

AD-A052 997 TRW DEFENSE AND SPACE SYSTEMS GROUP REDONDO BEACH CALIF F/G 9/2
FUNCTIONAL PROGRAMMING. (1)
FEB 78 J R BROWN, E C NELSON F30602-76-C-0315

TRW DEFENSE AND SPACE SYSTEMS GROUP REDONDO BEACH CALIF

F/G 9/2

FUNCTIONAL PROGRAMMING. (1)

FEB 78 J R BROWN, E C NELSON

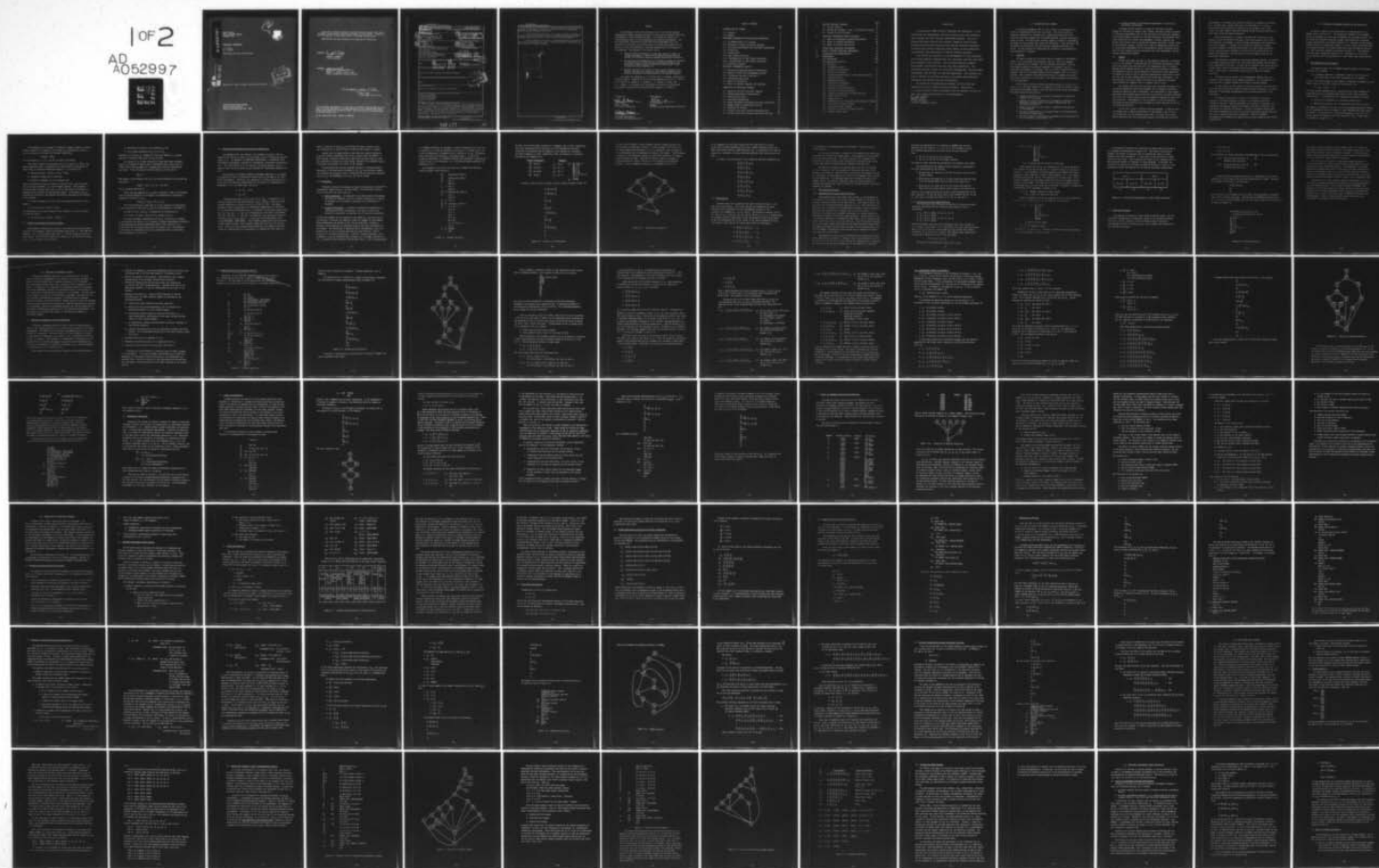
F30602-76-C-0315

UNCLASSIFIED

RADC-TR-78-24

NL

1 OF 2
AD
A052997



AD A 052997

RADC-TR-78-24
Final Technical Report
February 1978

FUNCTIONAL PROGRAMMING

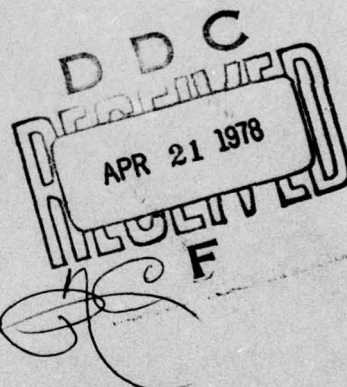
J. R. Brown
E. C. Nelson

TRW Defense and Space Systems Group



AD No. _____
DDC FILE COPY

Approved for public release; distribution unlimited.

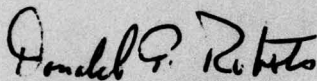


ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

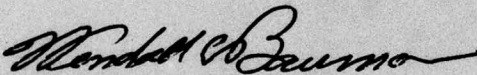
RADC-TR-78-24 has been reviewed and is approved for publication.

APPROVED:



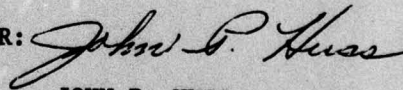
DONALD F. ROBERTS
Project Engineer

APPROVED:



WENDALL C. BAUMAN, Colonel, USAF
Chief, Information Sciences Division

FOR THE COMMANDER:



JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIS) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-78-24	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) FUNCTIONAL PROGRAMMING	5. TYPE OF REPORT & PERIOD COVERED Final Technical Report 9 Jun 77 - 15 Jul 77	6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) J. R. Brown E. C. Nelson	8. CONTRACT OR GRANT NUMBER(s) F30602-76-C-0315	9. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS P.E. 62702F J.O. 55811415
10. PERFORMING ORGANIZATION NAME AND ADDRESS TRW Defense and Space Systems Group One Space Park Redondo Beach CA 90278	11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIS) Griffiss AFB NY 13441	12. REPORT DATE February 1978
13. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same	14. NUMBER OF PAGES 101	15. SECURITY CLASS. (of this report) UNCLASSIFIED
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release, distribution unlimited.		17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) Same
18. SUPPLEMENTARY NOTES RADC Project Engineer: Donald F. Roberts (ISIS)		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Computer Programming Programming Techniques Software Requirements Software Reliability		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This document is the final technical report and computer program documentation for the project entitled Functional Programming, Contract F30602-76-C-0315. It presents the results of a thirteen month study of a new methodology for software development. The new techniques, collectively called Functional Programming (FP), are described in the report as are the results of varied applications of the methods in both the rewriting of existing programs and the development of new programs. The purpose of the report is to;		

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

409 637

set

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

- ≥ a. Describe the FP methodology in sufficient detail to permit an average programmer to understand and employ FP practices and succeed in developing computer programs embodying FP principles.
- b. Provide detailed documentation of the sample programs as necessary to illustrate the contrast between conventional programs and functional programs including functional descriptions, source listings and logic diagrams. and
- c. Present findings of the study of the potential benefits to be derived from using FP, especially with regard to improved testability, reliability, and maintainability of computer programs.

ACCESSION for	
MTIS	White Section <input checked="" type="checkbox"/>
DDC	Buff Section <input type="checkbox"/>
UNANNOUNCED	<input type="checkbox"/>
JUSTIFICATION	
BY	
DISTRIBUTION/AVAILABILITY CODES	
Dist.	A/AIL and/or SPECIAL
A	

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

PREFACE

This document is the final technical report and computer program documentation (CDRL Items A002 and A003) for the project entitled Functional Programming, Contract F30602-76-C-0315. It presents the results of a thirteen month study of a new methodology for software development. The new techniques, collectively called Functional Programming (FP), are described in the report as are the results of varied applications of the methods in both the rewriting of existing programs and the development of new programs. The purposes of the report are to:

- describe the FP methodology in sufficient detail to permit an average programmer to understand and employ FP practices and succeed in developing computer programs embodying FP principles,
- provide detailed documentation of the sample programs as necessary to illustrate the contrast between conventional programs and functional programs including functional descriptions, source listings and logic diagrams, and
- present findings of the study of the potential benefits to be derived from using FP, especially with regard to improved testability, reliability and maintainability of computer programs.

The report was prepared by J. R. Brown and E. C. Nelson, incorporating valuable support, constructive criticism and suggestions from M. W. Alford, E. K. Blum, B. W. Boehm, R. H. Hoffman, J. T. Lawson, M. Lipow, and F. G. Spadaro. The authors also acknowledge the positive encouragement and guidance received from the RADC Project Engineer, D. Roberts.

Prepared by:

John R Brown
J. R. Brown
Project Manager
Functional Programming

Approved by:

F. G. Spadaro
F. G. Spadaro
Manager
Software Systems Engineering Laboratory

Eldred Nelson
E. C. Nelson
Principal Investigator
Functional Programming Project

TABLE OF CONTENTS

	Page
1. INTRODUCTION AND SUMMARY	1
1.1 Overview	1
1.2 Summary	2
2. FUNCTIONAL PROGRAMMING DEFINITION AND DESCRIPTION	4
2.1 The SEMANOL Model of a Program	4
2.2 Extension to Program Structural Elements	5
2.3 Functional Requirements and Functional Capabilities	7
2.4 FP Notation	8
2.5 Phantom Paths	12
2.6 Input Domain Partitions	13
2.6.1 Determination of Input Domain Partitions	13
2.6.2 Specification of Input Domain Partitions	14
2.7 Functional Programs	16
3. FUNCTIONAL PROGRAMMING ANALYSIS	19
3.1 Functional Programming Analysis Methodology	19
3.2 Example of Functional Programming Analysis	21
3.3 A Functional Version of Routine B	28
3.4 Performance Improvement	34
3.5 Effect of Subroutines	35
3.6 Effect of Assembly Language and Interrupts	41
4. COMPOSITION OF FUNCTIONAL PROGRAMS	47
4.1 Method for Writing Functional Programs	47
4.2 Informal Requirements Specification	48
4.3 Formal Specification	49
4.4 Functional Redundancies	52
4.5 Branch Expressions Specifying Partition Constraints	53
4.6 Segment Sequences Specifying Functions	55
4.7 Designing the Program	57
4.8 Writing a Functional Program Containing Loops	61
4.9 Writing a Functional Program Containing a DO Loop	70

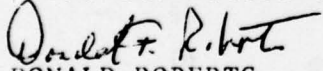
5.	TESTING FUNCTIONAL PROGRAMS	Page 73
5.1	Testing Routine A	74
5.2	Testing the Triangle - Type - Determination Program	77
5.3	Testing the Stack Program	84
6.	FUNCTIONAL PROGRAMMING IMPACT EVALUATION	86
6.1	Impact on Programming Language Requirements	86
6.2	Impact on Software Reliability	88
6.3	Impact on Software Maintenance	90
7.	SYSTEM LEVEL FUNCTIONAL PROGRAMMING	92
8.	TOOLS TO SUPPORT FUNCTIONAL PROGRAMMING	93
9.	CONCLUSIONS AND RECOMMENDATIONS	96
10.	BIBLIOGRAPHY	100
10.1	References	100
10.2	Related Publications	100
	LIST OF FIGURES	
2-1.	FORTRAN Subroutine A	9
2-2.	Routine A in FP Notation	10
2-3.	Structure of Routine A	11
2-4.	Graphical Representation of Input Domain Partitions	16
2-5.	Revised Routine A	17
3-1.	FORTRAN Subroutine B	21
3-2.	Routine B in FP Notation	22
3-3.	Structure of Routine B	23
3-4.	Structure of Revised Routine B	32
3-5.	Structure of Routine Collection	42
4-1.	Graphical Representation of STACK Functions	50
4-2.	INSERT without DO Loop	66
4-3.	INSERT Structure	67
5-1.	FORTRAN Listing of Triangle-type-Determination Program	78
5-2.	Structure of Triangle Program	79
5-3.	Functional Triangle Program	81
5-4.	Structure of Functional Triangle Program	82
5-5.	Triangle Program Paths	83

EVALUATION

An objective of RADC TPO-R5A, Software Cost Reduction, is the exploration and development of novel software tools and procedures which enhance the USAF software development process. Tools are hardware/software devices which enable a human to bring greater leverage upon software production and quality assurance processes. Procedures are uniform guidelines which assist in the unambiguous specification and development of desired software products.

This effort has resulted in the development of a new procedure for writing computer programs that are consistent with the functional specification of the program. The procedure directly supports requirements traceability and test case data generation. The resulting programs contain no functional redundancies (non-executable statements), and are easy to read and comprehend. The concepts can also be applied to existing software to eliminate non-executable statements and improve execution performance.

The Functional Programming methodology was developed and demonstrated using isolated computer programs. Application on a large software system development will be required in order to prove the actual effectiveness of the methodology.


DONALD ROBERTS
Project Engineer
Software Sciences Section

1.0 INTRODUCTION AND SUMMARY

Functional Programming (FP) is both a theory of programming and a method of programming derived from the theory. The theory provides a conceptual framework for thinking about programs and analytical tools usable in analyzing programs and in writing programs, whatever programming method is used. The theory does, however, suggest a method of programming described in this document and called the "FP method". Both the theory and the method are evolving as they are used and problems encountered are solved. FP has potential for improving the reliability, maintainability, and performance of programs.

1.1 Overview

FP theory is based on the formal model of programs in the SEMANOL¹ system. It is concerned with questions, such as: What is a program? What is execution of a program? How are programs structured? What are the relations among the structural elements of a program? What is a functional requirement on a program? What is a formal specification of a program? What is a functional capability of a program? What is a correct program? What inference concerning correctness of a program can be made from correct execution of a test case? The theory provides precise answers to these questions.

The FP method involves viewing (and partitioning) the total set of inputs (i.e., the input domain) to a program as a number of distinct subsets associated with program logical structures and with functional requirements. The partitioning and association:

- provide traceability of functional requirements to structural elements of the program,
- simplify the control structure of a program to eliminate non-executable (phantom) paths, functional redundancies, and unnecessary executions,
- support development of test cases to demonstrate satisfaction of requirements,
- provide a description of the program usable in solving problems encountered in maintenance,
- provide a method of constructing a program from the specification of its functional requirements, and

- provide a method for performance improvement by eliminating unnecessary processing.

A program written using FP is called a "functional program".

Because its structure is simpler, more visible, and more directly traceable to functional requirements than is the structure of conventional programs, a functional program is easier to read and understand. The increased structural visibility also enables the programmer to detect, analyze, and correct many errors as he is writing the program, resulting in fewer errors in the developed program. Because test cases to demonstrate satisfaction of a functional requirement can be constructed by merely selecting inputs from the input domain partition associated with that requirement, testing becomes substantially simpler.

1.2 Summary

Studies have shown that much of the apparent complexity of programs is due to the presence of phantom paths and functional redundancies (logic paths that compute the same function). By eliminating them, FP produces programs having simpler structure - fewer paths and executable statements. The presence of loops, usually regarded as the source of much complexity, is treated in terms of sets of execution sequences, and association of each such set with an input domain partition and functional requirement. The partitioning of the input domain also aids in the analysis of program usage problems and suggests ways to eliminate some of them.

The simpler structure and resulting fewer executable statements in a functional program may reduce the program's use of computing resources - storage and execution time. Additionally, the association of input domain partitions with program logical structures can identify statements executed for inputs from a given partition but not needed to compute the output for that partition. Rearrangement of the statements so that processing is performed for each partition only if needed can reduce processing time for the affected partitions.

FP can also be used to analyze existing programs. It derives from the program text a complete description of the program - what it does, how it executes, and its response to any input - in terms of the input domain partitions and their associated logical structures and functional

requirements. FP analysis can identify problems in a program and indicate how to correct them, thereby improving the reliability of a program. It also provides data enabling the rewriting of the program, with little effort, as a much simpler functional program having no phantom paths and in some cases, fewer executable statements.

The FP description of a program, derived from FP analysis of the program or from the process of writing a functional program, can be used by a maintenance programmer to solve problems occurring in the operational use of the program. It enables him to relate the problem to specific structural elements, functional requirement, and input domain partition. He can then more easily determine the source of the problem and how to change the program in a way that ensures that the change satisfies the functional requirement.

Many functional programs are structured programs; however, FP analysis of existing programs has shown that some structured programs have unnecessarily complicated structures. It shows how to simplify those structures and, in many cases, obtain a structured functional program. It has also shown that, in some cases, a structured program does not have the simplest structure.

FP theory does not depend on the programming language used, but details of the FP method can be language dependent. FP is applicable to both high level language programs and assembly language programs.

In this report we have provided a detailed definition and description of FP and have incorporated numerous examples wherever necessary to clarify important FP principles and demonstrate the intended application of FP techniques.

FP theory is described in section 2, FP analysis in section 3, writing of functional programs in section 4, and testing functional programs in section 5. Studies performed on the impact of functional programming on programming language requirements, software reliability, and maintenance are reported in section 6. Section 7 discusses system level functional programming and section 8 contains preliminary specifications on two tools to support functional programming. Section 9 presents conclusions and recommendations and section 10 lists references and related publications.

2.0 FUNCTIONAL PROGRAMMING DEFINITION AND DESCRIPTION

FP theory is based on the formal model of a program given in the SEMANOL system (section 2.1). The theoretical concepts in the SEMANOL system are extended to the structural elements of a program (section 2.2). The distinction between functional program requirements and functional program capabilities is discussed (section 2.3). Notation for describing program structure is developed (section 2.4); the concept of phantom paths is discussed (section 2.5); input domain partitions are defined (section 2.6); and functional programs are defined (section 2.7). The examples used here and in subsequent sections (i.e., Routines A, B, and C) are actual routines that were developed for a very large scale real-time system.

2.1 The SEMANOL Model of a Program

The semantic theory of the SEMANOL system² provides the following definition of a program:

- A program p specifies a computable function F on the set E of inputs specified by the input expressions in the program.

The set E , the input domain of the program, is composed of members E_i , each member being a set of input values for an execution of p .

$$E = \{E_i: i=1, 2, \dots, N\}$$

The input values composing each E_i include all the values necessary to cause execution of p , including those saved, if any, from a previous execution of p . E identifies all the computations program p can make:

- Each E_i in E corresponds to a possible execution of p .
- Each actual execution of p is initiated by an input E_i from E .

The number N of members in the set E is finite, although perhaps very large, for all programs in which the number of input variables and their ranges are finite. The function F is a rule which assigns to each E_i in E a value, called the "function value" and denoted by $F(E_i)$, chosen from a set called the "range" of F .

The definition of a program is formalized through a semantic operator Φ which, applied to a program p and an input E_i , prescribes the execution of p to produce as its output the function value $F(E_i)$:

$$\Phi(p, E_i) = F(E_i)$$

E is the domain of F - i.e., the set on which F is defined.

The correct output of an execution of p with input E_i is $\hat{F}(E_i)$, the desired function value. An execution of p which does not produce as its output $\hat{F}(E_i)$ is called an "execution failure". It can result in:

- incorrect output: $\Phi(p, E_i) = F(E_i) \neq \hat{F}(E_i)$
- premature termination of execution,
- failure to terminate as in an endless loop.

Since an endless loop does not compute a function value, code containing one is not only incorrect, it is an "illegal" program. Some programs include apparent endless loops used to use up time until an interrupt is initiated by an operator or some external signal. In FP theory, such an interrupt is an input to the program and should be included in the analysis of the program.

A "correct" program is one for which all executions produce correct output:

- For all E_i in E , $\Phi(p, E_i) = \hat{F}(E_i)$

This definition of a correct program can be extended to allow a tolerance in execution output:

- For all E_i in E , $|\Phi(p, E_i) - \hat{F}(E_i)| < \epsilon$

2.2 Extension to Program Structural Elements

When program p executes with input E_i , the execution proceeds through a specific code sequence, called an executable logic path, L_j . The same code sequence - i.e., the executable logic path L_j - may be caused to execute by other inputs. The set of all inputs which cause L_j to be executed is denoted by G_j . G_j is a subset of E such that:

- execution of p with E_i in G_j executes L_j , and
- if E_i causes execution of L_j , it is in G_j .

Execution of p with an E_i a member of E but not a member of G_j causes execution of another logic path, say L_k with $k \neq j$.

All inputs E_i in E cause execution of some logic path; therefore each E_i is a member of some G_j and is associated with a logic path L_j . (There can be code sequences in a program which cannot be executed by any input.) Thus the union of all the subsets G_j is the set E :

$$\bigcup_j G_j = E$$

The subsets G_j are disjoint, for no E_i can cause execution of more than one logic path.

$$G_j \cap G_k = \phi \text{ for } j \neq k \text{ } (\phi: \text{ null set})$$

The G_j therefore partition E .

Since the code sequence L_j is itself a program, it may, in accordance with the formal definition of a program, be interpreted as specifying a computable function F_j on G_j :

$$\phi(L_j, E_i) = F_j(E_i) \text{ for } E_i \text{ in } G_j$$

The "total" function F , specified by p , may therefore be represented by the collection of functions F_j , each F_j being defined on its domain G_j .

