    LBLFST(NMLBS) = (IC+1)*2;
    SAID = 1;

```

REPLY = -3;	00006850
RETURN(MXLRLS);	00006860
END;	00006870
NMLRS = I;	00006880
LBLVLU(NMLRS) = L;	00006890
RETURN(NMLRS);	00006900
END FINDLR;	00006910
	00006920
	00006930
	00006940
FINDID:PROCEDURE(K) RETURNS(BINARY FIXED (15));	00006950
DCL K CHAR(4) VAR ;	00006960
DCL SAID BIN FIXED (15) ALIGNED STATIC INITIAL (0);	00006970
DCL SAID2 BIN FIXED (15) ALIGNED STATIC INITIAL (0);	00006980
	00006990
/* ROUTINE TO FIND AN IDENTIFIER */	00007000
	00007010
IF NMNDXS >= 0 THEN	00007020
DO I = 0 TO NMIDS;	00007030
IF IDTYPE(I) = 0 THEN	00007040
DO;	00007050
IF IDNAME(IDOFST(I)) = K THEN RETURN(I);	00007060
END;	00007070
END;	00007080
I = NMNDXS+1;	00007090
IF NMNDXS >= MXNDXS THEN	00007100
DO;	00007110
IF SAID = 0 THEN	00007120
CALL DISPLAY(	00007130
'EXCEEDED MAX NUMBER OF IDENTIFIERS. MAX IS' MXNDXS);	00007140
SAID = 1;	00007150
REPLY = -3;	00007160
I = MXNDXS;	00007170
END;	00007180
NMNDXS = I;	00007190
IDNAME(NMNDXS) = K;	00007200
	00007210
GETIDNT: ENTRY RETURNS (BIN FIXED (15));	00007220
I = NMIDS+1;	00007230
IF I >= MXIDS THEN	00007240
DO;	00007250
IF SAID2 = 0 THEN	00007260
CALL DISPLAY('IDENTIFIER TABLE OVERFLOW. MAX IS' MXNDXS);	00007270
SAID2 = 1;	00007280
REPLY = -3;	00007290
RETURN(MXIDS);	00007300
END;	00007310
NMIDS = I;	00007320
IDOFST(NMIDS) = NMNDXS;	00007330
IDTYPE(NMIDS) = 0;	00007340
RETURN(NMIDS);	00007350
END FINDID;	00007360
	00007370
	00007380
	00007390
FINDLT: PROCEDURE(M,N) RETURNS(BIN FIXED (15));	00007400
DCL M CHAR(256) VAR ;	00007410

DCL N BIN FIXED (15):	00007420
DCL (POS,J,INDX,TMPNMIDS) BIN FIXED (15) STATIC:	00007430
DCL NMAR BIN FIXED (31) STATIC INITIAL (0):	00007440
DCL TBITSTR BIT (8) UNALIGNED STATIC INITIAL ('0'B):	00007450
DCL CHR1 CHAR(1) STATIC:	00007460
DCL CHR4 CHAR(4) STATIC:	00007470
DCL SAID BIN FIXED ALIGNED STATIC INITIAL (0):	00007480
DCL LENGTH(0:8) BIN FIXED (15) STATIC ALIGNED INITIAL (	00007490
0,2,8,16,79,79,10,10,2):	00007500
DCL TRNSL8R(1:2,0:78) BIT (8) UNALIGNED STATIC EXT INITIAL (	00007510
'11110000'B,	00007520
'11110001'B,	00007530
'11110010'B,	00007540
'11110011'B,	00007550
'11110100'B,	00007560
'11110101'B,	00007570
'11110110'B,	00007580
'11110111'B,	00007590
'11111000'B,	00007600
'11111001'B,	00007610
'11000001'B,	00007620
'11000010'B,	00007630
'11000011'B,	00007640
'11000100'B,	00007650
'11000101'B,	00007660
'11000110'B,	00007670
'11000111'B,	00007680
'11001000'B,	00007690
'11001001'B,	00007700
'11010001'B,	00007710
'11010010'B,	00007720
'11010011'B,	00007730
'11010100'B,	00007740
'11010101'B,	00007750
'11010110'B,	00007760
'11010111'B,	00007770
'11011000'B,	00007780
'11011001'B,	00007790
'11100010'B,	00007800
'11100011'B,	00007810
'11100100'B,	00007820
'11100101'B,	00007830
'11100110'B,	00007840
'11100111'B,	00007850
'11101000'B,	00007860
'11101001'B,	00007870
'01000000'B,	00007880
'01001011'B,	00007890
'01001101'B,	00007900
'01001110'B,	00007910
'01010000'B,	00007920
'01011011'B,	00007930
'01011100'B,	00007940
'01011101'B,	00007950
'01100000'B,	00007960
'01100001'B,	00007970
'01101011'B,	00007980

'01101100'B,	00007990
'01111011'B,	00008000
'01111103'B,	00008010
'01111101'B,	00008020
'01111110'B,	00008030
'10000001'B,	00008040
'10000010'B,	00008050
'10000011'B,	00008060
'10000100'B,	00008070
'10000101'B,	00008080
'10000110'B,	00008090
'10000111'B,	00008100
'10001000'B,	00008110
'10001001'B,	00008120
'10010001'B,	00008130
'10010010'B,	00008140
'10010011'B,	00008150
'10010100'B,	00008160
'10010101'B,	00008170
'10010110'B,	00008180
'10010111'B,	00008190
'10011000'B,	00008200
'10011001'B,	00008210
'10100010'B,	00008220
'10100011'B,	00008230
'10100100'B,	00008240
'10100101'B,	00008250
'10100110'B,	00008260
'10100111'B,	00008270
'10101000'B,	00008280
'10101001'B,	00008290
'00000000'B,	00008300
'01010000'B,	00008310
'01010001'B,	00008320
'01010010'B,	00008330
'01010011'B,	00008340
'01010100'B,	00008350
'01010101'B,	00008360
'01010110'B,	00008370
'01010111'B,	00008380
'01011000'B,	00008390
'01011001'B,	00008400
'10100001'B,	00008410
'10100010'B,	00008420
'10100011'B,	00008430
'10100100'B,	00008440
'10100101'B,	00008450
'10100110'B,	00008460
'10100111'B,	00008470
'10101000'B,	00008480
'10101001'B,	00008490
'10101010'B,	00008500
'10101011'B,	00008510
'10101100'B,	00008520
'10101101'B,	00008530
'10101110'B,	00008540
'10101111'B,	00008550

'10110000'B,
'10110001'B,
'10110010'B,
'10110011'B,
'10110100'B,
'10110101'B,
'10110110'B,
'10110111'B,
'10111000'B,
'10111001'B,
'10111010'B,
'10100000'B,
'01001110'B,
'01001000'B,
'01001011'B,
'01000110'B,
'01000100'B,
'01001010'B,
'01001001'B,
'01001101'B,
'01001111'B,
'01001100'B,
'01000101'B,
'01000011'B,
'10100000'B,
'01000111'B,
'01011101'B,
'11100001'B,
'11100010'B,
'11100011'B,
'11100100'B,
'11100101'B,
'11100110'B,
'11100111'B,
'11101000'B,
'11101001'B,
'11101010'B,
'11101011'B,
'11101100'B,
'11101101'B,
'11101110'B,
'11101111'B,
'11110000'B,
'11110001'B,
'11110010'B,
'11110011'B,
'11110100'B,
'11110101'B,
'11110110'B,
'11110111'B,
'11111000'B,
'11111001'B,
'11111010'B,
'00000000'B):

/* ROUTINE TO FIND A LITERAL */

00008560
00008570
00008580
00008590
00008600
00008610
00008620
00008630
00008640
00008650
00008660
00008670
00008680
00008690
00008700
00008710
00008720
00008730
00008740
00008750
00008760
00008770
00008780
00008790
00008800
00008810
00008820
00008830
00008840
00008850
00008860
00008870
00008880
00008890
00008900
00008910
00008920
00008930
00008940
00008950
00008960
00008970
00008980
00008990
00009000
00009010
00009020
00009030
00009040
00009050
00009060
00009070
00009080
00009090
00009100
00009110
00009120

```

NMHR = 0: 00009130
REPLY = -3: 00009140
TMPNMIDS = GETIONT: 00009150
IF TMPNMIDS >= MXIDS THEN RETURN(-1): 00009160
NMIDS = NMIDS-1: 00009170
IDTYPE(TMPNMIDS) = N: 00009180
IDLNG(TMPNMIDS) = LENGTH(M): 00009190
IF IDTYPE(TMPNMIDS) = 0 | IDLNG(TMPNMIDS) = 0 THEN RETURN(-1): 00009200
IF (LTEND+IDLNG(TMPNMIDS) > MXLITS & 00009210
( N > 3 | N < 6)) | (LTEND+4 > MXLITS & 00009220
( N <= 3 | N >= 6)) THEN 00009230
DO: 00009240
IF SAID = 0 THEN 00009250
CALL DISPLAY( 00009260
'LITERAL TABLE OVERFLOW. MAX IS'||MXLITS||' BYTES.'): 00009270
SAID = 1: 00009280
RETURN(-1): 00009290
END: 00009300
IDDEST(TMPNMIDS) = LTEND: 00009310
IF N = 5 THEN INDX = 2: ELSE INDX = 1: 00009320
DO POS = 1 TO IDLNG(TMPNMIDS): 00009330
TRITSTR = UNSPEC(SUBSTR(M,POS,1)): 00009340
DO J = 0 TO 78: 00009350
IF TRITSTR = TRANSLAR(1,J) THEN GO TO GUID: 00009360
END: 00009370
ERROR: 00009380
CALL DISPLAY('LITERAL:'): 00009390
CALL DISPLAY(M): 00009400
CALL DISPLAY('IS NOT IN ITS SPECIFIED MODE'): 00009410
RETURN(-1): 00009420
GUID: 00009430
IF J >= LENGTH(N) THEN GO TO ERROR: 00009440
IF N <= 3 | N >= 6 THEN NMHR = J+LENGTH(N)*NMHR: 00009450
ELSE 00009460
DO: 00009470
UNSPEC(CHR1) = TRANSLAR(INDX,J): 00009480
SUBSTR(LTRLS,LTEND+POS,1) = CHR1: 00009490
END: 00009500
END: 00009510
REPLY = 0: 00009520
IF N <= 3 | N >= 6 THEN 00009530
DO: 00009540
IF N = 8 THEN NMHR = -NMHR: 00009550
IDLNG(TMPNMIDS) = 4: 00009560
UNSPEC(CHR4) = UNSPEC(NMHR): 00009570
SUBSTR(LTRLS,LTEND+.4) = CHR4: 00009580
END: 00009590
DO POS = 0 TO NMIDS: 00009600
IF IDTYPE(POS) = IDTYPE(TMPNMIDS) & 00009610
IDLNG(POS) = IDLNG(TMPNMIDS)*8 & 00009620
SUBSTR(LTRLS,IDDEST(POS)+1,IDLNG(POS)/8) = 00009630
SUBSTR(LTRLS,IDDEST(TMPNMIDS)+1,IDLNG(TMPNMIDS)) 00009640
THEN RETURN(POS): 00009650
END: 00009660
NMIDS = TMPNMIDS: 00009670
LTEND = IDDEST(TMPNMIDS)+IDLNG(TMPNMIDS): 00009680
IF N > 3 & N < 6 THEN 00009690

```

```

LTEND = TRUNC((LTEND+3)/4)*4:
      IDLNG(TMPNMIDS) = IDLNG(TMPNMIDS)*8:
      RETURN(TMPNMIDS):
      END FINDLT:

