1. (a) I give permission for my thesis, entitled

COBOL Program Analyser and Workbench ---- Phase 1

to be made available to readers in the Library under the conditions determined by the Librarian.

(b) I agree to my thesis, if asked for by another institution, being sent away on temporary loan under conditions determined by the Librarian.

(c) I also agree that my thesis may be copied for Library use.

2. I do not wish my thesis, entitled

~~COBOL Program Analyser and Workbench ---- Phase 1~~

to be made available to readers or to be sent to other institutions without my written consent within the next two years.

Signed ..... Kaw Ming, Tan.

Date 12/9/85

Strike out the sentence or phrase which does not apply.

The Library
Massey University
Palmerston North, N.Z.

The copyright of this thesis belongs to the author. Readers must sign their name in the space below to show that they recognise this. They are asked to add their permanent address.

Name and Address Date


FLOREAT·SCIENTIA

COBOL PROGRAM ANALYSER

AND

WORKBENCH --- PHASE I

A thesis presented in partial fulfilment

of the requirements for

the degree of **MASTER of SCIENCE**

in COMPUTER SCIENCE

at MASSEY UNIVERSITY

by

TAN, KAY MENG B Sc (Hons)

Augest 1985.

ABSTRACT

This dissertation addresses a number of important software maintenance problems which fall within the framework of software management and software engineering. Compiling techniques were used to present in a standard form the structure of the COBOL program PROCEDURE DIVISION; as well as providing a framework for some complexity measures of a given COBOL program. The latter can give some measure of the ease of software modification during the software maintenance phase.

The purpose of the project of which this thesis is a part is to build up a database which will completely specify a COBOL program and enable it to be analysed in various ways related to maintenance, enhancement and other tasks.

A COBOL program information system (MUCAS) has been designed for use by the applications programmer, to acquire the information about a COBOL program (e.g. complexity measures, control flow analysis) by analysing the structure of the COBOL program. Various software methods such as structured programming, structured design, top-down design and bottom-up program testing were used throughout the MUCAS system development process. The finite-state machine concept was used to construct a **COBOL lexical scanner**. The COBOL programming syntax was parsed by the top-down recursive-descent method. The COBOL program was reformatted according to a coding standard to show the structure of the COBOL program.

Many complexity measures are mentioned in this thesis. A combination complexity measure method and guidelines are suggested. A COBOL program complexity factor is proposed. In general, the program complexity measure could apply to other programming languages.

ACKNOWLEDGEMENTS

Firstly, I thank my supervisor **Prof. Graham Tate** for his constructive efforts and supervision.

Secondly, I am grateful to those people who also assisted and directed my research. Although there are many who give me both ideas and encouragement, I specifically thank **Mr. Peter Parkin,** Mr. J. L. Alexander, Mrs. Nola M. Simpson, Mr. Chris Philips and Miss June M. Verner.

Thirdly, I acknowledge the many Systems Analysts, Managers, Computer Scientists and Computer programmers/operators who have enriched me with many ideas of relevance to this project during my association with them.

I appreciate the assistance by Mr. Kevin Kelliher and Mr. Brain Kirk for their proof-reading of my thesis.

I also thank my family for their financial and moral support which made this opportunity for further study possible.

Finally, I wish to recognise the following friends who helped me in the preparation of this thesis:

Mr. Ooi, Inn Tong
Mrs. Ooi, Phaik Kim
Miss Phuah, Phaik Lean
Mr. Wu, Chiaw Ching
Mr. Du, Yuchun
Mr. Walt Abell
Mr. Giovanni Moretti
Mr. John Holley

LIST OF FIGURES

ABBREVIATIONS

ACM	Association for Computing Machinery
ADP	Automatic Data Processing
AFIPS	American Federation of Information Processing Societies
ANS	American National Standards
ANSI	American National Standards Institute
BCS	British Computer Society
BNF	Backus-Naur Form
CAD	Computer-Aided Design
CAD/CAM	Computer-Aided Design/Computer-Aided Manufacturing
CAI	Computer-Aided Instruction
CAI	Computer-Assisted Instruction
CICS	Customer Information Control System
CF	Control Flow
CFG	Control Flow Graph
CLS	Cobol Lexical Scanner
CM	Complexity Measure
COBOL	COmmon Business Oriented Language
CODASYL	Conference on Data Systems Languages
CPU	Central Processing Unit
CVC	Control Variable Complexity
DBA	Direct Bucket Address
DBA	DataBase Administrator
DBMS	DataBase Management System
DMS	Data Manipulation System
EDP	Electronic Data Processing
FIPS	Federation Information Processing Standards
IBM FSD	IBM Federal Systems Division
HGL	Hierarchical Graph Language
HIPO	Hierarchy plus Input-Process-Output
IBM	International Business Machines
ICL	International Computers Ltd.
IDS	Integrated Data Store
IEEE	Institute of Electrical and Electronic Engineers
IFIP	International Federation for Information Processing
LL	Leftmost Looking grammar
LSI	Large-Scale Integration
MC	Module Complexity
MUCAS	Massey University Cobol program Analyser System
PC	Program Complexity
PCM	Program Complexity Measure
PL/I	Programming Language I
PSL/PSA	Problem Statement Language/Problem Specification Analyser
SD	Structured Design
SIGPLAN	Special Interest Group on Programming Languages (of ACM)
SP	Structured Programming
SRE	Software Requirements Engineering
ST	Symbol Table
UNIVAC	UNIVersal Automatic Computer
VLSI	Very Large-Scale Integration

TABLE OF CONTENTS **PAGE**

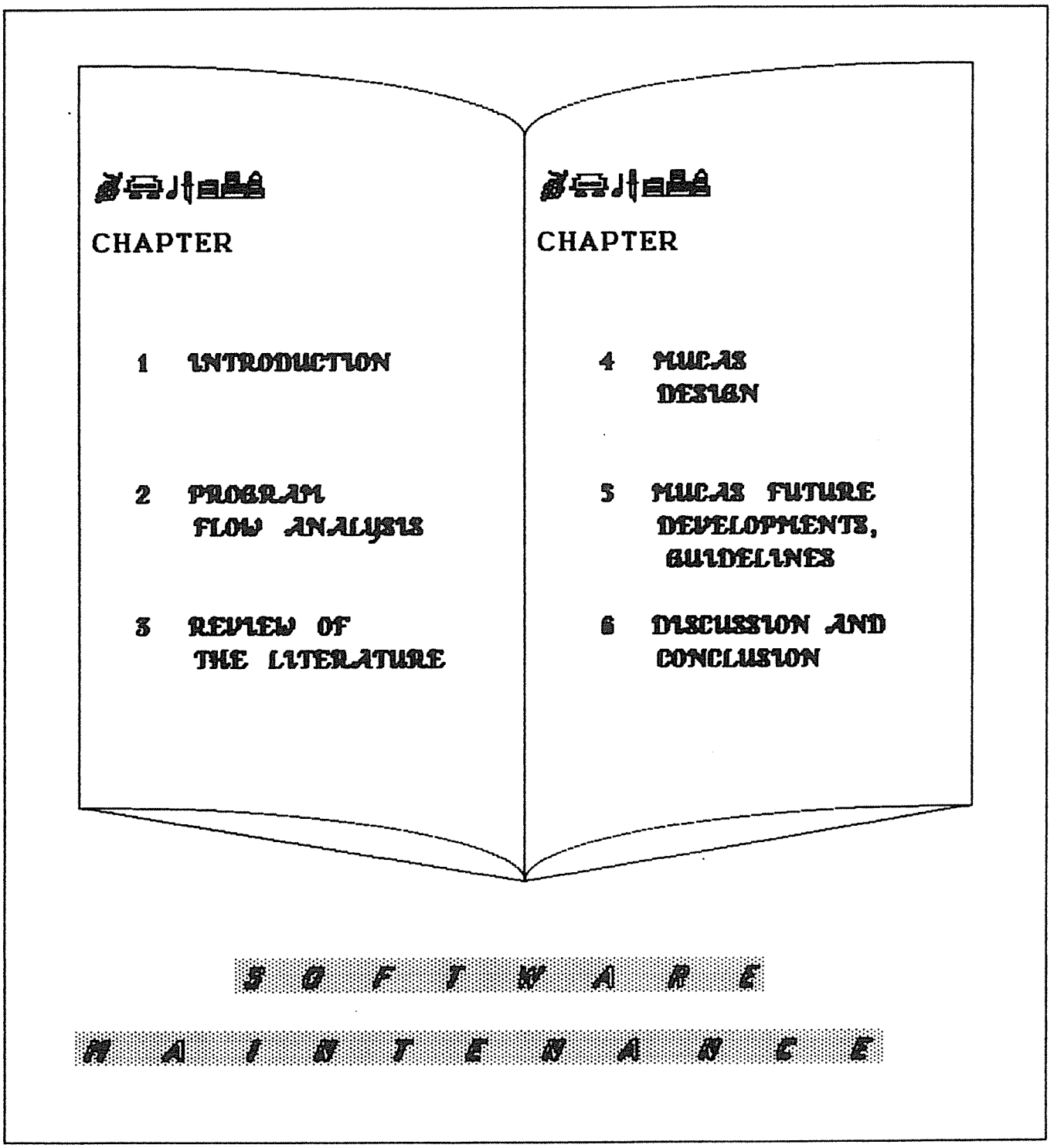


Figure P.1 What This Thesis is About.

CHAPTER 1

INTRODUCTION

CHAPTER 1 INTRODUCTION

The areas of **Software Management** and **Software Engineering** aim for an understanding of how to achieve reliability in software maintenance [BOE79] [FIS80]. There has been a worldwide software maintenance problem revealed by the computing literature ([WEX84] and Chapter 3 of this thesis). Too many production systems have been developed which are very costly or impossible to maintain [POD85] [MOR85]. Too often systems have become unmaintainable making it necessary to write new systems and to start another systems development cycle [GLA81].

For almost 30 years of computing history, the hardware cost has decreased greatly, whereas in recent years, the software cost has increased steadily [BRO82] [FOX82]. At present the majority of the software cost (about 70%) is due to maintenance [LEA83]. There is a need for new software tools which will help to solve software maintenance problems. The most significant problem of software maintenance is verifying the correctness of software modifications [BOY81] :

(a) What techniques can be used to implement this ?
(b) Are these techniques feasible to apply ?
(c) What are the implementation considerations ?

One attempt to solve these problems is the development of the ad hoc retest technique which uses graph theory [BEI78] and **0-1 integer** programming [FIS80], to examine performance characteristics, and practical implementation considerations. Although various methods have been tried, no universally acceptable approach for software modification and software maintenance has been agreed upon. A COBOL program information system **MUCAS** (**M**assey **U**niversity **C**obol program **A**nalyser **S**ystem) has been designed for use by the applications programmer, to acquire information about a COBOL program (e.g., the

complexity measure, control flow analysis) by analysing and reformatting the COBOL program. Developments of MUCAS will both advance computer software theory and benefit the management of the software maintenance process. The current computer complexity measure method approach to computer program testing will be extended to computer program maintenance. Practical benefits are anticipated to be significant because now managers will have a tool to analyse COBOL program. MUCAS will increase confidence that modifications to one module of a COBOL program do not affect the proper execution of the entire software system.

This dissertation addresses a number of important software maintenance problems which fall within the framework of software management and software engineering. Compiling techniques are used to analyse the COBOL program, from a view point of both reformatting the COBOL program PROCEDURE DIVISION and deriving a complexity measure of a given COBOL program, to assist in assessing the ease of software modification made during the software maintenance phase.

The purpose of the project of which this thesis is a part of build up a database which will completely specify a COBOL program and allow it to be analysed in various ways related to maintenance, enhancement and other tasks.

The used of MUCAS in a COBOL production project could also significantly ease the COBOL software maintenance process. Figure 1.1 shows the position of MUCAS in a modern COBOL production environment. Upon receiving the system change specification (come originally from user) from the project personnel (may be manager, project leader or systems analyst), the systems analyst or programmer will analyse the existing applications programs and design the program change specification. MUCAS will be used to analyse the existing programs which are to be modified in order to help determine both their existing

structure (and its quality) and also the extent of the structural changes necessary to implement the required modifications. This is followed by the programmer code creation according to the program change specification. While testing and verify output results using the test data and live data, any error will be feedback to redesign, retest or re-confirm user request accordingly until the program changes are correct and the modified program performs it specific task [BEI83]. At this stage, MUCAS will be used again to analyse it, generate the program analysis report and store the program information in the program database. If the program analysis report was satisfactory, the program changes are considered as successfully done; otherwise, the re-analysis of the existing program procedure will be repeated.

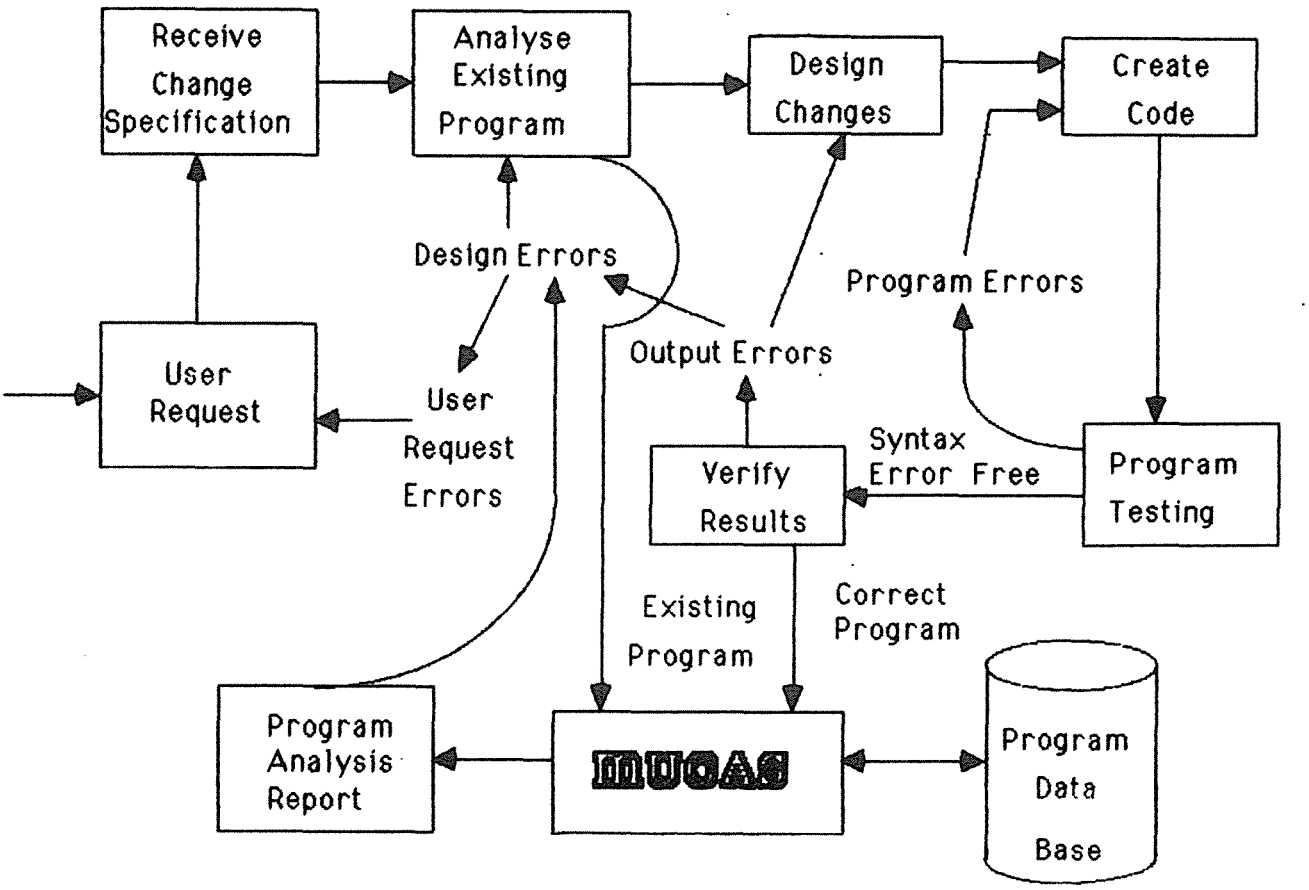


Figure 1.1 The COBOL Program Processing Activities
In a MUCAS Aided Production Project.

The writing of MUCAS mainly involved the computing techniques : compiler design techniques ([WEI73], [TRA63], [SEV74], [PRI71], [MOR68], [LUM71], [GLA69], [SIM73a], [COB68], [COB74], [COB81], [COR81], [HOP69], [MCK74], [AHO77], [GRI71] and [GRO74]), structured programming ([DAH72], [DOR72a], [DOR72b], [BAK72], [MCC78], [BAK74], [DIJ76], [ELS76], [LIN79] and [TRI79]), structured design techniques ([STE74], [WIR74], [RIC80], [OGD72], [NIC75], [MYE76] and [MYE78]), data base techniques ([OLL78], [OKO85], [CAR79a], [CHA77], [DAT81], [FER83] and [VET81]), software systems design ([BRO82], [AMS76], [ART84], [HAL77], [KER79], [KRU84], [LAN79], [LEA83], [MET81], [CAR81], [MCG80], [BAB85] and [JAC70]), program computational complexity ([OTH81], [SCH81], [WOO79], [ZOL80], [MCC76a], [BAK80], [CHA79], [CHE78], [GON84], [BAC57] and [MCT80]) and computer program testing ([BAS80], [BEI83], [BOY81], [GLA81], [GOU81] and [MUC81]).

Development of Structured Programming (SP) has evolved in conjunction with the top-down concept [MCC78]. MUCAS is designed to analyse the composition of COBOL programs so that it can not only show the structure of unstructured COBOL programs but can also test how good the structure is. **Testing** of MUCAS is bottom-up while **design** is top-down stepwise refinement [MCC78].

Each procedure was tested individually until it was error free and performed the correct functions, then it was added to the MUCAS system. Various software methods such as SP, structured design, top-down stepwise refinement design [MCC78] and bottom-up program testing were used throughout the MUCAS system development process [BEL76] [BAS75]. The finite-state machine concept [AHO77] was used to construct a COBOL lexical scanner. The COBOL programming syntax was parsed by the top-down recursive-descent method ([GRI71], [MCG80], [MCK74] and [AHO77]). The COBOL program was reformatted according to a coding standard to show the structure levels of the COBOL program.

Generally, the purpose of the MASSEY UNIVERSITY COBOL PROGRAM ANALYSER SYSTEM (MUCAS) AND WORKBENCH PHASE I is

(i) To process the text of an existing COBOL program (which has previously been compiled and run, and therefore has no syntax errors) and identify all major structural components and relationships.

(ii) To format the program in a standard manner (A coding standard is suggested in this thesis).

(iii) To output tables and files of structural information for later use in the construction of the program specification database.

Many complexity measure methods are mentioned in this thesis. A combination complexity measure method and guidelines are suggested. A COBOL programming syntax complexity factor is proposed. In general, this type of program complexity measure could be applied to other programming languages.

The design goals of the MUCAS were :

(1) **Research**
To investigate the extent to which structural information relevant to the maintenance function can be derived automatically from existing COBOL programs.

(2) **Development**
To enable application maintenance programmers to check program logic and complexity of a given program.

(3) **Economics**
To reduce software maintenance effort and cost.

This thesis consists of six chapters. The first chapter is the introduction. Chapter 2 summarises the relevant concepts and methods. Chapter 3 reviews the literature and describes some of the history and current status of software trends in business data processing. Chapter 4 deals with design rules and algorithms for reformatting COBOL's PROCEDURE DIVISION and covers the implementation of MUCAS. Chapter 5 discusses future developments of MUCAS Phase I. Chapter 6 is the discussion and conclusion.

CHAPTER 2

COMPUTER PROGRAM FLOW ANALYSIS AND APPLICATIONS

CHAPTER 2 COMPUTER PROGRAM FLOW ANALYSIS AND APPLICATIONS

2.1 DESCRIPTION OF FLOW ANALYSIS

Hecht [HEC77] mentioned that the pre-execution analysis of computer programs is usually referred to as **control flow analysis** and **data flow analysis. Flow** analysis **(both control and data) is a fundamental prerequisite for many** important types of code improvement. The information obtained from flow analysis can be used to drive the optimisation phase of a compiler, to generate invariant assertions for program verification, to improve software reliability by aiding the generation of program documentation and automation of a large part of the debugging process, and to direct the actions of a goal-oriented program documentation system [MUC81]. In general, control flow analysis precedes data flow analysis [ACK82]. Control flow analysis is the encoding of possible program control flow structures, or flow of control of the computer program, included usually in the form of one or more **flow graphs.** Flow graphs are usually represented by the mathematical system of **Directed Graphs** (Digraphs). The definition and applications of digraphs are explained in Harary [HAR65] Chapter 1, [AND77] Chapter 3 and [MCK82] Chapter 5. For further reading on this, refer to [BOL82], [WIL79], [BOF82], [ENR72] and [CAR79b].

Data flow analysis [ACK82] is the process of ascertaining and collecting information prior to program execution about the possible modification, preservation, and use of certain entities (such as values or various attributes of variables) in a computer program. In later phases of the project, we are concerned with both **control flow analysis** and **data flow analysis.**

2.2 **DISCUSSION OF CONTROL FLOW ANALYSIS**

Since 1970, computer scientists have made important contributions to **graph theory** by developing the notion of computational complexity [CAR79b]. They have also achieved remarkable improvements in performance of graph algorithms. Thus, in the 1970s, computer scientists began to apply methods (such as SP methodology, graph algorithms [STI74]) to organise the classification, comparison, and application of various programming languages [COL83]. Landin [LAN64], Strachey [STR65] and Wegner [WEG68] provided insight into operational models of computation, which led to a unified view of computation structures. At the individual program level, the program constructs, program semantics, and the expressive power resulting from modularity, can now be studied in terms of data structures and access paths. One can make use of the **static analysis** method to derive the control statements of the language [ORG78].

At the statement level, statements that invoke procedure entry, function entry, coroutine resumption, and block entry are intended to produce a shift of context which redefines the data structures accessible to the processor. On the other hand, procedure return, function return, and block exit are steps specifying the restoration of an earlier context. The declaration of a variable, a procedure, or a new data type is essentially a rule to allocate or restructure the workspace.

Storage allocation processes cause the establishment of access paths and these paths may result in the sharing of data. Organick [ORG78] explained the structures of programming languages in terms of data structures and accessibility concepts. The **contour model** proposed by Johnston [JOH71] and others like Berry [BER71] and Wegner [WEG71] explain the semantics of a variety of programming languages in terms of data structures and program structures.

Increasing importance is being attached to the idea of measuring software characteristics [BOE76a] [MCC76a]. Boehm [BOE76a] discusses whether new programming techniques are having the effect of reducing the problems of reliable software production. Unfortunately, many of the qualities such as ease of testing and maintenance etc., are highly subjective [AMS76]. Therefore experiments will need to be carried out in order to determine more reliable measurements of program structural characteristics. Such a process will be very dependent on the language used and the in-house programming standards.

Two major factors influence the reliability of software which has limited testing : The **complexity** and **unstructuredness** of the program involved. Intuitively, we would expect that more complex programs will have a greater incidence of errors ([WOO79], [BAK80], [MCC78] and [FEU79]). It will also be desirable to be able to compare the use or lack of use of the principles of SP used within a program and the incidence of errors produced [ELS77]. These facets of a program need to be measured objectively and compared with alternative programs in order to select a good program structure method [WOO79].

2.3 DIFFICULTIES AND POSSIBLE SOLUTIONS IN BUSINESS DATA PROCESSING

COBOL (**CO**mmon **B**usiness **O**riented **L**anguage) is a programming language developed specifically for the programming requirements of business applications. It is the most popular business programming language in the world and will probably retain its popularity in the foreseeable future [LIM80] [MCC78]. ANS (American National Standard) COBOL is an expanded and standardised version of the original COBOL.

According to Baker [BAK74], in a programming system, the organisation, procedures and tools associated with the implementation of SP are critical to its success. This is particularly true in production programming environments [ROB76] [BAK75] where program systems (rather

than a single program) are developed - people come and go - and the attainment of reliable, maintainable software on time and within cost estimates is a prime management objective. In this environment, module level coding and debugging activities typically account for about 20% of the effort spent on software development.

Wolberg [WOL83], Aho [AHO77] and Lientz [LIE80] emphasised that a structured program is generally easier to read, write, and test for correctness than an unstructured program. A structured program should provide features of modularity, and the control flow should be expressed in an easily understandable manner. The advantages of a structured program are as follows [BAK74]:

1. It is easier to read. A listing of a program in structured form can be read and understood quite quickly. This is often impossible if the program is written using older coding methods.
2. It is easier to maintain and to extend. A programmer can learn structured programming in about a week of coursework and practice; and once learnt, the programmer generally will not revert to former methods.

SP originated in late 1960s [DAH72] and was quickly utilised by one of the most important software organisations, IBM Federal Systems Division (FSD), at that time [BAK74]. The IBM FSD is an organisation involved in production programming on a large scale. FSD experienced problems of software maintenance in pure applications programming using commercially available operating systems and program products. However, it still took some time before the use became widespread in the business data processing sector.

SP refers primarily to coding standards governing control flow, and module organisation and construction. It requires three basic control flow construct (SEQUENCE, IFTHENELSE and DOUNTIL). These three constructs refer to general standards for SP, as well as to detailed standards for use of control flow constructs in various programming languages. More detailed descriptions of control flow analysis are given in [HEC77], [ELS76], [CHA79], [MCT80], [SCH81], [WOO79] etc.

SP provides enhanced readability of code, enforces modularity and thus encourages changeability and maintainability, simplifies testing, and permits improved manageability and accountability. These are all well-known properties of structured programming ([BAK72], [BAK74], [DIJ76], [DOR72a], [LIN79], [MCC78] and [ELG76]). Historically, programming has been a very individualistic, undisciplined activity [MOR85]. Thus, the introduction of discipline in practice yielded double benefits, i.e. the advantages inherent in the methodology itself, plus those due to better standardisation and control. Other advantages of structured coding are described in Richardson [RIC80] and Lim [LIM80].

We are now some twelve years into the **structured programming** revolution, and have gained some insight into its strengths and limitations. Although development of programs with well-defined structures has forged ahead, the maintenance enhancement of these has lagged well behind especially as programs have increased in complexity.

2.4 PROGRAM COMPLEXITY MEASURES

When one steps back and views the entire process of program development from beginning to end, one is surprised at its present complexity. Initially (Pre 1950s), the input to the program development process was a small, and often incomplete statement of requirements. This was coded as a string of many thousands of zero and one bits which instructed a set of electronic devices how to perform as a system.

The use of assemblers and assembly languages in the 1950s removed the need to write say the bit string 10101010 to compare the value of two registers. Later development of higher-level languages and their compilers (1960s) allowed the programmer to work without knowledge of the details and idiosyncracies of the machine. In the 1970s, people began to recognise that programmers needed better methods and tools to manage complexity; therefore, such ideas as structured programming, code reading, and program development libraries arose. By this time, people were recognising that mastering complexity was the key to successful software development.

Although these contributions have been valuable, they still leave much to be desired. For example, the world's most troublesome program is perhaps the **operating system**. In other words, there is much more to the subject of complexity than simply attempting to minimise the local complexity of each part of a program. A much more important type of complexity is **global complexity** : i.e. the complexity of the overall structure of a program or system (This is related to the degree of association or interdependence among the major pieces of a system).

Tools are needed in order to permit effective implementation of, and to achieve maximum benefits from, the ideas of SP and Program Complexity (PC). MUCAS (Massey University Cobol Analyser System) is aiming towards this direction.

2.5 A TOOL FOR OBTAINING SOME COBOL PROGRAM COMPLEXITY MEASURE (MUCAS)

The progress in computer information technologies and advancement in solid state electronics has encouraged the development of a large number of hardware systems resulting in a correspondingly large drop in costs : 10 times in 10 years [BJ082] Part 1 pp 6-17. However, despite a lot of improvements in the programming process, the cost of producing software has not decreased significantly over the last 20 years due to increasing labour costs [MYE78] Chapter 1. The tendency is for software to comprise an increasing proportion of the total cost. For example, the larger the computer system is, the greater is its associated software cost.

The on-going developments of COBOL since 1959 have been COBOL-60 [CUS60], COBOL-61 and COBOL-61 Extended, COBOL-1965, COBOL-1968, COBOL-1969, COBOL-1970, COBOL-1973, COBOL-1974 [ROB76], COBOL-1978 and COBOL-1980 [GAR84]. MUCAS is designed to support SP in COBOL [COB81] language. The programming standards required cover both the implementation of the control flow constructs in COBOL as well as conventions for formatting and indenting COBOL programs.

The incompatibilities of the previous mentioned COBOL versions range from the serious to the trivial; but even the trivial may cause a program to **crash** - either at compile-time or run-time - in a 'new standard' environment. Further problems may arise depending on the operating system used since some operating systems are more compatible with some COBOL versions than others. MUCAS is a COBOL preprocessor which parse a COBOL program and analyse its program complexity, hence it must be able to cope with various versions of COBOL.

2.5.1 **BASIC CONTROL FLOW STRUCTURE FOR COBOL**

There are four approaches which can be taken to provide the basic and optional control flow constructs in the COBOL programming language, and each was evaluated in certain situations in MUCAS.

1. The control flow construct may be directly available as a statement in the language. In COBOL, the IFTHENELSE (with slight restrictions) and the DOUNTIL (as a PERFORM UNTIL) were present.
2. The construct may be easily simulated using a few standard statements.
3. A standard preprocessor may be used to augment the basic language statements to provide necessary features.
4. A special preprocessor may be written to compile augmented language statements into standard ones, which may then be processed by the normal compiler.

SYNTAX CONTROL STRUCTURE FOR COBOL

(1) Sequence structures
    Sequence
(2) Alternation structures
    IFTHENELSE
    IFTHEN
(3) Iteration structures
    PERFORM-TIMES
    PERFORM-VARYING
    PERFORM-UNTIL
(4) Transfer structures
    GO TO
    GO TO DEPENDING ON

Some uses of GO TO DEPENDING ON, however, have similarities to CASE statements in other languages. These could thus be classified under (2) above. Certainly coding conventions employing GO TO DEPENDING ON as a limited case facility can be used.

The result of using these four approaches was a complete set of control flow constructs for COBOL in MUCAS design. To assist in making programs readable and in standardising communication between programmers, it is desirable that programs in a given language should be organised, formatted and indented in a standard way [KER79]. Coding conventions were developed for each of the permitted control structures, segment formats, namings, use of comments, labels, and indentation and formatting for all control flow and special (e.g., OPEN, CLOSE etc) statements. A basic usage of MUCAS is to produce COBOL programs which are consistent in format and are readable, and maintainable (see formatting guidelines in Chapter 5 of this thesis). A set of program development standards was developed in order to achieve this objective (Chapter 4).

2.5.2 MUCAS DESIGN TECHNIQUES

The writing of MUCAS mainly involved the computing techniques : compiler design techniques, structured programming, structured design techniques, data base techniques, software design, program computational complexity and computer program testing.

Development of SP has evolved in conjunction with the top-down stepwise refinement concept. MUCAS must be designed in such a manner as to follow the composition of COBOL programs so that it can not only reformat unstructured COBOL programs but can also test how good the structure is. Each program module or procedure was tested individually until it was error free and performed the correct functionality, then it was added to the MUCAS system.

2.5.3 MUCAS FORMATTING STANDARDS

The development of software systems has been recognised as involving a management problem [POD85] [GLA82]. The increasing emphasis on software cost and long-term maintenance of programs means that standards are being developed which go beyond those required for SP. These standards are the foundation for controlling entire software developmental efforts [GLA82].

There is much value in utilising a good set of formatting standards [KER79]. First, management as well as other programmers may more easily read and understand the structure of programs written by others. Second, development of formatting techniques reduces the textual complexity of programs. Finally, standards provide management with guidelines to control the overall programming effort.

Sample program standards are presented (Appendix E) which are regarded as helpful for the development of quality programs in this thesis. The primary objectives of these standards are to enhance the readability of code and to facilitate maintenance of programs. To promote high-quality programs, these standards can be expanded to encompass features unique to different languages. The standards given in this thesis are only broad guidelines for assisting managers and programmers in developing usable standards.

2.6 **APPLICATION AREA OF MUCAS**

MUCAS was implemented partly to ease COBOL program reading. Linger [LIN77] mentioned that a structured program of any size can be read and understood in a completely systematic manner, by reading and understanding its hierarchy of programs. The ability to read programs methodically and accurately is a crucial skill in programming. Programmers can become more effective through critical study of programs written by others. This allows one to modify and validate programs written by others, to verify the correctness of one's own programs, and to select and adapt program designs from the literature. There are two reasons for reading a program :

1. Verification that a program is correct with respect to a given function. This constitutes a design review for its correctness.
2. The determination of the function of a program. This involves design discovery so that modifications can be done as simple as possible.

2.6.1 **REFORMATTING COBOL PROGRAMS**

SP is almost exclusively concerned with the pragmatics of programming languages. One of the important goals of SP is to write programs which are correct and thus do not need debugging in the usual sense [GLA82]. The observations of Dahl [DAH72] "structured programming" are that some language features (such as the unrestricted GO TO statement) are harmful to the goal of correct programming [ELG76] [DIJ68], primarily because they reduce the readability of programs. Thus, many of the initial efforts of SP have been directed toward improving languages to make them more readable and easy to use [BAK74] [LIN77]. MUCAS did not provide the COBOL program restructuring [OTH81] facility, instead it provide the COBOL programs reformatting and hence improve existing COBOL programs readability and clarity. Correspondingly, MUCAS also

show the complexity measure of the reformatted COBOL program.

2.6.2 **COBOL PROGRAM COMPLEXITY MEASURES**

The program complexity measurement results of MUCAS may reveal valuable program information, in addition to reformatting COBOL programs. It can give significant quantitative information about the complexity of the program in each run. The program complexity measures which shown by MUCAS run was McCabe's cyclomatic approach complexity ([MCC76a], [HAN78] and [ZOL80]) and Halstead's software science effort ([BOH75], [HAL77] and [ZOL80]). The results of the program complexity measurement should encourage improvements in programming maintainability where MUCAS has been used. Others proposed program complexity measures method which could incorporate in MUCAS may refer to Jean Zolnowski and Dick B. Simmons [ZOL80].

2.6.3 **COBOL PROGRAM MAINTENANCE**

While some new languages well-suited to the aims of SP have been proposed, the fact that millions of programs already exist in current languages and thousands of programmers are already expert in their use makes it doubtful that the languages will be adopted [LIM80]. Instead, the current trend is toward enhancing existing languages such as FORTRAN [BAK80] [MCC76a] and COBOL [ZOL80], with new features or emphasising the use of certain languages such as PL/I [ELS76] [ELS77] (PL/I is however declining in use). The features that are currently being emphasised are "well-behaved" control structures (that is, ones which do not involve unrestricted GOTOs [ELG76] and which make readily apparent the meanings of statements) and block structure which permits a number of statements to be grouped together so that the programmer can treat them as a single unit [MCC76a]. Other style features include reformatting of statements on the program listing [KER79]. At this point, a number of different constructs and approaches are being tried, but most of them are useful and none has gained dominance.

CHAPTER 3

REVIEW OF

THE LITERATURE

CHAPTER 3 REVIEW OF THE LITERATURE

3.1 SURVEY OF THE SOFTWARE LIFE CYCLE

Based on the description [LEA83], the **software life cycle** is best viewed as a succession of steps, that is **requirements statement, software design, coding, testing** and **maintenance.** Generally, the software cost greatly affects the success of a computing project of the computer system [BOE81].

The result of a survey by Zelkowitz (1978) [ZEL78] indicates that in many large-scale software systems, the **maintenance phase** typically accounts for 70% of the total cost. The development cost is summarised as follows :

Software Development Phase	Cost
1. Requirements	3%
2. Specifications	3%
3. Design	5%
4. Code	7%
5. Module Test	8%
6. Integration Test	7%
7. Maintenance	67%

Therefore, the **maintenance phase** is the main area which should be considered in attempting to reduce costs of software development.

3.2 THE CURRENT USAGE OF PROGRAMMING LANGUAGES

According to the survey of programming language usage in 1977 [PHI77], 63% of 132 sites used COBOL. Whereas, in [BRO75] Chapter VIII.B, a 1975 survey of about 100 data processing centers by the Software House CAP-SOGETI indicated that 56% of the data processing centers used COBOL as their programming language. Several surveys of language usage have been published by [WOL83] and are summarised in Figure 3.1.

LANGUAGE	Percent Usage
COBOL	**38**
RPG	20
FORTRAN	12
Assembler	12
BASIC	7
PL/I	2
APL	1
Other	8

Figure 3.1 Datapro Survey : Percentage Usage of Languages in 5813 Data Processing Centers [GEP77].

The results of all these studies confirm that COBOL is by far the most popular programming language. Undoubtedly, this trend will continue for some time to come.

3.3 TRENDS IN PROGRAMMING LANGUAGES

Languages are ubiquitous in human affairs and are no less universal in software systems. To use most computer languages, we must master a great deal of detail because of their role as a vehicle for communication between people and systems [BRA77] [FRE75]. Reviews of the history of programming languages by [SAM69], and [ACM78] place emphasis on the technical factors that have influenced the development of programming languages. These indicate that the basic trend of languages tended more and more toward simplification from the user's point of view. As a result of this trend, new applications can be developed in a much more cost effective manner than was possible a decade ago.

3.4 THE PROBLEMS OF SOFTWARE RELIABILITY

Deutsch [DEU82] mentions a major difficulty of business data processing, that is the control over the quality of the software, especially in proving the **correctness** of programs. Software reliability guidelines provided by [GLA81] show that when a program's overall structure and related procedures or sub-procedures do not completely correspond to the program specification, then unexpected results can occur. The major techniques and considerations associated with adequate control have a cost associated with them, for example (apart from maintenance) **program testing constitutes the major cost of the software life cycle ([BEL71], [BOE81],** [CAV78], [DEU82], [FOX82] and [LIE80]). Detection of software errors at the later stage of software life cycle normally incurs greater expense than earlier detection [GLA82] [FOX82]. In general, adequate controls on the software system are very important in terms of reliability, quality and cost. Most of the effective techniques are seldom sufficient on their own, and different methods detect different errors.

3.5 **IMPLICATIONS FOR MASSEY UNIVERSITY COBOL ANALYSER SYSTEM**

Before considering the developments leading to the proliferation of languages, it is useful to analyse the current situation. From the survey of the **software life cycle** in Section 3.1, it was implied that the **maintenance phase** was of primary importance for software development. The current usage of programming languages (Section 3.2) shows us that COBOL is the most important programming language which we should select for maintenance purposes. Section 3.1 and Section 3.2 indicated that there is a worldwide software maintenance crisis. The trend of programming languages (Section 3.3) and the problems of software reliability (Section 3.4) suggest that the software that we implement should be simple, as well as being able to give information about the **reliability** and **quality** of the COBOL source program. These factors would also increase the ease with which other programming professionals could use, modify and maintain programs. This would reduce the software maintenance cost [HER85]. Thus the objective of this project is to develop a software system MUCAS (**M**assey **U**niversity **C**obol **A**nalyser **S**ystem), which can be of assistance in solving some of the management problems of the previously mentioned worldwide software maintenance crisis, by determining the structure and **complexity characteristics** of a given source COBOL program. Those programs which fail to meet satisfactory criteria could be modified accordingly.

In this thesis, the Design Method of MUCAS (Chapter 4) describes the methods and techniques that were used to design and implement the MUCAS system, and evaluate the PC problem. The approach was to reformat the PROCEDURE DIVISION of a given source COBOL program and convert it to a standard structured form. It should also generate the source program **control flow** information (MUCAS Phase II), since the software maintenance of a COBOL program requires the understanding of the **control flow** of the given program, and the PC itself.

At this stage (MUCAS Phase I), we will not be concerned with the data declaration portion (DATA DIVISION) of the COBOL program. Most of the work done will be concentrated at the executable portion of a COBOL program, which is the PROCEDURE DIVISION.

CHAPTER 4

THE DESIGN METHOD OF MUCAS

CHAPTER 4 THE DESIGN METHOD OF MUCAS

4.1 FUNCTIONAL DECOMPOSITION OF MUCAS

The MUCAS functional decomposition design and implementation are separated into four stages as follows :

1. Lexical Analysis stages (MUCAS phase I).
   Input : character stream.
   Output : lexical errors, symbol stream.

2. Syntax Analysis stages (MUCAS phase I).
   Input : symbol stream.
   Output : syntax errors, program data attributes information.

3. Formatting Analysis stages (MUCAS phase I).
   Input : refined symbol stream.
   Output : format errors, reformatted program.

4. COBOL Program Analysis stages (MUCAS phase II).
   Input : program data attributes information.
   Output : program complexity measure, control flow graph, data base information. (Note that the latter two are not covered in this thesis and that only part of the complexity analysis is covered).

Figure 4.1 MUCAS Implementation Procedure.

Therefore, the MUCAS project was started from the design and implementation a COBOL lexical scanner, COBOL parser, COBOL formatter and then the COBOL program analyser. The stages of the MUCAS project were implemented in the previously stated order (COBOL lexical scanner, COBOL parser, COBOL formatter and then the COBOL program analyser) while the design was in the reverse order.

4.2 THE METHOD OF IMPLEMENTING MUCAS

In Chapter 2 of this thesis, it was mentioned that a current software engineering trend was an increasing interest in **measuring software complexity.** This measures many aspects, including control flow constructs and the structuredness of a program, so that the software complexity can be minimised. In Chapter 3, it was pointed out that COBOL was the most important business programming language that we could assess for **complexity.** This was the reason for developing MUCAS. The main methods for implementing MUCAS are **top-down stepwise refinement** design and **bottom-up** testing. The functions of individual procedures are defined prior to the subordinate procedures. In order to minimise unnecessary errors which occur in the expensive subsystem or system testing, individual subprocedures are tested until they are both error free and can perform the correct function before addition to the system. Figure 4.1 shows the order of MUCAS implementation procedures.

In MUCAS design, the implementation approach was first defined, followed by defining MUCAS's ANS COBOL syntax [COB81] in Appendix B, and then implementing it on a computer. Since at Massey University only the PRIME 750 system had the necessary facilities, this machine was used to implement MUCAS. There were two basic objectives of the MUCAS implementation :

(a) **Transferability** : Implementation was to be as machine independent as possible. It is advantageous to write a translator to run on different machine [MCK74].

(b) **Ease of use and debugging** : The incorporation of satisfactory error detection and error diagnostics.

Implementation on the PRIME 750 for the ideal of machine independence has its limitations. For example, character sets normally vary between different machines. Thus, some features of the PRIME 750 have influenced the implementation of MUCAS.

MUCAS has good diagnostic facilities at compilation time. Two listings of the COBOL source program are generated during MUCAS compilation. The first corresponds exactly to the submitted program, except that line numbers have been prefixed to each line in the source program. Error messages will be printed whenever any error is detected. A symbol trace is provided if the proper option is selected. The second listing is a reformatted COBOL program generated by the MUCAS run.

The **Parser** itself is top-down recursive-descent [MCG80] [AHO77] and uses tram-line diagrams with links to the associated language production (according to the BNF COBOL programming syntax in Appendix B). However, a top-down method for parsing has the difficulty of precisely locating the occurrence of an error [SIM73a]. Errors are more precisely pin-pointed using a bottom-up method, so it is advisable to introduce the bottom-up method to have good error detection later.

4.3 TRENDS IN SYSTEMS DESIGN

4.3.1 THE PROGRAMMING PROCESS

Cushing [CUS60] said that the programming process has three parts. First, systems definition (systems requirements [DAV80]) deciding the field of application. Secondly, systems design [LEA83] [BRA84] deciding the work flow to the computer and the solution of the problem. Thirdly, coding [SHE79] [KER79] - deciding the detailed programming steps which the computer is to use in solving the problem. Ledgard [LED75] mentioned that students and professionals tend to be over-optimistic about their ability to write programs or to make programs work according to pre-established design goals. While in programming it is not enough to be inventive and ingenious, one also needs to be disciplined and controlled in order not to become entangled in one's own complexities [MET81]. The new generation of programmers will need to acquire discipline and control, mainly by learning to write programs correctly from the start, and to master programming complexity [WOO79]. The debugging process should take the new form of verifying that no errors are present, rather than the old form of finding and fixing errors over and over again [RIC80]. It is the responsibility of the programmer to program in a systematic way so that his or her work can be shared and utilised by others.

4.3.2 FUNCTIONAL DEFINITION AND INTERFACES

Lano [LAN79] specified that the significant problems facing system designers and master planners today, are the timely and accurate definition of system element interfaces and tasks or activity interrelationships. The classical approach to requirements and/or activity definition and allocation consists of the subdivision of large functions into multiple, understandable, discipline-oriented modules or blocks [STE74]. Each of the procedures defines its inputs, outputs and

transfer functions. The emphasis is usually placed on transfer functions, while the inputs and outputs (Interfaces) are often ignored [LAN79].

Naturally, transfer function definitions are more readily understood and communicated than interface definitions since these function are quantifiable, structured, of similar design to that of the implementation, and can be easily related to past experience.

A structured technique must therefore be acquired by the effective system designer and master planner, which places a balanced weighting on all of the definition and allocation parameters in order to keep the program size smaller, less complicated and reasonably efficient.

Lano [LAN79] suggested that a more equal weighting should be placed on both the functional definition and the interface parameters transfer. A good balance between these two factors for **program design** will give performance efficient software. This means that the use of fewer functions and hence "clearer" functional definition will have less interface transfer, more global variables between modules, longer coding and high complexity. Conversely, too much emphasis on SP will have more interfaces, shorter coding, low complexity and low clarity of functional definition.

Interfaces and activity relationships must be dealt with in a top-down, hierarchical manner, which is as structured and timely as the current transfer function definition process [LEA83]. The implementation of an efficient interface definition methodology has become essential due to the increasing relationship complexity of current tasks, organisations and systems.

4.3.3 TOP-DOWN DESIGN AND BOTTOM-UP TESTING

Metzger [MET81] stated that the traditional program design has been analysed and designed top-down, and coded and tested from the bottom up. During analysis and design it seemed natural to start by first considering the system as a whole and then breaking it down into smaller and smaller pieces which individuals could handle. These pieces were coded and tested, and combined into increasingly larger and more complex groupings until finally the entire system had been assembled from the bottom up. Many systems are still being constructed this way. The current trend is toward complete top-down development, wherein not only analysis and design are attacked from the top, but so are coding and integration testing.

The **top-down, stepwise refinement** proposals by Dijkstra [DIJ72a] were intended to address the decomposition problem, i.e. the separation of system functions into subfunctions. In the stepwise refinement process, one attempts to organise program components by identifying the subordinate parts in such a way that their function is kept simple and is easily verified. These subordinate parts will in turn have parts that are subjected to the same methodology. Ultimately, the program is described in terms of the "primitive" operations of a language or virtual machine. No further refinement is possible on these primitive components. This process is an extension of the functional decomposition process [MYE78]. Since complete top-down testing could be very costly and difficult to implement, bottom-up testing was adopted as one of the methods of MUCAS design. Individual low level modules are tested first, then entered into the software system for more costly integrated sub-system or system testing. **Stepwise refinement** was used as another method in MUCAS design. SP techniques are applied throughout the Lexical Analysis stage, Syntax Analysis stage, Formatting Analysis stage and the Program Analysis stage of the MUCAS I design.

4.4 THE MUCAS DESIGN

The MUCAS project began with some of the proposed requirements specified in Chapter 3 of this thesis. At the analysis phase, a precise, structured problem description document was prepared as the baseline for subsequent design and programming, and a detailed plan was written to guide all the remaining phase of the project.

The next phase was designing a software system to best match the problem defined. The overall software system design was completed first, then a baseline for detailed design and coding was established, and turned over to the implementation phase. In the implementation phase, the baseline design was further refined into smaller modules until the refinements reach the level of actual code. The modules were coded, tested and carefully merged with one another in a previously planned manner.

As modules were added successfully, the software system grew in complexity and usefulness. It was constructed to a number of high levels where it could perform some subset of intended functions. The system fitted together well, because design integrity was sought and achieved in the earlier phases, yet analysis, design or coding mistakes or changes occurred. Changes were made, but they were strictly controlled through a simple mechanism originally planned for.

System tests were carried out to assure integrity and objectivity. Where these failed, changes were made to correct the problems found until a cleanly compiled system, complete with clean documentation, was attained, and the software system met those criteria earlier proposed.

The MUCAS design development processes are : define the function of each module, express the design in a structure diagram, consider how to implement each module, determine the position of this module (i.e. the higher level and lower level related modules). The objectives of this design are : to insure that program specification requirements are met, to improve understanding of the design, to identify design errors or inconsistencies, and to make possible an efficient, modifiable implementation. The MUCAS design process proceeded back and forth among design alternatives until a satisfactory design was developed.

The MUCAS design features involved compiler design methods ([AHO77], [GLA69], [GRI71] and [FRE75]). For further reading refer to [MOR68], [POL81] and [PRI71].

4.4.1 AN OVERVIEW OF THE STRUCTURE OF MUCAS

The task of MUCAS phase I was to extract basic information from the input COBOL source program and to convert it into an equivalent reformatted COBOL program. This applied mainly to the PROCEDURE DIVISION. MUCAS phase I also produced an output listing of the source input program and generated a set of output database entry data for MUCAS phase II. A symbol trace is provided if the appropriate run option is chosen. It also prints out a report of the Symbol Table (ST) data attributes and a brief complexity measure report based on the generated reformatted COBOL program. Basically, MUCAS I consisted of the following modules :

OVERVIEW MUCAS I DESIGN

1. HOUSEKEEPING.
2. FIRST PASS.
3. SECOND PASS.
4. TERMINATION.

The HOUSEKEEPING module catered for user enquiry, and opening input and output files which are used by the system. It loads the standard entries in the Symbol Table (ST) from an input file. In the FIRST PASS of MUCAS, the input COBOL source file is scanned line by line, until the PROCEDURE DIVISION line is reached; this line number is noted. The PROCEDURE DIVISION is then scanned. A search update to the ST is performed whenever a section name or paragraph name is encountered. This information is used in the later SECOND PASS procedure. The SECOND PASS (analyser pass) procedure of MUCAS first does the housekeeping for the COBOL formatter (FORMATTER PROGRAM procedure), then scans the input COBOL source file line by line, and outputs the source line to the COBOL Formatted File (CFF) until the PROCEDURE DIVISION statement was reached. It determines this by using the line number of PROCEDURE DIVISION statement which was obtained in the FIRST PASS. The **parsing, formatting** and **program complexity analysis** of the PROCEDURE DIVISION then followed, in the order of section by section, paragraph by paragraph, sentence by sentence, statement by statement. It produces a formatted COBOL program, plus the corresponding information about this formatted COBOL program. It also performs a search update to the ST whenever a symbol is encountered.

The TERMINATION procedure closes all files and prints out the information and a PC report about the created reformatted COBOL program. It also prints a termination message to the user terminal.

In Figure 4.7, the functional design structure of MUCAS is shown, and the overview design structure of MUCAS is shown in Figure 4.2. Each of the procedures in Figure 4.2 will further subdivide into several different sub-procedures.

MUCAS System Overview Structure

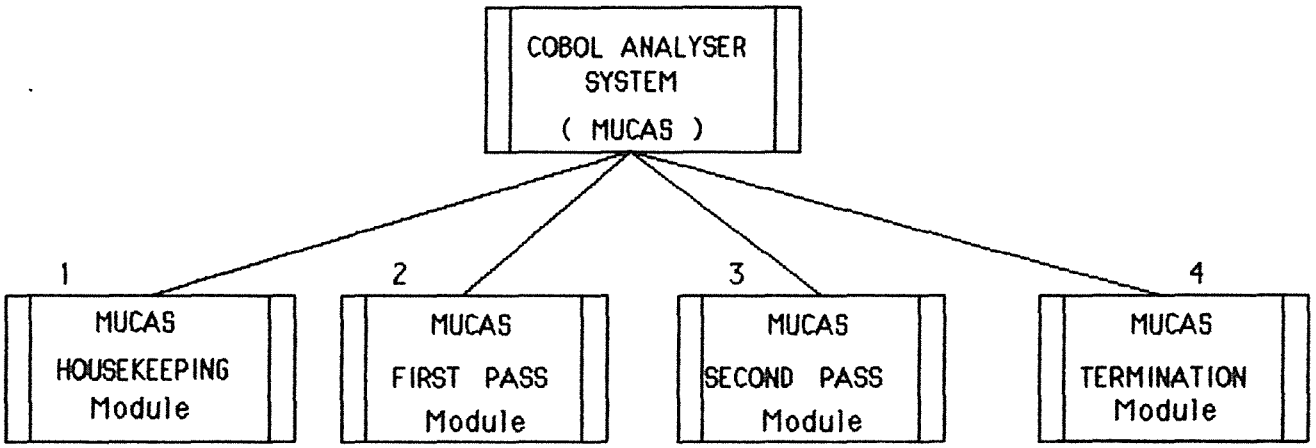


Figure 4.2 MUCAS System Overview Structure Diagram.

1 MUCAS HOUSEKEEPING Module

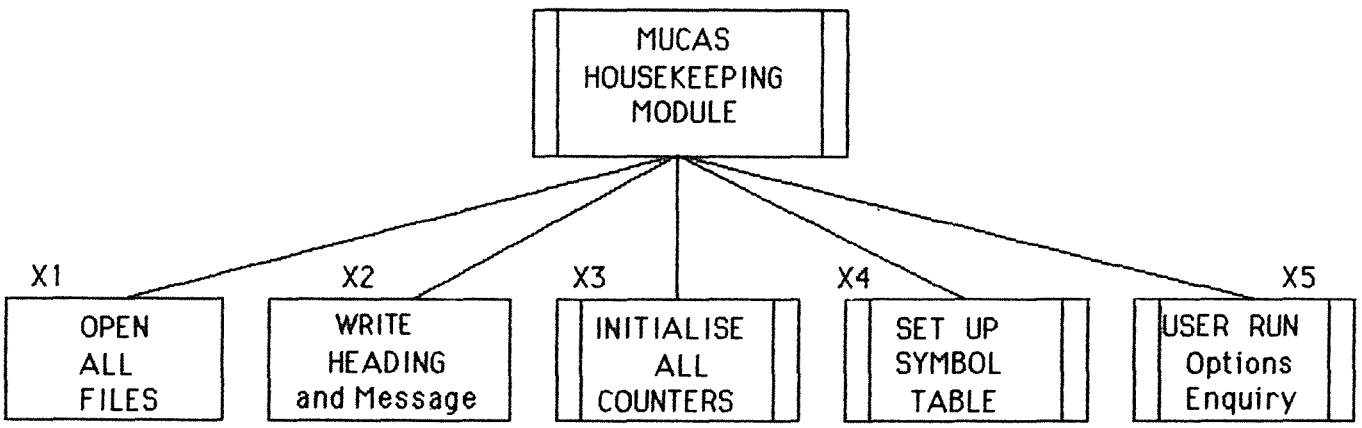


Figure 4.3 MUCAS HOUSEKEEPING Module Structure Diagram.

2 MUCAS FIRST PASS Module

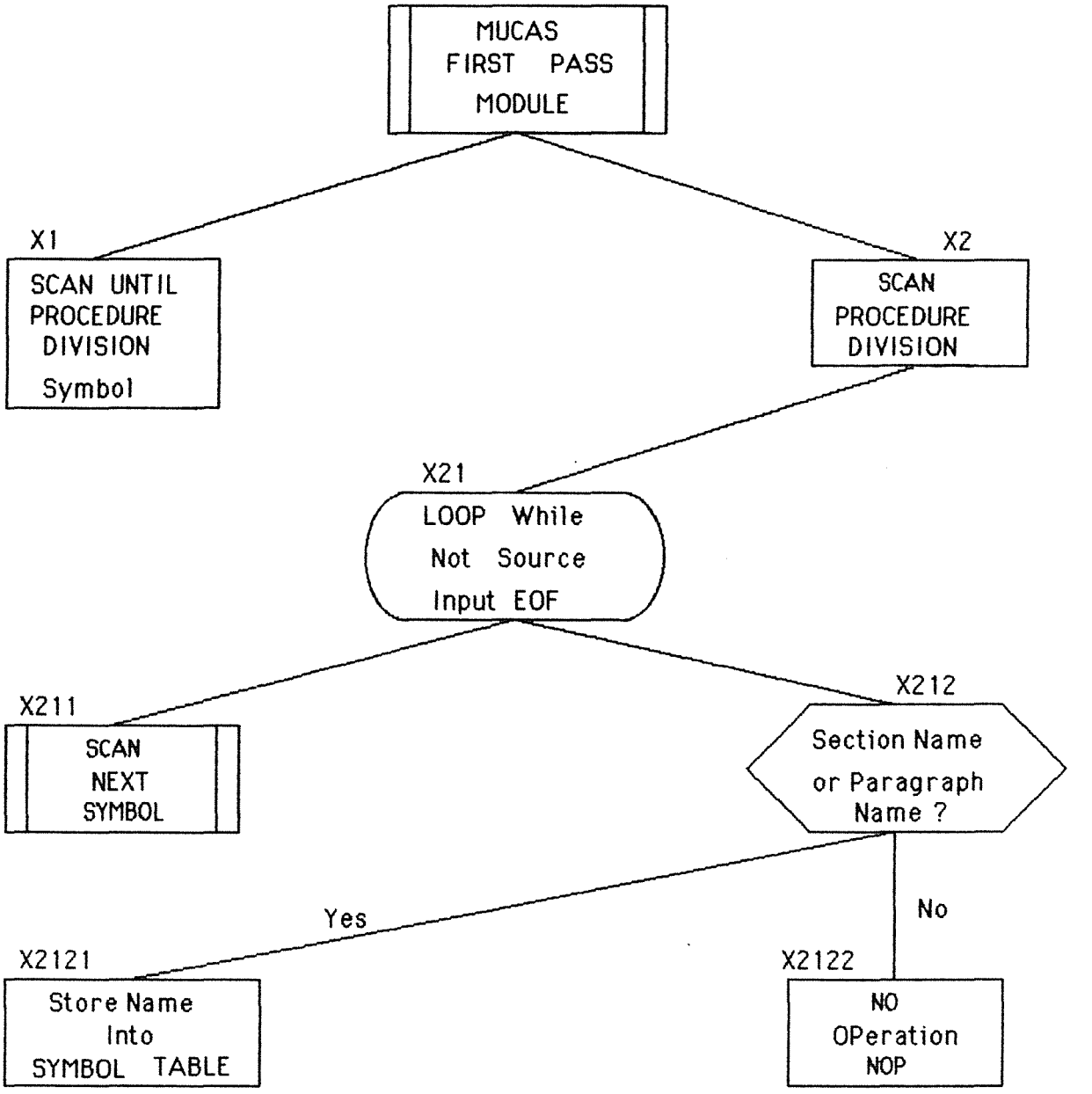


Figure 4.4 MUCAS FIRST PASS Module Structure Diagram.

3 MUCAS SECOND PASS Module

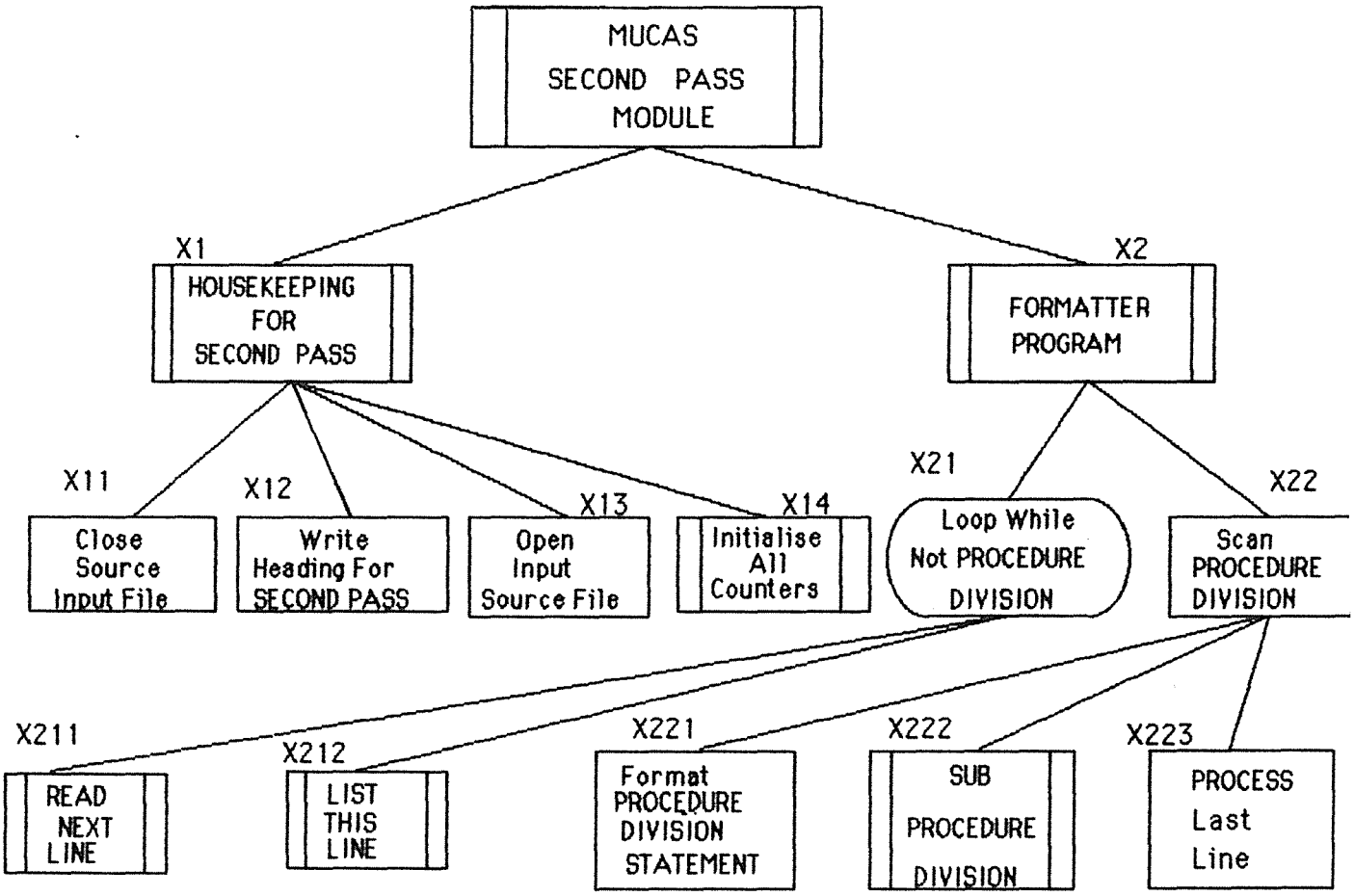


Figure 4.5 MUCAS SECOND PASS Module Structure Diagram.

4 MUCAS TERMINATION Module

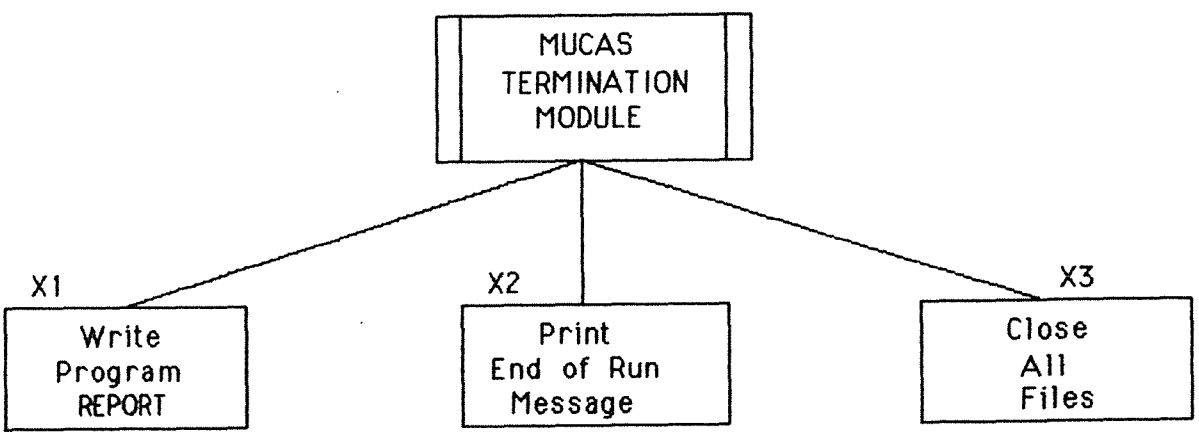


Figure 4.6 MUCAS TERMINATION Module Structure Diagram.