In terms of the G_j and F_j , a program may be represented as:

- For all j in $1 \leq j \leq n$ (If E_i is in G_j , Compute $F_j(E_i)$)

In actual programs, determination of the G_j of which E_i is a member is accomplished by evaluation of a sequence of branch expressions. Evaluation of the sequence may include transforming some of the values in E_i to a form such that the branch expressions are simpler; e.g., the variables $X1$ and $X2$ may be transformed into polar coordinates before a particular branch expression is evaluated.

2.3 Functional Requirements and Functional Capabilities

Partitioning of \hat{F} , the function p is intended to compute, and its domain \hat{E} into the set of pairs (\hat{G}_j, \hat{F}_j) defines a set of "functional requirements" on p . (In addition to functional requirements, a program may have performance requirements, reliability requirements, maintenance requirements, etc.) The partitioning of F , the function p actually computes, and its domain E into the set of pairs (G_j, F_j) defines the "functional capabilities" of p .

The objective of reliable software development generally is to produce the program such that its functional capabilities are the same as its functional requirements. Similarly, the objective of software verification is to verify that the functional capabilities are the same as its functional requirements - i.e., to verify that, for all j :

- $G_j = \hat{G}_j$ and
- $F_j = \hat{F}_j$

Problems arise when $G_j \neq \hat{G}_j$ and/or the $F_j \neq \hat{F}_j$. In addition to the obvious problem of an execution failure, $F_j(E_1) \neq \hat{F}_j(E_1)$, there are other problems analyzable in terms of the (G_j, F_j) and (\hat{G}_j, \hat{F}_j) . One common problem is that a program may have functional capabilities not specified in its functional requirements; e.g., in addition to functional capabilities $(G_1, F_1), (G_2, F_2), \dots, (G_n, F_n)$ corresponding to functional requirements $(\hat{G}_1, \hat{F}_1), (\hat{G}_2, \hat{F}_2), \dots, (\hat{G}_n, \hat{F}_n)$, a program may have functional capabilities $(G_{n+1}, F_{n+1}), (G_{n+2}, F_{n+2}), \dots, (G_m, F_m)$ not corresponding to any stated functional requirements. These "hidden capabilities" are the source of many usage problems. This commonly arises when the \hat{G}_j do not collectively cover all of the variable ranges defined by the computer word size. A user not understanding the limitations of a program may then select an

input E_i from one of the G_{n+j} 's and obtain as output a function value not interpretable in terms of what he expected the program to compute.

Another type of usage problem arises when the required functions \hat{F}_j are not clearly associated with the \hat{G}_j on which they are defined. A user wanting to perform a function F_j may select an input from G_k , with $k \neq j$, and when he obtains the output $F_k(E_i)$ instead of the expected $F_j(E_i)$, he ascribes the problem to an error in the program.

FP methodology for writing programs (section 4) involves explicit definition of the functional requirements (\hat{G}_j, \hat{F}_j) and provides a method for designing the program so that it contains only the functional capabilities (G_j, F_j) corresponding to the (\hat{G}_j, \hat{F}_j) and no others.

2.4 FP Notation

Functional analysis of programs is aided by introducing a notation for representing program structure in more detail. The structural elements at the level below the logic path are:

- branch expression - an expression which when evaluated determines the execution sequence, such as a boolean expression in a logical IF statement or an arithmetic expression in an arithmetic IF statement.
- in-line code segment - a sequence of executable expressions not containing any branch expressions, such that, if one of the executable expressions is executed, all of them will be executed.

In-line code segments will be denoted by the symbol S_i , with the value of the subscript i denoting the numerical order of a segment in the program. Branch expressions are denoted by S_j^k , with the subscript j taking on numerical values denoting the numerical order of the branch expression in the program. The superscript k identifies S_j^k as representing a branch expression, and is also a variable denoting the value resulting from an evaluation of the branch expression. If S_j^k is a boolean expression, such as $MN.EQ.0$, it may be evaluated TRUE, denoted by t , or FALSE, denoted by f ; e.g., S_j^t denotes branch expression S_j^k evaluated TRUE. If S_j^k is an expression

in a FORTRAN arithmetic IF statement, it may be evaluated to be less than 0, denoted by -; it may be evaluated to be equal to 0, denoted by 0; or it may be evaluated to be greater than 0, denoted by +; i.e., S_j^+ denotes branch expression S_j^k evaluated to be greater than 0. If S_j^k is an arithmetic expression in a FORTRAN computed GOTO statement, k denotes the integer result of its evaluation; i.e., S_j^3 denotes branch expression S_j^k evaluated to be 3.

As an example of the use of the notation, consider the following FORTRAN program, called Routine A:

```

S1k      IF(GN.NE.0) GOTO 10
S2k      IF(CN.LT.CT) GOTO 5
S1        IE = 1
           GOTO 25
S2 5      IE = 0
           GOTO 25
S3k 10    IF(CN.LT.TR) GOTO 20
S3        IE = 1
           GOTO 25
S4 20      IE = 0
S4k 25    IF(IE.NE.1) GOTO 40
S5        JE = JE + 1
           KI = JD
           KI = 2
           KR = 3
           KB = JA
           KE = JB
           JV = JV + KI + 1
           KG = 1
S6 40      RETURN
           END

```

FIGURE 2-1. FORTRAN Subroutine A

Routine A has 20 executable statements, 6 segments, and 4 branch expressions. The branch expressions S_j^k are defined to be the expressions evaluated within the IF statements and the segments S_i are defined not including any GOTO statements at their termination; thus, for Routine A, the branch expressions and segments are:

<u>Branch Expressions</u>	<u>Segments</u>	
S_1^k : GN.NE.0	S_1 : IE = 1	S_5 : JE = JE + 1 KI = JD
S_2^k : CN.LT.CT	S_2 : IE = 0	KM = 2 KR = 3
S_3^k : CN.LT.TR	S_3 : IE = 1	KB = JA KE = JB
S_4^k : IE.NE.1	S_4 : IE = 0	JV = JV + KI + 1 KG = 1
		S_6 : RETURN

In terms of this notation, Routine A can be written as shown in Fig. 2-2.

IF S_1^k GOTO S_3^k

IF S_2^k GOTO S_2

S_1

GOTO S_4^k

S_2

GOTO S_4^k

IF S_3^k GOTO S_4

S_3

GOTO S_4^k

S_4

IF S_4^k GOTO S_6

S_5

S_6

Figure 2-2. Routine A in FP Notation

Routine A in FP notation is more compact than the original program and its structure is shown more vividly. In this notation, "IF" denotes that the branch expression following it is to be evaluated, with the results of the evaluation used to determine the branch to be taken. "GOTO" denotes a transfer of the execution sequence to the branch expression or segment named following the GOTO. From the FP notation form of a program, a diagram showing the logical structure of the program can be prepared directly. Such a diagram for Routine A is presented in Fig. 2-3.

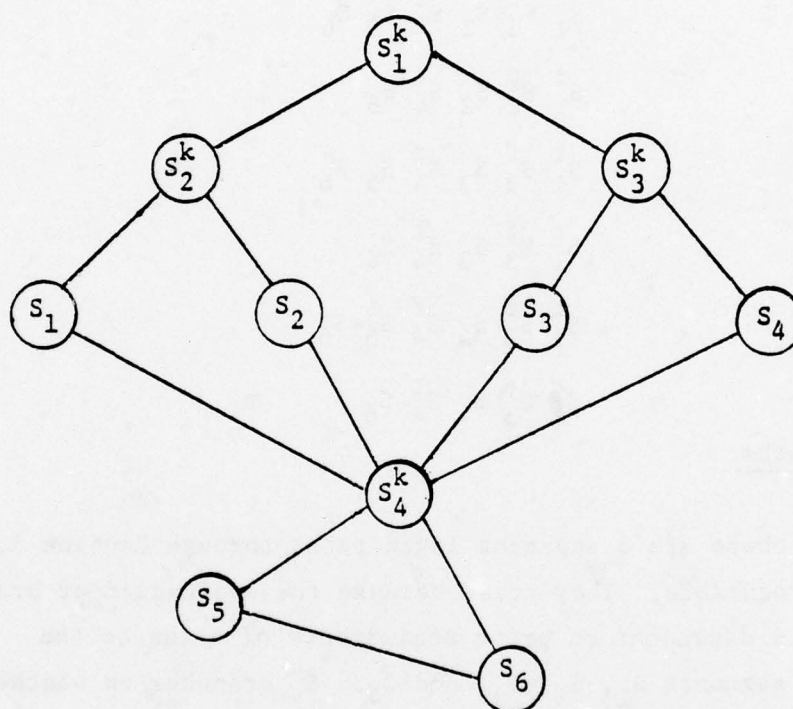


Figure 2-3. Structure of Routine A

In the diagram, the left hand branch from the lower side of a circle surrounding a branch expression S_j^k denotes a FALSE evaluation S_j^f and the right hand branch denotes a TRUE evaluation S_j^t . By convention, the flow of execution is downward, so arrows denoting execution flow direction are not used.

In terms of the FP notation, the 8 potential execution sequences are:

$$S_1^f S_2^f S_1 S_4^f S_5 S_6$$

$$S_1^f S_2^f S_1 S_4^t S_6$$

$$S_1^f S_2^t S_2 S_4^f S_5 S_6$$

$$S_1^f S_2^t S_2 S_4^t S_6$$

$$S_1^t S_3^f S_3 S_4^f S_5 S_6$$

$$S_1^t S_3^f S_3 S_4^t S_6$$

$$S_1^t S_3^t S_4 S_4^f S_5 S_6$$

$$S_1^t S_3^t S_4 S_4^t S_6$$

2.5 Phantom Paths

Although there are 8 apparent logic paths through Routine A, 4 of them are not executable. They arise because the evaluation of branch expression S_4^k is dependent on prior assignments of value to the variable IE in segments S_1 , S_2 , S_3 , and S_4 . S_4^k branches on whether IE has the value 0 or 1. Since S_1 , S_2 , S_3 , and S_4 set IE to 1, 0, 1, 0 respectively, the sequence $S_1^f S_2^f S_1$ must result in S_4^f , the sequence $S_1^f S_2^t S_2$ must result in S_4^t , the sequence $S_1^t S_3^f S_3$ must result in S_4^f , and the sequence $S_1^t S_3^t S_4$ must result in S_4^t . Only the following logic paths are executable:

• $S_1^f S_2^f S_1 S_4^f S_5 S_6$: L_1

• $S_1^f S_2^t S_2 S_4^t S_6$: L_2

• $S_1^t S_3^f S_3 S_4^f S_5 S_6$: L_3

• $S_1^t S_3^t S_4 S_4^t S_6$: L_4

The remaining 4 code sequences are not executable. They are therefore "phantom paths".

In the case of Routine A, one-half of the potential logic paths are executable and one-half are phantom paths. Some programs have more phantom paths than executable paths. The actual structure of a program can be concealed in a web of phantom paths obscuring the executable logic paths so that they cannot be readily seen by reading the program text.

Because of the existence of phantom paths, the text of a program does not continually cue the programmer on the actual structure of the program as he writes it. He has to keep a picture of the structure entirely in his mind. If the program is more than a few statements long, this may be quite difficult. The resulting structural confusion may lead to programmer errors. Such errors, when made, cannot be easily detected by scanning the program text; they are found only through extensive and costly debugging and testing.

2.6 Input Domain Partitions

2.6.1 Determination of Input Domain Partitions

Partitioning of the input domain into subsets, G_j is determined by constraints on the values of the input variables imposed by the branch expression evaluations selecting the logic path L_j . Each input variable ranges over a set of values specified explicitly in p by a data declaration or implicitly by a convention in the programming language. A branch expression partitions the range set of a variable into two or more subsets. As a logic path is executed each successive evaluation of a branch expression constrains the range of one or more input variables. The net effect of all of the branch expression evaluations for logic path L_j is to constrain the values of the input variables to lie within G_j . Some of the input variables may not have their ranges constrained by the branch expression evaluations. Also, some of the input variables may not enter into the calculations in L_j . Thus the number of variables to which values must be assigned may differ from logic path to logic path.

Determination of the G_j may be illustrated by constructing the input domain partitions for Routine A. Routine A has 9 input variables - GN, CN, CT, TR, JE, JD, JA, JB, JV - to which values are assigned prior to entering the routine. It has no data declarations, so the range sets of the input

variables are determined by the convention in FORTRAN that variables beginning with the letters, I, J, K, L, M, or N are integer variables and variables beginning with any other letter are real variables.

Thus:

- GN, CN, CT, and TR are real variables.
- JE, JD, JA, JB, and JV are integer variables.

The ranges of these variables are determined by the computer word length.

The partitioning of the ranges of these variables is determined by the branch expressions S_1^k , S_2^k , and S_3^k :

- S_1^k partitions the range set of GN into the value 0 and all other possible values.
- S_2^k partitions the range set of CN into values less than the value of CT and values greater than or equal to the value of CT.
- S_3^k partitions the range set of CN into values less than the value of TR and values greater than or equal to the value of TR.

S_4^k acts on the internal variable IE and has no partitioning effect on the input variables. S_1^k , S_2^k , and S_3^k act only on the variables GN, CN, CT, and TR. They have no partitioning effect on JE, JD, JA, JB, and JV.

2.6.2 Specification of Input Domain Partitions

The input domain partitions may be specified by listing the branch expression evaluations and any variables whose range sets are not partitioned:

- G_1 : GN = 0, CN \geq CT, JE, JD, JA, JB, JV
- G_2 : GN = 0, CN < CT
- G_3 : GN \neq 0, CN \geq TR, JE, JD, JA, JB, JV
- G_4 : GN \neq 0, CN < TR

Note that although there are 9 input variables, only 8 are included in G_1 , 3 in G_2 , 8 in G_3 , and 3 in G_4 . Thus each E_i in G_1 is a set of 8 values; each E_i in G_2 is a set of 3 values; each E_i in G_3 is a set of 8 values; and each E_i in G_4 is a set of 3 values. The complete input domain may be constructed by forming the union of its partitions:

$$E = G_1 \cup G_2 \cup G_3 \cup G_4$$

The functions corresponding to each of the G_j are:

- $F_1: S_5 S_6$: $JE = JE + 1$
 $KI = JD$
 $KM = 2$
 $KR = 3$
 $KB = JA$
 $KE = JB$
 $JV = JV + KI + 1$
 $KG = 1$
 RETURN

i.e., F_1 updates certain values in a data base.

Each computed value of F_1 is a combination of the values assigned to the variables $JE, KI, KM, KR, KB, KE, JV$, and KG at the end of execution of Routine A. It is, therefore, an 8-tuple having as its elements the values assigned to these variables. Three of these elements - viz., the elements associated with KM, KR , and KG - are restricted to a single value. The remaining 5 elements can individually range over the set of integers fitting within one word length. The range set of F_1 is the set of these 8-tuples. The mapping from G_1 to this output set of 8-tuples is defined by the equations corresponding to the assignment statements in S_5 .

- $F_2: S_6$: RETURN: Computes no output values;
 i.e., F_2 does not update the data base.

Since F_2 computes no value, the range set of F_2 is a set containing no values - viz., the null set \emptyset . F_2 then maps the set G_2 into \emptyset .

- $F_3: S_5 S_6$: $JE = JE + 1$
 $KI = JD$
 $KM = 2$
 $KR = 3$
 $KB = JA$
 $KE = JB$
 $JV = JV + KI + 1$
 $KG = 1$
 RETURN

F_3 updates the data base.

- $F_4: S_6$: RETURN: no update.

Note that $F_1 = F_3$ and $F_2 = F_4$. Routine A contains functional redundancies of the kind treated in the next section.

An alternative technique for specifying the input domain partitions is via a graphical representation (Figure 2-4). This type of illustration supports the analyst in assuring that all significant values of branch variables have been considered. It also helps in the derivation of input values for test cases. This technique has proven to be a valuable tool in the functional design (or redesign) of large programs. If desired, the functions (F_j) may also be included as shown in Sections 4.3 and 4.4, thus providing a concise illustration of the desired functional capabilities.

GN = 0		GN \neq 0	
CN \geq CT	CN < CT	CN \geq TR	CN < TR
G_1	G_2	G_3	G_4

Figure 2-4. Graphical Representation of Input Domain Partitions

2.7 Functional Programs

Can programs be written so they contain no phantom paths? Can the functional redundancies be eliminated? Based on the preceding analysis into the G_j and F_j , Routine A can be rewritten without functional redundancies and with no phantom paths. First combine the partitions of the redundant functions:

- $G'_1 = G_1 \cup G_3$
- $G'_2 = G_2 \cup G_4$

Then construct the branch expressions corresponding to the new partitions:

- $G'_1: (GN.EQ.0).AND.(CN.GE.CT).OR. : S_1^{k'}$
 $(GN.NE.0).AND.(CN.GE.TR)$
- $G'_2: (GN.EQ.0).AND.(CN.LT.CT).OR. : S_2^{k'}$
 $(GN.NE.0).AND.(CN.LT.TR)$

Since G'_1 and G'_2 are disjoint sets and are complements of each other relative to E , only one of the new branch expressions is needed. Routine A may therefore be written, in FP notation:

```

      IF  $S_2^{k'}$  GOTO  $S_6$ 
       $S_5$ 
       $S_6$ 

```

This new form of the program has only 2 paths, both executable, corresponding to the 2 distinct functions. Substituting the FORTRAN expressions for $S_2^{k'}$, S_5 , and S_6 provides FORTRAN code for the revised version of the Routine A as shown in Figure 2-5.

```

      IF((GN.EQ.0).AND.(CN.LT.CT).OR.
        (GN.NE.0).AND.(CN.LT.TR)) GOTO 10
      JE = JE + 1
      KI = JD
      KM = 2
      KR = 3
      KB = JA
      KE = JB
      JV = JV + KI + 1
      KG = 1
10    RETURN
      END

```

Figure 2-5. Revised Routine A

The original version of Routine A had 8 apparent paths, 4 of which were executable, and 20 executable statements. The revised version has 2 logic paths, both executable, and 10 executable statements. It therefore has a simpler structure and less code; however, it has introduced a complication in that it contains a compound Boolean expression.

The final version is called a "functional program", because its structure is explicitly related to the functions F_j performed by the program and their input domain partitions G_j . More generally, a functional program has no phantom paths and the functions F_j and input domain partitions G_j corresponding to each of its logic paths L_j have been explicitly defined; i.e., specifications for them have been written down, so they are explicitly known and not merely intuitive concepts in the programmer's mind.

Routines A and the other routines, Routine B and Routine C, used in this report as examples explaining FP concepts, are actual routines from a real time software project. These routines, written in FORTRAN, were required to be structured using only certain specified control structures, modelling the structured programming control structures of IF THEN ELSE, DO WHILE, etc. in terms of specified groups of FORTRAN IF statements and GOTO statements. Both the original and FP version of Routine A are "structured" in accordance with this definition.

3.0 FUNCTIONAL PROGRAMMING ANALYSIS

Functional programming analysis is the application of the basic ideas of functional programming to the analysis of existing programs. It derives from the program text a complete description of the program - what the program actually does, how it executes, and its response to any input. It therefore is a technique for independent analysis of programs. Since this independent analysis can identify problems in the program, it can contribute to improving the reliability of the program. The description of the program can also aid the maintenance programmer, who generally was not involved in developing the program, in understanding the program, analyzing problems, and determining the changes needed to solve the problems. Because it partitions the input domain into subsets associated with functional requirements, functional programming analysis can aid the generation of test cases to demonstrate satisfaction of the functional requirements. The description also provides data usable in rewriting the program as a functional program.

3.1 Functional Programming Analysis Methodology

Functional programming analysis closely follows the method used in Section 2 for analyzing Routine A to illustrate the concepts of functional programming. The objective of functional programming analysis of a given program is to partition the input domain E of that program into subsets G_j associated with executable logic paths L_j and to determine the function F_j each L_j computes. This analysis is accomplished by identifying in the program text the in-line code segments S_i and the branch expressions S_j^k , analyzing the evaluations of the S_j^k to determine the L_j , and analyzing the S_i composing each L_j to determine the F_j .

This analysis can be performed by carrying out the following steps:

1. Identify the segments S_i and branch expressions S_j^k in the program text by marking them on the left hand margin of the program listing.
2. Rewrite the program in FP notation. This provides a more compact representation of the program, aiding further analysis.
3. Construct a diagram (such as the one in Fig. 2-3) showing the S_i , the S_j^k , and their interconnections. Although this step is not absolutely necessary, it aids in visualizing the structure of the program.
4. Determine the subsets G_j , using the S_j^k evaluations to specify partitioning of the input variable ranges, by carrying out the following substeps:
 - a) identify the input variables and their range sets.
 - b) analyze the first branch expression, S_1^k , to determine its partitioning effect on input variable ranges.
 - c) successively analyze execution sequences involving 2, 3, ... branch expressions, determining at each stage the partitioning effect on input variable ranges.
 - d) on identifying a phantom (non-executable) sequence, eliminate it from further analysis.
 - e) continue the process until all the executable sequences have been defined. The corresponding partitions on the input variable ranges are the G_j .
5. Associate each G_j with a sequence of S_i 's.
6. Determine F_j from the sequence of S_i associated with G_j .
7. Construct E from the union of the G_j and F from the F_j .

