AUTOMATIC PROGRAM ANALYSIS

BY

ANN HALLER

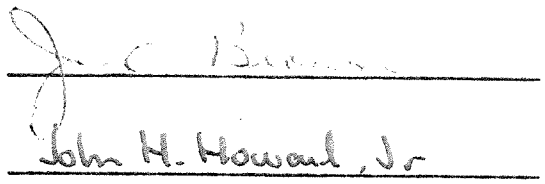
August 1974 TR 38

This report constituted the author's M. A. Thesis in Computer Science at The University of Texas at Austin.

DEPARTMENT OF COMPUTER SCIENCES
THE UNIVERSITY OF TEXAS AT AUSTIN

AUTOMATIC PROGRAM ANALYSIS

APPROVED:



AUTOMATIC PROGRAM ANALYSIS

by

JULIA ANN PEEK HALLER, B.A.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF ARTS

THE UNIVERSITY OF TEXAS AT AUSTIN

August 1974

ACKNOWLEDGEMENTS

I would like to acknowledge Dr. J. C. Browne for his guidance and support in the development of this thesis and Mr. James L. Turner, Mr. R. E. Meeker, Jr., and Mr. Gene L. Lasseter for their help on the FACES project.

This work was supported in part under NASA contract NAS8 28084.

Ann Haller

July, 1974

TABLE OF CONTENTS

I. GOALS AND METHODS IN DEVELOPING AUTOMATIC PROGRAM ANALYSIS SYSTEMS

Anyone who has had experience in developing a large software package is aware of the tremendous problem involved in debugging and testing the programs sufficiently enough to insure that they are reasonably correct and perform according to the writer's specifications. In fact, the largest percentage of a programmer's time is spent in debugging and testing programs rather than in designing or coding them. In an attempt to alleviate this problem, automatic program analyzers have been developed in which the programmer releases some of the more tedious tasks of program analysis to be done automatically by the program analyzer. It seems reasonable that the programmer should use the computer, a tool with which he is most familiar, not only for running programs, but for aiding him in analyzing them also. The computer is adaptable to this task because of its potential for collecting sets of data, organizing information, and discerning patterns which the human programmer cannot easily recognize.

Often, in debugging a program, a person spends hours searching through the code looking for the source of the problem. A program analyzer can prevent him from getting lost in the code by automatically extracting and displaying key information that may be obscure to the programmer. In

this way information from one portion of the program can be correlated with information from other parts, and the programmer can more easily recognize patterns in the overall structure of the program. Although the programmer still must judge the source of the problem from the information displayed to him by the analyzer, at least he has a better basis from which to search.

The program analyzer may also be helpful in the fine tuning of what seem to be "running" programs. Subtle problem areas that the programmer has overlooked may be pointed out and poor code constructs that can cause inefficient execution may be recognized. The analyzer may also aid in producing environments for testing of key code and in evaluating test runs to insure that key code has been executed. Estimates of core size and execution time requirements of the program can also be made automatically, and the structure of the program can be analyzed to aid in determining optimum overlay structures.

Program documentation and maintenance can also be simplified by automatic analysis. Improvements in documentation can be made by abstraction and display of program structure, interface information, and other details that should be recorded for proper program maintenance. When the program must be modified or extended, the programmer can use the automatic analyzer to outline necessary details of the program and thus avoid generating bugs by the addition of new code. Then as the program environment changes,

the analyzer can be used to recognize and perhaps change constructs as needed for the new environment.

BASIC TYPES OF AUTOMATIC PROGRAM ANALYSIS

There are two basic methods of automatic program analysis--static analysis and dynamic analysis. In the static method, information is extracted from the user's source code only. In this method, complex structural information about the program, that may be obscure to the user, is extracted and displayed. The analyzer can point out syntactically correct but semantically dubious constructions that are known to be dangerous or troublesome in the particular language. A static analyzer may contain a large variety of analytic routines that extract and display different aspects of the program. It is advantageous that the user be able to choose the routines that will be helpful to him in analyzing a specific program. He then can have before him only those details which are relevant for a particular purpose. This method lends itself to an interactive system whereby the user can type in a question about his program, analyze the results, and either modify his program or ask another question.

The second method of automatic analysis is called dynamic because the analysis is performed during execution time of the user's program. This method requires a preliminary analysis of the user's source program which alters the source code by inserting statements to gather the required

data. For example, the analyzer could dynamically monitor the values of certain variables or monitor the frequency of execution of each statement or routine in the program. The dynamic analysis method is helpful in evaluating the "live" performance of a program since the program is run with actual data. In the static method it is often times impossible to predict how a certain portion of the program will behave without knowing what the data will be. The dynamic method, however, limits the evaluation to the program's performance on a certain data set, whereas the static method gives a more general analysis. The additional code inserted by dynamic analyzers may produce side effects in the user's program by altering the program structure and may affect its performance. The optimum program analyzer should incorporate a combination of both dynamic and static methods.

POSSIBLE FUNCTIONS OF A PROGRAM ANALYZER

A program analyzer could perform a variety of functions to aid the user in his program evaluation. The specific purposes for which one would wish to have an automatic analyzer depend in part on the particular language one wishes to analyze. However, there are basic characteristics of programs that are universal to most languages which a good automatic aid may inspect and analyze.

Program Validation

One area of program validation is program data analysis. That is, an automatic analyzer may recognize data constructs (such as arrays in FORTRAN), noting the sizes and types of these constructs and where they occur, and storing this information in a table for further reference. Using this information, the analyzer can then check data references and note possibly dangerous constructs. For example, in FORTRAN, the analyzer could check subscript ranges in array references. It could also inspect the usage flow structure of the program, that is, the relationship of data objects to each other and the possible ways they could affect each other. For instance, one may wish to know what variables can affect the value of a specific variable at a certain point in the program or what other variables a particular variable could affect. This would be helpful in determining the effects of changes in input data, in searching for the cause of variable value errors, and in determining appropriate data for test cases. One may also want to verify that all variables are properly initialized before they are used. This usage flow analysis can be done on a modular or global level depending on the user's needs. That is, one may wish to trace the usage flow only to within subroutine boundaries, treating each subroutine as a separate entity; or it may be preferable to get an overall view of how data items interact by perform-

ing a global usage flow analysis over the entire program. A dynamic analysis could also be done to monitor the actual values of variables during execution of the program.