PUCODE: PROCEDURE:
      DCL (I,J,K,L) BIN FIXED (15) STATIC:
      DCL TEXT CHAR(72) STATIC INITIAL (''):
      DCL SYMBOL CHAR(4) STATIC INITIAL (''):
      DCL OPRTRS(64) CHAR (4) STATIC EXT INIT
      ('LD','IC',,,,'AD','ARB','NULL','FOT','FOR','ADD','SUB','MUL',
      'DIV','CON',,,,,'LIV','LIL','LIT','LUL',,,,,'STO',,,,,'
      ',,,,,'RET',,,,,'BT','BF','BU',,,,,'CFQ',
      'CNE','CLE','CLT','CGF','CGT','SCIP','SICP',,,,,'
      'INS','IND',,,,,'OUT',,,,,'
      ');
      DCLARE BYTE(4) BIN FIXED STATIC :
      DCL OPRTR CHAR(4) STATIC:
      DCL OPRND BIN FIXED STATIC:

      /* ROUTINE TO REFORMAT AND LIST INSTRUCTIONS OF A RULE */

      CALL DISPLAY(' ');
      CALL DISPLAY(' ISN      INSRCT      OPCD  OPRND  SYMBOL');
      DO I=1 TO ICO-IC:
      IC = IC+1:
      K = CODE(IC):
      DO J = 1 TO 4:
      L = K/10:
      BYTE(5-J) = K-L*10:
      K = L:
      END:
      OPRTR = OPRTRS(BYTE(1)+1):
      IF BYTE(1) = 2 | BYTE(1) = 4 | BYTE(1) = 5 THEN
      DO:
      K = 0:
      DO J = 1 TO 4:
      K = BYTE(J)+K*16:
      END:
      CODE(IC) = K:
      IF BYTE(1) = 2 THEN
      DO:
      OPRTR = OPRTRS(9+BYTE(3)):
      IF BYTE(2) /= 0 THEN
      DO:
      OPRTR = OPRTRS(14+BYTE(3)*3+BYTE(4)):
      IF BYTE(2) /= 1 THEN
      DO:
      OPRTR = OPRTRS(23+BYTE(3)*6+BYTE(4)):
      END:
      END:
      END:
      PUT STRING(TEXT) FDI(1,IC,BYTE(1),BYTE(2),BYTE(3),BYTE(4),OPRTR)
      (F(4),X(4),F(1),X(1),F(1),X(1),F(1),X(1),F(1),X(4),A(4)):
      END:
      ELSEF

```

00009700
00009710
00009720
00009730
00009740
00009750
00009760
00009770
00009780
00009790
00009800
00009810
00009820
00009830
00009840
00009850
00009860
00009870
00009880
00009890
00009900
00009910
00009920
00009930
00009940
00009950
00009960
00009970
00009980
00009990
00010000
00010010
00010020
00010030
00010040
00010050
00010060
00010070
00010080
00010090
00010100
00010110
00010120
00010130
00010140
00010150
00010160
00010170
00010180
00010190
00010200
00010210
00010220
00010230
00010240
00010250
00010260

DO;	00010270
OPRND = BYTE(2)*100+BYTE(3)*10+BYTE(4);	00010280
CODE(IC) = 4096*BYTE(1)+OPRND;	00010290
SYMBOL = ' ';	00010300
IF BYTE(1) = 0 THEN	00010310
DO;	00010320
IF IDTYPE(OPRND) = 0 THEN	00010330
SYMBOL = IDNAME(IDOFFST(OPRND));	00010340
END;	00010350
PUT STRING(TEXT) EDIT (IC,BYTE(1),OPRND,OPRTR,OPRND,SYMBOL)	00010360
(F(4),X(4),F(1),X(2),F(4),X(4),A(4),X(2),F(4),X(2),A(4));	00010370
END;	00010380
CALL DISPLAY(TEXT);	00010390
END;	00010400
CALL DISPLAY(' ');	00010410
END SPUCCODE;	00010420
	00010430
	00010440
	00010450
	00010460
TABLES: PROCEDURE:	00010470
DCL LN BIN FIXED STATIC INITIAL (0);	00010480
DCL VAL CHAR (6) VAR STATIC INITIAL ('');	00010490
DCL VALU BIT (48) ALIGNED STATIC INITIAL ('0'R);	00010500
DCL TEXT CHAR (72) VAR ALIGNED STATIC INITIAL ('');	00010510
DCL TPNAME(0:8) CHAR (2) UNALIGNED STATIC INITIAL (	00010520
' U',' B',' D',' X',' E',' A',' FD',' AD',' SR');	00010530
	00010540
/* ROUTINE TO LIST THE CONTENT OF THE COMPILER TABLES */	00010550
	00010560
CALL DISPLAY(' ');	00010570
CALL DISPLAY(' ');	00010580
CALL DISPLAY('***** LABEL TABLE *****');	00010590
CALL DISPLAY(' ');	00010600
IF NMLRS < 0 THEN CALL DISPLAY('NO LABELS DECLARED');	00010610
ELSE	00010620
DO;	00010630
CALL DISPLAY('ENTRY LABEL OFFSET');	00010640
CALL DISPLAY(' ');	00010650
DO I = 0 TO NMLRS;	00010660
PUT STRING(TEXT) EDIT (I,LR VAL(I),LRLOFST(I);	00010670
(F(5),X(5),F(5),X(4),F(6));	00010680
CALL DISPLAY(TEXT);	00010690
END;	00010700
END;	00010710
CALL DISPLAY(' ');	00010720
CALL DISPLAY('***** IDENTIFIER TABLE *****');	00010730
CALL DISPLAY(' ');	00010740
IF NMNDXS < 0 THEN CALL DISPLAY('NO IDENTIFIERS DECLARED');	00010750
ELSE	00010760
DO;	00010770
CALL DISPLAY('ENTRY IDENTIFIER');	00010780
CALL DISPLAY(' ');	00010790
DO I = 0 TO NMIDS;	00010800
IF IDTYPE(I) = 0 THEN	00010810
DO;	00010820
PUT STRING(TEXT) EDIT (I,IDNAME(IDOFFST(I)))	00010830

(F(5),X(18),A(4)):	00010840
CALL DISPLAY(TEXT):	00010850
END:	00010860
END:	00010870
END:	00010880
CALL DISPLAY(' '):	00010890
CALL DISPLAY('***** LITERAL TABLE *****'):	00010900
CALL DISPLAY(' '):	00010910
IF LTEND = 0 THEN CALL DISPLAY('NO LITERALS DECLARED');	00010920
ELSE	00010930
DO:	00010940
CALL DISPLAY('ENTRY TYPE LENGTH OFFSET LITERAL');	00010950
CALL DISPLAY(' '):	00010960
DO I = 0 TO NMIDS:	00010970
IF IDTYPE(I) = 0 THEN	00010980
DO:	00010990
VALU = '0'0;	00011000
VAL = SUBSTR(LTRLS,IDOEST(I)+1,IDLNG(I)/8):	00011010
IF IDLNG(I) > 48 THEN LN = 48: ELSE LN = IDLNG(I):	00011020
VALU = UNSPEC(VAL):	00011030
PUT STRING(TEXT) EDIT (I,TPNAME(IDTYPE(I)),IDLNG(I),IDOEST(I),VALU	00011040
)	00011050
(F(5),X(1),A(2),X(1),F(5),X(1),F(6),X(2),B(LN)):	00011060
CALL DISPLAY(TEXT):	00011070
END:	00011080
END:	00011090
END:	00011100
END TABLES:	00011110
END SMNTC:	00011120

SMFSIO: PROCEDURE(DS,STRUCT):

/* SHARED VARIABLES */

DCL TRACE BIN FIXED (31) STATIC EXT;
DCL LINESZ BIN FIXED (15) STATIC EXT INITIAL (72);
DCL LINE CHAR (332) VAR ALIGNED STATIC EXT INITIAL ('');
DCL INPUT CHAR (84) VAR ALIGNED STATIC EXT INITIAL ('');

/* ROUTINES INVOKED BY SMFSIO AND ENTRY POINTS */

DCL DDISPLAY ENTRY (CHAR (72) VAR):
DCL DWRITE ENTRY (BIN FIXED (31), (0:1) BIN FIXED (31),
BIN FIXED (31), BIN FIXED (31));
DCL DREAD ENTRY (BIN FIXED (31), (0:1) BIN FIXED (31),
BIN FIXED (31), BIN FIXED (31));
DCL DOPEN ENTRY (BIN FIXED (31), BIN FIXED (31),
(2) BIN FIXED (31), (2) BIN FIXED (31), (2) BIN FIXED (31));
DCL DCLOSE ENTRY (BIN FIXED (31), BIN FIXED (31));
DCL POINT ENTRY:

/* VARIABLES FOR SMFS */

DCL 1 WRTORJT STATIC EXT,
2 OPCD BIT (8) UNALIGNED INITIAL ('00000011'B),
2 FLGS BIT (16) UNALIGNED INITIAL ('0010000000000000'B),
2 DLNG BIT (32) UNALIGNED INITIAL (0):

DCL 1 STRUCT ALIGNED,
2 DUMMY BIT (8) ALIGNED,
2 OPCD BIT (8) UNALIGNED,
2 FLGS BIT (16) UNALIGNED,
2 DLNG BIT (32) UNALIGNED,
2 DATA BIN FIXED (15) ALIGNED:

DCL 1 STRCT UNALIGNED,
2 OPCD BIT (8) UNALIGNED,
2 FLGS BIT (16) UNALIGNED,
2 NLNG BIT (8) UNALIGNED,
2 FNAM CHAR (36) UNALIGNED,
2 FLNG BIN FIXED (31) ALIGNED:

DCL 1 WRTDIAG STATIC UNALIGNED EXT,
2 OPCD BIT (8) INITIAL ('00000011'B),
2 FLGS BIT (16) INITIAL ('001000000010000000'B),
2 DLNG BIT (32) INITIAL ('0000000000000000000000001001000000'B),
2 DATA CHAR (72) INITIAL ('');

DCL 1 GTSRCE STATIC EXT,
2 OPCD BIT (8) UNALIGNED INITIAL ('00000101'B),
2 FLGS BIT (16) UNALIGNED INITIAL ('00100000000000000000'B),
2 DLNG BIT (32) UNALIGNED INIT
('0000000000000000000000001010000000'B):

00000010
00000020
00000030
00000040
00000050
00000060
00000070
00000080
00000090
00000100
00000110
00000120
00000130
00000140
00000150
00000160
00000170
00000180
00000190
00000200
00000210
00000220
00000230
00000240
00000250
00000260
00000270
00000280
00000290
00000300
00000310
00000320
00000330
00000340
00000350
00000360
00000370
00000380
00000390
00000400
00000410
00000420
00000430
00000440
00000450
00000460
00000470
00000480
00000490
00000500
00000510
00000520
00000530
00000540
00000550
00000560
00000570

DCL WRKSPS1(2) BIN FIXED (31) ALIGNED STATIC EXT:	00000580
DCL CMPLT1(2) BIN FIXED (31) ALIGNED STATIC EXT:	00000590
DCL CMPLT2(2) BIN FIXED (31) ALIGNED STATIC EXT:	00000600
DCL SOKT BIN FIXED (31):	00000610
DCL SSKT(2) BIN FIXED (31) STATIC INIT (3,0):	00000620
DCL RSKT(2) BIN FIXED (31) STATIC INIT (3,0):	00000630
DCL LTIME28 BIN FIXED (31) STATIC INITIAL (10000):	00000640
DCL STIME28 BIN FIXED (31) STATIC INITIAL (300):	00000650
	00000660
	00000670
	00000680
	00000690
/* LOCAL VARIABLES */	00000700
	00000710
DCL WS(2) BIN FIXED (31):	00000720
DCL DS(2) BIN FIXED (31):	00000730
DCL LEN BIN FIXED (31) STATIC INIT (0):	00000740
DCL OPCODE BIN FIXED (15) STATIC INITIAL (0):	00000750
DCL NOJAGFL BIN FIXED (15) STATIC INITIAL (0):	00000760
DCL TEXT CHAR (72) VAR:	00000770
DCL AFR(0:1) BIN FIXED (31) STATIC EXT INIT ((2)0):	00000780
DCL BFR(0:1) BIN FIXED (31) STATIC EXT:	00000790
	00000800
	00000810
	00000820
OPCODE = WRTORJT.OPCD:	00000830
WRTORJT.DLNG = STRUCT.DLNG:	00000840
LEN = WRTORJT.DLNG:	00000850
IF TRCF = 1 THEN CALL DISPLAY(	00000860
'SMFS OPCODE:' OPCODE ' LENGTH OF DATA:' LEN):	00000870
CALL POINT(BFR,ADDR(WRTORJT.OPCD)):	00000880
CALL @WRITE(DS(1),BFR,56,LTIME28):	00000890
IF DS(1) > 0 THEN GO TO REPORT:	00000900
CALL POINT(BFR,ADDR(STRUCT.DATA)):	00000910
CALL @WRITE(DS(1),BFR,LEN,LTIME28):	00000920
COMMON:	00000930
IF DS(1) > 0 THEN GO TO REPORT:	00000940
CALL @READ(DS(2),AFR,8,LTIME28):	00000950
IF DS(2) > 0 THEN GO TO REPORT:	00000960
AFR(1) = AFR(0)/(256**3):	00000970
IF AFR(1) < 2 AFR(1) > 8 THEN	00000980
DO:	00000990
CALL DISPLAY(	00001000
'NO I/O: SMFS REPORTS COMPLETION CODE ' AFR(1)):	00001010
END:	00001020
RETURN:	00001030
	00001040
	00001050
	00001060
SMFLIO: ENTRY(DS,STRUCT):	00001070
OPCODE = STRUCT.OPCD:	00001080
LEN = 0:	00001090
IF OPCODE = 3 THEN	00001100
LEN = STRUCT.FLNG:	00001110
LEN = LEN+352:	00001120
CALL POINT(BFR,ADDR(STRUCT.OPCD)):	00001130
CALL @WRITE(DS(1),BFR,LEN,LTIME28):	00001140

LEN = LEN-352;	00001150
GO TO COMMON;	00001160
	00001170
	00001180
	00001190
SMF010: ENTRY(DS,SOKT,WS):	00001200
SSKT(2) = SOKT;	00001210
RSKT(2) = 1025;	00001220
CALL @OPEN(DS(1),STIME2R,SSKT,RSKT,WS);	00001230
IF DS(1) = 0 THEN	00001240
DO;	00001250
CALL @READ(DS(1),AFR,32,LTIME2R);	00001260
IF DS(1) = 0 THEN	00001270
DO;	00001280
CALL @CLOSE(DS(1),STIME2R);	00001290
SSKT(2) = SOKT+2;	00001300
RSKT(2) = AFR(1)+1;	00001310
CALL @OPEN(DS(2),STIME2R,SSKT,RSKT,WS);	00001320
IF DS(2) = 0 THEN	00001330
DO;	00001340
SSKT(2) = SOKT+3;	00001350
RSKT(2) = AFR(0);	00001360
CALL @OPEN(DS(1),STIME2R,SSKT,RSKT,WS);	00001370
IF DS(1) = 0 THEN RETURN;	00001380
END;	00001390
END;	00001400
END;	00001410
CALL DISPLAY	00001420
'SEND SOCKET:' SOKT+3);	00001430
CALL DISPLAY	00001440
'RECEIVE SOCKET:' SOKT+2);	00001450
CALL DISPLAY	00001460
'NO OPEN';	00001470
GO TO REPT;	00001480
REPT01:	00001490
CALL DISPLAY	00001500
'NO INPUT/OUTPUT';	00001510
REPT:	00001520
CALL DISPLAY	00001530
'NCP REPORTS COMPLETION CODE ' DS(1) ' ON SEND SOCKET';	00001540
CALL DISPLAY	00001550
'NCP REPORTS COMPLETION CODE ' DS(2) ' ON RECEIVE SOCKET';	00001560
RETURN;	00001570
	00001580
	00001590
	00001600
SMFC10: ENTRY(DS):	00001610
CALL @CLOSE(DS(1),STIME2R);	00001620
CALL @CLOSE(DS(2),STIME2R);	00001630
RETURN;	00001640
	00001650
	00001660
	00001670
DISPLAY: ENTRY(TEXT):	00001680
CALL DISPLAY(TEXT);	00001690
RETURN;	00001700
	00001710

READ: ENTRY RETURNS(BIN FIXED (15));	00001720
IF CMPLT1(2) = 20 THEN RETURN(0);	00001730
CALL POINT(BFR,ADDR(GTSRCE,OPCD));	00001740
LEN = 56+GTSRCE.DLNG;	00001750
CALL @WRITE(CMPLT1(1),BFR,LEN,LTIMF28);	00001760
IF CMPLT1(1) > 0 THEN GO TO NOINPT;	00001770
CALL @READ(CMPLT1(2),AFR,8,LTIMF28);	00001780
IF CMPLT1(2) > 0 THEN GO TO NOINPT;	00001790
AFR(1) = AFR(0)/(256**3);	00001800
IF AFR(1) = 5 THEN	00001810
DO;	00001820
CALL DISPLAY(	00001830
'NO INPUT. SMFS REPORTS COMPLETION CODE:' AFR(1));	00001840
IF AFR(1) = 42 AFR(1) = 22 AFR(1) = 23 AFR(1) = 32	00001850
AFR(1) = 33 AFR(1) = 34 AFR(1) = 39 THEN	00001860
RETURN(0);	00001870
END;	00001880
CALL @READ(CMPLT1(2),AFR,32,LTIMF28);	00001890
IF CMPLT1(2) > 0 THEN GO TO NOINPT;	00001900
LEN = AFR(0);	00001910
IF LEN > 0 THEN	00001920
DO;	00001930
CALL POINT(BFR,ADDR(INPUT));	00001940
CALL @READ(CMPLT1(2),BFR,LEN,LTIMF28);	00001950
IF CMPLT1(2) > 0 THEN GO TO NOINPT;	00001960
END;	00001970
ELSE	00001980
DO;	00001990
IF LEN = 0 THEN RETURN(0);	00002000
NOINPT:	00002010
IF TRACE = 1 THEN	00002020
DO;	00002030
CALL DISPLAY(	00002040
'NO INPUT');	00002050
CALL DISPLAY(	00002060
'NCP REPORTS COMPLETION CODE ' CMPLT1(1) ' ON SEND SOCKET');	00002070
CALL DISPLAY(	00002080
'NCP REPORTS COMPLETION CODE ' CMPLT1(2) ' ON RECEIVE SOCKET');	00002090
END;	00002100
RETURN(0);	00002110
END;	00002120
RETURN(LEN/8);	00002130
	00002140
	00002150
	00002160
	00002170
	00002180
POINT: PROCEDURE(I,J);	00002190
DCL (I,J) POINTER;	00002200
I = J;	00002210
END POINT;	00002220
	00002230
	00002240
	00002250
DISPLAY: PROCEDURE(TXT);	00002260
DCL TXT CHAR (72) VAR;	00002270
WRTDIAG.DATA = TXT;	00002280

CALL POINT(HFR,ADDR(WRTDIAG,NPCD));	00002290
IF NODIAGFL = 0 THEN	00002300
DO:	00002310
LEN = 56+WRTDIAG.DLNG;	00002320
CALL @WRITE(CMPLT2(1),AFR,LEN,LTIME28);	00002330
IF CMPLT2(1) = 0 THEN	00002340
DO:	00002350
CALL @READ(CMPLT2(2),AFR,8,LTIME28);	00002360
IF CMPLT2(2) = 0 THEN	00002370
DO:	00002380
AFR(1) = AFR(0)/(256**3);	00002390
IF AFR(1) = 3 THEN RETURN;	00002400
END:	00002410
END:	00002420
IF NODIAGFL = 0 THEN	00002430
DO:	00002440
PUT SKIP LIST (	00002450
'UNABLE TO USE DIAGNOSTIC FILE. OUTPUT DIVERTED TO SYSPRINT');	00002460
PUT SKIP LIST(	00002470
'SMFS REPORTS COMPLETION CODE:',AFR(1));	00002480
PUT SKIP LIST(	00002490
'NCP REPORTS COMPLETION CODE:',CMPLT2(1),' ON SEND SOCKET');	00002500
PUT SKIP LIST(	00002510
'NCP REPORTS COMPLETION CODE:',CMPLT2(2),' ON RECEIVE SOCKET');	00002520
NODIAGFL = 1;	00002530
END:	00002540
END:	00002550
PUT SKIP LIST(WRTDIAG.DATA);	00002560
RETURN;	00002570
END DISPLAY;	00002580
END SMFSIO;	00002590