Carrying out this procedure is relatively easy for small programs (<100 statements). For larger programs, the analysis can be made more manageable if the program contains subroutines. The subroutines are analyzed first. The set of inputs for each subroutine are then matched with the range of values computed for each input variable in the calling routine.

3.2 Example of Functional Programming Analysis

Application of the functional programming analysis steps, given in 3.1, is explained by using them to analyze a FORTRAN routine, called "Routine B". The first step in the FP analysis has been completed by marking the S_i and S_j^k on the left hand margin of the listing (Figure 3-1).

S_1		KF = ZR
		K = KF + 1
		U1 = -ZT(2)
		U2 = -ZT(3)*ZS(K) + ZT(7)*ZC(K)
		U3 = ZT(3)*ZC(K) + ZT(7)*ZS(K)
S_1^k		IF (U3.LT.ZE(K)) GOTO 100
S_2		S2 = U1**2/(U1**2 + U2**2)
		IT = 1
S_2^k		IF (S2.LT.ZA) GOTO 20
S_3^k		IF (U1.LT.0) GOTO 10
S_3		KF = KF + 1
		GOTO 50
S_4	10	KF = KF + 3
		GOTO 50
S_4^k	20	IF(U2.LT.0) GOTO 30
S_5		IT = 0
		GOTO 50
S_6	30	KF = KF + 2
S_5^k, S_7	50	IF (KF.GT.3) KF = KF - 4
S_6^k		IF (IT.EQ.0) GOTO 200
S_8		TL(1) = 5
		TL(2) = KF
		GOTO 200
S_9	100	TL(1) = 6
		TL(2) = KF
S_7^k		IF (ZD.EQ.1) GOTO 200
S_{10}		ZRF = ZRF + 1
		ZQ(Z3) = ZF
		ZV(Z3) = ZN
		ZB(Z3) = ZI
		ZDQ(Z3) = 3
		Z3 = Z3 + ZQ(Z3) + 1
		ZD = 1
S_{11}	200	ZFC = ZR
		RETURN
		END

Figure 3-1. FORTRAN Subroutine B

Routine B has 35 executable statements, 7 branch expressions, and 11 segments.

The second step is to rewrite the routine in FP notation, producing the following more compact representation shown in Figure 3-2.

```
S1  
IF S1k GOTO S9  
S2  
IF S2k GOTO S4  
IF S3k GOTO S4  
S3  
GOTO S5k  
S4  
GOTO S5k  
IF S4k GOTO S6  
S5  
GOTO S5k  
S6  
IF (S5k) S7  
IF S6k GOTO S11  
S8  
GOTO S11  
S9  
IF S7k GOTO S11  
S10  
S11
```

Figure 3-2. Routine B in FP Notation

A graphical representation of the structure of Routine B (Figure 3-3) shows 18 apparent logic paths.

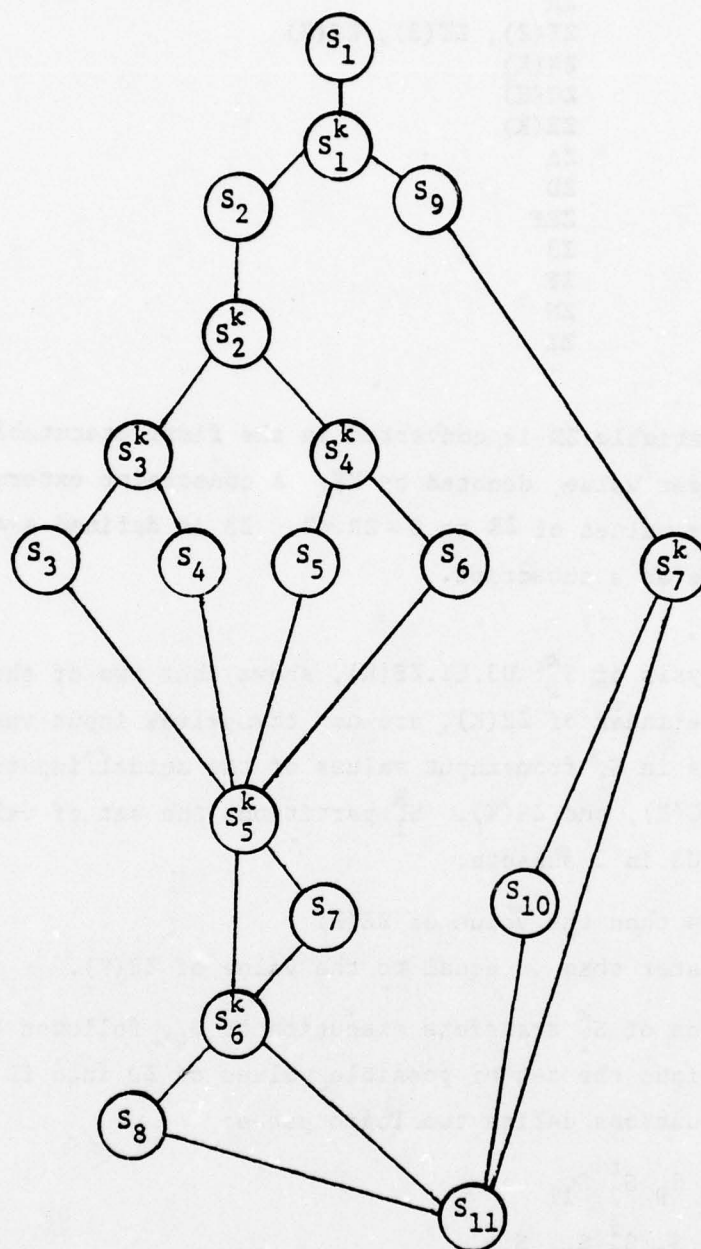


Figure 3-3. Structure of Routine B

Step 4a applied to Routine B finds all the variables to which values must be assigned external to the routine in order for it to execute:

ZR
 ZT(2), ZT(3), ZT(7)
 ZS(K)
 ZC(K)
 ZE(K)
 ZA
 ZD
 ZRF
 Z3
 ZF
 ZN
 ZI

The value of input variable ZR is converted in the first executable statement to an integer value, denoted by KF. A constraint external to the subroutine limits values of ZR to $0 \leq ZR \leq 3$. Z3 is defined externally as an integer for use as a subscript.

Step 4b, analysis of S_1^k : U3.LT.ZE(K), shows that two of the variables in S_1^k , U3 and K, the index of ZE(K), are not themselves input variables but are computed by code in S_1 from input values of the actual input variables ZR, ZT(3), ZT(7), ZC(K), and ZS(K). S_1^k partitions the set of values which can be assigned to U3 in 2 subsets:

- values less than the value of ZE(K)
- values greater than or equal to the value of ZE(K).

TRUE evaluation of S_1^k transfers execution to S_9 , followed by evaluation of S_7^k , which partitions the set of possible values of ZD into $ZD = 1$ and $ZD \neq 1$. These evaluations define two logic paths:

- L_1 : $S_1 S_1^t S_9 S_7^t S_{11}$
- L_2 : $S_1 S_1^t S_9 S_7^f S_{10} S_{11}$

The input domain partitions for these paths are:

- G_1 : ZR, U3 < ZE(K), ZD = 1;
 $U3 = ZT(3)*ZC(K) + ZT(7)*ZS(K)$, KF = ZR, K = KF + 1
- G_2 : ZR, U3 < ZE(K), ZD \neq 1, ZRF, Z3, ZF, ZN, ZI;
 $U3 = ZT(3)*ZC(K) + ZT(7)*ZS(K)$, KF = ZR, K = KF + 1

In the description of the G_j , the partitioning relations and any unpartitioned input variables are listed preceding the semicolon. After the semicolon, transformations on the input variables that must be made before applying the partitioning relations are listed.

FALSE evaluation of S_1^k transfers execution to S_2 . Then evaluation of $S_2^k:S2.LT.ZA$, followed by evaluation of $S_3^k:U1.LT.0$ or $S_4^k:U2.LT.0$, selects whether S_3 , S_4 , S_5 , or S_6 is executed:

- $S_2^f S_3^f$ selects S_3
- $S_2^f S_3^t$ selects S_4
- $S_2^t S_4^f$ selects S_5
- $S_2^t S_4^t$ selects S_6

S_3 , S_4 , and S_6 change the value of KF , while S_5 does not. Evaluation of $S_5^k:KF.GT.3$ tests KF to determine whether or not its value is greater than 3. For initial values of $KF \leq 3$, execution of the sequence $S_5 S_5^k$ can only result in S_5^k being evaluated FALSE. If an initial value of $KF > 3$ (i.e., $ZR > 3$) is input, the external constraint on ZR is violated, causing Routine B to compute something having no physical significance. Thus the sequence $S_5 S_5^t$ will not be executed for all meaningful inputs. Therefore it is a phantom sequence. This analysis suggests that the routine should include a test for $ZR > 3$ with an error message in case it is.

$S_6^k:IT.EQ.0$ tests the value of the internal flag IT , which is set to the value 1 by S_2 and to the value 0 by S_5 . Thus the sequence $S_5 S_5^f S_6^k$ must result in S_6^k evaluated TRUE and all sequences not involving S_5 must result in S_6^k evaluated FALSE. Accordingly, the following are phantom sequences:

- $S_5 S_5^f S_6^f$
- $S_3 S_5^f S_6^t$
- $S_3 S_5^t S_7 S_6^t$
- $S_4 S_5^f S_6^t$

- $S_4 S_5^t S_7 S_6^t$
- $S_6 S_5^f S_6^t$
- $S_6 S_5^t S_7 S_6^t$

These 7 phantom sequences plus the two identified earlier - $S_5 S_5^t S_7 S_6^t$ and $S_5 S_5^t S_7 S_6^f$ - indicate that 9 of the 18 apparent paths of Routine B are phantom paths. The remaining 9 paths are executable.

Definitions for two of the input domain partitions, G_1 and G_2 , and their associated logic paths, L_1 and L_2 , were given previously. The sequences for the 7 other executable paths and their input domain partitions are:

- $L_3: S_1 S_1^f S_2 S_2^t S_4^t S_6 S_5^t S_7 S_6^f S_8 S_{11}$ $G_3: ZR, U3 \geq ZE(K), S2 < ZA, U2 < 0, KF > 1;$
 $KF = ZR, K = KF + 1,$
 $U3 = ZT(3)*ZC(K) + ZT(7)*ZS(K),$
 $S2 = (U1)^2 / ((U1)^2 + (U2)^2),$
 $U1 = -ZT(2)$
 $U2 = -ZT(3)*ZS(K) + ZT(7)*ZC(K)$
- $L_4: S_1 S_1^f S_2 S_2^t S_4^t S_6 S_5^f S_6^f S_8 S_{11}$ $G_4: ZR, U3 > ZE(K), S2 < ZA, U2 < 0, KF < 1;$
 $KF, K, U3, S2, U1$ and $U2$ as
defined in G_3
- $L_5: S_1 S_1^f S_2 S_2^t S_4^f S_5 S_5^f S_6^t S_{11}$ $G_5: ZR, U3 > ZE(K), S2 < ZA, U2 > 0, KF \leq 3;$
 $KF, K, U3, S2, U1,$ and $U2$ as
defined in G_3
- $L_6: S_1 S_1^f S_2 S_2^f S_3^t S_4 S_5^t S_7 S_6^f S_8 S_{11}$ $G_6: ZR, U3 > ZE(K), S2 > ZA, U1 < 0, KF > 0;$
 $KF, K, U3, S2, U1, U2$ as
defined in G_3
- $L_7: S_1 S_1^f S_2 S_2^f S_3^t S_4 S_5^f S_6^f S_8 S_{11}$ $G_7: ZR, U3 > ZE(K), S2 > ZA, U1 < 0, KF = 0;$
 $KF, K, U3, S2, U1,$ and $U2$ as
defined in G_3

- $L_8: S_1 S_1^f S_2 S_2^f S_3 S_3^f S_5 S_7 S_6^f S_8 S_{11}$ $G_8: ZR, U3 \geq ZE(K), S2 \geq ZA, U1 \geq 0, KF=3;$
 $KF, K, U3, S2, U1, \text{ and } U2 \text{ as}$
 $\text{defined in } G_3$
- $L_9: S_1 S_1^f S_2 S_2^f S_3 S_3^f S_5 S_6^f S_8 S_{11}$ $G_9: ZR, U3 \geq ZE(K), S2 \geq ZA, U1 \geq 0, KF < 3;$
 $KF, K, U3, S2, U1, \text{ and } U2 \text{ as}$
 $\text{defined in } G_3.$

The segment sequences for each logic path specify the functions F_j . The range set of each of the F_j 's is a set of n-tuples, the value of n being 1 for F_5 , 4 for $F_1, F_3, F_4, F_6, F_7, F_8$, and F_9 , and 11 for F_2 . The mappings F_j from the G_j to the range sets are specified by the equations associated with the assignment statements in the segment sequences listed below:

- $S_1 S_9 S_{11}$ $F_1: KF=ZR, TL(1)=6, TL(2)=KF, ZFC=ZR$
- $S_1 S_9 S_{10} S_{11}$ $F_2: KF=ZR, TL(1)=6, TL(2)=KF, ZRF=ZRF+1$
 $ZQ(Z3)=ZF, ZV(Z3)=ZN$
 $ZB(Z3)=ZI, ZDQ(Z3)=3$
 $Z3=Z3+ZQ(Z3)+1, ZD=1, ZFC=ZR$
- $S_1 S_6 S_7 S_8 S_{11}$ $F_3: KF=ZR-2, TL(1)=5, TL(2)=KF, ZFC=ZR$
- $S_1 S_6 S_8 S_{11}$ $F_4: KF=ZR+2, TL(1)=5, TL(2)=KF, ZFC=ZR$
- $S_1 S_5 S_{11}$ $F_5: ZFC=ZR$
- $S_1 S_4 S_7 S_8 S_{11}$ $F_6: KF=ZR-1, TL(1)=5, TL(2)=KF, ZFC=ZR$
- $S_1 S_4 S_8 S_{11}$ $F_7: KF=3, TL(1)=5, TL(2)=KF, ZFC=ZR$
- $S_1 S_3 S_7 S_8 S_{11}$ $F_8: KF=ZR-3, TL(1)=5, TL(2)=KF, ZFC=ZR$
- $S_1 S_3 S_8 S_{11}$ $F_9: KF=ZR+1, TL(1)=5, TL(2)=KF, ZFC=ZR$

S_2 is not listed in the segment sequences because it does not affect the computation of the output values. (Its value is involved only in branch expressions.) In the equations for the value of KF in F_3, F_4, F_6, F_7, F_8 , and F_9 the algebra involved in the several executable statements involving KF has been performed. The description of the F_j given above defines the values of all output variables in terms of functions of values of input variables.

3.3 A Functional Version of Routine B

The information developed in the FP analysis of Routine B - viz., the G_j , L_j , and F_j - can be used to rewrite the routine as a functional program. Examination of this information shows that S_6^k : IT.EQ.0 is not needed to partition the input domain. If S_6^k is eliminated, the executable statements involving IT can also be eliminated, for they also do not contribute to the computation of the F_j . IT appears in two segments, S_2 and S_5 . It can be removed from S_2 by defining a new segment S_2^* :

- S_2^* : $S2=U1**2/(U1**2 + U2**2)$

Since S_5 is the statement $IT = 0$, it can be eliminated completely.

To determine the branching sequence for the FP version, it is convenient to list the branch expressions for the input domain partitions G_j :

- G_1 : $U3.LT.ZE(K), ZD.EQ.1$
- G_2 : $U3.LT.ZE(K), ZD.NE.1$
- G_3 : $U3.GE.ZE(K), S2.LT.ZA, U2.LT.0, KF.GT.3$
- G_4 : $U3.GE.ZE(K), S2.LT.ZA, U2.LT.0, KF.LE.3$
- G_5 : $U3.GE.ZE(K), S2.LT.ZA, U2.GE.0$
- G_6 : $U3.GE.ZE(K), S2.GE.ZA, U1.LT.0, KF.GT.3$
- G_7 : $U3.GE.ZE(K), S2.GE.ZA, U1.LT.0, KF.LE.3$
- G_8 : $U3.GE.ZE(K), S2.GE.ZA, U1.GE.0, KF.GT.3$
- G_9 : $U3.GE.ZE(K), S2.GE.ZA, U1.GE.0, KF.LE.3$

To structure Routine B as a functional program, list the execution sequences for the 9 logic paths, removing S_5 and S_6^k and substituting S_2^* for S_2 :

- L_1 : $S_1 S_1^t S_9 S_7^t S_{11}$
- L_2 : $S_1 S_1^t S_9 S_7^f S_{10} S_{11}$
- L_3 : $S_1 S_1^f S_2^* S_2^t S_4^t S_6^t S_5^t S_7 S_8 S_{11}$
- L_4 : $S_1 S_1^f S_2^* S_2^t S_4^t S_6^f S_5^f S_8 S_{11}$
- L_5 : $S_1 S_1^f S_2^* S_2^t S_4^f S_{11}$

- $L_6: S_1 S_1^f S_2^* S_2^f S_3^t S_4 S_5^t S_7 S_8 S_{11}$
- $L_7: S_1 S_1^f S_2^* S_2^f S_3^t S_4 S_5^f S_8 S_{11}$
- $L_8: S_1 S_1^f S_2^* S_2^f S_3^f S_3 S_5^t S_7 S_8 S_{11}$
- $L_9: S_1 S_1^f S_2^* S_2^f S_3^f S_3 S_5^f S_8 S_{11}$

S_5^f is also removed from L_5 since it is not necessary.

Examination of the F_j shows there are no functional redundancies; however, it shows that F_3 , F_4 , F_6 , F_7 , and F_8 differ only in their handling of KF. KF is defined initially in S_1 as $KF = ZR$, for all F_j . The KF computation sequences for these F_j 's are:

- $F_3: S_6 S_7: KF = KF+2, KF = KF-4$
- $F_4: S_6 : KF = KF+2$
- $F_6: S_4 S_7 : KF = KF+3, KF = KF-4$
- $F_7: S_4 : KF = KF+3$
- $F_8: S_3 S_7 : KF = KF+1, KF = KF-4$
- $F_9: S_3 : KF = KF+1.$

Since the KF computation sequences involve incrementing KF by 1, 2, or 3 and since the variable K is defined in S_1 as $KF + 1$, the computation sequences can be simplified by redefining K to be $ZR + 1$ and replacing KF by K. The incrementing sequences then become:

- $F_3: K = K+1, K = K-4$
- $F_4: K = K+1$
- $F_6: K = K+2, K = K-4$
- $F_7: K = K+2$
- $F_8: K = K-4$
- $F_9:$

This can be done by eliminating segment S_3 , since it does not change the value of K and redefining segments S_1 , S_4 , S_6 , S_7 , and S_5^k .

- S_1^* : $K = ZR+1$
 $U1 = -ZT(2)$
 $U2 = -ZT(3)*ZS(K)+ZT(7)*ZC(K)$
 $U3 = ZT(3)*ZC(K)+ZT(7)*ZS(K)$
- S_4^* : $K = K+2$
- S_6^* : $K = K+1$
- S_7^* : $K = K-4$
- S_5^{k*} : $K.GT.3$

Since S_8 and S_9 involve KF, they must be changed:

- S_8^* : $TL(1) = 5$
 $TL(2) = K$
- S_9^* : $TL(1) = 6$
 $TL(2) = K - 1$

Also note that the FALSE evaluation of S_4^k branches directly to S_{11} while the TRUE evaluation is followed by $S_6 S_5^k$ involving further branching. Thus the code can be simplified by replacing S_4^k by its complement:

- S_4^{k*} : $U2.GE.0$

With these substitutions, the execution sequences become:

- L_1 : $S_1^* S_1^t S_9^* S_7^t S_{11}$
- L_2 : $S_1^* S_1^t S_9^* S_7^f S_{10} S_{11}$
- L_3 : $S_1^* S_1^f S_2^* S_2^t S_4^{f*} S_6^* S_5^{t*} S_7^* S_8^* S_{11}$
- L_4 : $S_1^* S_1^f S_2^* S_2^t S_4^{f*} S_6^* S_5^{f*} S_8^* S_{11}$
- L_5 : $S_1^* S_1^f S_2^* S_2^t S_4^{t*} S_{11}$
- L_6 : $S_1^* S_1^f S_2^* S_2^f S_3^t S_4^* S_5^{t*} S_7^* S_8^* S_{11}$
- L_7 : $S_1^* S_1^f S_2^* S_2^f S_3^t S_4^* S_5^{f*} S_8^* S_{11}$
- L_8 : $S_1^* S_1^f S_2^* S_2^f S_3^f S_5^{t*} S_7^* S_8^* S_{11}$
- L_9 : $S_1^* S_1^f S_2^* S_2^f S_3^f S_5^{f*} S_8^* S_{11}$

A program having these logic paths can be written in FP notation:

```
S1*  
IF S1k GOTO S9  
S2*  
IF S2k GOTO S4k*  
IF S3k, S4*  
GOTO S5k*  
IF S4k* GOTO S11  
S6*  
IF S5k*, S7  
S8*  
GOTO S11  
S9*  
IF S7t GOTO S11  
S10  
S11
```

The logic diagram shown in Figure 3-4 for this form of Routine B shows that it has 9 paths.