Another area of possible inspection is flow of control analysis in which the structure of the program is extracted and represented as a graph. The graph can be used to determine a variety of structural details such as looping structure, unreachable code, code without exits, and undefined functions. Static path analysis capabilities can be incorporated into the analyzer to trace possible paths through the program, starting at a certain point or ending at a certain point. This automatic tracing technique can replace the old-fashioned method of hand-tracing through a program searching for the cause of a bug. One may also wish to include dynamic path analysis to monitor the frequency of execution of different branches or modules of the program. This enables the user to pinpoint those areas of the program that add most to the execution time, thereby giving him an idea about which areas should be optimized to reduce the execution time most effectively.

Interface analysis is another useful area of automatic aid to program evaluation. This involves inspecting the interface structure between program modules, such as global variable storage and argument transmission. The analyzer can check to see if any inconsistencies exist that may be hidden from the programmer. For instance, the FORTRAN programmer may want to know if all common blocks of the same

name have the same length and structure or if an actual parameter is passed to a formal parameter of a different type. Such constructs are not illegal in FORTRAN, but can be dangerous if the programmer is unaware of them. The automatic analyzer can extract the information and print it in readable form for the benefit of the user.

Another possibility is to automatically check all output statements in the user's program, comparing them with format statements to insure that no inconsistencies exist. The format specifications can also be checked for conformity with the particular mechanical devices to be used. This output analysis can be especially useful for programs that write to files or other devices that cannot be readily seen by the programmer.

One of the most time-consuming tasks in preparing a program for use is testing the program after it appears to be in running order. Ideally, the programmer must prepare a set of test cases that will insure that the entire program has been tested. An automatic analyzer can help by monitoring the test runs dynamically and keeping statistics on which statements and branches of the program are actually executed during the tests. Of course it is impractical and usually impossible to test all cases of a program, but it is helpful to know at least that all parts of the program have at some time been executed or that all branches have been taken, since errors are often found in those parts that are rarely used. Conceivably, the analyzer could generate

the proper data for test cases that would insure that all parts of the program are executed. This is complex but offers great benefits.

Structural and Timing Analysis

A program analyzer can aid the programmer in predicting the approximate anount of memory it will use and the approximate amount of time it will take to execute. This is done by recognizing the program structure and the machine code equivalents for the source code statements. Of course the size and execution time of a program often depend largely on the data, so that a static prediction can only produce rough estimates. It can however aid the user in determining which variables will be most crucial in affecting the size or execution time.

This aspect of inspection also incorporates an analysis of the program's modularity. That is, it recognizes the overall structure of the program, noting groups of routines that seem to represent somewhat independent modules. This information can be useful in helping the user to determine an overlay structure for very large programs or in determining portions of the program that could be tested independently.

Documentation

Documentation of large programs can be augmented

by a program analyzer since much of the information collected by the analyzer is that which should be recorded in the documentation. For instance, the structure of the program could be displayed in flowchart form by the automatic analyzer. The programmer could be given the option to choose the level of detail at which the flowchart is to be constructed. The overall structure of the program could then be displayed in abbreviated form,or a graph of each subroutine could be displayed in detail.

A tabulation of other information could also be displayed for documentation purposes. Interface information describing global variables and argument transmission between routines are important aspects of complete documentation that could be recorded in organized form by the analyzer. Data structure information such as variable sizes and types and information concerning their use could also be tabulated to simplify the alteration of these structures due to changing specifications. All loops within the program could be documented along with the restrictive conditions needed to insure proper execution.

Performance and Maintenance

In most organized programming projects, standards are set up to prohibit programmers from using constructs that may be legal in a language but hazardous or inefficient to use on the specific system. It would be helpful to include in an automatic analyzer the capability for

recognizing these constructs. The constructs could be simple statement types or more complicated structures that would be subtle to the programmers' eyes. Whatever the case, the automatic analyzer would replace having to search through the code looking for these constructs.

Performance efficiency can also be enhanced by automatic analysis. A frequency of execution study on the source code statements can point out areas of code where most of the execution time is spent, thereby stimulating the programmer to improve those areas for better efficiency. Constructs that are known to be inefficient can be recognized by the analyzer and perhaps automatically replaced with improved ones. Maintenance of large programs can be simplified by using the analyzer to recognize and tabulate specific code constructs so that when compilers or other aspects of the program's environment are changed, alterations in the source program can be done automatically.

METHODS FOR DEVELOPING A PROGRAM ANALYZER

There are many things to consider in developing a program analyzer to perform the functions described above. One must first decide if the analyzer is to be a general purpose aid, capable of performing a variety of functions, or if its use will be restricted to a narrower specified purpose. Each task of the analyzer requires certain information to be extracted from the source code, and many of the

tasks require similar data. Therefore it is possible to build a large, general-purpose pre-processor that will gather a wide variety of information about the program to build a data base from which many tasks can be performed. This will make possible the continual expansion of the automatic analyzer with little adjustment to the data collecting routines. However, if the analyzer is only intended to aid the user in a specific, predefined task, then the large data base will be unnecessary.

One possibility is to incorporate the program analyzer into a compiler. Options could be included in the compiler to give the user the choice to use or not to use each aspect of the analyzer. This has the advantage of simplifying the process for the user, but it could make the compiler cumbersome and inefficient. If the alternative method of developing the analyzer as a separate tool is chosen, it will simplify the development of the analyzer itself since much of the code-generating information will not be needed. The analyzer could also be written in a more machine-independent form, thereby simplifying its transportation to other computer installations.

PROPOSED STRUCTURE FOR A GENERAL-PURPOSE ANALYZER

A general-purpose program analyzer could be developed with the following four main modules:

1. Source code pre-processor--scan of source code with concurrent data collection and storage, and code insertion if dynamic analysis is to be used.
2. Static analysis processors--processors of data collected during scan; organizing, interpreting, and printing information about the source code in the following areas:
   (a) Validation
   (b) Structural and timing analysis
   (c) Documentation
   (d) Performance and maintenance
3. Dynamic monitors and data collection during execution.
4. Post-processing and printing of information collected during execution.

The remainder of the chapter will discuss each of the four modules and the requirements for performing the functions discussed in the previous section.