```

@OPEN: PROCEDURE(CMPCD,TIME,LCLSCK,FGNSCK,WS):
00000010
00000020
00000030
00000040
00000050
00000060
00000070
00000080
00000090
00000100
00000110
00000120
00000130
00000140
00000150
00000160
00000170
00000180
00000190
00000200
00000210
00000220
00000230
00000240
00000250
00000260
00000270
00000280
00000290
00000300
00000310
00000320
00000330
00000340
00000350
00000360
00000370
00000380
00000390
00000400
00000410
00000420
00000430
00000440
00000450
00000460
00000470
00000480
00000490
00000500
00000510
00000520
00000530
00000540
00000550
00000560
00000570

/* SHARED VARIABLES */

DCL TRACE BIN FIXED (31) STATIC EXT:
DCL WRKSPS1(2) BIN FIXED (31) ALIGNED STATIC EXT:
DCL WRKSPS2(2) BIN FIXED (31) ALIGNED STATIC EXT:
DCL WRKSPS3(2) BIN FIXED (31) ALIGNED STATIC EXT:
DCL WRKSPS4(2) BIN FIXED (31) ALIGNED STATIC EXT:

/* LOCAL VARIABLES */

DCL EOF BIN FIXED (31) STATIC EXT INIT (0);
DCL TB BIN FIXED (31) STATIC INITIAL (0);
DCL TC BIN FIXED (31) STATIC:
DCL TD BIN FIXED (15) STATIC INITIAL (0);
DCL TEMP BIN FIXED (31) STATIC INITIAL (0);
DCL I BIN FIXED STATIC:
DCL J BIN FIXED STATIC INITIAL (1);
DCL K BIN FIXED STATIC:
DCL TEXT CHAR(80) STATIC INITIAL ('');
DECLARE CMPCD BIN FIXED (31);
DECLARE TIME BIN FIXED (31);
DECLARE LCLSCK(2) BIN FIXED (31);
DECLARE FGNSCK(2) BIN FIXED (31);
DECLARE WS(2) BIN FIXED (31);
DCL BFR(0:50) BIN FIXED (31);
DCL RESPONSE(0:50) BIN FIXED (31);
DECLARE LEN BIN FIXED (31);
DCL STRCT CHAR (75) STATIC EXT:

IF TRACE = 1 THEN
PUT SKIP LIST('@OPEN:',CMPCD,TIME,LCLSCK,FGNSCK,WS):
CMPCD = 0;
RETURN:

