AD/A-002 322

CRITERIA FOR EVALUATING THE PERFORMANCE OF COMPILERS

SofTech, Incorporated

Prepared for

Rome Air Development Center

October 1974

This report has been reviewed by the Office of Information, RADC, and approved for release to the National Technical Information Service (NTIS)

This report has been reviewed and is approved:

APPROVED: *Douglas A. White*

DOUGLAS A. WHITE
Project Engineer
Software Sciences Section

APPROVED: *Robert D. Krutz*

ROBERT D. KRUTZ, Col USAF
Chief, Information Sciences Division

FOR THE COMMANDER: *James G. McGinnis*

JAMES G. McGINNIS, Lt Col, USAF
Deputy Chief, Plans Office

Do not return this copy. Retain or destroy.

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-74-259	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER *ADjA-ØØ2,32,2,*
4. TITLE *(and Subtitle)* Criteria for Evaluating the Performance of Compilers		5. TYPE OF REPORT/& PERIOD COVERED Final Report June 73 – June 74
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) Burton H. Bloom Clare G. Feldman Mac H. Clark Robert K. Coe		8. CONTRACT OR GRANT NUMBER(s) F30602-73-C-0321
9. PERFORMING ORGANIZATION NAME AND ADDRESS SofTech, Inc. 460 Totten Pond Road Waltham MA 02154		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62702F Job Order No. 55811206
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIS) Griffiss AFB, NY 13441		12. REPORT DATE October 1974
		13. NUMBER OF PAGES 348
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)* Same		15. SECURITY CLASS. *(of this report)* Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N//

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

DDC
RECEIVED
DEC 23 1974
D

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Douglas A. White (ISIS)

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Compiler Performance	Compiler Gibson Mix	JOVIAL J3B
Compiler Comparison	User Profile	
Compiler Evaluation	Compiler Profile	
Compiler Gibson Mix	Compiler Demand Profile	**PRICES SUBJECT TO CHANGE**
User Profile	AED	

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The main purpose of this study was to develop criteria by which it will be possible to qualitatively measure and evaluate the performance of compilers, possibly operating on different computers, and possibly having different features. To satisfy this purpose, three technical questions were studied: (1) How can two compilers with the same features and operating in the same environment be compared? (2) If two compilers with the same features operate in different environments, how can their measured differences in performance be attributed to the environmental differences vs. the compiler differences? (3) How

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

20.

should a compiler buyer deal with the problem of evaluating compilers with different special features? These three questions were studied from a point of view that the answers should help provide a basis for conducting dollar cost/benefit analysis of compilers. In addition, a fourth question was studied to satisfy a specific secondary purpose of the study: Can analysis of a compiler's architecture and algorithms provide a basis for making valid judgments about the performance that should be expected from a compiler?

The general conclusion from studying the fourth question was that analysis of the internal organization of a compiler was not useful in providing a basis for compiler evaluation. The study of the first question established criteria and methods for assigning four measures (time and space for both compiler and object code) to a compiler which quantitatively define its performance with respect to a "typical" user program. The study of the second question established criteria for defining a "compiler Gibson mix", and established methods for using this "mix" to "equalize" environments. The study of the third question established contractual methods for providing a cost component of cost/benefit analysis of special features; also, several specific areas for future study were identified which would provide needed data for establishing a basis for the benefit side of such analyses.
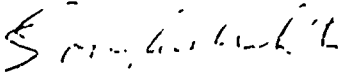
Technical Evaluation

1. This effort, Compiler Performance Criteria and Measurement Study, was undertaken to investigate methods of determining and evaluating the performance of compilers, that will be more objective, informative and reliable than methods currently in use.

2. At present, compiler performance measurements are taken in terms of cards per minute and machine instructions per source statement without any scientific basis for the contents of the card, the size, type or complexity of the statement, or the planned environment of the compiler.

3. This effort defines the methodology by which compiler performance may be objectively measured and compared to other compilers, including factors such as application and environment.

4. This effort establishes the criteria necessary for the evaluation of compilers that will insure that the compiler selected for a particular use is the most efficient compiler that will meet the requirements. The methods by which test programs are developed and the measured results analyzed are specified. Guide lines for data to be collected, and methods for collecting the data are presented for the compiler and user profiles involved. Methods developed in this effort will be valuable in future procurements of compilers in insuring cost effectiveness.

Douglas White
Project Engineer
Software Sciences Section

SUMMARY

The main purpose of this study was to develop criteria by which it will be possible to qualitatively measure and evaluate the performance of compilers, possibly operating on different computers, and possibly having different features. To satisfy this purpose, three technical questions were studied: (1) How can two compilers with the same features and operating in the same environment be compared? (2) If two compilers with the same features operate in different environments, how can their measured differences in performance be attributed to the environmental differences vs. the compiler differences? (3) How should a compiler buyer deal with the problem of evaluating compilers with different special features? These three questions were studied from a point of view that the answers should help provide a basis for conducting dollar cost/benefit analysis of compilers. In addition, a fourth question was studied to satisfy a specific secondary purpose of the study: Can analysis of a compiler's architecture and algorithms provide a basis for making valid judgements about the performance that should be expected from a compiler?

In studying the fourth question, fourteen functional elements of a compiler were identified and described. A one-pass architecture involving eight elements and a thirteen phase multi-pass architecture involving thirteen elements were developed to illustrate the extremes of architectural choice in compiler design. Four elements were studied in detail: table look-up, parsing, optimization, and code generation. The general conclusion reached from studying the fourth question was that analysis of a compiler's internal organization was _not_ useful in providing a basis for compiler evaluation.

The first question was studied in terms of establishing criteria for defining elements of a User Profile in terms of specific categories of the language on which a compiler operates. A User Profile is specified quantitatively as the fractions of a "typical" user program that are in the form of specific language elements. The same language elements can also be used to establish a Compiler Performance Profile, which describes

quantitately how a compiler performs with respect to each language element. This profile consists of (generally) four measures for each language element: time and space for both compiler and object code. Criteria were established for writing test programs for each language element to be used in making these four measurements. A method was developed for combining the User Profile and Compiler Performance Profile to produce a Compiler Evaluation Profile. This profile consists of the four quantitative measurements of a compiler's performance with respect to a "typical" user program.

The study of the second question established criteria for defining a "compiler Gibson mix". This "mix" defines how well a computer/operating system supports the activity of compiling. The basis for defining this "mix" is the establishing of a "typical" Compiler Demand Profile. This profile is similar to a User Profile, but describes a compiler as a user of the language in which the compiler is written. Two such profiles were established during the study: one for an AED compiler and one for a J3B compiler. These two profiles turned out to be very similar and provided the basis of an example of a "compiler Gibson mix" which was developed during the study. Methods were also established for using the "mix" to "equalize" environments as an answer to the second question.

The third question was studied only to a limited extent. A contractural basis for the cost side of cost/benefit analysis of special features was developed. Also, several specific areas were identified for future study which would provide data needed for establishing a basis for the benefit side of such analysis.

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont)

LIST OF ILLUSTRATIONS

8

LIST OF TABLES

CHAPTER 1
INTRODUCTION

1.   Background

Compiler performance is one of the most critical aspects in today's software development process, and the admitted lack of scientifically based, useful evaluation criteria poses frustrating problems for buyers and sellers alike.   For the Air Force, the value of having standard and fair techniques for evaluating compilers is clear:

- It will help to insure selection of compilers for Air Force use that really meet Air Force needs.