Module 1: Pre-processor

The pre-processor serves as the data-collecting agent for information to be processed from the source code. It scans the user's source program storing information needed for the static analysis. This information can be stored in a set of tables, the details of those tables depending

on the specific functions the analyzer is designed to perform. In order to perform the data analysis as mentioned in the previous section, the tables must contain descriptors for all data values that occur in the program. That is, the name, class (constant, variable, label, etc.), and type (integer, floating point, etc.) of the data items along with a record of how and when they are used must be kept. If interface analysis is to be done, it would be advantageous to organize the global data information (such as common blocks and parameters in FORTRAN) into separate tables for a simplified information retrieval process. Other information coud be stored in these tables as needed for specific demands. For instance, detailed representations of output lists and format specifications would be needed if the output analysis is to be performed. Also, a record of statement types and corresponding machine code information would be needed for sizing or execution time estimations.

It is necessary to extract structural information from the source program (branch points, transfer points, entries, exits) if any type of path analysis is to be done. The graph structure of the program can be stored in the form of a list of transition pairs (in which pairs of statements are matched indicating that a direct path exists from the first to the second) or in some other form from which the graphic structure can be derived. This information can be used for usage flow analysis, flow of control analysis,

execution time prediction, and generation of test data, as described in the previous section. The global structure of the program, such as transfer points between routines, would be needed for the interface analysis; and an abbreviated graph of the entire program structure, in which sets of nodes could be coalesced into larger nodes, would be required for the modularity analysis.

Besides gathering and storing the required information, the pre-processor is also responsible for inserting code into the user's source program if any of the dynamic analysis methods are going to be used. For example, if a dynamic variable trace is to be used, then the pre-processor might insert a "trace line" into the source code every time the value of the variable is changed. This "trace line" might be simply a print statement to display the value of the variable or it could be a call to a run-time routine that would check the value of the variable against pre-specified bounds. The pre-processor might also insert "trace lines" at branch points in the program so that the frequency of execution of different nodes of the program could be monitored at run-time. These "trace lines" would only be temporarily added to the user's program in order to facilitate the collection of information during execution.

Module 2: Static Analysis

After the user's source code has been scanned and

the appropriate information stored, each static analysis processor may organize and analyze the stored information and print a record for the user of the requested analytical information. The validation processor might incorporate path analysis capabilities to trace the possible ways that a particular variable could affect other variables or trace possible paths to a certain statement in the program. An arithmetic processor might also be needed to test subscript bounds or to generate an optimal test data set. It could also include a routine to check the initialization of all variables used in the program, making sure that they are in some way initialized before being used. Since this stage of the validator would be used as an aid to debugging, it would be advantageous to develop it on an interactive system. The user could then request information periodically during the validation process.

If a processor for structural and timing analysis is used, it would correlate the graph structure of the program (as stored by the pre-processor) with information about the machine code to which the source code will be converted and would estimate the size and time requirements for the program. It could also give the user information for developing an optimum overlay structure.

A processor to be used for documentation could include such tasks as organizing interface information and printing it in readable form. For instance, each global variable and its corresponding size, type, and use could be documented, and

transfer points between routines could be noted. A flow-chart of the program structure might also be produced. The basic information needed for these tasks would have already been collected and stored by the pre-processor. The documentation processor would simply re-organize the information to be displayed in a form that could be easily understood by the programmer.

A processor might also be developed for use in performance measurement and maintenance. Its task would be to do such things as point out inefficient code or check for non-standard code constructs. It might also find sections of code that must be changed for new specifications or for a changing environment and either point them out to the user or automatically replace them. The analyzer might then gather information about the updated version of the program to aid the user in determining the effects of the change.

If no dynamic analysis methods are to be used, then the language analyzer is composed solely of the pre-processor and the static analyzers. That is, the source code is scanned while information is stored; and this information is organized, analyzed, and printed according to specifications.

Module 3: Dynamic Monitor

The dynamic monitor portion of the analyzer is incorporated into the actual execution of the user's program.

It is at this stage that the "trace lines" inserted during the pre-processing stage will be executed. These "trace lines" can be used to call routines that have been added to the user's program by the analyzer to gather information during execution. For example, statistics may be collected concerning the frequency of execution of different nodes in the program or variable values may be compared with prespecified bounds. The routines that have been added to the user's program either print information as it is found or store the information on file to be accessed by the post-processor.

Module 4: Post-processor of Dynamic Analysis

This portion of the analyzer acts in the same manner as the static processors except that it organizes and analyzes the information collected by the dynamic monitors during execution of the user's program. The information is printed according to the user's specifications.

CONCLUSION

There is a great need to decrease the amount of time spent in debugging, testing, documenting, and maintaining computer software systems. Automatic program analyzers present a viable answer for improving the slow and inefficient methods of program validation and maintenance that exist today. An automatic analysis system can be developed to incorporate the following four basic functions:

(a) program validation (debugging and testing)

(b) structural and timing analysis

(c) program documentation

(d) performance evaluation and maintenance.

These tasks can all be developed from a general-purpose data base containing information about the source program. Both static and dynamic methods can be included in the analysis to broaden the capabilities of the automatic aid.

A proposal for the basic design of a general-purpose program analysis system has been suggested in which a pre-processor would collect information about a program and construct a large data base containing this information. The pre-processor could also insert "trace lines" in the source program in order to perform dynamic analysis during execution time. A post-processor would be designed to extract information from the data base and analyze it in such a way as to aid the programmer in validating, documenting, or in some other way analyzing his program.

Some effort has already been directed toward the development of program analysis systems, but most of the systems have been limited to performing a few specific functions. The remainder of this thesis will discuss the work that has been done and will describe in detail a specific FORTRAN analysis system.

II. PREVIOUS WORK IN PROGRAM ANALYSIS

In the past ten years the need for program analysis has been recognized, and a variety of systems have been developed to aid the programmer in analyzing programs. The interest began with early debugging aids which had limited capabilities and has recently been expanded in the development of large analysis systems that have a wide range of functions. It is the purpose of this chapter to discuss some of the previous accomplishments in the area of program analysis.

PROVING PROGRAMS CORRECT

A considerable amount of work has been done in an attempt to develop methods for "proving" programs correct [2]. The ultimate goal of this effort is certainly worthy of praise, since the dream of every programmer is to be able to know for sure that his program performs properly. There are problems however in attempting to prove programs correct. At the present, workers in this area have only succeeded in proving the correctness of small and somewhat trivial programs.

Two approaches, termed formal and informal, have been taken in proving programs correct. In the formal method, proof of correctness is transformed into the proof of a theorem in the first order predicate calculus. This method

is cumbersome because of the difficulty in expressing program characteristics in such low-level terms as the first order predicate calculus. Once expressed, the automatic proof of a large theorem becomes difficult because of time and memory limitations. In the informal method, the programmer must provide assertions about the state of the program at different points in the code, and a proof of the consistencies of these assertions is then attempted. This method is limited by the human intervention and tedious proving procedures it requires.