4.4.2 THE HOUSEKEEPING FUNCTIONS OF MUCAS

Any implementation of MUCAS should be designed to run in a demand mode or batch mode. Every usual run of the system will consist of a stack of programs to be run one after the other. The HOUSEKEEPING module performs the inputs/outputs of MUCAS, and carries out the following functions :

(a) It initialises the system for the start of a new batch run.
(b) It controls the flow of jobs through the system.
(c) It provides the following types of input to the parsing, formatting and complexity measure modules :
    (1) A source COBOL program.
    (2) A set of input data to initialise the ST.
    (3) The run options selected by the user.
(d) It produces output as follows :
    (1) A reformatted COBOL program.
    (2) A set of control flow graphs (produced by MUCAS phase II);
    (3) The corresponding data attributes for storage in a program database.
    (4) A source input COBOL program symbol trace listing.
    (5) A program complexity report of the reformatted COBOL program.

4.4.3 **THE VARIOUS MODULES OF MUCAS**

The HOUSEKEEPING module controls the flow of jobs through the system and is responsible for providing the input to the Parsing, Formatting and Complexity measure modules. The source COBOL program is read from the input file, and superfluous blanks are trimmed off and an internal end of line character is inserted where applicable. The amended source text is then inputted to the parsing module. The input function module includes a lexical scanner to scan the characters of the source program, from left to right, and generate the actual **symbols** of the program such as : identifiers, reserved words, numbers, character symbols. The following method of scanning is used. The original source text needs to be available at the actual syntactic analysis time, and later at both formatting and complexity measurement for meaningful error messages to be produced. It is implemented by lexical analysis in the parsing module via a subroutine called SCAN_SYMBOL. This subroutine is called by the syntax analyser (NEXT_SYMBOL subroutine), whenever it needs a new **symbol.**

Separation of the lexical analysis at the beginning of the design phase makes the input function module very simple, with all the work load falling on the parsing, formatting and complexity measuring modules.

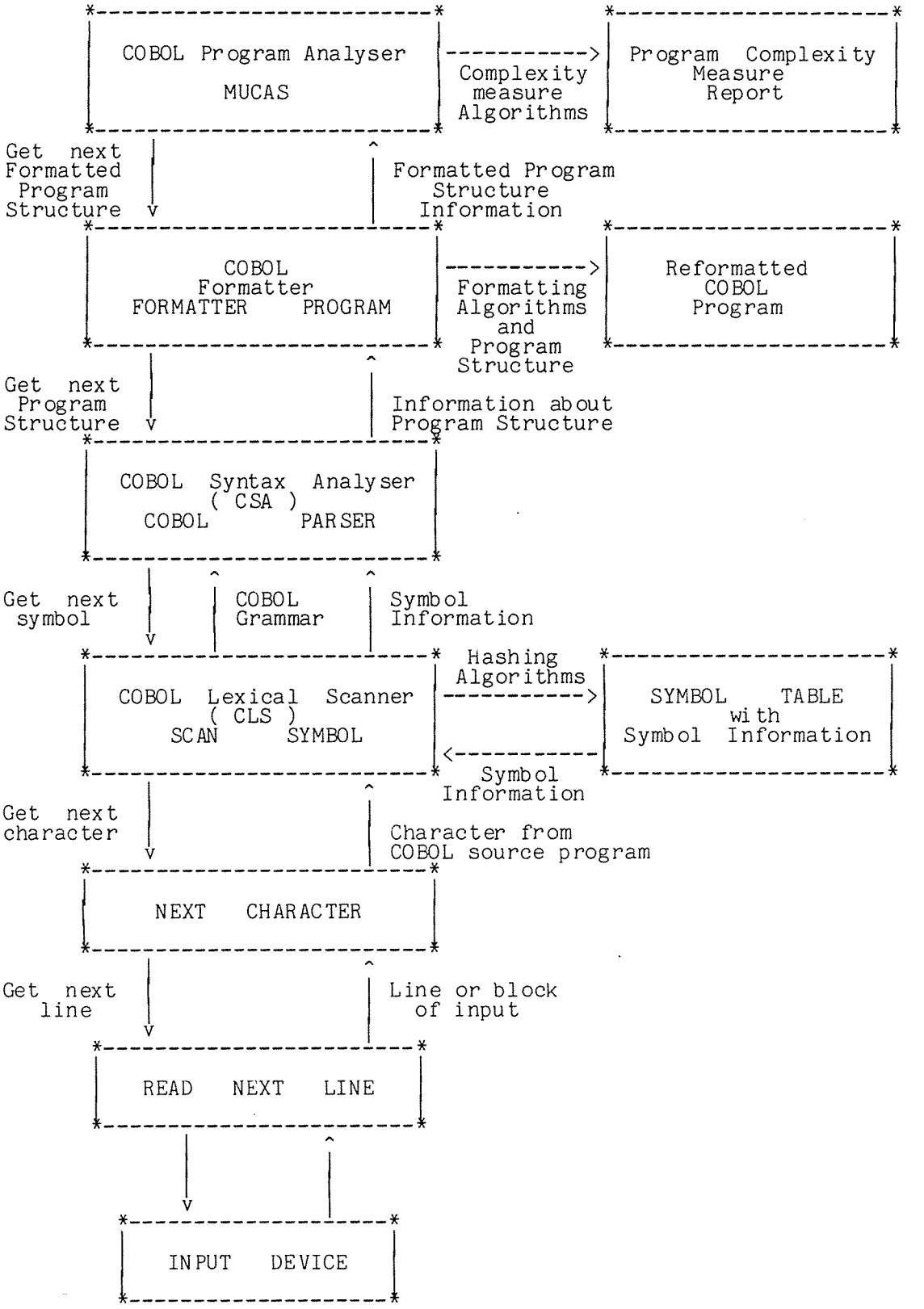


Figure 4.7 Functional Modules of MUCAS.

4.4.4 THE PARSING FORMATTING AND COMPLEXITY MEASURE OF MUCAS

The formatter module (**FORMATTER_PROGRAM**) module performs most of the workload required to compile a COBOL source program. Both lexical analysis and syntactic analysis are performed to recognise a symbol. The formatting and complexity measurement also operates whenever a construct in the source language is evaluated. The parsing and formatting actions check for syntax correctness, store information in the ST or output reformatted language operations as well as measuring a complexity characteristic of the reformatted program. For example when a symbol is recognised, a search update routine (UPDATE_SYMBOL) will check the ST to make sure it has not been used before. If not, it adds the new symbol to the ST with its attributes (if any).

4.4.4.1 THE LEXICAL ANALYSIS STAGES OF MUCAS

The design of the lexical analysis stages of MUCAS is based on the approach described in Chapter 11 of [FRE75], Chapter 3 of [AHO77] and [GLA69]. It is usual to separate the lexical scanning process from the other stages, for example, the syntax analysis stages. There are many reasons for this separation, the most important one being that lexical scanning is a simpler and more efficient process than syntax analysis [MCK74]. It allows machine independence despite differing character sets. Character set dependence can be reduced to one small module of the lexical scanner containing the relevant I/O statements.

The lexical analysis is performed by a routine called SCAN_SYMBOL (Cobol Lexical Scanner CLS). This routine is called by the syntax analyser whenever it needs a new symbol (one at a time or for an entire statement) [FRE75]. When it is called, SCAN_SYMBOL recognises the next source language symbol and passes it to the Syntax Analyser (SA). In this way, the whole of the internal form of the source program is not required to be in core at the same time, as would have been the case if the lexical analysis had been performed in the input module. The symbols in COBOL fall into one of the following

classes :

(a) Identifiers e.g. 1020-READ-ROUTINE, 01-STUDENT-NAME.
(b) Reserved words e.g. ADD, IF, PERFORM, WRITE.
(c) Literal - numeric literals (686, 56.38) and non-numeric literals ('PROGRAM RUN').
(d) Single character delimiters e.g. +,-,*,/,=,;,.
(e) Double character delimiters e.g. **
(f) Relational, logical and string operators e.g. <,>,=

The tokens in the COBOL language are expressed by a regular grammar (context free grammar) [AHO77] [MCG80], it may needed maximum of k characters ($1 \leq k \leq 18$) in order to determine a symbol whether it is a identifier (e.g. 01-STUDENT-NAME) or a numeric literal (e.g. 1026) [MCK74]. Further explanation concerning the grammar is discussed by [MCK74] page 104-105.

Based on the concept of a finite-state machine [FRE75], a finite state recogniser (SCAN_SYMBOL routine) was built. This type of recogniser was implemented by keeping a table of states and inputs. An input character is used to index the entries for their current state and to retrieve an action and a new state from the table. The recogniser begins in an initial state. When an exit state is reached, a token is recognised. The associated action is executed (for example, to accumulate the digits of an integer) and the new state is entered and the process subsequently repeated. In the Lexical Analysis (LA) module of MUCAS, many routines are encoded within it for resolving the various source symbols. These routines are :

(A) **NEXT_CHARACTER** routine - This routine gets the next character from the COBOL source language record (line) and performs the following functions.
    (a) Ignores blank characters - this is handled by a routine to get the next non blank character.
    (b) Obtains the source character and make it available to the

routine requesting it within SCAN_SYMBOL. This is almost entirely a machine-dependent operation, and was straightforward regardless of the next line fetched.

(c) Prints a source language line on picking up the internal end of line character (Hexadecimal code $0A_1X$).

(d) Keeps track of where in the source text the MUCAS analyser is operating by a character position pointer.

(e) Detects the end of a source line, prints it, then automatically fetches the next line and updates the character position pointer.

(B) **NEXT_SYMBOL** routine - This routine picks up the next symbol in the source text, which may be a identifier, reserved word, literal (numeric or nonnumeric), operator, etc. Each type of symbol in the language has a special routine to process it. Thus, the NEXT_SYMBOL by means of the NEXT_CHARACTER routine determines enough characters to resolve which type of symbol it is and then turns control over to the special routine for processing that symbol. These special routines are :

(a) **SCAN_NAME** routine - NEXT_CHARACTER having detected the first character of symbol as being a letter, turns control over to the SCAN_NAME routine. This initialises the name holding area to blank and then by means of NEXT_CHARACTER routine builds up the name character by character until it detects an end of name character (i.e. a character other than a letter, a digit or a hyphen "-"). Since COBOL imposes the restrictions of a maximum of 30 characters in length for a name, going beyond this produces an error message inserted in the output trace file.

(b) **SCAN_NUMBER** routine - This routine picks up an integer or real number. It performs the following functions :

(1) Initialises the number holding area.

(2) Transfers control to SCAN_NAME routine if an alphabetic character is detected.

(3) Accumulates the number digit by digit.

(4) Deals with any decimal point occurring within the number.

(5) Detects illegally large numbers.

(6) Distinguishs between real and integer numbers.

(7) Detects the end of the number where a separator character (e.g. blanks) is encountered. The separator character terminates the group being read.

(c) **SCAN_OP** routine - This routine resolves operators and separators, such as +,-,*,/,.,(,) and the multiple operator ** (for exponentiation). The relational, logical and string operators are also resolved using this routine. SCAN_OP does the following :

(1) The operator is read into the operator holding area and multiple character operators are detected.

(2) Detects end of character group whenever a separator character (e.g. blanks) is encountered. The separator character terminates the group being read.

(d) **READ_NEXT_LINE** routine - This routine reads in the next line from the input COBOL source file. Since the CLS can work on an entire statement at a time, the NEXT_CHARACTER routine can in turn call upon the READ_NEXT_LINE routine to get the next block of information from the input COBOL source file (Figure 4.7).

The division of the input functions into NEXT_CHARACTER and READ_NEXT_LINE in CLS will reduce maintenance. For example, the MUCAS analyser may accept input from several different types of devices - a new device may be added. The READ_NEXT_LINE routine can be modified easily to interface to the new device without changing any other routines of the analyser.

All of these routines process their character groups until separator characters are encountered. The separator character (e.g. blanks) terminates the group being read. The SCAN_SYMBOL routine, thus supplies the syntax analyser with a representation of each symbol in the source language and a pointer to the actual symbol itself, as well as an attribute showing the symbol type (name, number or operator). This is more efficient than returning the variable length strings of actual symbols to the syntax analyser. Note that the SCAN_NAME routine

makes no distinction between identifiers (section name, paragraph name, etc.) and language keywords (reserved words).

The input of a COBOL program is unit-record oriented (80 significant columns) and so continuation marks have to be recognised. The MUCAS analyser also needs to keep track of errors in order to indicate clearly where a syntax error occurred. A special error mark (^) is placed in the stream after removing excess blanks and other control characters.

Any symbols in the input stream are differentiated as to type, and their values are stored. The symbol name of each variable is saved for later printing and correlation with other occurrences of the name. A pointer to the characteristics of a symbol, as well as its class, is included as a descriptor in the ST.

4.4.4.2 **SYMBOL TABLE CONSTRUCTION**

A symbol table provides a way of associating pairs of objects [GLA69]. This permits the efficient association of information with the symbols in COBOL program translation that are used in all stages (lexical analysis stages, syntax analysis stages etc.) [FRE75]. The symbol table construction is a symbols database. During the compilation phase, it is organised as follows :

(a) **Name Pointer** - This occupies four digits and represents a pointer, to the actual name which is stored in the name area. The first two digits are the bucket number and the record number consists of the next two digits. The bucket hashing method [TRA63] is used in the table lookup technique.

(b) **Overflow** - In the event of hashing to a bucket in the symbol table when the bucket is full, the next bucket will be referenced as the overflow area. Details of this procedure are described in symbol table lookup techniques section.

(c) **Type** - One digit was used to identify the various types of symbols present.

Currently there are two thousand positions (separated into 100 buckets numbered 00 to 99) allocated for the ST in the present MUCAS implementation, and three hundred and eighty four of them are allocated for COBOL reserved words. The remainder are used for identifiers, literals etc. and the overflow area. Subsequent manipulation of the ST in MUCAS II will be based on the MUCAS I ST. Detailed individual symbol information (number of occurrences, locations of occurrences, etc.) of the reformatted COBOL program will in due course be stored in mass storage files.

4.4.4.2.1 SYMBOL TABLE LOOKUP TECHNIQUES

The basic computing methodology for static table searching is discussed in Price [PRI71]. Techniques for handling dynamic tables are mentioned in [MOR68], [GLA69], [HOP69], [GON84] and [LUM71]. Methods described are : sequential search, sort/merge search, binary search, tree search, estimated entry, and direct entry (hashing method). All these have advantages depending on the application for which they are being used ([MCK74] Page 296-301). Table lookup time is the primary criterion of technique evaluation, and the hashing method is selected as the best searching method for COBOL ST lookup. There are many schemes which may be adapted to this method. Since ST access consumes a major portion of the processor time during translation [MCK74], an efficient table lookup technique is very important for run-time efficiency. It may be a fruitful area for run-time savings since fast techniques can be programmed almost as easily as slow ones. Two important considerations with STs are their size and the speed of access to their entries. The format of the ST and algorithms for finding information in it were therefore investigated with a view to achieving good performance. The selection of search structures for the MUCAS system must optimise the balance between the search requirements and the structure requirements ([KRU84], [MCK74], [BAK74], [GON84] and [GRI71]). The speed of the associated structure mechanisms, namely,

the search and insert mechanisms, is the criterion for selecting a search structure [FRE75]. The search structure that optimises overall system data processing requirements is the proper search structure for the system. After much testing, a technique combining the bucket hashing method [TRA63] and the radix transformation technique [CHA84] was used for the MUCAS ST lookup, because of the size and frequency of reference to the ST.

The design methods for the ST data structure and update procedure are derived from the database management system SEQUEL (Macleod, Ian A. [AUE80], pp 375-389), SMART (Salton, G. [AUE80], pp 391-403) and FIRST (Dalotta, Robert T. [AUE80]). The data considered consists of an abstract, from which a linked list of index terms are extracted. Descriptive information about the program construct is also listed. The distributed CODASYL database processing method of Olle [OLL78], Germano [GER81] was not found to be suitable for data structures of the MUCAS ST, because of its slower processing time and implementation difficulty. For further reference to the CODASYL database method such as UNIVAC's DMS 1100 (Currently at level 10), HONEYWELL's IDS/S, CULLIMANE's IDMS (For IBM and ICL machines) and SIMENS' UDS, see [OLL78], [DAT81], [FER83] and [VET81].

The MUCAS ST contains the symbol name, together with other descriptors associated with the symbol. The key is the first six characters of the symbol name. When information about the symbol is desired, the ST is searched for that name. The ST must be looked up on each reference to any symbol in the COBOL program. In lexical analysis, symbols (tokens) are looked up to determine whether or not they are already in the ST and if not, they are entered in.

In the MUCAS system, the ST developed at translation time is retained during the execution of the MUCAS system. Each symbol in the ST is self-defining in that the string of characters representing it defines the intended value. The characters are stored and an internal name is

provided to refer to them.

4.4.4.2.2 **RELATE SYMBOL TO NAME LIST**

The procedure UPDATE_SYMBOL was implemented for ST lookup to relate the symbol to its name list value. The ST entries are searched by name using the search key (first six character of the symbol). This was converted or hashed to a **D**irect **B**ucket **A**ddress DBA (hash address) into the ST. This is then used to locate or place the proper symbol.

There are many variations and schemes which can be to used with this method. The one chosen for use in MUCAS is the division-remainder method, which operates as follows :

A prime number close to the number of buckets in the ST is chosen (in MUCAS case it was 101) and is used as a divisor to extract a quotient and a remainder (DBA) from the original key N_α (generated from the input symbol). The remainder is the transformed key and is the original index of the chosen bucket number in the ST. A hashing function was implemented to perform the transformation process. It uses the input symbol to obtain an address (DBA, hashed address) where information concerning it can found :

$$B_\eta(N_\alpha) = N_\alpha \bmod M$$

Where N_α is the original key,
M is the prime divisor (close to the symbol table bucket size) and
$B_\eta(N_\alpha)$ the hashed function.

The sequence of processes for symbol search update are as follows :

(1) Randomisation of DBA is achieved by the hash function (in order to make the symbol evenly distributed among buckets). The first six characters of the input symbol are converted to their corresponding machine character sequence number C_i, where i is

the input character position number and $1 \leq i \leq 6$. The number C_i is matched to a prime number between 3 and 733 (128 of them) to give $F(C_i)$. The six $F(C_i)$s are summed (giving N_α) then divided by the prime number 101. The remainder is used as a **bucket number** $B_\eta(N_\alpha)$ (hashing number). If the bucket number $B_\eta(N_\alpha)$ is equal to 100, then $B_\eta(N_\alpha)$ is re-calculated as N_α divided by 99 instead of 101.

This randomisation process is outlined as follows :

$$N_\alpha = \sum F(C_i) \qquad \text{Where } 1 \leq i \leq 6$$
$$B_\eta(N_\alpha) = N_\alpha \text{ MOD } 101 \qquad \text{iff } B_\eta(N_\alpha) \neq 100$$
$$B_\eta(N_\alpha) = N_\alpha \text{ MOD } 99 \qquad \text{iff } B_\eta(N_\alpha) = 100$$

Where C_i is the collating sequence number (For Prime 750) of character number i of the first six characters of the input symbol.

$F(C_i)$ is a function of character number i of the first six characters of the input symbol.

$F(j)$ is the first jth prime number [ORE48] in the range 3 to 733, and

$B_\eta(N_\alpha)$ is the hash address or DBA or bucket number.

iff means if and only if.

(2) A sequential search for an empty slot or a slot pointing to the given symbol is begun at the randomly obtained symbol name list slot. If the symbol is found in the slot, then the information from the ST is retrieved.

(3) If the slot is empty, it is filled with the name's attributes.

(4) The pointer which indicates the number of symbols present in that bucket is adjusted to give the location of the next unused cell.

(5) If the bucket is full, the name list overflow in that bucket number is detected. The new symbol's attributes are then stored in the next available bucket.

(6) When ST overflow is detected (indicating that the buckets are full), processing stops and an error message is sent to the user.

Collision is the hashing of two different hash addresses (keys) to the same table position. While it is possible to devise schemes to minimise "hash clash", its possibility can never be eliminated because the probability exists that duplicate keys can be generated by the transform method. The resolution of duplicate keys or "hash collisions" are handled by the bucket hashing method (by means of the UPDATE_SYMBOL routine). All the same hash key number for the input symbol will enter to the same bucket. If the specific bucket number is full, the next bucket number bucket will serve as the overflow area until the ST is full. This method requires that enough space be allocated to provide for all overflows. It causes inefficient use of storage space. The choice of the divisor can lead to substantial bias in symbol distribution to buckets. For example, if the number chosen is even then the hashed key B_η will be even when the original key N_α is even, and odd when original key N_α is odd. Also choosing a number as the divisor which is a power of the radix of the computer being used could produce many collisions.

In many references ([AHO77], [GLA69], [GRI71] and [MCK74]) it is stated that the use of a prime number as a divisor in the **division-remainder method** most evenly distributes the symbols throughout the ST. Practically, it was found after a number of tests that using a prime number in the radix transformation technique [CHA84] resulted in more even symbol distribution than using a non-prime number.

In using this method the ST must start from zero and not one. It should be noted that using the division-remainder method symbols (identifiers) with names such as, 010-ST, 100-TS, ST-001 all hash to the same bucket number in the ST. In many translators the reserved words are "hashed" directly into the ST for ease of access. In the case of MUCAS, adoption of this method (direct hashing method) would involve larger overhead in the system. The specific hash algorithm chosen generally represents the best tradeoff in storage requirements and search time. As the hash table is expanded the probability of a

collision is reduced. This in turn reduces the size of the collision table (overflow area), thus reducing search time.

Chang [CHA84] mentioned another method, the ordered minimal perfect hashing scheme which is based upon the chinese remainder theorem and prime number functions. It is a perfect hashing [JAE81] [COR85] scheme which guarantees that the keys are hashed in ascending order without collision. Actually, there are several ways to tradeoff search and insert timing with available storage space. For this reason there is no general set of timings applicable for all hash algorithms. Timing characteristics may be tailored by tradeoffs in storage space and hash methods [PRI71]. If there is an abundance of storage space a superior hash algorithm can always be constructed [GLA69].

4.4.4.3 **THE SYNTAX ANALYSIS STAGES OF MUCAS**

Syntax analysis is concerned with grammars that can represent the language, the amount of processing and which of the feasible parsers is best [FRE75]. The grammar in COBOL language are mostly expressed by the LL(1) grammar (Left to right, Leftmost derivation grammar) [AHO77] [MCG80], certain part of the COBOL programming syntax needed k characters ($1 \leq k \leq 3$) in order to determine a path, i.e., some parts of COBOL programming syntax are considered as LL(2) and LL(3) [BAK74] [MCK74]. For example, the DIVIDE, EVALUATE, INSPECT, PERFORM, SUBTRACT and USE statements. In MUCAS we required a parser that will give a meaningful diagnosis of syntactic errors, and still be able to continue processing subsequent source text with a reasonable degree of certainty of arriving at a correct diagnosis. The top-down recursive-descent [MCG80] [AHO77] method was used for the language syntax analysis. The syntactic structure of the COBOL language is expressed as tram-line diagrams in Appendix C. Syntactic analysis was then achieved by having a routine (syntax driver) to step through this tram-line diagram.

4.4.4.3.1 MUCAS PARSING METHOD

The Backus-Naur Form (BNF) [LIN77] partial syntax for COBOL [COB81] is defined in Appendix B. The COBOL parser parses the COBOL program text on these grammatical rules. There are many different algorithms for parsing and these are influenced by the form of the language to be parsed [FRE75].

The development of efficient parsing algorithms is a highly developed area of abstract theoretical analysis, practical analysis and practical construction of software systems [FRE75] Chapter 12. Many practical algorithms have been devised and studied from a mathematical stand-point. A thorough, programmer-oriented survey of the theory for parsing algorithms can be found in [WEI73] and [AHO77]. The MUCAS COBOL parser skips the IDENTIFICATION DIVISION, ENVIRONMENT DIVISION and DATA DIVISION, parsing only the PROCEDURE DIVISION, to determine the constructs of the source COBOL program.

The syntax analyser receives symbols from the source handler, and the text position coordinates indicate the current symbol offered by the lexical scanner. Whenever necessary, the syntax analyser uses this value in reporting errors to the source handler.

The previous interface defined is sufficient for the process of syntax analysis. But the knowledge of user-defined symbols, the spelling of each identifier and the value denoted by each integer or character constant must be catered for by the MUCAS analyser. The syntax analyser and MUCAS analyser must transmit this information, and hence the lexical scanner must enable the syntax analyser to do so.

4.4.4.3.2 TOP-DOWN RECURSIVE-DESCENT METHOD

Top-down parsing checks to see whether or not a group of one or more symbols in the input stream satisfies a particular production rule and assumes that the string satisfies the right-hand side of this rule. If it is correct, it has recognised the string. If not, it goes back up to the beginning of the string to see whether the string satisfies another production rule. An alternative approach that eliminates the backup problem is a parser that makes all possible predictions, ultimately keeping only the one correct parse. A parse is incorrect when :

(1) A match cannot be made with any prediction,
(2) The grammar is satisfied before the string has been completely parsed, or
(3) The string is exhausted before a production rule with the distinguished symbol as its left-hand side is satisfied.

The parser searches the table for a rule whose left part corresponds to that being currently examined. If the production rules are well arranged and the table contains information about what rule to look for next, the search time is shortened. Once a rule has been chosen, the parser attempts to match the input with this rule. The parser saves pointers for backup and saves the information in stacks pending satisfaction of production rules. It also provides a means of outputting information. The latter can be achieved by having a formatting routine associated with each production rule placed in an additional column in the table. Whenever a production rule is satisfied, the associated formatting routine is invoked.

A parser that uses a set of recursive procedures to recognise its input with no backtracking is called a recursive-descent parser [AHO77] [MCG80]. In this method the parser or syntax analyser has one recursive routine for each production in the language. The routine uses the rules of grammar as productions and checks where in

the program it can start looking for a phrase for a particular nonterminal (NT). It is goal oriented. The routine finds its phrase by comparing the program at the point indicated with the right parts of rules for NTs and calls on other routines to recognise subgoals where necessary. The action to be taken on recognition of a production is built into the recursive routine. The recursive procedures are usually quite easy to write and are fairly efficient if written in a language that implements the procedure call efficiently. This method therefore allows changes to be readily implemented and so is ideal when the translator is written in a high level language such as PL/I. When assembler language is used changes would be more difficult to implement since it would be necessary to set up the mechanism for recursive calls.

4.4.4.3.3 **TOP-DOWN SYNTAX-DIRECTED METHOD**

In this method the syntax of the language is encoded directly. A syntax control routine or syntax driver is used to step through the syntax starting with the initial goal of the syntax and attempts to generate a string of terminal symbols, to match the source language statement being translated. One of the problems with this method, is how to represent the syntax in the computer. A method used by Cheatham and Sattley to represent the syntax by a pair of tables is the usual method of approach [SIM73a]. On small computers, syntax tables are not particularly suitable, since they occupy a large amount of core storage although a great many entries in the table are empty. Another method of representing the syntax is in a list structure form known as a syntax graph, in which pointers are used to link the various components and alternatives of a production (Cohen and Gotlieb [SIM73a]). This method of representation is economical on core storage and uses less complicated parsing algorithms. Since backtracking makes parsing very slow, more attention has been focused on ways of reducing or eliminating backtracking.

4.4.4.3.4 BOTTOM-UP METHOD

Bottom-up parsing treats grammar rules as reductions [AHO77]. It attempts to reduce the entire program to the initial symbol of the grammar and checks to see whether or not part of the string satisfies the right-hand side of a production rule. If a reduction can be made, that part of the string is replaced by the left-hand side of the production rule. Parsing then continues with the modified string. If at any point in this process the string cannot satisfy a production rule, backup must occur. To avoid backup, the grammar is usually modified or restricted [FRE75] [AHO77]. Since most of the COBOL syntax that we are parsing is expressed in a top-down hierarchical manner, it is not advisable to use bottom-up parsing techniques. Other reasons for not implementing bottom-up parsing are that it is more complicated in design and consumes more processing time.

4.4.4.3.5 THE STRUCTURE OF THE PROCEDURE DIVISION OF COBOL

COBOL was designed to be easy to use and English-like in format. It has sentences, clauses, paragraphs, and sections in addition to the usual statements and expressions. COBOL programs thus tend to have more structure than programs of many other languages [FRE75]. Figure 4.8 shows the major parts of a COBOL program.

```
==========================================================================
DATA
  TYPES      : Alphanumeric, Computational ( Binary ), Conditions
  STRUCTURES : Simple variables, Arrays ( up to 3 dimensions, COBOL-80 has
               up to 7 dimensions [GAR84] ), Tables, Groups, Hierarchical
               structures, Files.

VERB Classification
 Hardware Dependent Input-output ( verbs: partial list ):
    ACCEPT       CLOSE        DELETE       DISABLE      DISPLAY     ENABLE
    OPEN         PURGE        READ         RECEIVE      RELEASE     RETURN
    REWRITE      SEND         START        WRITE

 Hardware Independent
  Data Manipulation ( verbs: partial list ) :
    ACCEPT (DATE, DAY or TIME)    ACCEPT MESSAGE COUNT
    EVALUATE       INITIALIZE     INSPECT (REPLACING)   MOVE
    SEARCH         SET            STRING                UNSTRING

  Arithmetic ( verbs: partial list ) :
    ADD            COMPUTE        DIVIDE          INSPECT (TALLYING)
    MULTIPLY       SUBTRACT

  Program Control ( verbs: partial list ) :
    CALL           CANCEL         CONTINUE      EXIT      GO TO
    IF             PERFORM        STOP          USE

  Productivity Aids ( verbs: partial list ) :
    GENERATE       INITIATE       MERGE           SORT
    SUPPRESS       TERMINATE      USE (Debugging)

  Conditional :
    ADD (SIZE ERROR)          CALL (OVERFLOW)       COMPUTE (SIZE ERROR)
    DELETE (INVALID KEY)      DIVIDE (SIZE ERROR)   IF
    MULTIPLY (SIZE ERROR)     READ (END or INVALID KEY)
    RECEIVE (NO DATA)         RETURN (END)          REWRITE (INVALID KEY)
    SEARCH                    START (INVALID KEY)   STRING (OVERFLOW)
    SUBTRACT (SIZE ERROR)     UNSTRING (OVERFLOW)   WRITE (END or INVALID KEY)

  Ending :    STOP

  Inter-program communication :        CALL       CANCEL

  Ordering :                 MERGE     RELEASE    RETURN      SORT

  Procedure branching :      CALL      EXIT       GO TO       PERFORM

  Report writing :           GENERATE  INITIATE   SUPPRESS    TERMINATE

  Table handling :           SEARCH    SET

PROGRAM STRUCTURES :   Four program parts :
     IDENTIFICATION DIVISION ( Required )
     ENVIRONMENT DIVISION    ( Optional )
     DATA DIVISION           ( Optional )
     PROCEDURE DIVISION      ( Optional )
    Sections   (   named   )              within divisions
    Paragraphs (  may be named  )         within sections
    Sentences  ( one or more statements ) within paragraphs
    Statements composed of verbs and their arguments ( May be nested within
               conditional statements )

SYSTEM ENVIRONMENT:
    Intended for batch processing
    Compilable
    Library facilities

==========================================================================
```

Figure 4.8 Major Parts Of COBOL Program.

The PROCEDURE DIVISION of a COBOL program looks like a complete program in other languages. The flow of control proceeds sequentially in the normal manner, guided by conditional and transfer statements. Program text from a library can be included in a program at compile time. The PERFORM verb permits a named paragraph to be executed (a controlled number of times) out of its usual sequence. External programs written in other languages can be linked to a COBOL program [ROB76].

COBOL programs are largely self-documenting and tend to be quite readable [KER79]. A COBOL program consists of four divisions, these are : IDENTIFICATION DIVISION, ENVIRONMENT DIVISION, DATA DIVISION and PROCEDURE DIVISION. COBOL's PROCEDURE DIVISION consists of **sections**, each of which consists of **paragraphs** which are composed of **sentences** which can be divided into **statements.** This is the order into which we divided the PROCEDURE DIVISION. Statements can be further sub-divided into verb, modifier and others tokens (Figure 4.9 and Figure 4.10) [STE79] [COB81]. In MUCAS I, we formatted the COBOL program in this manner. Individual statements and modifiers were indented to show structure level.

The most interesting features in COBOL are found in the DATA DIVISION. Every piece of data to be accessed or manipulated by a COBOL program must be defined explicitly. There are two basic forms of data that are defined : external files of information (which are defined in the FILE SECTION of the DATA DIVISION) and internal working storage (defined in the WORKING-STORAGE SECTION) [COB81]. Files of information existing on an external storage medium can be described to whatever level of detail is necessary. This information can be brought into memory and then moved into working storage for processing [FRE75].

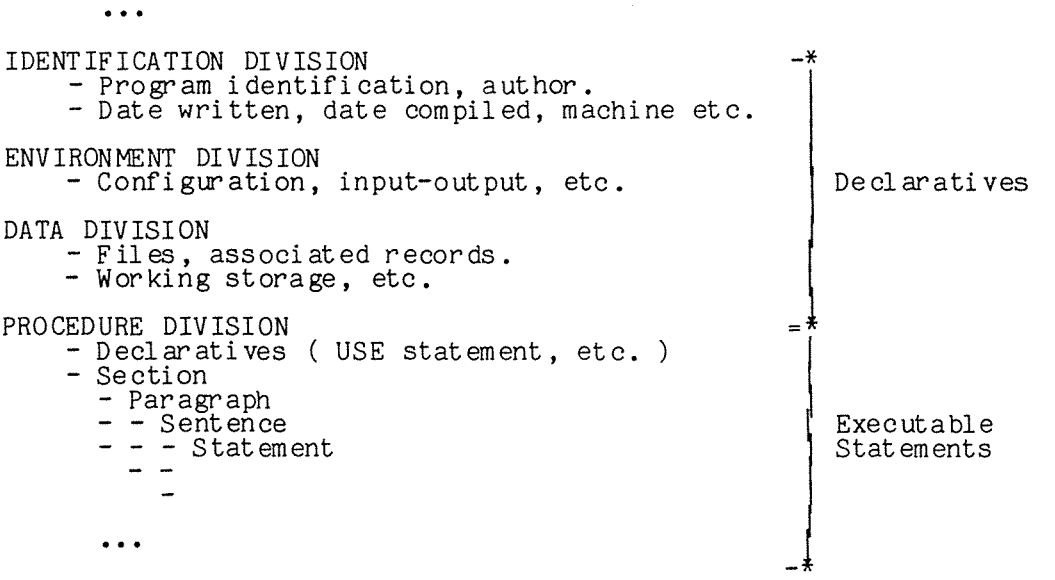


Figure 4.9 COBOL Program Structure.

UNIT NAME	FUNCTION
Program	Unit of execution
Section, Paragraph	Unit of modularity
Sentence, Statement	Unit of sequencing
Expression	Value producing unit
Token (Symbol) Reserved words Identifiers Literals	Molecular unit of syntax
Character	Atomic unit of syntax

Figure 4.10 COBOL Program Elements Hierarchy.

4.4.4.3.6 COBOL STATEMENT ANALYSIS

4.4.4.3.6.1 COBOL STATEMENTS

The elementary actions of evaluation, assignment, and control of evaluation order are specified by the statements of a programming language. Statements in COBOL are rather diverse in form and in meaning. For example, A simple statement does not contain any embedded statements. ACCEPT, DISPLAY and GO TO are three examples of simple statements. A compound statement may contain one or more embedded statements. The IF statement is an example of an embedded statement.

4.4.4.3.6.2 TYPES OF STATEMENTS IN COBOL

(1) Computation statements. - These are statements that apply operators to operands to compute new values. The COMPUTE statement is the prime example.
(2) Sequence Control statements. - In a COBOL program, control automatically flows from one statement to the next. Certain statements such as GO TO, GO TO DEPENDING ON, CALL and PERFORM deliberately alter the flow of control.
(3) Structural statements. - Certain statements such as EVALUATE serve to group simple statements into structures.
(4) Declaration statements. - These statements generally produce no executable code and inform the compiler about the attributes of names encountered in the program. This type of statement is not considered in MUCAS I.
(5) Input and output statements. - These statements serve the input and output purpose of a program, for example READ, WRITE and REWRITE statement.

There are forty seven different types of statements in COBOL's PROCEDURE DIVISION (see Appendix B and Appendix C). The statements look like sentences. In a COBOL program, identifiers denote constants, variables, paragraph names and section names. Their association must

be unique within each program in which they are declared. For an identifier, a maximum of 48 levels of structures can be declared in the COBOL program's DATA DIVISION [COB81]. COBOL syntax is very complicated in terms of programming making the writing and design of a COBOL parser difficult.

4.4.4.3.6.3 **OBTAIN STATEMENT TYPE**

The STATEMENT routine distinguishes statement types by the **verb symbol** which is a COBOL reserved word. This routine perform functions such as :

(1) Read the verb symbol by using the NEXT_SYMBOL routine.
(2) Verify that the name is a verb symbol, not a normal name.
(3) Use the table lookup technique to identify the **symbol** and then branch to the appropriate statement type routine.

4.4.4.3.6.4 **PROCESS STATEMENT TYPE**

The process statement module is a collection of routines, one per statement type in the source language. Taking an example, an arithmetic statement such as COMPUTE contains arithmetic expressions, the COMPUTE statement routine must perform the following functions :

(1) Recognise the end of the arithmetic statement, or the end of the arithmetic portion embedded in a nonarithmetic statement.
(2) Detect bad punctuation.

Implementing the COBOL parser involves considerable difficulty, such as recognising conditional expressions. A large volume of statement syntax is involved (some statements have very complicated syntax).

4.4.4.4 FORMATTING ANALYSIS STAGES OF MUCAS

4.4.4.4.1 METHODOLOGY OF FORMATTING

When the syntax analyser recognises a source language construct it invokes a formatting routine for that construct. This takes it, checks it for correctness and stores the necessary information about it into the ST before outputting it to the reformatted program file. Proper alignment of data names and verbs within the COBOL program will greatly enhance the readability of the listing [KER79]. The reason for making the COBOL source text indented is so that indentation can be closely linked to the flow of control [OTH81] and therefore guides :

(a) The program writer to check that the control sequence is as intended.

(b) The program reader as to how the sequence control may be understood. Because GOTO's break the nested SP structure of a program, they should be highlighted in the source text. The rewards for enforcing the indentation rules are :

(1) The program will be more likely to be correct because conditions must be considered carefully.

(2) The structure of the COBOL source text will reflect the flow of control [OTH81] at execution time, thus making the program more readable. The general flow of the program will be indicated on the least indented lines and detailed flow by the lines in between. In fact the structure of the text is based on the structure diagram that which was drawn when the algorithm was initially developed. The setting down of precise rules for indentation enables the indenting to be checked and performed automatically wherever necessary during the parsing, formatting and complexity measure stages. Thus the source text in this stages can be correctly structured regardless of whatever or not the input COBOL source text was well laid out.

The guidelines for the indentation conventions were :

(a) That changes of level of indentation should coincide with important and recognisable constructions in the algorithm.

(b) That needless indentation should be avoided since this results in crowding of the text to the right.

(c) The rules should be simple and easy to carry out.

4.4.4.4.2 **LEVEL OF INDENTATION DESIGN**

The **parser, formatting analyser** and **complexity analyser** were incorporated in the **FORMATTER_PROGRAM** routine. In MUCAS, we achieved the formatting goals by setting the indentation rules for the COBOL source text as follows :

(1) Section names and paragraph names started at column 8. These are coded as a separate line with no COBOL instruction on that same line.

(2) Statements began in column 12 unless they were conditioned by an IF.

(3) When a single statement exceeded one line, the successive line(s) were indented. Two columns of indenting is recommended.

(4) All statements related to an IF were indented. When IF's were nested, each level and its conditional statements was further indented.

(5) READ and WRITE statements AT END, INVALID KEY, AFTER ADVANCING and other clauses associated with file processing were coded on a separate line and indented from the READ or WRITE.

(6) Liberal use of page ejects and blank lines were used to greatly improve a program's readability.

(7) Commas were unnecessary and were not recommended unless they were contained in REMARKS lines or comments. Commas may be easily keyed as periods and hence cause errors and/or logic problems.

(8) Periods were used to close every COBOL statement (even when they were not required).

(9) Asterisk comments line were used rather than NOTE statements.

When NOTE appears as the first verb in a paragraph, all statement in that paragraph are also considered notes. NOTE statements were not supported by the MUCAS system.

Formatted and indented COBOL program examples are given by [TRI79], [LIM80], [GAR84] or Appendix E of this thesis.

Routines which were designed and used in the syntax analysis stages and formatting analysis stages are as follows :

(a) SUB_PROCEDURE_DIVISION - Parsed and formatted the COBOL program PROCEDURE DIVISION.
(b) USE_SECTION - Parsed and formatted a COBOL program USE SECTION.
(c) SECTION - Parsed and formatted a COBOL program SECTION in the PROCEDURE DIVISION.
(d) PARAGRAPH - Parsed and formatted a COBOL program PARAGRAPH in the PROCEDURE DIVISION.
(e) SENTENCE - Parsed and formatted a COBOL program SENTENCE in the PROCEDURE DIVISION.
(f) STATEMENT - Parsed and formatted a COBOL program STATEMENT in the PROCEDURE DIVISION.
(g) LINE_INDENT - Handled the current line indentation for the Formatter.
(h) NEXT_MEDIUM_CHARACTER - Formatted one character to the formatted COBOL program file current source line.
(i) NEXT_MEDIUM_SYMBOL - formatted one symbol to the formatted COBOL program file current source line and updated this symbol to the simulated database (symbol table).
(j) MED_ORG_SEQ - Initialised and formatted the next formatted COBOL source line column 1 to 6.
(k) MED_PROG_ID_GEN_SEQ - Formatted the current COBOL source line column 72 to 80 (program identification area), then outputted it to the formatted COBOL source file.

Note that the corresponding complexity measure routines are also invoked by the previous mentioned formatting routines, the complexity measure routines are STAT_PROGRAM (for the whole PC measure), STAT_SECTION (for a section complexity measure), and STAT_PARAGRAPH (for a paragraph complexity measure).

4.4.4.5 **PROGRAM COMPLEXITY MEASURE ANALYSIS STAGES OF MUCAS**

In developing a well structured program, the methodologies used for program design in MUCAS are applicable to any high level programming language (e.g., PL/I, ALGOL, COBOL or FORTRAN) that provides the basic constructs required by SP. Although there is a lack of agreement on a definition of SP, its objective clearly appears to be the control of PC. This increases program readability and maintainability [WOO79] [HIE74]. SP is intended to move programming further from an intuitive art towards a more explicit, disciplined procedure. A major objective of SP is to control PC [CHE78]. The more complex the program, the more difficult it becomes to understand how that program works [ELS76]. PC is a function of program size, the number of possible program execution paths and the control variables that direct path selection. There are a number of factors which can used to give a measure of the complexity of a computer program. These factors are :

(1) Data representation and structure.
(2) Data volume and distribution.
(3) Data communication and management.
(4) Programming language.
(5) Arithmetic and non-arithmetic logic and
(6) Program control logic.
(7) Hardware dependencies.

In a particular business environment with a given computer language configuration and a specific data structure, the complexity factor of greatest important is the program control logic. Although there are a

number of proposed measures for PC, very few computer professionals analyse programming productivity as a function of PC [CHE78].

4.4.4.5.1 BACKGROUND AND METHODOLOGY

Programming is a difficult task. In its early years, programming was considered a simple art guided only by the creative instinct of the programmer. By the 1960s, it became apparent that the difficulty of programming had been grossly underestimated and that adequate software methodologies and tools for dealing with PC had not been developed [FRE75]. Complex programs were difficult to design, difficult to write and to test, and virtually impossible to understand and to maintain [WIR74]. They were seldom produced on schedule and often did not meet specification requirements, i.e. were not reliable [BRO74]. Cost became of increasing concern in programming. The cost of hardware was decreasing, but the software cost increased steadily [BJO82] Part 1 pp 6-17. Software is becoming the most expensive and complex item in a computer installation. It is not unusual for half the cost of a computing system to be attributed to software [FEU79]. SP was introduced as a partial answer to these programming problems. The basic constructs required for SP include the control structures of concatenation, selection, iteration, and a construct for creating single entry point-single exit point modules. The COBOL programming language was selected because it is the most extensively used business application programming language today [HOW78] and is likely to remain so for some time to come.

The SP approach attempts to control PC by restricting module invocation and, in general, by restricting the use of control structures. SP does not limit the use of control variables, but in order to minimise PC the control variables are accessed in as few program modules as possible. By applying quantitative measures of PC and by inserting complexity control as a postdesign/preimplementation step in the programming process, the programmer can more effectively analyse and control the quality of a program before it is implemented.

SP was introduced to the academic community in the late 1960s (1968 to 1972) by Dijkstra [DIJ76]. It was introduced to industry in the early 1970s by Baker [BAK72]. Since that time, SP has gained immense popularity and promises to have a profound impact on the art of programming. By adopting the SP approach, the programming productivity rate increases as compared to unstructured programs and the error rate in programs is reduced as a realised benefit of SP [ZOL80].

The modularisation concept was the first step toward SP. It suggested control of PC by division of a program into independent modules. The top-down approach is an inherent part of SP. Program design is performed in steps, and at each step, a program component is divided into subcomponents.

Some guidelines have been suggested for controlling PC by IBM, e.g., a program module be limited to 50 lines of code. That is, a module is confined to one page in the source listing, thus making it easier to read and to understand [KER79]. Most programmers will agree that the length of a piece of code is not a complete indicator of PC. The task that the code performs, the control structures used, and the program variables referenced may also contribute to its complexity. Another way of controlling complexity is based on program flow [MCC76a]. PC is measured by its number of basic program paths. McCabe's [MCC76a] approach used graph theory to measure the number of program paths by simply counting the number of predicates (conditions) in a module (i.e., module complexity = number of predicates + 1).

4.4.4.5.2 **PROGRAMMING SYNTAX COMPLEXITY CONTRIBUTION TO PC**

Different users have different ways of expressing the complexity value of a program [WOO79]. Information binding, numbers of control variables or numbers of program paths could be used for measuring the PC value from a program design point of view. From a program coder's point of view, use of programming syntax will be an important contribution to program complexity measure [CHE78]. It is because

programmer capability (e.g. number of years of experience, academic background, etc.) will contribute significant factor to programming maintenance ability. The maintenance programmer who intends to modify a program can be confused by the programming syntax written by the previous programmer (i.e. the statements used in the program, his/her intention to use a statement and the way of using that statement, etc.). Therefore, to determine the module and PC value, from the program coder's and program path measure point of view, we must first evaluate the statement verb type and modifier verb type weighted value of the COBOL programming language listed in the Figure 4.11. Each verb type and modifier verb type can be assigned a numerical value in order to determine its programming complexity. It is an indicator of the likelihood that the particular statement will produce error output or give unexpected results.

Verb	Weighted Value	Modifier Verb	Weighted Value
ACCEPT	3	None	0
ADD	3	ON SIZE	1
CALL	5	ON OVERFLOW	1
CANCEL	2	None	0
CLOSE	3	None	0
COMPUTE	3	ON SIZE	1
CONTINUE	1	None	0
DELETE	2	INVALID KEY	1
DISABLE	4	None	0
DISPLAY	3	None	0
DIVIDE	4	ON SIZE	1
ENABLE	4	None	0
EVALUATE	7	WHEN	1
EXIT	1	None	0
EXIT PROGRAM	1	None	0
GENERATE	1	None	0
GO TO	7	None	0
GO DEPENDING	7	DEPENDING	1
IF	5	ELSE	1
INITIALIZE	4	None	0
INITIATE	2	None	0
INSPECT	6	None	0
MERGE	3	OUTPUT	1
MOVE	3	None	0
MULTIPLY	4	ON SIZE	1
OPEN	3	None	0
PERFORM	5	Imperative	1
PURGE	2	None	0
READ	3	END/INVALID	1
RECEIVE	4	NO DATA	1
RELEASE	4	None	0
RETURN	4	AT END	1
REWRITE	3	INVALID	1
SEARCH	6	END/WHEN	1
SEARCH ALL	6	END/WHEN	1
SEND	4	None	0
SET	3	None	0
SORT	3	INPUT/OUTPUT	1
START	4	INVALID	1
STOP	1	None	0
STRING	6	ON OVERFLOW	1
SUBTRACT	3	ON SIZE	1
SUPPRESS	1	None	0
TERMINATE	1	None	0
UNSTRING	6	ON OVERFLOW	1
USE	8	None	0
WRITE	3	END/INVALID	1

Figure 4.11 Suggested COBOL Statement Type Complexity Measure Values.

Started Complexity Report of COBOL ANALYSER SYSTEM Rev. 19.1

Started Complexity Report of 1000-LOOP-BEGIN Paragraph
Started Line Number 40

Verb Type	Modifier Type	V. Count	V. Weight.	M. Count	M. Weight.	Weighted Value
MOVE	None	2	3	0	0	6
PERFORM	Imperative	1	5	0	1	5
WRITE	END/INVALID	2	3	0	1	6

Total No. of Weighting = 17
Total No. of Verbs = 5
Total No. of Modifiers = 0

Total No. of Weighted Value / Total No. of Verbs is
Average Verb Complexity Weighted Value in Paragraph = 3.40

Total No. of Weighted Value is
Paragraph Complexity Weighted Value = 17
Ended Line Number 48

Completed Complexity Report of 1000-LOOP-BEGIN Paragraph

Figure 4.12 One paragraph of simple program complexity measure report.

The Complexity Report of COBOL ANALYSER SYSTEM Rev. 19.1

Verb Type	Modifier Type	V. Count	V. Weight.	M. Count	M. Weight.	Weighted Value
ADD	ON SIZE	5	3	0	1	15
CLOSE	None	1	3	0	0	3
DIVIDE	ON SIZE	3	4	0	1	12
IF	ELSE	3	5	3	1	18
MOVE	None	25	3	0	0	75
OPEN	None	1	3	0	0	3
PERFORM	Imperative	13	5	1	1	66
READ	END/INVALID	1	3	1	1	4
STOP	None	1	1	0	0	1
WRITE	END/INVALID	7	3	0	1	21

Total No. of Weighting = 218
Total No. of Verbs = 60
Total No. of Modifiers = 5
Total No. of Sections = 0
Total No. of Paragraphs = 13

Average No. of Verbs per Module is
Total No. of Verbs / Total No. of Modules (Sections + Paragraphs)
Average No. of Verbs per Module is = 4.615

Average Module Complexity Value is
Total No. of Weighted Value / Total No. of Modules (Sections + Paragraphs)
Average Module Complexity Value = 16.769

Average Statement Complexity Weighted Value is
Total No. of Weighted Value / Total No. of Verbs
Average Statement Complexity Weighted Value = 3.633

Program Complexity Weighted Value is
Total No. of Weighted Value = 218

Completed Complexity Report of COBOL ANALYSER SYSTEM Rev. 19.1 ####

Figure 4.13 A simple program complexity measure report.

The complexity measure of a simple COBOL program is assessed for each individual section or paragraph. These complexity measures are then summed to give a value for the whole program. An example of a simple COBOL program complexity measure report, for a single paragraph (Figure 4.12) and a whole program (Figure 4.13) is shown.

Programmer productivity is the main criterion for measuring PC from an information theory view point. An empirical study by [CHE78] showed that a complex program takes more time to develop, and that the programmer productivity with respect to time is non-linear so that the time requirements for program development cannot be satisfactoraly predicted. The report also suggested that when the productivity index (number of programmer valid statements per programmer busy hour) dropped to a very low level (about 0.1 statement per programmer busy hour) for a very complex program, the programming problem should be practically considered not soluble.

4.5 DISCUSSION ON MUCAS WRITING

4.5.1 MUCAS WRITING OVERVIEW

We are interested here in the control flow analysis of the COBOL program, where the executable portion of a COBOL program is only the PROCEDURE DIVISION. Therefore, the other COBOL program's DIVISIONs will be ignored at this stage. Basically, MUCAS is organised in the following manner :

(a) The source COBOL program to be processed is read into memory as a stream of characters.
(b) The character stream is processed in character groups (symbols), symbols are either language keywords (CALL, PERFORM, etc.), or parameter names, or literals (numeric, alphanumeric, etc.), or values, or

operators, etc. A process of **parsing** processes the source text symbol by symbol then line by line.

(c) The formatted source code is generated along with the **parsing** and the information gathered is thus encoded into the ST (such as name list, sequential streams of information).

(d) At the end of the process the information of this source program is printed out.

4.5.2 INTERESTING FEATURES OF MUCAS WRITING

Before 1970s, most compiler writing has been done in assembler language. There are two important reasons for doing this [GLA69]. One, non-machine-language (or problem-oriented language) code was often intolerably inefficient for heavy-use programs such as language processors. Secondly, the problem-oriented languages often lack facilities needed by compiler writers. Machine language coding and checkout is often considerably slower than problem-oriented language coding. This increases the cost and complexity of compiler development. Advances in hardware technologies, i.e. the LSI and VLSI electronic circuit chips provides faster CPU time which compensate for these factors through ease of maintenance and flexibility of development. However, the categories of machine language and problem-oriented language are not precise enough.

The main criterion for choosing an implementation language is to minimise the effort and maximise the quality of the processor or preprocessor [MCK74]. Although, we have chosen a suitable implementation language (PL/I G), yet this PL/I G compiler often gives irrelevant error messages.

4.5.3 THE RELIABILITY AND EFFICIENCY OF MUCAS

The reliability of MUCAS is the result of having a simple well-structured design which gives a logical separation of techniques. This was only achieved by the use of modular programming and modular testing. Efficiency is always a design goal in computer programming. Most of the efficiency must be incorporated in the program during its planning and coding stages. To evaluate the efficiency, it is important to determine where the critical points are in the program. The computer itself can be used to provide the answer. The efficiency of MUCAS depends on four factors :

(1) Speed of compilation.
(2) Storage used during compilation.
(3) I/O handling and symbol table set up.
(4) Backing store usage during compilation.

MUCAS is constructed as a multi-pass translator because of the size of the overall design, the limitation of main storage, and to retain conceptual simplicity. Care must be taken to account for backing store (the computer's auxiliary storage which backs up internal storage, e.g., the reels of magnetic tape, disc or drum units which attached to a central processor [AND82] [MAY81]) transfers, since these make use of considerable CPU time and space.

4.5.4 THE FLEXIBILITY OF MUCAS

MUCAS has been written to be machine independent. This increases MUCAS's capacity to accommodate changes once the program is in production.

In general, the design criteria to make MUCAS machine-independent are as follows :

(1) It must compile the COBOL source program as defined in the language definition;
(2) It should achieve an acceptable level of error recovery.
(3) It must be reliable.
(4) It should be reasonably efficient in time and storage.
(5) It should be adaptable to a variety of I/O devices on any given machine.

After completion and acceptance of a software system, it is not unusual for the maintenance process to consume as much manpower as all previous developmental phases. Demands for maintenance may arise for a variety of reasons :

(1) Change of language specifications;
(2) Detection of errors;
(3) Detection of inefficiencies;
(4) New environment;
(5) New facilities.

4.6 DIAGNOSTIC FACILITIES AND IMPLEMENTATION TECHNIQUE

If an error condition occurs in the MUCAS run, a error indicator will be returned to the system routine indicating a syntax error such as a punctuation error, a misspelling, or an illegal construction within a source statement. The top-down method for syntactic analysis does not accurately determine where an error condition arises because it can not always identify what has actually caused the error. The only information given is that the position where the error arises is listed in the source statement. As the syntax control routine (FORMATTER_PROGRAM) steps through the tram-line graph, structures

within the statement are recognised and the NEXT_SYMBOL routine advances to the next symbol. When an error condition arises, the control routine (parser) has not yet advanced passed the symbol which has caused the error. When this condition occurs, an error is reported and a routine ERROR_RECOVERY is called to skip through the text until an end of statement occurs. Those formatting errors concerned with the incorrect use of identifiers and expressions are simply reported by a number message and nothing is done about error recovery. Statements are also checked for overlength numeric literals and a error message is printed if this condition arises. There is considerable room for the improvement of error diagnostics. Errors detected by MUCAS are listed individually within the text and a total number of errors is given at the end of run.

Producing good error diagnostics is a language implementation design goal. In a checkout system, it is helpful to obtain a maximum amount of information concerning errors from each computer run. The cost of obtaining this information is usually insignificant compared to the time wasted while the program is "run on". For time-sharing systems, where the user can rapidly ferret out errors, it is possible to design a system so that user himself defines his "fatality level".

During compilation, MUCAS prints the source text containing the errors and places five hash characters under each portion of the text which is in error. A "^" sign serves to locate the error precisely, making it easy to detect errors by a quick scan of the source listing.

4.6.1 **ERROR DIAGNOSTICS**

The MUCAS partial COBOL **Parser** module returns an error code to the system whenever an error is detected. This code indicates where the error occur during the parsing process. Syntax error diagnostics should include provision to detect and diagnose function errors, misspellings, illegal constructions, etc. in a program statement. All these may be grouped in the syntax error category. When such an error occurs, MUCAS reports not only its occurrence, but also the position where it occurred in the statement and a message is given to identify the symbol name used originally.

4.6.2 **COMPILATION TIME ERROR MESSAGES**

The error messages at compile time must be of sufficient clarity and conciseness so that a user does not have to consult a special list to obtain their meaning. One technique to achieve this is to split each error message into separate phrases and make up the message. This method would save space and allow for complex and sophisticated error messages.

Error messages can be printed out as they occur within a program or together after the source program is listed. Printing them within the text with possibly a line with hash characters (#) makes pinpointing the exact position of error easy with only a quick scan of the listing. On the other hand, printing the messages after the source listing avoids spoiling the structure of the program. In MUCAS's case, we prefered the error messages printed out as they occur within a program, since we found that was not too difficult to print out where the error has occurred in the source statement itself.

4.6.3 **TRACING AND SYMBOL TABLE DUMP**

Tracing is an important and easy to implement facility that can be added to the system by selecting the symbol trace option. MUCAS has a **symbol trace** facility which helps to locate errors and to determine the correctness of parsing and formatting. At the end of the formatting analysis stages, MUCAS can produce a symbolic dump of the ST.

4.6.4 **SKIP TO SPECIFIED SOURCE**

It is occasionally desirable to skip certain section of code. This may happen if an error is discovered in processing a source statement and it is desired to skip the remaining text until a logical break point is found [AHO77]. For example, if the true portion of an IF statement is in error, the COBOL parser would skip the false portion and move to the end of the IF statement (e.g. END IF reserved words). Specifically, this routine (which is called ERROR_RECOVERY) must also perform the function of repeatedly calling NEXT_SYMBOL and checking for the desired symbol or operator.

4.7 **POSSIBLE MODIFICATIONS OF MUCAS**

In future, Massey University will have VAX 750 computers so one can envisage a much more sophisticated version of MUCAS being implemented. By taking an entirely different approach, it may be possible to have an 'Interactive MUCAS' using the visual display unit. After having defined our partial COBOL syntax for MUCAS and then implemented it on a computer, it is inevitable that modifications to the language will be carried out by the future development. At present, MUCAS can only parse the COBOL syntax according to [COB81]. In general, MUCAS should parse any version of ANS COBOL syntax with only minor changes in the system. In the forseeable future, we may expect an analyser which can analyse any programming syntax when supplied with the correct programming syntax.

Careful and extensive documentation of previous processes are required because of the largely unpredictable demands of maintenance. Belady [BEL71] proposed a theory which predicted maintenance costs (which may be useful for MUCAS design). In MUCAS I implementation and design, MUCAS reformatted the COBOL program and provided a measure of COBOL PC. Other techniques which improve MUCAS capability will be discussed in future development of MUCAS. Some expected work should be incorporated in the future development (Chapter 5).

CHAPTER 5

DESIGN PROBLEMS, GUIDELINES AND POSSIBLE MUCAS DEVELOPMENTS

CHAPTER 5 DESIGN PROBLEMS, GUIDELINES AND POSSIBLE MUCAS DEVELOPMENTS

5.1 OUTSTANDING DESIGN PROBLEMS

It is always dangerous to anticipate the future, but we can at least suggest some potentially productive areas for research and development. The need for graphics is described in Chapter 4. The MUCAS processor not only performs a structured COBOL preprocessor function ([APP70], [IBM70] and [SOF70]), but serves as a tool to evaluate a COBOL PC measure. Although at this stage not implemented with graphics to show the program control flow constructs [PAI75] [PAI77] (since graphics features are normally machine dependent), there should be little difficulty in implementing graphics representations. A directed graph could be used to incorporate graphics features in MUCAS [VAN81] [STI74]. This would involve a representation of the complete syntactic structure of program. The approach of [VAN81] would be useful for MUCAS graphics implementation, as it displays the control flow of a program using a graphical representation.

Hoare [HOA78] said that although a computer professional can apply procedures effectively no matter what tool he uses, he may not be so available for the tedious manual drawing and redrawing of design charts. However, incorporating graphics features on MUCAS could save a lot of programming maintenance time for software documentation and increase software maintainers maintenance motivation.

The requirements of current computer technology tend toward larger and larger software systems. It is not unusual for todays systems to have 10,000 operational program modules, span 10 computers, and require a programming staff of over 100. Practical problems such as drawing 8m by 8m structure charts and integrating millions of instructions, make computer aided software design imperative. The APPLE Macintosh [APP83]

micro-computer has good graphics facilities. With some modification of its graphics software it would be suitable. However, currently, the response time for Macintosh is slightly slower than the users expectation.

Integrating a new software package (like MUCAS) into an existing computer organisation have to consider the individual system environment technical and economical situation [WIL81]. There is always the case that some computing organisation programming systems have their own documentation methods for security or privacy purposes.

5.1.1 FACTORS OF PROGRAMMER DEMOTIVATION IN COBOL PROGRAM MAINTENANCE

Programmers are always demotivated or burdened by design, but they are motivated by coding. Design configuration control appears to them as dictatorial and of secondary importance to their primary motivation, it is partly because of the systems analyst may be worried that the programmer knows too much about the system, and will write the program which would suit their view point instead of the systems analyst design. Others reasons such as for the security and privacy factor. That is why misleading communication always makes the systems analyst design conflict with the programmer coding. Beside this, incorrect design or erroreous modification would result during the system design phase, these could made the programmer have no confidence and be frustrated about the systems analyst design (no confidence or no guarantee that it is correct) [HAL75]. In order to avoid the discrepencies, a design team will help to improve the programmer motivation [WIL81]. For example, in TELECOMS (Telecommunications Authority of Singapore), the project team leader remains on the team at least through software subsystem design to assist in design configuration control and system optimisation decisions. After initial design, the team becomes the software design team and consists of system analysts and senior programmers. As the design progresses, the programmers become part of the design team. Therefore, everybody is involved in the design. This tends to increase motivation and decrease

resistance to design configuration control, and results in higher quality designs at each level of decomposition.

5.2 COBOL ENVIRONMENT SOFTWARE QUALITY GOALS

In order to facilitate the use of digital computers in critical control systems (e.g. commercial cash on-line retrieval system), COBOL software must be thoroughly validated. Software reliability is a measure of the confidence in the operational correctness of the software. Since the early 70's, most of the assumptions for several software reliability models have not validated. The theory of input domain of a program [BAS80] states that the size of the errors and the number, complexity and continuity of equivalence classes are characteristic of the program. The input domain method is applied to evaluate the software validation techniques and programming languages. It is indicated that some language constructs and documentation technique can improve the reliability of software, such a method would be able to improve the COBOL software quality.