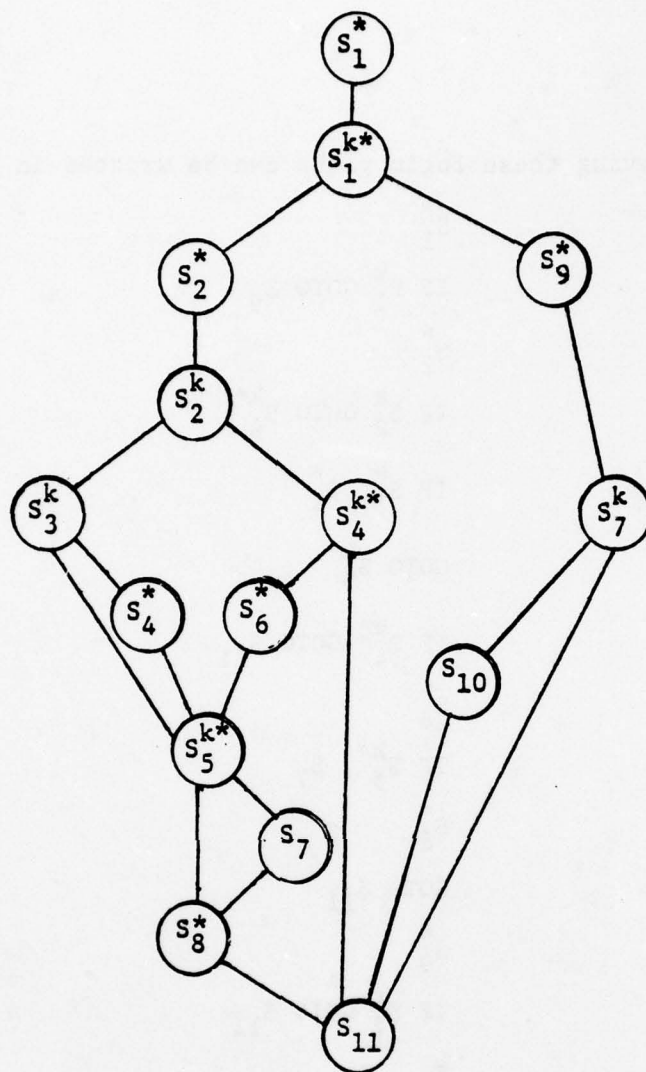


Figure 3-4. Structure of Revised Routine B

This version is not a structured program, for the S_4^{f*} branch of S_4^{k*} creates a cross-link, forbidden by structured programming rules. It can be converted into a structured FORTRAN program (modelling structured programming control structures in terms of FORTRAN IF and GOTO statements), while remaining a functional program, by introducing some logical complications. One way is to combine S_2^k and S_4^{k*} into a compound branch expression $S_2^k \text{ AND } S_4^{k*}$. This involves replacing the structure:

	by:	
IF S_2^k GOTO S_4^{k*}		IF (S_2^k .AND. S_4^{k*}) GOTO S_{11}
IF S_3^k, S_4^*		IF S_2^k GOTO S_6^*
GOTO S_5^{k*}		IF S_3^k, S_4^*
IF S_4^{k*} GOTO S_{11}		GOTO S_5^{k*}
S_6^*		S_6^*

which, while having the same number of lines of code, has the complication of a compound Boolean expression. Another way of writing a functional FORTRAN program conforming to structured programming rules (modelling structured programming control structures in terms of FORTRAN IF and GOTO statements) combines segments S_4^* and S_7 , combines S_6^* and S_7 , and replaces S_5^{k*} by three branch expressions: one in the FALSE branch of S_3^k , one in the TRUE branch of S_3^k , and one in the FALSE branch of S_4^{k*} . This version increases the number of executable statements but involves no compound expressions.

Substituting the definitions of the segments and branch expressions into the first FP version produces the following code for Routine B:

```

      K = ZR+1
      U1 = -ZT(2)
      U2 = -ZT(3)*ZS(K) + ZT(7)*ZC(K)
      U3 = ZT(3)*ZC(K) + ZT(7)*ZS(K)
      IF (U3.LT.ZE(K)) GOTO 100
      S2 = U1**2/(U1**2 + U2**2)
      IF (S2.LT.ZA) GOTO 10
      IF (U1.LT.0) K = K + 2
      GOTO 50
10    IF (U2.GE.0) GOTO 200
      K = K+1
50    IF (K.GT.3) K = K-4
      TL(1) = 5
      TL(2) = K
      GOTO 200
100   TL(1) = 6
      TL(2) = K-1
      IF (ZD.EQ.1) GOTO 200
      ZRF = ZRF+1
      ZQ (Z3) = ZF
      ZV (Z3) = ZN
      ZB (Z3) = ZI
      ZDQ (Z3) = 3

```

```

                Z3 = Z3 + ZQ(Z3) + 1
                ZD = 1
200            ZFC = ZR
                RETURN
                END

```

This version of Routine B has 27 executable statements compared to 35 in the original version.

3.4 Performance Improvement

Because the FP version of Routine B reduced the number of executable statements and did not introduce any complication, it undoubtedly improves the performance - i.e., reduces primary storage requirement and execution time. A further small performance improvement, a reduction in execution time, is possible by investigating whether the execution sequences in any of the logic paths contain any unnecessary executions - i.e., executable statements that do not contribute to branching or computing of output values. It is readily apparent that two statements in S_1 - viz., the statements assigning values to $U1$ and $U2$ - are not needed in paths L_1 and L_2 although they are needed for the remaining 7 paths. These unnecessary executions in L_1 and L_2 can be removed by redefining S_1^* and S_2^* :

```

S1**: K = ZR + 1
        U3 = ZT(3)*ZC(K)+ZT(7)*ZS(K)

S2**: U1 = -ZT(2)
        U2 = -ZT(3)*ZS(K)+ZT(7)*ZC(K)
        S2 = U1**2/(U1**2+U2**2)

```

This change does not change the number of executable statements but it should reduce execution time for L_1 and L_2 .

Test runs were made on Routine A. It was found that the FP version, in spite of its having a compound Boolean expression, compiled on the CDC 7600 computer into approximately 2/3 the amount of primary storage as the original version and it executed in 2/3 the time. This performance improvement is, of course, dependent on the compiler.

3.5 Effect of Subroutines

Although Routine A and Routine B do not contain subroutines, many programs do. FP analysis of programs containing subroutines proceeds similarly to FP analysis of programs having no subroutines, in terms of input domain partitions, functions and logic paths; however, the analysis of the input domain partitions must determine, for each input variable, whether the value is obtained from sources external to the program or as a result of execution of one of the subroutines. For those input values obtained from execution of a subroutine, the range of output values produced by the subroutine must be compared with the allowable input values. The description of the function associated with each input domain partition must include the effects of execution of the subroutines included in its associated logic path.

To illustrate FP analysis of such programs, a program called "Routine C", containing calls to subroutines is used:

Routine C

```
S1      CALL TPR
S1k     IF (ZR) 500, 500, 100
S2 100  CALL TED
S2k 150  IF (Z3) 200, 200, 550
S3 200  ZG = ZG + 1
        ZC = 0
        CALL TCO
S4 300  CALL TRA
        GOTO 2000
S5 500  CONTINUE
        Z3 = 1
        GOTO 150
S6 550  CONTINUE
        CALL TEC
        ZB = ZB + 1
        ZC = ZC + 1
        GOTO 300
```

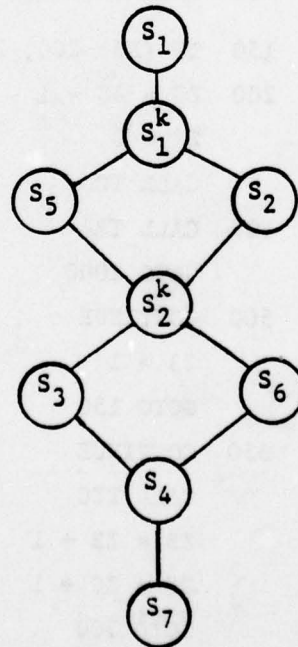
S₇ 2000 RETURN
END

Routine C has 7 segments and two branch expressions. In the assignment of FP symbols to elements of Routine C, the subroutine calls are treated as executable statements.

Following the steps in the FP analysis procedure, the second step in the analysis is to write Routine C in FP notation:

S₁
IF (S₁^k) S₅, S₅, S₂
S₂
IF (S₂^k) S₃, S₃, S₆
S₃
S₄
GOTO S₇
S₅
GOTO S₂^k
S₆
GOTO S₄
S₇

The logic diagram is then:



Despite the somewhat confusing placement of S_3 , S_4 , and S_5 in the program code, the logic diagram shows that Routine C has the structure of a structured program.

The input variables for Routine C are:

- ZR, Z3, ZG, ZB, and ZC.

Branch expression S_1^k partitions the set of possible values of ZR into those specified by $ZR \leq 0$ and those specified by $ZR > 0$. Evaluation of S_1^k to be less than 0 or to be equal to 0 transfers execution to S_5 followed by evaluation of S_2^k , which partitions the set of possible values of Z3. Since S_5 sets $Z3=1$, the only executable sequence is $S_5 S_2^+$. $S_5 S_2^-$ and $S_5 S_2^0$ are phantom sequences. Evaluation of S_1^k to be greater than 0 transfers execution to S_2 , followed by evaluation of S_2^k . Since the value of Z3 is not assigned in any executions in this sequence, all S_2^k evaluation possibilities are possible. The logic paths of Routine C are then:

- $L_1: S_1 S_1^{-0} S_5 S_2^+ S_6 S_4 S_7$
- $L_2: S_1 S_1^+ S_2 S_2^{-0} S_3 S_4 S_7$
- $L_3: S_1 S_1^+ S_2 S_2^+ S_6 S_4 S_7$

Routine C has one phantom path, the path containing the sequence $S_5 S_2^{-0}$. The notation S_2^{-0} is used to denote that the < 0 and $= 0$ evaluations of the arithmetic IF statement transfer to the same segment and therefore do not produce distinct sequences.

The input domain partitions for these paths are:

- $G_1: ZR \leq 0, ZB, ZC.$
- $G_2: ZR > 0, Z3 \leq 0, ZG.$
- $G_3: ZR > 0, Z3 > 0, ZB, ZC.$

The segment sequences for each logic path specify the functions F_j :

- $S_1 S_5 S_6 S_4 S_7$ $F_1: TPR, Z3=1, TEC, ZB=ZB + 1, ZC= ZC + 1, TRA.$
- $S_1 S_2 S_3 S_4 S_7$ $F_2: TPR, TED, ZG=ZG + 1, ZC = 0, TCO, TRA.$
- $S_1 S_2 S_6 S_4 S_7$ $F_3: TPR, TED, TEC, ZB=ZB + 1, ZC=ZC +1, TRA.$

In the description of the functions, F_j , the effect of each subroutine is represented by its name. From these function descriptions, it is apparent that Routine C acts principally to call subroutines TPR, TED, TEC, TCO, and TRA. It always calls TPR and TRA. Calling of TED, TEC, and TCO depends on the G_j to which the input E_i belongs.

For the most part, these subroutines when called obtain their input from a common data base, used by many routines in the large software system from which these routines were chosen. They record their output in that data base. Routine TED, alone of these routines, changes a value of a variable, Z3, used in the explicit calculations of Routine C. TED outputs $Z3 = 0$ or $Z3 = 1$.

Thus for G_2 and G_3 , the values for input variable Z3 are obtained as a consequence of the execution of TED. These values are within the range specifications for Z3; therefore, execution of TED is completely compatible with the execution of Routine C. Input values for ZR, ZB, ZC, and ZG are obtained from the common data base, where they have been placed as the result of execution of routines outside of Routine C.

In general, analysis of subroutine interfaces is more complicated than in this example. It will generally involve:

- determination, for each subroutine, of the inputs, if any, it receives from execution of the calling routine.
- comparison of the allowable values for these inputs with the values computed by the calling routine.
- determination, for each subroutine, of output values, if any, computed by it and used in execution of the calling routine.
- comparison of these output values with the allowable ranges of the variables to which they are assigned in the calling program.

If any incompatibilities of ranges are found, they may indicate a problem and the effect of the incompatibilities must be analyzed further.

From the FP notation representations of the L_j for Routine C, it is relatively easy to rewrite the routine as a functional program. In FP notation, it is:

```

S1
IF (S1k) S5, S5, S2
S2
IF (S2k) S3, S3, S6
S3
GOTO S4
S5
S6
S4
S7

```

and in FORTRAN notation:

```

CALL TPR
IF (ZR) 500, 500, 100
100 CALL TED
IF (Z3) 200, 200, 550
200 ZG = ZG + 1
    ZC = 0
    CALL TCO
    GOTO 600
500 Z3 = 1
550 CALL TEC
    ZB = ZB + 1
    ZC = ZC + 1
600 CALL TRA
    RETURN
    END

```

It contains 14 executable statements compared to 16 in the original version. Although the new version has fewer executable statements and no phantom paths, it is not a structured program. The branch $S_2^+ S_6$ creates the path L_3 by cross-linking L_1 and L_3 . To convert it to a program having the structure of a structured program requires either routing S_5 through S_2^k , as is done in the original program, creating a phantom path, or duplicating the code for S_6 , as in the following:

```

S1
IF (S1k) S5, S5, S2
S2
IF (S2k) S3, S3, S6*
S3
GOTO S4
S5
S6
GOTO S4
S6*
S6
S4
S7

```

S_6^* denotes a block of code identical to the code in S_6 . It is apparent that either method produces a program structurally more complicated than the unstructured functional program.

3.6 Effect of Assembly Language and Interrupts

Although the initial exploration of FP concepts was performed by analyzing higher order language programs, the concepts are also applicable to assembly language programs. For such programs, the inputs include storage addresses, contents of machine registers, and interrupts. To show how FP analysis can be used to analyze assembly language software, a collection of routines from a communications software package on the H 316 computer were analyzed. These routines are concerned with interrupt processing.

The text of these routines, annotated by writing segment labels in the left hand margin is:

Segment	Storage Location	Routine	Instruction
S ₁	12634	HI0E:	INT H2I
	12635		JST HISB2
S ₂	12636	HI1E:	INT H2I
	12637		JST HISB2
S ₃	12640	HI2E:	INT H2I
	12641		JST HISB2
S ₄	12642	HI3E:	INT H2I
	12643		JST HISB2
S ₅	12644	HISB2:	Ø
	12645		STA (HIA)I
	12646		STX(HIX)I
	12647		LDA HISB2
	12650		SUB (HI1E)
	12651		STA Ø
	12652		LDA HI0EX
	12653		STA HISB
	12654		LDA Ø
	12655		ARS 1
	12656		STA Ø
	12657		JMP HISB11
S ₆	12660	HISB:	Ø
	12661		STX (HIX)I
	12662		STA (HIA)I
S ₇	12663	HISB1:	DXA
	12664		JMP (HISB11)I

S ₈	13046	HISB11:	STX HIP
	13047		INK
	13050		STA HIK
	13051		LDA HIM
	13052		SMK INTM
	13053		IMA PRIM
	13054		STA HIMS
	13055		ENB H2I
	13056		JMP HILO

Each of these routines consists of a single segment. The structure of this collection of routines is shown in the diagram in Figure 3-5.

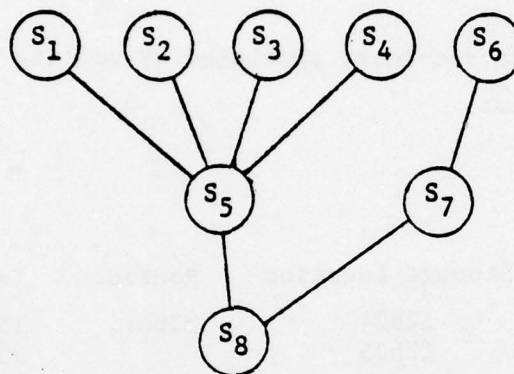


Figure 3-5. Structure of Routine Collection

Note that there are no branch expressions in these routines. The routine collection can be entered from S₁, S₂, S₃, S₄, or S₆, which connect to either S₅ or S₇.

S₁, S₂, S₃, and S₄ are entered respectively when one of four possible interrupts occurs. The H316 computer responds to an interrupt signal by interrupting the processing, setting the computer in the extended address mode, and making a jump to the routine for processing the interrupt, HI0E, if interrupt signal 0 occurs, HI1E, if interrupt signal 1 occurs, HI2E, if interrupt signal 2 occurs, and HI3E, if interrupt signal 3 occurs. The addresses of these routines are given in storage locations to which the respective interrupts trap and are loaded into those locations in an initialization process. The H316 interrupt operates by storing the address L of the next instruction that would have been processed, had the interrupt not occurred, in the first storage location of the interrupt processing routine.

Each of the initial interrupt processing routines, and therefore the segments S_1 , S_2 , S_3 , and S_4 , consists of a single instruction JST HISB2, which causes execution to transfer to subroutine HISB2 at storage location 12644. JST stores in that location the address of the next sequential location following the JST - viz., 12636 for HI0E, 12640 for HI1E, 12642 for HI2E, and 12644 for HI3E. These addresses are used in HISB2 indicating which interrupt signal occurred.

Thus the inputs to the routines, HI0E, HI1E, HI2E, and HI3E, are the hardware signal I, initiating the interrupt that selects the routine, and the address L of the instruction that would have been processed had the interrupt not occurred. The outputs of these routines are the address L and the address B, denoting the specific interrupts. The functions performed by the routines are to:

- save the address L, and
- convert the hardware signal into B

The hardware signal I can only have four possible values and to each of these values there corresponds a unique value of B.

The inputs to the routine HISB2 are B and the contents A of the A-register and X of the Index register. Since B can have only 4 possible values - 12636, 12640, 12642, or 12644 - its range is limited to these values. From the internal structure of HISB2, the range could be any possible value that could be stored in the 16 bit word; however, the possible values of B are limited by the 4 routines which transfer to HISB2. The outputs of HISB2 are A, S, the interrupt number N, and the address L. The functions of HISB2 are:

- save the values of A and X by storing them in HIA and HIX,
- compute the interrupt number N from B and store it in the index register,
- compute address of L and store in HISB.

HISB2 will compute the correct interrupt number (0,1,2, or 3) if presented with the correct values of B; however, if presented with any other value of B, it will compute and store in the index register a value which is not the number of any interrupt. HISB2 therefore has capabilities which are not

needed for the interrupt processing and which if used could lead to improper functioning. It does appear that any such trouble is unlikely, for the software inputs to HISB2 should only produce proper input values and only hardware errors can lead to improper inputs. Nevertheless, the routines HI0E, HI1E, HI2E, HI3E, and HISB2 are examples of "clever" programming, which is hard to interpret, and which produces hidden capabilities that frequently are sources of problems.

Routine HISB provides a software entrance to this routine collection. Its inputs are return addresses stored in the first location of HISB for later use by the routine HIDONE, X, and A. Its outputs are the return addresses, X, and A. Its functions are:

- save the return address in HISB.
- save the contents of the A-register, in HIA, and the contents of the X register, in HIX.

The input to the routine HISB1 is the address state of the computer (extended address). The output is a signal to change the address state to normal address. The function of the routine is to change the address state of the computer from extended address mode to normal address mode.

The inputs to routine HISB11 are the current interrupt number (stored in the index register), keys (obtained from hardware registers, the new interrupt mask (stored in HIM), the old priority mask (stored in PIM)).

The outputs are:

- current interrupt number, stored in HIP.
- the keys stored in HIK.
- new interrupt mask stored in PRIM and output to register INTM.
- the old priority mask stored in HIMS.
- a bit enabling interrupts after exit from the routine.

The functions of the routine are:

- save current interrupt number.
- acquire and store keys.
- set up new interrupt mask.
- save old interrupt mask.
- enable interrupts.

No computations are performed, only the places where various values are stored are changed.

There are 5 logic paths through this collection of routines:

- $L_1 : S_1, S_5, S_8$
- $L_2 : S_2, S_5, S_8$
- $L_3 : S_3, S_5, S_8$
- $L_4 : S_4, S_5, S_8$
- $L_5 : S_6, S_7, S_8$

The inputs to the routines are:

- I, the interrupt signal which selects the entrance routine.
- L, the return address.
- A, the contents of the A register.
- X, the contents of the index register.
- K, the keys, from hardware registers.
- NM, the new priority mask, obtained from HIM.
- OM, the old priority mask, obtained from PRIM.

The subsets G_j are:

- A, X, K, NM, and OM, which are common to all G_j 's.
- The set of addresses, L, of the portions of the IMP software which are interruptable are common to G_1, G_2, G_3 , and G_4 .
- G_1 : The value of I which selects routine HIOE.
- G_2 : The value of I which selects routine HI1E.
- G_3 : The value of I which selects routine HI2E.
- G_4 : The value of I which selects routine HI3E.

The outputs of the routines are:

- N, the current interrupt number, stored in HIP.
- L, the return address for use when the interrupt processing is completed, stored in HISB.
- A, the contents of the A register before the interrupt, stored in HIA.

- X, the contents of the index register before the interrupt, stored in HIX.
- K, the keys, set into a hardware register and stored in HIK.
- OM, stored in HIMS.
- NM, stored in PRIM.
- mode change to extended addressing and enable interrupts.