It seems evident that, despite the noble goals of these efforts in proving programs correct, the methods at present are of little practical value to the programmer. Hopefully, in the future when the state of the art advances, programmers will have access to program-proving systems. In the meantime, study and implementation of the more practical methods of program analysis seems justified. The program analyzer makes no attempt to "prove" that a program is correct, but rather it furnishes the user with information that will enable him to better judge the program's performance. The focus of this thesis will be on the practical approach of program analysis.

EARLY DEBUGGING SYSTEMS

The birth of program analysis began with the development of early debugging systems whose main purpose was to help

users find errors in their programs. Three of these systems, BUGTRAN, DEBUG, and AIDS, all used in analyzing FORTRAN programs, will be discussed in this section.

BUGTRAN

One of the first debugging systems for a high-level language was made available as early as 1963. The package, called BUGTRAN, was developed at the University of California at San Diego for use in debugging FORTRAN programs [3]. The system requires the user to insert cards of the proper BUGTRAN syntax before the FORTRAN source deck to serve as selective commands to the BUGTRAN analyzer. The system has the capabilities to trace variable values between certain statements, trace flow of control, ititiate dumps, check entries to subroutines, and terminate control of the program under certain conditions. These actions are made possible by modification of the user's source code to insert monitors for tracing purposes and add code for dump and terminate conditions.

The BUGTRAN package, although limited in scope, was useful for its specific purposes and was an innovative idea in its time.

DEBUG

A similar debugging aid called DEBUG was written at United Aircraft Research Laboratories for the UNIVAC 1108

computer system [13]. In order to use DEBUG, a programmer submits requests within his program by inserting special comment cards in the deck. The DEBUG system pre-compiles the program and adds new FORTRAN statements to the code to incorporate the necessary requests. The program is compiled and executed with the added code, and a report is listed for the user according to his requests.

The following two functions are performed by the DEBUG system:

(1) Program variable values are automatically printed as requested. (Calls to an external routine are inserted at pre-compile time which at execution time will print the value of the appropriate variable at that point in the program.)

(2) Logic flow at certain points in the program is traced as requested. (The user inserts trace cards surrounding the code to be examined. At pre-compile time, DEBUG replaces all logic decision statements within that portion of the program with equivalent code that will also print the value of the decision expression.)

AIDS

Another early debugging system for FORTRAN programs was developed at New York University for the CDC 6600 [4].

The package is called AIDS (All-purpose Interactive Debugging System). Although its functions are similar to those of BUGTRAN and DEBUG, its methods are different. AIDS requires the following three inputs:

(1) object code of user's program

(2) listing generated by compilation

(3) "debug file" containing user's commands to AIDS.

Identifier addresses and other information about the program are stored, and commands from the "debug file" are decoded and stored in tables. The program is then simulated with additional monitoring and tracing capabilities specified in the commands to AIDS.

In order to use the system, a programmer must learn a special debug language. This has proved to be a drawback since it requires a large amount of effort from the programmer. The computer time necessary to carry out the simulation is also a disadvantage of this system. AIDS has been a worthwhile debugging aid for those programmers who have taken the time to become familiar with its capabilities.

AIDS, DEBUG, and BUGTRAN are three examples of the early program analysis systems that have been developed. In all three systems the emphasis is on debugging programs only. Tracing facilities and other mechanisms for extracting information about specific problem areas in a program are included to help the users find errors in their programs. The three systems served a useful purpose for what they were

trying to do, but their capabilities are limited when compared with the wide variety of other functions that program analyzers may accomplish.

LATER PROGRAM ANALYSIS EFFORTS

In recent years program analyzers have been developed with broader goals and capabilities than the original debugging systems. More emphasis is being placed on using program analyzers for a wider range of duties than simply to aid a programmer in finding errors in pre-specified areas of the program. These duties include such tasks as aiding program optimization, documentation, maintenance, timing and structural analysis, and performance evaluation. A description of some of these newer analysis systems will be discussed in this section.

FLOW

A software verification tool called FLOW was developed at TRW Systems as the first stage of the Product Assurance Confidence Evaluator (PACE) [1]. FLOW is a dynamic monitoring system designed to support test evaluation activities for FORTRAN programs. The system scans the user's source code, inserting additional code throughout the program. When the program is re-compiled and executed, the additional code will monitor the flow along paths in the program, gathering statistics along the way. This path trace may be done

at optional levels of detail, depending upon requests from the user.

After execution of the instrumented source program, FLOW will output the following information:

(1) number of executions of each statement

(2) per cent of total executable statements exercised

(3) per cent of total number of subroutines executed

(4) names of subroutines not executed

(5) total execution time in each routine.

The FLOW output helps the user in the following four areas:

(1) evaluating test effectiveness

(2) recognizing inefficient subroutine structure

(3) finding areas where code is never used

(4) optimizing overused sections of code.

For example, the programmer may discover from the FLOW output that a certain subroutine is never used. He then can devise a test in which the subroutine would be needed, or he might find that the subroutine is extraneous. The user might also be able to optimize code in areas of the program that are being executed often, thereby reducing the execution time of his program in an effective way.

FLOW is an example of a program analyzer that has been designed for purposes other than finding errors in specific areas of a program. It is not intended to be a general-purpose analyzer, but instead is limited to the function of

aiding in testing procedures. FLOW serves well the purpose for which it was intended and acts as a useful subsystem in conjunction with the "optimal software test planning system" to be described in the next section.

Optimal Software Test Planning

An innovative system for designing software tests has been developed at TRW Systems for NASA Johnson Space Center [7]. The system is an automated method for designing an optimal set of test cases which will exercise all branches of a FORTRAN source program. It's authors, Krause, Smith, and Goodwin, emphasize the impracticality of testing all possible paths of a program--a method which in theory would be the optimum way to insure that a program is sufficiently tested. They propose instead that it is better to design a set of tests that will exercise all branch options in the program. This method will insure more thorough testing than simply executing all statements but will not be as exhaustive as attempting to execute all possible paths.

The following two definitions are given by Krause, Smith, and Goodwin [7] in order to describe the proposed test planning method:

> "segment of code--smallest set of consecutively executable statements to which control can be transferred during program execution. The first statement will be directly accessible from another segment

and the last will be a transfer to a new segment."

"segment relationship--the relationship between two segments of code resulting from the transfer of control of execution from the first segment to the second."