The ARCTURUS system designed by Standish [STA81] is aimed at programming by many people, on large programs, with maintenance lifetimes of many years. The software quality goals as specified by Standish are general dimensions, such as reliability, correctness, efficiency, maintainability, responsiveness to user needs, timeliness of delivery, unit cost, and transferability, quality in parallel processing, fault-tolerance, self-diagnosis, meeting real-time contraints, commercial marketability and modifiability in the face of rapidly changing system requirements [ALF77]. In [FIS80], Fischer indicated the techniques for validation of software modifications made during the maintenance phase. He defined various retest strategies, developed techniques for their implementation, using graph theory and 0-1 integer programming, to examine the program performance characteristics, and discuss practical implementation considerations.

In view of previous standpoints, when considering software quality goals, no single goal is to be pursued at the expense of the others and no single optimisation criterion can be established [POD85]. In many large programming projects that extend over many years, there is likely to be personnel turnover. This implies that new project personnel will have to read and understand programs written by others. In such a context, training and documentation play key roles in the effectiveness of a programming organisation. MUCAS is designed to support effective documentation. The maintainer can use MUCAS to get information as to what modules do at concise, lucid, abstract levels of description. Therefore, many programming inconsistencies can be detected easily.

Scanning the literature, one can identify many tools and methodologies claiming to aid the software development process. These include such formalisms as HIPO diagrams [DEU82], developed by IBM, and specification systems like PSL/PSA developed at the university of Michigan [TEI77] [BRO82]. An analyst can use PSL/PSA to capture various forms of the early specification and preliminary design for a new system on a computer. These software systems provide insight into the completeness and consistency of the early stage of system development. However, information gained in the early specification process is not linked to the subsequent design, or to the delivered software system. Specifications and the preliminary design must be manually transformed into program design languages or flow charts which are manually translated into a programming language [KRU84].

Code reading has been the most reliable method for debugging programs. Using MUCAS to reformat COBOL coding aids the programmer in analysing the individual module. Code conversion from one computer to another requires the programmer to verify that the program will function the same way on each machine. Therefore, MUCAS reformatted only the syntax error-free COBOL program to avoid unnecessary error.

5.3 INCORPORATE GRAPHIC FACILITY INTO MUCAS SYSTEM

The Hierarchical Graph Language (HGL) modeling technique described in [BAR80] may be used as a tool to evaluate the structure of a COBOL program. But care must be taken into account when incorporating a graphic facility into the MUCAS system. As reported by [GNA81], a lot of computer systems currently using methods of automatically presenting information are largely limited to variations of tabular form. Mathematical models are central to human understanding of natural phenomena, and computer simulation is a powerful aid in the evaluation of these models. However, such simulations generally produce a great deal of data, which the analyst must analyse [MOR81]. Reading a sequence of lines and understanding their important is a tedious job, reminding people of the old adage, "A picture is worth a thousand words". As a result, efforts are now being directed toward presenting data graphically. Unfortunately, experience with interactive graphics and database tools have revealed factors : cost and response time [TEI77]. Response time has proven to be a problem when an interactive graphics system is implemented on a large mainframe computer. Therefore, careful consideration must be taken of the existing hardware device in the computing environment. Normally, a remote terminal must access the computer over telephone lines. Where the line speed is 1200 baud (maximum number of bits transmitted per second on a communication line) to 4800 baud, the picture generated as a system's presentation language can require several seconds, or even minutes to draw at these speeds. Current computer graphics systems are inadequate because (1) they required too much effort to learn, (2) they are restricted to displaying data in the traditional formats (like bar graphs etc.), (3) they discourage experimentation with too much work required to set up each display, or (4) they require knowledge of graphics algorithms and hardware [MOR81]. Therefore, we proposed that an international standard graphic language system should be established, similar to the ANS COBOL standard committee. Even a country or organisation standard would be helpful to graphics programming. There are also various special constructs that are needed in graphics programming. The dissertation by [MAL81] develops specification techniques for graphics programming languages that

encompasses interactive control constructs. Important concepts associated with graphics programming, such as picture, graphical transformation, and hierarchy picture structure are expressed as graphical data types, allowing them to be specified using the algebraic specification technique. More detail of graphic algorithms and their application are referred to in [LAS82].

The development of tools and techniques for the design and production of software has been a process of continuous evolution, it is the result of demand for capabilities [SNO81] [HAN85]. For a COBOL environment tool like MUCAS, graphic output information will be an important factor for its success. The graphics could be used as :

(1) To show program control flow construct [ALL70].
(2) To convert well-structured parts of programs into structure charts or NS charts [PET81].
(3) To display histrograms of program, sections and paragraphs complexity distribution (probably using SPSS [NIE75] or SPSS X [SPS83]).
(4) To display program complexity statistics information such as mean, standard deviation and variance etc. (probably using MINITAB [MIN82]).

5.4 IMPROVEMENT OF MUCAS SYSTEM OBJECTIVES

The main future development objective of MUCAS is to attain the software reliability goal. Techniques that facilitate the design of reliable software are described by Schlichting [SCH82], who mentions two distinct phenomena which can cause execution of a program to deviate from its specifications. The first is the failure of the computing system on which the program is running. When this occurs, the system might not be capable of following the instructions specified by the program. Secondly, the program may be inconsistent with its

specifications, even on a failure-free computing system. A methodology is examined for designing programs that can cope with failures in the underlying system. Axiomatic program verification techniques are extended for use in developing provably correct programs as a fail-stop precessor (i.e. a processor with well-defined failure mode operating characteristics). MUCAS should improve by adopting the axiomatic program verification techniques and implementing graphic structured diagram output such as NS charts [PET81] (NS chart should be considered as one of the best structured chart because it could display more structured control flow information within the 24 X 80 characters terminal screen).

5.4.1 **PROVING COBOL PROGRAM CORRECTNESS**

In the real world situation, we could state the computer's game rule in one or two sentences, whereas in the practice of automatic computing, the program (the rules of the computational game) often requires many thousands, and sometimes apparently even millions of lines of text [BOY81].

For ease of understanding, programs are expressed in linguistic structures, and different linguistic structures require different patterns of reasoning for the justification of their usage. Intuitively people tend to prefer the ones they are most familiar with [POD85]. While programming we don't only consider different programs enbodying the same solution to a given problem; usually we have to invent the solution as well. Human inventiveness usually invents more than one solution, with the results for us to choose. When different solutions to the same problem can be shown to be correct by proofs, there could be some common arguments and program text. All should be treated as equally important to the problem to be solved. Different programs share codes, different proofs share arguments, different theories share subtheories, and different problems share aspects. The need for such sharing is characteristic of the design of anything big. The control of such sharing is at the heart of the

problem of "scaling up" and it is the challenge to the computing scientist or mathematician to invent the abstractions that will enable us to exert this control with sufficient precision.

The cost of "bug" in critical software is great. Consequently the problem of program correctness is being intensively investigated worldwide. There are many approaches to the problem, Boyer [BOY81] views programming as a mathematical activity. A programmer ought to be able to prove in the mathematical sense that his programs are correct. Programming requires of the programmer the same kinds of precise thinking, creative leaps, and attention to detail normally required of the mathematician, who employs a set of powerful formal and informal tools.

The view of programming as a mathematical activity can be contrasted with the image of the programmer as the tinker, who produces machines that work without knowing precisely what they do, by changing his programs until they no longer fail under certain tests. Successful tests increase one's confidence in the program's correctness. The thesis [GOU81] includes a number of practical testing methods such as mutation analysis which generate tests from specifications and programs to confirm the correctness of a program. [BOY81] Chapter 2 to Chapter 5 contains various mathematical proofs on program correctness, possibly eventually useful for MUCAS future development in proving COBOL program correctness. Mili [MIL81] developed and investigated a methodology for program design and verification. The design part is described by a stepwise refinement process (which we used in our program design). The verification part is a set of induction rules. This methodology produces programs with a self-checking property which is weaker than proving correctness. The AVID (Aid Verification through the techniques of Interactive program Development) system [KRA81] provides a syntax-directed editing environment for the development by stepwise refinement of programs and proofs in the PL/CV2 programming logic. This is another step in providing methods and software tools for developing correct programs. An ATTAIN BLOCK

program construct was built, which allows the independent verification of different refinement levels, and guarantees that the refinement in a top-down development actually implements its high-level specification. This system supports the interactive display of logical dependency in proofs, and demonstrates the feasibility of a system to support and enforce the development by stepwise refinement of provable correct programs.

Regarding produce reliable program and proving program correctness, Dijkstra [BOY81] suggested mathematics as the right candidate, he mentioned that mathematical characteristics applicable to computing are generality, precision and trustworthiness. Generality is applicable to a very large, often unlimited number of cases. Almost all computer applications are general purpose, and have to have the characteristic of precision. To be worthwhile, almost all computer applications must be trustworthy. For example, in banking systems, applications software have to deal with different transactions and customers (generality), any amount from any account to any other account (precision), and it must be able to correctly keep track of the flow of money (trustworthiness).

In systems design and development, programming reliability must be an activity of an undeniably mathematical nature. By an unfortunate accident of history, however, programming became an industrial activity in the United States when the American manager was extremely fearful of relying on the education and the intelligence of his company's employees [BOY81]. Management tried to organise the industrial programming task in such a way that each individual programmer would need to think as little as possible. This has been a sad mistake. But the more management, with this conception of programming, failed to meet its deadlines and to achieve reasonable quality standards, the more has been invested in these ill-directed efforts. And now we have reached the paradoxical situation that, the intrinsic difficulty of the programming task has become worse each year. The recognition of the difficulty has become both politically and socially, more and more

unpalatable. Under these circumstances, there is a need for programs in informatics in which programs and programming are considered to be worthy. Research and education in computer sciences will be more concerned with the mathematicians of the future.

Eventually, MUCAS development should provide information which could be used to build a certain capability in proving program correctness.

5.4.2 **COMPLEXITY MEASURE ANALYSIS**

PC is a measure of our ability to understanding programs. It includes program readability. From the programmer's point of view, the more complex a program, the more difficult it is to understand what the program does [BAK80] [MCC78]. PC is also related to the difficulty of the programming problem and the size of the program. It can be considered as a function of the number of possible paths in the program and the difficulty of determining the path for an arbitrary set of input data [MCC76a]. For example, a program without alternative paths has no complexity by this measure -- but if it uses complicated instructions and data structures it may still be difficult to understand.

There have been many attempts to quantify complexity measures of computer programs e.g. using graph theory [HEC72] [FIS80] or software science ([BAK79a], [BAK80] and [ZOL80]). The most well-known methods are McCabe's cyclomatic complexity ([MCC76a], [HAN78] and [ZOL80]), Halstead's software science effort ([BOH75], [HAL77] and [ZOL80]), and Woodward complexity measure based on the number of crossings, or "knots," of arcs in a linearisation of the flowgraph ([BAK77], [WOO79] and [ZOL80]). McCabe's cyclomatic complexity is on firm analytical ground and adequately quantifies control flow complexity with the exception of the linearisation problem [BAK80]. The investigation by Baker [BAK80] pointed out each method's weaknesses and suggested that a combination of these basic measures should do better.

The basic program design approaches are the top-down approach ([HIE74], [GLA82] and [PET81]) and the bottom-up approach [DEU82]. Both approaches are compatible with the SP methodology. The top-down approach follows the rules of refinement process, and the bottom-up approach proceeds as the concatenation process.

One object of MUCAS is to measure COBOL PC. If we give each branching statement a measure of **1** and other statements a measure of **0**, we essentially get the cyclomatic number. If we weight every statement by its average number of tokens, we get the software science measure. More accurate software science measures can be obtained from the detailed token analysis performed by MUCAS. These enhancements have not been made at this stage but could be implemented quite easily.

Since the 1960s, the traditional programming approach for combatting complexity has been modularisation - dividing the program into independent units (called modules) which are allowed to interact in limited ways. The modularisation philosophy is that constructing the program as a set of conceptually and operationally independent modules will improve program readability and simplify the programming task [MCC78].

SP techniques was applied in 1970s where it is intent to reduce PC in order to increase program readability in the following ways :

(1) Reduce the number of program paths by imposing a simple module invocation system.
(2) Clarify program paths by requiring the program control flow return to the invoking module.
(3) Limit program path patterns by restricting the set of allowable control structures.

There are innumerable modularisation schemes for any program even when the SP restrictions are applied. The success of a particular modularisation scheme in reducing complexity is affected by the manner and the degree to which program modules interact. Program module interaction is classified as explicit interaction and implicit interaction. Explicit interaction occurs when program control is passed from one module to another. Implicit interaction occurs when two or more modules communicate by referencing and/or modifying the value of a control variable.

Program understanding became an important issue when program maintenance became a serious and costly problem ([DEU82], [POD85], [TER83] and [LIE78]). Usually, when a program was used for a period of time, it became necessary to make modifications; either because program errors which were not discovered during testing became apparent or because the program specifications were changed. Maintaining a program that could not be easily understood was a difficult, time consuming task. Often it was easier to write a new program rather than to modify an existing program [LIE83]. SP represents a change in attitude toward programming. Today, programming is considered a complex task viewed more as a discipline and less as an individual art [MCC78]. The SP methodology has defined the characteristics of a good program, but does not offer enough guidance in how to develop such a program.

5.4.2.1 **COBOL PROGRAM STRUCTURAL ISSUES**

Technically, a program is a sequence of instructions as written by a programmer [SPI77]. Syntax, semantics, and pragmatics are three aspects of the study of computer languages. Syntax deals with the structure of statements in a language. Semantics is concerned with their meaning. Pragmatics includes everything else dealing with the language, especially its usefulness and practicality [FRE75]. To solve the existing worldwide COBOL maintenance problem and produce well structured new COBOL programs, more concern should be shown to the

syntax and pragmatics of COBOL program.

Two of the most important underlying structural issues in COBOL are information binding and control structures [FRE75]. Information binding is the binding of different pieces of information together. Control structures involve the execution units of a language. There are four important execution units of program text : programs, modules, statements, and expressions. We are interested in the logical control flow between different units of execution, rather than the actual language statements for specifying the flow.

In MUCAS, a COBOL program is considered as the most global unit of execution. It has no linguistic bonds to other programs and does not depend in a direct manner on the execution of other programs; whereas on a system level, there may be flow of control between program processes.

The next lower level is modules (sections or paragraphs). A module is a unit of text which contains linguistic bonds to other pieces of code. The argument transmission in COBOL CALL statements to subprograms is transmission by reference. i.e. COBOL subprograms meet the definition of a module, and can have multiple entry points [MYE76]. A performed paragraph (or section) is an internal procedure. No arguments can be passed to a perform paragraph. It is not recommended that performed paragraphs be used as substitutes for modules in composite design, because module interfaces are not explicitly identified in code (i.e. all performed paragraphs become common coupled). If performed paragraphs are used as substitutes for modules (in MUCAS COBOL program analyser), then all data within a program is global data. Therefore, we should implement a structured design method, which makes the COBOL program PROCEDURE DIVISION's section and paragraph meet the definition of the explicit module. Then we can reduce the program complexity in term of data flow [ACK82].

The third level of execution unit is the statement. A statement is totally dependent upon the lexical or linguistic context. Normally, the more complicated statement will have greater possibilities to cause error. Because the average programmer's programming capability is a function of the number of years of programming experience, the next programmer who maintains the programmer software programs may misunderstand the previous programmer's intention.

Finally, the lowest level of execution unit is expressions. The simplest example is a single variable (identifier). An expression is the most fundamental flow of control. Identifiers in the performed paragraph refer to variables in the DATA DIVISION of the module.

There are a number of different types of control flow. The most common and generally supported in COBOL programming languages are:

(a) Sequential.
(b) Simple branch.
(c) Conditional branch.
(d) Loop.
(e) Subroutine jump.

Sequential control means the execution of one statement after another as written. For simple branch, the flow of control is to branch unconditionally to a section or paragraph within the program. Conditional branches involve the evaluation or testing of some expression and the subsequent execution of one of two alternative instructions, or paths of control flow. In a loop, the flow of control is sequential through a set of statements, but execution of the set of statements is repeated a given number of times, or until one or more specified conditions is satisfied. The subroutine call or return jump performs a transfer of control to some location, yet retaining the address of the next instruction location if sequential flow of control has been maintained. This saved address is then available to effect a

jump back to the place where sequential processing was interrupted.

5.4.2.2 A METHOD TO DEFINE A WELL-STRUCTURED PROGRAM

SP is a software methodology that provides consistency and form to the program. It was developed to improve the art of programming to ensure that management goals such as meeting programming schedules [POD85], producing reliable programs that fulfilled specification requirements, improving program readability [BEL79], simplifying program maintenance and reducing PC. SP proposed to achieve these by standard programming restrictions upon the program control structure, modularisation, and documentation. Based on the McClure [MCC78] definition (with some modification to it), a well-structured program should have two respects : conceptually and structurally. Conceptually, a program is divided into a set of tasks arranged in a definitional hierarchy. Each level in the hierarchy further defines the tasks residing one level above. The task in the successive levels of hierarchy is defined in more and more detail. Structurally, a program is divided into a set of hierarchically ordered modules. The set of program tasks form the conceptual base from which program modules are constructed. Depending upon its implementation complexity, a task is implemented as a set of instructions within a module, an entire module, or a hierarchy of modules. The conceptual relationship among program modules is determined by the invocation relationship existing between two modules. The rules for module construction are as follows :

(a) A module has a unique entry point and a unique exit point.
(b) A module is a self-contained structure.
    (1) A module performs a unique task in the program.
    (2) A module is bounded by its entry point and its exit point.
(c) Legal control structures are restricted to the following set :
    (1) Concatenation.
    (2) Selection.
    (3) Repetition.
    (4) Branch (restricted use only).

Branches should be restricted where possible to, for example, loop exits.

The use of program variables should be made explicit. Understanding a program requires understanding how its variables are used. A variable is represented by a name that may be assigned one or more values during program execution. A control variable is used to direct program execution flow. Control variables are important in understanding a program, because the current value of a control variable may determine what program instruction is executed next. Understanding how a variable is used can determine the value of a variable at a particular point in the program.

To improve program understanding with respect to program variables two simple guidelines are suggested :

(a) Each program variable should have only one purpose in the program. This may result in defining more program variables, but the increased space requirements will be outweighed by the increased program clarity.

(b) The set of program modules that access the variable should be apparent and limited. A program variable should be accessed in as few modules as possible. If it is not, its value becomes more difficult to determine at some point in the program.

When a recoverable error occurs in the program, error processing follows normal control flow and the module invocation rules are preserved. In the case of unrecoverable error, control is not returned to the invoking module. A branch from the module in which the error is detected, to the entry point of the error function module (whose function is to process the error and abort the run) will be executed.

Documentation is necessary to explain the task performed by each program module, its data requirements, and its invocation relationship to other program modules. This documentation is invaluable to the programmer during program coding, testing, and maintenance. But it gives the programmer an opportunity to rethink the purpose of each module as it is coded. During testing, it serves as a review of the module. During maintenance, documentation clarifies the original programmer's intention. The additional time required for documentation is well spent whenever a program is to be used for a period of time. The following is the suggested minimal amount of documentation :

(a) The name of each module should suggest its task.
(b) The source code for each program module should be preceded by the following documentation :
    (1) An explanation of the task performed by the module.
    (2) A list of the modules in the program that invoke this module.
    (3) A list of the modules in the program that this module invokes.
    (4) A list of the control variables that this module modifies.
(c) The name of each program variable should suggest its purpose and where it is used in the program.

5.4.3 COBOL PROGRAM COMPLEXITY MEASURE

McClure [MCC78] said that a good modularisation scheme would minimise PC. A modularisation scheme that evenly distributes complexity among program modules is even better. To minimise complexity and to be well structured will further improve program readability and maintainability.

In order to measure the suitability of a modularisation scheme, a way of measuring module complexity is required. Counting techniques, such as the number of lines of code per module, the number of control

variables accessed per module, or the number of modules invoked per module, are inadequate means for measuring program complexity, because they cannot measure the complexity aspects of both types of module interaction (explicit and implicit). In quantitatively measuring the complexity module interaction two measures are included : Control Variable Complexity (CVC) and Module Complexity (MC) [MCC78]. CVC is a complexity measure of a control variable on module interaction, indicating the difficulty of understanding how a control variable affects module invocation and how a control variable is used in implicit module interaction. MC is a measure of the invocation complexity of a module relative to the other modules in the program. It shows the difficulty of determining how program control is passed among modules.

The following are three steps in measuring the suitability of a modularisation scheme (a combined method of complexity measure which may applicable to some high-level programming languages) :

(a) Each control variable is assigned a numeric value C_i.
(b) Each type of statement is assigned a numeric value T_i.
(c) Each module is assigned a numeric value M_i.
Modularisation scheme complexity is the sum of the module complexity values $\sum M_i$.

Since PC value is the summation of M_i, i.e.

Program Complexity value $PC = \sum M_i$
Where $1 \leq i \leq n$ and
n is the number of modules in the program.

Complexity analysis is complete when the programmer is satisfied that the modularisation scheme complexity has been adequately minimised and evenly distributed among the modules.

5.4.3.1 CONTROL VARIABLE COMPLEXITY MEASURE

The complexity of a control variable C_i is a function of the set of modules that access the control variable and the invocation relationship among these modules. In general, the more modules that access a control variable, the more difficult it becomes to identify and understand all explicit and implicit module interaction via this control variable.

Two modules interact implicitly whenever they access the same control variable. A module is said to access a control variable if the value of the control variable is referenced and/or changed in the module. One way of reducing the complexity value of a control variable is to reduce the number of modules that access it. The control variables of a program can be compared by means of their complexity values. Control variables with greater complexity value contribute more complexity to the design than control variables with lesser complexity values. Control variable complexity can be used as a criterion in selecting the best program. If two programs perform the same task, the program where control variables contribute the least to complexity can be selected as the better of the two, as such a program will be easier to understand and easier to maintain.

5.4.3.2 MODULE COMPLEXITY MEASURE

The Module Complexity (MC) M_i is evaluated by the following :

(a) The control variables referenced in the module invocation (C_j).
(b) The control structures used for module invocation (S_K) [concatenation, selection and repetition].
(c) The commonality of a module (N_1) [i.e., the number of fathers for a module].
(d) The branches from a module to an another module (B_m) [GO TO and GO TO DEPENDING ON statement] [KNU74].
(e) The statements used in the module (T_n). [ACCEPT, ADD etc.].

Therefore, the formula for Module Complexity value will be :

$$M_i = \sum C_j + \sum S_k + \sum N_l + \sum B_m + \sum T_n$$

Where i, j, k, l, m and n are dummy variables.

C_j is the control variable complexity of MC, and S_k is the control structures that direct module invocation. Module complexity increases if a module is invoked within a repetition structure (i.e. loop), rather than within a selection or concatenation structure. The N_l of MC accounts for the difficulty of understanding as to when and how the module is invoked. When a change is made to a common module, the possible effect of this change on each father must be carefully examined. B_m indicates that MC increases when a branch from the module is made. In this case, the father-son invocation relationship is overridden.

For complexity evaluation, the complexity value of each module defined in the program coding must first of all be examined. The average module complexity value for the program is then computed. If no module complexity value deviates greatly from this average (depending on the programmer's intuitive feel for a reasonable deviation such as mean, standard deviation or etc), then the distribution of complexity is acceptable. When each module complexity has been judged acceptable, then the program complexity value is considered to be acceptable. If the complexity distribution is not satisfactory, then some guidelines to minimise and redistribute the modularisation scheme must be carried out as follows :

(a) Divide the program module into three categories : highest, medium and lowest complexity valued modules.
(b) Examine the highest complexity value category to :
    (1) Look for alternative modularisation schemes which redefine the module as a subhierarchy of modules, each module having a lower complexity value.

(2) Reduce high complexity value control variables by reducing their use in various modules.

(c) Examine the lowest complexity value category so that :

(1) If the module is not common (i.e., is invoked by only one module in the program) and its father has a relatively low complexity value, then absorb it into its father.

(2) If the module is not common and has sons that have relatively high complexity value, then incorporate some tasks performed by the son into this module.

(d) Whenever possible, choose modularisation scheme modifications that will reduce control variable complexity values, as such reductions will reduce module complexity values and simplify the modularisation scheme.

Complexity is important because it is an indicator of program readability. The more complex a program, the more difficult it is to understand the program [POD85]. The problem of program complexity affects program testing and program maintainability as well as program coding. If the complexity problem can be dealt with as part of the design phase, then the task of design implementation may be simplified. Three sequential steps to examine the complexity measure during program design are :

(1) Determine the complexity value of each control variable defined in the design.

(2) Determine the complexity value of each module defined in the design.

(3) Evaluate the complexity of the design.

Using the complexity measure formula :

$$\text{Module Complexity value } M_i = \sum C_j + \sum S_k + \sum N_l + \sum B_m + \sum T_n$$

$$\text{Program Complexity value } PC = \sum M_i$$

Where $1 \leq i \leq p$ and

p is the number of modules in the program and

i, j, k, l, m, n and p are dummy variables.

CVC values and MC values can be used as criteria in selecting the best program from a set of programs that perform the same task. This is a measure of the suitability of the modularisation scheme proposed by the design. The criteria for a good modularisation scheme are : Minimisation of complexity in each module and an even distribution of complexity among the modules in the design. The complexity values give the programmer an intuitive feel for what makes a design good. A calculation of complexity measure example may refer to [MCC78] Chapter 5. The complexity measures described here could relatively easily be built into a future version of MUCAS.

5.5 SOME COBOL PROGRAM MAINTENANCE PROBLEMS AND CODING GUIDELINES

Program testing and program verification have long been considered to be unallied, competing approaches to demonstrating program reliability. Neither approach is infallible, yet used in conjunction they can complement one another well. Richardson, Debra Jane [RIC81] introduces the partition analysis method by integrating testing and verification in an attempt to counterbalance the weaknesses of each approach by the strengths of the other.

Because of the labour intensive nature of software production, there is an urgent need to understand and measure the underlying causes of complexity so that software cost may be reduced [POD85]. To increase understanding, models of modularity factors have been developed which have been used to better comprehend both program construction and comprehension. Two main models developed by Woodfield [WOO80] are the logical module model and the interconnection model. The first reflects the logical segmentation of large physical modules. The second model enables the estimation of the complexity associated with model coupling. The complexity measure was able to account for 80% of the variance in the data's complexity. The inclusion of all factors reduced the average relative program error from 150% to 1% [WOO80]. The logical module model enables us to realise why the monolithic program from the comprehension experiment was understood. The

interconnection model showed the perfect structure of the abstract data type modularisation version, and indicated why module connections can cause functionally modularised programs to be complex. Although the programs used as data were quite small, the basic factors affecting these programs should also be valid for larger programs. Therefore, we should consider the logical module model and the interconnection model in MJCAS COBOL program complexity measure analysis. This could increase the contribution to COBOL program maintenance facilities.

5.5.1 **CODING GUIDELINES FOR COBOL PROGRAM**

By understanding the program complexity measure, we should have a set of coding guidelines for producing a good COBOL program. Based on Kerr [KER79], Lim [LIM80], Robinson [ROB76], McClure [MCC78], and Stern, Nancy and Stern, Robert A. [STE79], the coding standard are as follows :

(a) Modularisation.
  (1) A module is coded as a COBOL section or subprogram.
  (2) A section performs a certain function. It is composed of a section-header and at least two paragraphs. The section-header is the module entry-point. The entry-paragraph contains all the executable code for the section. The exit-paragraph is the module exit-point.
  (3) A module has one unique entry-point and one unique exit-point. Program control must enter at its entry-point and leave at its exit-point.
  (4) The exit-paragraph will be the next paragraph that physically follows the entry-paragraph in the source code. The exit-paragraph will physically precede the next section-header in the source code.
  (5) A module is invoked only by means of a PERFORM statement.
  (6) The order of modules in the source listing follows their execution order (father is followed by its sons according to their invocation order).

(b) Statements.

(1) ALTER statement should not be used.

(2) At most one COBOL statement is coded per line. All continuation lines for a statement are indented two characters. For IF statements, the ELSE must be coded and lined up with its corresponding IF.

(3) Commas and semicolons should not be used as delimiters. Only mandatory periods are used.

(4) Section-headers or paragraph names should be coded on a separate line.

(5) The statement verb and verb modifier must be on separate lines. The verb modifier is indented two characters. Further sub-verb modifiers are indented two characters accordingly.

(c) Control Structures.

(1) Selection is implemented with the IF statement. The ELSE clause is always required. In a nested IF, four levels of nesting are allowed at the most.

(2) Repetition is implemented with the PERFORM statement. Only a section can be repeatedly executed. Only one section can be executed with one PERFORM statement.

(3) The branch is implemented with the GO TO statement. A branch from within a section can be made only to the exit-point of that section. The one exception is in the case of an unrecoverable error. In this case, a branch can be made to the entry-point of another section whose function is to abort the run.

(d) Organisation of the WORKING-STORAGE SECTION.

(1) Each data item should have only one purpose in the program and its name should suggest its purpose.

(2) WORKING-STORAGE SECTION data items are grouped together as records in the order that the main module invokes.

(3) A data item is placed in a record belonging to the module that use this data item.

(4) The record name is the module name prefixed by WS- to indicate that it is declared in WORKING-STORAGE SECTION.

(5) Record descriptions are disjoint and are ordered in the source listing according to the same ordering rules as for

their corresponding sections.

(e) Documentation.

(1) A COBOL program should be properly documented.

(2) The documentation preceding each module is as follows :

(i) An explanation of what the module does (one or two sentences should suffice).

(ii) A list of the variables that the module can modify.

(iii) A list of the modules that this module invokes.

(iv) A list of the modules that invoke this module.

(3) A program hierarchy flow diagram (stating only the section names and paragraph names) should accompany the program in REMARKS paragraph of IDENTIFICATION DIVISION.

However, simply following a set of coding conventions to produce a structured program is not enough to combat the complexity problem [HAR71], because the sources of complexity are not well understood and there are no means, other than intuition, for measuring complexity. To solve the complexity problem, we impose quantitative measures for the programmer to control complexity in the program as it is being developed. Using this approach will result in programs that are easier to understand, to test, and to maintain.

5.6 **OTHER INFLUENCES ON THE MUCAS DESIGN**

Minor changes to the COBOL parser of MUCAS in order to parse different versions of COBOL program should be very useful for the future developments of MUCAS. The MUCAS syntactic program transformation techniques imply that some readability factors were not considered. Thus even though the reformatted versions are more readable, there is room for improvement. From both theoretical and empirical points of view, any significant improvements will be the result of the use of semantic information. Another program transformation technique is to transform the program executable process using the control flow graph ([BAK77], [HAR65] and [KAR60]) of the source program and the use of structured control structures. This is called

restructuring [OTH81]. The program restructuring method was used to reduce the cost of program execution by improving the locality of the program's reference behavior [LAN82]. Effective restructuring methods are sampling, granularity, localities and critical principles. Critical principle is found to be more effective than the others.

The prime features of MUCAS developments should have good error diagnostics with meaningful error messages at compile time, and also the ability to collect and record the COBOL program complexity reports of the generated formatted program. Given these guidelines, the development of the COBOL programming system could proceed.

CHAPTER 6

DISCUSSION AND CONCLUSION

CHAPTER 6 DISCUSSION AND CONCLUSION

6.1 CAUSES OF PROBLEMS IN COMPUTER PROGRAMMING

6.1.1 THE CHANGES IN INPUT MEDIUM AND COMPUTER PROGRAMMING

The input medium to the computer in the 1940s was mainly paper tape. In the 1950s punch cards, in the 1960s teletype terminal, in the 1970s video terminal, and in the late 1970s, some organisations were using mini-computers as buffer input to their mainframe. In the 1980s, some bigger computing organisations started international telecommunication application. For example, the CMOC project (design 1977 - 1980, implementation 1980 - 1983) carried out by Singapore TELECOMS and the Japanese government, made use of the undersea cable to transmit information between two countries, using some of the big mainframe computer systems as the input interface. As time goes on, we can see that the information flow has increased in volume and complexity. Correspondingly, the input interface has grown in size.

Whenever the hardware changed the software had to be changed accordingly [MOR85]. With programming development prior to the 1950s, input to the program development process consisted of a string of perhaps millions of zero and one bits which instructed a set of electronic devices on how to perform as a system. The use of assemblers and assembly languages in the 1950s removed the need to write bit strings such as 01010101 [DEN80a] [MOR85]. Higher-level languages and their compilers in the 1960s allowed the programmer to work without knowledge of the machine language [SAM69] [ACM78]. Programmers worked with higher level virtual machines defined by the language they used. Structured systems were produced during the 1970s (SP revolution period), and ideas such as SP, code reading, and program development libraries arose [VER85a]. By this time, we are

considered as in the requirement analysis phase ([ALF77], [ALF85], [BOE81], [BOR85] and [BEL77]). There are numerous fourth-generation languages which have been developed mainly to serve wide a range of users (customers) requirements - thousands of them in different business and other fields ([RUL80a], [FOX82] and [GRA85]). The majority of the software product efforts are based on requirements analysis rather than programming ([GRA85], [NAM81] and [DAV80]). Fifth-generation computer systems are in development ([TRE82], [MYE85], [MOT82] and [RUL80a]).

6.1.2 COMPUTER PROGRAMMING DIFFICULTIES

Many of the basic results of computability theory were discovered by mathematical logicians prior to the development of the first stored-program computer [BOY81]. Computer science seeks to provide a scientific basis for the study of information processing, the solution of problems by algorithms, and the design and programming of computers [GON84]. The last 40 years have seen increasing sophistication in the science, in microelectronics which have made machines of staggering complexity economically feasible, in advances in programming methodology which allow immense programs to be designed with increasing speed and reduced error, and in the development of mathematical techniques to allow the rigorous specification of program, process, and machine [KFO82].

The methodology of SP has facilitated rapid progress in many areas of theoretical computer science [DEE85]. Structured programs are generally easier to debug, test, prove and analyse [WOL83] [MCC78]. The development of these achievements into commercially viable applications and products has been slower than expected [HOL82]. The primary reason is that most of the programs currently in use are unstructured and theories based on assumed structure are not relevant to them [OTH81].

To solve this problem, we have proposed COBOL coding guidelines to produce structured COBOL program, and used the syntactic technique (parsing) to reformat the PROCEDURE DIVISION of the unstructured COBOL programs. i.e., to exhibit the existing COBOL program structure in a standard way.

Control flow is the most primitive and fundamental form of computation for derivation of measures of the psychological complexity of programs [HOL82]. Therefore, MUCAS is designed to reformat the program in the order of section, paragraph, sentence and statement with predicates that influence the program logic, dealing with the control flow constructs of the COBOL program. The associating statements in a COBOL program and its predicates that influence their execution are independent of the program text layout (program structure) [TRI79], and provide a characterisation of the statements which correspond to the property of nestedness in the control flow of structure programs.

6.1.3 **EXPERIENCE FROM THE REVISION OF ANS COBOL-80**

Fiorello [FIO84] describes a survey of COBOL-related effects on the U.S. federal ADP community, and the estimated costs and benefits to the federal government that would result from adoption of the proposed revision of ANS COBOL-80 as a FIPS (Federal Information Processing Standards). The new version of COBOL provide better programming facilities for SP constructs than older versions. Potential benefits of $90.2 million have been identified, stemming primarily from improved productivity in both the development and maintenance of COBOL programs. Estimated costs of $17.9 million have been identified, arising principally from the effort needed to convert old COBOL programs to the new specification, incompatible in some respects with the current one. Therefore, a potential area of use for MUCAS can be identified. Since MUCAS could reformat COBOL program and measure program complexity, it should potentially produce better productivity improvements than the implementation of a new version of COBOL. In view of this, the MUCAS design goals in research, in development and in economics are well

satisfied. The study by Fiorello also shows that the effect of revising the COBOL standard as proposed should not be dramatic, either for good or ill.

6.1.4 THE PROGRAM DESIGN INCOMPETENCE PROBLEM

Realistically, there must be more perfect function information binding before coding. However, this really represents only the tip of the traditional design iceberg [GLA69]. Much more serious problems are "why programs and information systems fail all too often after they are in production", and why many users are unhappy with existing levels of performance (e.g., turnaround or response time) ?

Beside the possibility of designer incompetence, there are several reasons why approximately one-third of the data processing budget is spent on reworking existing software [RIC80]. Much of this results from errors in the original systems design or sloppy implementation of a reasonably good design. More specifically, errors can usually be traced back to improper requirements specifications in one of the following areas :

(1) Output function - doesn't work as specified or perceived.
(2) Input formats - bad data not filtered out before entering the program.
(3) Logical process - legitimate data produces erroneous results.
(4) Stored data - needed historical results not saved or difficult to retrieve.
(5) Access controls - users illegally modify stored data.
(6) Testing specifications - logic errors not uncovered during testing because of poor procedures.
(7) Documentation - an apparently simple modification causes catastrophic results elsewhere in the system.

There are other scenarios which could result from the existing problems of system design, but they are too voluminous to fully describe. However, the results of a system or program design activity are all too often characterised by the following:

(1) Documentation for users and maintainers is lacking or absent (e.g., many organisations feel that source code is good enough).
(2) An examination of existing programs reveal that the programs are filled with GOTO instructions, and vague. The resulting listing would look like a spaghetti bowl.
(3) Design projects are late in delivery, and users are generally not happy with what they get.
(4) Productivity of programmers is apparently low (approximately four to twelve lines of code per day); although this is certainly a weak measure of output value [RIC80] [RUL80a].
(5) Overall software development cost is spiraling upward to 60% -- 90% of the DP budget [ZOL80].

Obviously, no single approach or control tool will solve all of these software quality problems ([POD85], [ART84], [AMS76] and [BOE76a]). However, a tool such as MUCAS can develop better systems which avoid many of the problems summarised above. To summarise, there are four goals to resolve program design problems :

(1) Improve reader understanding of functional and program design techniques [PET81].
(2) Introduce concepts of structured design [MYE76] [MYE78] and inductive stepwise refinement [DRO85].
(3) Outline a procedure for structured system development procedures ([BAK72], [DIJ76], [MCC78] and [STE74]).
(4) Generally describe the design tool and its relationship to the above.

6.2 SOFTWARE MAINTENANCE PROBLEMS AND THE EFFECT OF MUCAS TYPE SOFTWARE APPLICATIONS

According to Lientz [LIE83], the turnover of staff is a major concern for many organisation. Traditionally, computer scientists seem to pay more interest to creation of new software system or new program. Occasionally, once the system is in the production environment, some of the designers may no longer be with the organisation. The "new product" may have a painful task to maintain its operation, or may just scratch it off entirely and start a new program or new system development cycle. Many computer professionals taking over a new system have to spend day and night, struggling for the target date in order to make the right modification [ROB76].

Programmers are writing programs or software in their systems development cycle. Once the program is completed, it is said to be in the maintenance phase [DEU82]. The maintenance surveys by Lientz [LIE83] summarise 26 potential maintenance factors. Five of these seen as most severe were :

(1) Quality of application software documentation.
(2) User demand for enhancements and extensions.
(3) Competing demands for software programmer personnel time.
(4) Difficulty in meeting scheduled commitments.
(5) Turnover in user organisations.

In the future, consideration must be given to maintenance of user-intensive concerns, as well as system issues. Application system maintenance will continue to be an area of concern but will become focused on substantial maintenance and enhancement, instead of on short-term activities such as fixes and patches.

In the 1940s, computer programming was in the object code or assembly programming level. The number of computer systems and software programs existing was limited to highly skilled personnel. In the 1950s, the second generation programming language created was FORTRAN [WEX84] [PHI77], which is primarily scientific application. Therefore, we started to have some software maintenance problems. Because the significant software systems (high level programming language) were only a few [SAM69] at that time, the maintenance of the software programs or systems was insignificant. In the 1960s, the commercial and industrial application languages such as COBOL made the software maintenance gradually become obvious. Because more and more software systems were created and complexity was increasing, with an increasing number of software specialists this gave rise to the need for the SP revolution of the 1970s.

SP solved part of the software design and maintenance problems. Those software systems implemented during that time are much easier to maintain compared to their 1960s counterparts.

It is a worldwide feature that as people come and go, numerous programs and software systems are constructed. The amount of software has increased tremendously. This was especially manifest after the 1970s (The SP revolution period). Bigger and bigger fourth generation languages were invented and implemented, e.g. LOTUS 1-2-3, SYMPHONY, MULTI PLAN, dBASE [ROD83], KNOWLEDGE MAN, PROLOG (used in Europe) and OPS5 (U.S.) [WEX84]. The 1980s is the requirement analysis phase [ALF85]. The 1990s is expected to be the intelligence Phase. Numerous software systems have been developed. Consumer use of computer systems is occurring as we approach the 1990s [MOR85]. We have stopped trying to teach the computer to solve our software problems. Rather we help it to learn to do so [WEX84]. This makes the unsolved software maintenance problem worse. In the foreseeable future fifth-generation computers will be invented. Supercomputers, VLSI processor architectures [HIG73], and integrated communications and computers provide us with four complementary views of future

computing [TRE84], yielding a composite image of the future computer system as highly decentralised at all levels with computers linked together in an integrated computer-communications network [MOR85]. At this stage, changes in the computing revolution will make the software maintenance problem even worse than ever. The software maintenance crisis is going to burst out in the software maintainers brain.

As I have worked for few years in TELECOMS (Telecommunications Authority of Singapore , where total number of staff is about 14,000, computing staff is about 400), I have experienced many cases of programming or non-programming staff spending day and night just to figure out how to modify, improve or enhance the software maintenance of their software system, ranging from the software systems which used COBOL, FORTRAN, MAPPER 1100, APL or others, to some machine dependent language like sort processor, query language processor (relational database), test file generator or some other assembly language writing software.

Numerous software has been implemented. In order to maintain these software systems, software control flow and data flow analysis constructs must be evaluated, and the software reliability goal attained. This is a very rigorious problem faced by the software maintainers (the software maintenance crisis). Now we are in the Information Technology period, computers are going to be part of the human being's daily life [BUR84]. Every computing company is competing in the information technology area. There are numerous mini-computers, micro-computers, software packages and mainframe computer systems being established. There are numerous software maintenance problems from all of these computer systems.

Singapore, New Zealand and other countries have set up a Information Services Centre, providing services to all the users worldwide. Burstein [BUR84] said that Sperry Univac, Hewlett-Packard, DEC, and IBM have already agreed to open software development facilities. A

Software Technology Center scheduled for occupancy in early 1986 will be an integral part of the Singapore Science Park now under construction, offering ready-to-use workspace for software developers. Will Singapore achieve its aim of becoming the software center of Asia in the 1990s and succeed in their goal of making a shift to a 21st century economy based on brain power and high technology ? Success will depend on how they face the software maintenance crisis. Any country that solves the software maintenance crisis problems correctly, and attains the software reliability goal, is going to win the race, win the customers, and win the software services. MUCAS is one of the software maintenance tools that aims toward this direction.

6.3 THE SUMMARY AND CONCLUSIONS

In the current revolution of program design, attention is increasingly focused on the human aspects of program development [HAN85] [DRU85]. Development, testing, and maintenance costs are now recognised as keys to reduced processing cost [POD85] [MOR85]. As a result, techniques such as SP and its associated coding standards are being utilised to provide more control over the programming effort [ELS77]. One objective of this thesis was to present an overview of the associated construction of source code for the most popular languages : COBOL, FORTRAN, and PL/I. Obviously, production of high-quality software is required. However, the issues presented here are important elements of the current trend toward increased standardisation. The increasing number of mini-computers and fourth-generation software packages make the computer generalised to everybody [ROD83]. But we have few significant advances in software maintenance. This problem has existed throughout computing history. Just like playing soccer, most of the players like to be striker, scorer, but not goalkeeper. Similarly in computing, most of the professionals would like to adopt this philosophy; produce new software programs or systems (striker job), but don't like to perform as a software maintainer (goalkeeper job), to prevent the system going down. This current situation is partly because of management staff pay more effort on software creators activities rather than the software maintainers job. In view of this,

the balance must be adjusted between new system creation and software maintenance. According to the statistics in Chapter 3 of this thesis, it should have the ratio of 30% of new system development and 70% of software maintenance for an average computer system.

Program maintenance is a costly, time-consuming job, this is especially true for COBOL programs [VER85a]. Any software tool or kit that can make the maintenance job easier is valuable. One potentially valuable tool is a system that transforms programs into equivalent, more readable versions. The reformatting or restructuring of programs will enhance readability. For the reason of simplicity and necessity to show control flow, we reformat only the program executable portion of source code (the PROCEDURE DIVISION).

The program readability definition is based on the concept of limited human capacity of short-term-memory and chunking, i.e. the process of forming higher level constructs from more elementary items of information. It is assumed that any factors which help the programmer in the chunking process aid readability [OTH81].

To evaluate the efficacy of the reformatting algorithms, the reformatting process must consider the control structures that frequently occur in the program we wish to improve, and the rules for the control structures advocated by adherents of structured programming.

MUCAS uses the concept of chunking to reformat COBOL programs. We chose COBOL because it is deficient in control structures, and there are a large number of poorly-written COBOL program being maintained. MUCAS represents a particular direction for software development. Indeed, the original objectives concerning the early exposure of system structure to designers coupled with the emergent emphases on information control may be said to have been achieved, if only to a

limited extent. However, the need to determine and utilise design evaluation techniques for COBOL is still very much a primary concern. Advances have and will continue to be made. It is likely that significant advances in this area must derive from a better appreciation of what it is being designed and how the design process is carried out. Information is captured at acceptable times and made available in acceptable forms. Trends can be perceived in systems development which will have considerable impact on the software systems industry. Programming techniques must, of necessity change. The requirements for programming support systems must necessarily change also.

APPENDIX A

COBOL RESERVED WORDS AND SYMBOLS

APPENDIX A : COBOL RESERVED WORDS AND SYMBOLS

The following list of COBOL reserved words are based on the manual of [COB81] Draft Proposed Revised X3.23 American National Standard Programming COBOL.

ACCEPT	BEFORE	CONFIGURATION
ACCESS	BLANK	CONTAINS
ADD	BLOCK	CONTENT
ADVANCING	BOTTOM	CONTINUE
AFTER	BY	CONTROL
ALL	CALL	CONTROLS
ALPHABET	CANCEL	CONVERSION
ALPHABETIC	CD	CONVERTING
ALPHABETIC-LOWER	CF	COPY
ALPHABETIC-UPPER	CH	CORR
ALPHANUMERIC	CHARACTER	CORRESPONDING
ALPHANUMERIC-EDITED	CHARACTERS	COUNT
ALSO	CLOSE	CURRENCY
ALTERNATE	CODE	DATA
AND	CODE-SET	DATE
ANY	COLLATING	DATE-COMPILED
ARE	COLUMN	DATE-WRITTEN
AREA	COMMA	DAY
AREAS	COMMON	DAY-OF-WEEK
ASCENDING	COMMUNICATION	DE
ASSIGN	COMP	DEBUG-CONTENTS
AT	COMPUTATIONAL	DEBUG-ITEM
AUTHOR	COMPUTE	DEBUG-LENGTH

DEBUG-NAME
DEBUG-NUMERIC-CONTENTS
DEBUG-SIZE
DEBUG-START
DEBUG-SUB
DEBUG-SUB-ITEM
DEBUG-SUB-N
DEBUG-SUB-NUM
DEBUGGING
DECIMAL-POINT
DECLARATIVES
DELETE
DELIMITED
DELIMITER
DEPENDING
DESCENDING
DESTINATION
DETAIL
DISABLE
DISPLAY
DIVIDE
DIVISION
DOWN
DUPLICATES
DYNAMIC
EGI
ELSE
EMI
ENABLE
END
END-ADD
END-CALL
END-COMPUTE
END-DELETE
END-DIVIDE
END-EVALUATE
END-IF
END-MULTIPLY
END-OF-PAGE
END-PERFORM
END-READ
END-RECEIVE
END-RETURN
END-REWRITE
END-SEARCH
END-START
END-STRING
END-SUBTRACT
END-UNSTRING
END-WRITE
ENVIRONMENT
EOP
EQUAL
ERROR
ESI
EVALUATE
EXCEPTION
EXIT
EXIT PROGRAM
EXTEND
EXTERNAL
FALSE
FD
FILE
FILE-CONTROL
FILLER
FINAL
FIRST
FOOTING
FOR
FROM
GENERATE
GIVING
GLOBAL
GO
GREATER
GROUP
HEADING
HIGH-VALUE
HIGH-VALUES
I-O
I-O-CONTROL
IDENTIFICATION
IF
IN
INDEX
INDEXED
INDICATE
INITIAL
INITIALIZE
INITIATE
INPUT
INPUT PROCEDURE
INPUT-OUTPUT
INSPECT
INSTALLATION
INTO
INVALID
IS

JUST
JUSTIFIED
KEY
LABEL
LAST
LEADING
LEFT
LENGTH
LESS
LIMIT
LIMITS
LINAGE
LINAGE-COUNTER
LINE
LINE-COUNTER
LINES
LINKAGE
LOCK
LOW-VALUE
LOW-VALUES
MERGE
MESSAGE
MODE
MOVE
MULTIPLE
MULTIPLY
NATIVE
NEGATIVE
NEXT
NO
NO DATA
NOT
NUMBER
NUMERIC
NUMERIC-EDITED
OBJECT-COMPUTER
OCCURS
OF
OFF
OMITTED
ON
OPEN
OPTIONAL
OR
ORDER
ORGANIZATION
OTHER
OUTPUT
OUTPUT PROCEDURE
OVERFLOW
PADDING
PAGE
PAGE-COUNTER
PERFORM
PF
PH
PIC
PICTURE
PLUS
POINTER
POSITION
POSITIVE
PRINTING
PROCEDURE
PROCEDURES
PROGRAM
PROGRAM-ID
PURGE
QUEUE
QUOTE
QUOTES
RANDOM
RD
READ
RECEIVE
RECORD
RECORDS
REDEFINES
REEL
REFERENCE
REFERENCE-MODIFIER
REFERENCES
RELATIVE
RELEASE
REMAINDER
REMOVAL
RENAMES
REPLACE
REPLACING
REPORT
REPORTING
REPORTS
RESERVE
RESET
RETURN
REWIND
REWRITE
RF
RH

RIGHT
ROUNDED
RUN
SAME
SD
SEARCH
SECTION
SECURITY
SEGMENT
SEGMENT-LIMIT
SELECT
SEND
SENTENCE
SEPARATE
SEQUENCE
SEQUENTIAL
SET
SIGN
SIZE
SIZE ERROR
SORT
SORT-MERGE
SOURCE
SOURCE-COMPUTER
SPACE
SPACES
SPECIAL-NAMES
STANDARD
STANDARD-1
STANDARD-2
START
STATUS
STOP
STOP RUN
STRING
SUB-QUEUE-1
SUB-QUEUE-2
SUB-QUEUE-3
SUBTRACT
SUM
SUPPRESS
SYMBOLIC
SYNC
SYNCHRONIZED
TABLE
TALLYING
TAPE
TERMINAL
TERMINATE
TEST
TEXT
THAN
THEN
THROUGH
THRU
TIME
TIMES
TO
TOP
TRAILING
TRUE
TYPE
UNIT
UNSTRING
UNTIL
UP
UPON
USAGE
USE
USING
VALUE
VALUES
VARYING
WHEN
WHEN OTHER
WITH
WORKING-STORAGE
WRITE
ZERO
ZEROES
ZEROS
+
-
*
/
**
>
<
=
.
,
(
)
'
"
:
;

APPENDIX B

COBOL PROGRAMMING

LANGUAGE SYNTAX

APPENDIX B : COBOL PROGRAMMING LANGUAGE SYNTAX

The following list of COBOL programming language syntax are based on the manual of [COB81] Draft Proposed Revised X3.23 American National Standard Programming COBOL.

I. NOTATIONAL CONVENTIONS

In presenting the description of the COBOL syntax, the following notation (Basically based on BNF, Backus-Naur Form) will be used :

1. The symbol "::=" should be read as "is defined to be."
2. The symbol "|" should be read as "or"
3. Strings in capital letters will denote COBOL symbols.
4. Strings in lower case letter surrounded by "<" and ">" will denote grammar definition symbols.
5. {x} will be used to denote an occurrence of the string x.
6. $\{x\}^{\#}$ will be used to denote one or more occurrences of the string x.
7. [x] will be used to indicate that the string x is optional.
8. $[x]^{\#}$ will be used to denote zero or more occurrences of the string x.
9. {a|b|c|d|...} will be used to denote an occurrence of string a or b or c or d, etc.

10. [a|b|c|d|...] will be used to denote an optional occurrence of string a or b or c or d, etc.
11. $\{a|b|c|d|...\}^{\#}$ will be used to denote an occurrence of one or more strings where each string is a or b or c or d, etc.
12. $[a|b|c|d|...]^{\#}$ will be used to denote an occurrence of zero or more strings where each string is a or b or c or d, etc.
13. { a|b|c } will be used to denote an occurrence of a string from the set {a, b, c, ab, ba, ac, ca, bc, cb, abc, acb, bac, bca, cab, cba}.

II. LANGUAGE CONCEPTS

(A) Character Set

Characters used in :

1.	words	0-9 A-Z a-z - (hyphen)
2.	punctuation	' " ( ) . , ; space : (colon)
3.	editing	B + - . , Z * $ 0 CR DB /
4.	arithmetic operations	+ - * / **
5.	relation conditions	> < =
6.	subscripting	+ -

(B) Separators

" ( ) . , ; space : (colon)

(C) Character Strings

(1) COBOL words
- (i) Maximum of 30 characters
- (ii) User-defined words(Identifiers)
  - (a) Alphabet-name
  - (b) Condition-name
  - (c) Data-name
  - (d) Index-name
  - (e) Level-number
  - (f) Mnemonic-name
  - (g) Paragraph-name
  - (h) Program-name
  - (i) Section-name
  - (j) Symbolic-character
- (iii) System-name
  - (a) Computer-name
  - (b) Implementor-name
- (iv) Reserved words
  - (a) Required words
    1. Key words
    2. Special character words
    3. Arithmetic operators + - * / **
    4. Arithmetic operators used in subscripting + -
    5. Relation characters > < =
  - (b) Optional words
  - (c) Special purpose words
    1. Figurative constants : ZERO, ZEROS, ZEROES, SPACE, SPACES, HIGH-VALUE, HIGH-VALUES, LOW-VALUE, LOW-VALUES, QUOTE, QUOTES
    2. Figurative constants : symbolic-character, ALL literal, ALL figurative-constant, ALL symbolic-character
- (v) Literals
  - (a) Numeric literals : 1 through 18 digits
  - (b) Nonnumeric literals : 1 through 160 characters

(2) PICTURE character-string

(3) Comment-entry

(D) **Uniqueness of Reference**

(1) Qualification
 (i) No qualification permitted; names must be unique if referenced
 (ii) Qualification permitted; 50 qualifiers maximum
(2) Subscripting
 (i) 3 levels of subscripts
 (ii) 48 levels of subscripts
 (iii) Subscripting with a literal
 (iv) Subscripting with a data-name
 (v) Subscripting with an index-name
 (vi) Relative subscripting
(3) Reference modification

(E) **Reference Format**

(1) Sequence number
(2) Continuation of lines
 (i) Continuation of nonnumeric literal
 (ii) Continuation of COBOL word, numeric literal, PICTURE character-string
(3) Blank lines
(4) Comment lines
 (i) Asterisk (*) comment line
 (ii) Slant (/) comment line
(5) Debugging line with D in indicator area

(F) **Source Program Structure**

(1) Identification Division required
(2) Environment Division optional
(3) Data Division optional
(4) Procedure Division optional
(5) End program header

(G) **IDENTIFICATION DIVISION**

(1) PROGRAM-ID paragraph
 (i) Program-name
(2) AUTHOR paragraph
(3) INSTALLATION paragraph
(4) DATE-WRITTEN paragraph
(5) DATE-COMPILED paragraph
(6) SECURITY paragraph

(H) ENVIRONMENT DIVISION

(1) Configuration Section
 (i) SOURCE-COMPUTER paragraph
  (a) computer-name
  (b) WITH DEBUGGING MODE clause
 (ii) OBJECT-COMPUTER paragraph
  (a) computer-name
  (b) PROGRAM COLLATING SEQUENCE clause
 (iii) SPECIAL-NAMES paragraph
 (iv) ALPHABET clause
  (a) STANDARD-1 option
  (b) STANDARD-2 option
  (c) NATIVE option
  (d) Implementor-name option
  (e) Literal option
 (v) CURRENCY SIGN clause
 (vi) DECIMAL-POINT clause
 (vii) Implementor-name clause
  (a) IS mnemonic-name option
  (b) ON STATUS IS condition-name option
  (c) OFF STATUS IS condition-name option
 (viii) SYMBOLIC CHARACTERS clause

(I) DATA DIVISION

(1) Working-Storage Section
(2) Record description entry
(3) 77 level description entry
(4) Data description entry
 (i) BLANK WHEN ZERO clause
 (ii) Data-name clause
 (iii) FILLER clause
 (iv) JUSTIFICATION clause
 (v) Level-number clause
  (a) 01 through 49; one or two digit representation
  (b) 66
  (c) 77
  (d) 88
 (vi) OCCURS clause
  (a) Integer TIMES
  (b) ASCENDING/DESCENDING KEY clause
  (c) INDEXED BY phrase
  (d) Integer-1 TO integer-2 TIMES DEPENDING ON phrase
 (vii) PICTURE clause
  (a) Character-string has a maximum of 30 characters
  (b) Data characters : X 9 A
  (c) Operational symbols : S V
  (d) Operational symbols : P
  (e) Nonfloating insertion characters
   B + - . , $ 0 CR DB /
  (f) Replacement or floating insertion characters
   % + - Z *
  (g) Currency sign substitution
  (h) Decimal-point substitution
 (viii) REDEFINES clause
  (a) May not be nested
  (b) May be nested
 (ix) RENAMES clause
 (x) SIGN clause
 (xi) SYNCHRONIZED clause
 (xii) USAGE clause
  (a) COMPUTATIONAL
  (b) DISPLAY
  (c) INDEX
 (xiii) Value clause
  (a) Literal
  (b) Literal series
  (c) Literal-1 THROUGH literal-2
  (d) Literal range series

(J) **PROCEDURE DIVISION**

(1) Arithmetic expression
(i) Binary arithmetic operators + - * / **
(ii) Unary arithmetic operators + -
(2) Conditional expressions
(i) Simple condition
(a) Relation condition
1. Relational operators
i. [NOT] GREATER THAN
ii. [NOT] >
iii. [NOT] LESS THAN
iv. [NOT] <
v. [NOT] EQUAL TO
vi. [NOT] =
2. Comparison of numeric operands
3. Comparison of nonnumeric operands
4. Comparison of index-name and/or index data items
(b) Class condition
1. NUMERIC
2. ALPHABETIC
3. ALPHABETIC-LOWER
4. ALPHABETIC-UPPER
(c) condition-name condition
(d) Switch-status condition
(e) Sigh condition
(ii) Complex condition
(a) Logical operators AND OR NOT
(b) Negated condition
(c) Combined condition
(d) Parenthesized condition
(iii) Abbreviated combined relation conditions
(iv) Arithmetic statements
(a) Arithmetic operands limited to 18 digits
(b) Composite of operands limited to 18 digits

III. THE COBOL GRAMMAR

The Symbols (words) of a COBOL program are composed either of individual characters or of strings of characters which may have complicated syntactic constructions.

COBOL words are classified as follows :

(A) Identifiers
(B) Reserved words
1. Key words
2. Special characters
3. Arithmetic operators + - * / **
4. Arithmetic operators used in subscripting + -
5. Relation characters > < =
(C) Literals
1. Numeric literals : 1 through 18 digits
2. Nonnumeric literals : 1 through 160 characters

IV. LOW LEVEL SYNTAX OF COBOL

The BNF grammar for low level syntax of COBOL.

<Alpha>::= {A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|
a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z}

<Digit>::= {0|1|2|3|4|5|6|7|8|9}

<Special Symbol>::= {<Operator>|<Special Character>}

<Operator>::= {<Arithmetic Operator>|<Logical Operator>|
<Relational Operator>}

<Arithmetic Operator>::= {+|-|*|/}

<Logical Operator>::= {NOT|AND|OR}

<Relational Operator>::= {<|>|=}

<Special Character>::= {Blank|.|'|"|<Separator>}

<Separator>::= {,|;|(|)|:}

<Identifier>::= {<Alpha>|<Digit>|-}#

<Numeric Constant>::= <Decimal Number>

<Decimal Number>::= {Decimal Digit}#

<Decimal Digit>::= {0|1|2|3|4|5|6|7|8|9}

<String Constant>::= '<Character String>'

<Character String>::= {<Character>}#

As part of the definition of COBOL language, a set of rules [STE79] defining the formation of words out of characters was provided as following :

RULES FOR FORMING IDENTIFIERS

1. 1 to 30 characters.
2. Letters, digits, and hyphens (-) only.
3. May not begin or end with a hyphen.
4. No embedded blanks (i.e. blanks within the identifier).
5. Must contain at least one alphabetic character.
6. May not be a COBOL reserved word(Listed in Appendix A).

RULES FOR FORMING NUMERIC LITERALS

1. A maximum of 18 digits.
2. An optional plus or minus sign to the **left** of the number.
3. A decimal point **within** the literal. The decimal point, however, may not be the last character of the literal.

RULES FOR FORMING NONNUMERIC LITERALS

1. The literal must be enclosed in quotation marks [Single quotation mark(') or double quotation mark(")].
2. A maximum of 160 characters, including spaces, may be used.
3. Any character in the COBOL Character Set may be used except the quotation mark as represented by two adjacent quotation marks.

MARGIN RULES

1. Division, section, and paragraph names began in Margin A.
2. All other statements and clauses began in Margin B.

COBOL CODING FORM

Columns	Use	Explanation
1-6	Sequence numbers (optional)	Used as a method for ensuring that cards are sequenced properly.
7	Continuation or Comment	Used to continue to next line or denote comment line.
8-11	Margin A	Specific entries such as DIVISION, SECTION, and paragraph names must be coded in Margin A.
12-72	Margin B	Most COBOL entries, particularly those in the PROCEDURE DIVISION, are coded in Margin B.
73-80	Program identification (optional)	Used to identify program.