The functions of the routine collection are:

- Compute current interrupt number.
- Save the contents of A and X registers.
- Save the return address.
- Set keys in hardware.
- Save old priority mask.
- Store new priority mask in PRIM.
- Enable interrupts after completion of the processing.
- Change address mode from extended address to normal address after current interrupt number computation is complete.

Although this collection of routines is structurally and functionally simple, the analysis of the routine collection by FPA demonstrates the applicability of the technique to assembly language software, shows that application of the technique is relatively straightforward, and shows the level of detail to which the analysis can be carried in obtaining a clear understanding of both intended and unintended functional capabilities of software.

4.0 COMPOSITION OF FUNCTIONAL PROGRAMS

FP theory can be used to guide the writing of programs. The basic FP association of input data partitions, logic paths, and functional requirements is used but is applied in the inverse order. That is, functional requirements are defined first, then the input domain partitions, next the logic paths, and finally the program. This approach assures traceability of the functional requirements to the code structures. It also assures that the requirements on the program are defined first rather than plunging into writing code based on a vague concept of what the program is intended to do. The FP method tells how to write the requirements and how to translate them into a formal specification of the program. The result is more explicit, complete, and precise requirements, ensuring that the program written is the one desired.

The FP method is described in section 4.1. In following sections, it is explained by showing how to use it in writing programs. Section 4.8 further discusses the writing of functional programs containing loops.

4.1 Method for Writing Functional Programs

The method for writing functional programs is a step-by-step application of the FP basic association, beginning with the functional requirements. The steps are:

1. Write an informal requirements specification - i.e., a set of statements defining the functions the program is required to perform.
2. Translate these functional requirements into a formal specification defining the G_j and F_j in mathematical and/or logical terms.
3. Identify any functional redundancies - i.e., two or more G_j 's associated with the same F_j .
4. Remove the functional redundancies by forming the union of the G_j 's associated with the same F_j .
5. Write, in the programming language to be used for the program, the branch expressions defining the partition constraints for each G_j . Assign FP symbols S_j^k to the branch expressions.

6. Write the code segment sequences specifying the F_j .
Assign FP symbols S_i to the segments.
7. Design program by:
 - a) determining sequencing for execution of branch expressions,
 - b) defining program logical structure in FP notation.
8. Write program in programming language by substituting code definitions for the S_i and S_j^k .

4.2 Informal Requirements Specification

The FP method will be explained by applying it to designing and writing a program to create and maintain a stack data structure. The first step is to decide what the program, named "STACK", is to do. This is done by writing an informal requirements specification, defining the functions STACK is required to perform.

Since the purpose of STACK is to create and maintain a stack data structure, that data structure must be defined. A stack data structure is an ordered finite list of data elements accessible from its top; i.e., elements of the stack can be added to the top or removed from its top. For the purpose of this example, the data elements will be chosen to be integers in the range $0 \leq I \leq IM$, the variable I denoting an arbitrary integer and IM denoting the maximum integer value allowed in the stack.

The informal requirements specification of STACK is:

- STACK is required to perform the following functions, as selected by the user:
 - Accept an integer input value I and
 - test the value of I for being within the acceptable range, $0 \leq I \leq IM$ and
 - create stack data structure JS , or
 - push value of I onto top of stack, unless stack is maximum size, $J = JM$.

- User options not involving integer input:
 - Pop stack, removing top value, unless stack is empty, $J = 0$.
 - Determine value of top element of STACK, $JS(J)$, unless stack is empty, $J = 0$.
 - Determine number of elements in stack, the value of J .
- Save after each execution,
 - the stack JS , and
 - the number J of elements in the stack.

4.3 Formal Specification

The next step is to translate the informal statement of the requirements into a formal specification defining the G_j and F_j precisely and completely. First the stack data structure JS is defined as an ordered J -tuple, (I_1, I_2, \dots, I_J) . Each I_j in the J -tuple is an integer in the range $0 \leq I_j \leq IM$ and J , the number of elements in the tuple (stack), is an integer in the range $0 \leq J \leq JM$, the value $J = 0$ denoting an empty stack; i.e:

- $JS = (I_1, I_2, \dots, I_J)$
 - $0 \leq I_j \leq IM$,
 - $1 \leq j \leq J$, unless $J = 0$,
 - $0 \leq J \leq JM$,
 - $J = 0$ denotes an empty stack.
- $JS(J) = I_J$, top element in stack.

To completely define the inputs, a design decision on how to represent the user option choices must be made. For simplicity, the user option choice will be represented by an integer value assigned to an input variable K .

The G_j and F_j may now be defined:

G_1 : $K=1, 0 \leq I \leq IM$

F_1 : $J=1, JS(1) = I,$

Print: STACK CREATED

G_2 : $K=1, I < 0$ or $I > IM$

F_2 : Print: RANGE ERROR

G₃: K=2, 0<I<IM, JS,
0<J<JM

G₄: K=2, 0<I<IM, J=JM

G₅: K=2, I<0 or I>IM

G₆: K=3, 0<J<JM

G₇: K=3, J=0

G₈: K=4, 0<J<JM, JS

G₉: K=4, J=0

G₁₀: K=5, 0<J<JM

G₁₁: K<1 or K>5

F₃: J' = J+1, JS(J') = I,
Print: VALUE ADDED

F₄: Print: STACK FULL

F₅: Print: RANGE ERROR

F₆: J' = J - 1,
Print: VALUE REMOVED

F₇: Print: STACK EMPTY

F₈: Print: Value of JS(J)

F₉: Print: STACK EMPTY

F₁₀: Print: Value of J

F₁₁: Print: OPTION ERROR

This set of G_j and F_j definitions may be graphically represented (Figure 4-1) to clarify the relationships of the input branch variables.

K=1		K=2			K=3		K=4		K=5	K<1 or K>5
I Valid 0<I<IM	I Not Valid I<0 or I>IM	I Valid and Stack Not Full	I Valid And Stack Full	I Not Valid	Stack Not Full And Stack Not Empty	Stack Empty	Stack Not Full And Stack Not Empty	Stack Empty	J Valid	K Not Valid
G ₁	G ₂	G ₃	G ₄	G ₅	G ₆	G ₇	G ₈	G ₉	G ₁₀	G ₁₁
F ₁	F ₂	F ₃	F ₄	F ₅	F ₆	F ₇	F ₈	F ₉	F ₁₀	F ₁₁
Create Stack	Print Range Error	Add Value	Print Stack Full	Print Range Error	Remove Value	Print Stack Empty	Print Value JS(J)	Print Stack Empty	Print Value J	Print Option Error

Figure 4-1. Graphical Representation of STACK Functions

Note that the function F_{11} is not defined in the informal specification. Its inclusion in the formal specification came from noting that the user option input variable was defined for the values 1, 2, 3, 4, and 5 but was undefined for other values. Unless a response is defined for these other values of K , the program could be made to execute by inputting other values of K , providing it with functional capabilities not specified in its requirements; i.e., explicit specification of the G_j showed that the G_j were not defined for possible values of the input K . Although a similar question could be raised for values of J outside the range $0 \leq J \leq JM$, provision for handling them is not specified, for the values of J are obtained from a prior execution of STACK and are not subject to arbitrary input by the user.

The formal specifications are in a mathematical notation in order to precisely define what they mean. The G_j 's are specified by listing the variables involved and any constraints on their ranges; e.g., G_3 is specified by specifying the single value 2 for the variable K , limiting the range of the variable I to the integer values in the interval 0 through IM , listing the variable JS denoting a stack of size J , and limiting J to integer values in the interval 0 to JM . The functions F_j are defined in terms of their allowed values and the rule by which one of the allowed values is assigned to each allowed input value; e.g., F_3 is a collection of three output values: J' an integer value denoting the number of items in the stack at the end of the computation of F_3 and constrained to values in the range $0 \leq J' \leq JM$, $JS(J')$ the value of the top item in the stack, and a message to the user stating a value has been added to the stack. The assignment rule is: J' is one plus the input value J , the value assigned to $JS(J')$ is the input value I , and the message "VALUE ADDED" is printed for all computations initiated by an input from G_3 .

Before writing the program, the input domain partition specifications should be examined to ensure that they completely specify all possible inputs. This is done by listing all variables, noting the maximum range of each one allowed by the computer (usually limited by the word length), and noting whether the G_j cover all possible values within those ranges. For STACK, the input variables are K , I , JS , and J . Values of K in the interval 1 through 5 are covered by $G_1 - G_{10}$ and G_{11} covers all values of

K less than 1 or greater than 5 up to the limits of one computer word length (no multiple precision computation is involved in STACK). Values of I in the interval 0 through IM are covered by G_1 , G_3 , and G_4 . Values of I less than 0 and greater than IM are covered by G_2 and G_5 . Both K and I are defined as integer values. Any non-integer values supplied as inputs should be truncated to integer values, if the compiler enforces the type specification. JS, the stack data structure, is specified for all sizes J of the stack from 0 through JM by G_3 and G_8 . Since it is not specified by any G_j for J greater than JM or J less than 0, a potential incomplete situation exists; however, since JS and J are both products of prior executions of STACK and are not supplied externally, this potential incompleteness should cause no problems.

In other studies relating to verifiable software, investigators have developed formal specifications in terms of assertions written in modified predicate calculus or in a record-like format called Parnas specifications. These approaches to formal specifications have tended to be more abstract than the specification of STACK given here and the question of completeness of the specifications generally has been ignored. There is a need for a language for writing formal specifications. Such a language should have a precisely defined syntax and semantics in order that users of it can know exactly what the statements in the language are specifying and it should be computer processable to permit the use of automated tools in working with the specifications.

4.4 Functional Redundancies

Examination of the 11 F_j 's shows that:

- $F_2 = F_5$
- $F_7 = F_9$

hence there are functional redundancies present in the formal specification as it is derived from the informal requirements specification. They can be removed by defining:

$$\begin{aligned} G_2' &= G_2 \cup G_5 : K = 1 \text{ or } 2, I < 0 \text{ or } I > IM \\ G_7' &= G_7 \cup G_9 : K = 3 \text{ or } 4, J = 0 \end{aligned}$$

This reduces the number of input data partitions and their associated functions to 9; hence the program STACK can be constructed so as to have 9 executable logic paths.

4.5 Branch Expressions Specifying Partition Constraints

The next step is to write the branch expressions specifying the partition constraints in the programming language chosen for the program, in this case FORTRAN. The preceding steps are independent of the programming language used.

```

G1: (K.EQ.1).AND.(I.GE.0).AND.(I.LE.IM)
G2: ((K.EQ.1).OR.(K.EQ.2)).AND.((I.LT.0).OR.(I.GT.IM))
G3: (K.EQ.2).AND.(I.GE.0).AND.(I.LE.IM).AND.(J.LT.JM)
G4: (K.EQ.2).AND.(I.GE.0).AND.(I.LE.IM).AND.(J.EQ.JM)
G6: (K.EQ.3).AND.(J.GT.0)
G7: ((K.EQ.3).OR.(K.EQ.4)).AND.(J.EQ.0)
G8: (K.EQ.4).AND.(J.GT.0)
G10: (K.EQ.5)
G11: (K.LT.1).OR.(K.GT.5)

```

Branch expressions constraining J within its range, 0 - JM, are not listed, for in accordance with the assumption stated in the preceding section that J is assigned values only by execution of STACK. Values of J will be adequately constrained by G₄ and G₇, which recognize J having a value at a range boundary, and by F₄ and F₇, which do not allow its value to go outside the range.

Design of the program is aided by representing the branch expressions in FP notation:

$S_1^k : K.EQ.k$

$S_2^k : I.GE.O$

$S_3^k : I.LE.IM$

$S_4^k : J.LT.JM$

$S_5^k : J.GT.O$

In terms of these symbols, the branch expression constraints for the G_j can be written:

$G_1: S_1^1 S_2^t S_3^t$
 $G_2^1: (S_1^1.OR.S_1^2) (S_2^f.OR.S_3^f)$
 $G_3: S_1^2 S_2^t S_3^t S_4^t$
 $G_4: S_1^2 S_2^t S_3^t S_4^f$
 $G_6: S_1^3 S_5^t$
 $G_7^1: (S_1^3.OR.S_1^4) S_5^f$
 $G_8: S_1^4 S_5^t$
 $G_{10}: S_1^5$
 $G_{11}: S_1^{k<1}.OR.S_1^{k>5}$

In this example, all of the branch expressions are tests made directly on input variables. If transformations on these variables had been necessary before applying a test, segments defining those transformations would need to be constructed.

4.6 Segment Sequences Specifying Functions

The next step is to develop algorithms for computing the functions F_j and code these algorithms. In this example, the algorithms are simple and are given in the definitions of the F_j . Therefore this step involves defining the segment sequences specifying each F_j .

The program will be designed for batch execution, with tape input and output, on the Cyber 174 at TRW. The input tape is assigned device number 5; the stack JS and number of elements J are stored on a tape assigned device number 7; the output tape is assigned device number 6.

Since the first action of the program is to read the input values, the first code segment is:

```
.S1: READ(5,INPUT)
```

The remainder of the segments are developed by examining the formal specification of the F_j 's in order and developing the code necessary to compute them.

```
.S2: J=1  
.S3: JS(J)=I  
.S4: REWIND 7  
      WRITE (7)J,JS  
.S5: PRINT(990)  
      990 FORMAT (1H1, 13STACK CREATED)  
.S6: PRINT(991)  
      991 FORMAT (1H1, 11RANGE ERROR)  
.S7: REWIND 7  
      READ (7) J, JS
```

```

.S8: J=J+1
.S9: PRINT (992)
      992 FORMAT(1H1, 1HVALUE ADDED)
.S10: PRINT (993)
      993 FORMAT (1H1, 10HSTACK FULL)
.S11: J=J-1
.S12: PRINT (994)
      994 FORMAT(1H1, 13HVALUE REMOVED)
.S13: PRINT (995)
      995 FORMAT (1H1, 11HSTACK EMPTY)
.S14: PRINT(996)
      996 FORMAT(1H1,5X,6HJS(J)=,I5)
.S15: PRINT(997)
      997 FORMAT (1H1,5X,2HJ=,I5)
.S16: PRINT (998)
      998 FORMAT (1H1,12HOPTION ERROR)
.S17: STOP

```

The F_j are then expressed in terms of sequences of the S_i :

```

.F1: S1S2S3S4S5
.F2: S1S6
.F3: S1S7S8S3S4S9
.F4: S1S7S10
.F6: S1S7S11S4S12
.F7: S1S7S13
.F8: S1S7S14
.F10: S1S7S15
.F11: S1S16

```

4.7 Designing the Program

From the list of the S_j^k sequences for the branch expression constraints of the G_j and the list of the segment sequences specifying the F_j , a functional program can be designed. It is apparent that a functional program can be written by using the compound branch expressions associated with each G_j as the subjects of IF statements with a TRUE evaluation branching to the segment sequence specifying the corresponding F_j . Such a program, however, has a lengthy text and poor performance.

A program having economical text and good performance can be designed by analyzing the branch expression sequences and segment sequences. First note that segment S_1 appears in all segment sequences; therefore the program begins execution with S_1 . Next the branch expression for G_{11} and the segment sequences for F_{11} have no common elements with the other branch expression sequences; so it simplifies matters to make that partitioning next:

$$\text{IF}(S_1^{k < 1} . \text{OR} . S_1^{k > 5}) \text{ GOTO } S_{16}$$

A similar argument suggests that the branching for G_2 should be performed next:

$$\text{IF}(S_1^1 . \text{OR} . S_1^2) \text{ GOTO } (S_2^f . \text{OR} . S_3^f) \\ S_7$$

The term "GOTO ($S_2^f . \text{OR} . S_3^f$)" in the above expressions mean a transfer is made to a statement later in the code sequence containing " $(S_2^f . \text{OR} . S_3^f)$ ". Segment S_7 follows the FALSE evaluation of $(S_1^1 . \text{OR} . S_1^2)$ since it is the next segment in the sequences for F_6 , F_7 , F_8 , and F_{10} . G_{10} and F_{10} have no common elements with G_6 , G_7 , G_8 and F_6 , F_7 , F_8 so they are partitioned next: i.e., IF S_1^5 GOTO S_{15}

The first branch expression for G_7 -viz. $S_1^3 . \text{OR} . S_1^4$ -is accommodated as the FALSE evaluation of S_1^5 , for $K=1$ and 2 were separated by an earlier test on K .

Thus: IF S_5^f GOTO S_{13}

IF S_1^4 GOTO S_{14}

S_{11}

S_4
 S_{12}
 GOTO S_{17}
 S_{13}
 GOTO S_{17}
 S_{14}
 GOTO S_{17}
 S_{15}
 GOTO S_{17}

The processing for F_6 , F_7 , F_8 and F_{10} having been accomplished, the next step is to begin processing for F_2 , F_1 , F_3 , and F_4 :

IF (S_2^f .OR. S_3^f) GOTO S_6
 IF S_1^2 GOTO S_7^*
 S_2
 S_3
 S_4
 S_5
 GOTO S_{17}
 S_6
 GOTO S_{17}
 S_7^*

S_7^* is the same as S_7 , the * denoting that the GOTO is forward to S_7^* not backward to be previous occurrence of S_7 . The final processing for F_3 , F_4 and F_{11} is:

IF S_4^f GOTO S_{10}
 S_8
 S_3
 S_4
 S_9

GOTO S₁₇

S₁₀

GOTO S₁₇

S₁₆

S₁₇

The program design having been created in FP notation, writing the actual code involves only substituting the definitions of the S_j^k and S_i and adding the non-executable statements. The maximum allowed values of J and I - i.e., JM and IM, are input in a data statement and are chosen, arbitrarily, for this example as 7 and 99,999. Accordingly, the program text is:

```
PROGRAM STACK(INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT,TAPE7)
DIMENSION JS (7)
DATA JM,IM/7,99999/
NAMELIST/INPUT/K,I
READ(5,INPUT)
IF((K.LT.1).OR.(K.GT.5))GOTO 80
IF(K.LE.2) GOTO 40
REWIND 7
READ(7) J,JS
IF(K.EQ.5)GOTO 30
IF(J.EQ.0)GOTO 10
IF(K.EQ.4) GOTO 20
J=J-1
REWIND 7
WRITE (7) J,JS
PRINT (990)
990 FORMAT(1H1,13HVALUE REMOVED)
GOTO 90
10 PRINT (991)
991 FORMAT (1H1,11HSTACK EMPTY)
GOTO 90
```

```

20 PRINT (992) JS(J)
992 FORMAT (1H1,5X,6HJS(J)=,I5)
    GOTO 90
30 PRINT (993) J
993 FORMAT (1H1,5X,2HJ=,I5)
    GOTO 90
40 IF ((I.LT.0).OR.(I.GT.IM)) GOTO 50
    IF (K.EQ.2) GOTO 60
    J=1
    JS(J)=I
    REWIND 7
    WRITE (7) J,JS
    PRINT (994)
994 FORMAT (1H1, 13HSTACK CREATED)
    GOTO 90
50 PRINT (995)
995 FORMAT (1H1, 11HRANGE ERROR)
    GOTO 90
60 REWIND 7
    READ (7) J,JS
    IF (J.EQ.JM) GOTO 70
    J=J+1
    JS(J)=I
    REWIND 7
    WRITE (7) J,JS
    PRINT (996)
996 FORMAT (1H1,11HVALUE ADDED)
    GOTO 90
70 PRINT (997)
997 FORMAT (1H1,10HSTACK FULL)
    GOTO 90
80 PRINT (998)
998 FORMAT (1H1, 12HOPTION ERROR)
90 STOP
    END

```

In the substitutions for S_5 , S_6 , S_9 , S_{10} , S_{12} , S_{13} , S_{14} , S_{15} , and S_{16} , the last digit of the 99X label of the FORMAT statements was reassigned to allow an orderly increase in the label numbers.

4.8 Writing a Functional Program Containing Loops

The programs analyzed in Section 3 and the one written using FP techniques in 4.1 - 4.6 contain no loops. How FP concepts are extended to deal with loops is illustrated by developing a sort routine called "INSERT". Two versions of INSERT are developed, one containing ordinary loops and the other containing a DO loop. Whereas the STACK program described in the preceding section was developed as an independently running batch program, INSERT is developed as a subroutine; i.e., it obtains its inputs from the calling program and its outputs are made available to the calling program.

The informal requirements specification for this subroutine is:

- INSERT is required to sort the first N values of a sequence of integer values into descending order.

The formal specification for INSERT begins with definition of the input variables, their types and their ranges:

- IN denotes an arbitrary M-tuple of integer values. Repetitions of integer values are permitted in the M-tuple.
 - IN (J) denotes the Jth element of the M-tuple.
 - M denotes an integer value representing the maximum number of integer values sorted by INSERT.
 - J denotes an integer value in the range $1 \leq J \leq M$.
 - The values assigned to IN (J) can range over the set of integer values fitting within one computer word length.
- N denotes an integer value representing the number of values in IN to be sorted.

The G_j and F_j may then be specified:

- G_1 : $N < 1$ or $N > M$
 - F_1 : Range: $L=0$, denoting an admissible value of N.
- Assignment Rule: For all values of N in G_1 , $L=0$.

• G_2 : $N=1$

F_2 : Range: $L=1$, denoting an admissible value of N .

Assignment Rule: For all values of N in G_2 , $L=1$. No sort required, only 1 value to be sorted.

• G_3 : $1 < N \leq M$, IN

F_3 : Range: $L=1$, IN_s , and N -tuple of integer values $IN_s(J)$ such that $1 < N \leq M$ and for all J in $1 \leq J \leq N-1$, $IN_s(J) \geq IN_s(J+1)$.