The objective as stated by Krause, Smith, and Goodwin is to "find the minimum set of paths which exercise all the segment relationships in any subject module."

The automated system analyzes the FORTRAN source code and forms a table of information necessary for the generation of optimal test paths. This information includes segment transfer structure and variables involved in the path control. A problem arises in generating paths since all branch options are not mutually independent. "Impossible pairs" of segment relationships exist that must be extracted from the possible paths. Some of these "impossible pairs" are detected automatically, but others must be found by the user.

After all of the previously described information is collected, all base paths and loops are automatically generated. ("A base path is defined as a concatenation of segment relationships which begins at an entry point and ends at an exit point of the subject module and contains no repeated segments. A loop is defined as a concatenation of segment relationships which begins and ends at a repeated segment and contains no other repeated segments." [7]). Then optimal paths are formed combining a base path with loops that are

accessible from it. Each path is displayed to the user who must check to see if any impossible pairs exist. If not, he must then determine the proper data to execute the path. (All variables affecting the path are automatically displayed to him.) After test execution, the segment relationships executed in that test are input to the automated tool and flagged. The next path given to the user will be chosen in order to execute a maximum number of segment relationships not executed by the previous test. The process continues until all segment relationships have been exercised.

The automated tool described by Krause, Smith, and Goodwin is a useful aid in planning optimum testing activities. Although it does not guarantee sufficient program testing (since all paths are not executed), it proposes a more practical solution of generating paths that will exercise all branches in the program. This seems to be a major improvement over the method of insuring simply that all statements are executed. A weakness in the system lies in the large amount of work the user must contribute by determining the input that will cause execution of a prescribed path. This task could be simplified if the user were automatically given the relationships that all input variables should form toward each other. (Eg., $0 < A < 10$, $B > A$, $C = B + 5$, etc.) Of course, this would greatly increase the complexity of the automatic analysis.

PET

An automatic program testing tool called PET (Program Evaluator and Testor) was developed by McDonnell Douglas Astronautics Company. L. G. Stucki describes PET as a system for automatically generating "self-metric" software. [11,12] That is, the system supplements the source program with code that, when executed, will gather statistics and syntactic information about the program. The program itself then becomes "self-measuring" and enables the programmer to better evaluate it.

PET is designed to perform analysis on FORTRAN programs. The system first scans the FORTRAN source code, inserting statements to be used in gathering statistics. Static information about the execution is saved. The PET post-processor analyzes the static information and the results of the execution and generates a report for the user. The following information is displayed in the report:

(1) Syntactic Profile
(number of executable, nonexecutable, and comment statements; number of CALL statements; total program branches)

(2) Program Performance Statistics
(number of statements executed; number of branches and CALLS taken; associated with each executable statement--execution counts, branch counts on IF's and GO TO's, minimum/maximum

data range values on assignment statements and DO-loop control variables)

PET has proven to be an effective tool in evaluating FORTRAN programs. For instance, one program that was believed to be "thoroughly tested" was found to be inefficient when PET helped to recognize that the program spent one fourth of total execution time in an inefficient DO-loop. [11,12] The analyzer has also been helpful in finding unreachable code in programs. The development of the PET system is an example of an effort to broaden the capabilities of a language analyzer. Although it cannot be termed "general-purpose" since it focuses mainly on dynamic analysis, it does at least include some static analysis in the "syntactic profile." The usefulness of counting comment statements, executable statements, etc. seems questionable, but at least the basis for static analysis is incorporated into the system.

FETE

Another existing FORTRAN language analyzer, similar to those previously described, is called FETE (FORTRAN Execution Time Estimator). [6] The purpose of FETE is to aid in optimizing program code in order to minimize execution time. The method is similar to those used in previously described systems. That is, counters are inserted into the source code, and the modified program is then executed. In

FETE, the modified program is then re-read and correlated with the final counter values obtained in execution. The output lists each statement and the number of times it was executed. It also shows the number of times the TRUE branch was taken in logical IF statements and shows the "cost" (relative to execution time) of each statement. The "cost" is based on such things as the number of operators, the number of parentheses, and the statement type.

In using FETE, Ingalls found that most programs spend a large percentage of total execution time in a small percentage of statements. [6] By pointing out this small percentage of statements, FETE can encourage the programmer to optimize his program code in the areas where it really matters. FETE could also be used to aid in evaluating test runs for completeness, although it was not specifically designed for that purpose. Although limited in scope, FETE serves as a useful tool in optimizing programs and develops a new idea in its "cost analysis" of each statement.

ACES

The Automated Code Evaluation System (ACES) is one of the first attempts at providing a general-purpose analyzer. It was developed at the University of Texas at Austin for the Safeguard System Evaluation Agency.[8,9] ACES analyzes an intermediate-level language called CENTRAN which is used specifically for Safeguard System Software. Both static and dynamic methods of analysis are used. The

main purpose of ACES is to create an organized data base containing information about a program. The programmer is then relieved of the task of searching through large amounts of code in validating a program. The system was designed to be used on large programs in which correct performance is critical.

The static analysis portion of ACES involves a scan of the CENTRAN source code and the formation of a set of tables with information about program variables, labels, statement types, and structure. During the lexical scan, a set of messages are collected which are printed to warn the user about possibly dangerous constructs. (The CENTRAN language is in particular need of this type of analysis because it gives the programmer freedom to use several undependable language constructs that may cause critical errors if not used wisely.) The tables may be printed if the user wants access to information about a program that would normally require a tedious search of the source code. The programmer may want to know such things as "in what statement does variable A appear?". This information could be found in the tables.

ACES also includes a dynamic monitoring feature in which the programmer may trace the value of certain pre-specified variables. At each point in the program when the value of a specified variable may be changed, ACES inserts a call to a CENTRAN routine which at run-time will check the value of the variable against pre-specified bounds.

If its value does not fall within the specified range, a warning message is printed.

The ACES system serves as a useful background for developing broadbased analysis systems, but the immediate practicality of ACES is questionable. The data base that is generated contains a wide variety of information that can be useful in many stages of program analysis. The system does not, however, use this information to its fullest advantage since the tables are simply printed for the user to decipher. Answers to questions about the program that the user must now find by examining the tables could be done automatically. The ACES system has served as a useful guide though to program analysis, particularly in the development of the FACES system to be described in the next chapter.

CONCLUSION

This chapter has briefly described some of the automatic program analysis systems that have been developed in the past. Most systems so far have attempted to aid the programmer in some specific area of program validation. Many of the systems deal with program testing procedures and include dynamic monitoring. Few of them give the user any helpful information about the program before it is in the execute mode. ACES attempts to do this but produces results that are somewhat impractical.