@CLOSE: ENTRY(CMPCD,TIME):
IF TRACE = 1 THEN
PUT SKIP LIST('@CLOSE:',CMPCD,TIME):
CMPCD = 0;
RETURN:

@READ: ENTRY(CMPCD,RESPONSE,LEN,TIME):
RESPONSE(0) = TB*(256**3);
IF LEN = 8 THEN RETURN:
RESPONSE(0) = TEMP;
IF LEN = 32 THEN RETURN:
RESPONSE(0) = 0;
IF LEN >= 0 & TB = 5 THEN
DO;
IF EOF = 1 THEN GO TO SIGNAL:
ON ENDFILE (SYSIN) EOF = 1;

```

GFT EDIT (TEXT) (A(RO));	00000580
IF EOF = 1 THEN GO TO SIGNAL;	00000590
K = (LEN/32);	00000600
DO I = 0 TO K-1;	00000610
RESPONSE(I) = UNSPEC(SUBSTR(TEXT,(4*I)+1,4));	00000620
END;	00000630
END;	00000640
IF TRACE = 1 THEN	00000650
PUT SKIP EDIT('READ:',CMPCD,RESPONSE,LEN,TIME)	00000660
(A,X(1),F(2),51(X(1),R(32)),X(1),F(5),X(1),F(5));	00000670
CMPCD = 0;	00000680
RETURN;	00000690
SIGNAL:	00000700
CMPCD = 20;	00000710
WRKSPS1(2) = 80;	00000720
RETURN;	00000730
	00000740
	00000750
@WRITE: ENTRY(CMPCD,BFR,LEN,TIME):	00000760
TR = BFR(0)/(256**3);	00000770
TC = (BFR(0)-TR*(256**3))/256;	00000780
TEMP = 0;	00000790
IF TR >= 2 & TR <= 6 THEN	00000800
DO;	00000810
IF TC = 8256 TC = 8192 THEN TEMP = BFR(1)/256;	00000820
ELSE TEMP = BFR(10);	00000830
IF TC = 8256 THEN	00000840
DO;	00000850
CALL POINT(STRCT,ADDR(BFR(1)));	00000860
PUT SKIP LIST(STRCT);	00000870
CMPCD = 0;	00000880
RETURN;	00000890
END;	00000900
END;	00000910
IF TD = 3 THEN TD = TR;	00000920
ELSE	00000930
DO;	00000940
TR = TD;	00000950
TD = 0;	00000960
END;	00000970
IF TRACE = 1 THEN	00000980
PUT SKIP EDIT('@WRITE:',CMPCD,BFR,LEN,TIME,TR,TC,TEMP)	00000990
(A,X(1),F(2),51(X(1),R(32)),5(X(1),F(5)));	00001000
CMPCD = 0;	00001010
RETURN;	00001020
	00001030
	00001040
POINT: PROCEDURE(I,J);	00001050
DCL (I,J) POINTER;	00001060
I = J;	00001070
END POINT;	00001080
END @OPEN;	00001090

Appendix D

EXAMPLE COMPILATION
(Diagnostic File)

0 K (, B , , 2) : (K .EQ. 2 : F (4)) ;

ISN	INSTRCT	OPCD	OPRND	SYMBOL
0	2 2 4 1	SICP		
1	5 0 0 0	NULL		
2	1 1	IC	1	
3	5 0 0 0	NULL		
4	1 2	IC	2	
5	2 2 5 0	INS		
6	3 10	AD	10	
7	2 2 2 1	BF		
8	0 0	LD	0	K