LINE CONTINUATION RULES

(A) Continuation of lines
  1. Continuation of nonnumeric literal
  2. Continuation of COBOL word, numeric literal
(B) Blank lines
(C) Comment lines
  1. Asterisk (*) comment line
  2. Slant (/) comment line
(D) Debugging line with D in indicator area

V. GENERAL FORMAT FOR COBOL SOURCE PROGRAM

Abbreviation :

(1) stm ::= Statement

```
<COBOL Program>::= {<Identification-division>}
                   [<Environment-division>]    [<Data-division>]
                   [<Procedure-division>]      [<End-program>]

<Identification-division>::=
        IDENTIFICATION DIVISION {.} PROGRAM-ID {.} {<Program-name-1>}
                     [ [IS] { COMMON | INITIAL } PROGRAM ] {.}

<Program-name>::= <Identifier>

<Environment-division>::=
        ENVIRONMENT DIVISION {.} environment-division-content

<Data-division>::= DATA DIVISION {.} data-division-content

<Procedure-division>::= PROCEDURE DIVISION procedure-division-content

<End-program>::= END PROGRAM <Program-name-1> {.}
```

VI. GENERAL FORMAT FOR PROCEDURE DIVISION

Format 1:

```
<Procedure-division>::= [ PROCEDURE DIVISION
     [ USING {<data-name-1>}# ] {.}
     [<declaratives>]     { <section-1> }#  ]
<Data-name>::= <Identifier>

<Declaratives>::= DECLARATIVES {.} {<use-section-1>}#
                  END DECLARATIVES {.}

<Use-section>::= {<section-name-1>} SECTION [<segment-number>] {.}
                  <USE-statement-1> [<paragraph-1>]#
<Section-name>::= <Identifier>
<Segment-number>::= <Integer>

<Section>::= {<section-name-1>} SECTION  [<segment-number>] {.}
             [<paragraph-1>]#

<Paragraph>::= {<paragraph-name-1>} {.} [<sentence-1>]#
<Paragraph-name>::= <Identifier>
<Sentence>::= [<statement-1> [;] ]# {.}

<Statement>::= { <Simple-statement-1>  |  <Structured-statement-1> }

<Simple-statement>::=
    { <Nucleus-statement-1>          | <Sequential-i-o-statement-1> |
      <Relative-i-o-statement-1>     | <Inter-program-statement-1>  |
      <Sort-merge-statement-1>       | <Report-writer-statement-1>  |
      <Communication-statement-1>    | <Assignment-statement-1>     }

<Structured-statement>::=
   { <Conditional-statement-1> | <Compound-statement-1> }

<Nucleus-statement>::=
   {  < ACCEPT-statement-1 >        |  < CONTINUE-statement-1 >    |
      < DISPLAY-statement-1 >       |  < EXIT-statement-1 >        |
      < GO-TO-statement-1 >         |  < INITIALIZE-statement-1 >  |
      < INSPECT-statement-1 >       |  < STOP-statement-1 >        }

<Sequential-i-o-statement>::=
   {  < CLOSE-statement-1 >         |  < OPEN-statement-1 >        |
      < READ-statement-1 >          |  < REWRITE-statement-1 >     |
      < USE-statement-1 >           |  < WRITE-statement-1 >       }
```

```
<Relative-i-o-statement>::=
   {  < DELETE-statement-1 >        |  < START-statement-1 >          }

<Inter-program-statement>::=
   {  < CANCEL-statement-1 >        |  < EXIT-PROGRAM-statement-1 >  }

<Sort-merge-statement>::=
   {  < MERGE-statement-1 >         |  < RELEASE-statement-1 >       |
      < RETURN-statement-1 >        |  < SORT-statement-1 >          }

<Report-writer-statement>::=
   {  < GENERATE-statement-1 >      |  < INITIATE-statement-1 >     |
      < SUPPRESS-statement-1 >      |  < TERMINATE-statement-1 >    }

<Communication-statement>::=
   {  < DISABLE-statement-1 >       |  < ENABLE-statement-1 >       |
      < PURGE-statement-1 >         |  < RECEIVE-statement-1 >      |
      < SEND-statement-1 >          }

<Assignment-statement>::=
   {  < MOVE-statement-1 >          |  < SET-statement-1 >           }

<Conditional-statement>::=
   {  < EVALUATE-statement-1 >      |  < IF-statement-1 >           |
      < SEARCH-statement-1 >        |  < SEARCH-ALL-statement-1 >   }

<Compound-statement>::=
   {  < Arithmetic-statement-1 >    |  < CALL-statement-1 >         |
      < PERFORM-statement-1 >       |  < STRING-statement-1 >       |
      < UNSTRING-statement-1 >      }

<Arithmetic-statement>::=
   {  < ADD-statement-1 >           |  < COMPUTE-statement-1 >      |
      < DIVIDE-statement-1 >        |  < MULTIPLY-statement-1 >     |
      < SUBTRACT-statement-1 >      }
```

Format 2:

```
<Procedure-division>::=  [ PROCEDURE DIVISION
             [ USING {data-name-1}# ] {.}  { <paragraph-1> }#  ]


<Paragraph>::= {Paragraph-name-1} {.} [<sentence-1>]#

<Sentence>::= [<statement-1>]# {.}
```

VII. GENERAL FORMAT FOR COBOL VERBS

Abbreviation :

(1) stm ::= Statement

[1] < ACCEPT statement >

Format 1 :

< ACCEPT stm >::= ACCEPT identifier-1 [FROM mnemonic-name-1]

Format 2 :

< ACCEPT stm >::= ACCEPT identifier-2 FROM
{ DATE | DAY | DAY-OF-WEEK | TIME }

Format 3 :

< ACCEPT stm >::= ACCEPT cd-name-1 MESSAGE COUNT

[2] < ADD statement >

Format 1 :

< ADD stm >::= ADD {identifier-1 | literal-1}# TO
{identifier-2 [ROUNDED]}#
[[ON] SIZE ERROR imperative-stm-1 [END-ADD]]

Format 2 :

< ADD stm >::= ADD {identifier-1 | literal-1}# TO
{identifier-2 | literal-2}
GIVING {identifier-3 [ROUNDED]}#
[[ON] SIZE ERROR imperative-stm-1 [END-ADD]]

Format 3 :

< ADD stm >::= ADD { CORRESPONDING | CORR } identifier-1
TO identifier-2 [ROUNDED]
[[ON] SIZE ERROR imperative-stm-1 [END-ADD]]

[3] < CALL statement >

```
< CALL stm >::= CALL {identifier-1 | literal-1}
   [ USING {[BY REFERENCE] {data-name-1}# |
            BY CONTENT     {data-name-1}#   }# ]
   [ [ON] OVERFLOW imperative-stm-1 [END-CALL] ]
```

[4] < CANCEL statement >

```
< CANCEL stm >::= CANCEL {identifier-1 | literal-1 }
```

[5] < CLOSE statement >

Format 1 :

```
< CLOSE stm >::= CLOSE  { file-name-1
   [{ REEL | UNIT } [FOR REMOVAL] |
      WITH { NO REWIND | LOCK }     ] }#
```

Format 2 :

```
< CLOSE stm >::= CLOSE  { file-name-1 [WITH LOCK]  }#
```

[6] < COMPUTE statement >

```
< COMPUTE stm >::= COMPUTE
   {identifier-1 [ROUNDED]}# = arithmetic-expression-1
   [ [ON] SIZE ERROR imperative-stm-1 [END-COMPUTE] ]
```

[7] < CONTINUE statement >

```
< CONTINUE stm >::= CONTINUE
```

[8] < DELETE statement >

```
< DELETE stm >::= DELETE file-name-1 [RECORD]
    [ INVALID [KEY] imperative-stm-1 [END-DELETE] ]
```

[9] < DISABLE statement >

```
< DISABLE stm >::= DISABLE
    { INPUT [TERMINAL] | I-O TERMINAL | OUTPUT } cd-name-1
```

[10] < DISPLAY statement >

```
< DISPLAY stm >::= DISPLAY {identifier-1 | literal-1}#
    [ UPON mnemonic-name-1 ] [[WITH] NO [ADVANCING] ]
```

[11] < DIVIDE statement >

Format 1 :

```
< DIVIDE stm >::= DIVIDE {identifier-1 | literal-1}
    INTO {identifier-2 [ROUNDED]}#
    [ [ON] SIZE ERROR imperative-stm-1 [END-DIVIDE] ]
```

Format 2 :

```
< DIVIDE stm >::= DIVIDE {identifier-1 | literal-1}
    INTO {identifier-2 | literal-2}
    GIVING {identifier-3 [ROUNDED]}#
    [ [ON] SIZE ERROR imperative-stm-1 [END-DIVIDE] ]
```

Format 3 :

```
< DIVIDE stm >::= DIVIDE {identifier-1 | literal-1}
    BY {identifier-2 | literal-2}
    GIVING {identifier-3 [ROUNDED]}#
    [ [ON] SIZE ERROR imperative-stm-1 [END-DIVIDE] ]
```

Format 4 :

```
< DIVIDE stm >::= DIVIDE {identifier-1 | literal-1}
   INTO {identifier-2 | literal-2}
   GIVING identifier-3 [ROUNDED]  REMAINDER identifier-4
   [ [ON] SIZE ERROR imperative-stm-1 [END-DIVIDE] ]
```

Format 5 :

```
< DIVIDE stm >::= DIVIDE {identifier-1 | literal-1}
   BY {identifier-2 | literal-2}
   GIVING identifier-3 [ROUNDED]  REMAINDER identifier-4
   [ [ON] SIZE ERROR imperative-stm-1 [END-DIVIDE] ]
```

[12] < ENABLE statement >

```
< ENABLE stm >::= ENABLE
   { INPUT [TERMINAL] | I-O TERMINAL | OUTPUT }  cd-name-1
```

[13] < EVALUATE statement >

```
< EVALUATE stm >::= EVALUATE
   {identifier-1 | literal-1 | expression-1 | TRUE | FALSE }#
   { { WHEN
         { ANY | condition-1 | TRUE | FALSE   |
           [NOT]
             { {identifier-2 | literal-2 | arithmetic-expression-1}
               [ { THROUGH | THRU }
               {identifier-3 | literal-3 | arithmetic-expression-2} ] }
         }#   }#
         imperative-stm-1   }#
   [ WHEN OTHER imperative-stm-2 ]
   [ END EVALUATE ]
```

[14] < EXIT statement >

< EXIT stm >::= EXIT [.]

[15] < EXIT PROGRAM statement >

< EXIT PROGRAM stm >::= EXIT PROGRAM

[16] < GENERATE statement >

< GENERATE stm >::= GENERATE {data-name-1 | report-name-1}

[17] < GO TO statement >

Format 1 :

< GO TO stm >::= GO [TO] procedure-name-1

Format 2 :

< GO TO stm >::= GO [TO] {procedure-name-1}# DEPENDING [ON] identifier-1

[18] < IF statement >

```
< IF stm >::= IF condition-1
   [ THEN] { {statement-1}# | NEXT SENTENCE }
   { ELSE {statement-2}# [END-IF] |
     ELSE NEXT SENTENCE           |
     END-IF  }
```

[19] < INITIALIZE statement >

```
< INITIALIZE stm >::= INITIALIZE {identifier}#
   [REPLACING { ALPHABETIC | ALPHANUMERIC |
                NUMERIC | ALPHANUMERIC-EDITED | NUMERIC-EDITED }
      [DATA] BY {identifier-2 | literal-1} ]
```

[20] < INITIATE statement >

```
< INITIATE stm >::= INITIATE {report-name-1}#
```

[21] < INSPECT statement >

Format 1 :

```
< INSPECT stm >::= INSPECT identifier-1 TALLYING
   {identifier-2 FOR
     { CHARACTERS
        [{ BEFORE | AFTER [INITIAL] {identifier-4 | literal-2}]# |
     { ALL | LEADING }
     { {identifier-3 | literal-1}
       [{ BEFORE | AFTER [INITIAL] {identifier-4 | literal-2}]#
     }#      }#     }#
```

Format 2 :

```
< INSPECT stm >::= INSPECT identifier-1 REPLACING
   { CHARACTERS BY {identifier-5 | literal-3}
       [{ BEFORE | AFTER } [INITIAL] {identifier-4 | literal-2}]# |
     { ALL | LEADING | FIRST }
     { {identifier-3 | literal-1} BY {identifier-5 | literal-3}
       [{ BEFORE | AFTER } [INITIAL] {identifier-4 | literal-2}]#
     }#    }#
```

Format 3 :

```
< INSPECT stm >::= INSPECT identifier-1 TALLYING
   {identifier-2 FOR
     { CHARACTERS
        [{ BEFORE | AFTER } [INITIAL] {identifier-4 | literal-2}]# |
       { ALL | LEADING }
       { {identifier-3 | literal-1}
         [{ BEFORE | AFTER } [INITIAL] {identifier-4 | literal-2}]#
       }#      }#     }#
   REPLACING
   { CHARACTERS BY {identifier-5 | literal-3}
       [{ BEFORE | AFTER } [INITIAL] {identifier-4 | literal-2}]# |
     { ALL | LEADING | FIRST }
       { {identifier-3 | literal-1} BY {identifier-5 | literal-3}
       [{ BEFORE | AFTER } [INITIAL] {identifier-4 | literal-2}]#
     }#    }#
```

Format 4 :

```
< INSPECT stm >::= INSPECT identifier-1
   CONVERTING {identifier-6 | literal-4}
   TO         {identifier-7 | literal-5}
   [{ BEFORE | AFTER } [INITIAL] {identifier-4 | literal-2}]#
```

[22] < MERGE statement >

```
< MERGE stm >::= MERGE file-name-1
   { [ON] { ASCENDING | DESCENDING } [KEY] {data-name-1}# }#
   [ [COLLATING] SEQUENCE [IS] alphabet-name-1 ]
   USING file-name-2 {file-name-3}#
   { OUTPUT PROCEDURE [IS]  section-name-1
      [ { THROUGH | THRU } section-name-2 ] |
     GIVING {file-name-4}#                        }
```

[23] < MOVE statement >

Format 1 :

```
< MOVE stm >::= MOVE {identifier-1 | literal-1} TO {identifier-2}#
```

Format 2 :

```
< MOVE stm >::= MOVE { CORRESPONDING | CORR }
   identifier-1 TO identifier-2
```

[24] < MULTIPLY statement >

Format 1 :

```
< MULTIPLY stm >::= MULTIPLY {identifier-1 | literal-1}
   BY { identifier-2 [ROUNDED] }#
   [ [ON] SIZE ERROR imperative-stm-1 [END-MULTIPLY] ]
```

Format 2 :

```
< MULTIPLY stm >::= MULTIPLY {identifier-1 | literal-1}
   BY {identifier-2 | literal-2}
   GIVING { identifier-3 [ROUNDED] }#
   [ [ON] SIZE ERROR imperative-stm-1 [END-MULTIPLY] ]
```

[25] < OPEN statement >

Format 1 :

```
< OPEN stm >::= OPEN
   { INPUT  {file-name-1 [[WITH] NO REWIND]}# |
     OUTPUT {file-name-2 [[WITH] NO REWIND]}# |
     I-O    {file-name-3}#                    |
     EXTEND {file-name-4}#                      }#
```

Format 2 :

```
< OPEN stm >::= OPEN
   { INPUT  {file-name-1}# |  OUTPUT {file-name-2}# |
     I-O    {file-name-3}# |  EXTEND {file-name-4}#   }#
```

Format 3 :

```
< OPEN stm >::= OPEN
   { OUTPUT {file-name-1 [ [WITH] NO REWIND ] }# |
     EXTEND {file-name-2}#                       }#
```

[26] < PERFORM statement >

Format 1 :

```
< PERFORM stm >::= PERFORM  [procedure-name-1
   [ { THROUGH | THRU } procedure-name-2] ]
   [ imperative-stm-1 END-PERFORM ]
```

Format 2 :

```
< PERFORM stm >::= PERFORM  [procedure-name-1
   [ { THROUGH | THRU } procedure-name-2] ]
   {identifier-1 | literal-1} TIMES
   [ imperative-stm-1 END-PERFORM ]
```

Format 3 :

```
< PERFORM stm >::= PERFORM  [procedure-name-1
   [ { THROUGH | THRU } procedure-name-2] ]
   [ [WITH] TEST { BEFORE | AFTER } ]
   UNTIL condition-1
   [ imperative-stm-1 END-PERFORM ]
```

Format 4 :

```
< PERFORM stm >::= PERFORM  [procedure-name-1
   [ { THROUGH | THRU } procedure-name-2] ]
   [ [WITH] TEST { BEFORE | AFTER } ]
   VARYING {identifier-2 | index-name-1}
     FROM  {identifier-3 | index-name-2 | literal-1}
     BY    {identifier-4 | literal-2}
     UNTIL  condition-1
   [AFTER  {identifier-5 | index-name-3}
     FROM  {identifier-6 | index-name-4 | literal-3}
     BY    {identifier-7 | literal-7}
     UNTIL  condition-2]#
   [ imperative-stm-1 END-PERFORM ]
```

[27] < PURGE statement >

```
< PURGE stm >::= PURGE cd-name-1
```

[28] < READ statement >

Format 1 :

```
< READ stm >::= READ file-name-1  [NEXT] [RECORD]
   [INTO identifier-1]
   [ [AT] END imperative-stm-1 [END-READ] ]
```

Format 2 :

```
< READ stm >::= READ file-name-1 [RECORD]
   [INTO identifier-1]
   [ INVALID [KEY] imperative-stm-1 [END-READ] ]
```

Format 3 :

```
< READ stm >::= READ file-name-1 [RECORD]
   [INTO identifier-1]
   [KEY [IS] data-name-1]
   [ INVALID [KEY] imperative-stm-1 [END-READ] ]
```

[29] < RECEIVE statement >

```
< RECEIVE stm >::= RECEIVE cd-name-1
   { MESSAGE | SEGMENT } INTO identifier-1
   [ NO DATA imperative-stm-1 [END-RECEIVE] ]
```

[30] < RELEASE statement >

```
< RELEASE stm >::= RELEASE record-name-1  [FROM identifier-1]
```

[31] < RETURN statement >

```
< RETURN stm >::= RETURN file-name-1 [RECORD]
   [INTO identifier-1]
   [ [AT] END imperative-stm-1 [END-RETURN ]
```

[32] < REWRITE statement >

Format 1 :

```
< REWRITE stm >::= REWRITE record-name-1  [FROM identifier-1]
```

Format 2 :

```
< REWRITE stm >::= REWRITE record-name-1  [FROM identifier-1]
   [ INVALID [KEY] imperative-stm-1 [END-REWRITE] ]
```

[33] < SEARCH statement >

```
< SEARCH stm >::= SEARCH identifier-1
   [VARYING {identifier-2 | index-name-1}]
   [ [AT] END imperative-stm-1 ]
   { WHEN condition-1 {imperative-stm-1 | NEXT SENTENCE } }#
   [END-SEARCH]
```

[34] < SEARCH ALL statement >

```
< SEARCH ALL stm >::= SEARCH ALL identifier-1
   [ [AT] END imperative-stm-1 ]
   WHEN
     {data-name-1 [IS] { EQUAL [TO] | = }
       {identifier-3 | literal-1 | arithmetic-expression-1} |
     condition-name-1 }
   [AND
     {data-name-2 [IS] { EQUAL [TO] | = }
       {identifier-4 | literal-2 | arithmetic-expression-2} |
     condition-name-2 }  ]#
   {imperative-stm-2 | NEXT SENTENCE }
   [END-SEARCH]
```

[35] < SEND statement >

Format 1 :

```
< SEND stm >::= SEND cd-name-1 FROM identifier-1
```

Format 2 :

```
< SEND stm >::= SEND cd-name-1 [FROM identifier-1]
   { [WITH] {identifier-2 | ESI | EMI | EGI} }
   [{ BEFORE | AFTER } [ADVANCING]
        { {identifier-3 | integer-1} [LINE | LINES]  |
          {mnemonic-name-1 | PAGE }  }  ]
   [ REPLACING LINE ]
```

[36] < SET statement >

Format 1 :

```
< SET stm >::= SET {index-name-1 | identifier-1}#
   TO {index-name-2 | identifier-2 | integer-1}
```

Format 2 :

```
< SET stm >::= SET {index-name-3}#
   { UP BY | DOWN BY }  {identifier-3 | integer-2}
```

Format 3 :

```
< SET stm >::= SET
   { {mnemonic-name-1}# TO { ON | OFF } }#
```

Format 4 :

```
< SET stm >::= SET {condition-name-1}#  TO TRUE
```

[37] < SORT statement >

```
< SORT stm >::= SORT file-name-1
   {[ON] { ASCENDING | DESCENDING } [KEY] {data-name-1}# }#
   [ [WITH] DUPLICATES [IN] [ORDER] ]
   [ [COLLATING] SEQUENCE [IS] alphabet-name-1]
   { INPUT  PROCEDURE [IS]  section-name-1
     [{ THROUGH | THRU }   section-name-2] |
     USING  {file-name-2}#    }
   { OUTPUT PROCEDURE [IS]  section-name-3
     [{ THROUGH | THRU }   section-name-4] |
     GIVING {file-name-4}#    }
```

[38] < START statement >

```
< START stm >::= START file-name-1
   [KEY [IS] { EQUAL [TO] | = | GREATER [THAN] | > |
               NOT LESS [THAN] | NOT <       } data-name-1 ]
   [ INVALID [KEY] imperative-stm-1 [END-START] ]
```

[39] < STOP statement >

```
< STOP stm >::= STOP { RUN | literal-1 }
```

[40] < STRING statement >

```
< STRING stm >::= STRING
   { { identifier-1 | literal-1 }# DELIMITED [BY]
     { identifier-2 | literal-2 | SIZE  }        }#
   INTO identifier-3
   [ [WITH] POINTER identifier-4 ]
   [ ON OVERFLOW imperative-stm-1 [END-STRING] ]
```

[41] < SUBTRACT statement >

Format 1 :

```
< SUBTRACT stm >::= SUBTRACT {identifier-1 | literal-1}# FROM
    { identifier-3 [ROUNDED] }#
    [ [ON] SIZE ERROR imperative-stm-1 [END-SUBTRACT] ]
```

Format 2 :

```
< SUBTRACT stm >::= SUBTRACT {identifier-1 | literal-1}# FROM
    {identifier-2 | literal-2} GIVING { identifier-3 [ROUNDED] }#
    [ [ON] SIZE ERROR imperative-stm-1 [END-SUBTRACT] ]
```

Format 3 :

```
< SUBTRACT stm >::= SUBTRACT { CORRESPONDING | CORR }
    identifier-1 FROM identifier-2 [ROUNDED]
    [ [ON] SIZE ERROR imperative-stm-1 [END-SUBTRACT] ]
```

[42] < SUPPRESS statement >

```
< SUPPRESS stm >::= SUPPRESS PRINTING
```

[43] < TERMINATE statement >

```
< TERMINATE stm >::= TERMINATE {report-name-1}#
```

[44] < UNSTRING statement >

```
< UNSTRING stm >::= UNSTRING identifier-1
    [ DELIMITED [BY] [ALL] {identifier-2 | literal-1}
      [ OR [ALL] {identifier-3 | literal-2} ]#       ]
    INTO
      {identifier-4  [DELIMITED [IN] identifier-5]
                     [COUNT     [IN] identifier-6]  }#
    [ [WITH] POINTER identifier-7 ]
    [ TALLYING [IN]  identifier-8 ]
    [ ON OVERFLOW imperative-stm-1 [END-UNSTRING] ]
```

[45] < USE statement >

Format 1 :

```
< USE stm >::= USE [GLOBAL] AFTER [STANDARD]
    { EXCEPTION | ERROR } PROCEDURE [ON]
    { {file-name-1}# | INPUT | OUTPUT | I-O | EXTEND } [.]
```

Format 2 :

```
< USE stm >::= USE [GLOBAL] BEFORE REPORTING identifier-1 [.]
```

Format 3 :

```
< USE stm >::= USE [FOR] DEBUGGING [ON]
    { [ ALL [REFERENCES] [OF] identifier-1 [ [WITH] CONVERSION ] |
      cd-name-1 | file-name-1 | procedure-name-1 | ALL PROCEDURES }# [.]
```

[46] < WRITE statement >

Format 1 :

```
< WRITE stm >::= WRITE record-name-1
   [FROM identifier-1]
   [ { BEFORE | AFTER } [ADVANCING]
     { {identifier-2 | integer-1} [LINE | LINES] |
       {mnemonic-name-1 | PAGE }  }  ]
   [ [AT] { END-OF-PAGE | EOP }
   imperative-stm-1 [END-WRITE] ]
```

Format 2 :

```
< WRITE stm >::= WRITE record-name-1 [FROM identifier-1]
   [ INVALID [KEY] imperative-stm-1 [END-WRITE] ]
```

VIII. GENERAL FORMAT FOR CONDITIONS

1. RELATION CONDITION

{identifier-1 | literal-1 | arithmetic-expression-1} [IS] [NOT]
{ GREATER [THAN] | > | LESS [THAN] | < | EQUAL [TO] | = }
{ identifier-2 | literal-2 | arithmetic-expression-2 }

2. CLASS CONDITIONS

identifier-1 [IS] [NOT]
{ NUMERIC | ALPHABETIC | ALPHABETIC-LOWER | ALPHABETIC-UPPER }

3. CONDITION-NAME CONDITIONS

condition-name-1

4. SWITCH-STATUS CONDITIONS

condition-name-1

5. SIGN CONDITIONS

arithmetic-expression-1 [IS] [NOT] { POSITIVE | NEGATIVE | ZERO }

6. NEGATED CONDITIONS

[NOT] condition-1

7. COMBINED CONDITIONS

condition-1 { { AND | OR } condition-2 }#

8. ABBREVIATED COMBINED RELATION CONDITIONS

relation-condition-1
{ { AND | OR } [NOT] [relational-operator] object }#

IX. GENERAL FORMAT FOR QUALIFICATION

Format 1 :

```
{data-name-1 | condition-name-1}
{ { { IN | OF } data-name-2 }#
  [ { IN | OF } {file-name-1 | cd-name} ] }
```

Format 2 :

```
paragraph-name-1 { IN | OF } section-name-1
```

Format 3 :

```
text-name { IN | OF } library-name
```

Format 4 :

```
LINAGE-COUNTER { IN | OF } file-name-1
```

Format 5 :

```
{ PAGE-COUNTER | LINE-COUNTER } { IN | OF } report-name-1
```

Format 6 :

```
data-name-3
{ { IN | OF } data-name-4 [{ IN | OF } report-name-1]  |
  { IN | OF } report-name-1   }
```

X. MISCELLANEOUS FORMATS

1. SUBSCRIPTING :

```
{condition-name-1 | data-name-1}
[()] {integer-1 | data-name-2 [{+|-} integer-2] |
                  index-name-2 [{+|-} integer-3]   }  [)]
```

2. REFERENCE MODIFICATION :

```
data-name-1 [(] leftmost-character-position [:] [length] [)]
```

3. IDENTIFIER :

```
data-name-1  [{ IN | OF } data-name-2]#
[ { IN | OF } {cd-name-1 | file-name-1 | report-name-1} ]
[ [(] {subscript}# [)] ]
[ [(] leftmost-character-position [:] [length] [)] ]
```

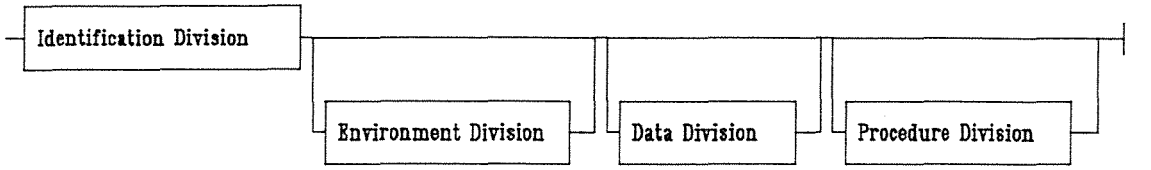
APPENDIX C

COBOL SYNTAX

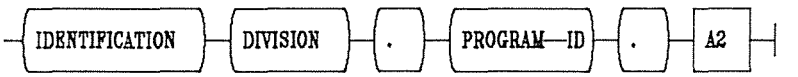
TRAM-LINE DIAGRAM

APPENDIX C : COBOL SYNTAX TRAM-LINE DIAGRAM

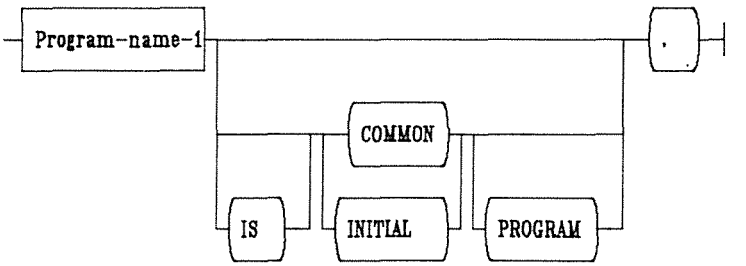
A1 NESTED-SOURCE-PROGRAM



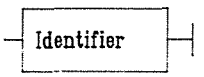
A2 Identification Division



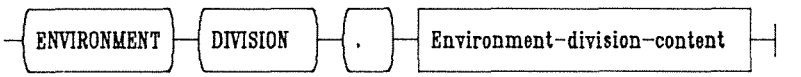
A2



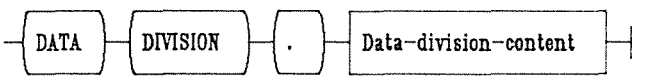
Program-name



A3 Environment Division



A4 Data Division



A5 Procedure Division

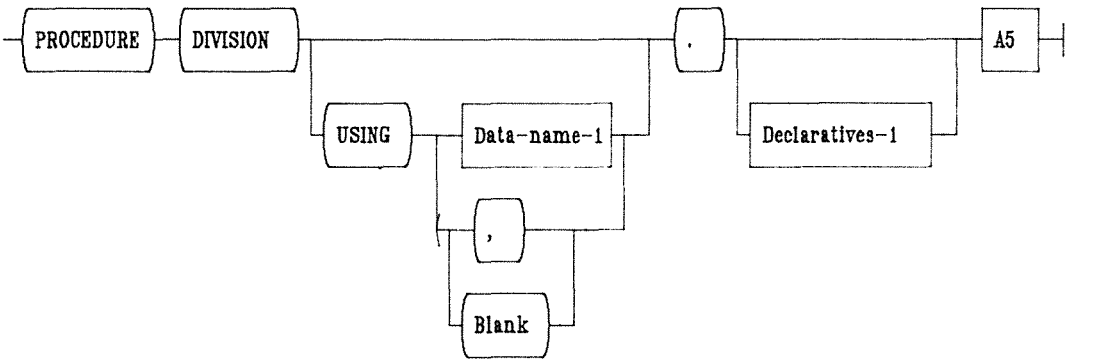
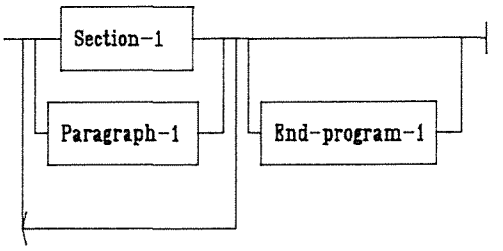
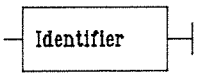


Figure C.1 COBOL Source Program Tram-Line Diagram.

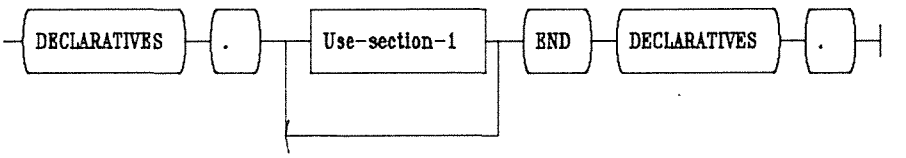
A5



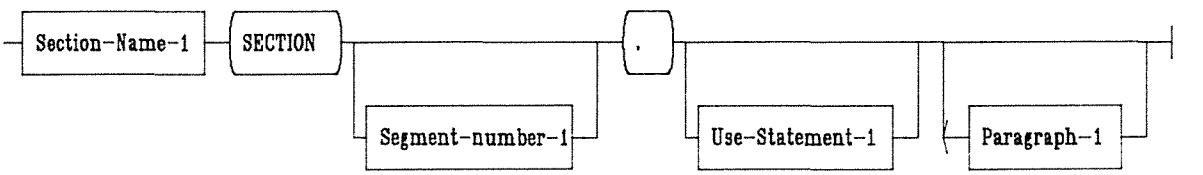
Data-name



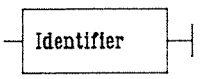
Declaratives



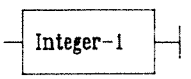
Use-Section



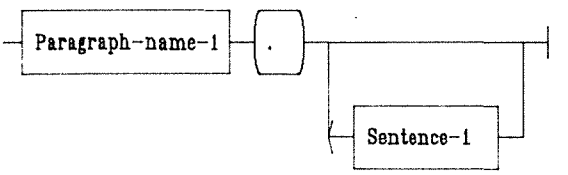
Section-name



Segment-number



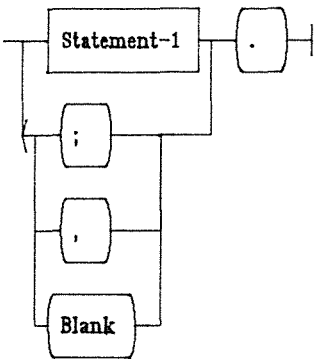
Paragraph



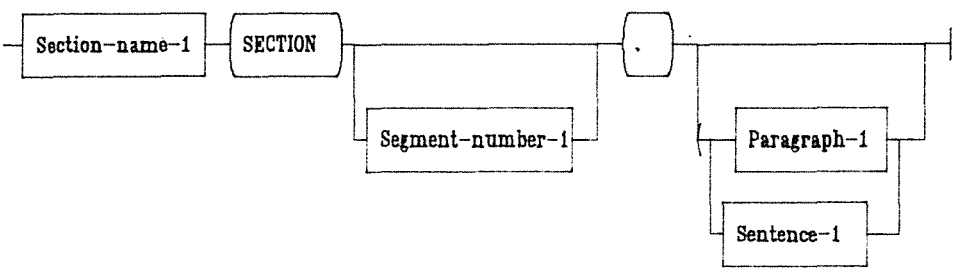
Paragraph-name

Figure C.2 COBOL Source Program Tram-Line Diagram (Continue).

Sentence



Section



End-program

Figure C.3 COBOL Source Program Tram-Line Diagram (Continue).

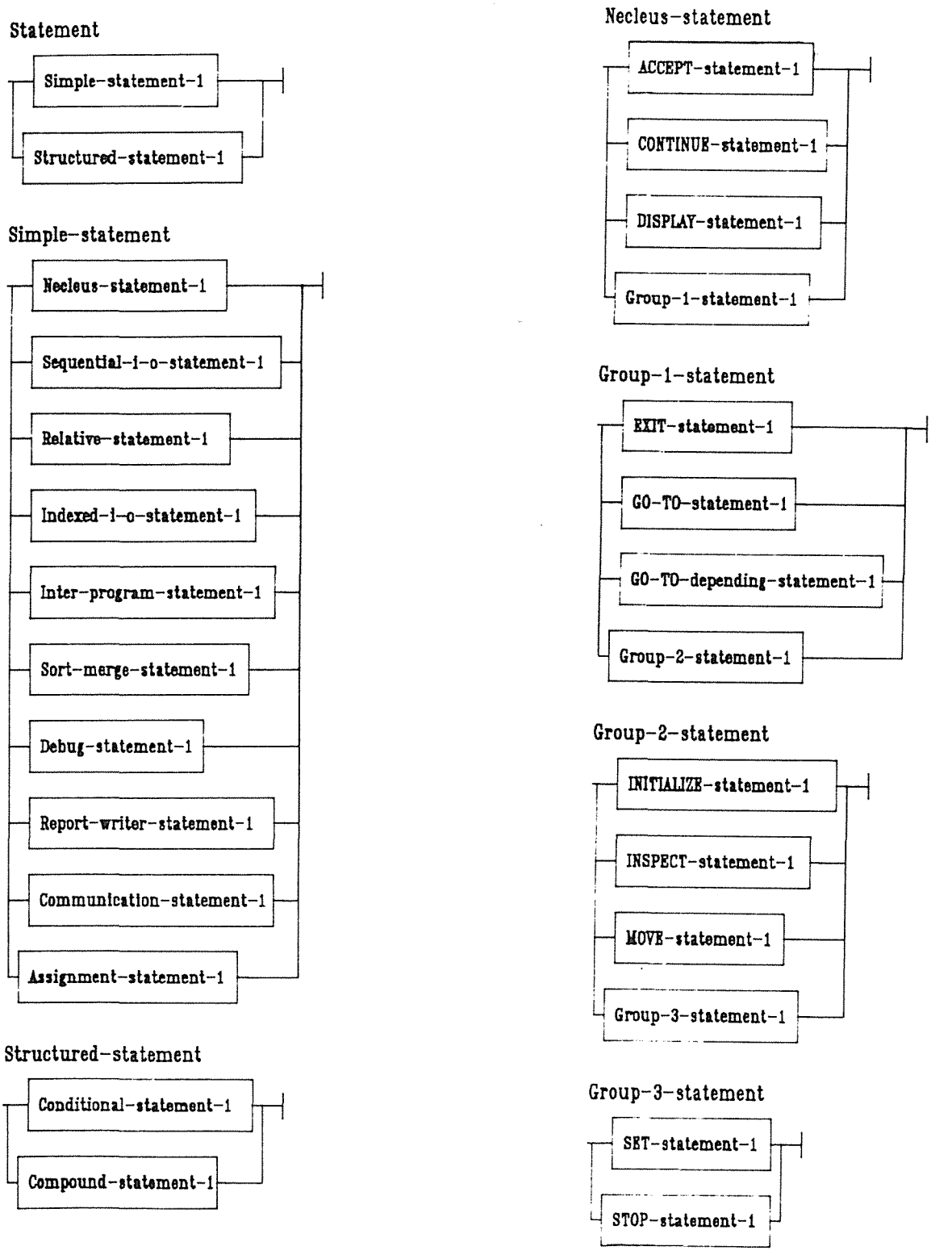
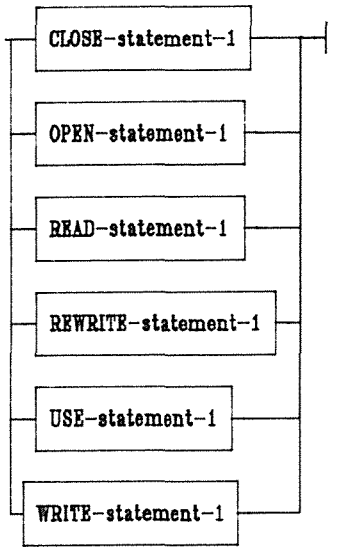
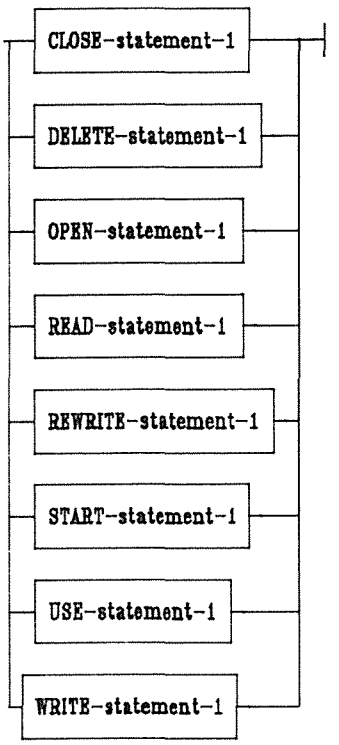


Figure C.4 COBOL Statement Type Tram-Line Diagram.

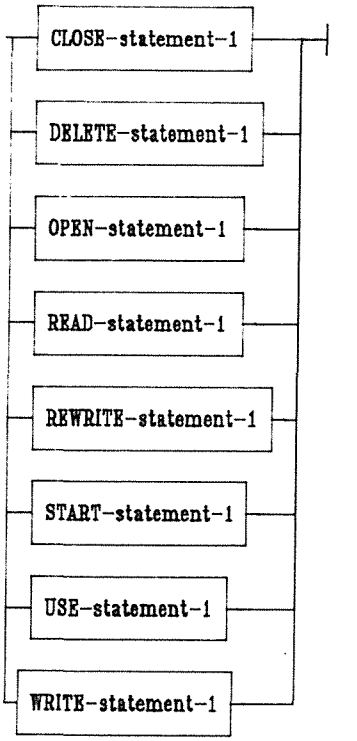
Sequential-i-o-statement



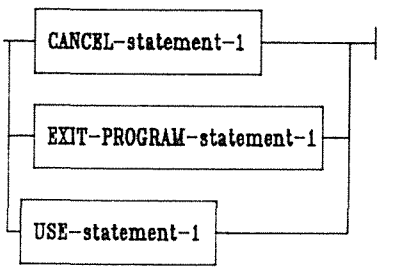
Relative-i-o-statement



Indexed-i-o-statement



Inter-program-statement



Sort-merge-statement

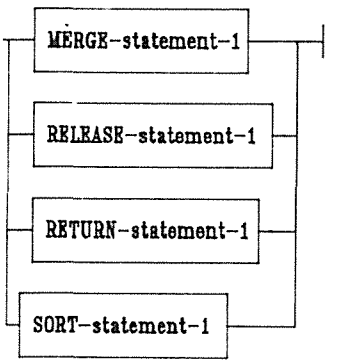
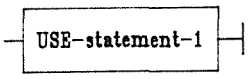
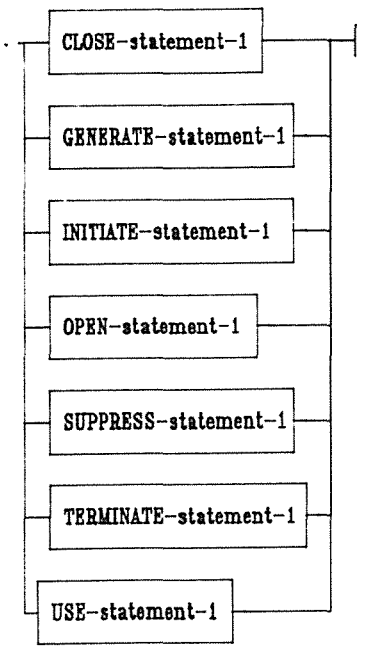


Figure C.5 COBOL Statement Type Tram-Line Diagram (Continue).

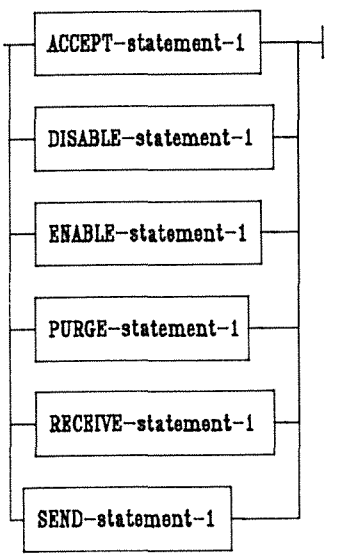
Debug-statement



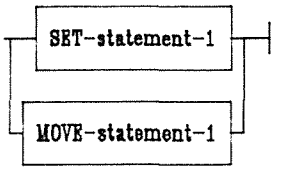
Report-writer-statement



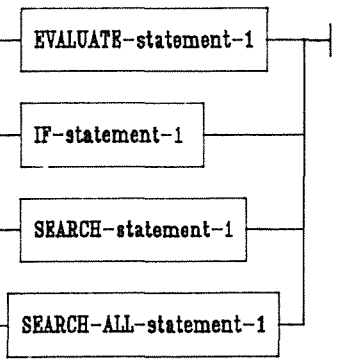
Communication-statement



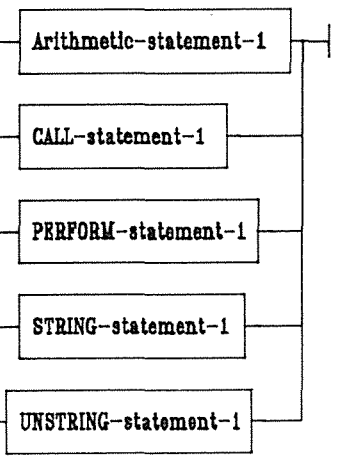
Assignment-statement



Conditional-statement



Compound-statement



Arithmetic-statement

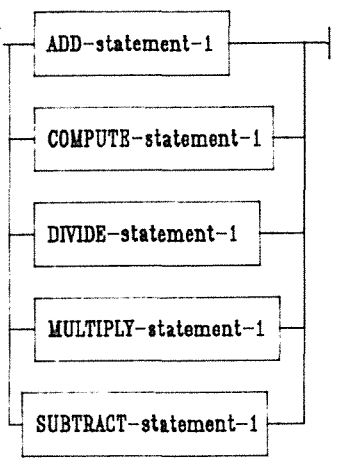


Figure C.6 COBOL Statement Type Tram-Line Diagram (Continue).

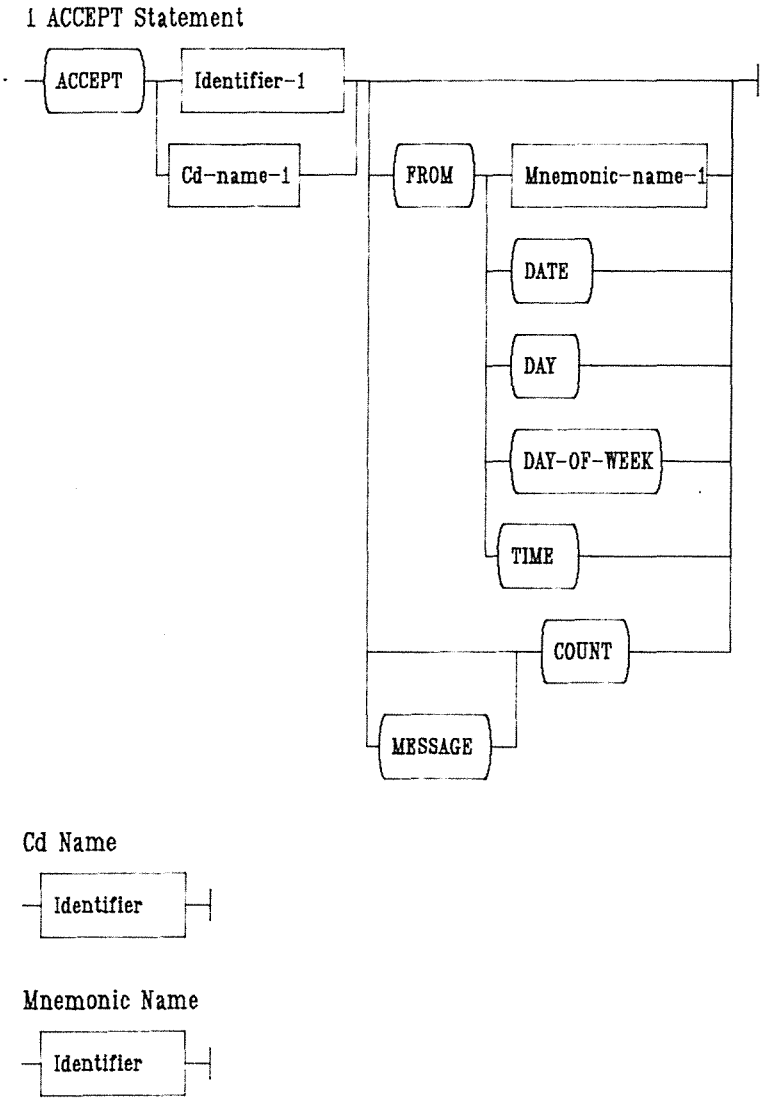
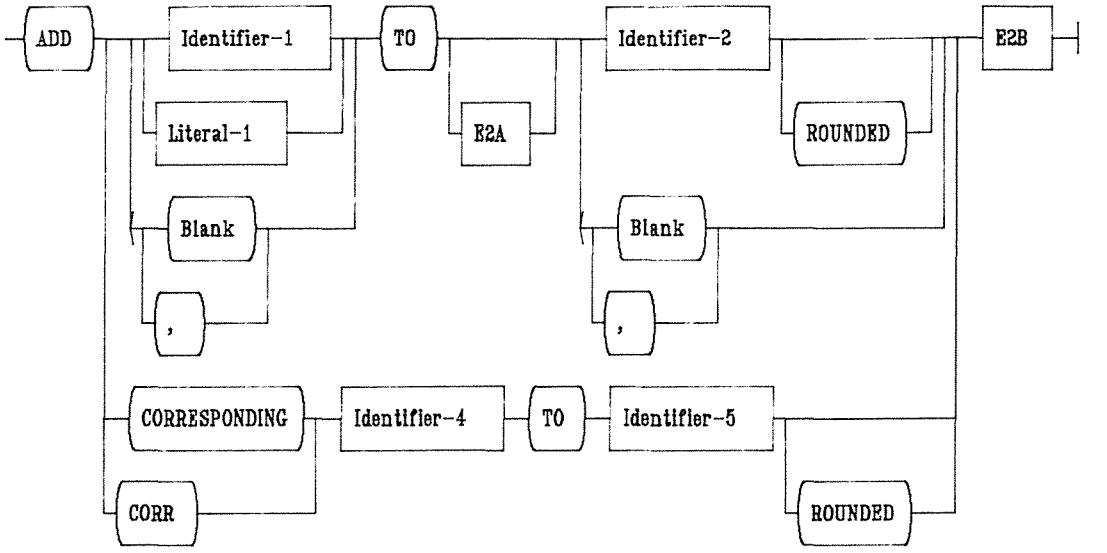
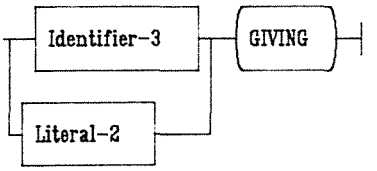


Figure C.7 COBOL Statement Tram-Line Diagram.

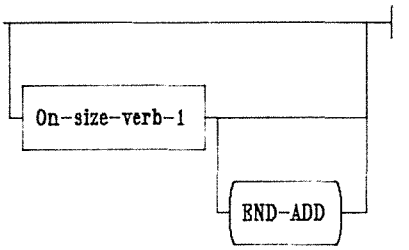
2 ADD Statement



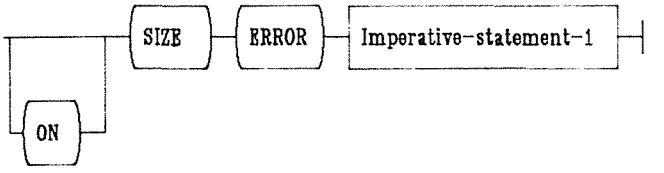
E2A



E2B



On Size Verb



Literal

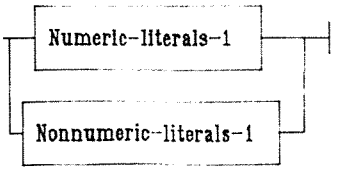
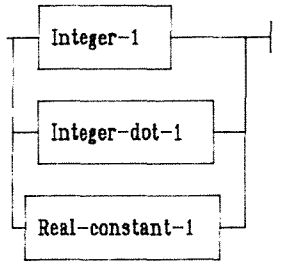
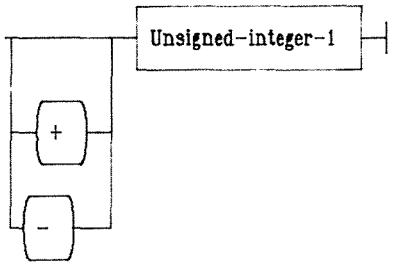


Figure C.8 COBOL Statement Tram-Line Diagram (Continue).

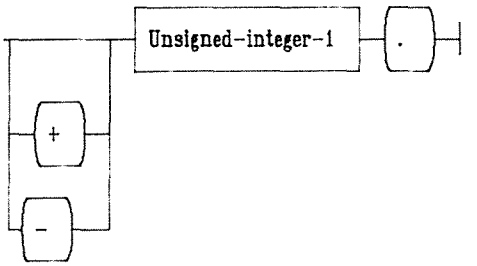
Numeric Literals



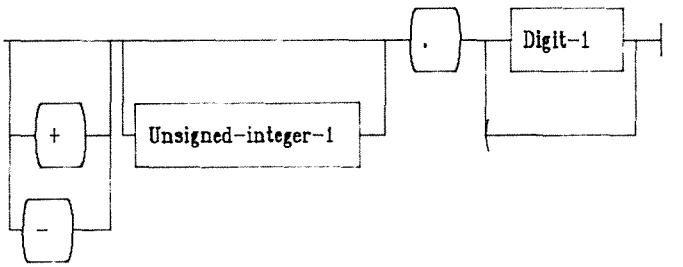
Integer



Integer Dot



Real Constant



Unsigned Integer

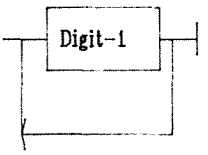
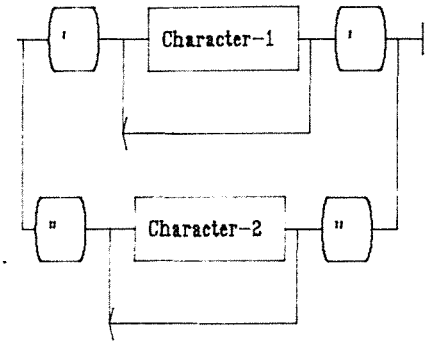
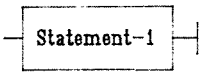


Figure C.9 COBOL Statement Tram-Line Diagram (Continue).

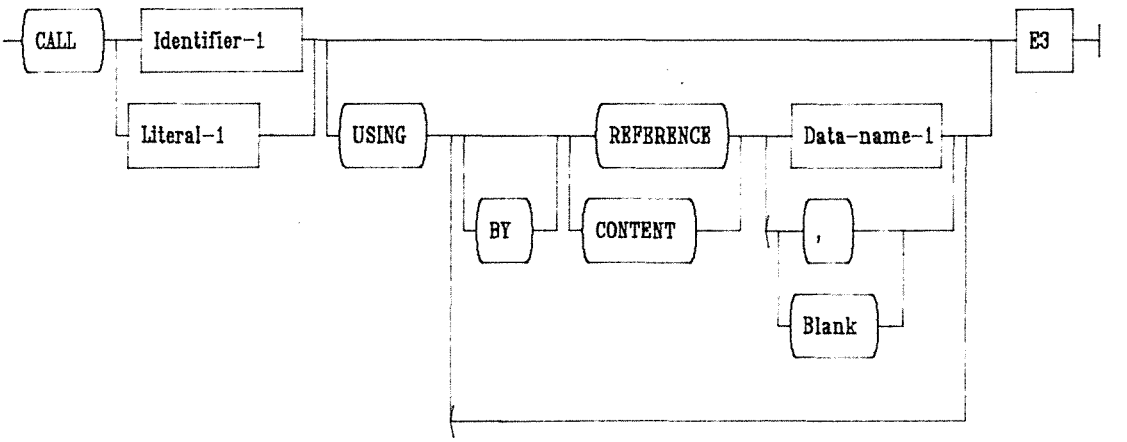
Nonnumeric Literals



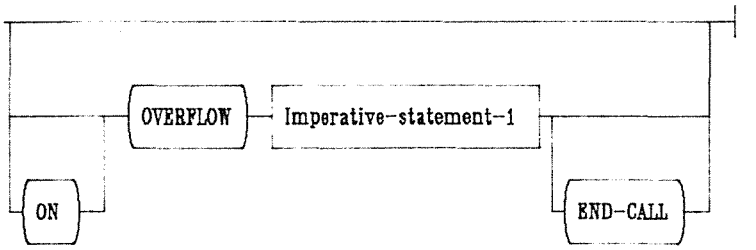
Imperative Statement



3 CALL Statement



E3



4 CANCEL Statement

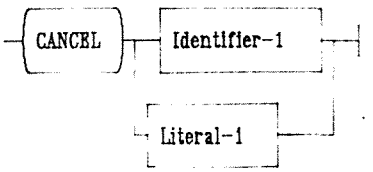
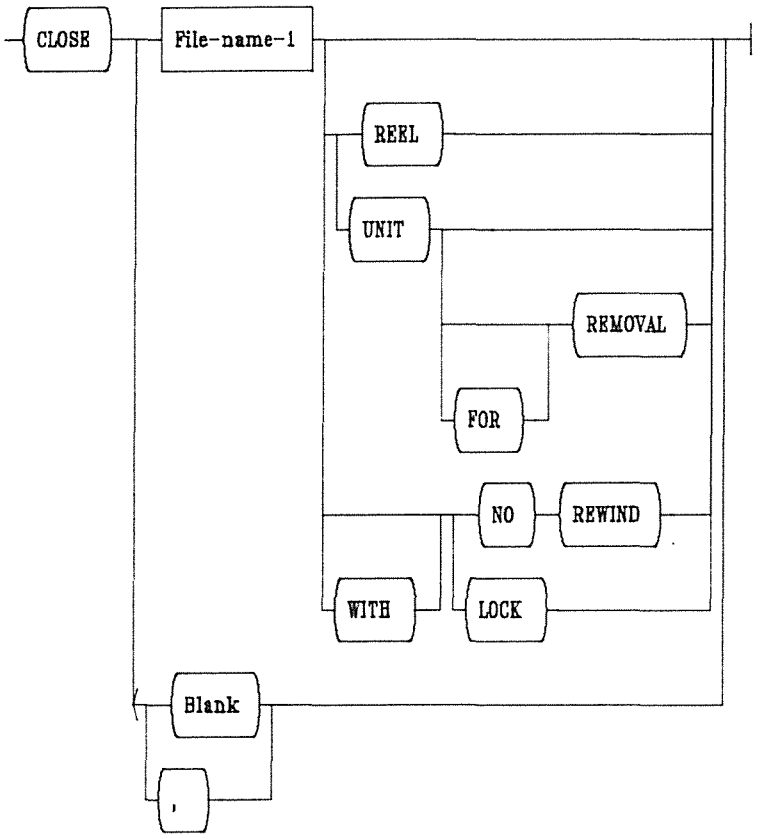
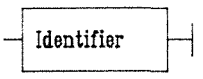


Figure C.10 COBOL Statement Tram-Line Diagram (Continue).

5 CLOSE Statement



File-name



6 COMPUTE Statement

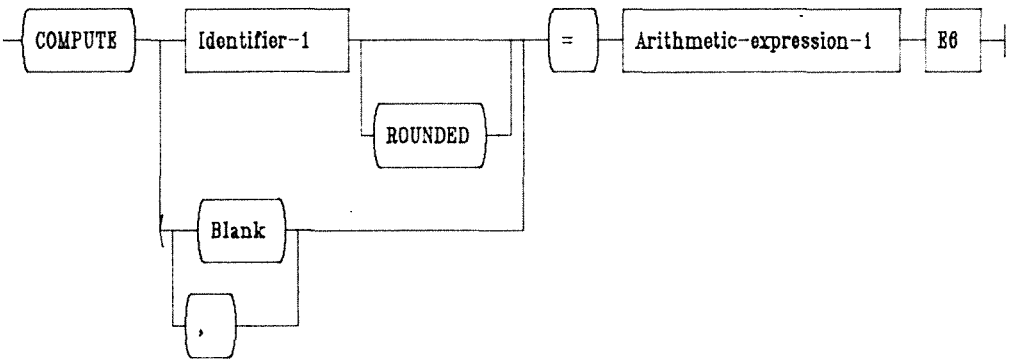
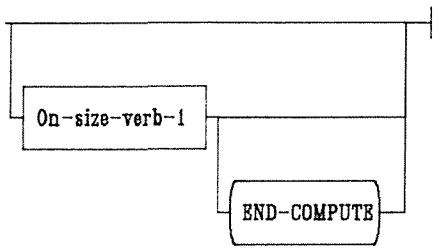
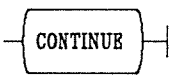


Figure C.11 COBOL Statement Tram-Line Diagram (Continue).

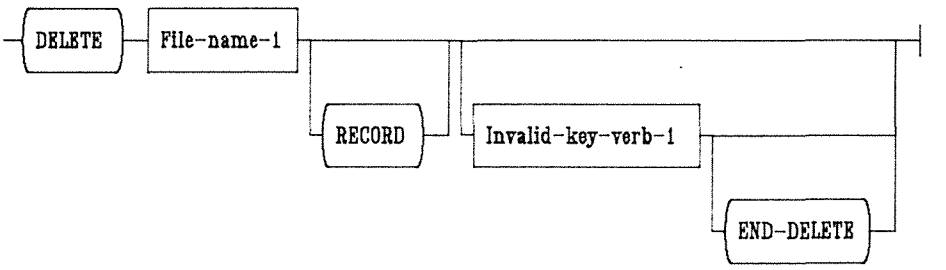
E6



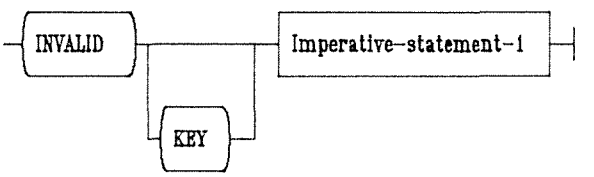
7 CONTINUE Statement



8 DELETE Statement



Invalid-key-verb



9 DISABLE Statement

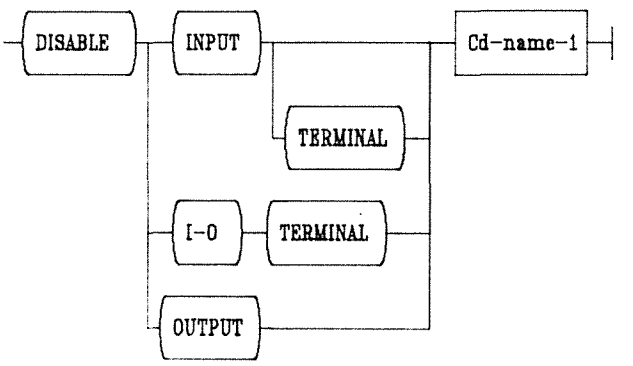
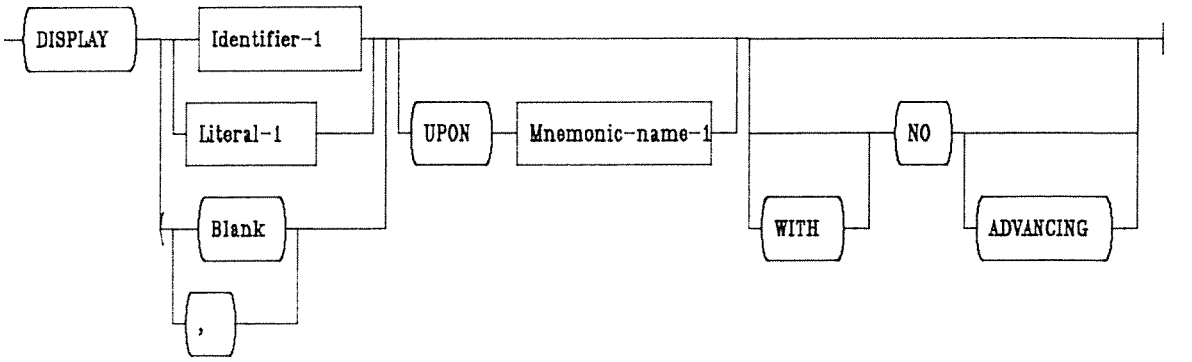
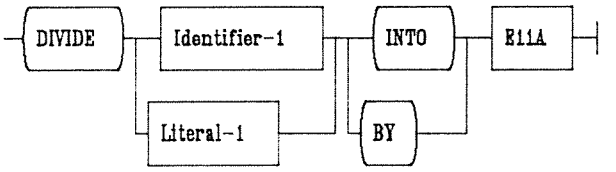


Figure C.12 COBOL Statement Tram-Line Diagram (Continue).

10 DISPLAY Statement



11 DIVIDE Statement



11A

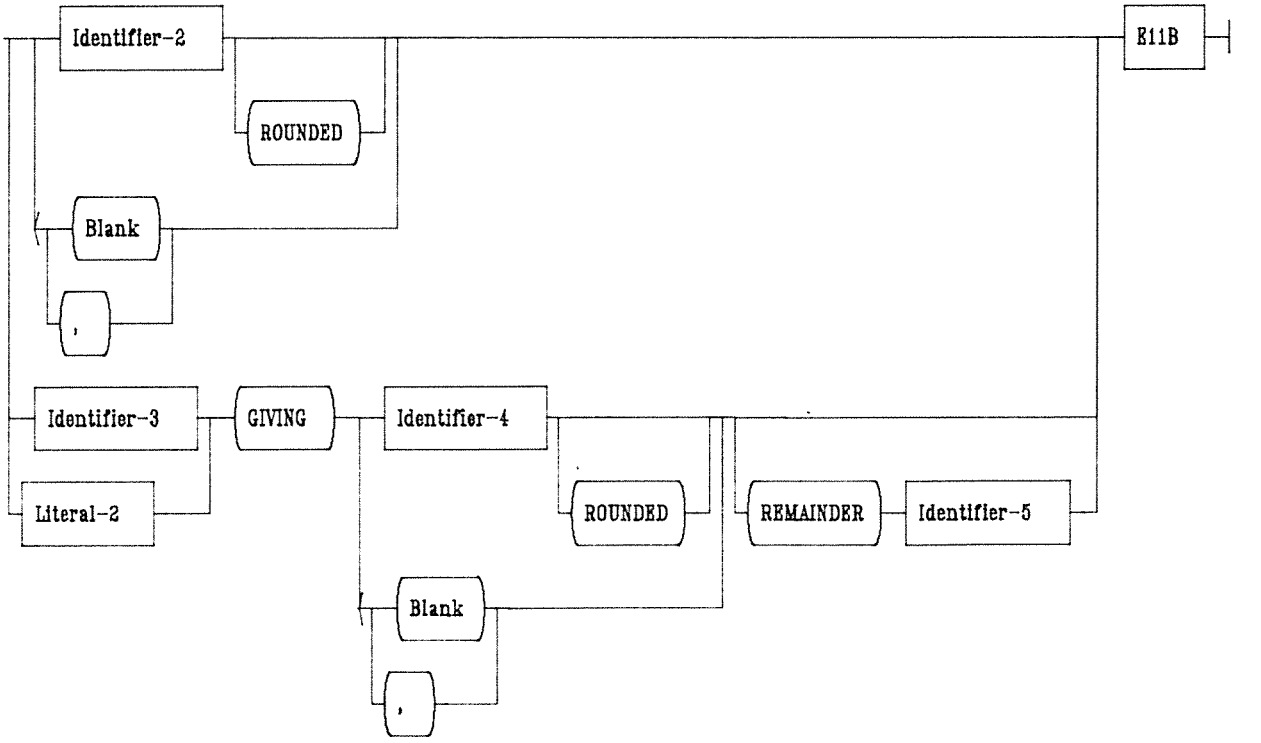
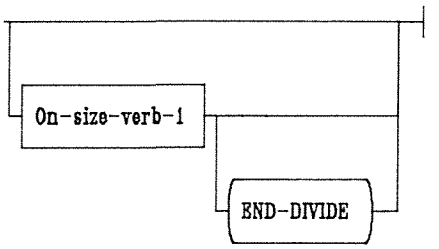
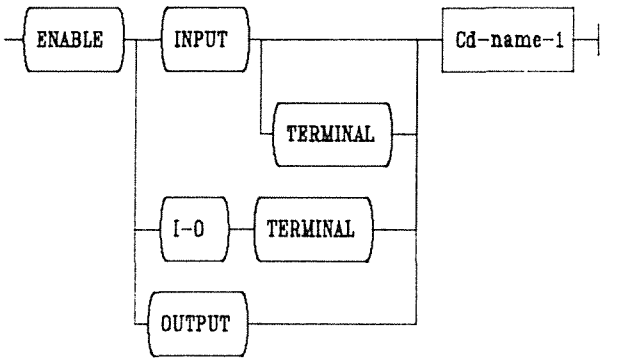


Figure C.13 COBOL Statement Tram-Line Diagram (Continue).