Another automatic analyzer, called FACES, has been

recently developed which uses information learned in the development of ACES as a background. FACES extends the ideas of ACES and attempts to develop a general-purpose data base from which extensive information about a program can be drawn to aid in many aspects of software development and maintenance. FACES will be the topic of the next chapter.

III. A FORTRAN AUTOMATIC CODE EVALUATION SYSTEM

The present chapter will describe the FORTRAN Automatic Code Evaluation System (called FACES) that was developed for the National Aeronautics and Space Administration, Marshal Space Flight Center, during 1972 and 1973. The basic goals of FACES will be discussed, and its operations on FORTRAN programs will be described. The chapter will include a discussion of the methods used in implementing the analysis procedure and suggestions for possible extensions to the system.

PURPOSES OF FACES

FACES was designed to serve as an automatic aid in analyzing FORTRAN programs. Because the system was designed to be used on programs that would be applied in crucial situations in which accuracy is of utmost importance, the system emphasizes aid in finding and correcting semantic and logical errors in FORTRAN programs. (Programs are assumed to be free of syntactic errors before being analyzed by FACES.) An effort is made to critically analyze program constructs and to point out those that could possibly hinder correct program execution. Operations are also provided that will aid the programmer in tracing the cause of errors once they have been found.

The application of FACES is intended mainly for use

with large FORTRAN programs. For this reason emphasis is placed on automatically assimilating and organizing information about the program. An attempt is made to condense and display the information to the user in a concise form. It is hoped that this aspect of the analysis will relieve the user of the tedious task of searching through pages of FORTRAN code when examining a program. Instead, the programmer can request that the information he needs be automatically gathered and displayed.

Although FACES was designed with an emphasis on validation, it was constructed in a manner that will allow it to be expanded to perform other services as well. In this analysis system, a large data base containing information about a program is collected. Information from the data base can then be used for a variety of purposes. Additions to the present validation features could be made and extensions in other areas of analysis, such as documention, optimization, performance evaluation, maintenance, and structural and timing analysis, could be implemented. Some would require substantial changes to the FACES system while others would require only minor additions.

In brief, the design of FACES aspires to the following three goals:

1. to aid in finding and correcting semantic errors,
2. to assimilate and organize information about large programs, and

3. to build a data base from which information can be collected for a variety of purposes.

TASKS PERFORMED BY FACES

The tasks performed by FACES are divided into two main sets. First, information about the FORTRAN source code is gathered and stored. This information serves as the data base mentioned in the previous section. Then the information stored in the data base is queried, at the user's request, in an effort to find program constructs that could possibly be hazardous or to organize and display information needed by the user in searching for the cause of an error.

Data Base Collection

The data base collecting part of FACES (called the FORTRAN Front End) gathers two basic types of information--global and local. The global information includes items that relate to the entire program, whereas the local information applies to each routine independently. (See Figure 1.) For example, the global information includes a list of all subroutine names along with locations in other routines from which the subroutines are called. Common block structures are stored in the global information set along with lists of the elements within each common block. Argument transmission information, including all actual and formal parameter lists, is also stored with type and size information noted

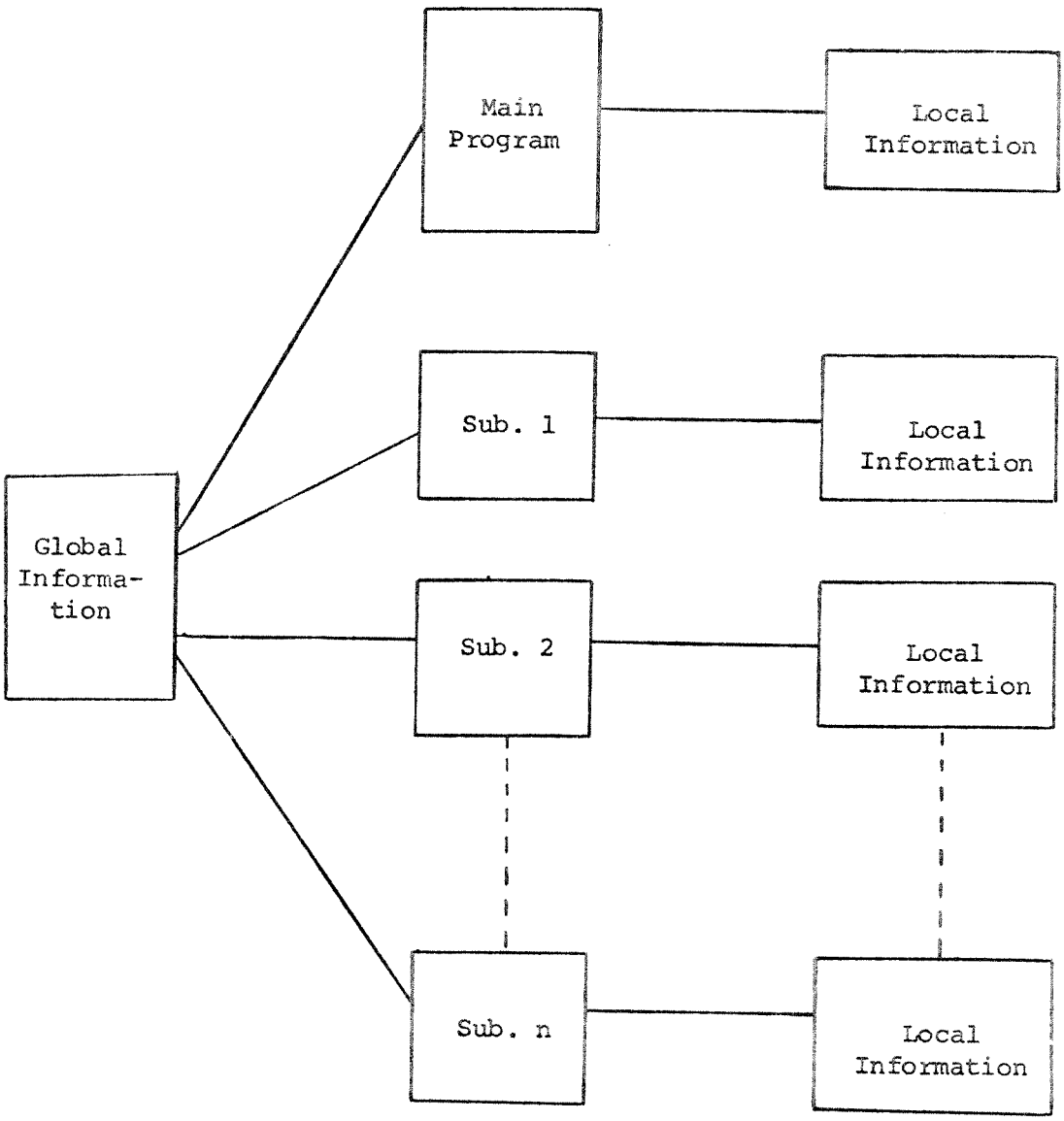


Figure 1: Structure of Local and Global Information

for each parameter.