Assignment Rule: Each pair (N, IN) in G_3 is assigned to the IN_s from the range of F_3 such that for all K in $1 \leq K \leq N$, there exists a J in $1 \leq J \leq N$, such that $IN_s(J) = IN(K)$.

The specification of F_3 given above defines the integer sort function sufficiently well for a programmer to develop the routine from it; however, F_3 has internal structure analyzable using FP concepts. Since the F_3 range specification applies to adjacent members of the N -tuple - i.e., to $IN_s(J)$ and $IN_s(J+1)$ and since the assignment rule involves only rearranging the positions of the integers and not changing their values, it is of interest to partition the N -tuple elements in accordance with application of the F_3 range specification. Because this is a partitioning of G_3 and F_3 , the notation G_{3j} and F_{3j} is used. The F_{3j} 's in this analysis do not compute the final output of the routine; they compute the result of applying the F_3 range specification to the G_{3j} inputs. In the F_{3j} specifications, $IN'(J)$ denotes the value of the J th element of the N -tuple after the F_{3j} computation has been performed. The G_{3j} and F_{3j} are specified as:

• G_{31} : $J=1$, $IN(1) < IN(2)$

F_{31} : Range: $IN'(1) > IN'(2)$

Assignment Rule: $IN'(1) = IN(2)$
 $IN'(2) = IN(1)$

• G_{32} : $1 < J \leq N-1$,
 $IN(J) < IN(J+1)$

F_{32} : Range: $IN'(J) > IN'(J+1)$

Assignment Rule: $IN'(J) = IN(J+1)$
 $IN'(J+1) = IN(J)$

• G_{33} : $1 \leq J \leq N-1$
 $IN(J) \geq IN(J+1)$

F_{33} : Range: $IN'(J) \geq IN'(J+1)$

Assignment Rule: $IN'(J) = IN(J)$
 $IN'(J+1) = IN(J+1)$

• G_{34} : $J = N$

F_{34} : Range: IN_s

Assignment Rule: $IN_s = IN$

From this analysis of (G_3, F_3) , an algorithm for computing values of F_3 can be constructed by setting $J = 1$ initially and applying the F_3 range specification to $IN(1)$ and $IN(2)$. If the values of $IN(1)$ and $IN(2)$ do not meet the specification, the input is in G_{31} and F_{31} applied to it interchanges the values of $IN(1)$ and $IN(2)$ so that $IN(1)$ and $IN(2)$ now meet the specification. The value assigned to J is then increased by 1. If the values of $IN(1)$ and $IN(2)$ meet the specification, the input is in G_{33} . F_{33} applied to this input leaves the values of $IN(J)$ and $IN(J+1)$ unchanged. The value of J is then increased by 1. In either case the value of J has become 2, so the next input is either in G_{32} or G_{33} . If it is in G_{32} , F_{32} interchanges the values of $IN(2)$ and $IN(3)$. The value of J must then be decreased by 1, because the new value of $IN(2)$ can change its relation to $IN(1)$. If the input is in G_{33} , the values $IN(2)$ and $IN(3)$ are not changed and the value of J is increased by 1. Therefore, depending upon the value of the input in G_3 , the appropriate assignment rule is applied and J is subsequently increased or decreased until $J=N$, placing the input in G_{34} and completing the sort.

Translation of the formal specification into a program design begins by writing the constraints on the input values for each G_j (and G_{3j}) in terms of branch expressions. Here again, the example is illustrated using FORTRAN, but any programming language could be just as easily used.

- $G_1: (N.LT.1).OR.(N.GT.M)$
- $G_2: N.EQ.1$
- $G_3: N.GT.1, IN$
 - $G_{31}: (J.EQ.1).AND.(IN(J).LT.IN(J+1))$
 - $G_{32}: (J.GT.1).AND.(J.LT.N).AND.(IN(J).LT.IN(J+1))$
 - $G_{33}: (J.LT.N).AND.(IN(J).GE.IN(J+1))$
 - $G_{34}: J.EQ.N$

In the branch expressions defining the constraints on G_{33} , the constraint $J.GE.1$ is not given explicitly, since J is set equal to 1 initially and successive applications of the F_{3j} 's will not result in J becoming less than 1.

FP symbols are next assigned to the branch expressions.

- $S_1^k: (N.LT.1).OR.(N.GT.M)$
- $S_2^k: N.EQ.1$
- $S_3^k: J.EQ.1$
- $S_4^k: J.EQ.N$
- $S_5^k: IN(J).LT.IN(J+1)$

In terms of these symbols, the branch expressions for each G_j are:

- $G_1: S_1^t$
- $G_2: S_1^f S_2^t$
- $G_3: S_1^f S_2^f$
 - $G_{31}: S_3^t S_5^t$
 - $G_{32}: S_3^f S_4^f S_5^t$

- $G_{33}: S_4^f S_5^f$
- $G_{34}: S_4^t$

The segments for computing the F_j 's (and F_{3j} 's) are:

- $S_1: J=1$
- $S_2: K=IN(J)$
 $IN(J)=IN(J+1)$
 $IN(J+1)=K$
- $S_3: J=J+1$
- $S_4: J=J-1$
- $S_5: L=1$
- $S_6: L=0$
- $S_7: RETURN$

In terms of these segments, the segment sequences for the F_j 's (and F_{3j} 's) are:

- $F_1: S_6 S_7$
- $F_2: S_5 S_7$
- $F_3: S_1 S_5$
 - $F_{31}: S_2 S_3$
 - $F_{32}: S_2 S_4$
 - $F_{33}: S_3$
 - $F_{34}: S_5$

The program design can now be written in FP notation:

IF S_1^k GOTO S_6

IF S_2^k GOTO S_5

S_1

IF S_5^k GOTO S_2

S_3

```

IF S4k GOTO S5
GOTO S5k

S2
IF S3k GOTO S3

S4
GOTO S5k

S5
GOTO S7

S6

S7

```

The program text is obtained by substituting the definitions of the FP symbols as shown in Figure 4-2.

```

SUBROUTINE INSORT (IN,N,M)
  DIMENSION IN(M)
  IF(N.LT.1.OR. N.GT.M) GOTO 100
  IF(N.EQ.1) GOTO 80
  J=1
30  IF(IN(J).LT.IN(J+1)) GOTO 50
40  J=J+1
    IF(J.EQ.N) GOTO 80
    GOTO 30
50  K=IN(J)
    IN(J)=IN(J+1)
    IN(J+1)=K
    IF(J.EQ.1) GOTO 40
    J=J-1
    GOTO 30
80  L=1
    GOTO 200
100 L=0
200 RETURN
END

```

Figure 4-2. INSORT without DO Loop

Figure 4-3 presents the logical structure of INSORT.

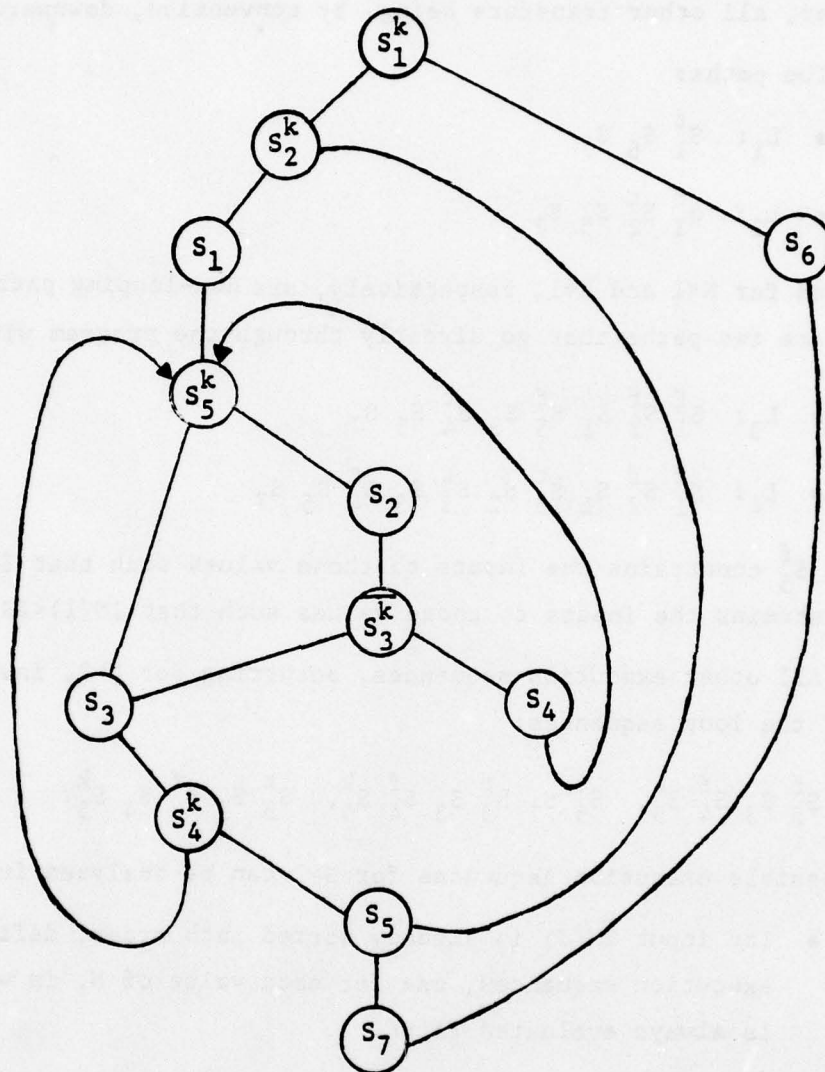


Figure 4-3. INSERT Structure

In the diagram of Figure 4-3., S_3^k has been replaced by its complement $\overline{S_3^k}$: J.NE.1 to avoid crossing of lines. The lines looping back from S_4 and S_4^k to S_5^k have an arrow at their S_5^k end to indicate the direction of the transfer, all other transfers being, by convention, downward.

The paths:

- $L_1: S_1^t S_6 S_7$
- $L_2: S_1^f S_2^t S_5 S_7$

executed for $N < 1$ and $N = 1$, respectively, are non-looping paths. For $N = 2$, there are two paths that go directly through the program without looping:

- $L_3: S_1^f S_2^f S_1 S_5^f S_3 S_4^t S_5 S_7$
- $L_4: S_1^f S_2^f S_1 S_5^t S_2 S_3^t S_3 S_4^t S_5 S_7$

In L_3 , S_5^f constrains the inputs to those values such that $IN(1) > IN(2)$; in L_4 , S_5^t constrains the inputs to those values such that $IN(1) < IN(2)$.

All other execution sequences, occurring for $N > 2$, involve at least one of the loop sequences:

$$S_5^f S_3 S_4^f S_5^k, S_5^t S_2 S_3^t S_3 S_4^f S_5^k, S_5^t S_2 S_3^f S_4 S_5^k.$$

The possible execution sequences for $N > 2$ can be analyzed into 3 cases:

- The input $IN(J)$ is already sorted into order, defining execution sequences, one for each value of N , in which S_5^k is always evaluated FALSE.

- $L_5: S_1^f S_2^f S_1 S_5^f S_3 S_4^f S_5^f S_3 S_4^t S_5 S_7 \quad : N=3$
- $S_1^f S_2^f S_1 S_5^f S_3 S_4^f S_5^f S_3 S_4^f S_5^f S_3 S_4^t S_5 S_7 \quad : N=4$
- \vdots
- $S_1^f S_2^f S_1 S_5^f S_3 S_4^f S_5^f \dots S_4^f S_5^f S_3 S_4^t S_5 S_7 \quad ; N=M$

These sequences involve only the first loop.

- The input value $IN(1)$ is greater than or equal to the value of $IN(J)$ for all J in $1 < J \leq N$, but those values of $IN(J)$ are not in descending order.

$$\begin{aligned}
 \bullet L_6: & S_1^f S_2^f S_1 S_5^f S_3 S_4^f S_5^t S_2 S_3^f S_4 S_5^f S_3 S_4^f S_5^f S_3 S_4^t S_5 S_7 \\
 & S_1^f S_2^f S_1 S_5^f S_3 S_4^f S_5^f S_3 S_4^f S_5^t S_2 S_3^f S_4 S_5^f S_3 S_4^f S_5^f S_3 S_4^t S_5 S_7 \\
 & \vdots
 \end{aligned}$$

In this set of execution sequences, the sequence $S_3^t S_3$ will never occur, for S_3^k can never be evaluated TRUE when $J \neq 1$.

- All other inputs.

$$\begin{aligned}
 \bullet L_7: & S_1^f S_2^f S_1 S_5^f S_3 S_4^f S_5^t S_2 S_3^f S_4 S_5^t S_2 S_3^t S_3 S_4^f S_5^f S_3 S_4^t S_5 S_7 \\
 & \vdots
 \end{aligned}$$

These sequences contain all 3 loop sequences.

Each of these sets (L_5, L_6, L_7) of sequences corresponds to a recursively defined function (F_5, F_6, F_7 , respectively); therefore an association of a set of inputs, a set of sequences, and a function is defined for these three cases; i.e., the triples:

- (G_5, L_5, F_5)
- (G_6, L_6, F_6)
- (G_7, L_7, F_7)

are defined. Because of this association, one may call L_5, L_6 , and L_7 "logic paths", meaning by a logic path L_j a set of executable sequences composed of the same segments and branch expressions and associated with a recursive function F_j defined on a domain G_j .

Note that in handling loops the FP concepts were applied at two levels: first to analyze the structure of F_3 and second to partition G_3 and F_3 into the 5 partitions (G_3, L_3, F_3), (G_4, L_4, F_4), (G_5, L_5, F_5), (G_6, L_6, F_6), and (G_7, L_7, F_7). This shows how a function F_j defined by the requirements on a program may have internal structure.

4.9 Writing a Functional Program Containing a DO Loop

Since the looping part of the INSERT Program is entered after setting $J=1$ and is exited when $J=N$, it may be contained in a DO loop. In FORTRAN, a DO loop of the form:

```
      DO 30 J=1, N
      :
      :
30    CONTINUE
```

executes by setting J initially to the value 1 and executing in sequence the statements following the DO statement until the statement labeled 30 is reached. Then the value of J is increased by 1 and execution continues with the statement immediately following the DO statement. This activity continues until the value of J becomes greater than N , whereupon the loop is exited with execution continuing with the statement immediately following the labeled statement 30.

The logic of the FORTRAN DO loop can be expressed in FP notation by denoting the labeled statement terminating the DO loop by S_j^k and the DO statement by $DO S_j^k$. Here the superscript k denotes the value of the index of the DO loop (in the case of INSERT, the value of J). Evaluation of S_j^k denotes executing the terminal statement, increasing the value of the index by 1, comparing the new value of the index with the value of the upper limit in the DO statement, and transferring to the statement immediately following $DO S_j^k$ unless the new value of the index exceeds the upper limit, in which case execution proceeds with the statement following S_j^k .

Thus INSERT can be rewritten to contain a DO loop having an index J . Execution of the DO loop will cause J to be set initially equal to 1 and be incremented automatically by 1 until $J > N$ at which time execution continues with the statement following the DO loop. Since DO loop semantics do not allow the value of J to be changed within the DO loop, the decrease in the value of J called for in the case (F_{32}) cannot take place inside the DO loop; consequently the sort algorithm must be changed, so that evaluating S_5^k branches outside the DO loop. The interchange of elements specified by S_2 is then executed and the DO loop execution is started over from the beginning, $J=1$. Denoting the terminal statement of the DO loop by S_4^k then leads to the following expression of the sort algorithm in FP notation:

```

DO S4k
IF S5k GOTO S2
S4k
GOTO S5
S2
GOTO DO S4k

```

and the revised sort program in FP notation is:

```

IF S1k GOTO S6
IF S2k GOTO S5
DO S4k
IF S5k GOTO S2
S4k
GOTO S5
S2
GOTO DO S4k
S5
GOTO S7
S6
S7

```

leading to the program text:

```

SUBROUTINE INSERT (IN,N,M)
DIMENSION IN(M)
IF((N.LT.1).OR.(N.GT.M)) GOTO 100
IF (N.EQ.1) GOTO 80
NM1 = N-1
30 DO 40 J=1, NM1
IF(IN(J).LT.IN(J+1)) GOTO 50
40 CONTINUE
GOTO 80
50 K=IN(J)
IN(J) = IN(J+1)
IN(J+1) = K
GOTO 30
80 L=1
GOTO 200
100 L=0
200 RETURN
END

```

This version of INSERT will run slower than the first version because it always returns to the beginning (J=1) of the list of integers after an interchange of two list members has been made.

The first two paths of this version are the same as for the other version because they do not pass through the DO loop.

- $L_1: S_1^t S_6 S_7$
- $L_2: S_1^f S_2^t S_5 S_7$

For $N \geq 2$, the paths involve the DO loop sequences. They can be analyzed as two cases:

- The input IN(J) is already in descending order, defining execution sequences in which S_5^k is always evaluated FALSE.

$$\bullet L_3: S_1^f S_2^f S_4^1 S_5^f S_4^2 S_5 S_7 \quad : N=2$$

$$S_1^f S_2^f S_4^1 S_5^f S_4^2 S_5^f S_4^3 S_5 S_7 \quad : N=3$$

⋮

$$S_1^f S_2^f S_4^1 S_5^f S_4^2 S_5^f \dots S_5^f S_4^M S_5 S_7 : N=M$$

- The input IN(J) is not in descending order, defining the following execution sequences.

$$\bullet L_4: S_1^f S_2^f S_4^1 S_5^t S_2 S_4^1 S_5^f S_4^2 S_5 S_7$$

$$S_1^f S_2^f S_4^1 S_5^t S_2 S_4^1 S_5^f S_4^2 S_5^f S_4^3 S_5 S_7$$

$$S_1^f S_2^f S_4^1 S_5^t S_2 S_4^1 S_5^f S_4^2 S_5^t S_2 S_4^1 S_5^f S_4^2 S_5^f S_4^3 S_5 S_7$$

$$S_1^f S_2^f S_4^1 S_5^f S_4^2 S_5^t S_2 S_4^1 S_5^f S_4^2 S_5^f S_4^3 S_5 S_7$$

⋮

Thus with the DO loop, the execution sequences can be partitioned into 2 sets, each set composed of members containing the same segments and branch expression evaluations.

5.0 TESTING FUNCTIONAL PROGRAMS

The earliest thinking about FP was largely motivated by testing concerns. In recent years, attempts to apply engineering discipline to software testing have brought about increased attention to thoroughly testing a program's structural components. Most recently this trend has extended to the desire, if not the requirement, to identify and test each and every program logic path at least once in the testing process. For small and/or simple programs, this objective can usually be reached with relative ease. For large and/or very complex programs, however, the large number of logic paths and the often complicated conditions associated with executing a specific path together pose overwhelming difficulty with which the tester must cope. In appreciation of the need for both reduced cost and increased rigor of testing, much effort has been devoted to research and development of very sophisticated tools to help the tester identify program logic paths and develop program inputs (i.e., test cases) to force their execution. The use of such tools (e.g., the Automated Test Data Generator^{3,4,5} developed by TRW for NASA/JSC) in support of testing even small (but relatively complex) FORTRAN programs has made it clear that a good deal of the testing effort can be wasted in looking for inputs to force the execution of phantom paths. Moreover, a program containing 8 executable logic paths to do the "functional" work of two generally requires more analysis and computer resources for testing than is actually necessary.

FP directly addresses these problems in several ways. First, as seen in the preceding examples of Routine A and B, FP yields programs of reduced logical complexity, containing only those logic paths needed to supply unique, explicitly required processing capabilities. Second, application of FP establishes and maintains the mapping between subsets (G_j) of the input domain, functions (F_j) required to operate on E_i in G_j , and the logic paths (L_j) which, when coded and executed, perform the intended function on the specified inputs. Clearly, the implication of FP, at least with respect to exercising all program logic paths, is strong and encouraging for those truly concerned with the cost and rigor of testing software.

The following discussion illustrates the primary benefits of FP with respect to testing. These benefits include:

- reduced difficulty in selecting input values for program test cases, and
- reduced difficulty in defining a set of test cases to thoroughly exercise program structure

They are presented in the context of sample FORTRAN programs, however, as with other aspects of FP, the concepts and techniques can be readily generalized to other languages and applications and the above benefits should be even more pronounced for larger and more complex programs.

5.1 Testing Routine A

We tackle Routine A first to demonstrate that, even for very simple programs, the question of what it takes to achieve a thorough test may not be easy to answer. The original version of Routine A listed in Section 2.3 was tested prior to the time that it was rewritten as a functional program. The programmer was required to devise those tests necessary to execute each branch at least once and, without much difficulty found four test cases that satisfied this requirement. They are:

Case 1: GN=0
CN=5
CT=4
TR=6

Case 2: GN=0
CN=3
CT=4
TR=6

Case 3: GN=1
CN=8
CT=4
TR=6

Case 4: GN=1
CN=5
CT=4
TR=6

The values assigned to the integer variables JA, JB, JD, JE, and JV were the same for all 4 test cases and are not listed here.

These test cases reflect the modus operandi of most testers - viz., to construct one test case and then construct the remaining cases by changing the values of the minimum number of variables. Comparing these test cases with the input domain partitions developed in the FP analysis of Routine A shows that each test case belongs to a different input domain partition G_j and the test case set provides a test for each of the partitions in the original version of Routine A.