- It will help to insure that when buying a new compiler, the Air Force will get value for its money, by providing solid acceptance test criteria.

- It will help to prevent and resolve disagreements about the performance quality of a particular compiler by highlighting the reasons for disagreement.

There are countless opportunities for consciously and unconsciously biasing an evaluation of compiler performance. It is no wonder that different evaluators reach widely varying conclusions. What constitutes a fair measurement of a compiler's performance? What are the sources of bias, and how can they be eliminated?

These questions provide the point of departure for the study described in this report. At present, compiler efficiency measurements are commonly expressed in terms of cards per minute and machine instructions per source statement. These expressions at best are inaccurate. When confronted with a measurement expressed in cards per minute one must determine what is included in this minute, what clock is used, if I/O and operating system time is included, how many statements are on a card, and the complexity of each statement. Similarly, when a measurement is expressed in terms of machine instructions produced per source statement, it is necessary to ask how complex the statement was, what machine instructions were necessary

10

to perform the function of the source statement, and how many times each instruction is executed. The inaccuracies caused by expressing performance in these terms has made possible widely differing claims of performance, or the lack of performance, by different individuals for the same compiler. Using the criteria of cards per minute and instructions per statement to measure performance, it is possible for the producer of a compiler to back up very optimistic performance claims simply by properly arranging a test program.

As will be seen, this report presents a basis for defining a methodology for evaluating the performance of compilers, together with supporting data, which justifies the conclusion that the methodology will be significantly more objective, fair, informative, and reliable than methods currently in use.

2. Purpose of the Study

In considering the objectives to be sought in performing the present study, two distinct points of view were taken into account. The first point of view required the identification of specific technical objectives. These objectives were, for the most part, spelled out explicitly in the statement of work for the project. The second part view looked beyond the technical goals toward eventual application of the results of the study. These application objectives provided insights which were useful in establishing an appropriate operational organization of the study activities.

The result of the organizational effort was to seek a basis for answering a small number of specific, well defined, technical questions. Here the intent was to separate the varied aspects of the numerous technical problems to be explored into a few clearly defined questions. The combined answers to these technical questions would provide the basis for fulfilling the technical objectives.

The technical objectives and the application objectives are discussed below. The organization of the study into four specific technical questions is discussed briefly in Section 3, and in greater detail in Chapter 2.

<u>Technical Objectives</u>. The overall technical objective of this study was to develop criteria by which it will be possible to qualitatively measure and evaluate compiler performance. These measurements should make possible valid performance comparisons of different compilers on different machines. If the results of the measurements are weighted appropriately, then it should be possible to equalize the environment in which the compiler is being measured with other environments in which similar compilers might be measured. This "equalization" of environments should take into account factors such as memory size, processor speed, instruction set, and operating system support.

Once an "equalization" of environmental differences has been performed, it is desired that criteria be established for properly taking into account other factors such as the type of compiler being tested (e.g. production, debugging, etc.), characteristics of the machine and/ or operating system for which the compiler was designed, and other features of the compiler that might adversely effect the performance of the compiler while producing an overall savings to the user. Other factors to be taken into account are the effects on compiler performance caused by including in a compiler functions such as optimization and debugging aids. Here it is desirable to determine a method whereby the final performance value calculated for a compiler can include an appropriately weighted component so as to negate the compromises in compiler efficiency made necessary by including these functions.

Thus, the overall technical objective can be summarized as follows. It is desired to determine criteria by which the performance of a compiler can be measured and compared to that of other compilers. In particular, the criteria shall include factors such as speed, size, ease of use and maintenance, and efficiency of generated code. Furthermore, the result of using the criteria obtained in the study should reduce the range of values obtained when different individuals measure the performance of any given compiler.

12

A specific secondary technical objective was identified for this study. This objective was to determine whether or not knowledge of the architecture and algorithms used in a compiler might provide a basis for making valid judgements about the performance that should be expected from a compiler, independent of actually measuring performance by means of test runs, etc. For example, can a useful generalization be made about an algorithms to the effect that it is most efficient for the particular language and/or application for which it is used, or that the complexity of the algorithm is wrong for that of the language and/or application.

With respect to parsing schemes and table searching methods in particular, a specific technical objective was to determine:

- If there is a particular parsing scheme that is most efficient for all languages and user types, or is each language better suited by a unique parsing scheme.

- If there is a relationship between table searching methods and the type of language which is being compiled.

In addition to these two major functional elements of a compiler (parsing and table look-up), two other elements (code generation and optimization) were also reviewed with the objective of seeking useful generalizations. Also, an analysis of architectural choices in compiler design was made. With this context in mind, it is convenient to summarize briefly here the major conclusions resulting from this part of the study.

- With respect to compiler architectures, one-pass compilers are faster and larger than multi-pass compilers.

- Multi-pass compilers permit more extensive optimizations, and therefore can produce more efficient object code.

- Choices of algorithms for parsing and code generation are generally made for other than performance reasons. Generally, the reasons relate to cost of development of the compiler.

13

- Some generalizations on the relative efficiency of table look-up algorithms are possible, but such generalizations are mainly related to specific internal architectural purposes for the table rather than external factors such as the language being compiled.

- Optimization methods are highly varied, and no useful quantitative generalization was found which could relate compiler performance and object code quality for a particular individual or class of optimizations.

In view of the above paucity of useful generalizations that resulted from the study of architectures and algorithms, it is clear that a different approach is necessary to establish the desired criteria to satisfy the technical objectives of the proposal. The approach taken is discussed briefly in Section 3, and in greater detail in Chapter 2.

Application Objectives. Given that criteria could be determined for compiler evaluation as discussed above, what application might be made of these criteria? Answering this question provided insights that were useful in organizing the activities of the study. These activities are discussed briefly in Section 3 and in greater detail in Chapter 2. The answer to the question is summarized below.

The criteria to be established by the study will be useful for the following tasks:

- Selecting the best performing compiler from among a number of off-the-shelf compilers.

- Preparing RFP's for compilers and providing the basis for reliable performance acceptance testing of the delivered product.

- Provide useful assistance in choosing computer hardware and compiler combinations as a package.

- Provide useful assistance in choosing computer hardware where compilers are to be purchased separately.

With these application objectives in mind, it becomes clear that the criteria to be established to satisfy the technical objectives discussed above must (at least in principal) be translatable into a common unit of value. The obvious unit of value that comes to mind is the dollar.

14

Consequently, it became an identified objective of the study to establish criteria which could provide the basis of dollar cost/benefit analysis of a compiler. This means that measurements to be taken of compiler performance should ultimately be translatable into a dollar value for the performance benefit of the compiler. The approach taken in this study is specially directed toward this objective.

Implicit in the above observation is that a better basis for establishing the dollar cost of a compiler is needed. In choosing off-the-shelf compilers, its price tag is an adequate cost measure. However, in preparing RFP's, the vendor must be given in incentive to produce a better performing compiler than minimum specifications. These cost and incentives issues are beyond the scope of the present study. However, the criteria for evaluating performance might be a suitable basis for eventually establishing appropriate incentives in procuring compilers, once sufficient experience in their use has accumulated.