9	2 2 0 0	STO		
10	3 24	AD	24	
11	2 2 2 1	BF		
12	6 0	EOT	0	
13	2 2 4 0	SCIP		
14	0 0	LD	0	K
15	1 2	IC	2	
16	2 2 3 0	SEQ		
17	3 22	AD	22	
18	2 2 2 0	BT		
19	1 4	IC	4	
20	2 1 2 0	LUL		
21	2 2 2 2	BU		
22	6 1	EOT	1	
23	7 0	EOR	0	

1 L (, B , , 1) , J (, B , , 5) : (L .EQ. 0 : F (3)) ;

ISN	INSTRCT	OPCD	OPRND	SYMBOL
24	2 2 4 1	SICP		
25	5 0 0 0	NULL		
26	1 1	IC	1	
27	5 0 0 0	NULL		
28	1 1	IC	1	
29	2 2 5 0	INS		
30	3 34	AD	34	
31	2 2 2 1	BF		
32	0 1	LD	1	L
33	2 2 0 0	STO		
34	3 60	AD	60	
35	2 2 2 1	BF		
36	6 0	EOT	0	
37	5 0 0 0	NULL		
38	1 1	IC	1	
39	5 0 0 0	NULL		
40	1 5	IC	5	
41	2 2 5 0	INS		

42	3	46	AD	46
43	2	2 2 1	BF	
44	0	2	LD	2 J
45	2	2 0 0	STO	
46	3	60	AD	60
47	2	2 2 1	BF	
48	6	1	EOT	1
49	2	2 4 0	SCIP	
50	0	1	LD	1 L
51	1	0	IC	0
52	2	2 3 0	CEQ	
53	3	58	AD	58
54	2	2 2 0	BT	
55	1	3	IC	3
56	2	1 2 0	LUL	
57	2	2 2 2	BU	
58	6	2	EOT	2
59	7	1	EOR	1

2 : (J , E , E " " , 1 : U (0)) ;

ISN	INSTRCT	OPCD	OPRND	SYMBOL
60	2 2 4 1	SICP		
61	2 2 4 0	SCIP		
62	0	2	LD	2 J
63	1	4	IC	4
64	0	3	LD	3
65	1	1	IC	1
66	2 2 6 0	OUT		
67	1	0	IC	0
68	2 1 2 0	LUL		
69	2 2 2 2	BU		
70	3	74	AD	74
71	2 2 2 1	BF		
72	6	0	EOT	0
73	7	2	EOR	2

3 CHAR (, E , , 1) : (J , E , CHAR , 1 : U (0)) ;

ISN	INSTRCT	OPCD	OPRND	SYMBOL
74	2 2 4 1	SICP		
75	5 0 0 0	NULL		
76	1	4	IC	4
77	5 0 0 0	NULL		
78	1	1	IC	1
79	2 2 5 0	INS		
80	3	84	AD	84
81	2 2 2 1	BF		
82	0	4	LD	4 CHAR
83	2 2 0 0	STO		

84	3	100	AD	100	
85	2	2 2 1	BF		
86	6	0	EOT	0	
87	2	2 4 0	SCIP		
88	0	2	LD	2	J
89	1	4	IC	4	
90	0	4	LD	4	CHAR
91	1	1	IC	1	
92	2	2 6 0	OUT		
93	1	0	IC	0	
94	2	1 2 0	LUL		
95	2	2 2 2	BU		
96	3	100	AD	100	
97	2	2 2 1	BF		
98	6	1	EOT	1	
99	7	3	EOR	3	

4 LJ (, 6 , , 6) , CHRS (LJ , E , , 1) : (, E , CHRS , L (CHRS) :
U (0)) ;

ISN	INSTRCT	OPCD	OPRND	SYMBOL
100	2 2 4 1	SCIP		
101	5 0 0 0	NULL		
102	1	IC	1	
103	5 0 0 0	NULL		
104	1	IC	6	
105	2 2 5 0	INS		
106	3	AD	110	
107	2 2 2 1	BF		
108	0	LD		LJ
109	2 2 0 0	STO		
110	3	AD	139	
111	2 2 2 1	BF		
112	6	EOT	0	
113	0	LD	5	LJ
114	1	IC	4	