A unique set of local information is collected for the main FORTRAN routine and for each subprogram in the whole program. Each set contains a table of all symbols used in the routine (e. g. variables, labels, common block names, subroutine and function names, etc.). For each name a code is attached to indicate what the name is (simple variable, array, common block name, etc.) and the type of the name (integer, floating point, etc.). For each occurrence of a name in a routine, a note is made to indicate how the name is used in that particular occurrence. For example, a variable could be used as input to an assignment statement, as B is used in the statement

```
A = B * 2.
```

The variable could also be used as an actual parameter in a subroutine call, as B is used in the statement

```
CALL SUB(B).
```

For another example, labels could be used in reference, as label 25 is used in the statement

```
GO TO 25.
```

Labels could also be used in definition, as label 25 is used in the statement

```
25    X = Y.
```

The occurrences of each name in a routine are linked in a way such that, given a particular occurrence of a name in the routine, the preceding occurrence (in order of appearance in the program) and the succeeding occurrence can easily

be found.

In addition, the local information about a routine includes the storage of all array names in the routine with the dimensions of each. Information about each DO-loop in the routine is also stored, including the ending label, index, initial value, increment, and terminal value. Information about structure of each routine is stored by noting the immediate predecessors and immediate successors of each statement (logically rather than in order of appearance). From this information, a complete graph of the routine can be derived.

The global and local information collected by the FORTRAN Front End is stored in a set of tables that may be displayed to the user or stored on file for later reference. The tables are organized in a manner such that a set of local tables can be accessed independently to allow the study of a single module; or information about the entire program may be examined. It is hoped that the information is organized in such a way to allow flexibility in its use.

Data Base Query

The second major part of the FACES system contains a set of routines that query the data base formed by the FORTRAN Front End. The routines are designed to be called independently by the user, according to his needs. Each routine extracts from the data base a certain type of infor-

mation to help the user in analyzing a program. The four following diagnostic routines have been implemented in the FACES system:

1. Common block alignment routine
2. Parameter alignment routine
3. Variable initialization check
4. Variable trace routine

Each will be discussed further in the succeeding paragraphs.

The common block alignment routine is used to verify that all common blocks in a program are aligned. The routine checks to see that all common blocks of the same name have the same number of entries, and that corresponding elements within each block have identical dimensions and are of the same type. An option allows the user to also verify that corresponding elements of each block have identical names. Although FORTRAN allows a programmer to construct common blocks with elements of differing dimensions and names, these practices are considered dangerous by some (especially if not intended). Thus FACES allows the user to verify that the above restrictions are met throughout a program.

The parameter alignment routine is used to verify the alignment of all parameters in subroutine and function calls. The routine checks all actual parameter lists in subroutine and function calls to see that they correspond with the formal parameter lists described in the subroutine or function definitions. To meet the standards, corresponding parameter lists must have the same number of para-

meters; and corresponding parameters within each list must be of the same type. The user also has the option to check that arrays in each parameter list have the same dimensions. If any of these conditions are not met, warning messages are issued.

The third diagnostic routine verifies that all variables within a routine are in some way initialized before being used in a way that assumes a value is already attached to the variable. Variables which appear in a common block, in a DATA statement, or as entry parameters are assumed to be initialized. For each of the remaining variables a trace is performed along each entry path to check that the variable is initialized before its value if used.

The fourth diagnostic routine (called a variable trace) may be used to trace the history or future of particular variables within subprogram bounds. That is, given a certain variable at a certain point in the program, the routine will list all variables that could possibly affect the value of that variable previous to the specified statement. This is called a history trace. In the future trace, the routine will list all variables that could possibly be affected by the value of the specific variable after a certain point in the program. This routine is particularly useful in helping a programmer to find the cause of an error in a program. For example, the programmer may know that in a specific statement the value of variable A is incorrect. He then could run a history trace to find all variables that might

have caused A to obtain the incorrect value. The programmer might also want to know what repercussions the incorrect value of A will cause in the remainder of the program. For this purpose, he could use the future trace to find all variables that might be affected by A. (See Figure 2 for further explanation.)

The routines described above aid the user both in finding errors and in correcting them. The common block alignment routine, parameter alignment routine, and variable initialization routine serve mainly to find program constructs that could possibly lead to error. The variable trace routine helps the programmer in finding the cause of errors by automatically doing the tedious job of tracing through the code. These four routines, however, are not a final list of tasks that can be performed by analyzing the data base. Possibilities for further extentions will be discussed in a later section.

METHODS USED BY FACES

The FORTRAN Automatic Code Evaluation System is composed of two distinct modules--the FORTRAN Front End and the Data Base Query (diagnostic routines). As mentioned earlier, the FORTRAN Front End collects information about the FORTRAN program being analyzed, and the Data Base Query analyzes this information, looking for particular constructs. The FORTRAN Front End and the Data Base Query will be discussed separately

```
      SUBROUTINE SUB1(N,B,C)
      F = 0
      K = N
      L = B * C
      IF (N.EQ.0) 10,20
10    A = K
      GO TO 25
20    A = L
25    PRINT 2, A
2     FORMAT(I10)
      IF(A.GT.0) F = A
      D = A / B
      DX = F - D
      RETURN
      END
```

Results of history trace on A at statement 25:

"The value of A could have been affected by

L, K, B, C, N."

Results of future trace on A from statement 25:

"The value of A could affect the future values of

F, D, DX."

Figure 2: Example of Results from Variable Trace Routine

in this section. (More detailed information about FACES can be found in the original documentation. [5])

FORTRAN Front End

The Front End is composed of three interacting modules--the lexical scanner, the parser, and the table generator. The lexical scanner reads each FORTRAN statement on a character-by-character level, grouping the character strings into larger entities to be examined by the parser. The parser determines the statement type of each statement, picks out valuable portions of the statement, and sends information to the table generator to be stored. The table generator handles all table manipulation upon commands from the parser. The parser is the main routine of the Front End and calls the scanner and table generator into action when necessary. (See Figure 3.)