3. Technical Questions Studied

In this section a brief discussion is presented of the specific technical questions whose answers were sought as the basis for fulfilling the technical objectives of the study. The reasons for choosing these questions, and how these questions determined the specific activities of the study are discussed in Chapter 2.

Four technical questions were pursued. These questions are conveniently organized into two groups. The first group consists of the following single question which constitues the basis for fulfilling the specific secondary technical objective discussed in Section 2:

- Can analysis of a compiler's architecture and algorithms provide a basis for making valid judgements about the performance that should be expected from a compiler?

We will refer to this question briefly as the "architecture/algorithms question".

The second part consists of three questions which jointly provide the basis for fulfilling the overall technical question discussed in Section 2. These questions are listed below.

- How can two compilers with the same features and operating in the same environment be compared?

- If two compilers with the same features operate in different environments, how can their measured differences in performance be attributed to the environmental differences vs. the compiler differences?

- How should a compiler buyer deal with the problem of evaluating compilers with different special features?

These three questions will be briefly referred to respectively as:

- The "same environment question",
- The "environment equalizing question", and
- The "special features question".

It should be noted that the first question is directly applicable to the application objective of selecting among off-the-shelf compilers (provided that these compilers have similar features). Furthermore, the question specifically applies to the overall technical question once the environmental contributions have been "equalized" and appropriate weights for special features have been calculated.

The second question is specifically aimed at the application objective of assisting in hardware selection where compilers are to be procured separately. Furthermore, this question specifically applies to the overall technical question in that its answer provides the basis for "equalizing" environments.

The third question is specifically aimed at the technical objective of calculating appropriate weights for making compensations in overall performance values calculated for compilers with different special features.

4. Organization of the Final Report

Chapter 2 presents a detailed discussion of how the technical questions studied contribute to fulfilling technical and application objectives.

Chapters 3, 4, 5, 6, and 7 respectively present analyses of architectural choices in compilers, and algorithms used for table look-up, parsing, optimization, and code generation. These analyses comprise the report on the study of the architecture/algorithms question.

Chapters 8 and 9 provide a basis for answering the same environment question. Chapter 8 provides a detailed overview and technical framework, and Chapter 9 specifically establishes methods of preparing test programs which would establish useful compiler performance measures.

Chapters 10, 11, and 12 provide a basis for answering the environment equalizing question. Chapter 10 describes the methods that can be used to measure environments in terms of a "compiler Gibson mix". That is, standard tests are described which can be applied to different environments to determine their relative efficiency in supporting compiler activities, and thus providing "equalization" factors for the environments. Chapters 11 and 12 present experimental data which provide the basis of establishing a "compiler Gibson mix".

Chapter 13 discusses a basis for seeking an answer to the special feature question. The study of this question quickly led to the conclusion that a great deal of work beyond the scope of the present study would be required in order to establish criteria to answer this question in an adequate manner. Consequently, the study was limited to identifying specific areas of possible future study that would contribute to providing an adequate answer.

Chapter 14 summarizes the conclusions reached in the study. Conclusions are generally included in the separate chapters where appropriate, and Chapter 14 presents an organized list of these conclusions with appropriate cross references to other sections of the report.

17

Chapter 15 presents recommendations which appear appropriate in the light of the results of the study. Specifically, these recommendations identify areas requiring further study, and summarize the specific procedures constituting a standardized methodology for evaluating compilers in different environments with different special features which were developed in the study.

CHAPTER 2
OVERVIEW OF TECHNICAL QUESTIONS

1.   Introduction

Section 2 of this chapter introduces the basic technical concepts on which the study was based. The concepts are discussed briefly in Section 2 and in further detail in other sections of this chapter. It is convenient to organize the concepts into two broad catagories:

- Concepts related to the establishing of a canonical representation of users, environments, and compilers.
- Concepts related to catagorizing the factors which influence the performance of compilers in measurable terms.

The first category consists of a number of profiles and "Gibson mixes". Profiles represent a user or a compiler in terms of elements of a high level language, such as AED, or the source language on which a compiler to be evaluated performs its compilation function, or in terms of performance factors. A "Gibson mix" is a representation of a collection of programs in terms of which a compiler/operating system environment can be measured. The result of such a measurement provides a quantitative statement of the degree to which such an environment supports the functions of the collection of programs.

The second category establishes a complete organization of factors affecting performance. The organization consists of five components which are introduced briefly in Section 2, and are discussed in detail in Section 7.

Sections 3, 4, 5, and 6 respectively present discussions of the architecture/algorithms question, the same environment question, the environment equalizing question, and the special features questions. For the architecture/algorithms question, the discussion consists of a general introduction to Chapters 3, 4, 5, 6, and 7 in which detailed discussions of algorithm choices and algorithms for four compiler functions are discussed. For the remaining three questions, the discussions in their respective sections provide an overview of the approach taken in this study to explore the question, and a brief summary of how the study of the questions contributes to establishing a basis for cost/benefit analysis of compilers.

2. Technical Concepts

This section introduces the technical concepts on which the study was based. Each of the concepts relate to representing users, environments and compilers, and each is discussed in a separate sub-section. The discussion includes a description of how these concepts interrelate. The five factors which influence performance are introduced in the first sub-section of this section.

Factors that influence compiler performance. Below is a brief introduction to the five classes of factors that influence the measureable performance of a compiler. These factors are discussed in further detail in Section 7.

1. Directly Measureable Factors

   Factors which define performance. (Time and space for both compiler and object code.)

2. Direct Internal Factors

   Factors internal to the compiler (architecture and algorithms) which directly affect performance as measured.

3. Direct External Factors

   Factors external to the compiler (environmental factors such as host machine and operating system, etc.) which directly affect performance as measured.

4. Indirect Internal Factors

   Factors which contribute to the "value" of a compiler (special features) but which cannot be directly measured in terms of performance as measured.

5. Indirect External Factors

   Factors which define how the compiler is to be used and thereby indicate the relative importance of the factors. In combination, these factors define the measured performance of a compiler with respect to a "typical" source language program.

User Profile. A User Profile defines how a user's application programs make use of the various elements and constructions of a language. It is specified quantitatively as the fractions of all the elements or constructions of a language in a "typical" user program which appear in the form of each element or construction. Chapter 8 presents a description of the language elements which might be used as a basis for defining User Profiles.

20

Two different User Profiles are meaningful. A static User Profile counts each occurrence of a language element in the user's collection of application programs equally. A dynamic User Profile weights each occurrence with the relative frequency with which the occurrence is executed in normal use of the collection of programs.

Compiler Performance Profile. A Compiler Performance Profile defines how well a compiler handles each of the language elements and constructs in terms of which User Profiles are defined. Each element is assigned four (or sometimes two) performance measures. These measures are described above as the "directly measurable factors."

Compiler Demand Profile. A Compiler Demand Profile defines how the source code in which a compiler is written makes use of the various elements and constructions of that language. It is specified quantitatively in the same manner as a User Profile. Both static and dynamic Compiler Demand Profiles are meaningful, as in the case of User Profiles.

Compiler Evaluation Profile. A Compiler Evaluation Profile defines how well a compiler performs on a "typical" user program. It is specified quantitatively by four quantities, one for each of the four "directly measurable factors" described above. It is calculated by taking a weighted sum of the four evaluation factors for all elements comprising a Compiler Performance Profile. The weights used for generating compiler space and time performance measures are the static User Profile weights, and the dynamic User Profile weights are used for generating the object code measures.