E11B



12 ENABLE Statement



13 EVALUATE Statement

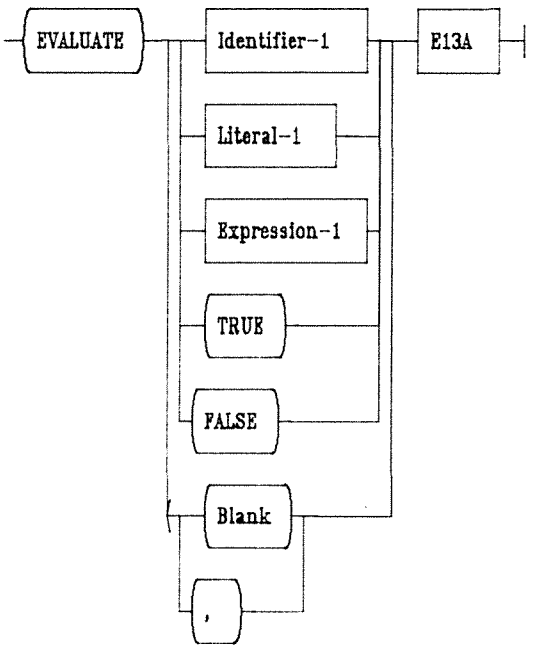
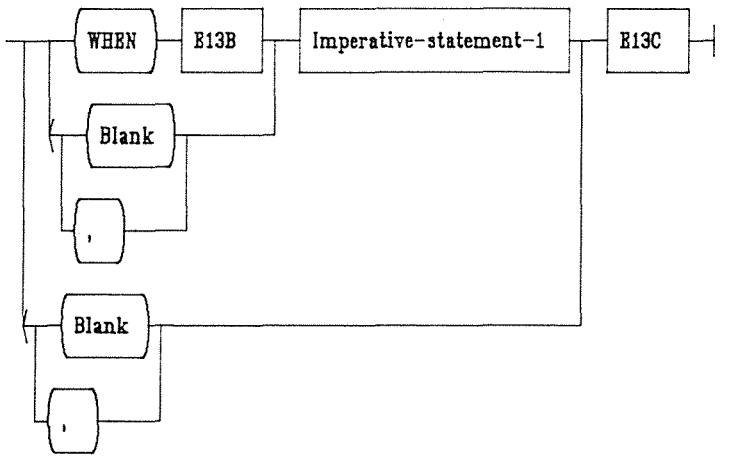
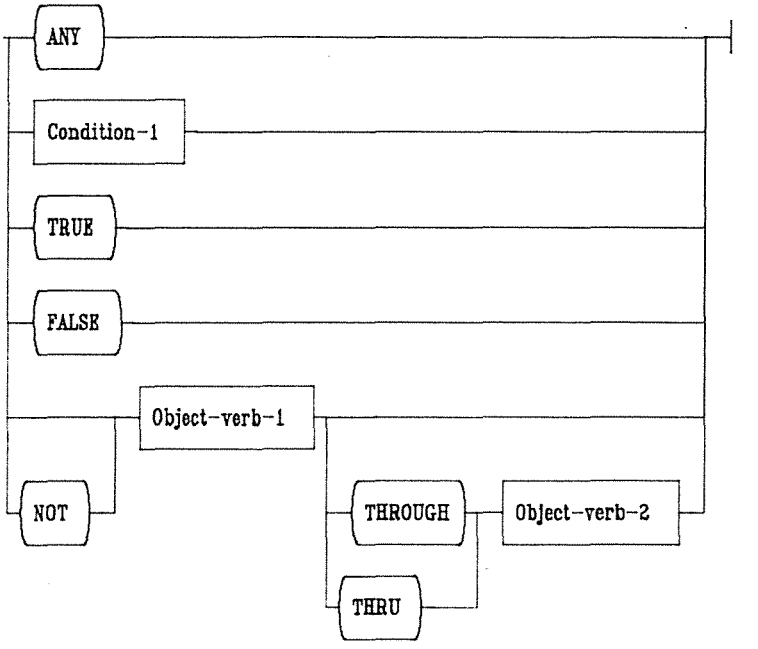


Figure C.14 COBOL Statement Tram-Line Diagram (Continue).

E13A



E13B



E13C

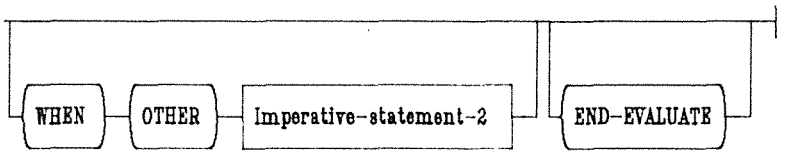
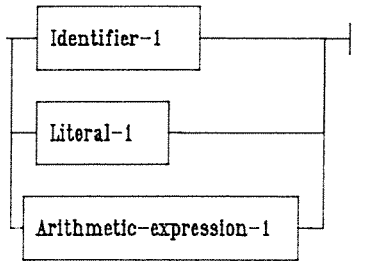
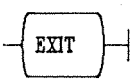


Figure C.15 COBOL Statement Tram-Line Diagram (Continue).

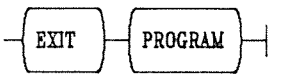
Object-verb



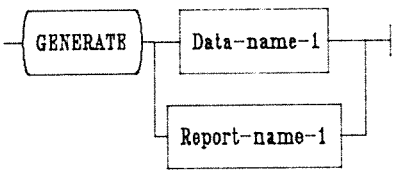
14 EXIT Statement



15 EXIT PROGRAM Statement



16 GENERATE Statement



Report-name

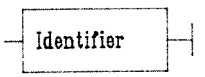
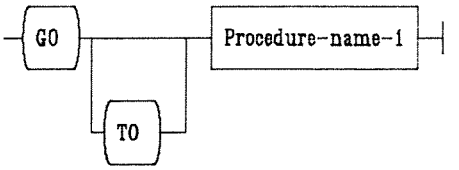
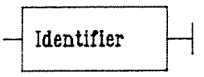


Figure C.16 COBOL Statement Tram-Line Diagram (Continue).

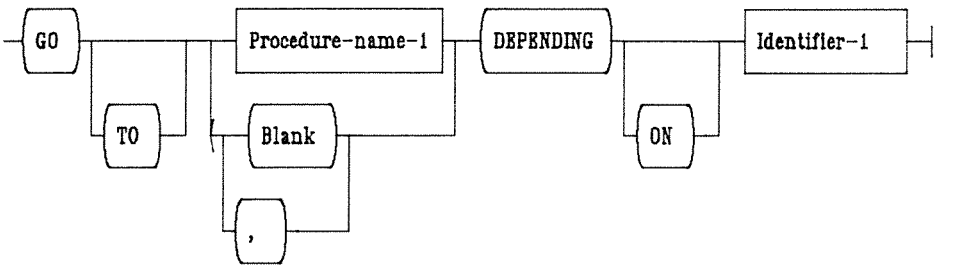
17 GO TO Statement



Procedure-name



18 GO TO DEPENDING Statement



19 IF Statement

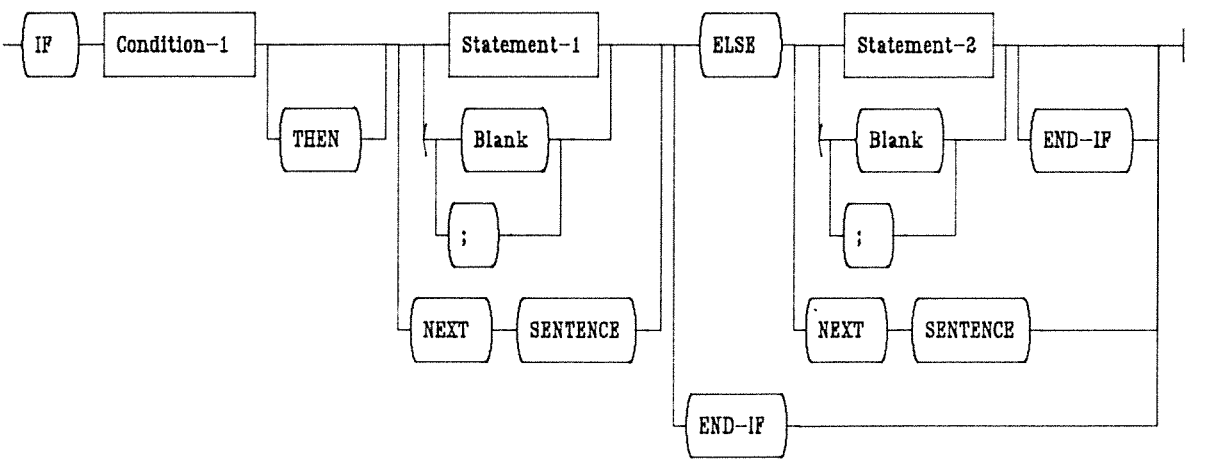
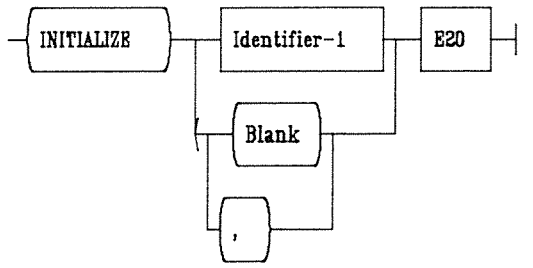
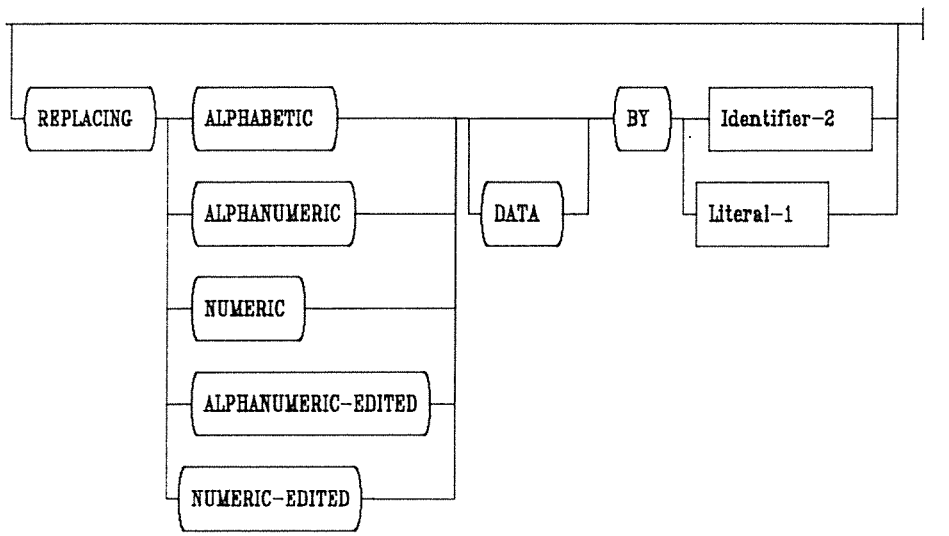


Figure C.17 COBOL Statement Tram-Line Diagram (Continue).

20 INITIALIZE Statement



E20



21 INITIATE Statement

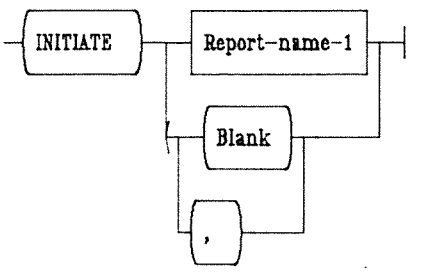
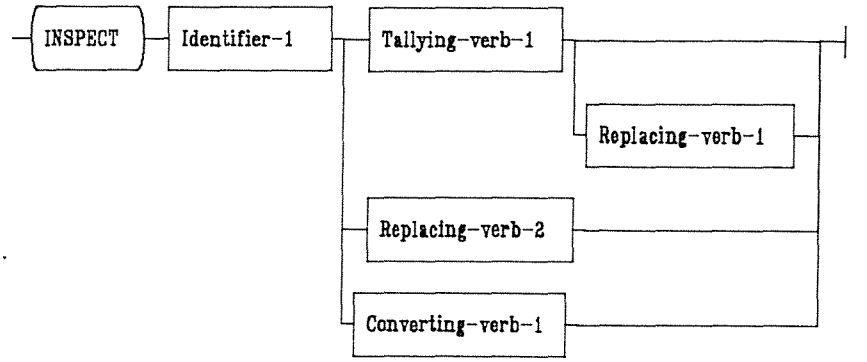
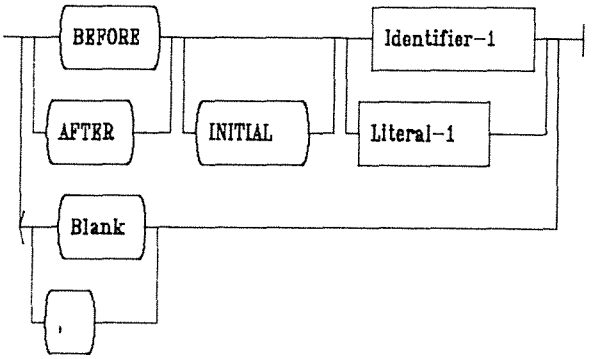


Figure C.18 COBOL Statement Tram-Line Diagram (Continue).

22 INSPECT Statement



Initial-verb



Tallying-verb

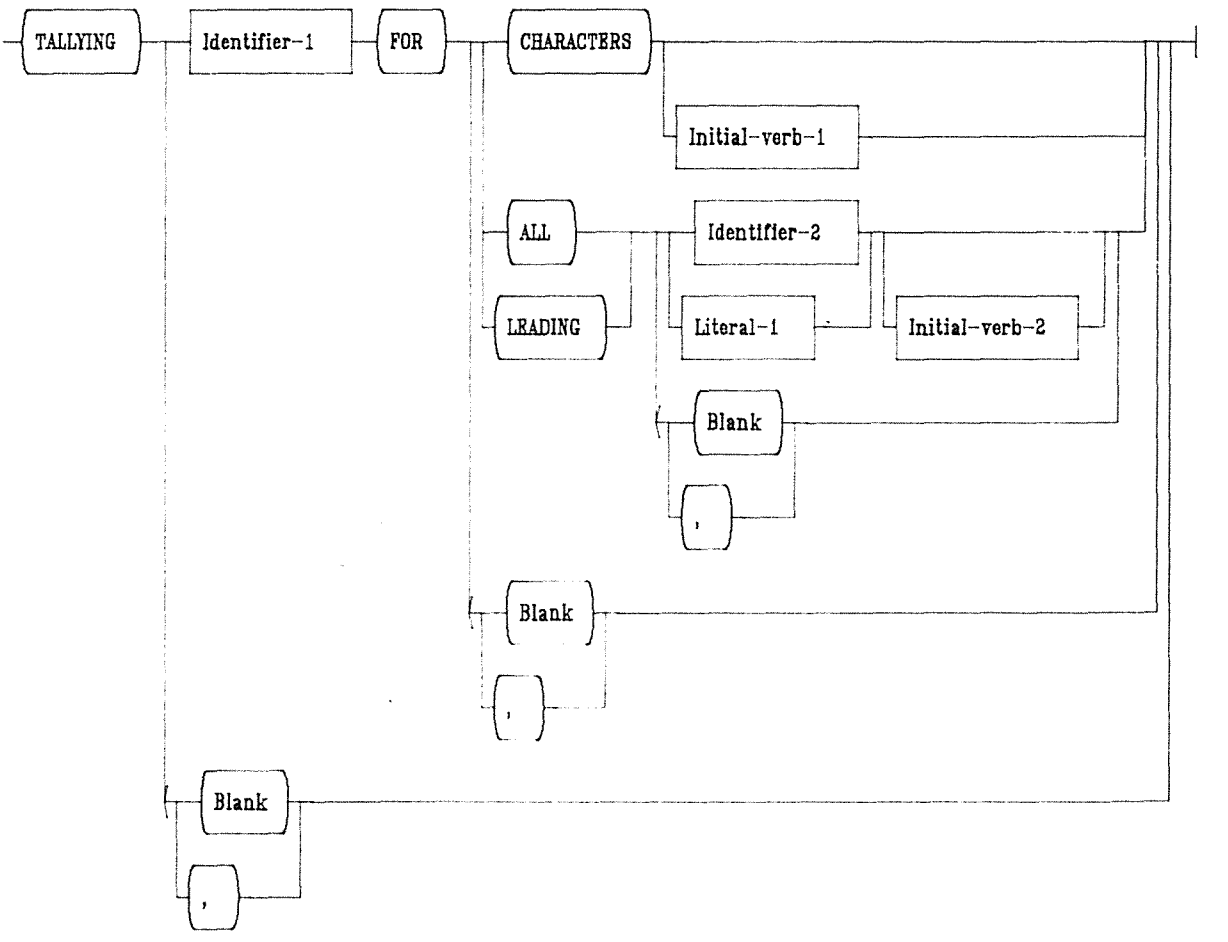
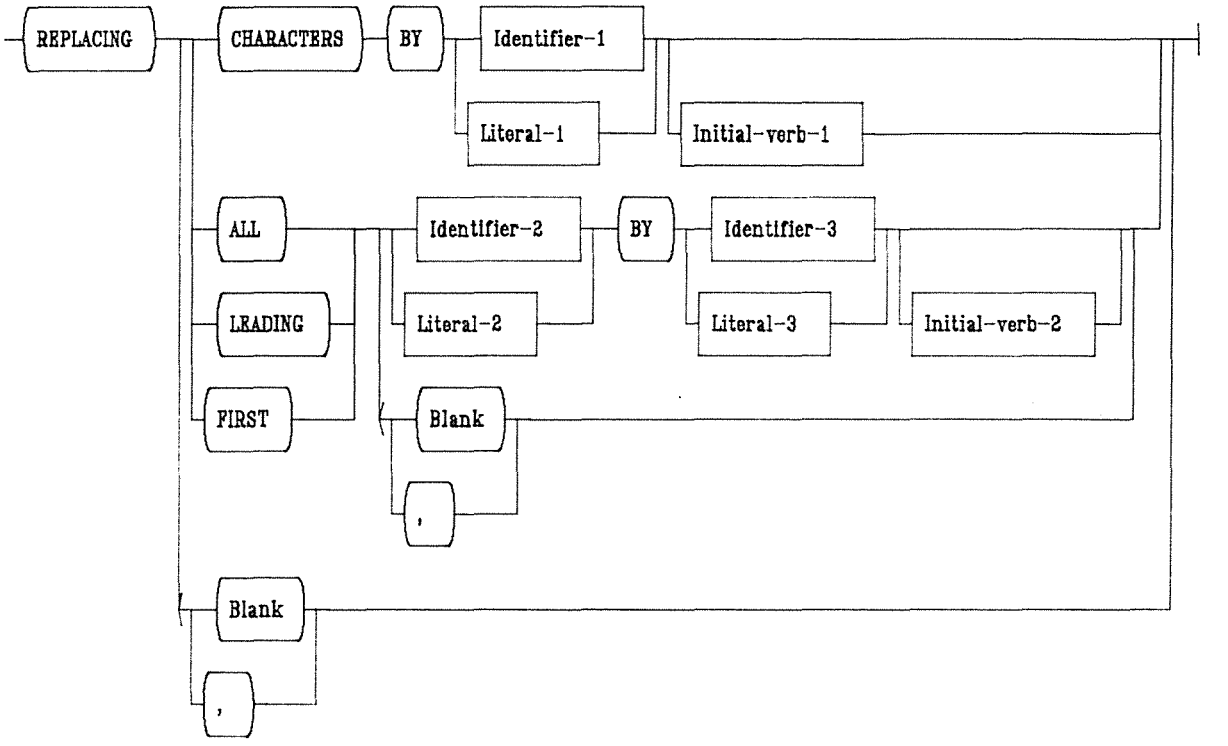


Figure C.19 COBOL Statement Tram-Line Diagram (Continue).

. Replacing-verb



Converting-verb

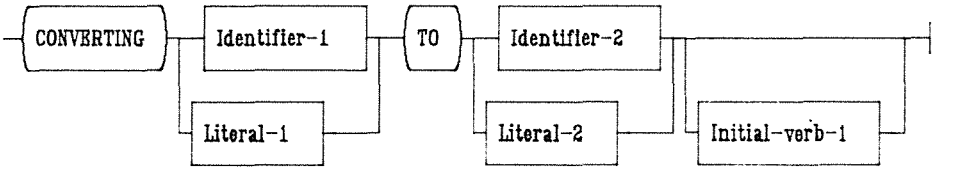
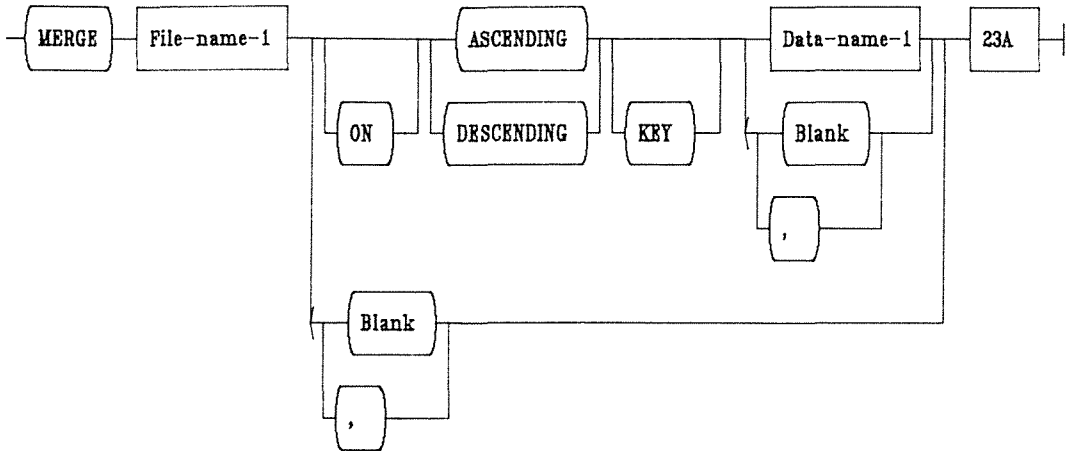
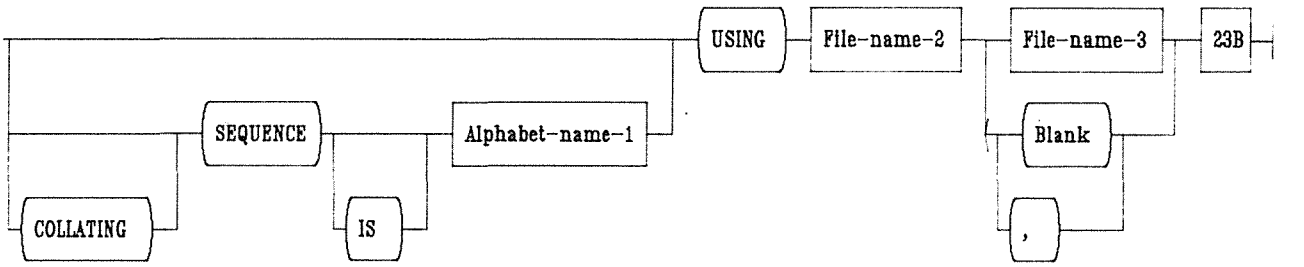


Figure C.20 COBOL Statement Tram-Line Diagram (Continue).

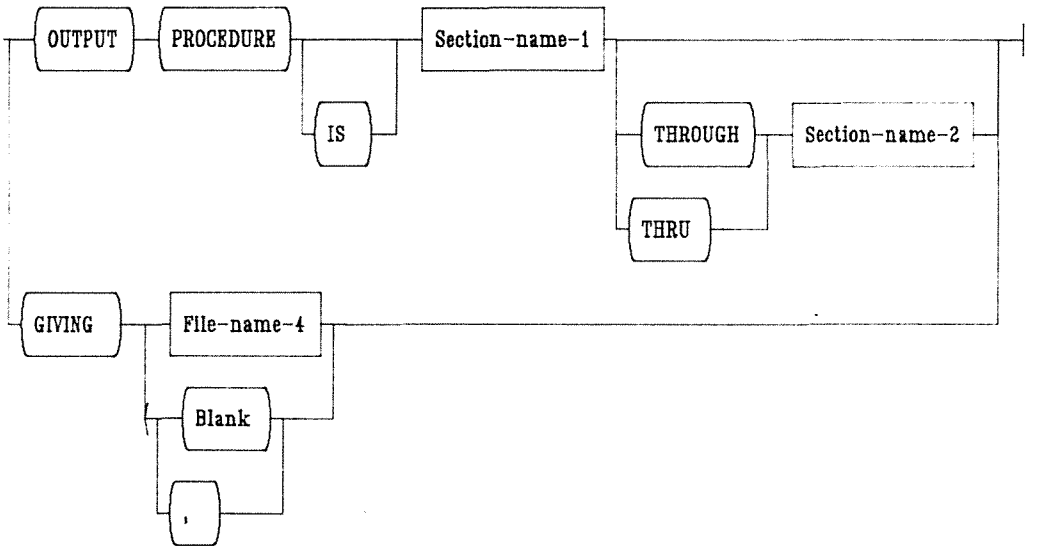
23 MERGE Statement



23A



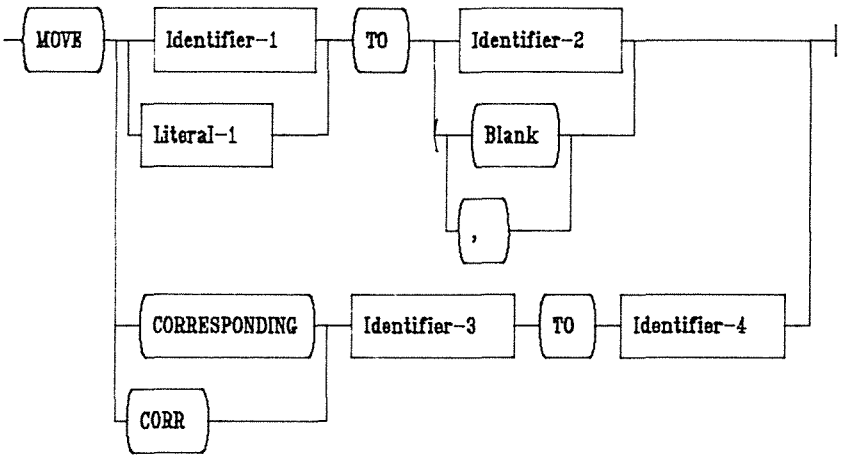
23B



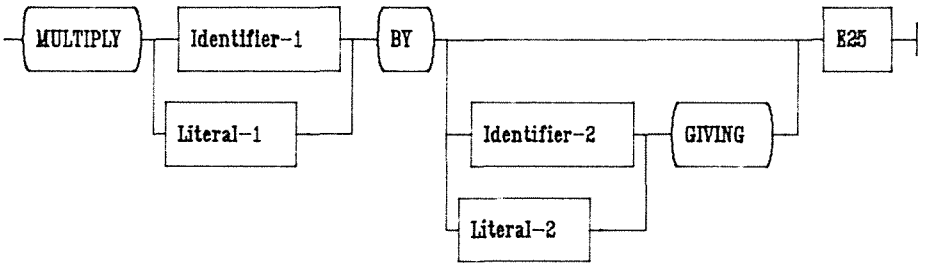
Alphabet-name

Figure C.21 COBOL Statement Tram-Line Diagram (Continue).

24 MOVE Statement



25 MULTIPLY Statement



E25

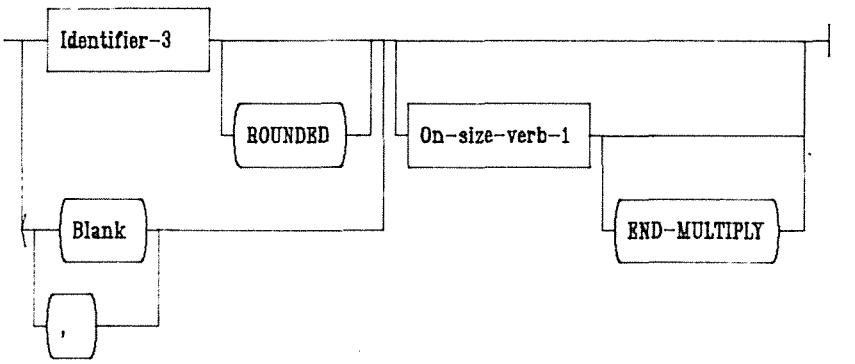
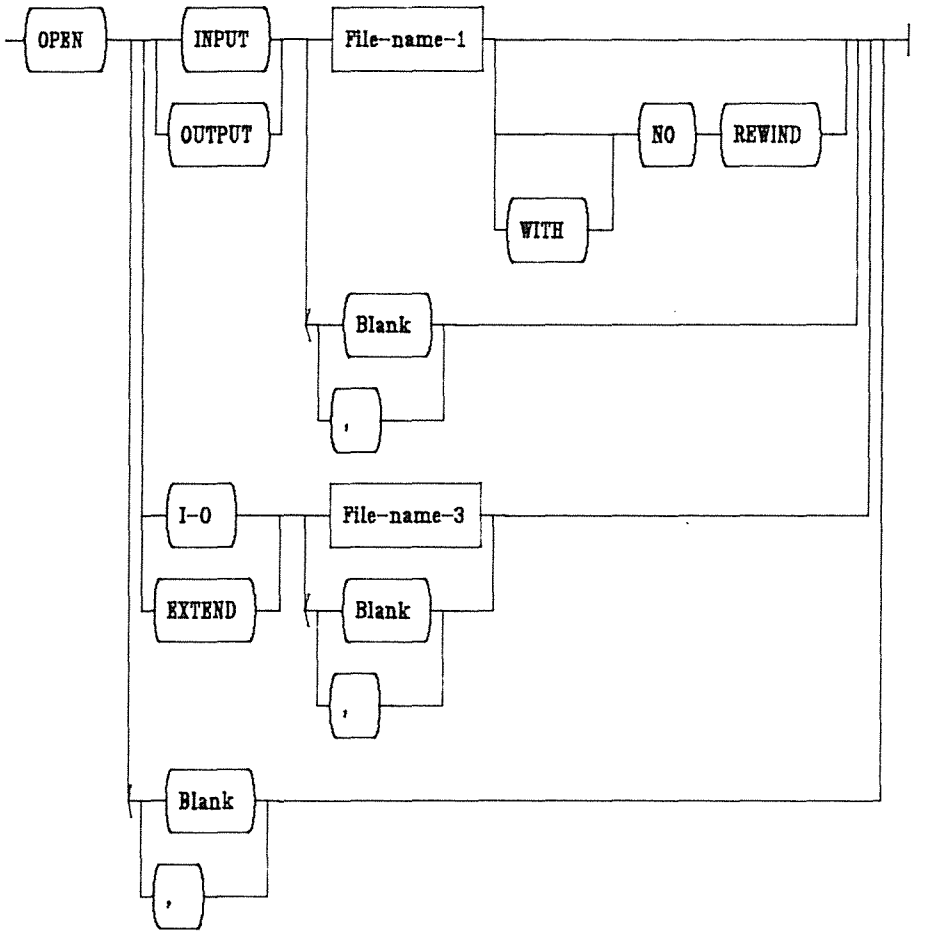


Figure C.22 COBOL Statement Tram-Line Diagram (Continue).

28 OPEN Statement



27 PERFORM Statement

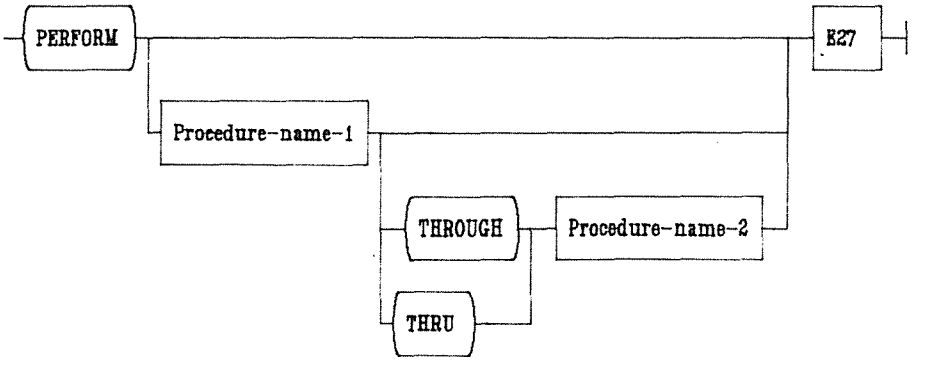
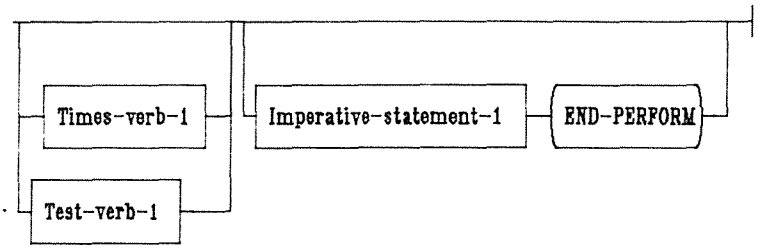
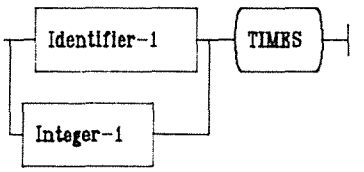


Figure C.23 COBOL Statement Tram-Line Diagram (Continue).

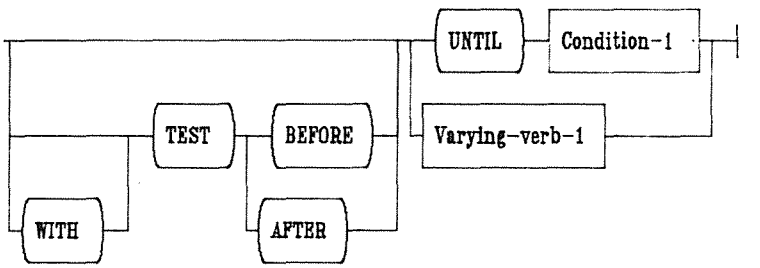
E27



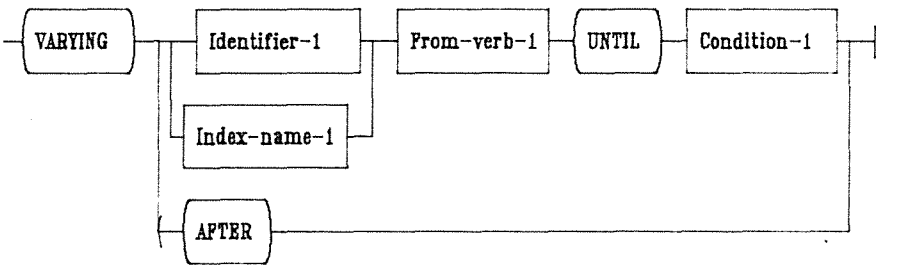
Times-verb



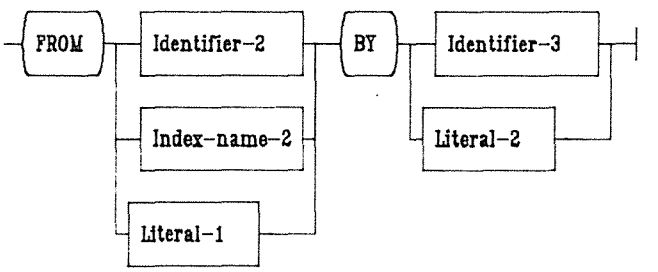
Test-verb



Varying-verb



From-verb



Index-name

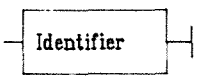
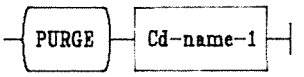
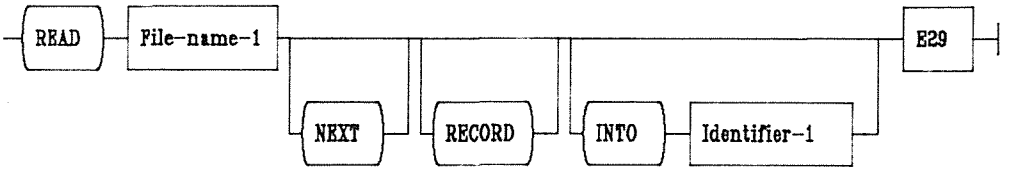


Figure C.24 COBOL Statement Tram-Line Diagram (Continue).

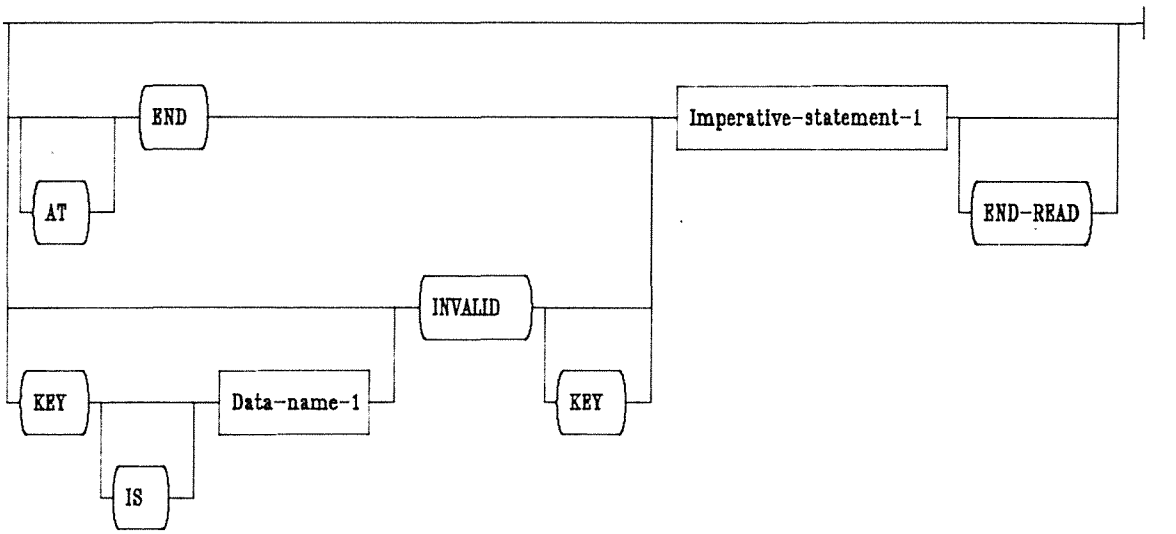
28 PURGE Statement



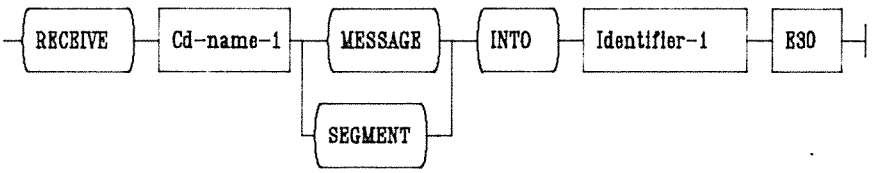
29 READ Statement



E29



30 RECEIVE Statement



E30

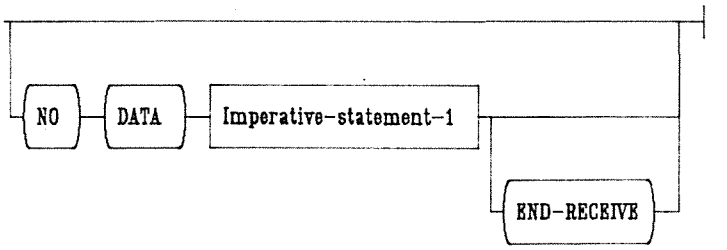
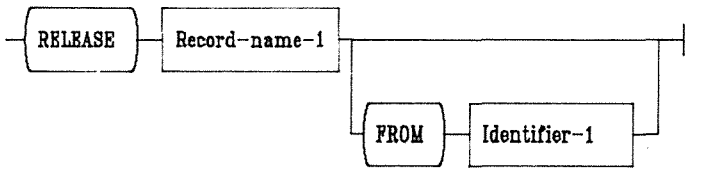
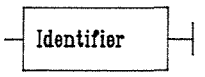


Figure C.25 COBOL Statement Tram-Line Diagram (Continue).

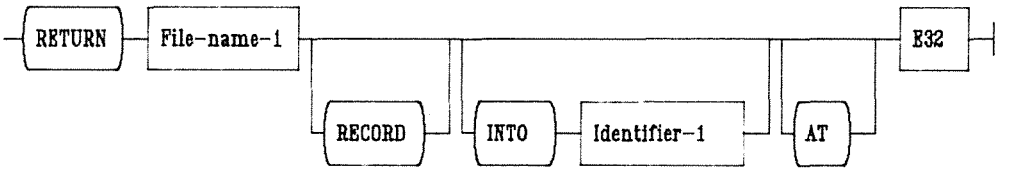
31 RELEASE Statement



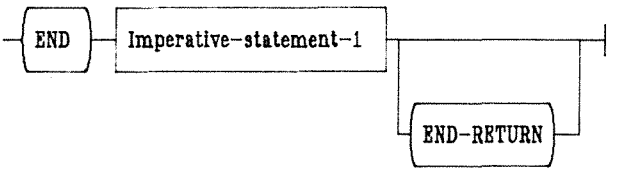
Record-name



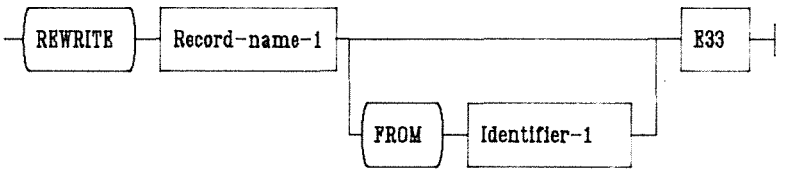
32 RETURN Statement



E32



33 REWRITE Statement



E33

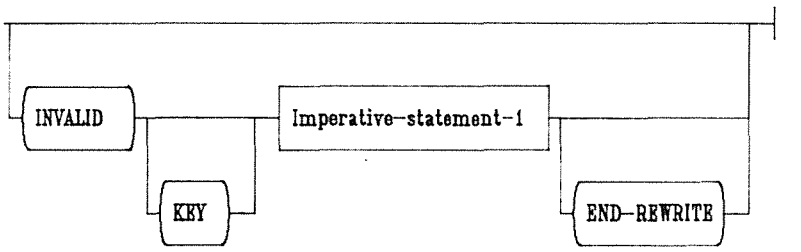
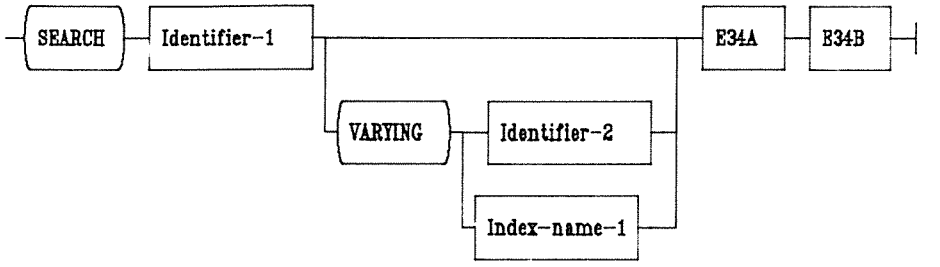
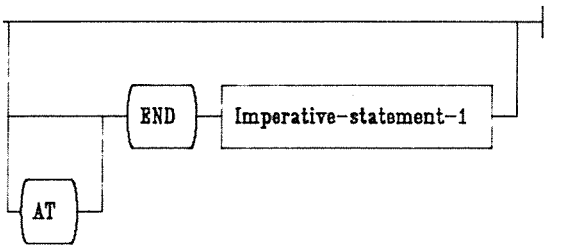


Figure C.26 COBOL Statement Tram-Line Diagram (Continue).

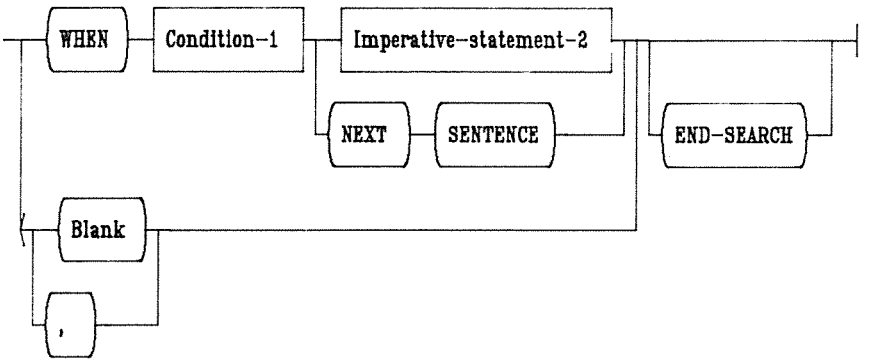
34 SEARCH Statement



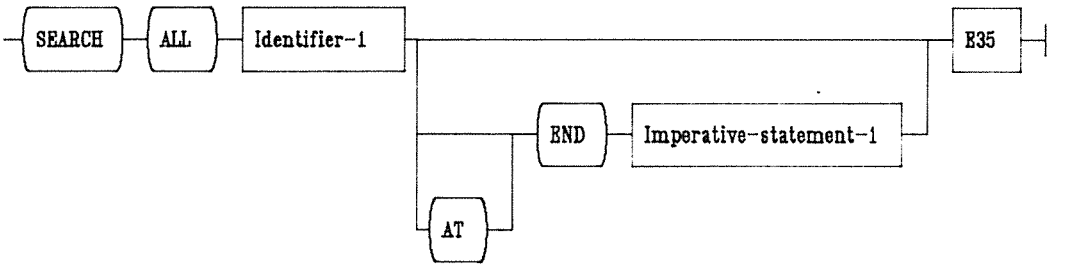
E34A



E34B



35 SEARCH ALL Statement



E35

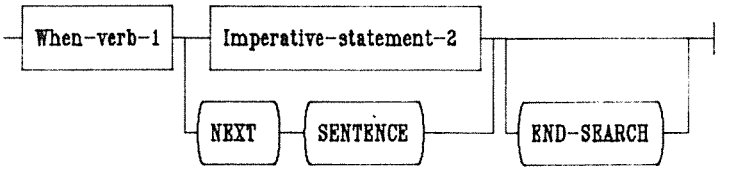
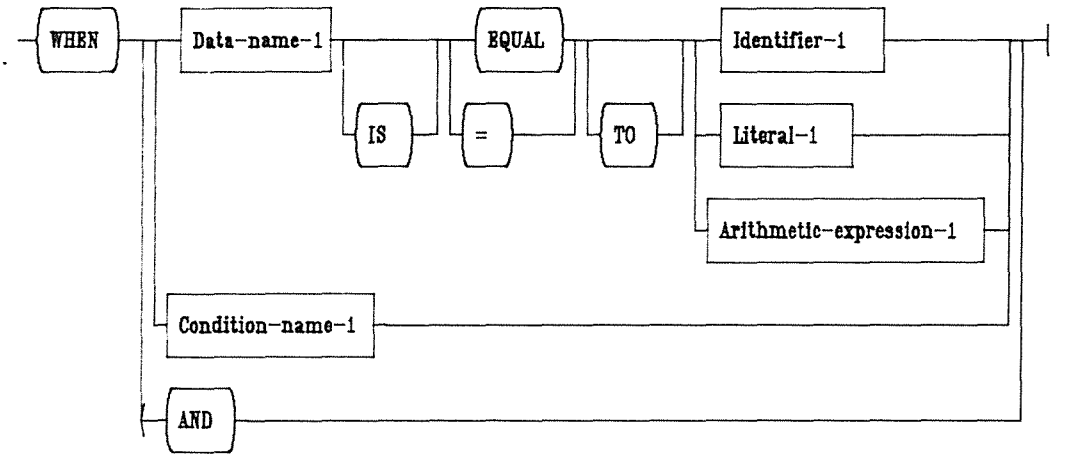
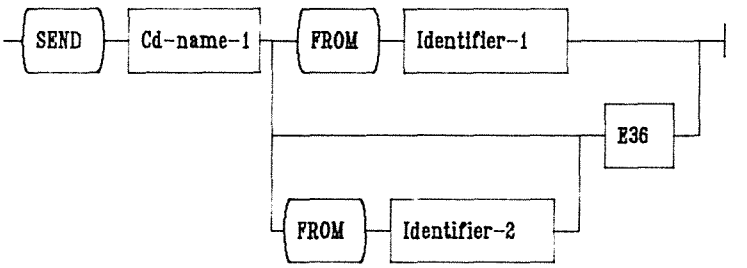


Figure C.27 COBOL Statement Tram-Line Diagram (Continue).

When-verb



36 SEND Statement



E36

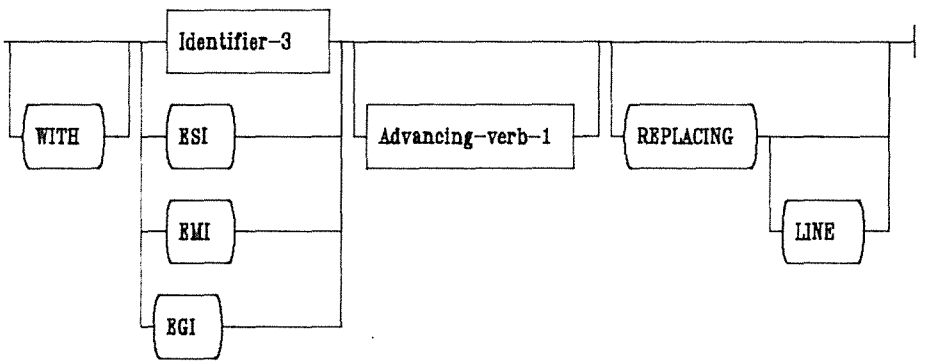
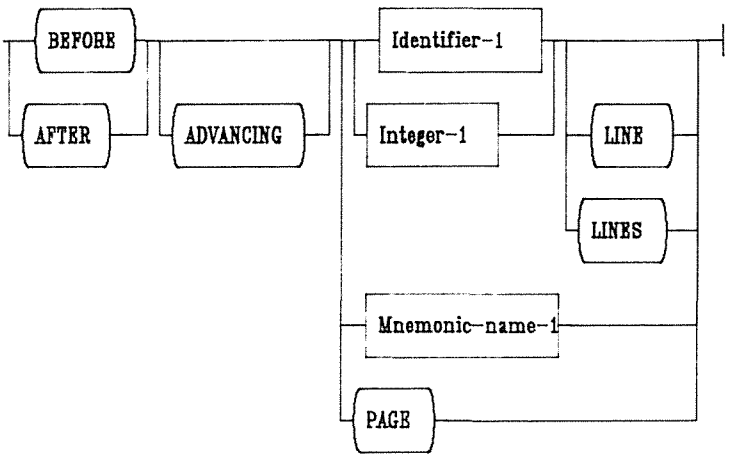


Figure C.28 COBOL Statement Tram-Line Diagram (Continue).

Advancing-verb



37 SET Statement

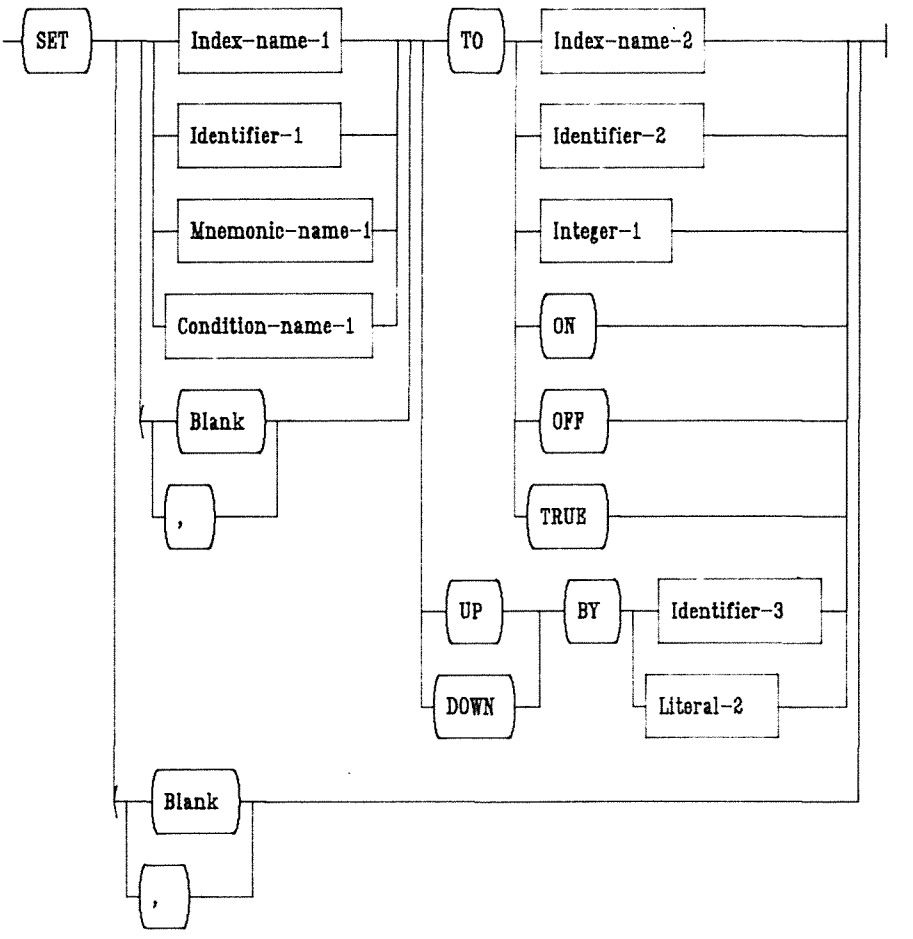
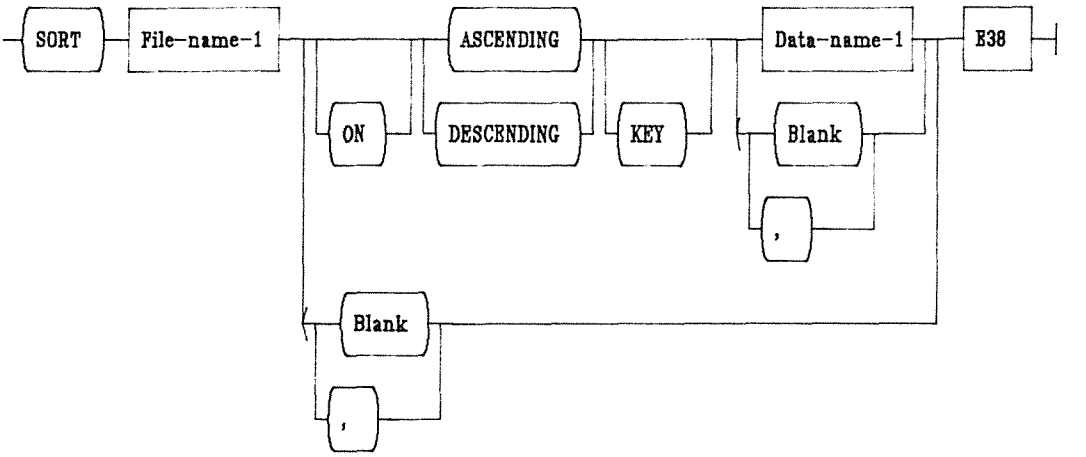
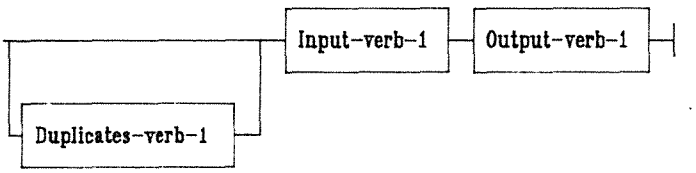


Figure C.29 COBOL Statement Tram-Line Diagram (Continue).

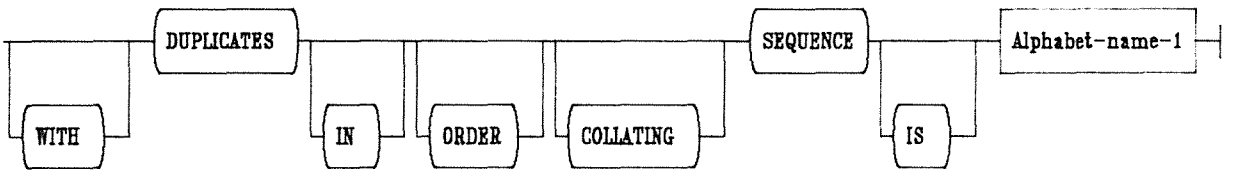
38 SORT Statement



E38



Duplicates-verb



Input-verb

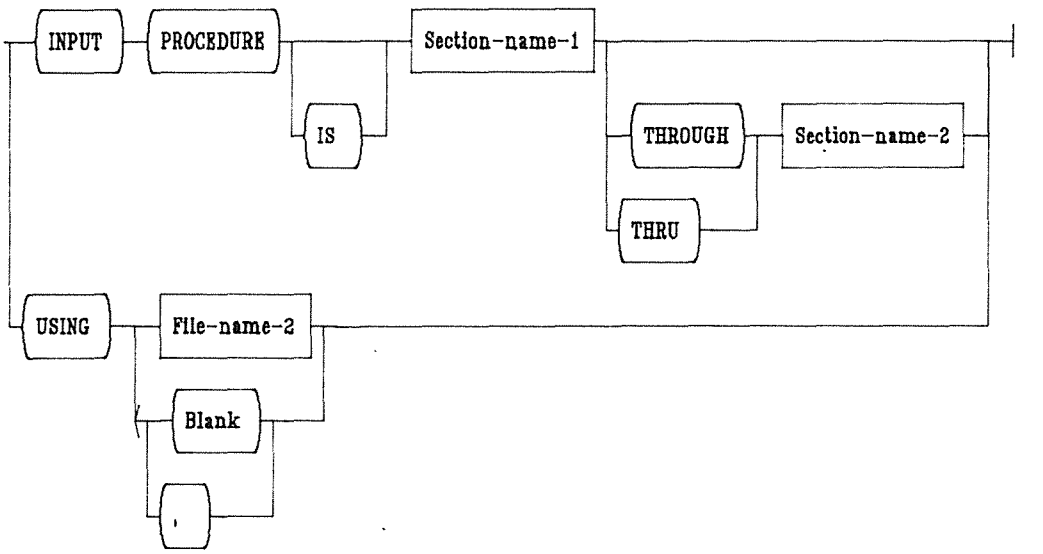
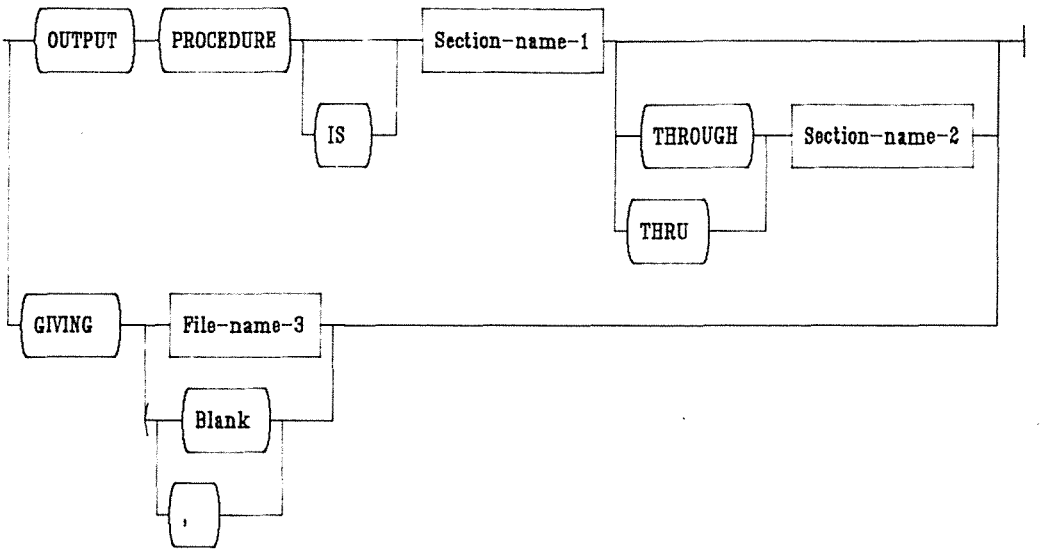


Figure C.30 COBOL Statement Tram-Line Diagram (Continue).

Output-verb



39 START Statement

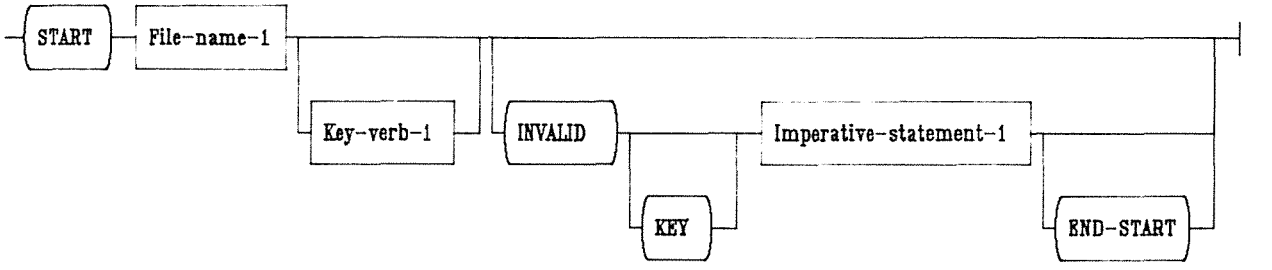
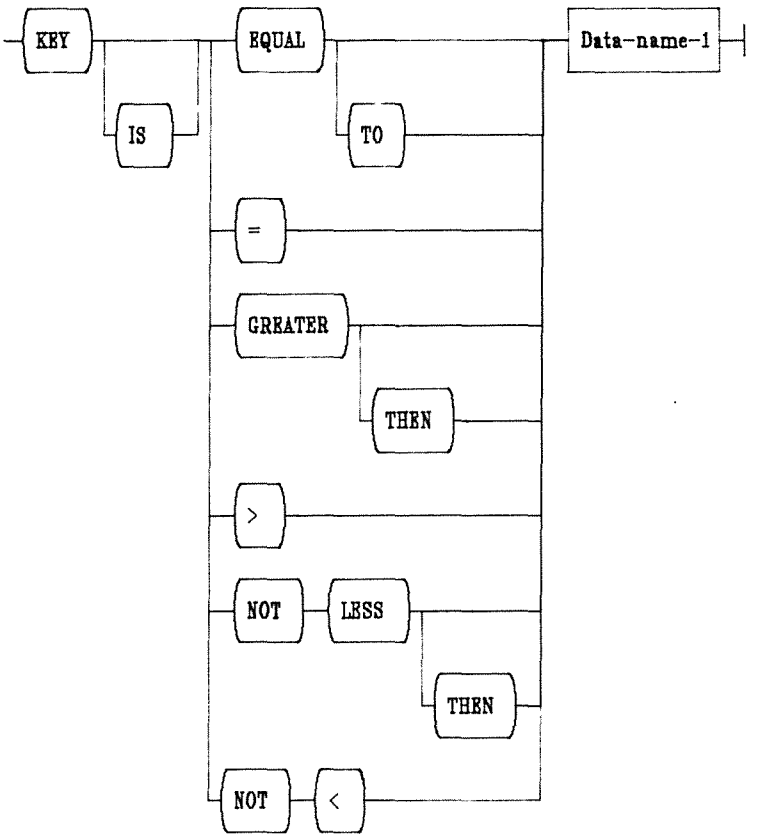


Figure C.31 COBOL Statement Tram-Line Diagram (Continue).