Lexical Scanner

The scanning routine of the FORTRAN Front End reads a statement of the source program and recognizes alphanumeric strings which appear to represent FORTRAN variables, constants, operators, and labels. It then passes to the parser an intermediate symbol string (ISS) and a temporary symbol table (TSTAB). The intermediate symbol string is a list of codes corresponding to lexical entities, and the temporary symbol table contains the actual characters for each alphanumeric

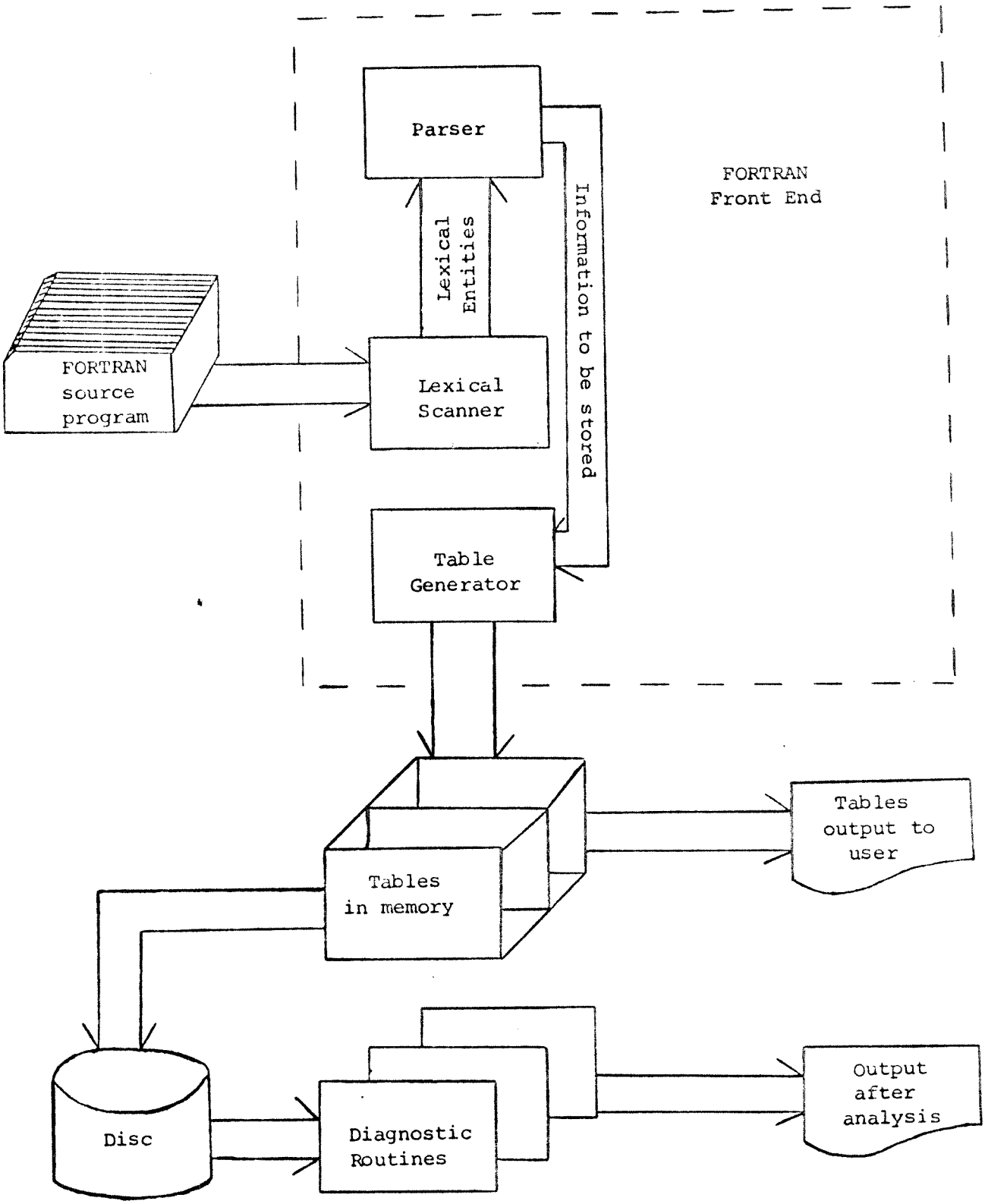


Figure 3: Overview of FACES

string that could possibly be a variable. Some typical entities that are recognized by the scanner are: logical operator--code L, floating point constant--code F, relational operator--code R, and "possible variable"--code V. For example, in the statement

```
IF(ALPHA.EQ.BETA)X = 2.5
```

the scanner would form the intermediate symbol string and temporary symbol table as below.

```
ISS (Intermediate Symbol String) = "V ( V R V ) V = F"

TSTAB (Temporary Symbol Table) = IF
                                 ALPHA
                                 BETA
                                 X
```

Information such as that above is formed and given to the parser for each statement in the FORTRAN source program.

Parser

The parser is the main controlling module of the FORTRAN Front End and handles the parsing of each statement in the FORTRAN source program. After receiving information about a statement from the scanner (in the intermediate symbol string and the temporary symbol table), the parser determines the statement type by the following algorithm:

1. Is there a zero-level equals sign? (an equals sign that is not within parentheses) If no, go to 6; if yes, go to 2.
2. Is there a zero-level comma? (A comma that is

not within parentheses) If no, go to 4; if yes, go to 3.

3. This is a DO statement. Statement recognition completed.

4. Does the statement have the form

   IF (⟨expression⟩) ⟨assignment statement⟩ ?

   If no, go to 5. If yes, this is an IF statement and statement recognition is completed.

5. This is an assignment statement. Statement recognition completed.

6. Determine the statement type by matching the first four characters of the statement with a list of special FORTRAN words (such as READ, PRINT, COMMON, etc.) Statement recognition completed.

After the parser has recognized the type of a statement, it calls a routine designed to analyze statements of that type which will pick out important information and call the table generator to store the information. (See next section for description of table generator.)

Program transfers are also recognized by the parser and stored in a transition table in the form of non-normal transition pairs. (A transition pair is a pair of statements having a direct transfer from the first to the second. A non-normal transition is any transition other than the normal fall-through case.) After an entire routine of the source program has been parsed, a table is formed from the