If additional administrative information is taken into account, then a number of useful dollar valuations can be assigned to a compiler once its compiler evaluation profile has been calculated. These considerations are discussed further in Section 4 of Chapter 8.

"Compiler Gibson mix". A "compiler Gibson mix" defines how well a computer/operating system environment supports the activity of compiling. An example of how a "compiler Gibson mix" might be defined is presented in Chapter 10. This example is based on two static Compiler Demand Profiles generated during this study, one for an AED compiler and one for a J3B compiler.

21

"User Gibson mix". A "user Gibson mix" defines how well a computer/operating system environment supports the collection of user application programs which will normally run in an environment. A "Gibson mix" for a computer has in the past been defined for such applications categories as COBOL applications. These "mixes" are based on specifying the relative importance of different instructions and addressing modes of a computer. It is suggested in Chapter 6 that a "user Gibson mix" could alternatively be developed using the methods of Chapter 10 for generating a "compiler Gibson mix." Whereas a "compiler Gibxon mix" is based on a Compiler Demand Profile, the "user Gibson mix" would be based on a dynamic User Profile.

3.   Overview of the Architecture/Algorithms Question

In Chapter 3, fourteen fuctional elements of a compiler are described. An architecture is described for a one-pass compiler using eight of these elements, and an architecture for a multi-pass compiler is described involving thirteen of the elements. The multi-pass architecture involves thirteen phases in order to demonstrate an extreme design directed toward minimizing space. This architecture also demonstrates the wide range of multi-pass architectures that are possible by means of recombining the twelve phases into a smaller number.

Four of the twelve functions are discussed in great detail in Chapters 4, 5, 6, and 7. The functions discussed in these chapters are table look-up, parsing, optimization, and code generation respectively. Discussed are four categories of table look-up algorithms, four categories of parsers, two broad categories of optimization techniques (involving a total of twenty-eight distinct optimizations), and three types of code generators.

The major conclusion reached from the study of the architecture/ algorithms question is that knowledge of these aspects of a particular compiler is not useful in evaluating the usefulness of the compiler. (A summary of these conclusions is presented in Chapters 1 and 14.)

22

4.   Overview of the Same Environment Question

The full statement of the same environment question is repeated below.

How can two compilers with the same features and operating in the same environment be compared?

Approach. A number of test programs are created which represent (in a suitable weighted combination) a "typical" program to be compiled or executed. The directly measurable factors are measured with respect to the test programs, and the measurements are suitably combined to create a performance measurement for each directly measurable factor with respect to the "typical" program. The direct external factors are not involved since the environments are identical. Direct internal factors are not explicitly considered, because better algorithms and architectures should result in better performance as measures. However, knowledge of variety of these factors is necessary in order to establish a suitable set of test programs. Consideration of indirect internal factors is avoided by assuming that the special features of the two compilers are identical. Indirect external factors comprise the User Profile which determine the weighting factors to use as combining test program results into measures for the "typical" program.

Basis for Cost/Benefit Analysis. Each directly measurable factor may be assigned a relative dollar worth. The performance measures for a "typical" program can then be used to assign a dollar benefit difference between the two compilers. Cost is price.

5.   Overview of the Environment Equalizing Question

The full statement of the environment equalizing question is repeated below.

If two compilers with the same features operate in different environments, how can their measured differences in performance be attributed to the environmental differences vs. the compiler differences?

Approach. Each compiler is evaluated as in the equal environment question to generate performance measures for a "typical" program.

However, these measures represent effects of both direct internal factors (architecture and algorithms) and direct external factors (machine and operating system). To separate these two contributions, generate a Compiler Demand Profile in terms of some suitable elements. This profile constitutes the relative use of the elements by a "typical" compiler. Each element should be represented as an assembly language program programmed by a highly skilled programmer to take advantage of all of the environmental special features which could be exploited by a compiler. These programs are then compiled and executed. The relative time and space for these Compiler Demand Profile programs can be combined to give an overall measure of how well the computer/operating system environment contribute to possible compiler performance for compiler time and space measures. If the test programs used to represent the "typical" user program (the User Profile) are used instead of the Compiler Demand Profile programs, then the resulting measures should show how the computer/operating system environment contribute to object code time and space measures.

Basis for Cost/Benefit Analysis. The answer to the environment equalizing question will contribute to improved purchase criteria for:

(1)   A package of compilers together with computer, operating system, etc.;

(2)   A computer, operating system, etc., when compilers are to be purchased separately; and

(3)   Writing specifications for expected performance of a compiler on a new computer operating system, etc., based on the known performance for a similar compiler on a different computer, operating system, etc.

6.   Overview of the Special Features Question

The full statement of the special features question is repeated below.

How should a compiler buyer deal with the problem of evaluating compilers with different special features?

Approach. Rather than attempt the very difficult allocation of performance degradation to the inclusion of special features (ease of use and ease of maintenance), we instead recommend that the benefit of the feature be calculated independently as a dollar value.

24

Basis for Cost/Benefit Analysis. For the ease of use factors, the benefit derives from reduction in number of debugging compilations, and related factors.* These can relatively easily be assigned a dollar benefit value. In comparing two compilers, their differences in performance may be assigned a dollar value as described above for the same environment question and the environment equalizing question. The dollar value of ease of use factors can then be added into the analysis in a straightforward manner. To apply this approach therefore requires research (beyond the scope of this study) on the psychology of using ease of use factors. For ease of maintenance factors, the suggested approach is to require vendors to supply option price for the features for which there is a clear expectation that they will be used.*

7. Factors That Influence Compiler Performance

With the above brief overview of the three main technical questions of our study to provide an understanding of the overall method of approach for the study, we present in this section a discussion of each of the five categories of factors, and how they interact in effecting the study of these three questions.

Directly measurable factors. The intention here is to make measurements which in combination provide a Compiler Performance Profile. The dimensions of the profile should be such that a User Profile can also be generated which assigns a relative importance factor to each dimension. Consequently, the study establishes criteria which corresponding to these dimensions, and establishes methods of creating test programs to be used in generating the Compiler Performance Profile.

For each dimension, a group of test programs is generated; for each test program, the following measurements are taken:

1. Compilation Time - This factor should be a combination of both CPU time and I/O time. (The present study was primarily based only on CPU time.) If the compiler output is assembly language code, the time to assemble may be included.

2. Object Code Execution Time — This factor is the CPU time required to execute the compiled code.

---

* See discussion on indirect internal factors in Section 7.

3. Compiler Space — This factor is the amount of core (e.g. size of partition) needed to compile the test program.

4. Object Code Size — This factor is the amount of core required to execute the test program.

A combination of the measurements for the test programs within the group results in four summary measures for the dimension.

Direct internal factors. These factors are internal to the compiler and directly affect the values of the four measurements indicated above. They are:

- Architectural organization of the compiler.
- Algorithms used.
- Data organizations used.
- Presence of special features.