Key-verb



40 STOP Statement

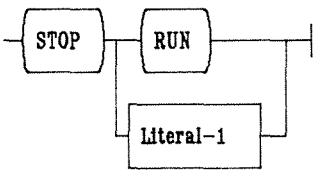
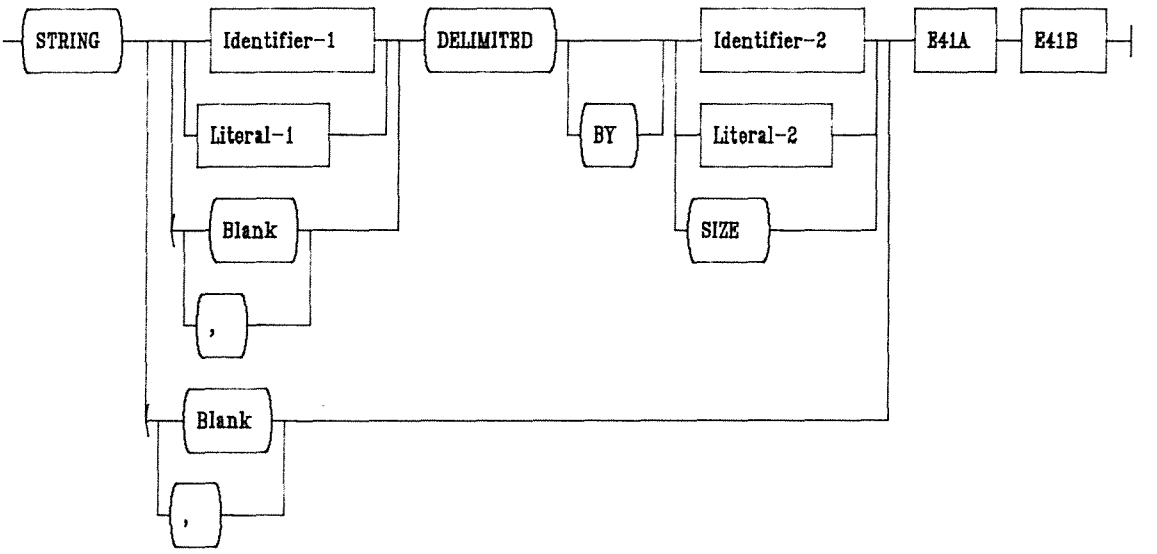
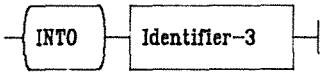


Figure C.32 COBOL Statement Tram-Line Diagram (Continue).

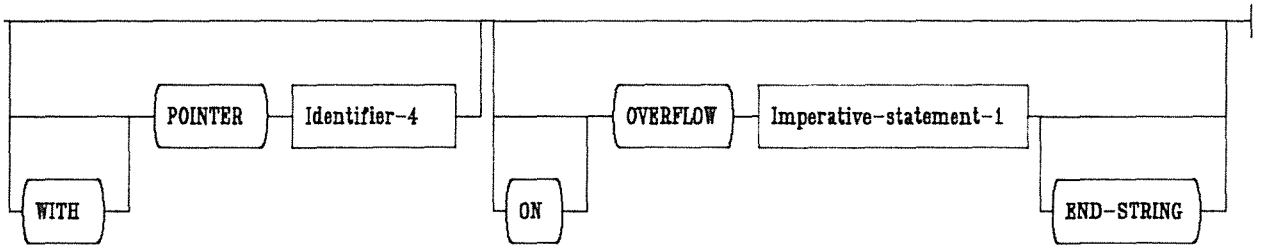
41 STRING Statement



E41A



E41B



42 SUBTRACT Statement

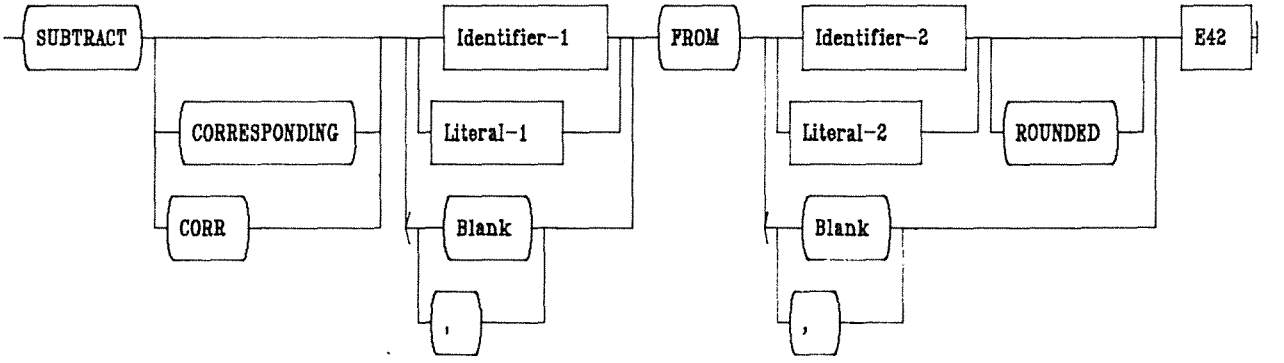
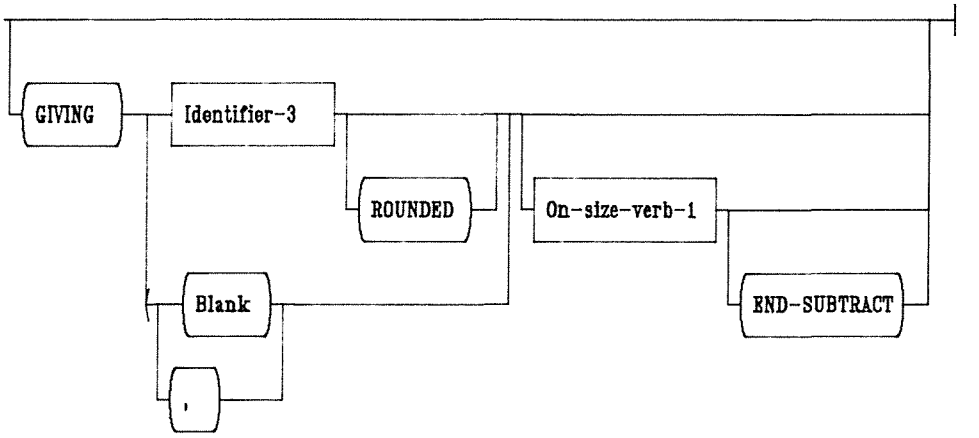
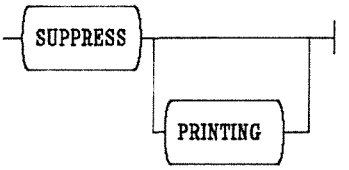


Figure C.33 COBOL Statement Tram-Line Diagram (Continue).

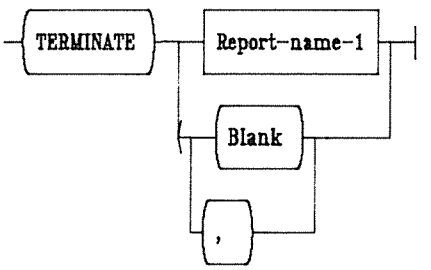
E42



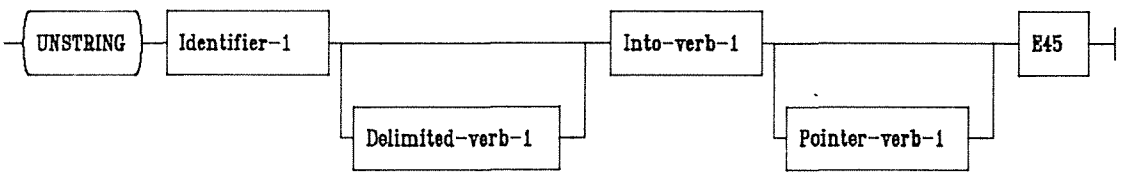
43 SUPPRESS Statement



44 TERMINATE Statement



45 UNSTRING Statement



E45

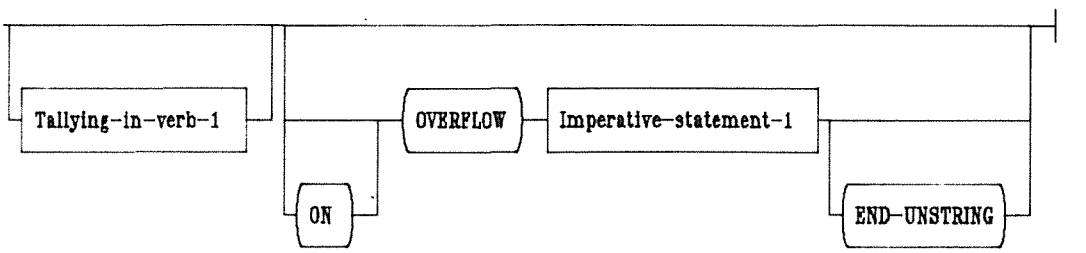
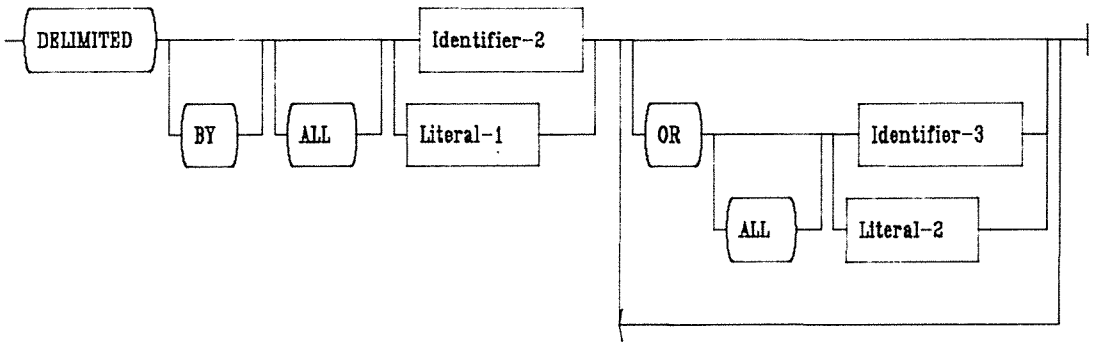
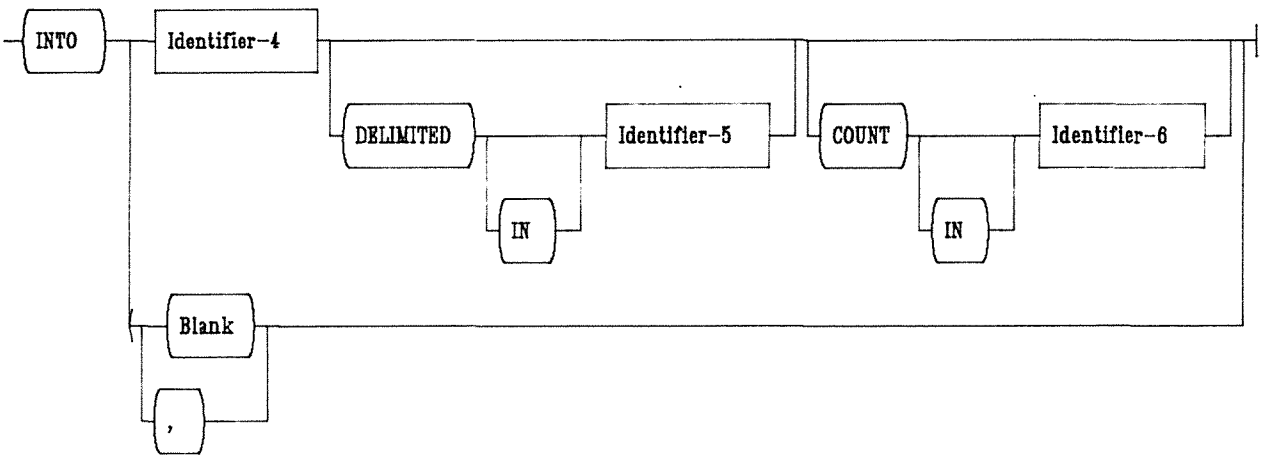


Figure C.34 COBOL Statement Tram-Line Diagram (Continue).

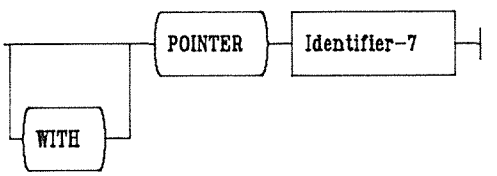
Delimited-verb



Into-verb



Pointer-verb



Tallying-in-verb

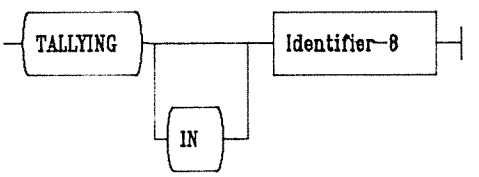
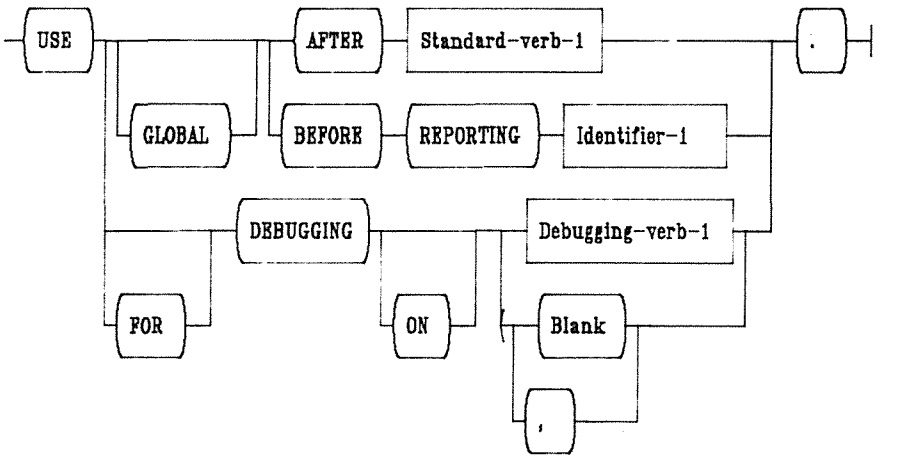


Figure C.35 COBOL Statement Tram-Line Diagram (Continue).

46 USE Statement



Standard-verb

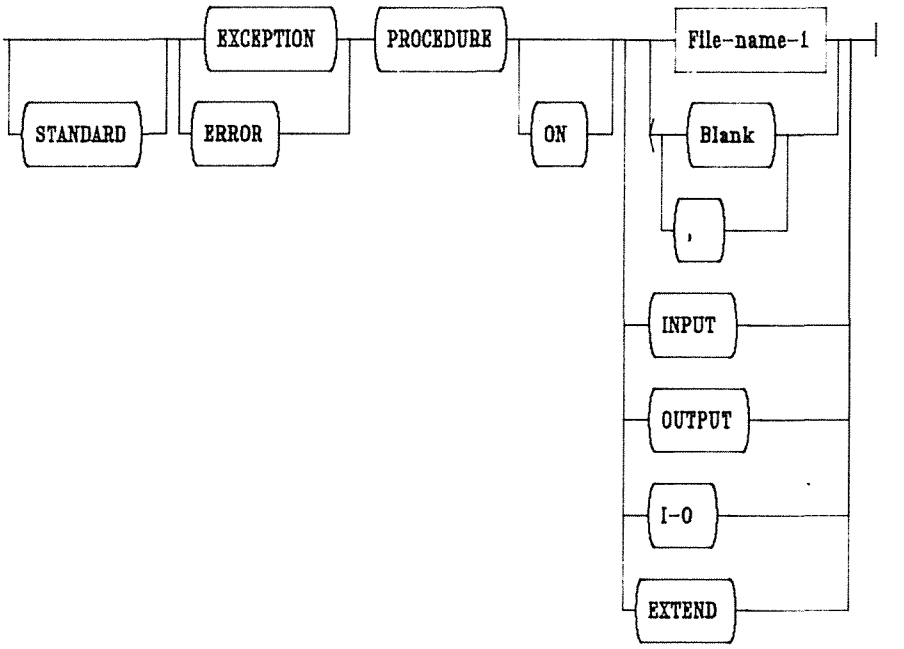
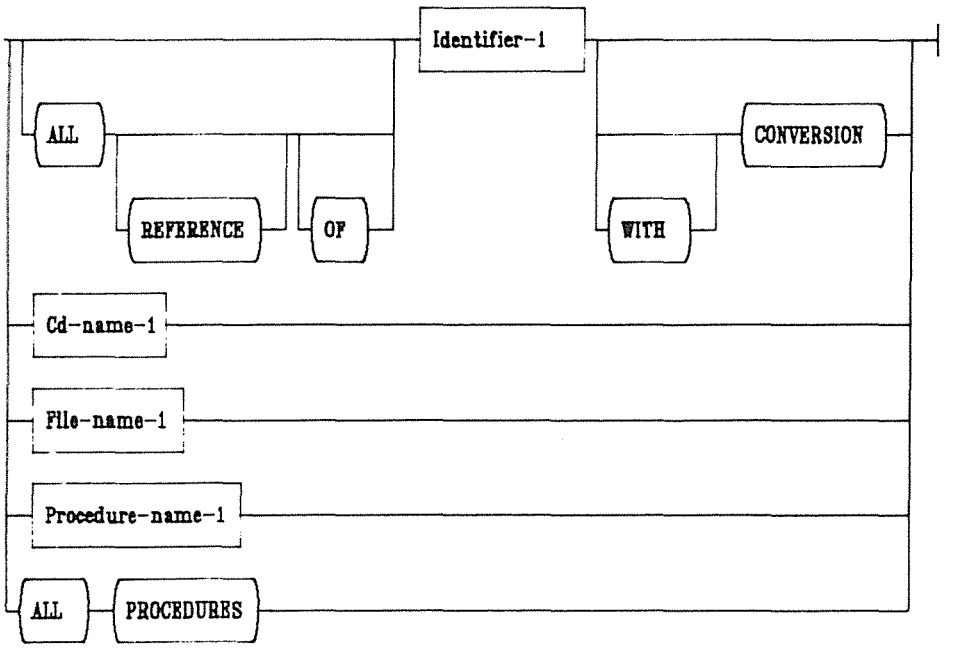
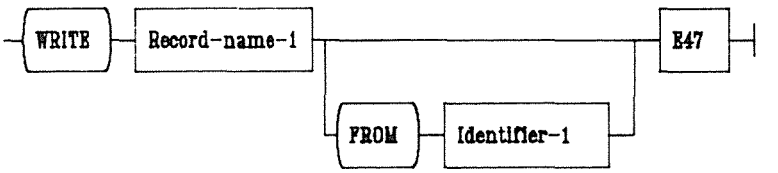


Figure C.36 COBOL Statement Tram-Line Diagram (Continue).

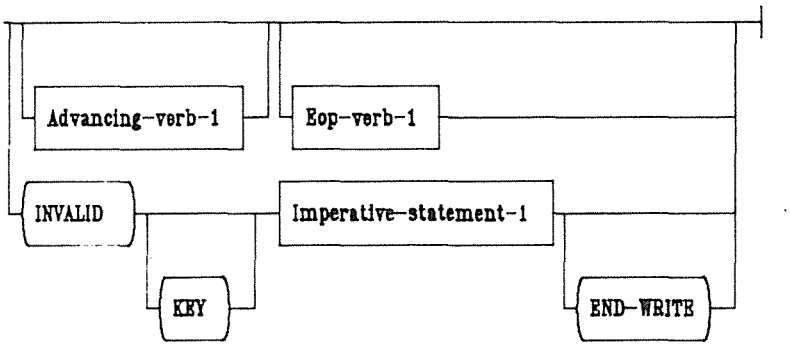
Debugging-verb



47 WRITE Statement



E47



Eop-verb

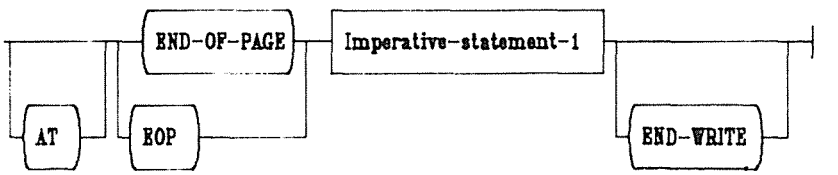


Figure C.37 COBOL Statement Tram-Line Diagram (Continue).

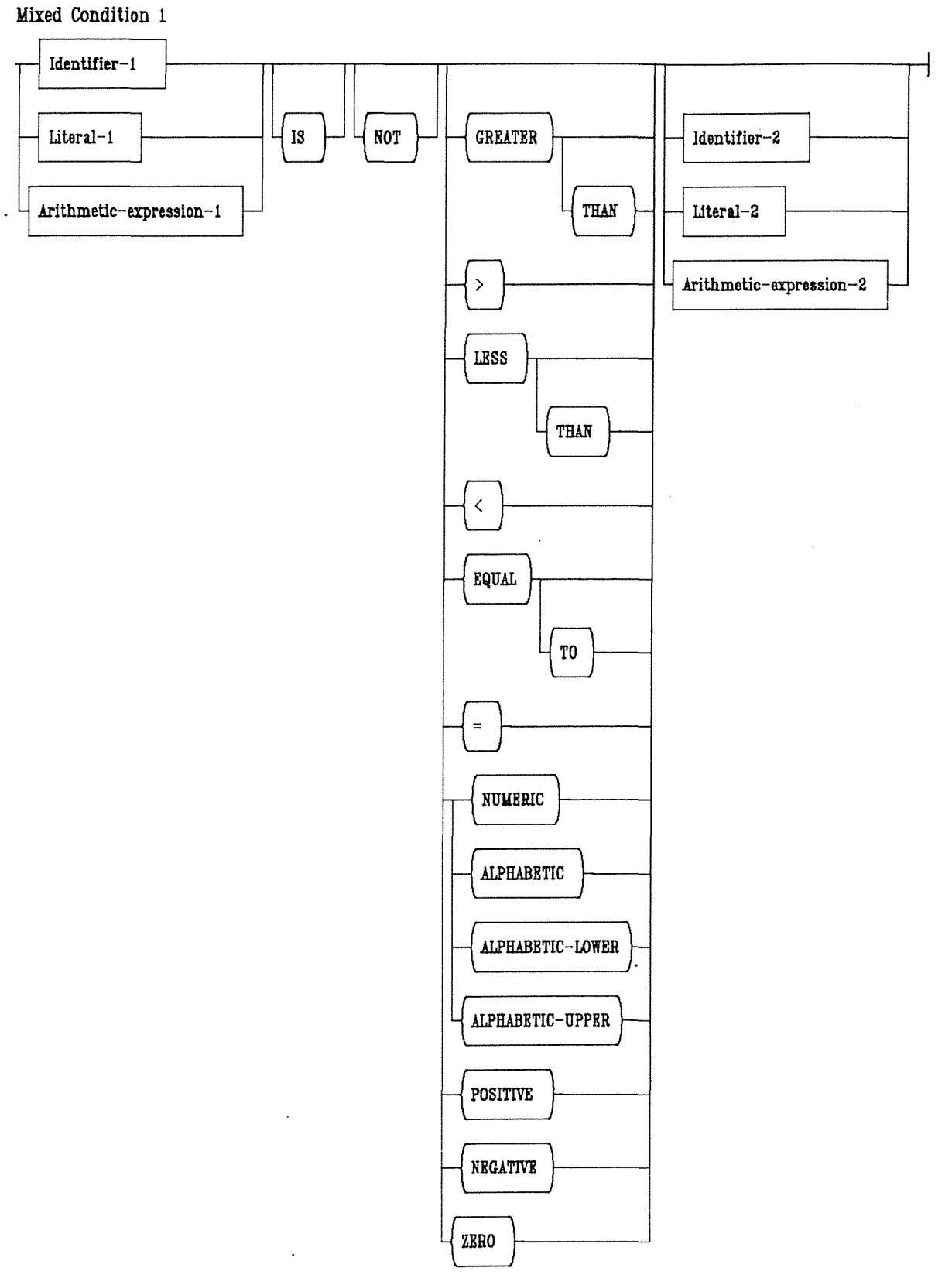
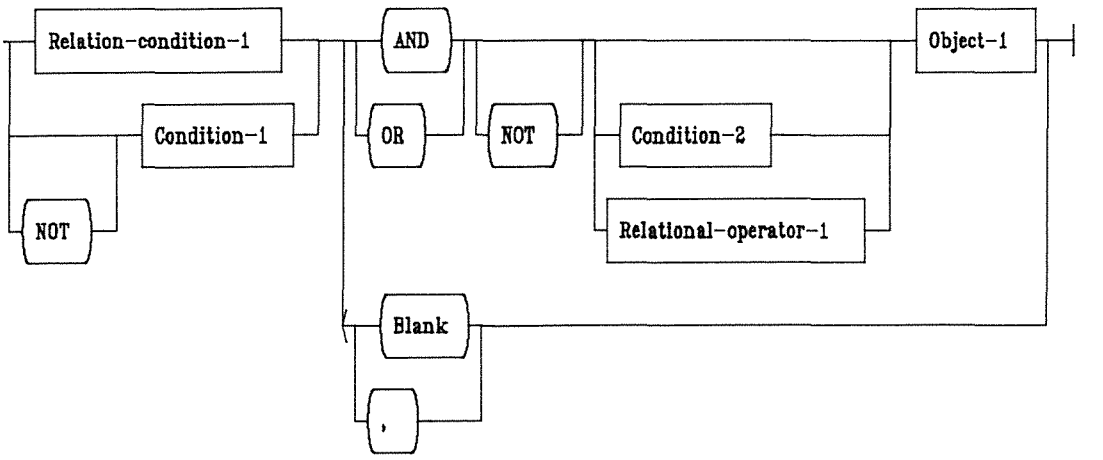


Figure C.38 COBOL Condition Tram-Line Diagram.

Mixed Condition 2



Relation-condition

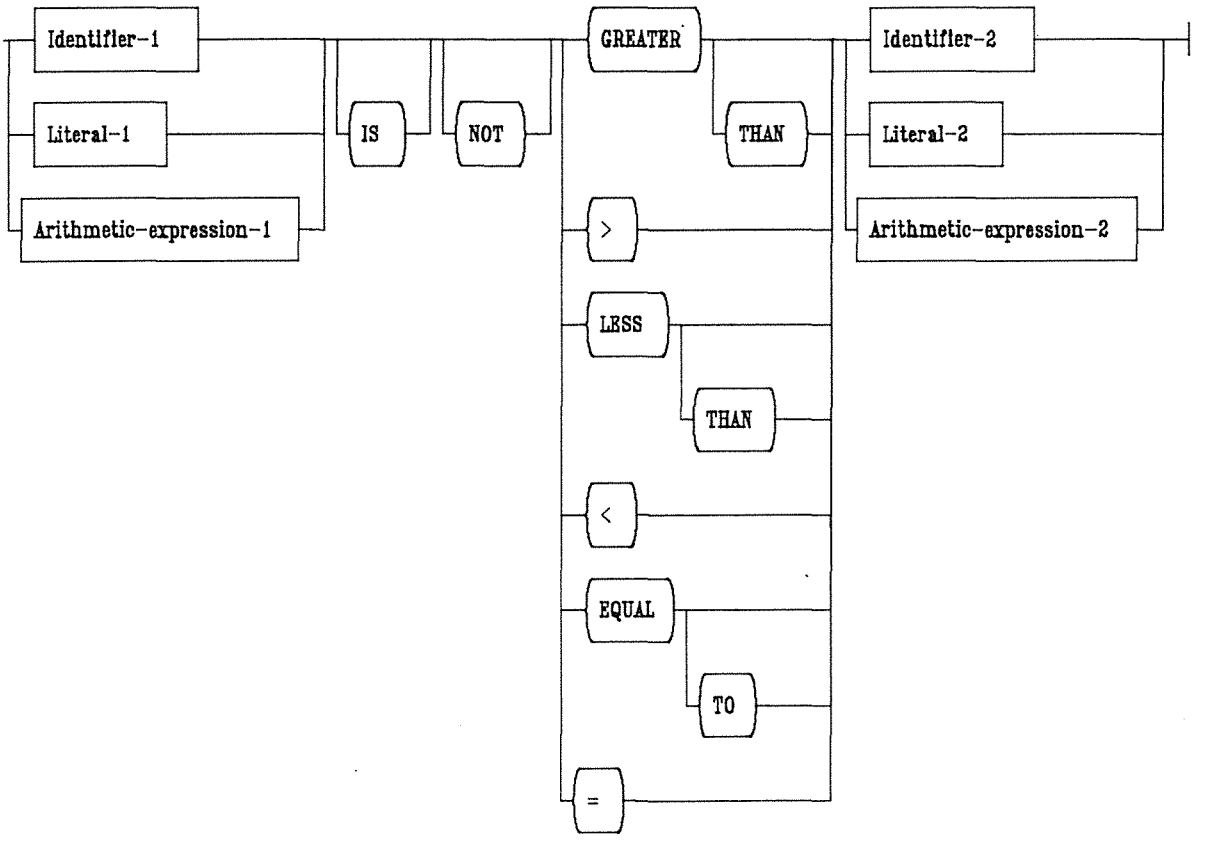


Figure C.39 COBOL Condition Tram-Line Diagram (Continue).

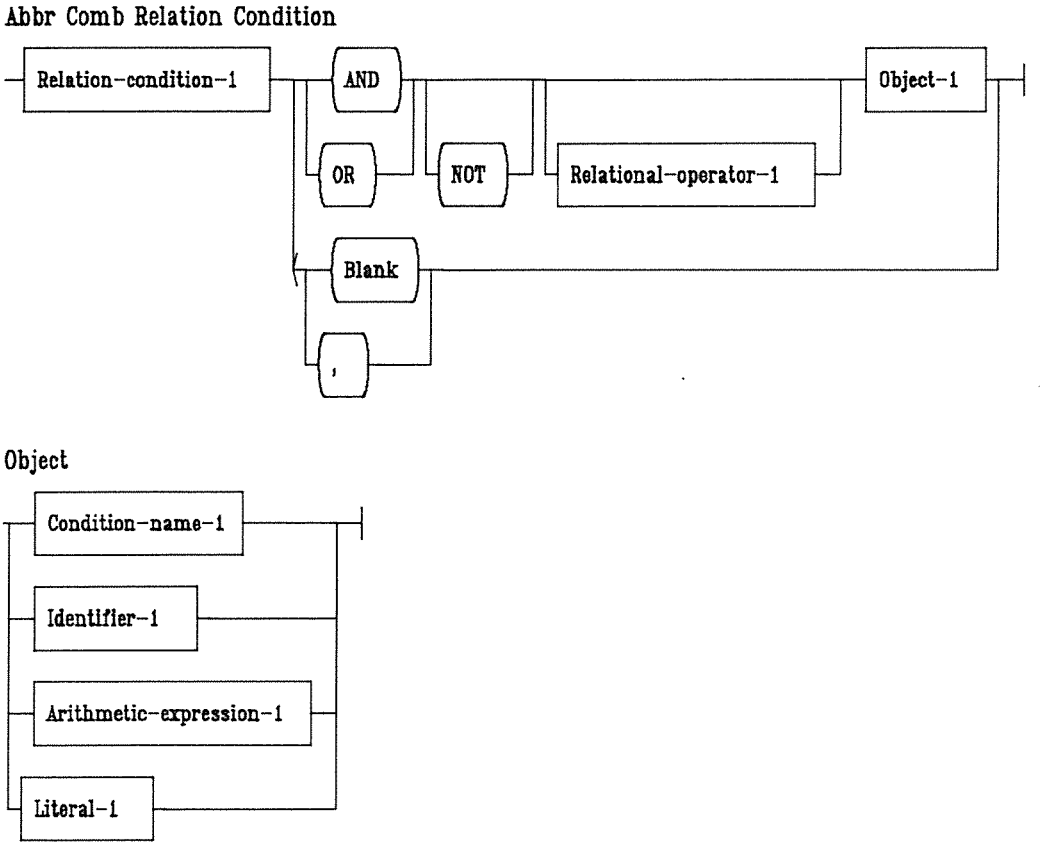
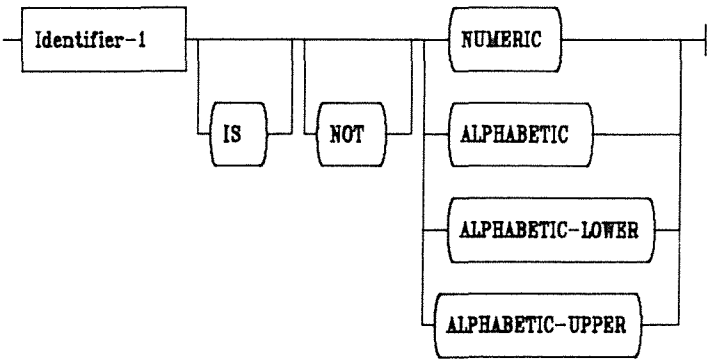
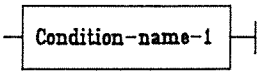


Figure C.40 COBOL Condition Tram-Line Diagram (Continue).

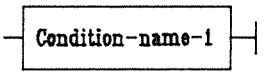
Class Condition



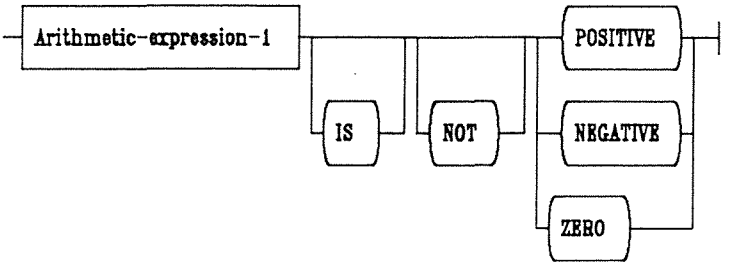
Condition-name Condition



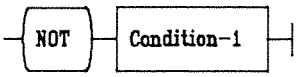
Switch-status Condition



Sign Condition



Negated Condition



Combined Condition

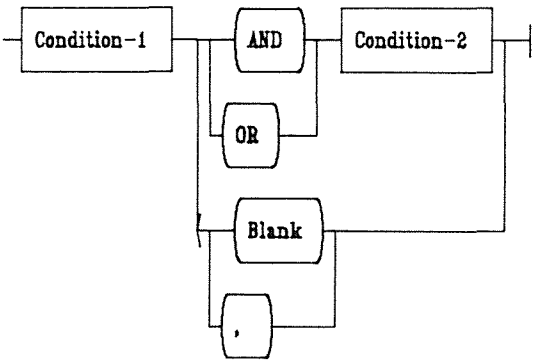
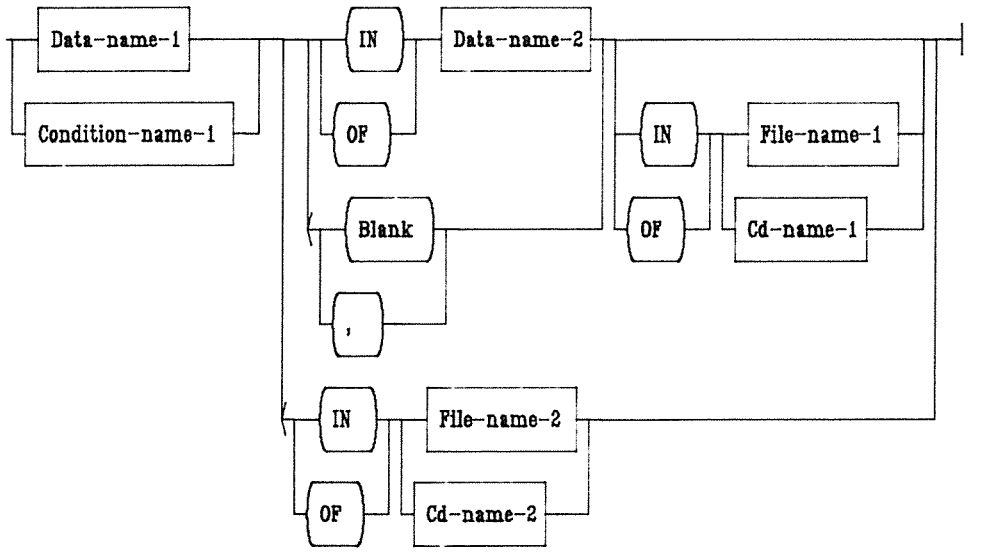
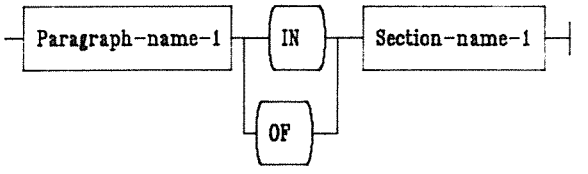


Figure C.41 COBOL Condition Tram-Line Diagram (Continue).

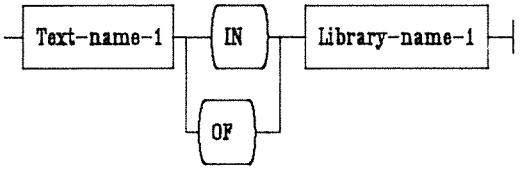
Format 1



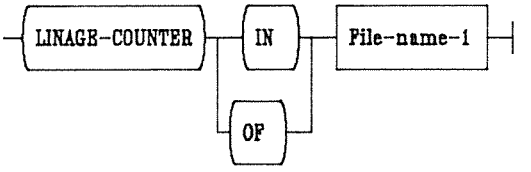
Format 2



Format 3



Format 4



Format 5

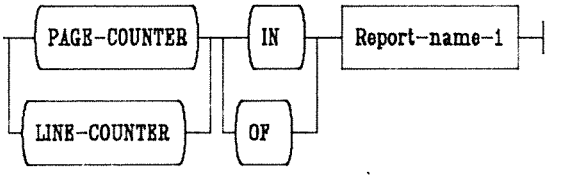
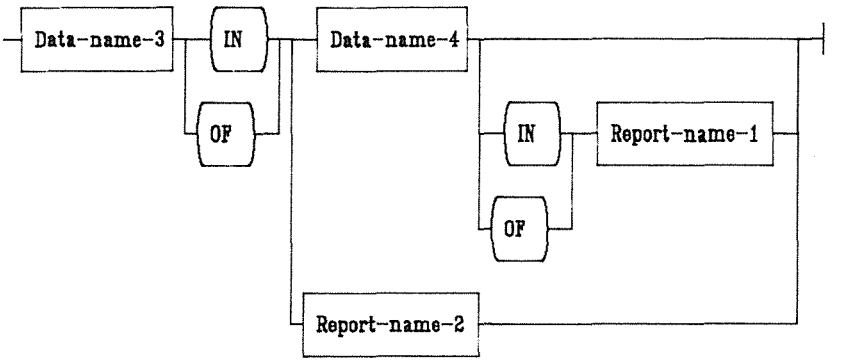
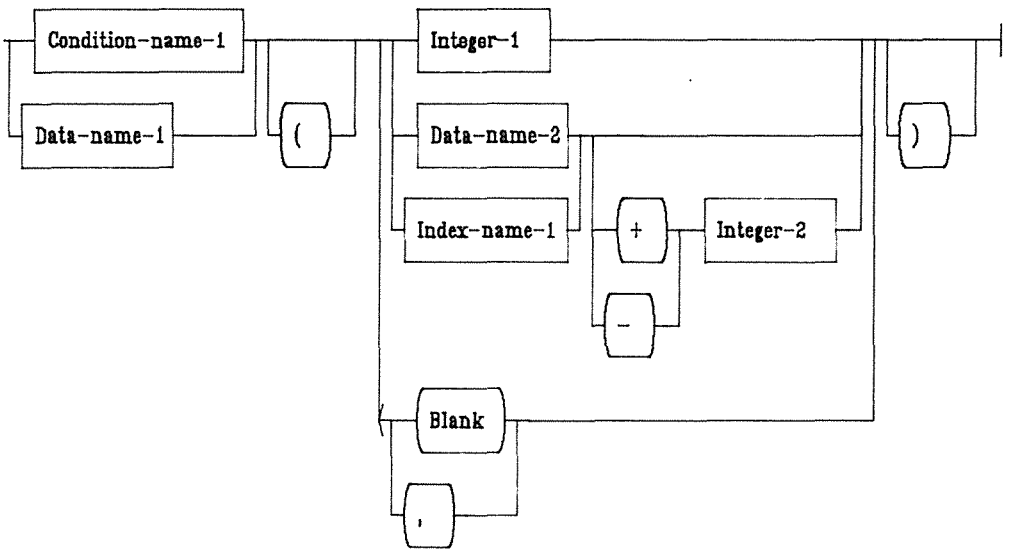


Figure C.42 COBOL Qualification Tram-Line Diagram.

Format 6



Subscripting



Reference-modification

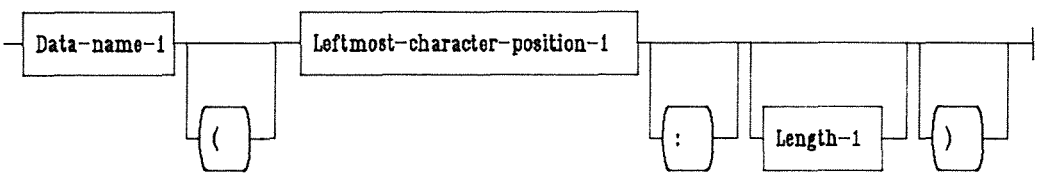
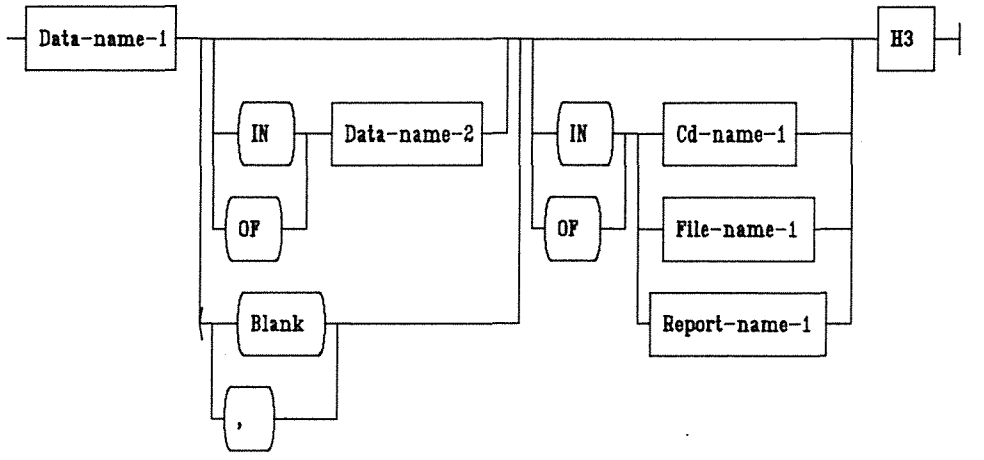
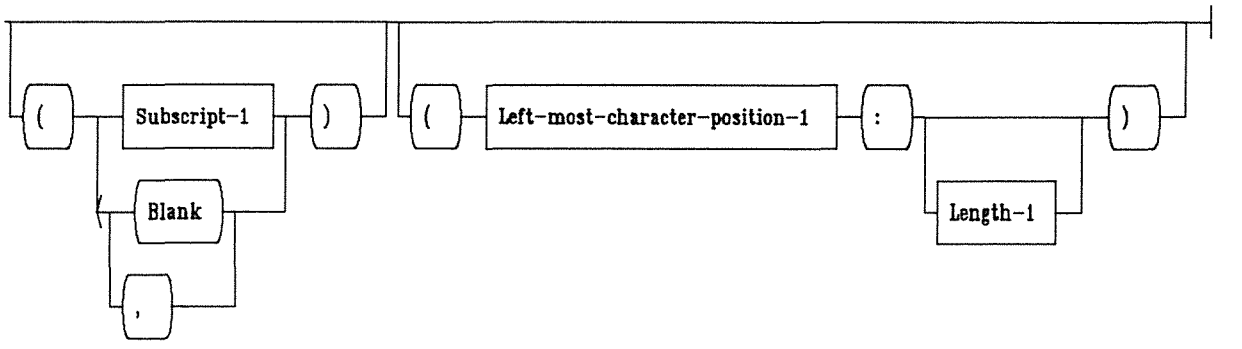


Figure C.43 COBOL Qualification, Subscripting and Reference-modification Tram-Line Diagram.

Identifier



H3



Length

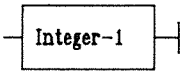


Figure C.44 COBOL Identifier Tram-Line Diagram.

APPENDIX D

MUCAS

STRUCTURE DIAGRAM

APPENDIX D : MUCAS STRUCTURE DIAGRAM

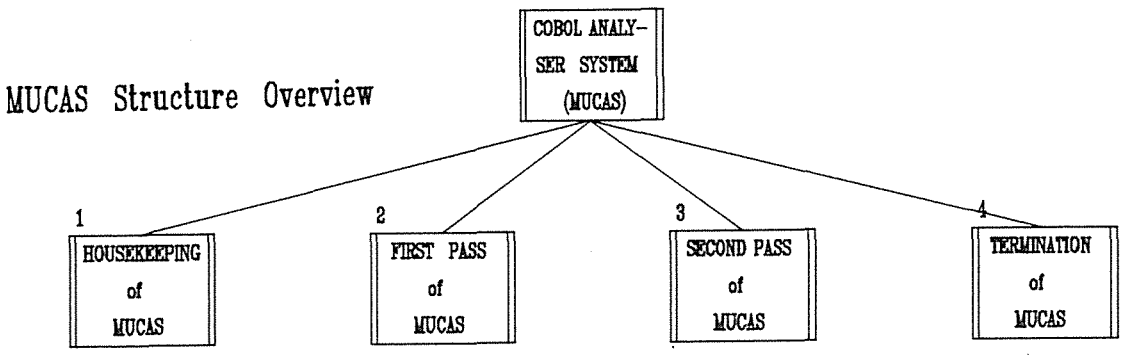


Figure D.1 MUCAS System Overview Structure Diagram.

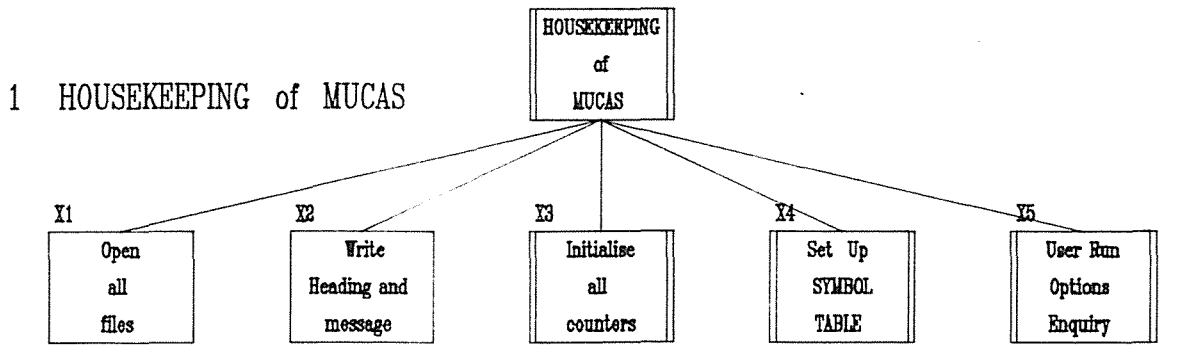


Figure D.2 MUCAS HOUSEKEEPING Module Structure Diagram.

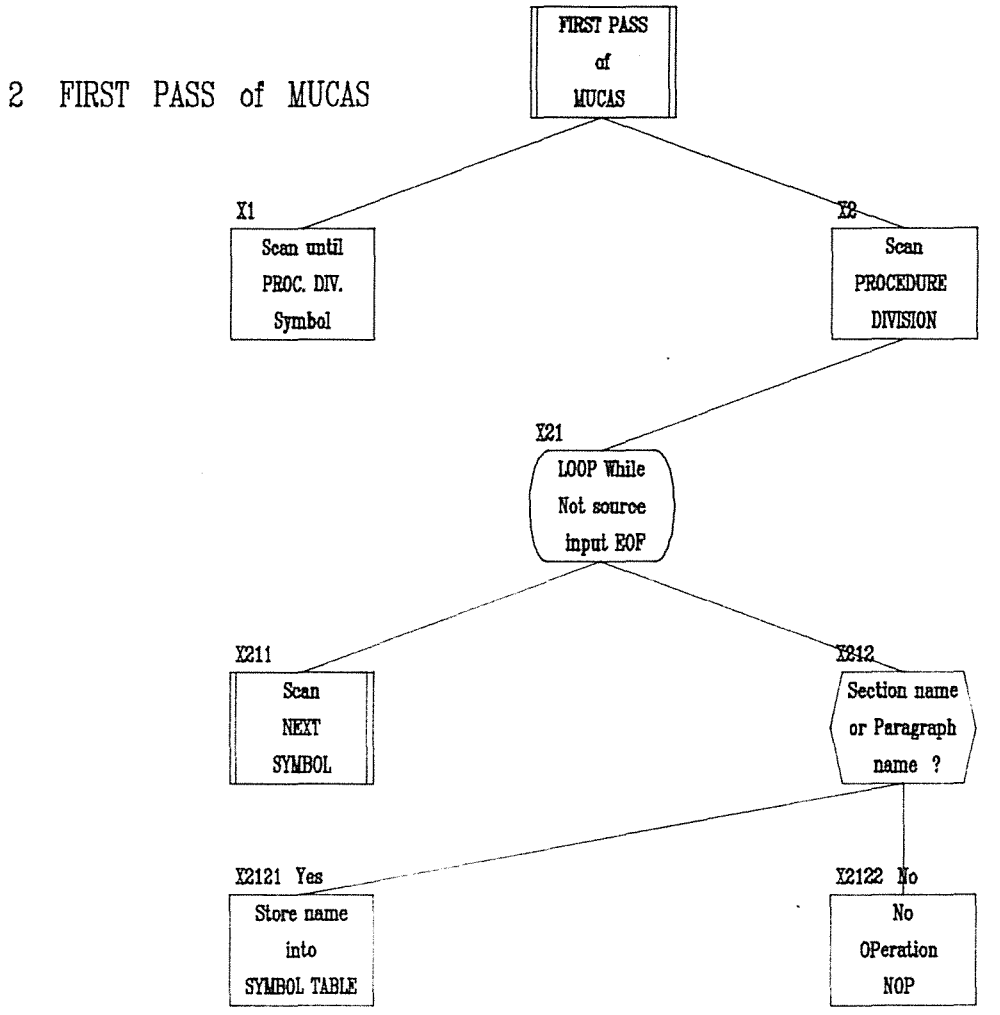


Figure D.3 MUCAS FIRST PASS Module Structure Diagram.

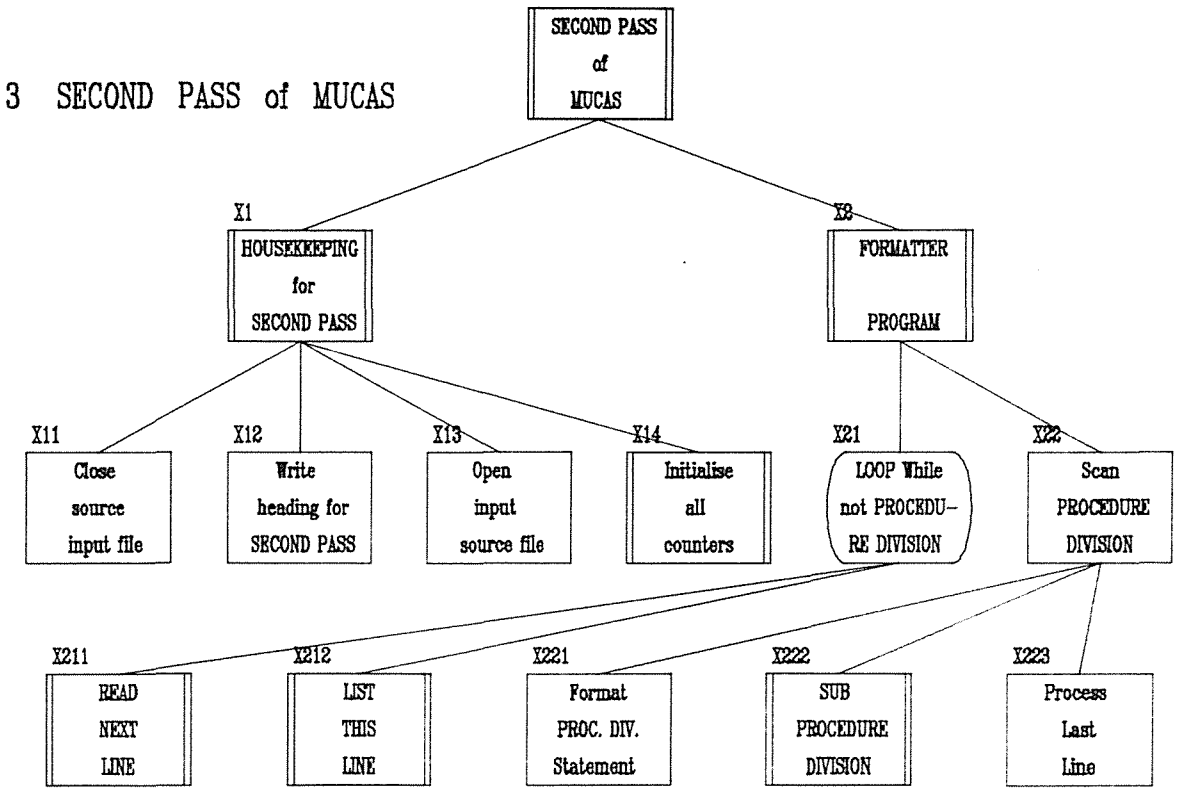


Figure D.4 MUCAS SECOND PASS Module Structure Diagram.

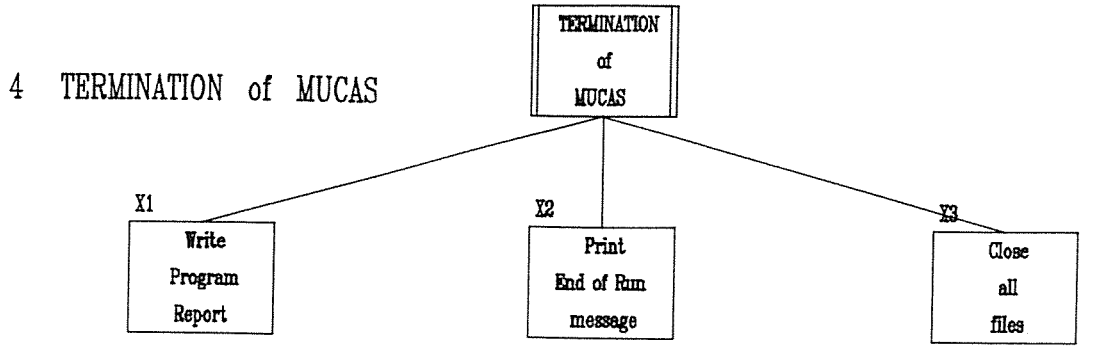


Figure D.5 MUCAS TERMINATION Module Structure Diagram.

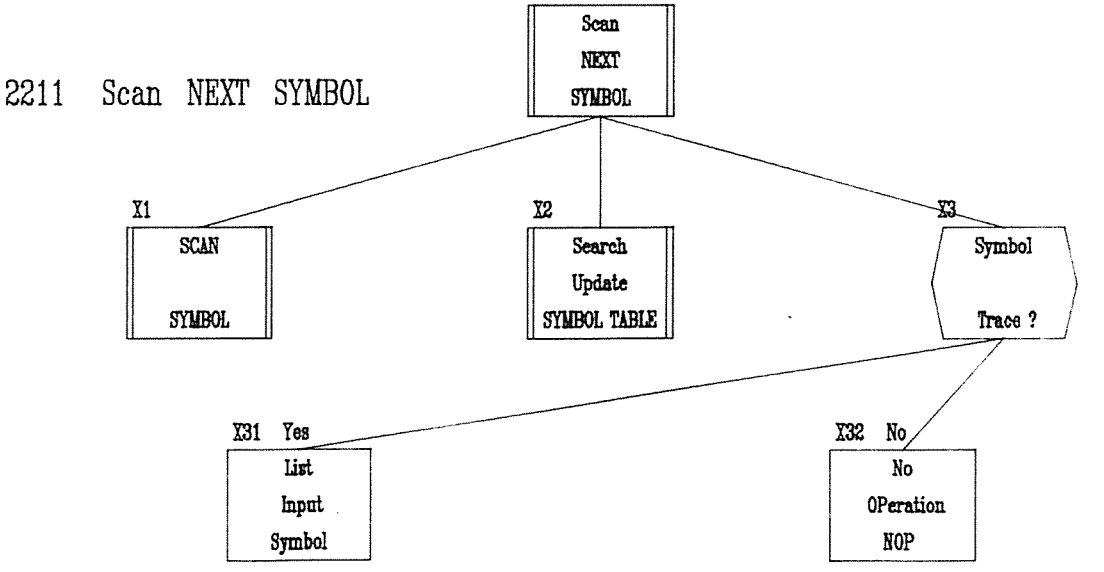


Figure D.6 Scan NEXT SYMBOL Structure Diagram.

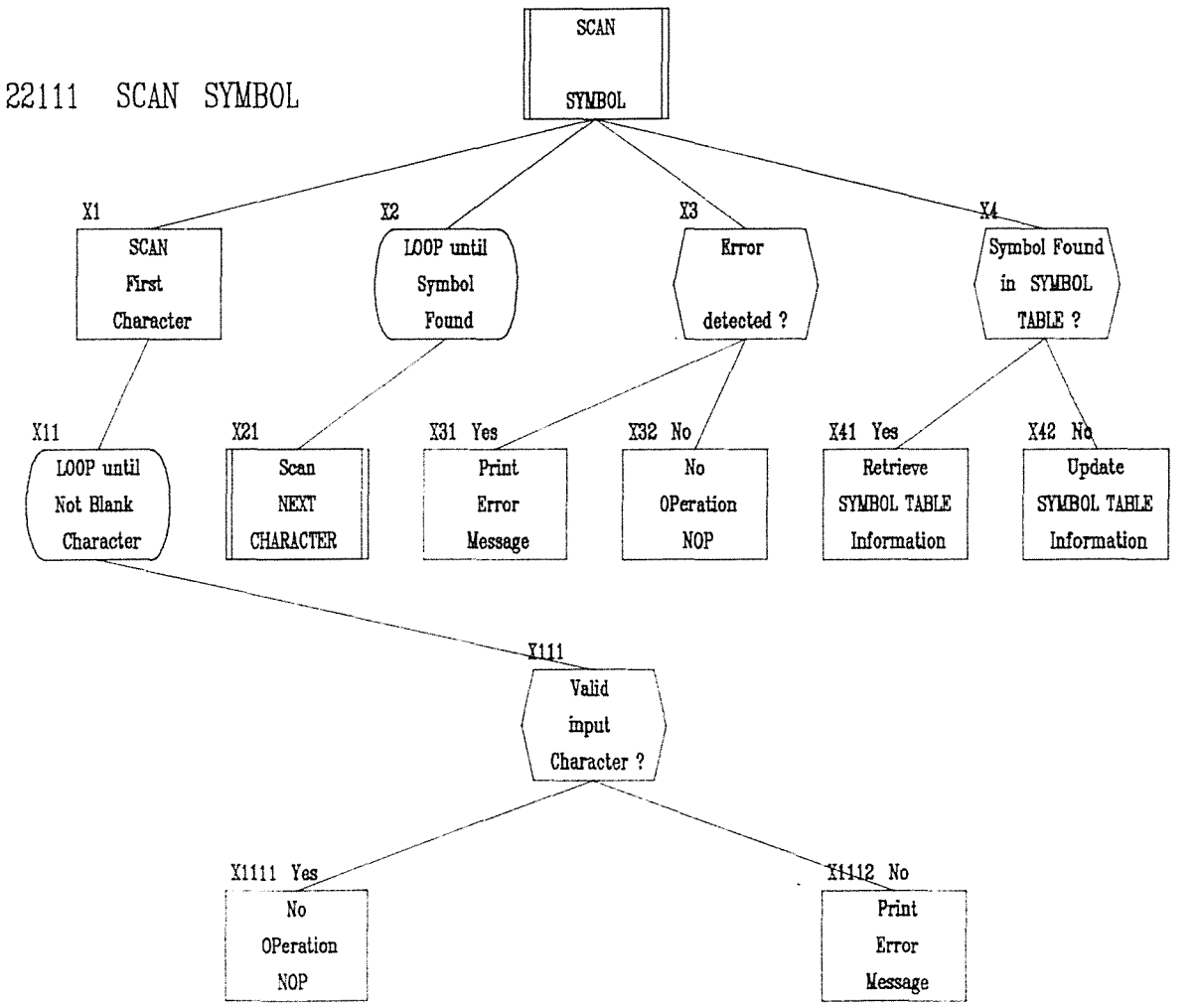


Figure D.7 SCAN SYMBOL Structure Diagram.

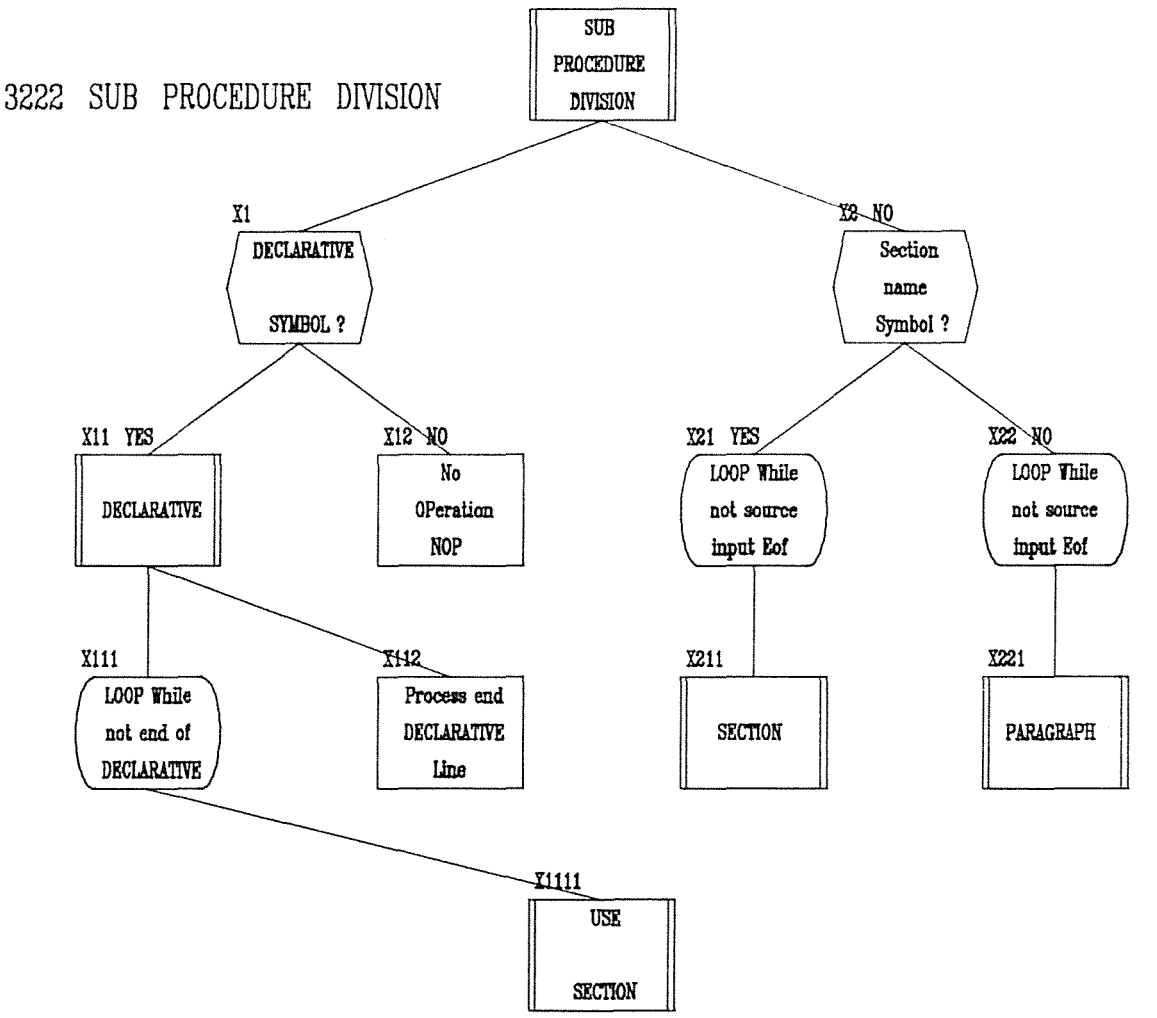


Figure D.8 Parsing COBOL PROCEDURE DIVISION Structure Diagram.