115	5 0 0 0	NULL		
116	1	IC	1	
117	2 2 5 0	INS		
118	3	AD	122	
119	2 2 2 1	BF		
120	0	LD	6	CHRS
121	2 2 0 0	STO		
122	3	AD	139	
123	2 2 2 1	BF		
124	6	EOT	1	
125	2 2 4 0	SCIP		
126	5 0 0 0	NULL		
127	1	IC	4	
128	0	LD	6	CHRS

129	0	6	LD	6	CHRS
130	2	1 1 1	LIL		
131	2	2 6 0	OUT		
132	1	0	IC	0	
133	2	1 2 0	LUL		
134	2	2 2 2	BU		
135	3	139	AD	139	
136	2	2 2 1	BF		
137	6	2	EOT	2	
138	7	4	EOR	4	

* END OF FORM */

***** LABEL TABLE *****

ENTRY	LABEL	OFFSET
0	0	0
1	1	0
2	2	0
3	3	0
4	4	0

***** IDENTIFIER TABLE *****

ENTRY	IDENTIFIER
0	K
1	L
2	J
4	CHAR
5	LJ
6	CHRS

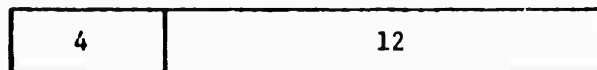
***** LITERAL TABLE *****

ENTRY	TYPE	LENGTH	OFFSET	LITERAL
3	E	8	0	01000000

COMPILATION TERMINATED

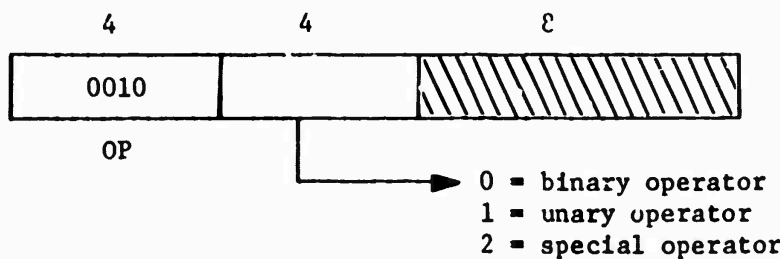
Appendix E

OBJECT LANGUAGE INSTRUCTION FORMATS

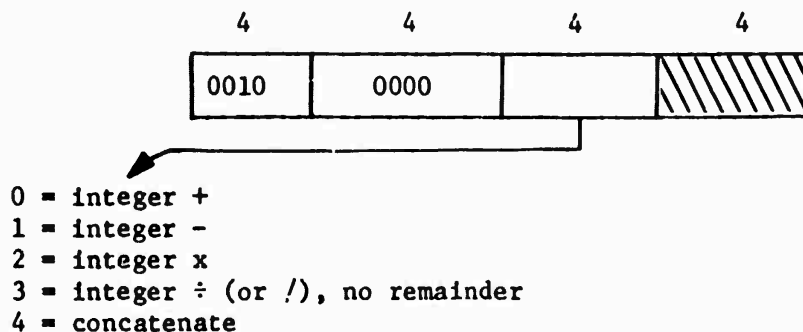


LD = 0 literal or identifier reference
 IC = 1 12-bit 2's complement integer constant
 OP = 2 operator
 AD = 3 address (12-bit positive integer)
 ARB = 4 indefinite replication factor
 NULL = 5 missing attribute of term

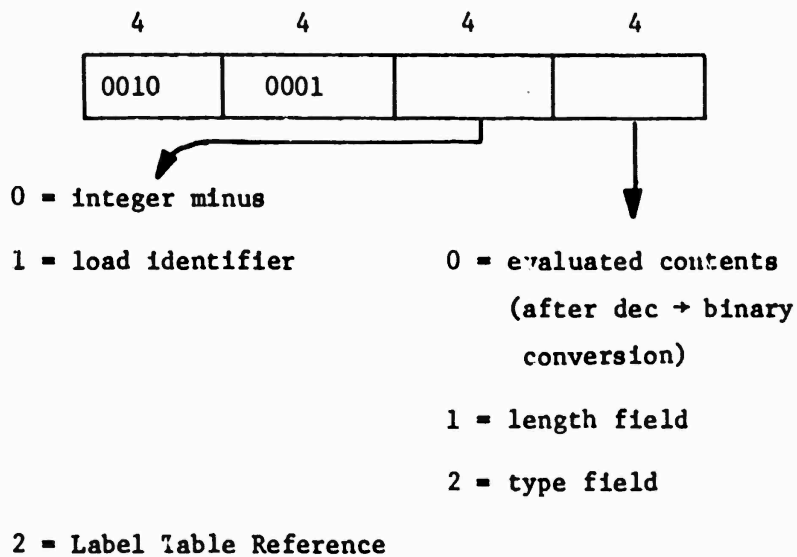
Basic Instruction Format



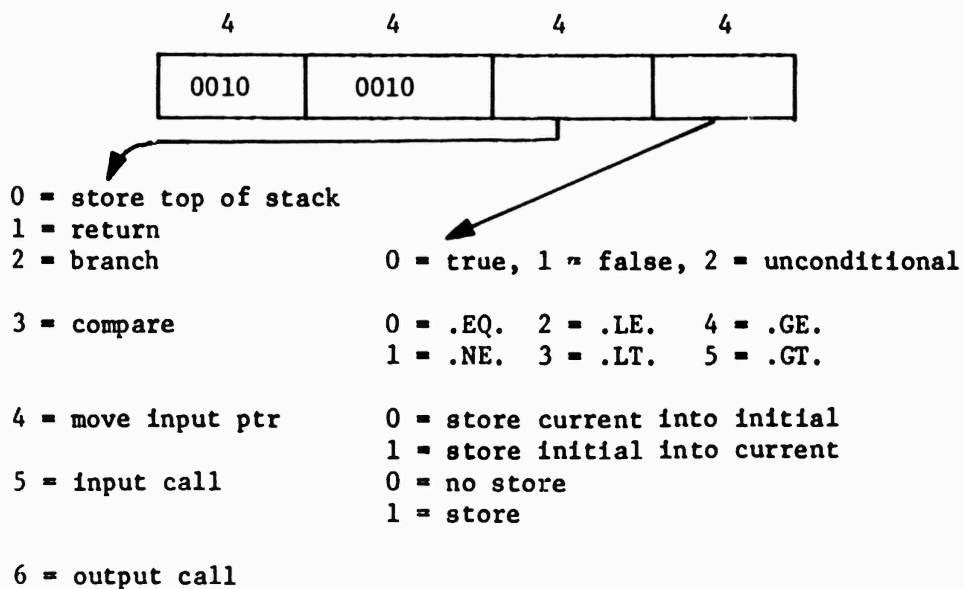
Operator Format



Binary Operator Encoding



Unary Operator Encoding



Special Operators Encoding

Appendix F

FLOWCHARTS OF COMPILER

Main routine:
Enter with the name of
the input file.

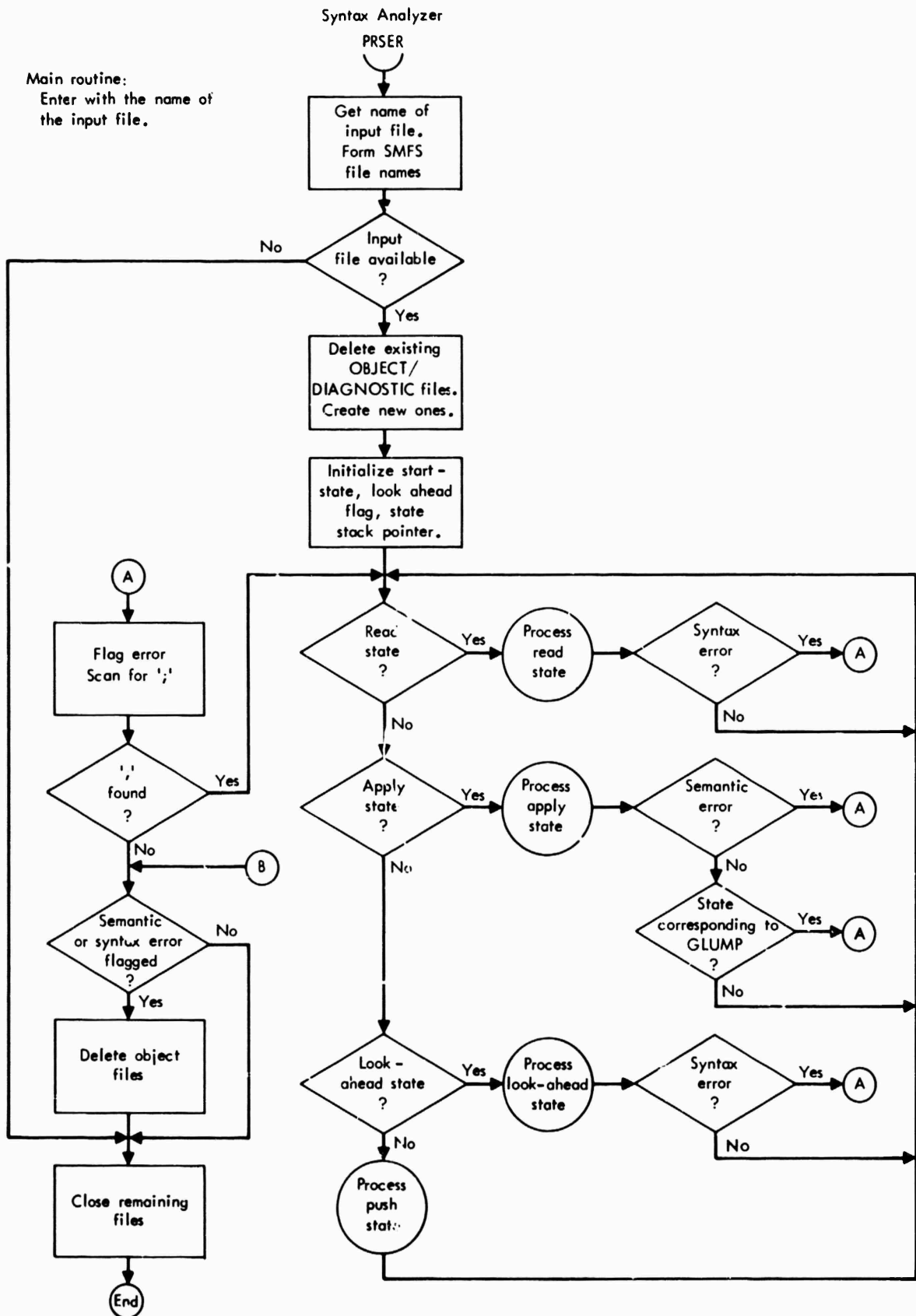


Fig. 10--Syntax Analysis Routine: Control Loop

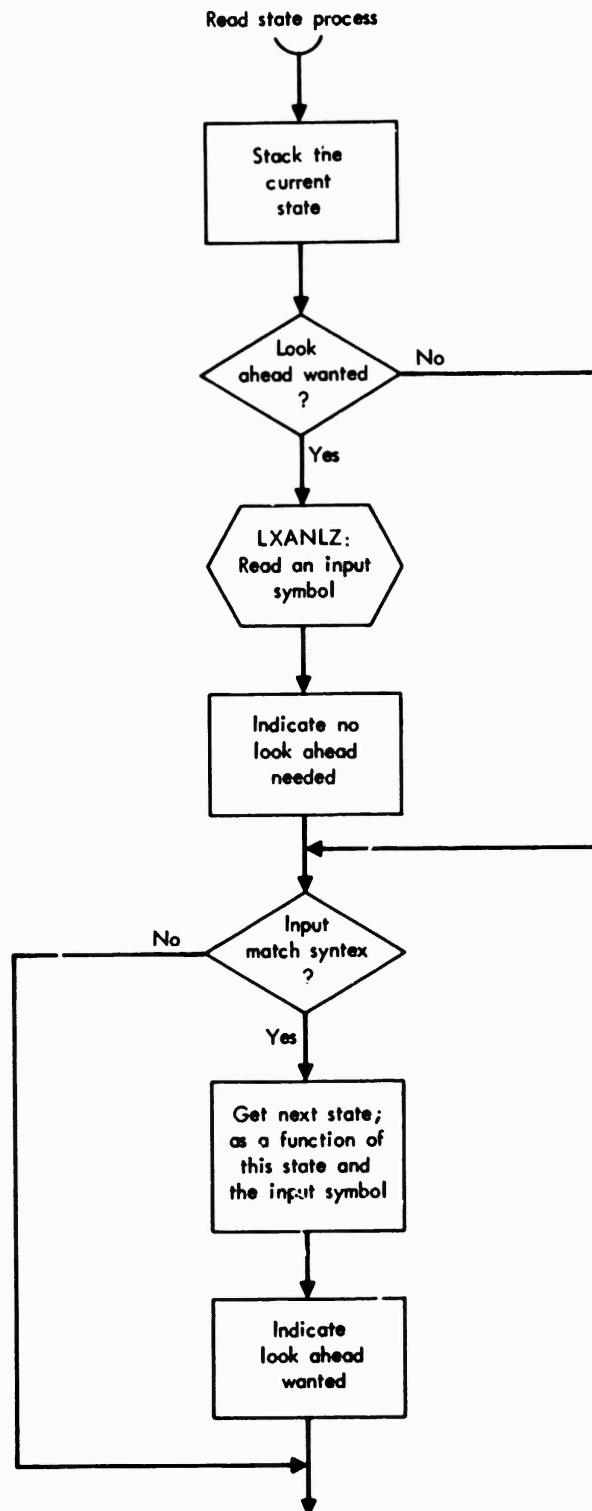


Fig. 11--Syntax Analysis Routine: Processing the Read State

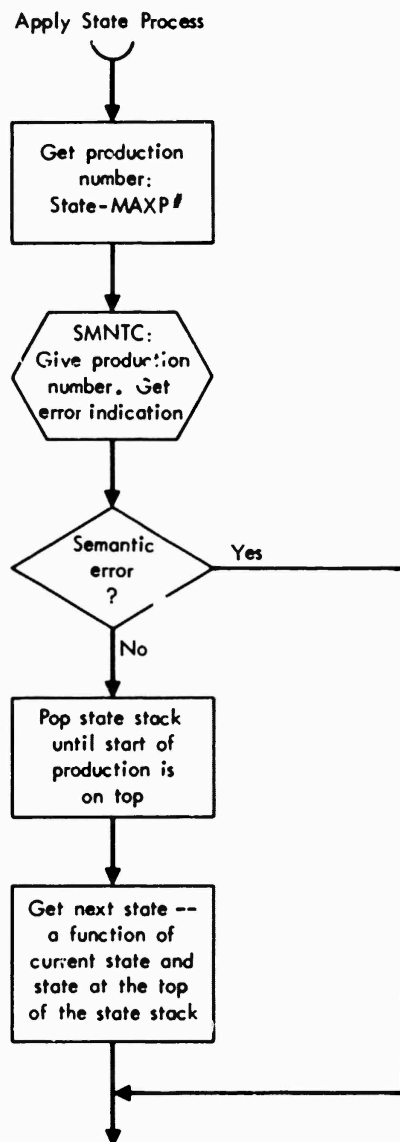


Fig. 12--Syntax Analysis Routine: Processing the Apply State

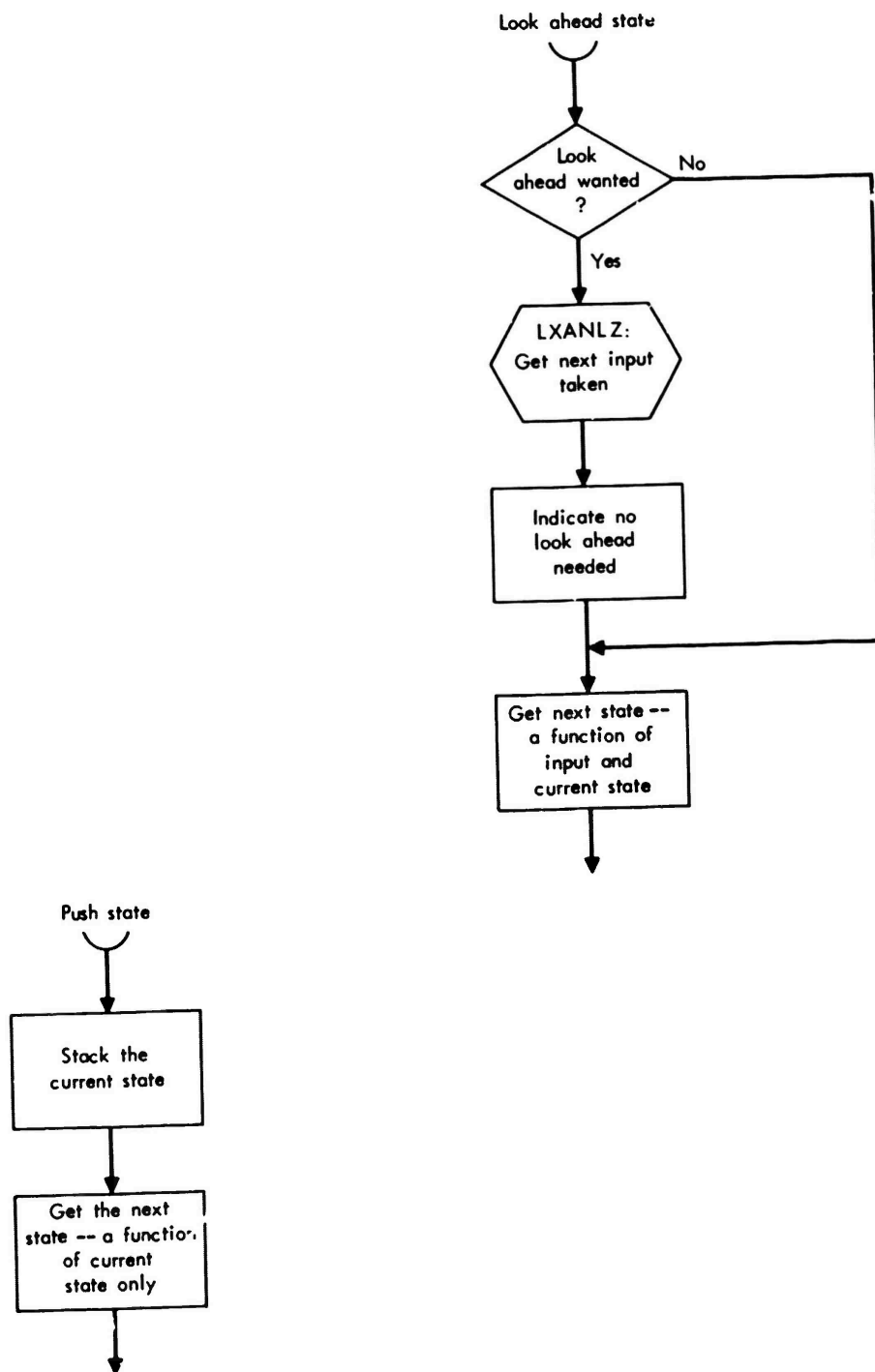


Fig. 13--Syntax Analysis Routine: Processing the Look-Ahead and Push States

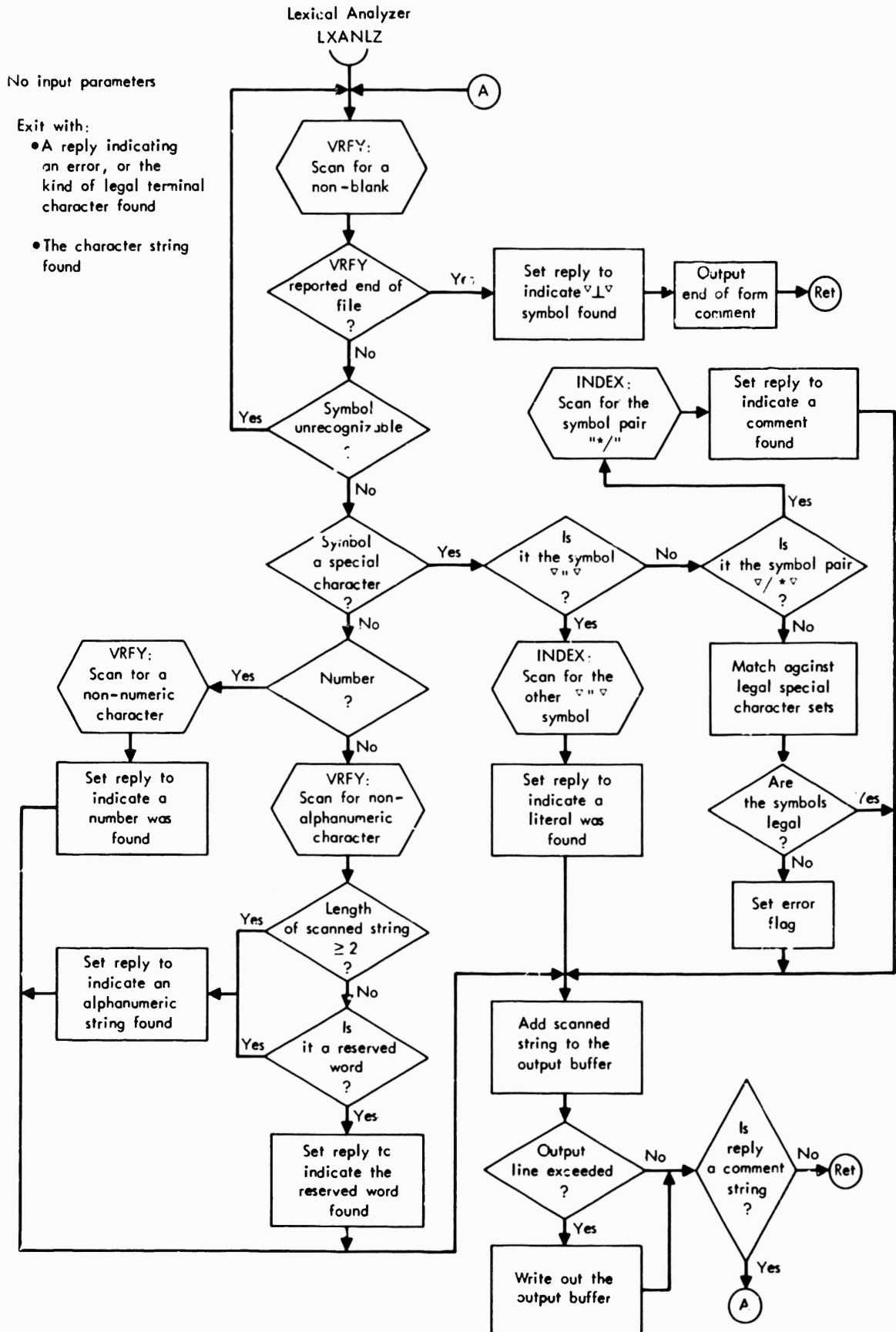


Fig. 14--Lexical Analysis Routine

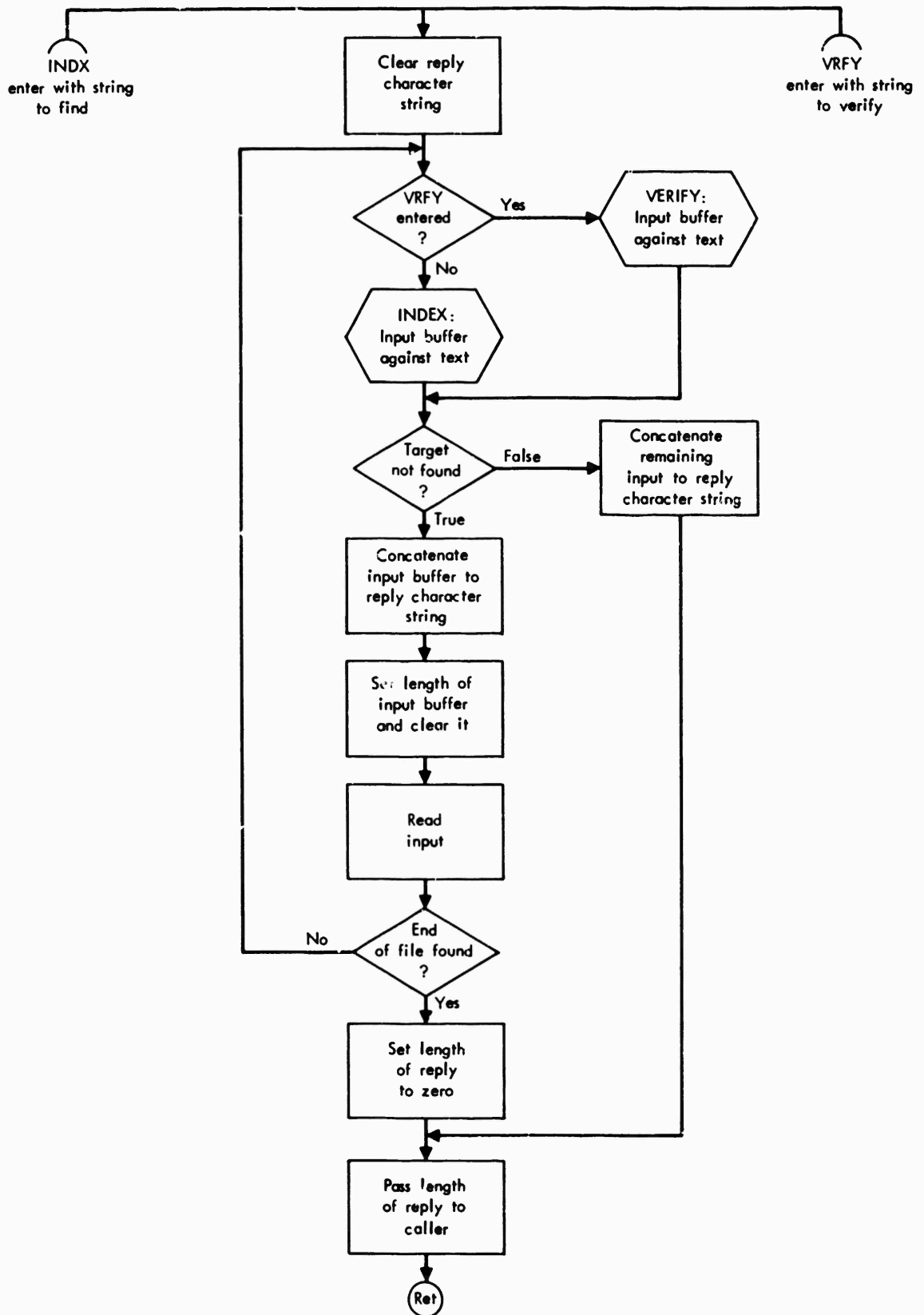


Fig. 15--Lexical Analysis Routine: Verify and Index Subroutines