The approach we take makes no attempt to allocate or distribute performance measurements among the above internal factors. Rather, the direct benefits and/or costs due to these factors should be accounted for in the dollar procurement cost of the compiler and the Compiler Performance Profile discussed in Section 2. Since the relative ease of maintenance may depend on the architecture of the compiler, and the ease of use depends upon the inclusion of special features, certain indirect benefits of architectural organization and special features will be discussed in a later sub-section on indirect internal factors. In spite of the fact that this study was not directly concerned with the effects of these factors on performance, it is important that the breadth of possible direct internal factors be taken into account in designing the sets of test programs. This means that potential differences in performance between two compilers due to their architecture or algorithms should show up as differences in Compiler Performance Profiles. The weights assigned to these differences, however, are entirely dependent on the User Profiles of the syntactic elements comprising the dimensions of the profiles. (See the later sub-section on indirect external factors.)

Direct external factors. These factors are external to the compiler and directly affect the values of the four measurements indicated above. They are:

26

- Hardware
  - Machine Configuration.
  - Machine Speed.
  - Machine Instruction Set.
  - Machine Instruction Word Format.

- Operating System
  - I/O Interface.
  - I/O Support Software.
  - Linker/Loader/Compilation Unit Format.
  - Inter-Module Interface Standard.
  - Parameter Passing Standard.
  - Error Handling Standard.
  - Assembly Language/Assembler Requirements.

Consider two compilers with the same architecture, algorithms, data organization, and features which are to run in different environments (i.e. different hardware and/or operating systems). One would expect differences in performance due to the above factors. If it is necessary to compare two compilers which have both different internal factors and different external factors, how does one attribute the differences in performance profiles as to internal vs. external factors?* The suggested approach is to develop methods of constructing a profile cf compilers which will be analogous to the "COBOL Gibson mix" (used for evaluating the comparative performance of machines with respect to business applications). The profile of compiler computational element usage may be more or less the source for a wide variety of language types and features, or may be highly sensitive to such differences. The study undertakes to at least approximately determine the degree of such variability. Given a profile for a certain class of compilers, the approach recommends that the profile be implemented by good coders in assembly language for each machine under consideration. The resulting programs constitute "compiler Gibson mix". Given two environments, the performance of the two environments with respect to this "mix" characterizes the overall compiler support performance factor for each

---

* This is simply a restatement of the environment equalizing question.

27

environment for compiler time and space measurements. These factors would then constitute the fraction of combined compiler and environment performance for compiler time and space as measured due solely to environmental differences. The remainder of the differences in compiler time and space performance as measured would be attributed to the direct internal factors of the compilers.

A "user Gibson mix" which characterized the applications to be run could be used in a similar manner to determine the environmental contributions to the object code time and space performance factors. One way in which such a "user Gibson mix" could be derived would be to use the test programs (or a suitable subset) developed to characterize the static and dynamic User Profiles. By hand coding the programs into assembly language, by a good coder, the static profile weights could be used to generate an overall object code space factor characterizing the environmental support. By using the dynamic profile weights, the overall object code time factor could be similarly generated.

The present study of the direct external factors is limited to:

- The approximate determination of the degree of Compiler Demand Profile variability.
- Development of methods for establishing Command Demand Profiles.
- Development of methods for measuring the compiler support performance profiles of various environments.

Indirect internal factors. There are several factors which are based on architecture and special features, and whose costs are therefore internal. However, the major benefits of these factors are external to the compiler.

Architectural factors may be present for such purposes as:

- Portability (changing host machine).
- Retargetability (changing target machine).
- Maintenance (bug fixing).
- Enhancability (adding features).

The suggested approach to these features is that the cost/benefit analysis can be best handled contractionally. This will be discussed further in Chapter 13.

28

Ease of use factors affect the value of a compiler by facilitating the manner in which a compiler can be used operationally. The following are a list of such factors:

- Diagnostics.
- Data dependent error detection.
- Parameter/argument data type matching.
- Syntactic error detection.
- Hooks for traces, breakpoints, symbolic debugging, patching, etc.

The determination of how the value of these factors can be evaluated is beyond the scope of the present study. How their value might be determined is discussed in Chapter 13.

Indirect external factors. The factors which define how a compiler is to be used are both external to the compiler (being determined by the user) and not directly measurable. The suggested approach to these factors is to establish a set of language elements in terms of which a User Profile can be ascertained. This profile will assign to each element a weight which will correspond to the relative importance the element has in the collection of user application programs. For each element, one or more test programs can be written which when compiled and executed generate the four direct measures of compiler performance with respect to these measurements. Section 5 of Chapter 8 presents an organized list of language elements which could provide the basis of establishing User Profiles. How test programs might be prepared to measure a compilers performance with respect to these elements, thereby establishing a Compiler Performance Profile, is described in Chapter 9.

CHAPTER 3
ARCHITECTURAL CHOICES IN COMPILER DESIGN

1.   Overview

Trade-off factors.   In designing a compiler, an architecture is chosen which balances a number of design objectives.   The trade-off factors normally taken into account in choosing an architecture are the following:

- Compiler speed
- Compiler size (including work space)
- Object code speed
- Object code size
- Retargetability
- Portability
- Ease of maintenance
- Debugging features
- Cost of development

Elements of compilers.   Thus, a compiler architecture can be regarded as the organization of the several functional elements required to perform the compiling of a source program.   These functional elements will be described briefly in Section 2.   For the purposes of the overview, they are listed below with short descriptive phrases:

- Lexical analysis -- forms lexemes.
- Table look-up -- maps lexemes to symbols.
- Declaration processing -- assigns types to symbols.
- Parsing -- identify groups of symbol strings.
- *Tree building -- organizes groups into tree structure.
- ***Set/used analysis -- identifies where variables are set or used.
- ***Flow analysis -- identifies straight line executable sequences.

- ***Global machine independent optimization -- restructures tree.
- Storage allocation -- maps variables onto storage locations.
- Register allocation -- maps variables onto hardware registers.
- *Machine independent code generation -- walks tree and maintains state information.
- Machine dependent code generation -- outputs object code.
- **Peephole object code post-processing -- modifies object code locally.
- ***General object code post-processing -- modifies object code globally.

Note: Those functional elements flagged with a single asterisk (*) are used in multi-pass compilers to facilitate information maintenance between phases (passes) of the compilation. The single element flagged with two asterisks (**) is used only within a one-pass architecture for optimization purposes, i.e., to improve object code quality. Those elements flagged with three asterisks (***) are only used in multi-pass compilers for a variety of optimization purposes. (See Chapter 6.)

Interactions among trade-off factors. Let us now consider how the trade-off factors influence compiler architecture. Compiler and object code speed and size constitute the four basic performance measures considered in the study. Compiler size trades-off with compiler speed by means of overlaying and one-pass vs. multi-pass design considerations. Object code speed and size generally both improve together at the cost of compiler speed by means of various optimization techniques.

Retargetability does not interact very strongly with the performance measures, but rather interacts for the most part in the ease of maintenance and cost of development. Retargetability is generally obtained by an appropriate choice of the method to be used for implementing the machine independent part of the code generator. (See Chapter 7.)

Portability interacts somewhat with compiler speed and size in that it is achieved by minimizing that part of the compiler implemented in assembly language. The assumption here is that assembly language

31

implementations are generally more efficient than high level language (say AED) implementations.

Ease of maintenance is generally improved at the expense of compiler speed and size by means of such techniques as top-down design, modularity, and structured programming. Consequently, the methods used to improve ease of maintenance also generally reduces the cost of development for the compiler, except possibly that some one-shot programming training costs may be required for learning the techniques.