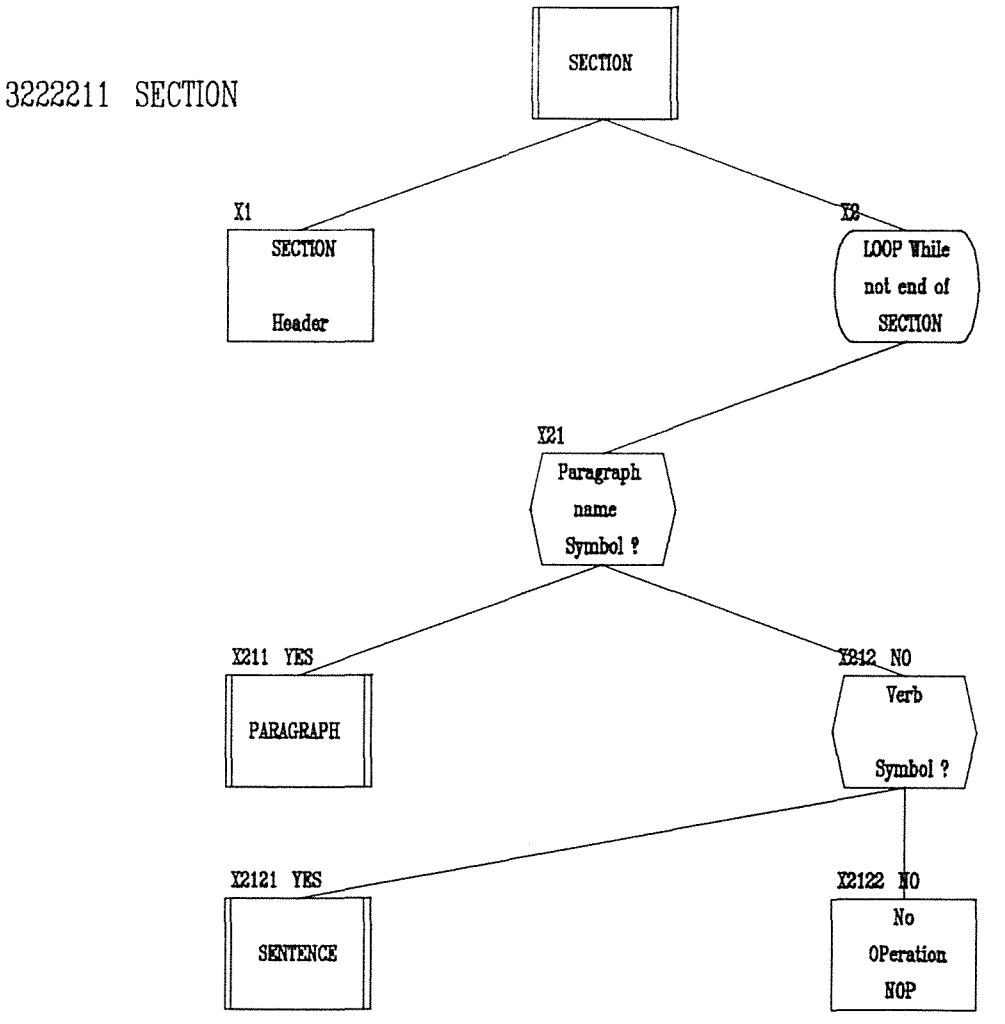


Figure D.9 Parsing a COBOL Section Structure Diagram.

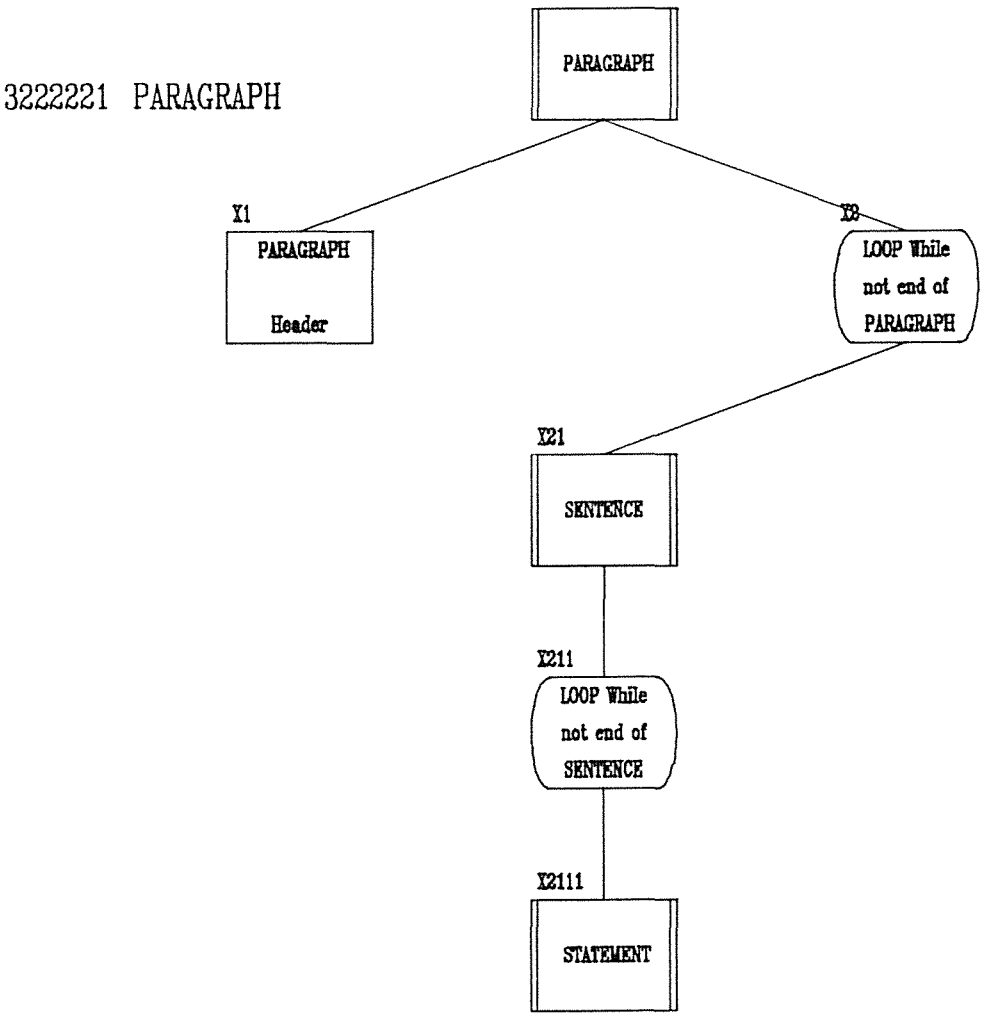


Figure D.10 Parsing a COBOL Paragraph Structure Diagram.

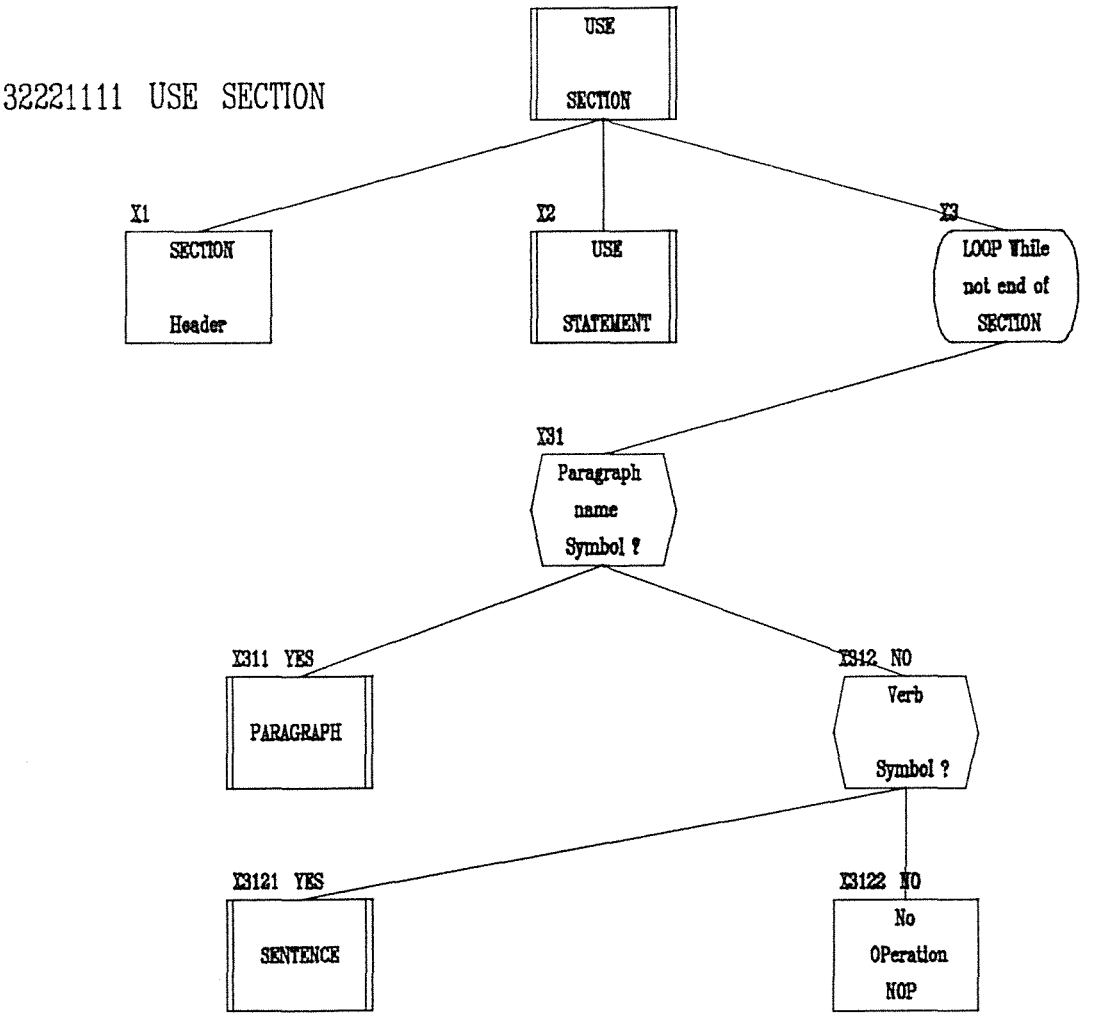


Figure D.11 Parsing a COBOL USE-Section Structure Diagram.

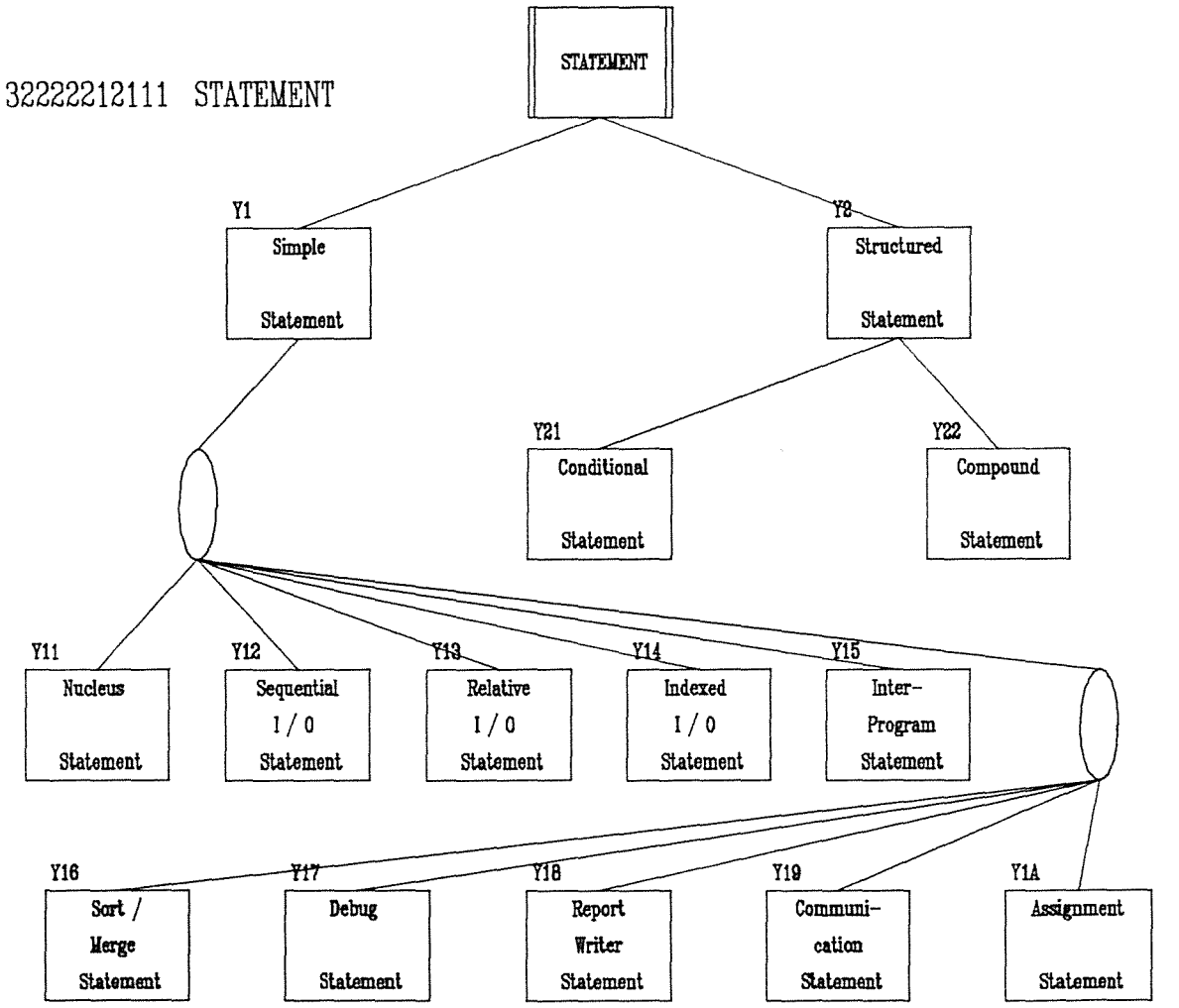


Figure D.12 Parsing a COBOL Statement Structure Diagram.

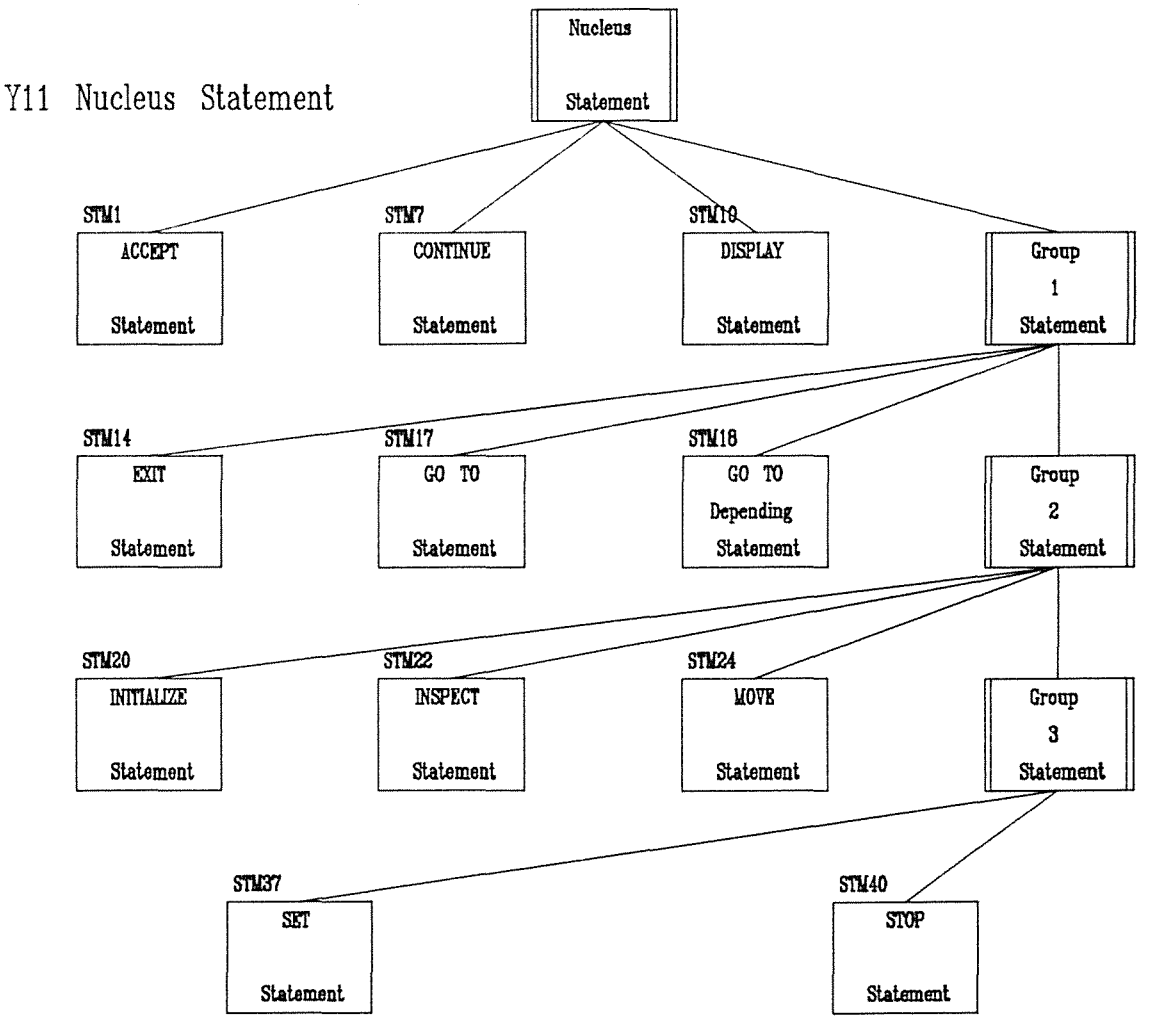


Figure D.13 COBOL Nucleus Statement Structure Diagram.

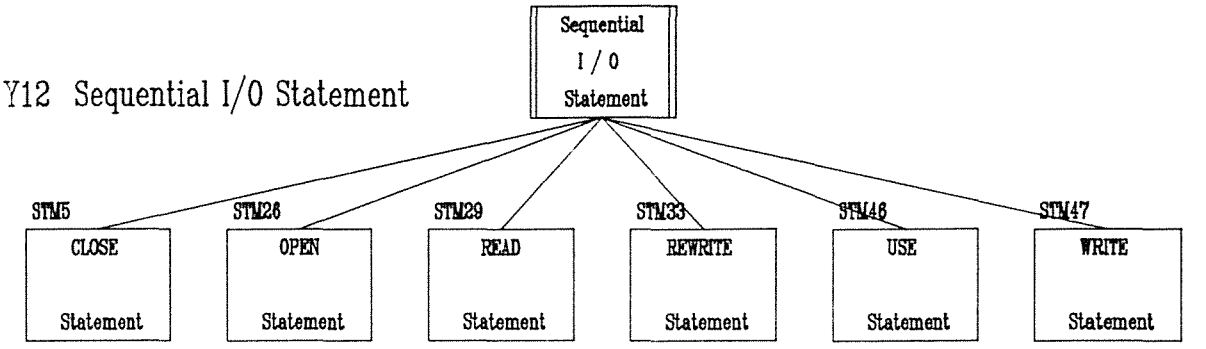


Figure D.14 Sequential I/O Statement Structure Diagram.

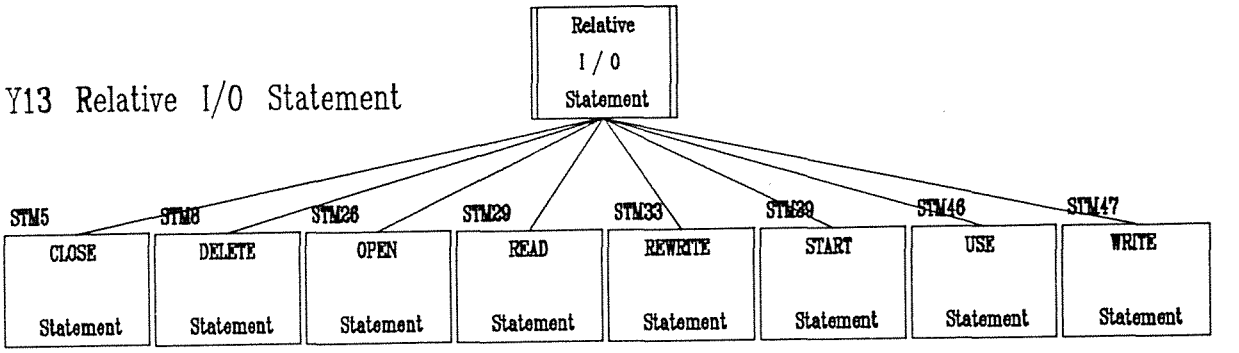


Figure D.15 Relative I/O Statement Structure Diagram.

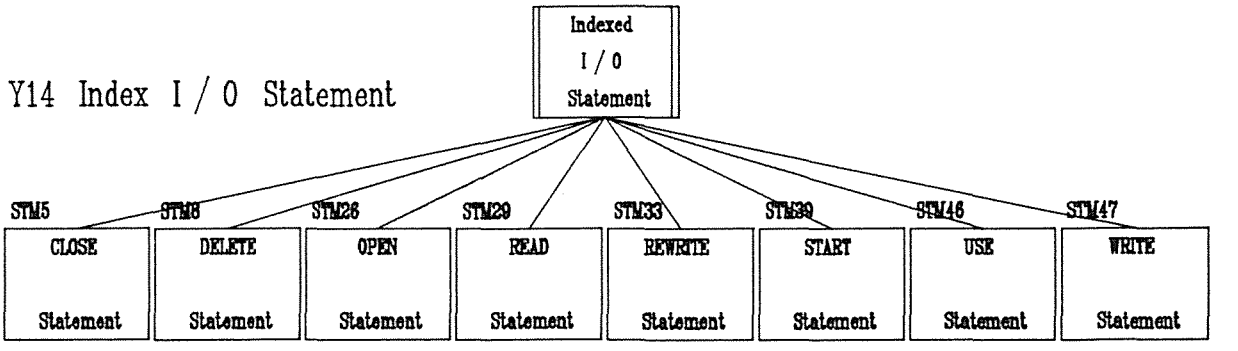


Figure D.16 Index I/O Statement Structure Diagram.

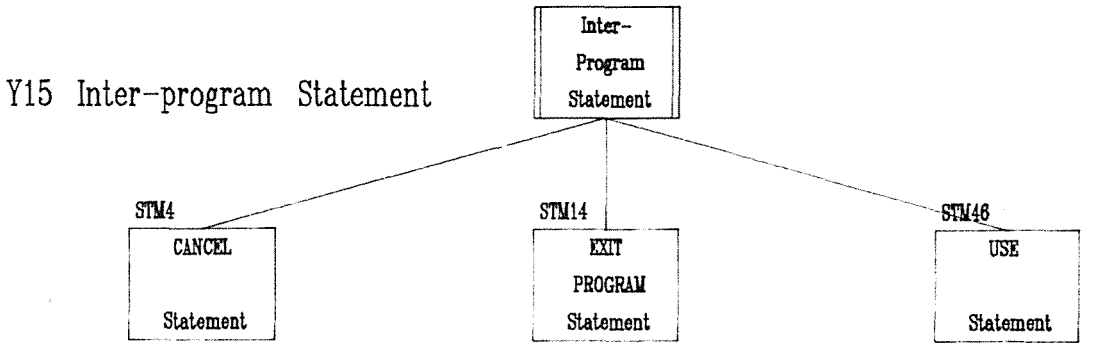


Figure D.17 Inter-Program Statement Structure Diagram.

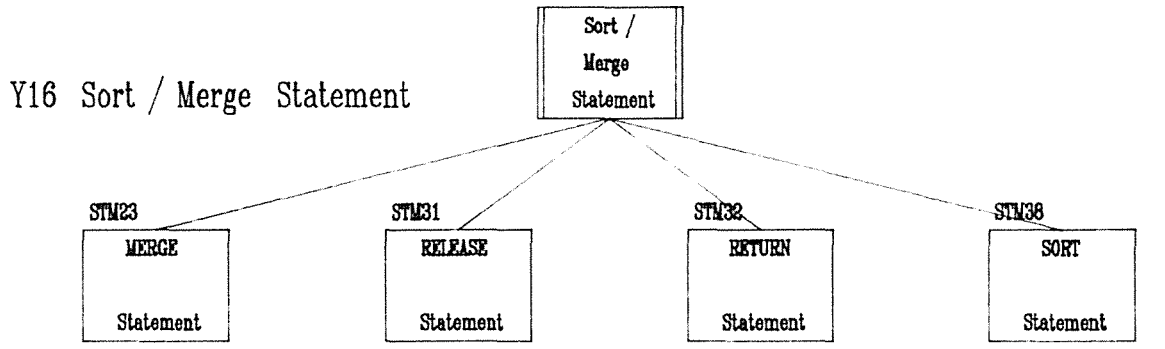


Figure D.18 Sort/Merge Statement Structure Diagram.

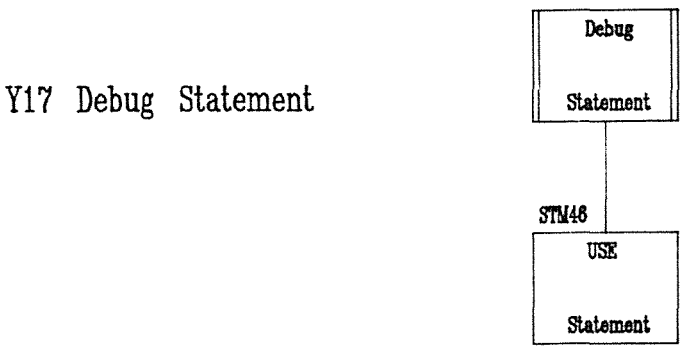


Figure D.19 Debug Statement Structure Diagram.

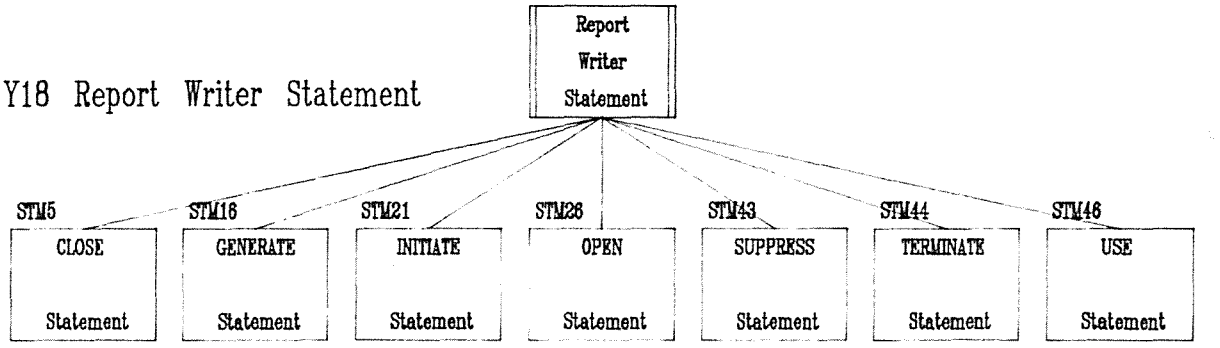


Figure D.20 Report Writer Statement Structure Diagram.

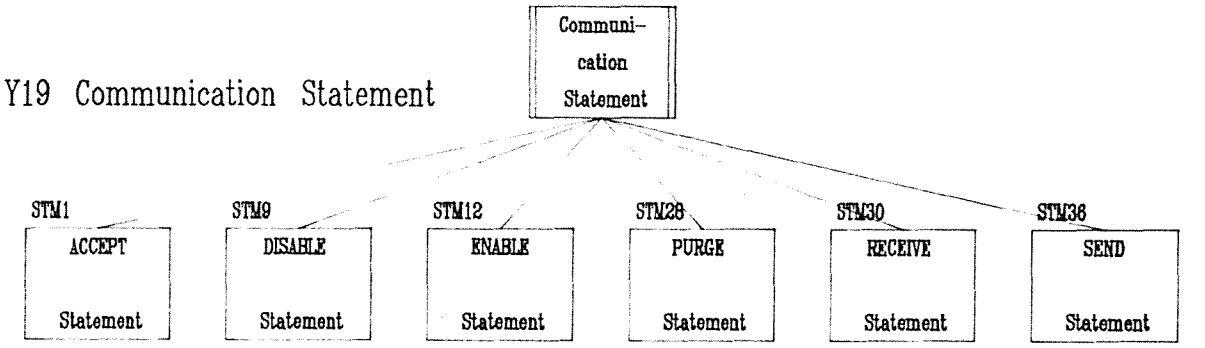


Figure D.21 Communication Statement Structure Diagram.

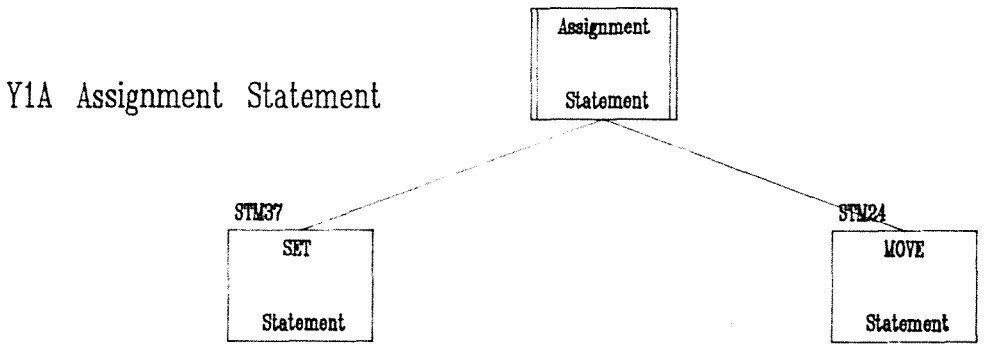


Figure D.22 Assignment Statement Structure Diagram.

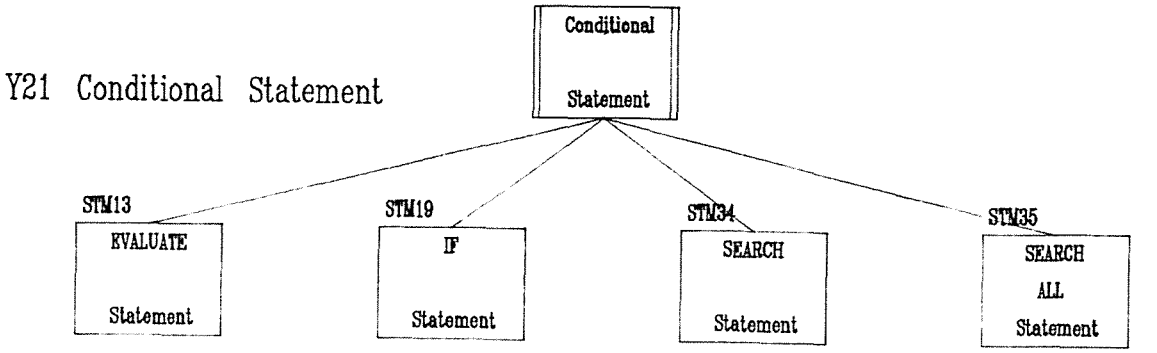


Figure D.23 Condition Statement Structure Diagram.

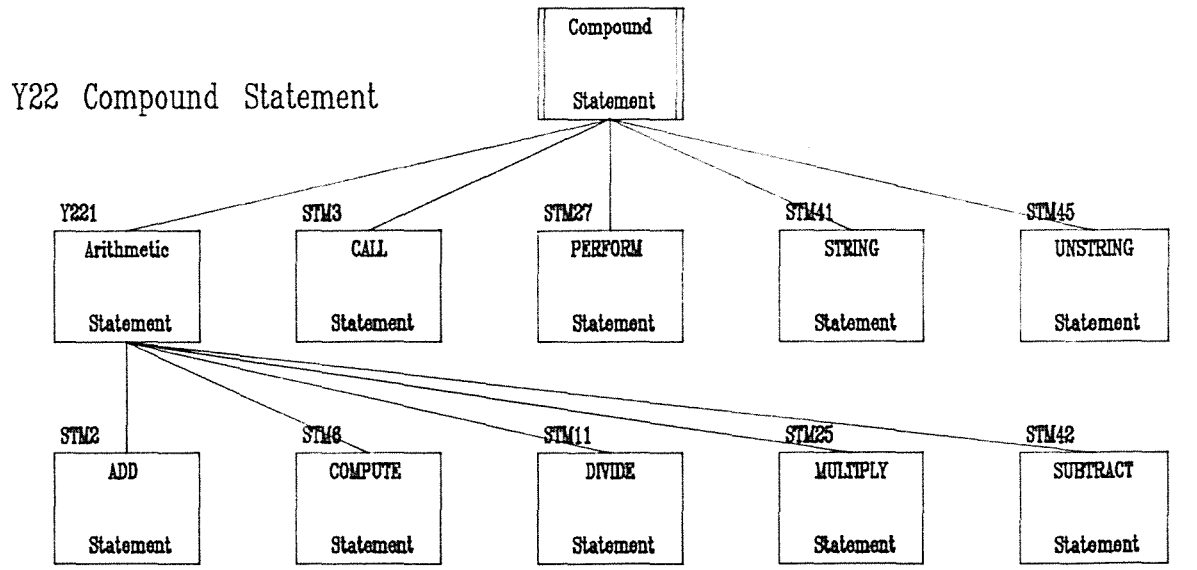


Figure D.24 Compound Statement Structure Diagram.

APPENDIX E

MUCAS

RUN EXAMPLE

APPENDIX E : MUCAS RUN EXAMPLE

```
       IDENTIFICATION DIVISION.

         PROGRAM-ID. MUCASB01.
         AUTHOR.     LUNG  C. P.

       ENVIRONMENT DIVISION.

        INPUT-OUTPUT SECTION.
        FILE-CONTROL.
           SELECT TESTFILE ASSIGN TO PFMS.
           SELECT OUTFILE ASSIGN TO PRINTER.

       DATA DIVISION.

           COPY 'MUCASB01.DATADIV'.
      /
       PROCEDURE DIVISION.
       0010-MAINLINE.
           PERFORM 0100-INITIALIZATION.
           PERFORM 0200-MAIN-PROCESSING.
           PERFORM 0900-TERMINATION.
           STOP RUN.
       0100-INITIALIZATION.
           OPEN INPUT  TESTFILE
                OUTPUT OUTFILE.
       0200-MAIN-PROCESSING.
           PERFORM 1000-LOOP-BEGIN.
           PERFORM 2000-CHECK-PROCESS-READ
                    UNTIL END-OF-FILE.
           PERFORM 3000-LOOP-END.
       0900-TERMINATION.
           CLOSE TESTFILE
                 OUTFILE.
       1000-LOOP-BEGIN.
           MOVE ZERO TO DATA1   VALID-DATA
                        SUM1    SUM2.
           MOVE FALSE TO IN-STATUS.
           WRITE PRINT-LINE FROM HEADING-1
                    AFTER ADVANCING PAGE.
           WRITE PRINT-LINE FROM HEADING-2
                    AFTER 2 LINES.
           PERFORM 4200-READ-DATA-TEST.
       2000-CHECK-PROCESS-READ.
           ADD 1 TO DATA1.
           PERFORM 4000-CHECK-DATA.
           IF DATA-VALID
               PERFORM 4100-PROCESS-DATA
               PERFORM 4300-WRITE-LINE
           ELSE
               PERFORM 4400-WRITE-ERROR.
           PERFORM 4200-READ-DATA-TEST.
        3000-LOOP-END.
           DIVIDE SUM1 BY VALID-DATA
                  GIVING TEST1 ROUNDED.
           DIVIDE SUM2 BY VALID-DATA
                  GIVING TEST2 ROUNDED.
           PERFORM 4500-WRITE-SUMMARY.
      /
```

Figure E.1 The Test Input COBOL Source Program.

```
4000-CHECK-DATA.
   MOVE FALSE TO DATA-STATUS.
   IF IN-TEST IS NOT NUMERIC
       MOVE 'NON-NUMERIC DATA' TO EL-MESSAGE
   ELSE
       IF IN-TEST1 < 101
           MOVE TRUE TO DATA-STATUS
       ELSE
           MOVE 'DATA OUT OF RANGE' TO EL-MESSAGE.

4100-PROCESS-DATA.
   ADD 1 TO VALID-DATA.
   MOVE ZERO TO SUM-ALL.
   MOVE IN-NAME TO OL-NAME.

   MOVE IN-TEST1 TO OL-TEST1  TEST1.
   MOVE IN-TEST2 TO OL-TEST2  TEST2.

   ADD TEST1 TO SUM1.
   ADD TEST2 TO SUM2.

   ADD TEST1 TEST2   GIVING SUM-ALL.
   MOVE SUM-ALL TO OL-SUM.
   DIVIDE SUM-ALL BY 2
       GIVING OL-AVERAGE ROUNDED.

4200-READ-DATA-TEST.
   READ TESTFILE
   AT END
       MOVE TRUE TO IN-STATUS.

4300-WRITE-LINE.
   WRITE PRINT-LINE FROM OUTPUT-LINE
           AFTER ADVANCING 2 LINES.

4400-WRITE-ERROR.
   MOVE IN-NAME TO EL-NAME.
   MOVE IN-TEST1 TO EL-TEST1.
   MOVE IN-TEST2 TO EL-TEST2.

   WRITE PRINT-LINE FROM ERROR-LINE AFTER 2.
/
4500-WRITE-SUMMARY.
   MOVE SPACES TO OUTPUT-LINE.
   MOVE 'SUBJECT AVERAGES' TO OL-NAME.
   MOVE TEST1 TO OL-TEST1.
   MOVE TEST2 TO OL-TEST2.

   WRITE PRINT-LINE FROM OUTPUT-LINE
           AFTER ADVANCING 2 LINES.
   MOVE SPACES TO OUTPUT-LINE.
   MOVE 'VALID RECORDS' TO OL-NAME.
   MOVE VALID-DATA TO OL-TEST1.
   WRITE PRINT-LINE FROM OUTPUT-LINE
           AFTER ADVANCING 2 LINES.
   MOVE SPACES TO OUTPUT-LINE.
   MOVE 'SUM RECORDS' TO OL-NAME.
   MOVE DATA1 TO OL-TEST1.
   WRITE PRINT-LINE FROM OUTPUT-LINE
           AFTER 2.
```

Figure E.2 The Test Input COBOL Source Program (Continued).

```
 IDENTIFICATION DIVISION.                                                00001
                                                                         00002
   PROGRAM-ID. MUCASB01.                                                 00003
   AUTHOR.     LUNG  C. P.                                               00004
                                                                         00005
 ENVIRONMENT DIVISION.                                                   00006
                                                                         00007
  INPUT-OUTPUT SECTION.                                                  00008
  FILE-CONTROL.                                                          00009
     SELECT TESTFILE ASSIGN TO PFMS.                                     00010
     SELECT OUTFILE ASSIGN TO PRINTER.                                   00011
                                                                         00012
 DATA DIVISION.                                                          00013
                                                                         00014
     COPY 'MUCASB01.DATADIV'.                                            00015
/                                                                        00016
                                                                         00017
 PROCEDURE DIVISION.                                                     00018
                                                                         00019
 0010-MAINLINE.                                                          00020
     PERFORM 0100-INITIALIZATION.                                        00021
     PERFORM 0200-MAIN-PROCESSING.                                       00022
     PERFORM 0900-TERMINATION.                                           00023
     STOP RUN.                                                           00024
                                                                         00025
 0100-INITIALIZATION.                                                    00026
     OPEN                                                                00027
       INPUT TESTFILE                                                    00028
       OUTPUT OUTFILE.                                                   00029
                                                                         00030
 0200-MAIN-PROCESSING.                                                   00031
     PERFORM 1000-LOOP-BEGIN.                                            00032
     PERFORM 2000-CHECK-PROCESS-READ                                     00033
       UNTIL END-OF-FILE.                                                00034
     PERFORM 3000-LOOP-END.                                              00035
                                                                         00036
 0900-TERMINATION.                                                       00037
     CLOSE TESTFILE OUTFILE.                                             00038
                                                                         00039
 1000-LOOP-BEGIN.                                                        00040
     MOVE ZERO TO DATA1 VALID-DATA SUM1 SUM2.                            00041
     MOVE FALSE TO IN-STATUS.                                            00042
     WRITE PRINT-LINE FROM HEADING-1                                     00043
       AFTER ADVANCING PAGE.                                             00044
     WRITE PRINT-LINE FROM HEADING-2                                     00045
       AFTER 2 LINES.                                                    00046
     PERFORM 4200-READ-DATA-TEST.                                        00047
                                                                         00048
 2000-CHECK-PROCESS-READ.                                                00049
     ADD 1 TO DATA1.                                                     00050
     PERFORM 4000-CHECK-DATA.                                            00051
     IF DATA-VALID                                                       00052
       PERFORM 4100-PROCESS-DATA                                         00053
         PERFORM 4300-WRITE-LINE                                         00054
     ELSE                                                                00055
       PERFORM 4400-WRITE-ERROR.                                         00056
     PERFORM 4200-READ-DATA-TEST.                                        00057
                                                                         00058
 3000-LOOP-END.                                                          00059
     DIVIDE SUM1 BY VALID-DATA                                           00060
       GIVING TEST1 ROUNDED.                                             00061
     DIVIDE SUM2 BY VALID-DATA                                           00062
       GIVING TEST2 ROUNDED.                                             00063
     PERFORM 4500-WRITE-SUMMARY.                                         00064
/                                                                        00065
                                                                         00066
 4000-CHECK-DATA.                                                        00067
     MOVE FALSE TO DATA-STATUS.                                          00068
```

Figure E.3 The Reformatted COBOL Source Program.

```
     IF IN-TEST IS NOT NUMERIC                                          00069
       MOVE 'NON-NUMERIC DATA' TO EL-MESSAGE                            00070
     ELSE IF IN-TEST1 < 101                                             00071
       MOVE TRUE TO DATA-STATUS                                         00072
     ELSE                                                               00073
       MOVE 'DATA OUT OF RANGE' TO EL-MESSAGE.                          00074
                                                                        00075
                                                                        00076
 4100-PROCESS-DATA.                                                     00077
     ADD 1 TO VALID-DATA.                                               00078
     MOVE ZERO TO SUM-ALL.                                              00079
     MOVE IN-NAME TO OL-NAME.                                           00080
                                                                        00081
     MOVE IN-TEST1 TO OL-TEST1 TEST1.                                   00082
     MOVE IN-TEST2 TO OL-TEST2 TEST2.                                   00083
                                                                        00084
     ADD TEST1 TO SUM1.                                                 00085
     ADD TEST2 TO SUM2.                                                 00086
                                                                        00087
     ADD TEST1 TO TEST2                                                 00088
       GIVING SUM-ALL.                                                  00089
     MOVE SUM-ALL TO OL-SUM.                                            00090
     DIVIDE SUM-ALL BY 2                                                00091
       GIVING OL-AVERAGE ROUNDED.                                       00092
                                                                        00093
                                                                        00094
 4200-READ-DATA-TEST.                                                   00095
     READ TESTFILE                                                      00096
       AT END                                                           00097
         MOVE TRUE TO IN-STATUS.                                        00098
                                                                        00099
                                                                        00100
 4300-WRITE-LINE.                                                       00101
     WRITE PRINT-LINE FROM OUTPUT-LINE                                  00102
       AFTER ADVANCING 2 LINES.                                         00103
                                                                        00104
                                                                        00105
 4400-WRITE-ERROR.                                                      00106
     MOVE IN-NAME TO EL-NAME.                                           00107
     MOVE IN-TEST1 TO EL-TEST1.                                         00108
     MOVE IN-TEST2 TO EL-TEST2.                                         00109
                                                                        00110
     WRITE PRINT-LINE FROM ERROR-LINE                                   00111
       AFTER 2.                                                         00112
/                                                                       00113
                                                                        00114
 4500-WRITE-SUMMARY.                                                    00115
     MOVE SPACES TO OUTPUT-LINE.                                        00116
     MOVE 'SUBJECT AVERAGES' TO OL-NAME.                                00117
     MOVE TEST1 TO OL-TEST1.                                            00118
     MOVE TEST2 TO OL-TEST2.                                            00119
                                                                        00120
     WRITE PRINT-LINE FROM OUTPUT-LINE                                  00121
       AFTER ADVANCING 2 LINES.                                         00122
     MOVE SPACES TO OUTPUT-LINE.                                        00123
     MOVE 'VALID RECORDS' TO OL-NAME.                                   00124
     MOVE VALID-DATA TO OL-TEST1.                                       00125
     WRITE PRINT-LINE FROM OUTPUT-LINE                                  00126
       AFTER ADVANCING 2 LINES.                                         00127
     MOVE SPACES TO OUTPUT-LINE.                                        00128
     MOVE 'SUM RECORDS' TO OL-NAME.                                     00129
     MOVE DATA1 TO OL-TEST1.                                            00130
     WRITE PRINT-LINE FROM OUTPUT-LINE                                  00131
       AFTER 2.                                                         00132
```

Figure E.4 The Reformatted COBOL Source Program (Continued).

```
  Started Complexity Report of COBOL ANALYSER SYSTEM    Rev. 19.1   ....

  Started Complexity Report of     0010-MAINLINE  Paragraph
  Started Line Number              20

Verb Type  Modifier Type  V. Count  V. Weight.  M. Count  M. Weight.  Weighted Value
---------  -------------  --------  ----------  --------  ----------  --------------
PERFORM    Imperative        3          5          0          1            15
  STOP     None              1          1          0          0             1


Total No. of Weighting  =          16
Total No. of Verbs      =           4
Total No. of Modifiers  =           0

 Total No. of Weighted Value / Total No. of Verbs   is
   Average Verb Complexity Weighted Value in Paragraph  =          4.00

 Total No. of Weighted Value is
   Paragraph     Complexity      Weighted     Value     =            16
  Ended   Line Number              25

  Completed  Complexity  Report of        0010-MAINLINE  Paragraph
```

Figure E.5 The Reformatted COBOL Source Program Paragraph 0010-MAINLINE Programming Syntax Complexity Measure Report.

```
  Started Complexity Report of     0100-INITIALIZATION   Paragraph
  Started Line Number              26

Verb Type  Modifier Type  V. Count  V. Weight.  M. Count  M. Weight.  Weighted Value
---------  -------------  --------  ----------  --------  ----------  --------------
  OPEN     None              1          3          0          0             3

Total No. of Weighting  =           3
Total No. of Verbs      =           1
Total No. of Modifiers  =           0

 Total No. of Weighted Value / Total No. of Verbs   is
   Average Verb Complexity Weighted Value in Paragraph  =          3.00

 Total No. of Weighted Value is
   Paragraph     Complexity      Weighted     Value     =             3
  Ended   Line Number              30

  Completed  Complexity  Report of        0100-INITIALIZATION   Paragraph
```

Figure E.6 The Reformatted COBOL Source Program Paragraph 0100-INITIALIZATION Programming Syntax Complexity Measure Report.

```
  Started Complexity Report of      0200-MAIN-PROCESSING  Paragraph
  Started Line Number               31

Verb Type  Modifier Type  V. Count  V. Weight.  M. Count  M. Weight.  Weighted Value
---------  -------------  --------  ----------  --------  ----------  --------------
PERFORM    Imperative        3          5          0          1             15

Total No. of Weighting  =          15
Total No. of Verbs      =           3
Total No. of Modifiers  =           0

 Total No. of Weighted Value / Total No. of Verbs   is
   Average Verb Complexity Weighted Value in Paragraph  =          5.00

 Total No. of Weighted Value is
   Paragraph     Complexity      Weighted     Value     =             15
  Ended   Line Number             36

  Completed  Complexity  Report of        0200-MAIN-PROCESSING  Paragraph
```

Figure E.7 The Reformatted COBOL Source Program Paragraph 0200-MAIN-PROCESSING Programming Syntax Complexity Measure Report.

```
  Started Complexity Report of      0900-TERMINATION      Paragraph
  Started Line Number               37

Verb Type  Modifier Type  V. Count  V. Weight.  M. Count  M. Weight.  Weighted Value
---------  -------------  --------  ----------  --------  ----------  --------------
 CLOSE     None              1          3          0          0             3

Total No. of Weighting  =           3
Total No. of Verbs      =           1
Total No. of Modifiers  =           0

 Total No. of Weighted Value / Total No. of Verbs   is
   Average Verb Complexity Weighted Value in Paragraph  =          3.00

 Total No. of Weighted Value is
   Paragraph     Complexity      Weighted     Value     =              3
  Ended   Line Number             39

  Completed  Complexity  Report of        0900-TERMINATION      Paragraph
```

Figure E.8 The Reformatted COBOL Source Program Paragraph 0900-TERMINATION Programming Syntax Complexity Measure Report.

Started Complexity Report of 1000-LOOP-BEGIN Paragraph
Started Line Number 40

Verb Type	Modifier Type	V. Count	V. Weight.	M. Count	M. Weight.	Weighted Value
MOVE	None	2	3	0	0	6
PERFORM	Imperative	1	5	0	1	5
WRITE	END/INVALID	2	3	0	1	6

Total No. of Weighting = 17
Total No. of Verbs = 5
Total No. of Modifiers = 0

Total No. of Weighted Value / Total No. of Verbs is
Average Verb Complexity Weighted Value in Paragraph = 3.40

Total No. of Weighted Value is
Paragraph Complexity Weighted Value = 17
Ended Line Number 48

Completed Complexity Report of 1000-LOOP-BEGIN Paragraph

Figure E.9 The Reformatted COBOL Source Program Paragraph 1000-LOOP-BEGIN Programming Syntax Complexity Measure Report.

Started Complexity Report of 2000-CHECK-PROCESS-READ Paragraph
Started Line Number 49

Verb Type	Modifier Type	V. Count	V. Weight.	M. Count	M. Weight.	Weighted Value
ADD	ON SIZE	1	3	0	0	3
IF	ELSE	1	5	1	1	6
PERFORM	Imperative	5	5	1	1	26

Total No. of Weighting = 35
Total No. of Verbs = 7
Total No. of Modifiers = 2

Total No. of Weighted Value / Total No. of Verbs is
Average Verb Complexity Weighted Value in Paragraph = 5.00

Total No. of Weighted Value is
Paragraph Complexity Weighted Value = 35
Ended Line Number 58

Completed Complexity Report of 2000-CHECK-PROCESS-READ Paragraph

Figure E.10 The Reformatted COBOL Source Program Paragraph 2000-CHECK-PROCESS-READ Programming Syntax Complexity Measure Report.

Started Complexity Report of 3000-LOOP-END Paragraph
Started Line Number 59

Verb Type	Modifier Type	V. Count	V. Weight.	M. Count	M. Weight.	Weighted Value
DIVIDE	ON SIZE	2	4	0	1	8
PERFORM	Imperative	1	5	0	1	5

Total No. of Weighting = 13
Total No. of Verbs = 3
Total No. of Modifiers = 0

Total No. of Weighted Value / Total No. of Verbs is
Average Verb Complexity Weighted Value in Paragraph = 4.33

Total No. of Weighted Value is
Paragraph Complexity Weighted Value = 13
Ended Line Number 66

Completed Complexity Report of 3000-LOOP-END Paragraph

Figure E.11 The Reformatted COBOL Source Program Paragraph 3000-LOOP-END Programming Syntax Complexity Measure Report.

Started Complexity Report of 4000-CHECK-DATA Paragraph
Started Line Number 67

Verb Type	Modifier Type	V. Count	V. Weight.	M. Count	M. Weight.	Weighted Value
IF	ELSE	2	5	2	1	12
MOVE	None	4	3	0	0	12

Total No. of Weighting = 24
Total No. of Verbs = 6
Total No. of Modifiers = 2

Total No. of Weighted Value / Total No. of Verbs is
Average Verb Complexity Weighted Value in Paragraph = 4.00

Total No. of Weighted Value is
Paragraph Complexity Weighted Value = 24
Ended Line Number 76

Completed Complexity Report of 4000-CHECK-DATA Paragraph

Figure E.12 The Reformatted COBOL Source Program Paragraph 4000-CHECK-DATE Programming Syntax Complexity Measure Report.

Started Complexity Report of 4100-PROCESS-DATA Paragraph
Started Line Number 77

Verb Type	Modifier Type	V. Count	V. Weight.	M. Count	M. Weight.	Weighted Value
ADD	ON SIZE	4	3	0	1	12
DIVIDE	ON SIZE	1	4	0	1	4
MOVE	None	5	3	0	0	15

Total No. of Weighting = 31
Total No. of Verbs = 10
Total No. of Modifiers = 0

Total No. of Weighted Value / Total No. of Verbs is
Average Verb Complexity Weighted Value in Paragraph = 3.10

Total No. of Weighted Value is
Paragraph Complexity Weighted Value = 31
Ended Line Number 94

Completed Complexity Report of 4100-PROCESS-DATA Paragraph

Figure E.13 The Reformatted COBOL Source Program Paragraph 4100-PROCESS-DATA Programming Syntax Complexity Measure Report.

Started Complexity Report of 4200-READ-DATA-TEST Paragraph
Started Line Number 95

Verb Type	Modifier Type	V. Count	V. Weight.	M. Count	M. Weight.	Weighted Value
MOVE	None	1	3	0	0	3
READ	END/INVALID	1	3	1	1	4

Total No. of Weighting = 7
Total No. of Verbs = 2
Total No. of Modifiers = 1

Total No. of Weighted Value / Total No. of Verbs is
Average Verb Complexity Weighted Value in Paragraph = 3.50

Total No. of Weighted Value is
Paragraph Complexity Weighted Value = 7
Ended Line Number 100

Completed Complexity Report of 4200-READ-DATA-TEST Paragraph

Figure E.14 The Reformatted COBOL Source Program Paragraph 4200-READ-DATA-TEST Programming Syntax Complexity Measure Report.

```
  Started Complexity Report of     4300-WRITE-LINE       Paragraph
  Started Line Number              101

Verb Type  Modifier Type  V. Count  V. Weight.  M. Count  M. Weight.  Weighted Value
---------  -------------  --------  ----------  --------  ----------  --------------
 WRITE     END/INVALID        1          3          0          1              3


Total No. of Weighting  =          3
Total No. of Verbs      =          1
Total No. of Modifiers  =          0

 Total No. of Weighted Value / Total No. of Verbs   is
   Average Verb Complexity Weighted Value in Paragraph  =          3.00

 Total No. of Weighted Value is
   Paragraph     Complexity      Weighted     Value     =              3
  Ended   Line Number             105

  Completed  Complexity  Report of        4300-WRITE-LINE       Paragraph
```

Figure E.15 The Reformatted COBOL Source Program Paragraph 4300-WRITE-LINE Programming Syntax Complexity Measure Report.

```
  Started Complexity Report of     4400-WRITE-ERROR      Paragraph
  Started Line Number              106

Verb Type  Modifier Type  V. Count  V. Weight.  M. Count  M. Weight.  Weighted Value
---------  -------------  --------  ----------  --------  ----------  --------------
 MOVE      None               3          3          0          0              9
 WRITE     END/INVALID        1          3          0          1              3


Total No. of Weighting  =         12
Total No. of Verbs      =          4
Total No. of Modifiers  =          0

 Total No. of Weighted Value / Total No. of Verbs   is
   Average Verb Complexity Weighted Value in Paragraph  =          3.00

 Total No. of Weighted Value is
   Paragraph     Complexity      Weighted     Value     =             12
  Ended   Line Number             114

  Completed  Complexity  Report of        4400-WRITE-ERROR      Paragraph
```

Figure E.16 The Reformatted COBOL Source Program Paragraph 4400-WRITE-ERROR Programming Syntax Complexity Measure Report.

```
  Started Complexity Report of     4500-WRITE-SUMMARY     Paragraph
  Started Line Number              115

Verb Type  Modifier Type  V. Count  V. Weight.  M. Count  M. Weight.  Weighted Value
---------  -------------  --------  ----------  --------  ----------  --------------
  MOVE     None               10          3         0         0             30
 WRITE     END/INVALID         3          3         0         1              9

Total No. of Weighting  =          39
Total No. of Verbs      =          13
Total No. of Modifiers  =           0

 Total No. of Weighted Value / Total No. of Verbs   is
   Average Verb Complexity Weighted Value in Paragraph  =          3.00

 Total No. of Weighted Value is
   Paragraph     Complexity      Weighted     Value      =             39
  Ended   Line Number             132

  Completed  Complexity  Report of        4500-WRITE-SUMMARY     Paragraph
```

Figure E.17 The Reformatted COBOL Source Program Paragraph 4500-WRITE-SUMMARY Programming Syntax Complexity Measure Report.

The Complexity Report of COBOL ANALYSER SYSTEM Rev. 19.1 ....

Verb Type	Modifier Type	V. Count	V. Weight.	M. Count	M. Weight.	Weighted Value
ADD	ON SIZE	5	3	0	1	15
CLOSE	None	1	3	0	0	3
DIVIDE	ON SIZE	3	4	0	1	12
IF	ELSE	3	5	3	1	18
MOVE	None	25	3	0	0	75
OPEN	None	1	3	0	0	3
PERFORM	Imperative	13	5	1	1	66
READ	END/INVALID	1	3	1	1	4
STOP	None	1	1	0	0	1
WRITE	END/INVALID	7	3	0	1	21

Total No. of Weighting = 218
Total No. of Verbs = 60
Total No. of Modifiers = 5
Total No. of Sections = 0
Total No. of Paragraphs = 13

Average No. of Verbs per Module is
Total No. of Verbs / Total No. of Modules (Sections + Paragraphs)
Average No. of Verbs per Module is = 4.615

Average Module Complexity Value is
Total No. of Weighted Value / Total No. of Modules (Sections + Paragraphs)
Average Module Complexity Value = 16.769

Average Statement Complexity Weighted Value is
Total No. of Weighted Value / Total No. of Verbs
Average Statement Complexity Value = 3.633

Program Complexity Weighted Value is
Total No. of Weighted Value = 218

Completed Complexity Report of COBOL ANALYSER SYSTEM Rev. 19.1 ####

Figure E.18 The Reformatted COBOL Source Program
Programming Syntax Complexity Measure Report.

APPENDIX F

BIBLIOGRAPHY

AND

REFERENCES

APPENDIX F : BIBLIOGRAPHY AND REFERENCES

+=====================================+
+ +
+ +
*+ **BIBLIOGRAPHY** and **REFERENCES** +*
+ +
+ +
+=====================================+

[ACK82] ACKERMAN, W. B.
Data Flow Languages.
Computer, Vol. 15, No. 2, February 1982,
pp. 15-25. 1982.

[ACM78] ACM, SIGPLAN
History of Programming Languages Conference.
ACM SIGPLAN Notices, Vol. 13, No. 8, August 1978,
1978. 310 p. COBOL Section. pp. 119-161.

[AHO72] AHO, ALFRED V. and ULLMAN, JEFFREY D.
Theory of Parsing, Translation and Compiling.
Vol. 1, Parsing; Vol. 2, Compiling.
Englewood Cliffs, New Jersey : Prentice-Hall,
1972.

[AHO74] AHO, ALFRED V. and JOHNSON, S. C.
LR Parsing.
ACM Computing Surveys, Vol. 6, No. 2, 1974,
pp. 99-124. 1974.

[AHO77] AHO, ALFRED V. and ULLMAN, JEFFREY D.
Principles of Compiler Design.
Massachusetts : Addison-Wesley,
1977. 604 p.

[ALE76] ALLEN, F. E. and COCKE, J.
A Program Data Flow Analysis Procedure.
Communications of the ACM, Vol. 19, No. 3, 1976,
pp. 137-147. 1976.

[ALF77] ALFORD, MACK
A Requirements Engineering Methodology for Real Time Processing Requirements.
IEEE Transactions on Software Engineering, Vol SE-3, No. 1, January 1977, pp. 60-69. 1977.

[ALF85] ALFORD, MACK
SREM at the Age of Eight; The Distributed Computing Design System.
Computer, April 1985, pp. 36-46. 1985.

[ALL70] ALLEN, F. E.
Control Flow Analysis.
ACM SIGPLAN Notices, Vol. 5, 1970, pp. 1-19. 1970.

[AMS76] AMSTER, S. J. and et al.
An experiment in automatic quality evaluation of Software.
In Proceedings Symposium Computer Software Engineering, MRI Symposia Series, Vol. XXIV, Fox, J. Editor, Polytechnic Institute of New York, April 1976, pp. 592-605. 1976.

[AND77] ANDRASFAI, BELA
Introductory Graph Theory.
Bristol : Adam Hilger, 1977. 268 p.

[AND82] ANDERSON, R. G.
A Dictionary of Data Processing and Computer Terms.
MacDonald and Evans Ltd., 1982. 108 p.

[ART84] ARTHUR, JAY
Software Quality Measurement.
Datamation, December 15, 1984, pp. 115-120. 1984.

[AUE80] AUERBACH, ISAAC L.
The AUERBACH Annual 1980 : Best Computer Papers.
New York : North Holland, 1980. 412 p.

[AUG79] AUGENSTEIN, MOSHE and AARON, TENENBAUM
Data Structures and PL/I Programming.
Englewood Cliffs, New Jersey : Prentice-Hall Inc.,
1979. 643 p.

[BAB85] BABB II, ROBERT G. and KIEBURTZ, RICHARD
Workshop on Models and Languages for Software Specification and Design.
Computer, March 1985,
pp. 103-108. 1985.

[BAC57] BACKUS, J. W. and et al.
The FORTRAN Automatic Coding System.
In Proceedings Western Joint C. C., Vol. 11, February 1957,
188 p. 1957.

[BAK72] BAKER, F. TERRY
System Quality Through Structured Programming.
In 1972 Fall Joint Computer Conference AFIPS Conference Proceedings 41, part 1,
Montvale, New Jersey : AFIPS Press, 1972,
pp. 339-343. 1972.

[BAK74] BAKER, F. TERRY
Organising For Structured Programming.
Lecture Notes in Computer Science 23,
Programming Methodology : 4th Informatik Symposium,
IBM Germany Wildbad, September 25-27, 1974.

[BAK75] BAKER, F. TERRY
Structured Programming in a Production Programming Environment.
ACM SIGPLAN Notices, Vol. 10, June 1975,
pp. 172-185. 1975.

[BAK77] BAKER, B. S.
An Algorithm for Structuring Flowgraphs.
Communications of the ACM, Vol. 24, January 1977,
pp. 98-120. 1977.

[BAK79a] BAKER, ALBERT L.
Software Science and Program Complexity Measures.
Thesis, Ph.D., Ohio State University,
1979. 276 p.