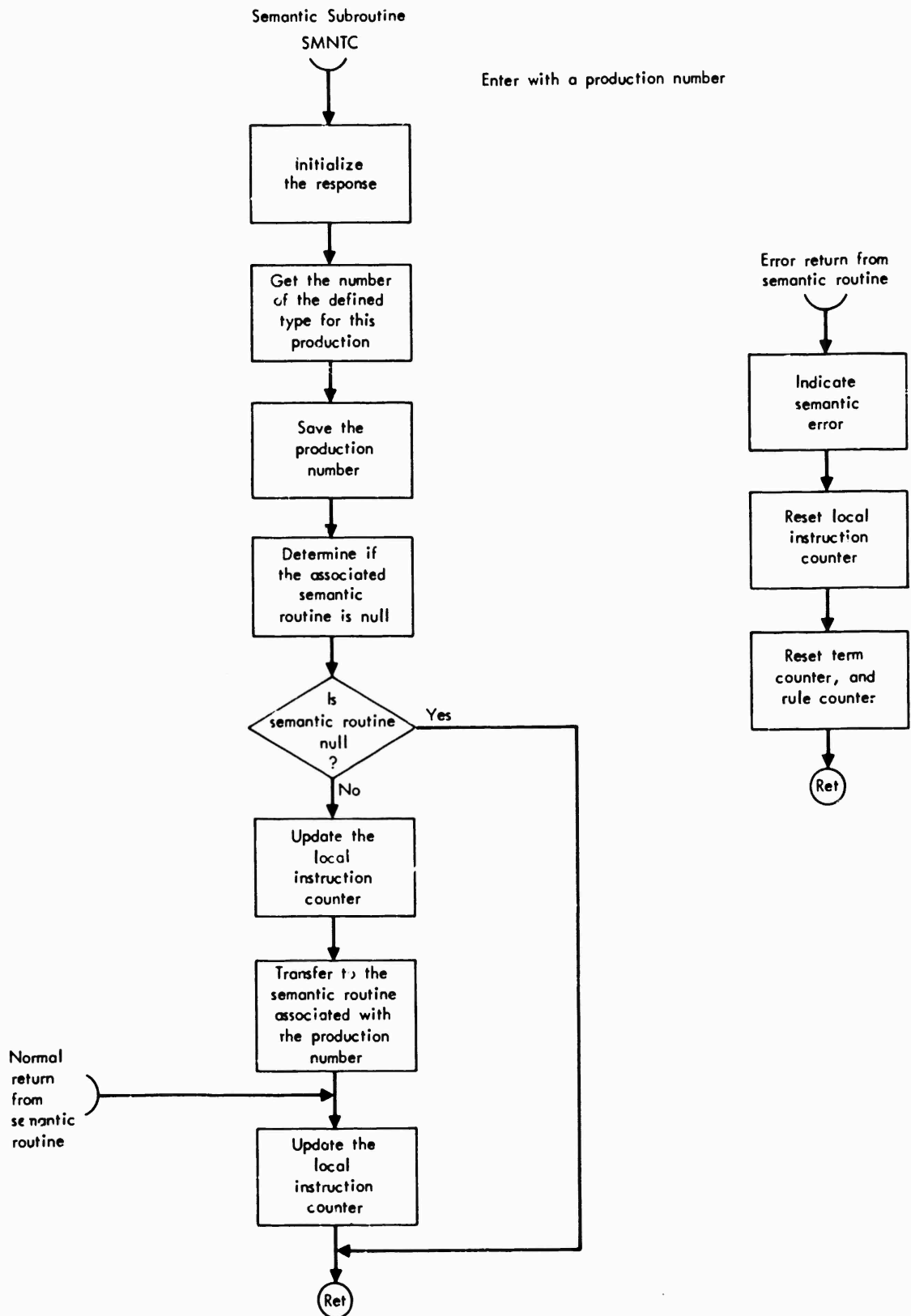


Fig. 16--Semantic Routine: Control Loop

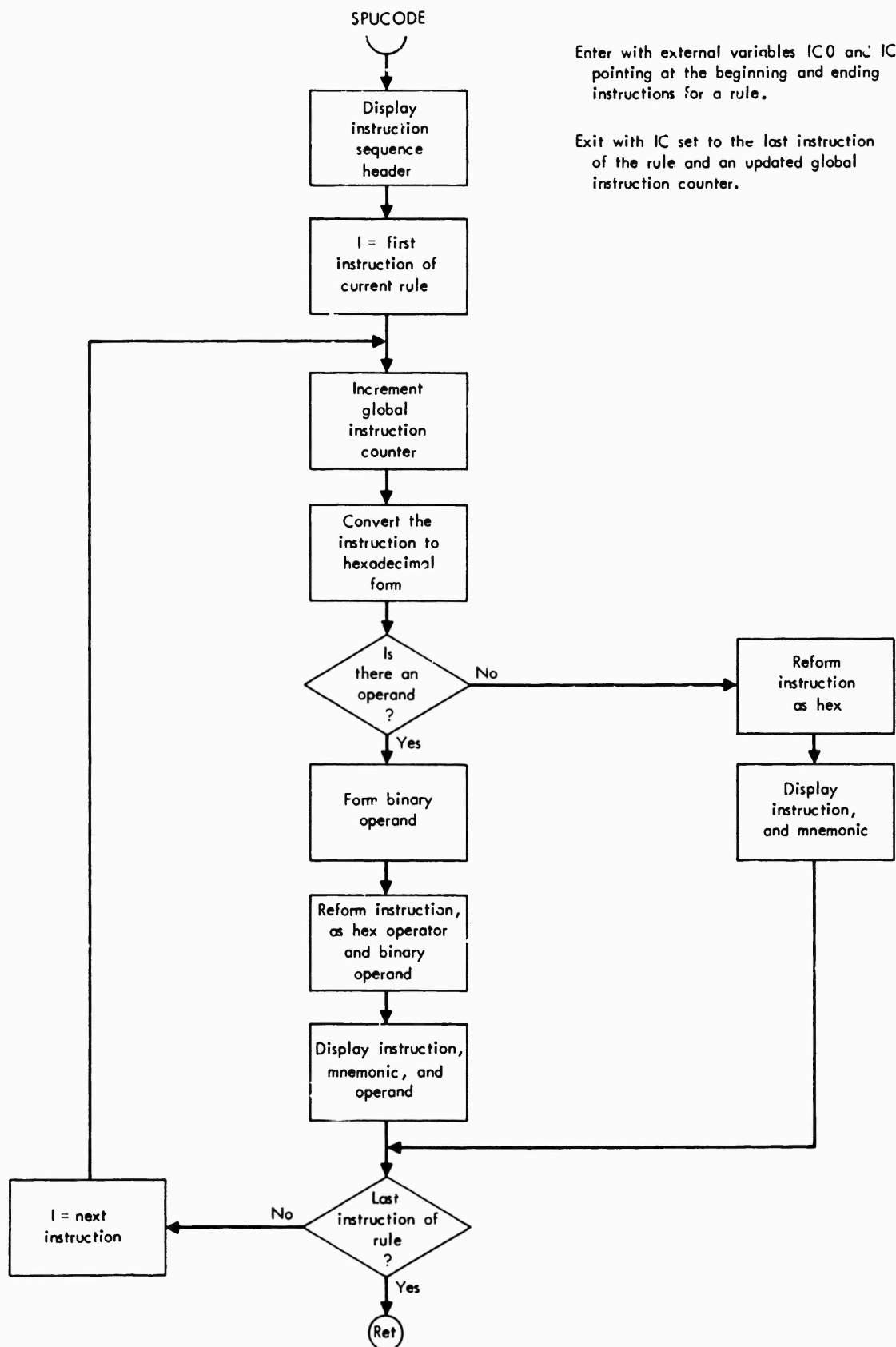


Fig. 17--Semantic Routine: Printing the Instruction Lists

Input/output routine
SMFSIO

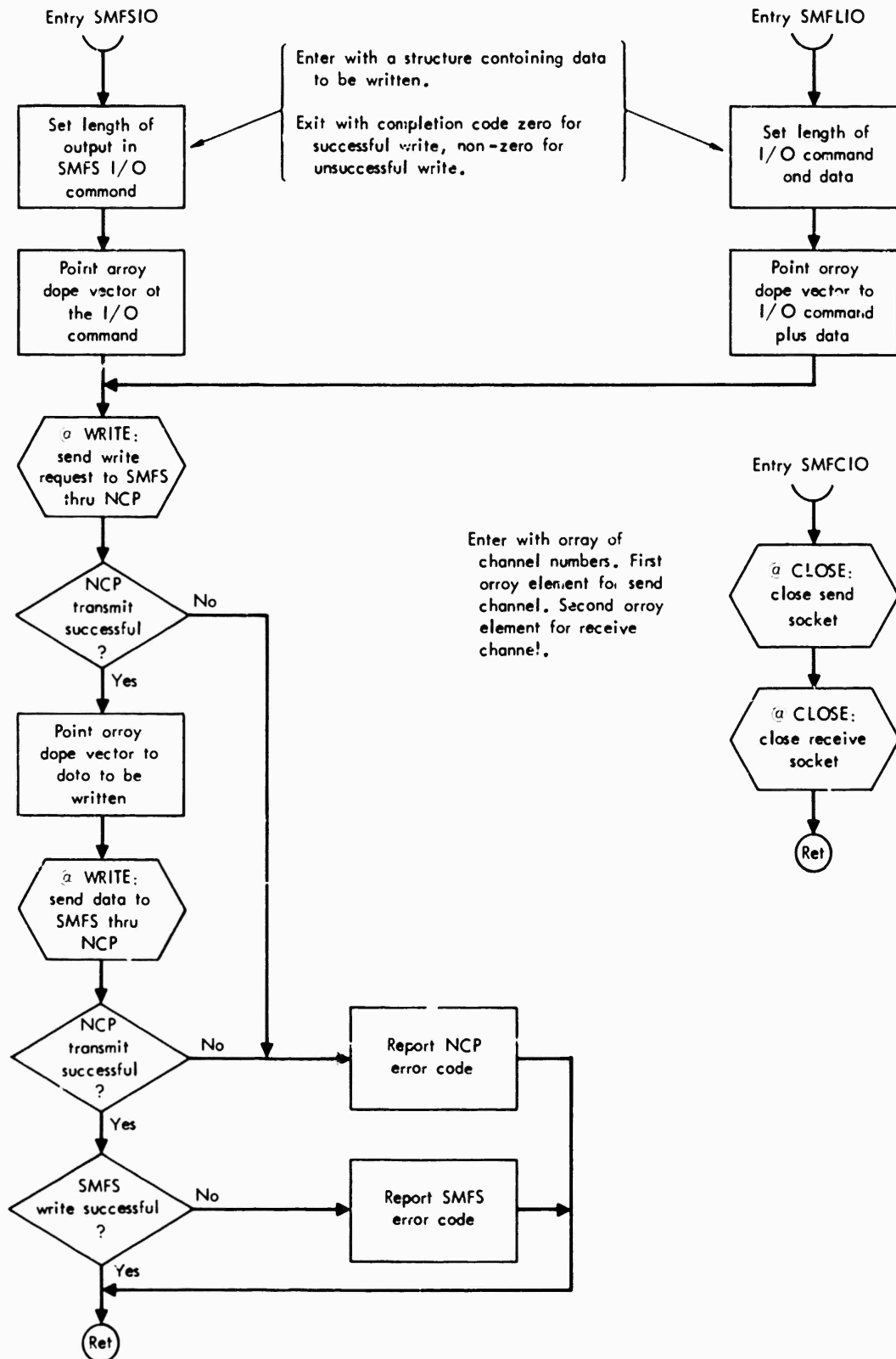


Fig. 18--Input/Output Routine: Executing SMFS Channel Commands and Closing SMFS Files

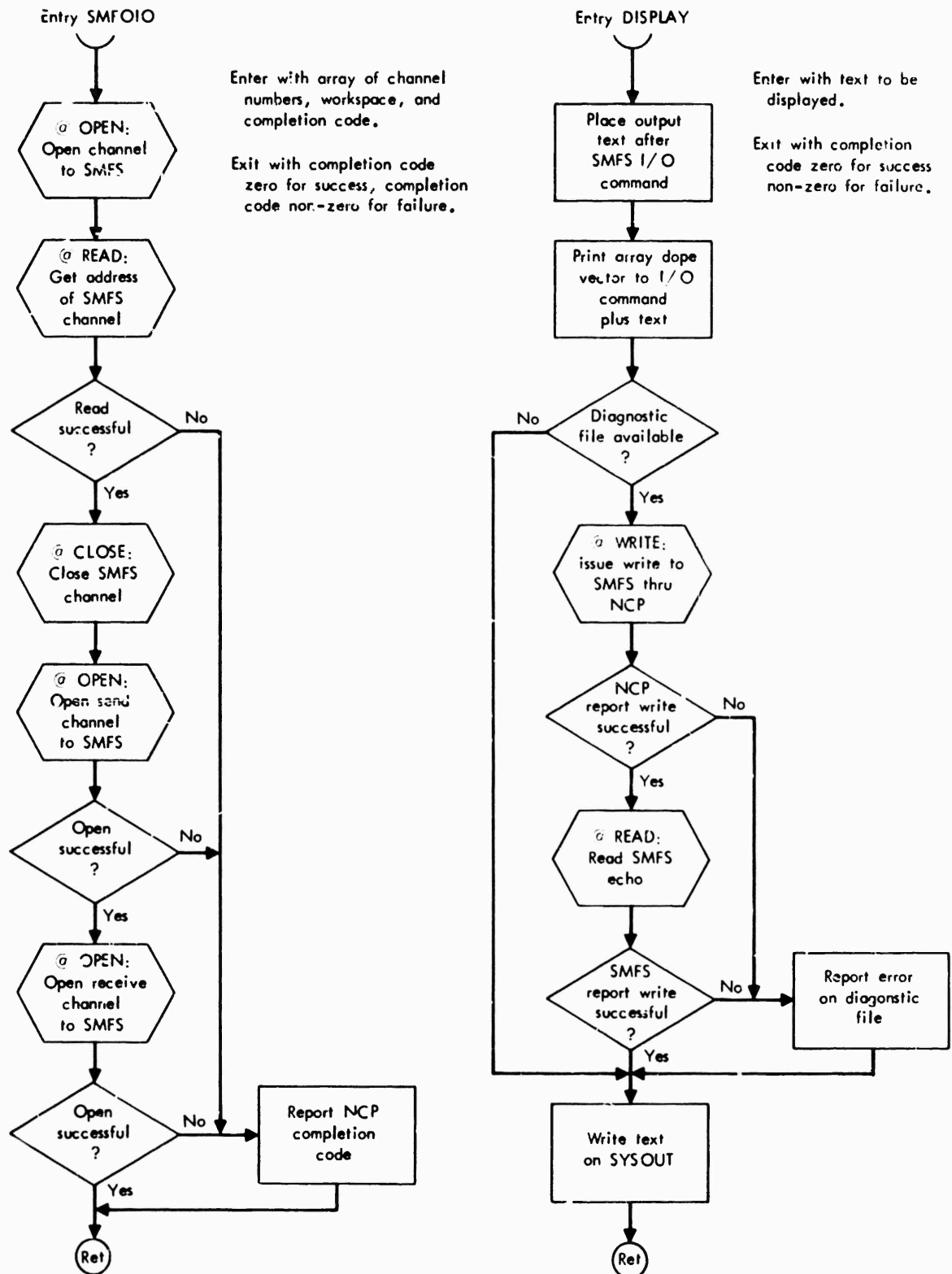


Fig. 19--Input/Output Routine: Opening and Writing an SMFS File

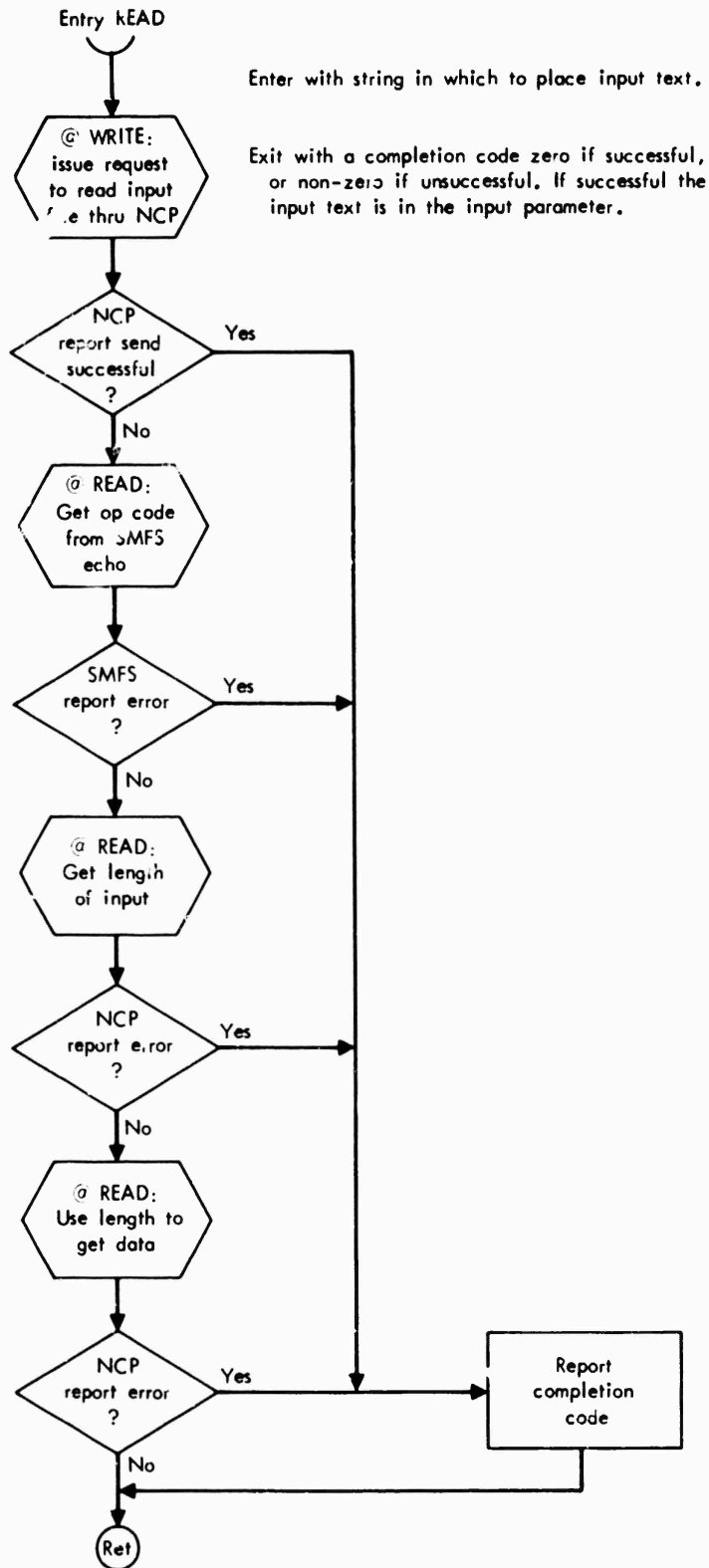


Fig. 20--Input/Output Routine: Reading an SMFS File

REFERENCES

1. Ellis, T. O., E. F. Harslem, J. F. Heafner, and K. W. Uncapher, *ARPA Network Series: I. Introduction to the ARPA Network at Rand and to the Rand Video Graphics System*, The Rand Corporation, R-664-ARPA, September 1971.
2. Roberts, L. G., and B. D. Wessler, "Computer Network Development to Achieve Resource Sharing," *AFIPS Conference Proceedings*, Vol. 36, 1970, pp. 543-549.
3. Heart, F. E., R. E. Kahn, S. M. Ornstein, W. R. Crowther, and D. C. Walden, "The Interface Message Processor for the ARPA Computer Network," *AFIPS Conference Proceedings*, Vol. 36, 1970, pp. 551-567.
4. Kleinrock, Leonard, "Analysis and Simulation Methods in Computer Network Design," *AFIPS Conference Proceedings*, Vol. 36, 1970, pp. 569-579.
5. Carr, C. S., S. D. Crocker, and V. G. Cerf, "HOST-HOST Communication Protocol in the ARPA Network," *AFIPS Conference Proceedings*, Vol. 36, 1970, pp. 589-597.
6. Anderson, R. H., V. Cerf, E. F. Harslem, J. F. Heafner, J. Madden, R. Metcalfe, A. Shoshani, J. White, and D. Wood, "The Data Reconfiguration Service--An Experiment in Adaptable, Process/Process Communication," Presented at the Second Symposium on Problems in the Optimization of Data Communication Systems, sponsored by the SIGCOMM of ACM and the IEEE Computer Society, Palo Alto, California, October 1971. (Also see R-860-ARPA, *The Data Reconfiguration Service--An Experiment in Adaptable Process/Process Communication*, E. F. Harslem and J. F. Heafner, The Rand Corporation, October 1971.)