Debugging features will generally add to development costs. Generally they also cost something in compiler performance. However, it appears that one can make a quite useful generalization concerning the desirability for debugging features in a compiler. This generalization will be presented below in the context of a general comparison of one-pass and multi-pass compilers.

Contrasting objectives of one-pass and multi-pass compilers. Generally a one-pass compiler is designed for speed as a primary objective. Multi-pass compilers, on the other hard are designed with a trade-off of the various factors in mind to achieve an appropriate balance of these factors. Consequently, it seems desirable to have two compilers, if budget permits, for a given language:

- A one-pass compiler intended primarily for debugging at the unit testing level.

- A multi-pass compiler intended for system integration and the compilation of final production run object code. This compiler should include instruments to facilitate the development of static and dynamic User Profiles.

Since the one-pass compiler is intended for repeated recompilations during the debugging (unit testing) of programs, it should be as fast as possible and should have a set of diagnostic and debugging features which are suitable for minimizing the number of recompilations and debugging runs required to successfully unit test a module. On the other hand, the only optimizations that could be used to improve object code quality are those of the peep-hole object code post processor

32

variety that will not significantly slow down the compiler. The reason for this rather extreme position favoring compiler speed over object code quality is that many recompilations are likely during debugging, and executions will generally be limited to partial or limited trial cases.

The multi-pass compiler must balance compiler performance against object code quality. It is probably desirable to be able to optionally include a variety of optimizing passes depending on whether one is in the system integration phase or the final compilation of production run programs. During these activities, compiler diagnostics are likely to be of much less value than in the debugging (unit-testing) phase using the one-pass compiler. Furthermore, it is desirable to instrument the system integration/production run compiler so that detailed static and dynamic user profiles can be established. These profiles and instruments should be able to locate bottlenecks in the production system so that the system can be fine tuned, and bottleneck programs can be re-programmed or recompiled with more optimizing options so as to improve overall performance. Compiler aids to debugging are less important in these stages as in the debugging stage, but the compiler should still create some support for symbolic referencing/modifying of programs and variables.

The remainder of the review. Section 2 following will present a brief description of each of the functional elements of a compiler introduced above. Section 3 will present a characteristic architecture for a one-pass compiler. Section 4 will present a generalized architecture for multi-pass compilers, in which almost every functional element is a separate pass of the compiler. This generalized architecture will illustrate how various architectural choices within the multi-pass frame-work can be seen as appropriate groupings of the elements or phases in to fewer phases.

2. Functional Elements of Compilers

In this section, a brief description is presented for each of the functional elements of a compiler that were introduced in Section 1.

Lexical analysis. The lexical analyzer scans the input strings character by character and applies lexical formation rules for tokens. If the input source language program is interpreted as a one long character string, then the output of the lexical processor is a sequence of short character strings, each such short string being a lexeme. In the process of lexical analysis, comments are generally removed, and blanks are either removed, or in some contexts, converted to a standard punctuation mark, such as a comma. As a by product of lexical analysis, a lexical code is usually attached to the lexeme which provides some information as to the lexeme's broad category. Examples of categories that might be used by a lexical analyzer are as follows:

- Punctuation mark (e.g. comma (,), semi-colon (;), etc.).
- Reserved works (e.g., GOTO, CALL, IF, THEN, etc.).
- User variable names.
- User labels.
- User literals (may be broken down by data type of literal).

In most architectures, the lexical analyzer is called as a subroutine by the parser (one-pass compilers) or by the declaration processor (multi-pass compilers). When called, the input character string is scanned, and the next single lexeme formed is returned as output. In the multi-pass architecture presented in Section 4, the lexical analyzer is organized as a separate phase in order to illustrate the many architectural choices available for grouping the functional elements.

Table look-up. The table look-up function accepts as input a lexeme which is the output of the lexical analyzer. Using one of a variety of mechanisms (see Chapter 4   ), it is determined whether or not an entry exists for the input lexeme in a symbol table. If so, usually the

location of the entry is returned as output. If not, a new entry is made for the lexeme, and, usually, the location of the new entry is returned.

In a one-pass compiler, the table look-up function is usually called by the parser immediately after a lexeme is returned by the lexical analyzer. In a multi-pass compiler, the table look-up function is generally called by the declaration processor, which is a separate phase. In the architecture presented in Section 4, the table look-up function is itself organized as a separate phase.

Declaration processing. The declaration processor accepts as input the location of an entry in a symbol table, output by the table look-up function, and an appropriate type identifier obtained from the lexeme string in one of several possible ways. The declaration processor stores the type information in the symbol table for the specified entry; it produces no output.

In a one-pass compiler, the declaration processor is called by the parser. In this case, the parsing rules include productions for declarations, and when the parser recognizes that a declaration production is to be performed, the parser calls the declaration processor to perform the work associated with this production.

In a multi-pass compiler, the declaration processor is usually a separate pass. In this case, the declaration processor usually is organized more-or-less as a special parser which recognizes just those simple syntactic productions related to declaration keywords. When such a production is recognized, then the appropriate type information is stored in the symbol table for the current active non-keyword symbol table entry.

In the architecture presented in Section 4, the declaration processor is organized as a separate phase which is distinct from all the other functional elements.

35

Parsing. The parser's function is to apply the syntax rules to the source program. In effect, the parser may be thought of as processing a sequence of symbols (each represented by a location in a symbol table) stacking the symbols on a push-down list (until a pattern matcher recognizes the applicability of a syntactic production), invoking an appropriate output function for the recognized production, and then appropriately modifying the state of the push-down list. (The variety of alternative parsing algorithms is discussed in Chapter 5.) If we ignore the output part of the parser's activity, then the parser can be thought of as placing a pair of syntactical marks (e.g., special parentheses), around a syntactic phrase, where a syntactic phrase is a sequence of elements, each of which is either a symbol or a lower level syntactic phrase. The particular form of output used depends upon how the parser fits into the overall compiler architecture.

In a one-pass compiler, generally the parser is "boss". The sequence of symbols to be processed are "generated" one at a time by successive calls to the lexical analyzer, each followed by a call to the table look-up function. The output function called by the one-pass parser "boss" is the code generator, which directly produces object code for the recognized syntactic phrase.

In many multi-pass compilers (e.g., the AED compiler), the parser is a separate phase which follows the declaration processing phase. The parser processes the linear sequence of symbols which constitutes the output of the declaration processing phase. Although the code generator could be combined with the parsing phase, in most multi-pass compilers the parser invokes a tree-builder as its output processor.

In the architecture presented in Section 4, the parser is organized as a separate phase in which the output generated by the parser is simply the linear string of symbols, together with the pairs of special syntactic parentheses. Also included for each pair of special syntactic parentheses is some information describing the global state of the push-down list (beyond the top portion of the list involved in the matching of a syntactic production) at the time the syntactic phrase is recognized.

<u>Tree building</u>. The tree builder is used in multi-pass compilers to organize the parsed phrases recognized by the parser into a hierarchical tree structure representation. Whenever the parser does not directly invoke the code generator as its output process, then usually a tree builder is invoked so that the information representing the source program that survives the parsing phase is in tree structure form. (Examples include the AED and J3B compilers.) At each node of the tree, the following information is generally maintained:

- The location of one symbol or keyword in a symbol/keyword table. (Generally bottom nodes correspond to user symbols representing operands, and non-bottom nodes correspond to keywords or punctuations corresponding to productions to be performed.)