[BAK79b] BAKER, ALBERT L. and ZWEBEN, S. H.
The use of Software Science in Evaluating Modularity Concepts.
IEEE Transactions on Software Engineering,
Vol SE-5, March 1979,
pp. 110-120. 1979.

[BAK80] BAKER, ALBERT L.
A Comparison of Measures of Control Flow Complexity.
IEEE Transactions on Software Engineering,
Vol SE-6, No. 6, November 1980,
pp. 506-512. 1980.

[BAR80] BARTH, CARL WRANDLE
The Design of a High-Level Coroutine Implementation
For the Production of Quality Software.
Thesis, Ph.D., University of Maryland,
1980. 268 p.

[BAS75] BASIU, V. R. and TURNER, A. J.
Iterative Enhancement : A Practical Technique for
Software Development.
IEEE Transactions on Software Engineering,
Vol 1, December 1975,
pp. 390-396. 1975.

[BAS80] BASTANI, FAROKH BOMAN
An Input Domain Based Theory of Software Reliability
and its Application.
Thesis, Ph.D., University of California, Berkeley,
1980. 175 p.

[BEI78] BEINEKE, LOWELL W. and WILSON, ROBIN J.
Selected Topics in Graph Theory.
London : Academic Press,
1978. 451 p.

[BEI83] BEIZER, BORIS
Software Testing Techniques.
New York : Van Nostrand Reinhold Company,
1983. 290 p.

[BEL71] BELADY, L. A. and LEHMAN, M. M.
Programming Systems Dynamics or the Metadynamics of
Systems in Maintenance and Growth.
IBM Research Report RC 3546, 1971.

[BEL72] BELADY, L. A. and LEHMAN, M. M.
An Introduction to Growth Dynamics in Statistical Computer Performance Evaluation.
Academic Press : New York,
1972. p. 503-511.

[BEL76] BELADY, L. A. and LEHMAN, M. M.
A Model of Large Program Development.
IBM Systems Journal, Vol. 15, No. 3, 1976,
pp. 225-252. 1976.

[BEL77] BELL, T. E. and et al.
An Extendable Approach to Computer Aided Software Requirements Engineering.
IEEE Transactions on Software Engineering,
Vol SE-3, No. 1, January 1977,
pp. 49-60. 1977.

[BEL79] BELADY, L. A.
Complexity of Programming : A Brief Summary.
In Proceedings of the Workshop on Quantitative Models of Software Reliability, Complexity, and Cost. New York : IEEE Cat. #TH0067-9, 1979, pp. 90-94. 1979.

[BER71] BERRY, D. M.
Introduction to Oregano.
ACM SIGPLAN Notices, Vol. 6, No. 2, 1971,
pp. 171-190. 1971.

[BER83] BERST, J.
The ABC's of Evaluating Packaged Software.
Interface Age, Vol. 8, No. 2, February 1983,
pp. 35-37. 1983.

[BJO82] BJORN-ANDERSEN, NIELS and et al.
Information Society : For Richer, For Poorer.
Amsterdam : North-Holland Publishing Company,
1982. 320 p.

[BLA85] BLAIR, DAVID C. and MARON, M. E.
An Evaluation of Retrieval Effectiveness for a Full-text Document-retrieval System.
Communications of the ACM, Vol. 28, No. 3, March 1985,
pp. 289-299. 1985.

[BOE73] BOEHM, BARRY W.
Software and Its Impact : A Quantitative Assessment.
Datamation, Vol. 19, May 1973,
pp. 48-59. 1973.

[BOE76a] BOEHM, BARRY W. and et al.
Quantitative evaluation of Software Quality.
In Proceedings 2nd International Conference of
Software Engineering, San Francisco, October 1976,
pp. 592-605. 1976.

[BOE76b] BOEHM, BARRY W.
Software Engineering.
IEEE Transactions on Computers,
Vol C-25, No. 12, December 1976,
pp. 1226-1242. 1976.

[BOE79] BOEHM, BARRY W.
Software Engineering - As It Is.
IEEE Fourth International Conference on Software Engineering,
September 1979,
pp. 11-21. 1979.

[BOE81] BOEHM, BARRY W.
Software Engineering Economics.
Englewood Cliffs, New Jersey : Prentice-Hall,
1981. 767 p.

[BOE84] BOEHM, BARRY W.
Verifying and Validating Software Requirements and
Design Specifications.
IEEE Transactions on Software Engineering,
Vol 1, No. 1, January 1984,
pp. 75-88. 1984.

[BOF82] BOFFEY, T. B.
Graph Theory in Operation Research.
London : The Macmillan Press Ltd,
1982. 301 p.

[BOH75] BOHRER, R.
Halstead's Criterion and Statistical Algorithms.
In Proceedings of Computer Science and Statistics :
Eighth Annual Symposium on the Interface,
Health Science Computing Facility, UCLA, February 1975,
pp. 262-266. 1975.

[BOL82] BOLLOBAS, BELA.
Graph Theory : Proceedings of the Conference on Graph Theory, Cambridge.
Amsterdam : North-Holland Publishing Company, 1982. 201 p.

[BOR85] BORGIDA, ALEXANDER and et al.
Knowledge Representation as the Basis for Requirements Specifications.
Computer, April 1985,
pp. 82-91. 1985.

[BOY81] BOYER, R. S. and MOORE, STROTHER
The Correctness Problem in Computer Science.
London : Academic Press,
1981. 279.

[BRA77] BRADY, J. M.
The Theory of Computer Science : A Programming Approach.
London : Chapman and Hall,
1977. 287 p.

[BRA84] BRANSCOMB, L. M. and THOMAS, J. C.
Ease of use : A System Design Challenge.
IBM Systems Journal, Vol. 23, No. 3, 1984,
pp. 224-235. 1984.

[BRO74] BROWN, P.
Programming and Documentating Software Projects.
ACM Computing Surveys, Vol. 6, No. 4, December 1974,
pp. 213-220. 1974.

[BRO75] BROOKS, F. P.
The Mythical Man-Month.
Massachusetts : Addison-Wesley Publishing Company., Inc.,
1975.

[BRO78] BROOKES, R.
Using a Behavioral Theory of Program Comprehension in Software Engineering.
In Proceedings of the 3rd. IEEE Conference on Software Engineering,
pp. 196-200. 1978.

[BRO82] BROOKES, CYRIL H. P. and et al.
Information Systems Design.
Prentice-Hall of Australia,
1979. 477 p.

[BRO83] BROWN, P. J.
Error Messages : The Neglected area of the
Man/Machine Interface ?
Communication of the ACM, Vol. 26, No. 4, April 1983,
pp. 246-249. 1983.

[BRO84] BROOKS, COLIN and SMITH, JOHN
How to Choose a Fourth Generation System
Development Environment.
Data Processing, Vol. 26, No. 9, November 1984,
pp. 23-25. 1984.

[BRO85] BROWN, PATRICK
Managing Software Development.
Datamation, May 15, 1985,
pp. 133-136. 1985.

[BUC63] BUCHHOLZ, W.
File organization and addressing.
IBM Systems Journal, Vol. 2, June 1963,
pp. 86-111. 1963.

[BUR68] BURKHARDT,W. H.
Metalanguage and Syntax Specification.
Communications of the ACM, Vol. 8, No. 5, 1968,
pp. 304-305. 1968.

[BUR82] BURTON, PHILIPS E.
A Dictionary of Minicomputing and Microcomputing.
New York : Garland STPM Press,
1982. 346 p.
Appendix A, pp. 281-288.

[BUR84] BURSTEIN, DANIEL
Singapore's 21st Century Dream.
Datamation, October 15, 1984,
pp. 155-158. 1984.

[BUS85] BUSH, CHANDLER M. and SCHKADE, LAWRENCE L.
In Search of the Perfect Programmer.
Datamation, March 15, 1985,
pp. 128-132. 1985.

[CAM84] CAMPBELL, J. A.
Implementations of PROLOG.
New York : John Wiley & Sons,
1984. 391 p.

[CAR79a] CARDENAS, ALFONSO F.
Data Base Management Systems.
Boston : Allyn and Bacon,
1979. 519 p.

[CAR79b] CARRE, BERNARD
Graphs and Networks.
Oxford : Oxford University Press,
1979. 277 p.

[CAR81] CARVER, DORIS LOVEDAY
The Effects of Complexity on COBOL Program Changes.
Thesis, Ph.D., Texas A & M University,
1981. 119 p.

[CAV78] CAVE, W. C. and SAHISBURY, B. A.
Controlling the Software Life Cycle :
The Project Management Task.
IEEE Transactions on Software Engineering,
Vol. SE-4, July 1978,
pp. 326-334, 1978.

[CHA77] CHAMPINE, G. A.
Six Approaches to Distributed Data Bases.
Datamation, May 1977,
1977.

[CHA79] CHAPIN, NED
A Measuring of Software Complexity.
In Proceedings AFIPS 79 National Computer Conference,
pp. 995-1002. 1979.

[CHA84] CHANG, CHIN CHEN
The Study of an Ordered Minimal Perfect Hashing Scheme.
Communications of the ACM, Vol. 27, No. 4, April 1984,
pp. 384-387. 1984.

[CHE78] CHEN, EDWARD T.
Program Complexity and Programmer Productivity.
IEEE Transactions on Software Engineering,
Vol. SE-4, No. 3, May 1978,
pp. 187-194. 1978.

[CIC80] CICHELLI, R. J.
Minimal Perfect Hashing Functions Made Simple.
Communications of the ACM, Vol. 23, No. 1, January 1980,
pp. 17-19. 1980.

[COB78] COBB, G. W.
A Measurement of Structure for Unstructured
Programming Languages.
In Proceedings of the Software Quality and
Assurance Workshop, November 1978,
pp. 140-147. 1978.

[COC70] COCKE, J.
Global Common Subexpression Elimination.
ACM SIGPLAN Notices, Vol. 5, 1970,
pp. 20-24. 1970.

[COL83] COLE, BERNARD
The Family Tree of Computer Languages.
Popular Computing, September 1983,
pp. 82-89. 1983.

[COL84] COLLINS, STEVE
Comparing MODULA-2 with PASCAL and ADA.
Data Processing, Vol. 26, No. 10, December 1984,
pp. 32-34. 1984.

[COM85] COMER, JAMES R. and et al.
Software Design and Development : A Graduate Curriculum in
Software Engineering.
ACM SIGCSE Bulletin, Vol. 17, No. 1, March 1985,
pp. 335-338. 1985.

[CON79] CONWAY, RICHARD and GRIES, DAVID
An Introduction to Programming :
A Structured Approach Using PL/I and PL/C.
Cambridge, Massachusetts : Winthrop Publishers,
1979. 728 p.

[CON82] CONNELL, J. and Brice, L.
Complexity Measures Applied to an Application Case Study.
In Proceedings Fourth International Conference on
Computer Capacity Management, 1982,
pp. 121-128. 1982.

[COO82] COOK, C. R. A.
Letter Oriented Minimal Perfect Hashing Function.
ACM SIGPLAN Notices, Vol. 17, No. 9, September 1982,
pp. 18-27. 1982.

[COO85] COOK, DEREK
Advances in Disc Storage Technology.
Data Processing, Vol. 27, No. 1, February 1985,
pp. 29-32. 1985.

[COR85] CORMACK, G. V. and et al.
Practical Perfect Hashing.
The Computer Journal, Vol. 28, No. 1, 1985,
pp. 54-58. 1985.

[COU72] COUGER, J. DANIEL
Evolution of Business System Analysis Techniques.
ACM Computing Surveys, Vol. 5, No. 3, September 1972,
pp. 167-198. 1972.

[CUL85] CULLUM, RONALD L.
Iterative Development.
Datamation, February 15, 1985,
pp. 93-98. 1985.

[CUR79a] CURTIS, W. and et al.
Measuring the Psychological Complexity of Software Maintenance Tasks with Halstead and McCabe Metrics.
IEEE Transactions on Software Engineering,
SE-5, No. 2, March 1979,
pp. 95-104, 1979.

[CUR79b] CURTIS, B.
In Search of Software Complexity.
In Proceedings of the Workshop on Quantitative Models of Software Reliability, Complexity, and Cost.
New York : IEEE Cat. #TH0067-9, 1979,
pp. 95-106. 1979.

[CUS60] CUSHING, GORDON
Automatic Programming and Business Applications.
Annual Review in Automatic Programming I.
Oxford : Pergamon Press,
pp. 189-195. 1960.

[CZA85] CZAJKIEWICZ, ZBIGNIEW J.
Optimization of the Maintenance Process.
Simulation, March 1985,
pp. 137-141. 1985.

[DAH72] DAHL, O. J. and et al.
Structured Programming.
New York : Academic Press, Inc.,
1972.

[DAL77] DALY, EDMUND B.
Management of Software Development.
IEEE Transactions on Software Engineering,
Vol SE-3, No. 2, 1977,
pp. 229-242. 1977.

[DAT81] DATE, C. J.
An Introduction to Database Systems.
3rd Ed. Massachusetts : Addison-Wesley,
1981. 574 p.

[DAT85] DATA DECISIONS
The Applications Software Survey.
Datamation, May 1, 1985,
pp. 118-138. 1985.

[DAV80] DAVIS, A. M.
Automating the Requirements Phase : Benefits to late Phases of the Software Life Cycle.
IEEE Fourth International Computer Software and Applications Conference, October 1980,
pp. 42-48. 1980.

[DEE85] DEE, DAVID
Developing PC Applications.
Datamation, February 15, 1985,
pp. 112-116. 1985.

[DEM78] DeMACRO, T.
Structured Analysis and System Specification.
New York : Yourdon, Inc.,
1978.

[DEN80a] DENCE, THOMAS P.
The Fortran Cookbook.
Tab Books Inc.,
1972. 334 p.

[DEN80b] DENOLA, LYNN ANDREWS
Performance and Timeliness in a Distributed Database.
Thesis, Ph.D., Brown University,
1980. 221 p.

[DEU82] DEUTSCH, MICHAEL S.
Software Verification and Validation :
Realistic Project Approaches.
Englewood Cliffs, New Jersey : Prentice-Hall,
1982. 327 p.

[DIJ68] DIJKSTRA, E. W.
GoTo Statement Considered Harmful.
Communication of the ACM, Vol. 11, 1968,
pp. 147-148. 1968.

[DIJ70] DIJKSTRA, E. W.
EWD 249 : Notes on Structured Programming
Report 70-WSK-03.
Tech. University Eindhoven,
1970. 84 p.

[DIJ72a] DIJKSTRA, E. W. and DAHL, O. J.
Notes on Structured Programming in : Structured Programming.
London : Academic Press,
1972. pp. 1-82.

[DIJ76] DIJKSTRA, E. W.
Structured Programming in Software Engineering Techniques.
New York : Petrocelli/Charter,
pp. 222-226. 1976.

[DOR72a] DORAN, B. and TATE, GRAHAM. N.
An Approach to Structured Programming :
Part I and Part II.
Department of Computer Science :
Massey University Publications,
1972.

[DOR72b] DORAN, B. and TATE, GRAHAM. N.
An Approach to Structured Programming :
Solutions to Exercises.
Department of Computer Science :
Massey University Publications,
June 1972.

[DRO85] DROMEY, R. GEOFF
Program Development by Inductive Stepwise Refinement.
Software Practice and Experience,
Vol. 15, No. 1, January 1985,
pp. 1-28. 1985.

[DRU85] DRUMMOND, STEVE
Measuring Applications Development Performance.
Datamation, February 15, 1985,
pp. 102-108. 1985.

[DUD81] DUDA, R. O. and GASCHNIG, J. G.
Knowledge-Based Expert Systems Come of Age.
Byte, Vol. 6, No. 9, September 1981,
pp. 238-281. 1981.

[DUN78] DUNSMORE, H. E. and GANNON, J. D.
Programming Factors - Language Features that help Explain Programming Complexity.
In Proceedings ACM 78 Annual Conference,
Washington, DC, 1978,
pp. 554-560. 1978.

[DUN80] DUNSMORE, H. E. and GANNON, J. D.
Analysis of the Effects of Programming Factors on Programming Efforts.
The Journal of Systems and Software, 1980,
pp. 141-153. 1980.

[DU 80] DU, M. W. and et al.
The Study of a New Perfect Hash Schemes.
In Proceedings, COMPSAC'80, Chicago, IL, October 1980,
pp. 341-347. 1980.

[ELG76] ELGOT, CALVIN C.
Structured Programming With and Without GO TO Statements.
IEEE Transactions on Software Engineering,
Vol. SE-2, No. 1, March 1976,
pp. 41-54. 1976.

[ELS76] ELSHOFF, JAMES L.
An Analysis of Some Commercial PL/I Program.
IEEE Transactions on Software Engineering,
Vol. SE-2, No. 2, June 1976,
pp. 113-120. 1976.

[ELS77] ELSHOFF, JAMES L.
The Influence of Structured Programming on PL/I Program Profiles.
IEEE Transactions on Software Engineering, Vol. SE-3, No. 5, September 1977, pp. 364-368. 1977.

[ENR72] ENRICK, NORBERT LLOYD
Effective Graphic Communication.
Princeton : Auerbach Publishers Inc., 1972. 184 p.

[FER81] FERGUSON, R.
PROLOG : A Step Toward the Ultimate Computer Language.
Byte, Vol. 6, No. 11, November 1981, pp. 384-399. 1981.

[FER83] FERNANDEZ, EDUARDO B. and et al.
Database Security and Integrity.
Massachusetts : Addison-Wesley, 1983. pp. 46, pp. 50-52, pp. 55-81, pp. 107-143.

[FEU79] FEUER, A. R. and FOWLKES, E. B.
Relating Computer Program Maintainability to Software Measures.
In Proceedings AFIPS 79 National Computer Conference, pp. 1003-1012. 1979.

[FIO84] FIORELLO, MARCO
Is COBOL-8x Cost effective ?
In Proceedings AFIPS 84 National Computer Conference, pp. 223-228. 1977.

[FIS80] FISCHER, KURT F.
A Graph Theoretic Approach To the Validation of Software Maintenance Modifications.
Thesis, Ph.D., University of California, Los Angeles, 1980. 141 p.

[FIT78] FITZSIMMONS, A. and LOVE, T.
A Review and Evaluation of Software Science.
ACM Computing Surveys, Vol. 10, March 1978, pp. 3-18. 1978.

[FOS75] FOSDICK, L. D. and OSTERWEIL, L. J.
DAVE - A Fortran Program Analysis System.
In Proceedings of Computer Science and Statistics : Eighth Annual Symposium on the Interface, Health Science Computing Facility, UCLA, February 1975, pp. 329-335. 1975.

[FOX82] FOX, JOSEPH M.
Software and its Development.
Englewood Cliffs, New Jersey : Prentice-Hall, 1982. 299 p.
pp. 1-13, pp. 34-277.

[FRE75] FREEMAN, PETER.
Software Systems Principles: A Survey.
Science Research Associates, Inc., 1975. 663 p.
pp. 1-13, pp. 34-277.

[FRI85] FRIEDMAN, ANDREW and GREENBAUM, JOAN
Japanese DP.
Datamation, February 15, 1985, pp. 112-118. 1985.

[GAR72] GARFINHEL, ROBERT and NEMHANSER, G. L.
Integer Programming.
New York : John Wiley and Sons, Inc., 1972.

[GAR84] GARFUNKEL, JEROME
COBOL-80 : The New Structured Language.
In Proceedings AFIPS 84 National Computer Conference, pp. 217-221. 1984.

[GEP77] GEPNER, H. R.
User Ratings of Software Packages.
Datamation, December 1977, pp. 117-162. 1977.

[GER81] GERMANO, FRANK
Automatic Transaction Decomposition in a Distributed CODASYL Prototype System.
Ann Arbor, Michigan : UMI Research Press, 1981. 134 p.

[GID74] GIDEADI, AMOS N.
On a Proposed Measure of Program Structure.
ACM SIGPLAN Notices, Vol. 9, No. 5, May 1974,
pp. 31-36. 1974.

[GIL84] GILLIN, P.
IBM Picks Comshare DSS for Information Center.
Computerworld, Vol. 18, No. 2, January 9, 1984,
pp. 5, 1984.

[GLA69] GLASS, ROBERT L.
An Elementary Discussion of Compiler/Interpreter Writing.
ACM Computing Surveys, Vol. 1, No. 1, March 1969,
pp. 55-77. 1969.

[GLA81] GLASS, ROBERT L.
Software Reliability Guidebook.
Englewood Cliffs, New Jersey : Prentice-Hall,
1981. 242 p.
p. 27-225.

[GLA82] GLASS, ROBERT L.
Modern Programming Practices : A Report from Industry.
Englewood Cliffs, New Jersey : Prentice-Hall,
1982. 311 p.

[GNA81] GNANAMGARI, SAKUNTHALA
Information Presentation Through Default Displays.
Thesis, Ph.D., University of Pennsylvania,
1981. 118 p.

[GON84] GONNET, G. H.
Handbook of Algorithms and Data Structures.
London : Addison-Wesley Publishing Company,
1984. 268 p.

[GOO77] GOODMAN, S. E. and HEDETNIEMI, S. T.
Introduction to the Design and Analysis of Algorithms.
Tokyo : McGraw-Hill Kogakusha Ltd.,
1977. 371 p.

[GOR76] GORDON, R. D. and HALSTEAD, M. H.
An Experiment Comparing FORTRAN Programming Time with the Software Physics Hypothesis.
In Proceedings AFIPS 76 National Computer Conference,
Vol. 45, 1976,
pp. 935-937. 1976.

[GOR79] GORDON, R. D.
Measuring Improvements in Program Clarity.
IEEE Transactions on Software Engineering,
Vol. SE-5, No. 2, March 1979,
pp. 79-90. 1979.

[GOU74] GOULD, J. D. and DRONGOWSKI, M.
An Exploratory Study of Computer Program Debugging.
Journal of Human Factors,
Vol. 16, No. 3, 1974,
pp. 258-277. 1974.

[GOU81] GOURLAY, JOHN STEWART
Theory of Testing Computer Programs.
Thesis, Ph.D., The University of Michigan,
1981. 132 p.

[GRA77] GRACE, ALONZO G. JR.
The Dimensions of Complexity.
Datamation, September 1977,
pp. 315-318. 1977.

[GRA85] GRANT, F. J.
Twenty-First Century Software.
Datamation, April 1, 1985,
pp. 123-130. 1985.

[GRE76] GREEN, T. F. and et al.
Program Structures, Complexity and Error Characteristics.
In Proceedings of the Symposium on Computer Software
Engineering, Polytechnic Press of the Polytechnic Institute
of New York, Brooklyn, New York,
pp. 139-154. 1976.

[GRI71] GRIES, DAVID
Compiler Construction For Digital Computers.
New York : John Wiley & Sons,
1971. 493 p.

[GRO74] GROVES, LINDSAY J.
The Provision of Debugging Facilities for
High Level Languages.
Thesis, M.Sc., Massey University,
1974. 167 p. pp. 3-25.

[HAL72] HALSTEAD, M. H.
Natural Laws Controlling Algorithm Structure ?
ACM SIGPLAN Notices, Vol. 7, No. 2, February 1972,
pp. 19-26. 1972.

[HAL75] HALSTEAD, M. H.
Toward a Theoretical Basis for Estimating Programming Effort.
In Proceedings of the ACM 1975, ACM, New York,
pp. 222-224. 1975.

[HAL77] HALSTEAD, M. H.
Elements of Software Science.
New York : Elsevier North-Holland,
1977. 127 p.

[HAN78] HANSON, WILFRED J.
Measurement of Program Complexity by the
Pair Cyclomatic Number, Operator Count.
ACM SIGPLAN Notices, Vol. 13, No. 3, March 1978,
pp. 29-33. 1978.

[HAN85] HANSON, STEPHEN JOSE and ROSINSKI, RICHARD R.
Programmer perceptions of Productivity and Programming Tools.
Communications of the ACM, Vol. 28, No. 2, February 1985,
pp. 180-189. 1985.

[HAR65] HARARY, FRANK and et al.
Structured Models : An Introduction to the
Theory of Directed Graphs.
New York : John Wiley & Sons Inc.,
1965. 415 p.

[HAR71] HARTMANIS, J. and HOPCROFT, J. E.
An Overview of the Theory of Computational Complexity.
Journal of the ACM, Vol. 18, No. 3, July 1971,
pp. 444-475. 1971.

[HEC72] HECHT, MATTHEW S. and ULLMAN, J. D.
Flow Graph Reducibility.
SIAM Journal of Computer, Vol. 1, 1972,
pp. 188-202. 1972.

[HEC77] HECHT, MATTHEW S.
Flow Analysis of Computer Programs.
New York : North-Holland,
1977. 232 p. pp. 1-10.

[HER85] HERRIN, WILLIAMS F.
Software Maintenance Costs : A Quantitative Evaluation.
ACM SIGCSE Bulletin, Vol. 17, No. 1, March 1985,
pp. 233-237. 1985.

[HIE74] HIEMANN, P.
A New Look at the Program Development Process.
Lecture Notes in Computer Science, No. 23,
Programming Methodology, 4th Informatik Symposium,
IBM Germany Wildbad, September 25-27, 1974,
pp. 11-37. 1974.

[HIG73] HIGBIE, L. C.
Supercomputer Architecture.
Computer, Vol. 6, No. 12, December 1973,
pp. 48-58. 1973.

[HIG78] HIGBIE, L. C.
Applications of Vector Processing.
Computer Design, April 1978,
pp. 139-145. 1978.

[HOA70] HOARE, C. A. R.
An Axiomatic Basis for Computer Programming.
Communications of the ACM, Vol. 12, October 1970,
pp. 576-583. 1970.

[HOA73] HOARE, C. A. R. and WIRTH, N.
An Axiomatic Definition of the Programming Language PASCAL.
ACTA Informatica, Vol. 2, 1973,
pp. 335-355. 1973.

[HOA78] HOARE, C. A. R.
Software Engineering : A Keynote Address.
3rd International Conference on Software Engineering,
May 1978,
1978.

[HOL82] HOLT, JOHN DOUGLAS
Analysis of Program Control Structure and Data Flow - with Applications for Program Complexity and Data Flow Machines.
Thesis, Ph.D., Oregon State University,
1982. 176 p.

[HOP69] HOPGOOD, F. R. A.
Compiling Techniques.
London : Macdonald,
1969. 126 p.

[HOW78] HOWKINS, T. J. and HARANDI, M. T.
Towards More Portable COBOL.
The Computer Journal, Vol. 22, No. 4, 1978,
pp. 290-295. 1978.

[JAC70] JACKSON, M. A.
JSP-COBOL Language Description and
Operating Characteristics.
London : Michael Jackson Systems Ltd.,
1970.

[JAE81] JAESCHKE, G.
Reciprocal Hashing : A Method for Generating Minimal
Perfect Hashing Functions.
Communications of the ACM, Vol. 24, No. 12, December 1981,
pp. 829-933. 1981.

[JOH61] JOHNSON, L. R.
An Indirect Chaining Method for Addressing on Secondary Keys.
Communications of the ACM, Vol. 4, No. 5, May 1961,
pp. 218-222. 1961.

[JOH71] JOHNSTON, J. B.
The Contour Model of Block Structured Processes.
ACM SIGPLAN Notices, Vol. 6, No. 2, 1971,
pp. 55-82. 1971.

[KAR60] KARP, R. M.
A Note on the Application of Graph Theory to
Digital Computer Programming.
Information Control, Vol. 3, 1960,
pp. 179-190. 1960.

[KAT76] KATZ, S. and MANNA, Z.
Logical Analysis of Programs.
Communications of the ACM, Vol. 19, April 1976,
pp. 188-206. 1976.

[KER79] KERR, EDWIN F. and SHAW, DAVID G.
Handbook of COBOL Techniques.
Wellesley, Massachusetts : Q.E.D. Information Sciences, Inc.,
1979. 86 p.

[KES76] KESSELL, OWEN DOUGLAS
COBOL Language implementation via English.
Thesis, M.Sc., Massey University,
1976. 93 p.

[KFO82] KFONRY, A. J. and et al.
A Programming Approach to Computability.
New York : Springer-Verlag,
1982. 251 p.

[KNU68] KNUTH, D. E.
The Art of Computer Programming, Vol. 1 :
Fundamental Algorithms.
Massachusetts : Addison-Wesley,
1968. 634 p.

[KNU69] KNUTH, D. E.
The Art of Computer Programming, Vol. 2 :
Seminumerical Algorithm.
Massachusetts : Addison-Wesley,
1969.

[KNU71] KNUTH, D. E.
An Empirical Study of FORTRAN Programs.
Software Practice and Experience, Vol. 15, June 1971,
pp. 105-133. 1971.

[KNU73] KNUTH, D. E.
The Art of Computer Programming, Vol. 3 :
Sorting and Searching.
Massachusetts : Addison-Wesley,
1973.

[KNU74] KNUTH, D. E.
Structured Programming with GO TO Statements.
ACM Computing Surveys, Vol. 6, December 1974,
pp. 261-301. 1974.

[KOB81] KOBAYASHI, K.
Computer, Communications and Man : The Integration of
Computer and Communications with Man as an Axis.
Computer Networks, Vol. 5, No. 4, July 1981,
pp. 237-250. 1981.

[KOP79] KOPETZ, HERMANN
Software Reliability.
London : Macmillan,
1979. 118 p.

[KOZ80] KOZDROWICKI, E. W. and THEIS, D. J.
Second Generation of Vector Supercomputers.
Computer, Vol. 13, No. 11, November 1980,
pp. 71-83. 1980.

[KRA81] KRAFT, DEAN BLACKMAR
AVID : A System for the Interactive Development of Verifiably Correct Programs.
Thesis, Ph.D., Cornell University,
1981. 136 p.

[KRU84] KRUSE, ROBERT L.
Data Structures and Program Design.
Englewood Cliffs : Prentice-Hall,
1984. 486 p.

[LAN64] LANDIN, P. J.
The Mechanical Evaluation of Expressions.
The Computer Journal, Vol. 6, No. 4, 1964,
pp. 308-320. 1964.

[LAN79] LANO, R. J.
A Technique For Software and Systems Design.
Amsterdam : North-Holland,
1979. 119 p.

[LAN82] LAN, KWAI-TING
A Study of Sampling, Granularity and Localities in Program Restructuring.
Thesis, Ph.D., Iowa State University,
1982. 206 p.

[LAS82] LASSNER, MICHAEL ALLEN
Graph Embedding Algorithms and Their Applications.
Thesis, Ph.D., Wayne State University,
1982. 130 p.

[LEA83] LEATHRUM, J. F.
Foundations of Software Design.
Reston Publishing Company Inc. : A Prentice-Hall Company,
1983. 182 p.

[LED75] LEDGARD, HENRY F.
Programming Proverbs.
Rochelle Park, New Jersey : Hayden Book Company, Inc.,
1975. 134 p.

[LED80] LEDGARD, HENRY F. and et al.
The Natural Language of Interactive Systems.
Communications of the ACM, Vol. 23, No. 10, October 1980,
pp. 556-563, 1980.

[LEG75] LEGARD, H. and MARCOTTY, M.
A Generalogy of Control Structures.
Communications of the ACM, Vol. 18, November 1975,
pp. 629-639. 1975.

[LIE78] LIENTZ, BENNET P. and et al.
Characteristics of Application Software Maintenance.
Communications of the ACM, Vol. 21, No. 6, 1978,
pp. 466-471. 1978.

[LIE79] LIENTZ, BENNET P. and SWANSON, E. B.
Software Maintenance A User/Management Tug-of-War.
Data Management, Vol. 17, No. 4, 1979,
pp. 26-30. 1979.

[LIE80] LIENTZ, BENNET P. and SWANSON, E. BURTON
Software Maintenance Management.
Massachusetts : Addison-Wesley,
1980. 214 p.

[LIE83] LIENTZ, BENNET P.
Issues in Software Maintenance.
ACM Computing Surveys, Vol. 15, No. 3, September 1983,
pp. 271-278. 1983.

[LIM80] LIM, PACIFICO A.
A Guide to structured COBOL :
With Efficiency and Techniques and special Algorithms.
New York : Van Nostrand Reinhold Company,
1980. 259 p.

[LIN73] LINDHORST, W. M.,
Scheduled Maintenance of Application Software.
Datamation, Vol. 19, 1973,
pp. 64-67, 1973.

[LIN77] LINSTONE, HAROLD A. and SIMMONDS, W. H. CLIVE
Futures Research : New Directions.
Massachusetts : Addison-Wesley Publishing Company,
1977. 270 p.

[LIN79] LINGER, RICHARD C. and et al.
Structure Programming : Theory and Practice.
Addison-Wesley Publishing Company,
1979. 402 p.

[LIS75] LISKOV, B. and ZILLES, S.
Specification Techniques for Data Abstractions.
ACM SIGPLAN Notices, Vol. 10, June 1975,
pp. 72-87. 1975.

[LIU76] LIU, C. C.
A Look at Software Maintenance.
Datamation, Vol. 22, No. 11, 1976,
pp. 51-55, 1976.

[LOV76] LOVE, L. T. and BOWMAN, A. B.
An Independent Test of the Theory of Software Physics.
ACM SIGPLAN Notices, Vol. 11, No. 11, November 1976,
1976.

[LOV77] LOVE, T.
An Experimental Investigation of the Effect of
Program Structure on Program Understanding.
ACM SIGPLAN Notices, Vol. 12, March 1977,
pp. 105-113. 1977.

[LOV78] LOVE, T. and FITZSIMMONS, A.
A Review and Evaluation of Software Science.
ACM Computing Surveys, Vol. 10, No. 1, March 1978,
pp. 3-18. 1978.

[LUK84] LUKAC, EUGENE G.
The Impact of a 4GL On Hardware Resources.
Datamation, October 1, 1984,
pp. 105-113. 1984.

[LUM71] LUM, V. Y. and et al.
Key-to-address Transform Techniques : A Fundamental
Performance Study on Large Existing Formatted Files.
Communications of the ACM, Vol. 14, No. 4, April 1971,
pp. 228-239. 1971.

[MAL81] MALLGREN, WILLIAM ROBERTS
Formal Specification of Interactive Graphics Programming Languages.
Thesis, Ph.D., University of Washington, 1981. 280 p.

[MAR60] MARIMONT, ROSALIND B.
Applications of Graphs and Boolean Matrices to Computer Programming.
SIAM Review, Vol. 2, No. 4, 1960, pp. 259-268. 1960.

[MAR83a] MARSH, ROBERT E.
Application Maintenance :
One Shop's Experience and Organization.
In Proceedings AFIPS 83 National Computer Conference, pp. 145-153. 1983.

[MAR83b] MARTIN, J. and McClure, C. L.
Software Maintenance : The Problem and its Solutions.
Englewood Cliffs, New Jersey : Prentice-Hall, 1983.

[MAU75] MAURER, W. D. and LEWIS, T. G.
Hash Table Method.
ACM Computing Surveys, Vol. 7, No. 1, 1975, pp. 5-19. 1975.

[MAY81] MAYNARD, JEFF
Dictionary of Data Processing.
2nd Ed. London : Butterworths, 1981. 275 p.

[MCC62] McCarthy, J. and et al.
LISP 1.5 Programmer's Manuals.
2nd Ed. Massachusetts : M. I. T., 1962.

[MCC76a] McCABE, THOMAS J.
A Complexity Measure.
IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December 1976, pp. 308-320. 1976.

[MCC76b] McCLURE, CARMA L.
Formalization and Application of Structured Programming and Program Complexity.
Thesis, Ph.D., Illinois Institute of Technology, 1976. 288 p.

[MCC78] McCLURE, CARMA L.
Reducing COBOL Complexity through Structured Programming.
New York : Van Nostrand Reinhold Company, 1978. 192 p.

[MCG75] McGOWAN, CLEMENT L. and KELLY, JOHN R.
Top Down Structured Programming Techniques.
New York : Van Nostrand Reinhold, 1975. 288 p.

[MCG80] McGETTRICK, ANDREW D.
The Definition of Programming Languages.
Cambridge : Cambridge University Press, 1980. 268 p.

[MCK74] McKEEMAN, W. M. and et al.
Compiler Construction.
Lecture Notes in Computer Science 21,
New York : Springer-Verlag, 1974. 621 p.

[MCK82] McKEOWN, G. P. and RAYWARD-SMITH, V. J.
Mathematics For Computing.
London : The Macmillan Press Ltd, 1982. 428 p.

[MCT80] McTAP, JOHN L.
The Complexity of an Individual Program.
In Proceedings AFIPS 80 National Computer Conference, pp. 767-771. 1980.

[MEE80] MEEK, BRAIN L. and HEATH, PATRICIA M.
Guide to Good Programming Practice.
Ellis Horwood Limited : Publisher Chichester Halsted Press, 1980. 181 p.

[MET81] METZGER, PHILIP W.
Managing a Programming Project.
2nd Ed. Englewood Cliffs, New Jersey : Prentice-Hall Inc., 1981. 244 p.
pp. 1-15.

[MIL74] MILLER, L. G.
Programming by Nonprogrammers.
International Journal of Man-Machine Studies, Vol. 6, 1974,
pp. 237-260. 1974.

[MIL75] MILLS, HARLAN D.
How to Write Correct Programs and Know it.
ACM SIGPLAN Notices, Vol. 10, June 1975,
pp. 363-370. 1975.

[MIL76] MILLS, HARLAN D.
Software Development.
IEEE Transactions on Software Engineering,
Vol. SE-2, No. 4, December 1976,
pp. 265-273. 1976.

[MIL81] MILI, ALI
Self-checking Programs : An Axiomatic Approach to
the Validation of Programs by the use of Assertions.
Thesis, Ph.D., Wayne State University,
1982. 130 p.

[MOH79] MOHANTY, S. N.
Models and Measurements for Quality Assessment of Software.
ACM Computing Surveys, Vol. 11, No. 3, 1979,
pp. 251-275. 1979.

[MOO75] MOONEY, J. W.
Organization Program Maintenance.
Datamation, Vol. 21, No. 2, February 1975,
pp. 63-66, 1975.

[MOR68] MORRIS, ROBERT.
Scatter Storage Techniques.
Communications of the ACM, Vol. 11, No. 1, January 1968,
pp. 38-43, 1968.

[MOR81] MORSE, ALAN CHARLES
Using Computer Graphics as an Aid to
Understanding Simulation Data.
Thesis, Ph.D., University of Massachusetts,
1981. 266 p.

[MOR83] MORLAND, D. VERNE
Human Factor Guidelines for Terminal Interface Design.
Communications of the ACM, Vol. 26, No. 7, July 1983,
pp. 484-494, 1983.

[MOR85] MORLAND, D. VERNE
The Evolution of Software Architecture.
Datamation, February 1, 1985,
pp. 123-132. 1985.

[MOT82] MOTO-OKA, TOHRU
Fifth Generation Computer Systems.
Amsterdam : North-Holland Publishing Company,
1982. 287 p.

[MUC81] MUCHNICK, STEVEN S. and JONES, NEIL D.
Program Flow Analysis : Theory and Applications.
Englewood Cliffs, New Jersey : Prentice-Hall,
1981. 415 p.

[MYE75] MYERS, GLENFORD J.
Reliable Software Through Composite Design.
New York : Van Nostrand Reinhold,
1975. 159 p.

[MYE76] MYERS, GLENFORD J.
Composite Design Facilities of six Programming Languages.
IBM Systems Journal, Vol. 15, No. 3, 1976,
pp. 212-224. 1976.

[MYE77] MYERS, GLENFORD J.
An Extension to the Cyclomatic measure of Program Complexity.
ACM SIGPLAN Notices, Vol. 12, No. 10, October 1977,
pp. 62-64. 1977.

[MYE78] MYERS, GLENFORD J.
Composite/Structured Design.
New York : Van Nostrand Reinhold,
1978. 174 p.

[MYE85] MYERS, WARE
An Assessment of the Competitiveness of the
United States Software Industry.
Computer, March 1985,
pp. 81-92. 1985.

[NAM81] NAM, CHONG WOO
Software Requirements Engineering :
Experience and New Techniques.
Thesis, Ph.D., University of California, Berkeley,
1981. 154 p.

[NAS73] NASSI, I and SHNEIDERMAN, B.
Flowchart Techniques for Structured Programming.
ACM SIGPLAN Notices, Vol. 8, No. 8, 1973,
pp. 12-26. 1973.

[NEE76] NEELY, P. M.
The New Programming Discipline.
Software Practice and Experience, Vol. 6, 1976,
pp. 7-27. 1976.

[NIC75] NICHOLLS, JOHN E.
The Structure and Design of Programming Languages.
Massachusetts : Addison-wesley Publishing Company,
1975. 572 p.

[NOO75] NOONAN, R. E.
Structured Programming and Formal Specification.
IEEE Transactions on Software Engineering,
Vol. 1, December 1975,
pp. 421-425. 1975.

[NOR82] NORRIE, D. H. and NORRIE, C. I. W.
Large Scale Computation : Architecture and Program Structure
for Special Purpose Machines.
In Proceedings Large Engineering Systems, 1982,
pp. 395-400. 1982.

[NOR84] NORRIE, CHRIS
Supercomputers For Superproblems :
An Architectural Introduction.
Computer, March 1984,
pp. 62-74. 1984.

[OGD72] OGDIN, J. L.
Designing Reliable Software.
Datamation, Vol. 18, 1972,
pp. 71-78, 1972.

[OKO85] OKORIE, AMBROSE SUNNY O.
Using the Decision-tree Model in Database Design.
Computer, March 1985,
pp. 94-102. 1985.

[OLL78] OLLE, T. WILLIAM
The CODASYL Approach to Data Base Management.
New York : John Wiley & Sons,
1978. 287 p.

[ORE48] ORE, OYSTEIN
Chapter 4 Prime Numbers, Factor Table.
In Number Theory and its History.
New York : McGraw-Hill,
1980. 370 p.

[ORG78] ORGANICK, ELLIOTT I. and et al.
Programming Language Structures.
Academic Press : New York,
1978. 659 p.
pp. 71-78.

[OTH81] OTHMER, BOBBIE ANN FREDSALL
Restructuring COBOL Programs to Improve Readability.
Thesis, Ph.D., Rutgers University,
The State University of New Jersey (New Brunswick),
1981. 229 p.

[PAI74] PAIGE, MICHAEL R.
The Use of Software Probes in Testing Fortran Programs.
Computer, July 1974,
pp. 40-47. 1974.

[PAI75] PAIGE, MICHAEL R.
Program Graphs, an Algebra, and their
Implication for Programming.
IEEE Transactions on Software Engineering,
Vol. SE-1, September 1975,
pp. 286-291. 1975.

[PAI77] PAIGE, MICHAEL R.
On Partitioning Program Graphs.
IEEE Transactions on Software Engineering,
Vol. SE-3, No. 6, November 1977,
pp. 386-393. 1977.

[PAR72a] PARNAS, DAVID L.
A Technique for Software Module Specification with examples.
Communications of the ACM, Vol. 15, No. 5, May 1972,
pp. 330-336, 1972.

[PAR72b] PARNAS, DAVID L.
On the Criteria to be used in Decomposing
Systems into Modules.
Communications of the ACM, Vol. 15, No. 12, 1972,
pp. 1053-1058, 1972.

[PAR75] PARNAS, DAVID L.
The Influence of Software Structure on Reliability.
Communications of the ACM, Vol. 10, June 1975,
pp. 358-362, 1975.

[PEA78] PEARSON, LANCE. W.
Generalized Editing : A Design Study of a COBOL Orientation Edit Program Generator.
Thesis, M.Sc., Massey University,
1978. 156 p.

[PEA82] PEAT, LINDSAY R.
Practical Guide to DBMS Selection.
Berlin : Walter de Gruyter,
1982. 340 p.

[PER81] PERRON, ROBERT
Design Guide for Data Base Management Systems.
Wellesley, Massachusetts : Q.E.D. Information Sciences, Inc.,
1981. 154 p.

[PET73] PETERSON, W. W. and et al.
On the Capabilities of While, Repeat and Exit Statements.
Communications of the ACM, Vol. 16, No. 11, November 1973,
pp. 503-512. 1973.

[PET81] PETERS, LAWRENCE J.
Software Design : Methods and Techniques.
New York : Yourdon Press,
1981. 234 p.

[PHI77] PHILIPPAKIS, A.
A Popularity Contest for Languages.
Datamation, December 1977,
pp. 81-87. 1977.

[POD85] PODOLSKY, JOSEPH L.
The Quest for Quality.
Datamation, March 1, 1985,
pp. 119-126. 1985.

[POL81] POLAK, WOLFGANG
Compiler Specification and Verification.
Berlin : Springer-Verlag,
1981. 269 p.

[PRA78] PRATT, T. W.
Control Computations and the Design of
Loop Control Structures.
IEEE Transactions on Software Engineering,
Vol. 4, March 1978,
pp. 81-89. 1978.

[PRI71] PRICE, C. E.
Table Lookup Technique.
ACM Computing Surveys, Vol. 3, No. 2, June 1971,
pp. 49-65. 1971.

[RAL83] RALSTON, ANTHONY and REILLY, EDWIN D.
Encyclopedia of Computer Science and Engineering.
2nd Ed. New York : Van Nostrand Reinhold Company,
1983. 1664 p.

[RAM73] RAMAMOORTHY, C. V.
Design and Construction of an Automated
Software Evaluation System.
In Proceedings 1973 IEEE Symposium
Computer Software Reliability,
pp. 28-37. 1973.

[RAM75] RAMAMOORTHY, C. V. and HO, S. F.
Testing Large Software with Automated
Software Evaluation Systems.
In Proceedings 1975 International Conference
Reliable Software, Los Angeles, April 1975,
pp. 382-394. 1975.

[REI85] REIMANN, BERNARD C. and WAREN, ALLAN D.
User-Oriented Criteria for the Selection of DSS Software.
Communications of the ACM, Vol. 28, No. 2, February 1985,
pp. 166-179. 1985.

[RHO73] RHODES, JOHN and COUGER, J. D.
A Step Beyond Programming.
Systems Analysis Techniques.
New York : John Wiley & Sons,
pp. 14. 1973.

[RIC74] RICHIE, D. M. and THOMPSON, K.
The UNIX Time-Sharing System.
Communication of the ACM, Vol. 17, No. 7, July 1974,
pp. 365-375. 1974.

[RIC80] RICHARDSON, GARY L. and et al.
A Prime on : Structured Program Design.
New York : Petrocelli Book,
1980. 232 p.

[RIC81] RICHARDSON, DEBRA JANE
A Partition Analysis Method to Demonstrate Program Reliability.
Thesis, Ph.D., University of Massachusetts,
1981. 313 p.

[ROB76] ROBINSON, P. B.
Advanced COBOL : ANS 74.
New York : Elsevier Inc.,
1976. 215 p.

[ROD83] RODWELL, PETER
The Personal Computer Handbook.
London : Dorling Kindersley Limited,
1983. 208 p.

[ROG80] ROGGIO, ROBERT F.
A Coded-Based Model for Predicting Path Faults in COBOL Programs.
Thesis, Ph.D., The Graduate Faculty of Auburn University,
1980. 214 p.

[ROS77] ROSS, DOUGLAS T.
Structured Analysis (SA) : A Language for Communication Ideas.
IEEE Transactions on Software Engineering,
Vol. SE-3, No. 1, January 1977,
pp. 16-34. 1977.

[ROS85] ROSS, DOUGLAS T.
Applications and Extensions of SADT.
Computer, April 1985,
pp. 25-34. 1985.

[ROU79] ROUSSOPOULOS, N.
CSDL : A Conceptual Schema Definition Language for the Design of Data Base Applications.
IEEE Transactions on Software Engineering,
Vol. SE-5, No. 5, September 1979,
pp. 481-496. 1979.

[RUL80a] RULLO, THOMAS A.
Advances in Distributed Processing Management : Volume 1.
London : Heyden & Son Inc.,
1980. 199 p.

[RUL80b] RULLO, THOMAS A.
Advances in Computer Programming Management : Volume 1.
London : Heyden & Son Inc.,
1980. 238 p.

[SAL71] SALTON, G.
The SMART Retrieval System :
Experiments in Automatic Document Processing.
Englewood Cliffs, New Jersey : Prentice-Hall,
1971.

[SAL75] SALVADORI, A. and et al.
Static Profile of COBOL Programs.
ACM SIGPLAN Notices, Vol. 10, August 1975,
pp. 20-33. 1975.

[SAM69] SAMMET, J. E.
Programming Languages : History and Fundamentals.
Englewood Cliffs : Prentice-Hall,
1969.

[SAT75] SATTERTHWAITE, EDWIN HALLOWELL
Source Language Debugging Tools.
Thesis, Ph.D., Stanford University,
Department of Computer Science,
1975. 338 p.

[SCH77] SCHNEIDER, N. F. and HOFFMANN, M. H.
An Experiment in Software Error Data Collection and Analysis.
In Proceedings of the Sixth Texas Conference on Computer
Systems, University of Texas, Austin, Texas, November 1977,
pp. 4A1-4A12. 1977.

[SCH79a] SCHNEIDER, HANS-JOCHEN
Formal Models and Practical Tools
for Information Systems Design.
In Proceedings of the IFIP TC-8 Working Conference on Formal
Models and Practical Tools for Information Systems Design.
Oxford, U.K., 297 p. April 17-20, 1979.

[SCH79b] SCHNEIDER, N. F.
Software Metrics for Aiding Program Development and Debugging.
In Proceedings AFIPS 79 National Computer Conference,
pp. 989-994. 1979.

[SCH81] SCHNEIDER, G. MICHAEL and et al.
On the Complexity of Measuring Software Complexity.
In Proceedings AFIPS 81 National Computer Conference,
pp. 317-322. 1981.

[SCH82] SCHLICHTING, RICHARD DALE
Axiomatic Verification to Enhance Software Reliability.
Thesis, Ph.D., Cornell University,
1982. 143 p.

[SCH83] SCHNEIDER, G. R. EUGENIA
Structured Software Maintenance.
In Proceedings AFIPS 83 National Computer Conference,
pp. 137-144. 1983.

[SCH85] SCHEFFER, PAUL A. and STONE, III ALBERT H.
A Case Study of SREM.
Computer, April 1985,
pp. 47-54. 1985.

[SEV74] SEVERANCE, D. G.
Identifier Search Mechanisms : A Survey and Generalized model.
ACM Computing Surveys, Vol. 6, No. 3, September 1974,
pp. 175-194. 1974.

[SHE59] SHERIDAN, P. B.
The Arithmetic Translator - Compiler of the IBM Fortran
Automatic Coding System.
Communications of the Association for Computing Machinery,
Vol. 2, February 1959,
9 p. 1959.

[SHE79] SHEPPARD, S. and et al.
Modern Coding Practices and Programmer Performance.
IEEE Computer, December 1979,
pp. 41-49. 1979.

[SHN76] SHNEIDERMAN, B.
Exploratory Experiments in Programmer Behavior.
International Journal of Man-Machine Studies,
Vol. 5, No. 2, 1976,
pp. 123-143. 1976.

[SHN77] SHNEIDERMAN, B.
Measuring Computer Program Quality and Comprehension.
International Journal of Man-Machine Studies,
Vol. 9, No. 3, 1977,
pp. 465-478. 1977.

[SHN79] SHNEIDERMAN, B. and MAYER, R.
Syntactic-Semantic Interactions in Programmer Behavior :
A Model and Experimental Results.
International Journal of Computer and Information Sciences,
Vol. 8, 1979,
pp. 219-238. 1979.

[SHO77] SHOOMAN, M. and LAEMMEL, A.
Statistical Theory of Computer Programs in
Information Content and Complexity.
In Proceedings of the 1977 Fall COMPCON, IEEE, Long Beach, CA,
pp. 341-347. 1977.

[SHR78] SHROBE, H. E.
Reasoning and Logic for Complex Program Understanding.
Thesis, Ph.D., M. I. T.,
August 1978.

[SIM73a] SIMPSON, NOLA M.
The Design and Implementation of a Structured Programming
Language with few Arbitrary Restrictions -
The Compilation Phase.
Thesis, M.Sc., Massey University,
June 1973. 54 p.

[SIM73b] SIME, M and et al.
Psychological Evaluation of two Conditional Constructions use
in Computer Languages.
International Journal of Man-Machine Studies, Vol. 5, 1973,
pp. 105-113. 1973.

[SLO71] SLOTNIK, D. L.
The Fastest Computer.
Scientific American, Vol. 224, No. 2, February 1971,
pp. 76-87. 1971.

[SNO81] SNOWDON, R. A.
CADES and Software System Development.
New York : North-Holland Publishing Company,
1981. pp. 81-95.

[SPI77] SPIRN, JEFFREY R.
Program Behaviour : Models and Measurements.
New York : North-Holland Inc.,
1977. 277 p.

[SPR77] SPRUGNOLI, R.
Perfect Hashing Functions : A Single Probe Retrieving Method for Static Sets.
Communications of the ACM, Vol. 20, No. 11, November 1977,
pp. 841-850. 1977.

[STA81] STANDISH, THOMAS A.
ARCTURUS : An Advanced Highly-Integrated Programming Environment.
New York : North-Holland Publishing Company,
1981. pp. 49-60.

[STE74] STEVENS, W. P. and et al.
Structured Design.
IBM Systems Journal, Vol. 13, No. 2, 1974,
pp. 115-139. 1974.

[STE79] STERN, NANCY and STERN, ROBERT A.
Structured COBOL Programming.
3rd Ed., John Wiley & Sons,
1979. 571 p.

[STI74] STIGALL, PAUL D. and TASAR, OMUR
Special Tutorial : A Review of Directed Graphs as Applied to Computers.
Computer, Vol. 7, No. 10, October 1974,
pp. 39-47. 1974.

[STR65] STRACHEY, C.
A General Purpose Macrogenerator.
The Computer Journal, Vol. 8, 1965,
pp. 225-241. 1965.

[STU73] STUCKI, LEON G.
Automatic Generation of Self-metric Software.
In Proceedings 1973 IEEE Symposium Computer Software Reliability,
pp. 94-100. 1973.

[STU81] STUCKI, LEON G. and WALKER, HARRY D.
Concepts and Prototypes of ARGUS.
New York : North-Holland Publishing Company,
1981. pp. 61-79.

[SWA60] SWANSON, D. G.
Searching Natural Language Text by Computer.
Science, Vol. 132, 3434, October 1960,
pp. 1099-1104. 1960.

[TEI77] TEICHROEW, D. and HERSHEY, E. A.
PSL/PSA : A Computer-Aided Technique for Structured
Documentation and Analysis of Information Processing Systems.
IEEE Transactions on Software Engineering,
Vol. SE-3, No. 1, January 1977.

[TER83] TERSINE, R. J.
Preventive Maintenance : A path to Higher Productivity.
S.A.M. Advanced Management Journal, Spring 1983,
pp. 38-44. 1983.

[TRA63] TRAINITER, M
Addressing for Random-access Storage with
Multiple Bucket Capacities.
Communications of the ACM, Vol. 10, No. 3, July 1963,
pp. 307-315. 1963.

[TRE82] TRELEAVEN, PHILIP C. and LIMA, ISABEL COUVEIA
JAPAN's Fifth-Generation Computer Systems.
Computer, Vol. 15, No. 8, Aug. 1982,
pp. 79-88. 1982.

[TRE84] TRELEAVEN, PHILIP C. and LIMA, ISABEL COUVEIA
Future Computers : Logic, Data Flow, ..., Control Flow ?
Computer, March 1984,
pp. 47-57. 1984.

[TRI79] TRIANCE, J. M.
Structured Programming in COBOL - The Current Options.
The Computer Journal, Vol. 23, No. 3, 1979,
pp. 194-200. 1979.

[TRI85] TRIANCE, J. M. and LAYZELL, P. J.
Macro Processors for Enhancing High-Level Languages -
Some Design Principles.
The Computer Journal, Vol. 28, No. 1, 1985,
pp. 34-43. 1985.

[UND63] UNDERHILL, L. H.
The Growth of Complexity of a General-purpose Program.
The Computer Journal, Vol. 6, No. 1, January 1963,
pp. 37-38. 1963.

[VAN81] VAN DEN BOSCH, PETER NICO
The Translation of Programming Languages Through
the use of a Graph Transformation Language.
Thesis, Ph.D., The University of British Columbia (Canada),
1981.

[VER85a] VERITY, JOHN W.
Bridging the Software Gap.
Datamation, February 15, 1985,
pp. 84-88. 1985.

[VER85b] VERITY, JOHN W.
1985 DP Budget Survey.
Datamation, March 15, 1985,
pp. 74-78. 1985.

[VET81] VELTER, M., and MADDISON, R. N.
Database Design Methodology.
Englewood Cliffs, New Jersey : Prentice-Hall,
1981. 306 p.

[WAN83] WANG, R. S.
Logic Analysis and its Tools.
In Proceedings AFIPS 83 National Computer Conference,
pp. 97-103. 1983.

[WAT79] WATERS, RICHARD C.
A Method for Analyzing Loop Programs.
IEEE Transactions on Software Engineering,
Vol. SE-5, No. 3, May 1979,
pp. 237-247. 1979.

[WEG68] WEGNER, P.
Programming Language Information Structures and
Machine Organization.
New York : McGraw-Hill,
1968.

[WEG71] WEGNER, P.
Data Structure Models for Programming Languages.
ACM SIGPLAN Notices, Vol. 6, No. 2, 1971,
pp. 1-54. 1971.

[WEI73] WEINGARTEN, F. W.
Translation of Computer Languages.
San Francisco : Holden-Day,
1973.

[WEI74] WEISSMAN, LARRY
Psychological Complexity of Computer Programs :
An Experimental Methodology.
ACM SIGPLAN Notices, 9, June 1974,
pp. 25-36. 1974.

[WEI85] WEIZER, NORMAN and WITHINGTON, FREDERIC
IBM : Mainframes in 1990.
Datamation, January 1, 1985,
pp. 97-104. 1985.

[WEX84] WEXELBLAT, RICHARD L.
Nth Generation Languages.
Datamation, September 1, 1984,
pp. 111-117. 1984.

[WIL76] WILKES, MAURICE V.
Software Engineering and Structured Programming.
IEEE Transactions on Software Engineering,
Vol. SE-2, No. 4, December 1976,
pp. 274-276. 1976.

[WIL79] WILSON, ROBIN J.
Introduction to Graph Theory.
2nd Ed. London : Longman Group Limited,
1979. 163 p.

[WIL81] WILLIS, R. R.
AIDES : Computer Aided Design of Software Systems - II.
New York : North-Holland Publishing Company,
1981. pp. 27-48.

[WIR71] WIRTH, N.
Program Development by Stepwise Refinement.
Communications of the ACM, Vol. 14, April 1971,
pp. 221-227. 1971.

[WIR74] WIRTH, N.
On the Composition of Well-Structured Programs.
ACM Computing Surveys, Vol. 6, No. 4 December, 1974,
pp. 247-259. 1974.

[WIR75] WIRTH, N.
An Assessment of the Programming Language PASCAL.
Communications of the ACM, Vol. 10, June 1975,
pp. 23-30, 1975.

[WIR76] WIRTH, N.
Algorithms + Data Structures = Programs.
Englewood Cliffs, New Jersey : Practice-Hall,
1976.

[WIT85] WITTEN, IAN H. and BRAMWELL, BOB
A System for Interactive Viewing of Structured Documents.
Communications of the ACM, Vol. 28, No. 3, March 1985,
pp. 280-288. 1985.

[WOL83] WOLBERG, JOHN R.
Conversion of Computer Software.
Englewood Cliffs, New Jersey : Prentice-Hall,
1983. 239 p.

[WOO77] WOODWARD, MARTIN R. and et al.
The Analysis of Control Flow Structure in Computer Programs.
In Proceedings Liverpool University Conference Combinatorial
Programming (CP77), BOFFEY, T. B. Editor, September 1977,
pp. 190-201. 1977.

[WOO79] WOODWARD, MARTIN R. and et al.
A Measure of Control Flow Complexity in Program Text.
IEEE Transactions On Software Engineering,
Vol. SE-5, No. 1 January 1979,
pp. 45-50. 1979.

[WOO80] WOODFIELD, SCOTT NORMAN
Enhanced Effort Estimation by Extending Basic Programming
Models to Include Modularity Factors.
Thesis, Ph.D., Purdue University,
1980. 152 p.

[YOD78] YODER, C. M. and SCHRAG, M. L.
Nassi-Shneiderman Charts - An Alternative to
Flowcharts for Design.
Software Engineering Notes of the ACM, Vol. 5, 1978,
pp. 79-86. 1978.

[ZEL78] ZELKOWITZ, M. V.
Perspectives on Software Engineering.
ACM Computing Surveys, Vol. 10, 1978,
pp. 197-216. 1978.

[ZOL77] ZOLNOWSKI, JEAN COCHRANE and SIMMONS, DICK B.
Measuring Program Complexity.
In Proceedings of the 1977 Fall COMPCON, IEEE, Long Beach, CA,
pp. 336-340. 1977.

[ZOL80] ZOLNOWSKI, JEAN COCHRANE and SIMMONS, DICK B.
Measuring Program Complexity in a COBOL Environment.
In Proceedings AFIPS 80 National Computer Conference,
pp. 757-766. 1980.

[ZOL81] ZOLNOWSKI, JEAN COCHRANE and SIMMONS, DICK B.
Taking the Measuring of Program Complexity.
In Proceedings AFIPS 81 National Computer Conference,
pp. 329-336. 1981.

APPENDIX G

MANUALS

APPENDIX G : MANUALS

[APP70] APPLIED DATA RESEARCH
MetaCOBOL Structured Programming Technique
User Guide (P375 M).
Applied Data Research,
1970.

[APP83] APPLE MACINTOSH
Computer Using MacPaint.
Apple Computer, Inc.,
California : USA,
1983. 32 p.

[COB68] COBOL X3.23.
USA Standard COBOL, X3.23 - 1968.
New York : American National Standards Institute,
1968.

[COB74] COBOL X3.23.
USA Standard COBOL, X3.23 - 1974.
New York : American National Standards Institute,
1974.

[COB81] COBOL X3.23
{SEP} Draft Proposed Revised X3.23,
American National Standard Programming Language COBOL.
Technical Committee X3J4 - COBOL,
American National Standards Committee X3,
Information Systems, September 1981.

[COR81] CORRESPONDENCE ON ACCEPTING FOR PUBLIC REVIEW
{SEP14} Proposed American National Standards.
wechnical Committee X3J4 - COBOL,
American National Standards Committee X3,
Information Systems,
September 14, 1981.

[DAT81] DATA PROCESSING STANDARDS SECTION E, VERSION 1.1
Program Design and Coding.
Singapore TELECOMS.
31 October 1981.

[IBM70] IBM
IBM Structured COBOL Pre-processor (SCOBOL),
Program Description and Operations Manual (SC19-5015).
IBM, 1970.

[LAD82] LADD, ANNE P.
COBOL 74 Programmer's Guide IDR5039.
Framingham, Massachusetts : Prime Computer Inc.,
1982.

[MC683] MC 6809
Instruction, Assembler and Linking Loader : User manual.
Department of Computer Science,
Massey University : New 1983.

[MIN82] MINITAB 82.1
Reference Manual.
Massey University Computer Centre,
Massey University : New Zealand,
November 1982.

[NA 80] NA, GRACE T.
The COBOL Reference Guide.
Framingham, Massachusetts : Prime Computer Inc.,
1980.

[NIE75] NIE, NORMAN H. and et al.
SPSS : Statistical Package for the Social Sciences.
2nd Ed., New York : McGraw-Hill Book Company,
1975. 675 p.

[PLI80] The PL/I SUBSET G
Reference Guide IDR4031.
Prime Computer, Inc.,
Massachusetts : USA 01701,
1980.

[SOF70] SOFTWARE SCIENCES
CASTLE Structured COBOL Pre-processor
User Reference Manual.
Software Sciences.
1970.

[SPE80a] SPERRY UNIVAC SERIES 1100
American National Standard,
COBOL (ASCII) Level 5R1,
X 3.23 - 1974, Summary. UP-8583.1.
1980.

[SPE80b] SPERRY UNIVAC
Applied COBOL For OS-1100,
Student Guide. UE-970 Rev.1.
September 1980.

[SPS83] SPSS Inc.
SPSS X, User's Guide.
New York : McGraw-Hill Book Company,
1983. 806 p.

APPENDIX H

INDEX

APPENDIX H : INDEX

G

H

T

U

V

W