The actual formulation of the required tests involved tracing through the flow diagram (or the code) and noting the branching actions which were directly controllable by the assignment of input values. Thus, it is not surprising that the description of the test cases (above) is much like that of the input domain partitions. That is, if the testing approach had been to simply select a single test case from each subset of the input domain, then the set of test cases thus derived would have been equivalent to the actual test cases developed by the programmer. Case 1, an input from G_1 , makes no use of the value assigned to TR; case 2, G_2 , makes no use of the values assigned to TR, JA, JB, JD, JE, and JV; case 3, G_3 , makes no use of the value assigned to CT; and case 4, G_4 , makes no use of the values assigned to CT, JA, JB, JD, JE, and JV.

On the other hand, the final version of the functional program (Section 2.7) was developed using a compound conditional branch expression which, upon evaluation, would choose between the two distinct functions to be performed.

This latter version of the program corresponds to the merged input domain subsets $G'_1 = G_1 \cup G_3$ and $G'_2 = G_2 \cup G_4$. Notice that one method of deriving test cases (i.e., simply sampling once from each subset G'_1 and G'_2) yields two test case sets, as follows:

Set 1: $(GN \neq 0, CN \geq TR)$ or $(GN = 0, CN \geq CT)$, JE, JD, JA, JB, JV

Set 2: $(GN \neq 0, CN < TR)$ or $(GN = 0, CN < CT)$

In contrast, if one attempts to devise test cases that will exhaust all of the logical possibilities for the evaluated branch expression,

i.e.,:

((GN.EQ.0).AND.(CN.LT.CT).OR.(GN.NE.0).AND.(CN.LT.TR)) then it is possible to specify eight distinct test case sets, as follows:

Set 1: GN=0, CN \geq CT, CN<TR, JE, JD, JA, JB, JV

Set 2: GN=0, CN \geq CT, CN \geq TR, JE, JD, JA, JB, JV

Set 3: GN \neq 0, CN<CT, CN \geq TR, JE, JD, JA, JB, JV

Set 4: GN \neq 0, CN \geq CT, CN \geq TR, JE, JD, JA, JB, JV

Set 5: GN=0, CN<CT, CN<TR

Set 6: GN=0, CN<CT, CN \geq TR

Set 7: GN \neq 0, CN<CT, CN<TR

Set 8: GN \neq 0, CN \geq CT, CN<TR

A more careful analysis of the compound boolean expression, however, shows that the combination (GN=0, CN \geq CT) causes the left half of the expression to be "FALSE" and the right half of the expression to be "FALSE" (because GN \neq 0 is "FALSE") independent of the relationship between CN and TR. Thus, Set 1 and Set 2 are logically equivalent and can be combined and represented as:

Set 1' : GN=0, CN \geq CT, JE, JD, JA, JB, JV

Similar arguments hold for Set 3 and 4, 5 and 6, and 7 and 8:

Set 3' : GN \neq 0, CN \geq TR, JE, JD, JA, JB, JV

Set 5' : GN=0, CN<CT

Set 7" : GN \neq 0, CN<TR

Finally, looking again at G'₁ and G'₂ and at the two test cases sampled, the need for a Set 1a and 1b and a Set 2a and 2b is indicated by the presence of the "or" in both domain definitions and test case specifications. Notice that the three separate attempts to derive Routine A test cases eventually produced sets of four cases, such that:

Case 1 is a member of Set 1' \equiv Set 1b

Case 2 is a member of Set 5' \equiv Set 2b

Case 3 is a member of Set 3' \equiv Set 1a

Case 4 is a member of Set 7' \equiv Set 2a

5.2 Testing the Triangle - Type - Determination Program

In a prior investigation⁶ of an approach and a tool (the Product Assurance Confidence Evaluator, PACE) used to assure thorough structural testing of programs, a small program written to "determine whether three integers representing three lengths constitute an equilateral, isosceles, or scalene triangle or cannot be the sides of any triangle" was used as an example. The problem was said by Fred Gruenberger⁷ to be a dandy one for teaching logic and flowcharting as well as program testing. In particular, he reported that "even the good students are astonished to find that it takes more than six cases to make a thorough test. . . .".

As part of an investigation of alternative measures of testing thoroughness, a programmer was given the problem statement and asked to write a Triangle-Type-Determination program. Figure 5-1 presents a listing of the FORTRAN source code developed by the programmer. The segments of the program are identified in the left margin, and the program logic is illustrated in Figure 5-2, showing the logical branching potential between the segments. From Figure 5-2 one can quickly identify 88 "apparent" paths through the program, but it is not so easy to tell which and how many paths are actually executable. As it turns out, only 11 of the paths can be executed, and the remaining 77 are phantom paths. Reference 3 further demonstrated the need for 5 separate test cases to cause all executable statements to be exercised and 11 cases to execute every transfer.

		NAMELIST/INPUT/I,J,K
S ₁		READ (5, INPUT)
		MATCH = 0
S ₁ ^k , S ₂		IF (I.EQ.J) MATCH = MATCH + 1
S ₂ ^k , S ₃		IF (I.EQ.K) MATCH = MATCH + 2
S ₃ ^k , S ₄		IF (J.EQ.K) MATCH = MATCH + 3
S ₄ ^k		IF (MATCH.EQ.0) GO TO 500
S ₅ ^k		IF (MATCH.EQ.1) GO TO 400
S ₆ ^k		IF (MATCH.EQ.2) GO TO 300
S ₇ ^k		IF (MATCH.EQ.3) GO TO 100
S ₅		WRITE (6,901)
	991	FORMAT (1H1, 11HEQUILATERAL)
		GO TO 900
S ₈ ^k	100	IF (J+K.LE.I) GO TO 600
S ₆	200	WRITE (6,902)
	992	FORMAT (1H1, 9HISOSCELES)
		GO TO 900
S ₉ ^k	300	IF (I+K.LE.J) GO TO 600
		GO TO 200
S ₁₀ ^k	400	IF (I+J.LE.K) GO TO 600
		GO TO 200
S ₁₁ ^k	500	IF (I+J.LE.K) GO TO 600
S ₁₂ ^k		IF (J+K.LE.I) GO TO 600
S ₁₃ ^k		IF (I+K.LE.J) GO TO 600
S ₇		WRITE (6,903)
	993	FORMAT (1H1, 7HSCALENE)
		GO TO 900
S ₈	600	WRITE (6,904)
	994	FORMAT (1H1, 14HNOT A TRIANGLE)
S ₉	700	STOP
		END

Figure 5-1 : FORTRAN Listing of Triangle-Type-Determination Program

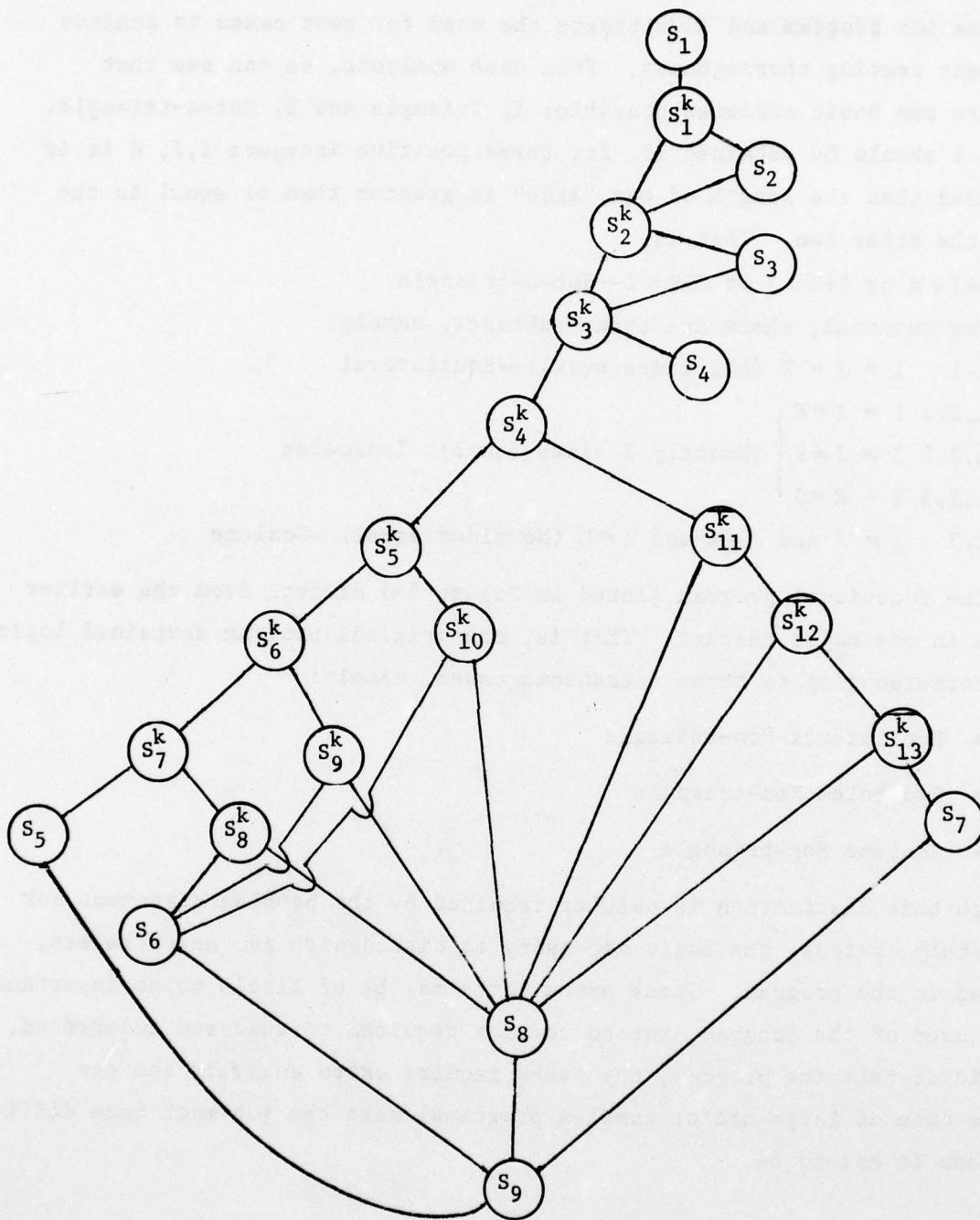


Figure 5-2 : Structure of Triangle Program

We now consider a more functional version of the triangle-type-determination program and investigate the need for test cases to achieve equivalent testing thoroughness. From case analysis, we can see that there are two basic outcomes possible: 1) Triangle and 2) Not-a-triangle. Outcome 2 should be obtained if, for three positive integers I, J, K it is determined that the length of one "side" is greater than or equal to the sum of the other two. That is,

$$I+J \leq K \text{ or } I+K \leq J \text{ or } J+K \leq I \rightarrow \text{Not-a-triangle}$$

For Outcome 1, there are three subcases, namely:

1.1 $I = J = K$ (All sides equal) \rightarrow Equilateral

1.2.1 $I = J \neq K$

1.2.2 $I \neq J = K$

1.2.3 $I = K \neq J$

(Exactly 2 sides equal) Isosceles

1.3 $I \neq J$ and $J \neq K$ and $I \neq K$ (No sides equal) Scalene

The functional program listed in Figure 5-3 differs from the earlier version in one major respect. That is, the original program contained logic paths corresponding to three extraneous cases, namely:

- Equilateral Non-triangle
- Isosceles Non-triangle
- Scalene Non-triangle

Although this distinction is neither required by the problem statement nor immediately obvious, the logic necessary to distinguish is, nevertheless, embedded in the program. These extra paths may be of little or no importance to the user of the program, but to someone required to read and understand, maintain or test the program, the paths require extra analysis and can (in the case of large and/or complex programs) make the job much more difficult than it has to be.

		NAMELIST/INPUT/I,J,K
S_1		READ (5, INPUT)
S_1^k		IF (I+J.LE.K) GO TO 400
S_2^k		IF (I+K.LE.J) GO TO 400
S_3^k		IF (J+K.LE.I) GO TO 400
S_4^k		IF (I.EQ.J) GO TO 100
S_5^k		IF (J.EQ.K) GO TO 200
S_6^k		IF (I.EQ.K) GO TO 200
S_2		WRITE (6,901)
	991	FORMAT (1H1, 7HSCALENE)
		GO TO 900
S_7^k	100	IF (J.EQ.K) GO TO 300
S_3	200	WRITE (6,902)
	992	FORMAT (1H1, 9HISOSCELES)
		GO TO 900
S_4	300	WRITE (6,903)
	993	FORMAT (1H1, 11HEQUILATERAL)
		GO TO 900
S_5	400	WRITE (6,904)
	994	FORMAT (1H1, 14HNOT A TRIANGLE)
S_6	700	STOP
		END

Figure 5-3 : Functional Triangle Program

The logical branching potential of this functional version of the program is illustrated graphically in Figure 5-4. As expected, there are eight distinct logic paths (L_i), they are easy to identify, and all are possible to execute (i.e., there are no phantom paths). It is also fairly easy to see that it is necessary to execute each path at least once in order to execute each and every branch in the program. Thus we need eight test cases to achieve testing thoroughness equivalent to that obtained with eleven test cases for the non-functional version. More important, however, is the ease with which the appropriate test cases can be specified. Figure 5-5 lists the segment-to-segment sequence for each path together with the logical conditions on the inputs which cause the path to be taken and the composite conditions which comprise the specification of the input domain partition.

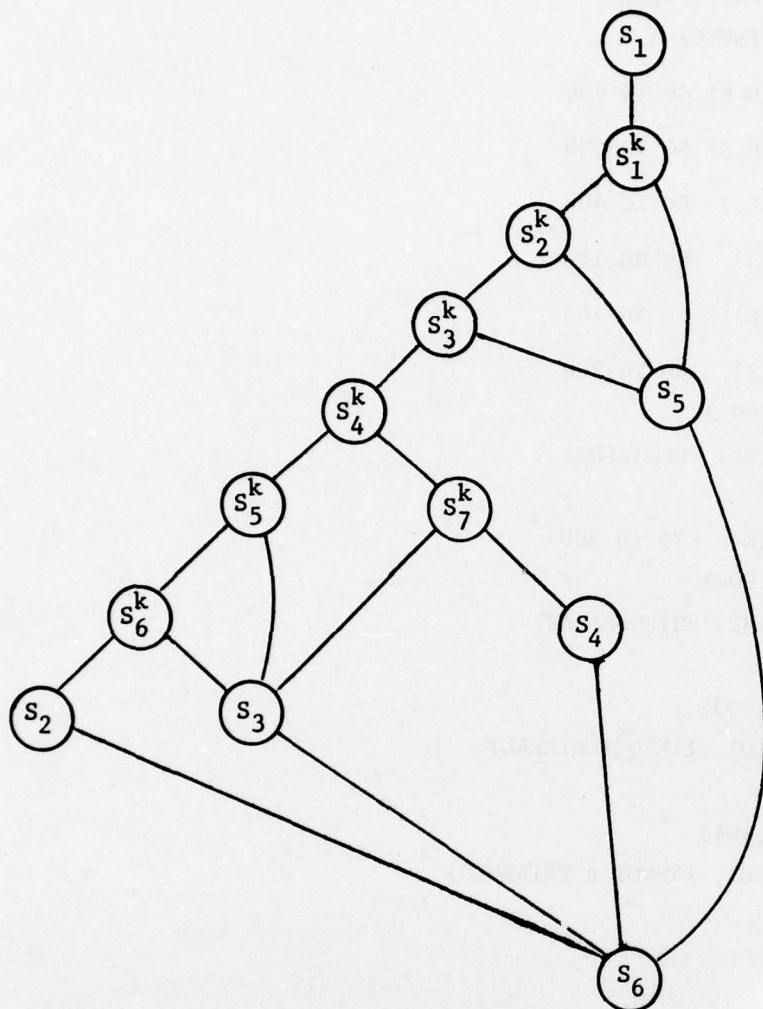


Figure 5-4 : Structure of Functional Triangle Program

<u>L_j</u>	<u>Code Sequence</u>	<u>Logical Conditions</u>
L ₁ :	S ₁ S ₁ ^f S ₂ ^f S ₃ ^f S ₄ ^f S ₅ ^f S ₆ ^f S ₂ S ₆	I+J>K, I+K>J, J+K>I, I ≠ J, J ≠ K, I ≠ K
L ₂ :	S ₁ S ₁ ^f S ₂ ^f S ₃ ^f S ₄ ^f S ₅ ^f S ₆ ^t S ₃ S ₆	Same as #1 except I=K
L ₃ :	S ₁ S ₁ ^f S ₂ ^f S ₃ ^f S ₄ ^f S ₅ ^t S ₃ S ₆	I+J>K, I+K>J, J+K>I, I ≠ J, J=K
L ₄ :	S ₁ S ₁ ^f S ₂ ^f S ₃ ^f S ₄ ^t S ₇ ^f S ₃ S ₆	Same as #3 except I=J and J ≠ K
L ₅ :	S ₁ S ₁ ^f S ₂ ^f S ₃ ^f S ₄ ^t S ₇ ^t S ₄ S ₆	Same as #4 except J ≠ K
L ₆ :	S ₁ S ₁ ^f S ₂ ^f S ₃ ^t S ₅ S ₆	I+J>K, I+K>J, J+K<I
L ₇ :	S ₁ S ₁ ^f S ₂ ^t S ₅ S ₆	I+J>K, I+K<J
L ₈ :	S ₁ S ₁ ^t S ₅ S ₆	I+J<K

G₁ = {I,J,K; (I+J>K) (I+K>J) (J+K>I) (I ≠ J ≠ K ≠ I)}

G₂ = {I,J,K; (I+J>K) (I+K>J) (J+K>I) (I ≠ J ≠ K = I)}

G₃ = {I,J,K; (I+J>K) (I+K>J) (J+K>I) (I ≠ J = K)}

G₄ = {I,J,K; (I+J>K) (I+K>J) (J+K>I) (I = J ≠ K)}

G₅ = {I,J,K; (I+J>K) (I+K>J) (J+K>I) (I = J = K)}

G₆ = {I,J,K; (I+J>K) (I+K>J) (J+K<I)}

G₇ = {I,J,K; (I+J>K) (I+K<J)}

G₈ = {I,J,K; (I+J<K)}

Figure 5-5 : Triangle Program Paths

5.3 Testing the Stack Program

To further investigate the testing of functional versus non-functional programs, several versions of the STACK program (Section 4.2) were developed and analyzed by the Automated Test Data Generator (ATDG)⁸. Although ATDG was originally developed by TRW to support the rigorous testing of software at the NASA Johnson Space Center, its capability of producing path-related test cases made its applicability to functional programs especially interesting.

The ATDG analyzes source code elements (e.g., subroutines), identifies conditional transfers, and generates a set of paths containing all of these transfers. By application of powerful mathematical techniques, the number of paths required to cover all transfers is minimized. Supplementary information is also provided by ATDG to support derivation of actual test input data to execute the paths.

Within ATDG, a rather elaborate algorithm is included for the sole purpose of avoiding the generation of unexecutable (i.e., phantom) paths. Since functional programs contain no phantom paths, it was expected that ATDG resulting test cases would correspond to the previously defined functional test cases. In each instance, the ATDG-generated cases (i.e., those required to exercise all conditional transfers at least once) corresponded exactly to those designed to test all functions⁸. Finally, a rough comparison was obtained for the ATDG execution time required for the functional versions of STACK versus that required for similar (i.e., approximately the same size and logical complexity) but non-functional programs. The execution time was substantially less for the functional STACK programs, and this is most likely attributable to the lack of time required to process, recognize and discard phantom paths.

In completing the STACK/ATDG experiment, it was recognized that an important relationship exists between the fundamental concepts underlying FP and ATDG. This relationship is based on the fact that ATDG (and related technologies like symbolic execution and program proving) attempt to generate and work with one (or more) path(s) for each of the G_j as defined in Section 2.2. It's natural to conclude that such a tool could be very useful as a management aid in analyzing functional programs to assure that none of the intended F_j are inadvertently omitted and (probably just as important

but much more difficult to detect) that no extraneous functions are included during implementation. In addition, the tool could be used to support FP analysis of existing code and aid in the identification of existing functions and thus an understanding of current functional content.

6.0 FUNCTIONAL PROGRAMMING IMPACT EVALUATION

Since FP is a new way of writing programs, it may be expected to have an impact on programming language requirements and on the reliability and maintainability of programs developed using it. The results of a study of this impact are reported in the following sections.

6.1 Impact on Programming Language Requirements

Although FP concepts are applicable whatever programming language is used, the following questions can be raised:

- Are there language features capable of making functional programming easier?
- Are there programming standards - i.e., restrictions on the use of language features - that would make functional programming easier?

FP partitions the input domain E into the subsets G_j associated with logic paths L_j and functional capabilities F_j . It therefore focuses on the inputs. A programmer writing a program should accordingly have a clear picture of the input variables, the values assignable to them, and how the ranges of the variables are partitioned to form the G_j . Most programming languages do not aid the programmer in clearly defining the inputs to the program he is writing. Generally, input variables may appear, for the first time, anywhere within a program and in many programming languages - e.g., FORTRAN - a programmer does not have to explicitly declare an input variable unless he wishes to restrict its range to a set of values different from that defined by a computer word.