- State information representing the "global" condition of the push-down stack at the time the syntactic phrase is recognized by the parser.

In the architecture presented in Section 4, the tree builder is a separate phrase distinct from the parser.

<u>Set/used analysis</u>. In a multi-pass compiler the set-used analyzer develops for each symbol corresponding to a user variable a list of information that specifies where in the program the data (contents of storage) associated with the variable is set (changed or stored) and where it is referenced. In addition, it may be convenient to generate a similar list for the setting and using of temporary variables associated with intermediate computational expressions. However, if machine-dependent optimization of temporaries is desired, then information as to how temporaries are used and set can be generated as late as general object code post-processing.

Set/used analysis can be performed during tree building, but in the architecture presented in Section 4, the set/used analyzer is a separate phase.

Set/used analysis is used in order to perform one or more global machine independent optimizations. These optimizations may result in further modifications of the set-used information. (See Chapter 6.)

Flow analysis. The flow analyzer is used in multi-pass compilers to support one or more optimization algorithms. (See Chapter 6.) The flow analyzer processes the tree structure representation of the source program being compiled to identify those nodes corresponding to straight line sequences of executable code. The output of the flow analyzer may either be a table of identified groups of nodes, or suitable codes entered into the nodes of the tree structure.

Architecturally, the flow analyzer could be attached to the parsing phase, or could be part of a global machine independent optimization phase which follows the parsing phase. In the architecture presented in Section 4, the flow analyzer is a separate phase that is invoked one or more times during iterative optimization passes. (See Chapter 6.)

Global machine independent optimization. In some multi-pass compilers, this function takes place following the parsing phase and before the code generation phase. The purpose of this function is to reorganize the tree structure representative of the source programs so as to produce improved performance in the object code to be generated. (See Chapter 7.) Since these optimizations may result in altered set/used patterns, it is assumed that appropriate changes are made to the set/used information as part of the function of optimization.

In the architecture presented in Section 4, global machine independent optimization is represented as one or more separate phases that may be iteratively invoked in conjunction with iterative invocations of the flow analyzer.

Storage allocation. In one-pass compilers, or in multi-pass compilers without set/used analysis or dead variable elimination (during global machine independent optimization), storage can be allocated for

38

variables immediately after declaration processing. Otherwise, allocation of storage to variables is done in conjunction with, or following, global machine independent optimization.

Allocation of storage for temporaries is fairly well handled by the following approach. This approach operates during object code processing by assigning the $n(i,j)$-th temporary of a given data type (say, data type i) to the $n(i,j)$-th sub-expression (which was not immediately used as an operand following its calculation) of that data type within the j-th expression. (Temporaries for common-sub-expressions are allocated during global machine independent optimization.) At the end of code generation, space for $N_i = \max_j n(i,j)$ occurrences of data type i are allocated. A somewhat more optimized approach would reduce the required number of temporaries during general object code post-processing by taking into account that the intermediate calculated result is preserved in a register until needed, and therefore does not have to be stored in a temporary location. (See discussion for register allocation which follows below.)

In the architecture presented in Section 4, storage allocation for variables is a separate phase, and storage allocation for temporaries is done as a separate global object code post-processing phase.

Register allocation. The allocation of hardware registers (e.g., base registers, operand registers, index registers) to variables and indices for subscripted variables is quite complex if a highly optimized approach is desired. Probably the best relatively simple approach is for a preliminary assignment of registers to variables to be made as part of machine dependent code generation using a least recently used algorithm. (The previous contents is allocated a temporary storage location at the same time, and the old contents is "tentatively" stored in the temporary at this time. The global machine dependent post-processor will eliminate these allocations and STORE instructions from the object code. This will take place when it is determined that the old contents will not be again required in a computation within an identified straight line sequence of executable code.)

39

A least recently used algorithm can be incorporated into a one-pass architecture (without the global optimization benefits, of course), as well as in multi-pass compilers. In the architecture described in Section 4, it is assumed that the above described approach is taken for register allocation, although the architecture includes the possibility of using alternative selection algorithms other than the least recently used algorithm.

Machine independent code generation. The separation of machine independent and machine dependent code generation functions is described in some detail in Chapter 7 . In the architecture presented in Section 4, a table driven code generator approach is assumed, and the machine independent code generator is organized as a separate phase which operates in a modified fashion from the procedure described in Chapter 2 for table driven code generation.

Machine dependent code generation. As indicated above, Chapter 7 presents a detailed discussion of the separation of machine independent and machine dependent code generation functions. In the architecture presented in Section 4, the machine independent code generator is also organized as a separate se which operates in a modified fashion from the procedure described in Chapter 7 .

Peephole object code post-processing.. This activity is used in one-pass compilers to improve the quality of the object code produced. A small buffer of object code lines examined to determine if simple machine independent optimizations can be performed. The most common use of this activity is to eliminate unnecessary contiguous LOAD STORE pairs of instructions which result from two consecutive code generation productions.

General object code post-processing. This function serves the purpose of performing a number of global machine dependent optimizations. As discussed in Chapter 6 , this type of optimization is probably best handled in this manner. As indicated above, register allocation and

storage allocation can be optimized in this way. In the architecture
presented in Section 4, a separate general object code post-processor
phase is included as the last phase of that architecture.

3.   An Architecture for a One-Pass Compiler

In this section we present an architecture for a one-pass compiler which is more-or-less representive of how one-pass compilers are organized generally. The one-pass compiler architecture is illustrated in Figure 1. Certain "boxes" in the flow chart are numbered in the upper left hand corner. These "boxes" correspond to basic compiler functions as discussed in Section 2. Note that four error conditions are identified in the flowchart (E1, E2, E3, and E4). It is beyond the intended scope of this discussion to elaborate on error-handling in the compiler.

In the architecture illustrated in Figure 1, the parsing function is "boss". That is, the main routine of the compiler performs the parsing function after appropriate initialization, and by appropriate invocations of the other compiler functions as subroutines. Following the flow in Figure 1, we see that after initialization and opening of files (and other preliminary I/O functions), the lexical analysis function (1) is invoked to obtain a character string corresponding to a lexeme. This function would normally use system I/O functions to fill input stream buffers as they were scanned. Any error detected by the lexical analyzer constitutes a lexical error and would be handled by some appropriate mechanism (not shown) following the node E1.

The character string lexeme is next used as input to the table look-up function (2). If no entry is found, then the table look-up function may or may not make a new entry for this lexeme. What determines this action is the value of state information that indicates whether or not declaration processing (4) is complete. The state variable is set to indicate that declaration processing is complete in conjunction with the storage allocation function (5). If declaration processing is complete, then not finding an entry for the lexeme constitutes an undefined symbol error, which is handled by appropriate action (not shown) following node E2. In all other cases, the address of the symbol table entry for the lexeme is returned and put on top of the push-down stack.