7. Cerf, V. G., E. F. Harslem, J. F. Heafner, B. Metcalfe, and J. White, "An Experimental Service for Adaptable Data Reconfiguration," to be published in *IEEE Transactions on Communication Technology*, June 1972.
8. LaLonde, W. R., *User's Guide to the LALR(k) Parser Generator*, University of Toronto, Computer Systems Research Group, April 1971.
9. -----, *An Efficient LALR Parser Generator*, University of Toronto, Computer Systems Research Group, CSRG-2, 1970.
10. Naur, P., *et al.*, "Revised Report on the Algorithmic Language ALGOL 60," *Communications of the Association for Computing Machinery*, Vol. 6, No. 1, January 1963.
11. White, J. E., *Network Specifications for UCSB's SIMPLE-MINDED FILE SYSTEM*, Network Information Center, Stanford Research Institute, Menlo Park, California, NIC 5834, April 26, 1971.

12. McKeeman, W. M., J. J. Horning, and D. B. Wortman, *A Compiler Generator*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1970.
13. Krilanovich, M., *Network PL/1 Subprograms*, Network Information Center, Stanford Research Institute, Menlo Park, California. NIC 5832, April 1971.
14. Shaw, J. C., "JOSS: A Designer's View of an Experimental On-Line Computing System," *AFIPS Conference Proceedings*, Vol. 26, Part 1, 1964, pp. 455-464.
15. *Conversational Programming System (CPS) Terminal User's Manual*, (360D-03.4-016), International Business Machines Corporation, Form No. GH200758-0, January 1970.
16. Baran, T., "On Distributed Communication Networks," *IEEE Transactions on Communication Systems*, Vol. CS-12, March 1964.
17. Cheatham, T. E., Jr., and Kirk Sattley, "Syntax-Directed Compiling," *AFIPS Conference Proceedings*, Vol. 25, 1964, pp. 31-57.
18. Marill, T., and L. G. Roberts, "Toward a Cooperative Network of Time-Shared Computers," *AFIPS Conference Proceedings*, Vol. 29, 1966, pp. 425-431.