Visibility of the input domain would be aided by requiring that all input variables be declared at the beginning of a program. EUCLID, e.g., requires a program to list all the identifiers input into a procedure.

Partitioning of the input domain is specified using branch expressions. Since FP associates an input domain partition G_j with an executable logic path L_j , visibility of that association is often improved through use of compound branch expressions. Thus, facilities to aid the writing of compound branch expressions and their consistent and efficient interpretation could contribute strongly to the writing of functional programs.

Structured programming has been successfully introduced into many programming groups by expressing it in terms of a few simple rules - viz., restricting code structural elements to:

- in-line code segments
- IF A THEN B ELSE C
- DO S WHILE A
- DO S UNTIL B

Since these rules are relatively simple, programmers can easily learn to follow them. Are there similar rules for FP that can be just as easily learned and followed?

That question can be investigated by using FP notation to analyze differences between FP and structured programming. As originally proposed by Dijkstra, structured programming is essentially writing a program in the form:

- IF S_1^k THEN C_{11} ELSE C_{12}
IF S_2^k THEN C_{21} ELSE C_{22}
:
IF S_n^k THEN C_{n1} ELSE C_{n2}

In this representation of a structured program, S_i^k represents a branch expression and C_{i1} and C_{i2} represent executable code segments containing the allowable code structure elements. This form of the program code provides a linear structure. It can be read sequentially; i.e., the code in line i is executed before the code in line $i+1$. Dijkstra sought by this means to obtain increased visibility of program structure from the program text, an objective of FP, too. This linear structure does improve structural visibility over older methods of programming because it eliminates the complex, often convoluted branching present in many programs; however, as noted in functional programming analysis of structured programs, it can also encourage the formation of phantom paths and, in the process, obscure both structural and functional visibility.

FP can be compared with structured programming by looking closely at a functional program in its simplest form:

- IF S_1^k THEN C_1
- ELSE IF S_2^k THEN C_2
- ELSE IF S_3^k THEN C_3
- ⋮
- ELSE IF S_n^k THEN C_n

In this representation of a functional program, S_1^k represents the branch expression which upon TRUE evaluation initiates execution of the logic path $L_1 = S_1^t C_1$; S_2^k represents the branch expression which upon TRUE evaluation initiates execution of the logic path $S_1^f S_2^t C_2$; and so on. C_1 represents an executable code segment, possibly containing DO structures. In this form, a functional program is a structured program in which C_{12} is represented by C_1 , C_{22} is represented by C_2 , etc.

The problem in this form of functional program is the code redundancy generally occurring in the C_i ; i.e., the code sequences in each of the C_i generally will have subsequences repeated in two or more C_i . This leads to writing more lines of code, potentially resulting in more errors, and a requirement for more computer resources (storage and time) to process the program. As is shown in sections 3 and 4, programs can be designed to eliminate all or most of the code redundancy, while retaining the functional property of no phantom paths. Therefore a requirement for a language to aid in the writing of functional programs is to provide facilities for specifying in a compact, visible way the intended execution sequence of code segments for each C_i .

6.2 Impact on Software Reliability

FP was developed to aid the production of reliable programs. Has it achieved this goal? The following evidence supports the conclusion that FP does contribute to producing more reliable programs:

- Rewrite of programs originally written using conventional programming methods (see Section 3) generally results in programs having fewer statements (as much as 50% fewer in the case of Routine A). The fewer statements means that fewer characters and words have to

AD-A052 997

TRW DEFENSE AND SPACE SYSTEMS GROUP REDONDO BEACH CALIF
FUNCTIONAL PROGRAMMING. (1)

F/6 9/2

FEB 78 J R BROWN, E C NELSON

F30602-76-C-0315

UNCLASSIFIED

RADC-TR-78-24

NL

2 OF 2

AD
A052997



END
DATE
FILMED

6-78

DDC

be written, resulting generally in fewer typographical errors. The shorter program text also makes the text easier to read and comprehend increasing the probability that a programmer will spot an error that has been made and correct it before the program goes into use.

- Since in a functional program all paths are executable, it is relatively easy to trace a path through the program mentally executing the path and assuring one's self that program operation will proceed as intended.
- Since a program written using FP methodology will have its functional requirements - i.e., the (\hat{G}_j, \hat{F}_j) - defined before the code is written and since the executable logic paths L_j are constructed to correspond to the functional requirements (\hat{G}_j, \hat{F}_j) , it is possible to mentally compare the code of logic path L_j with its associated functional requirement and by that process detect errors in the code.
- Since the \hat{G}_j for a functional program are explicitly defined, it is relatively easy to design test cases to demonstrate satisfaction of functional requirements - viz., select an E_i from each \hat{G}_j - assuring verification of all functional requirements.

The preceding arguments are theoretical, stating what should happen. An empirical argument is obtained from the experience with the STACK program (see Section 4.8). STACK was written using FP methodology - i.e., defining the functional requirements (\hat{G}_j, \hat{F}_j) first and then designing the program to have logic paths L_j corresponding to the (\hat{G}_j, \hat{F}_j) . That program was tested functionally, designing test cases by choosing E_i from each \hat{G}_j . Test cases for each functional requirement were executed and all executed correctly, supporting a conclusion that with a high probability there were not errors in STACK as it was originally written. Although this is only one case and the absence of detectable errors could be fortuitous, the results of STACK testing support the conclusion that FP contributes to producing reliable programs.

6.3 Impact on Software Maintenance

Software maintenance - solving of problems occurring in operational use of a program and adapting a program to a changing operational environment - is one of the largest components of life cycle cost. FP can impact software maintenance in several ways:

- By contributing to production of more reliable programs, FP reduces the number of problems maintenance programmers have to solve and hence reduces maintenance cost.
- FP produces a description of a program - viz., the functional requirements (\hat{G}_j, \hat{F}_j) and an association of each functional requirement with an executable logic path and functional capability - that a maintenance programmer, who was not involved in developing the program, can use to understand the program and to solve problems. E.g., a software problem generally can be associated with a specific input or a specific set of inputs. From this input, the FP program description readily aids associating the problem with a specific G_j , F_j , and L_j . From examination of the problem evidence and the G_j , F_j , L_j descriptions, the source of the problem can be more quickly identified and the problem solved at less than usual expense.
- If the problem is elusive, the FP description can aid the design of test cases to make the problem more visible.
- Once the source of a problem is identified, the FP description aids identification of the code to be modified.
- A new operational requirement, if it is functional in nature, can be expressed in the form (\hat{G}_j, \hat{F}_j) . This defines the functional requirement precisely. This form of the requirement when added to the FP description of the program will show immediately whether the new requirement is a modification of an old requirement or whether it is a completely new requirement. With the revised set of functions, the modifications to the code necessary to implement the revisions are easily designed and the test cases needed to verify the implemented revision can be readily constructed.

- If the new operational requirement is a performance requirement, generally it can be related to certain of the functional requirements - e.g., to speed up execution of function F_j - and the FP description can aid finding the code to be modified to fine tune program performance.

7.0 SYSTEM LEVEL FUNCTIONAL PROGRAMMING

The concepts of FP were developed and applied initially at the routine level. Programs are, however, collections of routines and software subsystems and systems generally are collections of programs and routines. Thus FP concepts can also be applied at those levels, for a routine is connected to another routine by having its output $F_j(E_i)$ serve as the input E_m to the connected routine where it initiates execution of logic path L_k specifying function F_k and resulting in output $F_k(E_m)$.

- $E_m = F_j(E_i)$
- $F_k(E_m) = F_k(F_j(E_i))$

Thus execution of two connected routines results in composition of the functions specified by the individual routines. This composition can be extended to the subsystem and system level.

Taken as a whole, the software for a fire control system specifies a function. That function generally will be analyzable into functions F_j corresponding to specific functional requirements. A particular functional requirement (\hat{G}_j, \hat{F}_j) may define the fire control rules corresponding to a specific combat situation while another functional requirement (\hat{G}_k, \hat{F}_k) will define the functional requirement corresponding to another combat situation. It is equally possible to envision FP application at the system level in terms of a computer operating system. Here the inputs are things such as application programs, files, and machine resources. Outputs are things such as initiation of execution of a program and execution of a service, a distinct function of the operating system. The complete requirements for an operating system can be defined by specifying, for each operating system service, its domain G_j and the function F_j to be performed.

8.0 TOOLS TO SUPPORT FUNCTIONAL PROGRAMMING

Although the concepts of functional programming are relatively simple, application of them can involve considerable analysis, writing, and book-keeping. The tedious nature of these chores may inhibit the use of the methodology. Accordingly, tools need to be developed to automate the manual effort required and to reduce possibilities for human error.

An informal preliminary requirements specification for a tool (FPA) to support functional programming analysis of existing programs is:

- FPA is required to:
 - Read the text of, for example, a JOVIAL (J3) program and identify each branch expression and in-line code segment in the program.
 - Assign to each branch expression a symbol $S(J,K)$, with J denoting an integer value representing the numerical order of the branch expression in the program.
 - Assign to each in-line code segment the symbol $S(J)$, with J denoting an integer value representing the numerical order of the segment in the program.
 - Produce a listing of the program annotated to show in the left hand margin a symbol $S(J,K)$ for each line of code containing a branch expression, with the integer J in the symbol denoting the numerical order of the branch expression and to show, also in the left hand margin, a symbol $S(J)$ for each line of code beginning an in-line code segment.
If a line of code contains both a branch expression and the beginning of a code segment, the branch expression symbol shall be printed first followed by a comma and the segment symbol printed to the right of the comma.
 - List all branch expression symbols and their corresponding branch expression in the format - the branch expression symbol followed by a colon followed by the branch expression.
 - List all in-line code segment symbols and their corresponding segments in the format - the segment symbol followed by a colon followed by the segment.

- List all input variables - i.e., variables to which values must be assigned in order for the program to be executed and any data declarations involving those variables.
- Print the program in FP notation.

Such a tool would greatly ease the job of analyzing existing programs, producing an annotated listing, a listing of the branch expression and segments, and a listing of the program in FP notation.

Another tool (FPS) to support the writing of functional programs has the following informal preliminary requirement specification:

- FPS is required to:
 - Accept as input, the name of each input variable and its range.
 - Accept as input, specifications on input domain partitions in the form of branch expressions.
 - Accept as input, specifications on the functions associated with each input domain partition - viz., a specification of the range of each function and a specification of the rule assigning to each input E_i in G_j a value $F_j(E_i)$ in the range set.
 - List the name of each input variable, followed by its range specification.
 - List the specifications of the input domain partitions in the format - G_j followed by a colon followed by the branch expressions defining the partition and a listing of any input variables belonging to G_j but not partitioned.
 - List the specifications for the functions corresponding to each input domain partition in the format - R_j followed by a colon followed by the range set specification and F_j followed by a colon followed by the assignment rule specification.

- Check the input variable ranges against the ranges specified in each input domain partition. Print any variables whose range is not completely covered in the input domain partition specifications and identify the portion of the range not covered.

Such a tool would assist in maintaining the functional requirements and in identifying any missing requirements.

9.0 CONCLUSIONS AND RECOMMENDATIONS

At the outset of the contractual project effort reported here, functional programming (FP) was little more than a gleam in the eye of the authors. To be sure, several applications of FP analysis to some small routines had been accomplished and the results (i.e., improved understandability, reliability and performance) were particularly encouraging. The ultimate goal then was to extend the scope of applications, investigate both the potential capabilities and limitations of the FP techniques, and, where appropriate, refine and add to the techniques as necessary to form a disciplined and comprehensive yet practically useable programming methodology. The more realistic goal of our initial study was to invest limited time and resources necessary to concentrate on key issues whose resolution should and will play a major role in the future evolution and acceptance of FP as an integral part of improving software engineering practice. With completion of the subject study and this report of key findings, the latter goal has been reached and substantial progress toward the ultimate goal has been achieved.

Still, the FP methodology as defined and demonstrated in this report is in its infant stage. For example, we have shown that in every sample application it was possible to carefully analyze a computer program and revise the logical structure so that the resulting program's functional content could be more readily discerned than that of the original, non-functional version. In some cases the improvement has been dramatic (in terms of reduced number of distinct logic paths), but in others only minor improvements were possible. In short, we have been able to show that in most cases we should be able to effect some improvement in the functional visibility of computer programs, but we have yet to prove that we can do so in all instances.

Similarly, we have formulated a sequence of eight basic steps to be taken in developing a functional program from scratch. In this study, however, we have gone no further than to demonstrate (by way of varied examples) that the procedure accurately implements the FP principles and that a good programmer can (with only a little training) actually follow the procedure in successful development of functional programs.

One of the major results of the FP study was to further confirm the hypothesis that time and effort spent in rigorously (i.e., completely and precisely) specifying software requirements leads to marked reductions in the usual problems and costs of life cycle software development, test, operation and maintenance.^{9,10} In fact, we observed from one experiment (the development of multiple versions of the stack creation and maintenance program) that if the original program requirements are written "functionally" (i.e., in terms of distinct, required functions corresponding to distinct subdomains of possible values of program inputs), then it is almost impossible to design and implement anything other than a functional program to do the job. As a consequence, a very high degree of traceability (from requirements to code) is achieved, and this traceability directly contributes to eliminating ordinary problems owing to differing interpretations and omissions due to oversights.

At this stage in the evolution of a more rigorous and disciplined software development process, many new techniques have been proposed and some have been heavily advertised as a panacea for the past problems of software production. Understandably, they who have experienced and/or paid for the problems in the past are eager to find new, more effective methods that can help to eliminate similar problems in the future. Such a climate can be very good or very bad with respect to the development, application, refinement and eventual, cost-effective utilization of any particular new technique. For instance, if one makes great claims about the potential of a technique (in order to get a lot of people to try it out), it is possible, in fact, likely that the technique will (in actual practice) fail to live up to the claims. On the other hand, if no claims are made it can be difficult to get the attention of those who can presumably use and benefit from the technique. In either case, the two things that are most needed (lots of experimental applications in carefully controlled situations followed by objective evaluation of not only current capabilities and limitations but also the probable and possible capabilities achievable through continued development and refinement) will not be forthcoming. The result; an otherwise sound and very promising new technique might die before it ever really gets off the ground.

We hope for and expect a nicer fate for Functional Programming. We have tried in this report to make only those claims for which we have reasonable supportive evidence of their validity. At the same time we are careful to point out that, although our limited experiences with FP have all been positive and encouraging, we see the need for a great deal more experimentation and study and still more experimentation before the full potential of FP can be brought effectively to bear on a wide variety of software development activities. It is worth noting that it has taken about eight years and much study for structured programming to grow from newly proposed and only slightly understood principle to commonly accepted and widely used modern programming practice. We suggest that the time has come to give strong attention to the outstanding problems that still plague developers and users of software and to invest appropriate time and energy to bring about an accelerated maturation of FP principles and methodology. If we really work at it, we can probably be producing truly reliable and highly maintainable software long before another eight years has passed us by.

During the course of the FP study, many reference documents were reviewed and many discussions were held with people currently working at advancing a variety of software development and test technologies, including: software requirements engineering, structured programming, automated test data generation, symbolic program execution and program proving. The overall outcome of this investigation of other technologies has been to expose the existence of very strongly related objectives and, in some cases, almost identical techniques. For example, we found that in completing a FP analysis (as defined in this report) we were actually accomplishing the basic steps and deriving the necessary information required for automated generation of test data. Moreover, it was found that those who are deep into symbolic execution and program proving are fully as interested in the actual functional content of program paths and are fully as frustrated by unwarranted program complexity as we are in attempting a thorough FP analysis. Among all these technologies, the dominant common thread is a particularly strong interest in striving (whether by constructive techniques as in FP, analytical approaches as in symbolic execution and program proving, or actual execution and evaluation of test results) to remove any and all

differences between functional requirements (i.e., what the software needs to do) and functional capabilities (i.e., what the software actually does). This close tie between technologies is fortunate, in that concerted research and development leading to new breakthroughs in any one area is almost certain to yield related advances in others. For example, continued refinement and broad application of FP can lead to production of programs having a much smaller number of logic paths than might otherwise be the case, and, as a direct result, it should be possible to complete path-by-path proofs of much larger programs than any attempted to date.

Based on the above consideration, it is recommended that we not only continue needed research into the fundamentals of software structure (in general) and FP constructive techniques (in particular), but also that we maintain the fullest possible understanding of ongoing research and new advances in related technology areas. A comprehensive research program should include:

- continued, iterative application and refinement of the FP methodology documented in this report,
- special studies of the impact of evolving FP methodology on other technologies,
- investigation (through experimental application) of the feasibility and merit of applying FP principles in the writing of formal system and software requirements specifications,
- detailed design, implementation and experimental application of the FP analysis and FP support tools specified in this report and continued analysis of the type and extent of support obtainable from existing tools (e.g., compilers, text editors, automated test data generators and symbolic execution systems), and
- study of new advances in other technology areas (e.g., improved network analysis and path representation techniques, development of more easily proven programming constructs and language features, and program test measurement and reliability assessment/prediction technology) and subsequent incorporation of appropriate new technology as integral elements of the FP methodology.

If the needed research and development is carried out, it is highly likely that FP will become an important, constructive element of future software engineering discipline and, perhaps more importantly, we should achieve unprecedented synergism among previously distinct (and not altogether cooperative) software technologists. If so, we may begin to reap the real rewards of truly advanced software production technology much sooner than one might expect.

10.0 BIBLIOGRAPHY

This section provides a list (Section 10.1) of the source materials referenced in the test of the report. In addition, many other papers and technical reports were reviewed prior to and during the course of the FP study, and the most relevant of these are listed in Section 10.2.

10.1 References

1. E. K. Blum, "The Semantics of Programming Languages, Part I," TRW SS-69-01, (1969).
2. E. R. Anderson, F. C. Belz, and E. K. Blum, "SEMANOL (73), A Metalanguage for Programming the Semantics of Programming Languages", Acta Informatica 6, 109-131 (1976).
3. Krause, K. W., M. A. Goodwin and R. W. Smith, "Optimal Software Test Planning through Automated Network Analysis," Record 1973 IEEE Symposium on Computer Software Reliability, May, 1973, pp. 18-22 and TRW-SS-73-01, April, 1973.
4. R. H. Hoffman, and G. L. Houser, User Information for the Interactive Automated Test Data Generator (ATDG) System, Revision 1, NASA Johnson Space Center Internal Note No. 75-FM-88, January, 1977.
5. R. H. Hoffman, ATDG Impossible Paris Detection Capability Study Report, TRW Technical Report 76: 2511.3-115, July, 1976.
6. Brown, J. R., "Practical Applications of Automated Software Tools," WESCON 1972, Session 21 and TRW-SS-72-05, September, 1972.
7. Gruenberger, F., "Program Testing: The Historical Perspective," Program Test Methods, ed. W. C. Hetzel, Prentice-Hall, 1973, pp. 11-14.
8. Hoffman, R. H., "ATDG Analyses of STACK Programs," TRW Interoffice Correspondence 77:2511.3-117, 27 May 1977.
9. Thayer, T., E. C. Nelson, et al, Software Reliability Study Final Technical Report, TRW Report No. 76-2260.1 9-5, March 1976.
10. Brown, J. R., Impact of MPP on System Development Final Technical Report, RADG-TR-77-121, May 1977 and TRW Report 29115-6001-RU00, January 1977.

10.2 Related Publications

Brown, J. R. and M. Lipow, "Testing for Software Reliability," Proceedings of the International Conference on Reliable Software, April, 1975 and TRW-SS-75-02, January, 1975.

Brown, J. R., and K. F. Fischer, "A Graph Theoretic Approach to the Verification of Program Structures," to be published in TRW Software Series.

Dahl, O. J., E. W. Dijkstra and C. A. R. Hoare, Structured Programming, Academic Press (London), 1972.

Boehm, B. W., J. R. Brown, E. Horowitz, et al, Practical Strategies for Developing Large Software Systems, Addison-Wesley, 1975.

Nelson, E. C., "A Mathematical Theory of Data Structures," TRW-SS-71-03, May, 1971.

Hecht, M. S., and J. D. Ullman, "Flow Graph Reducibility," SIAM Journal of Computing, Vol. 1, No. 2, June, 1972, pp. 188-202.

Earnest, C. P., K. G. Balke, and J. Anderson, "Analysis of Graphs by Ordering of Nodes," Journal of the Association for Computing Machinery, Vol. 19, No. 1, January, 1972, pp. 23-42.

Howden, W. E., "The DISSECT Symbolic Evaluation System," Computer Science Technical Report No. 8, University of California, San Diego, February, 1976.

Nelson, E. C., "A Statistical Basis for Software Reliability Assessment," TRW-SS-73-03, March, 1973.

King, J. C., "A New Approach to Program Testing," Proceedings of the International Conference on Reliable Software, April, 1975, pp. 228-233.

Liskov, B., "Specification Techniques for Data Abstractions," Proceedings of the International Conference on Reliable Software, April, 1975, pp. 72-87.

Parnas, D. L., "A Technique for Software Module Specification with Examples," Communications of the ACM, Vol. 15, No. 5, May, 1972, pp. 330-336.

Floyd, R. W., "Assigning Meaning to Programs," Proceedings of a Symposium in Applied Mathematics, Vol. 19, 1967, pp. 19-32.

Hoare, C. A. R., "An Axiomatic Basis for Computer Programming," Communications of the ACM, Vol. 12, No. 10, October, 1969, pp. 576-580, 583.