42
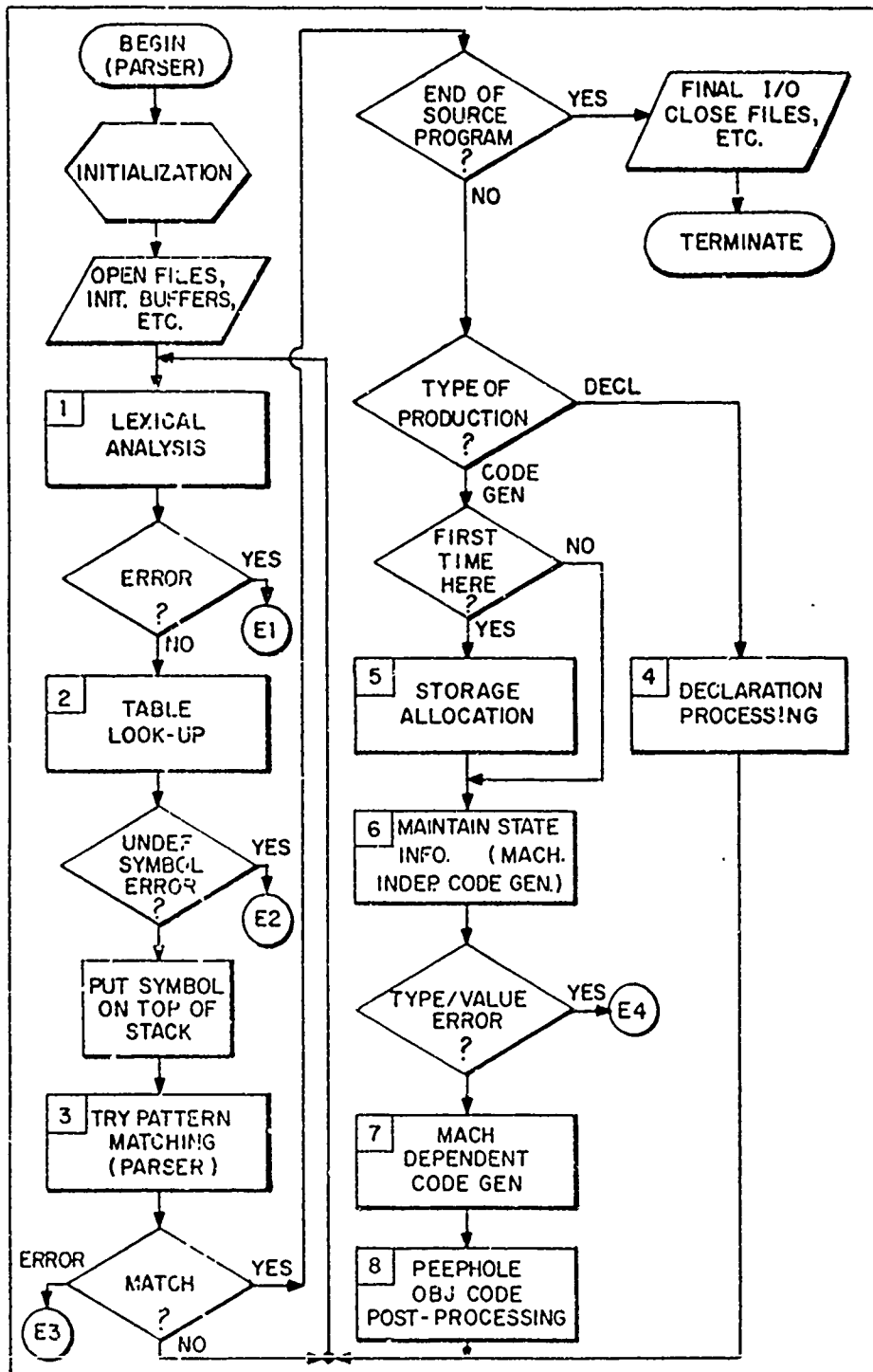
Figure 1. Architectural Flow Chart for One-Pass Compiler

At this point a test is made to the applicability of the syntax rules to the sequence of symbols on the top of the push down stack. This test constitutes the primary parsing function (3) activity. One of three possible results can occur from this test activity:

- A match is found indicating the applicability of a syntactic rule that permits a production to be applied to the input string of symbols.
- No match is found.
- A syntactic error is detected.

Syntactic errors are handled by appropriate functions (now shown) following node E3. The "no match" condition implies that no production is applicable, so control loops back to lexical analysis to obtain another lexeme.

A "match" is handled as follows. First, the test is made to determine if the matching production indicates that the end of the source program has been identified, indicating that the compiler has completed the parsing of the source program. If this is the case, Final I/O processing takes place including "flushing" output buffers and closing files, followed by termination of the compiler activity.

If the production is other than "end of program", then a test is made to determine if the production is a declaration processing production or a code generation production. Declaration processing productions are handled by the declaration processing function (4). This function generally stores type information in the symbol table for a user declared variable. The first time a code generation production is detected, the storage allocation function (5) is invoked to allocate space for user declared variables. (See Section 2.)

Whereas in a multi-pass compiler, the code generation production would usually result in some activity like the tree-builder to develop an intermediate representation of the source program, in a one-pass compiler, the production instead directly invokes appropriate code generation functions. First, the machine independent code generation function is invoked. In the one pass compiler, this generally consists

44

only of maintaining state information (6) to be used during machine dependent code generation (7). During the maintenance of state information activity, it is possible that the compiler might detect a fourth category of error dealing with illegal mismatching of data types or values. These errors are handled by appropriate functions (not shown) following node E4.

Since the result of the machine dependent code generator is a sequence of object code "lines" added to an output buffer, a peephole object code post-processor (8) examines a few lines in the buffer in order to make minor modifications constituting local machine independent optimizations. (See Section 2.)

Following both the declaration processing function (4) and peep-hold object code post-processor (8), flow loops back to the lexical analyzer (1) to obtain another lexeme.

4. An Architecture for a Multi-Pass Compiler

In this section we present an architecture for a multi-pass compiler in which almost every compiler function discussed in Section 2 is organized as a separate phase. The purpose of this architecture is to illustrate two points. First, the degree to which the various compiler functions can be separated so as to minimize core requirements. Secondly, the extremes of separability illustrated demonstrates unmistakenly that very numerous alternatives exist for combining what are presented as consecutive sequences of compiler phases into single phases encompassing more than one compiler function. When such combinations are made, changes naturally occur in the nature of the interface data used between phases as a means of inter-phase communication.

Phase 1. During Phase 1, initialization is performed, files are opened, and other I/O functions of a preliminary characters are performed, substantially the same as in the one-pass architecture. (See Section 3.) Then the lical analyzer is iteratively called to generate a sequence of lexemes, also in a similar manner as in the one-pass compiler. Here, however, the lexeme generated is directly put into a buffered output stream for processing by Phase 2.

The space required for Phase 1 consists of that needed to contain the initialization, I/O support, and lexical analysis programs, and data structures together with space for input and output buffers.

Phase 2. Functionally, Phase 2 consists entirely of the table look-up activity. In addition, I/O support functions are required. The input to Phase 2 consists of the sequence of lexemes, suitably buffered, as output by Phase 1. The output of Phase 2 consists of a symbol table with entries for each user declared (or defined) lexeme, together with a suitably buffered output stream of symbols, each being a location of the symbol in the symbol table. It is assumed that any permanent data used for the table look-up functions, such as, for example, a partial symbol table for punctuation and/or key words, are included with the total symbol table indicated above.

46

We note that in some languages, a suitable use of key words and/or punctuation would permit a separation of the output stream into two independent streams: one for declarations and one for executable statements.

Phase 3. Functionally, Phase 3 consists entirely of the declaration processor. In addition, I/O support functions are required. The input to Phase 3 consists of the symbol table and the sequence of symbol table locations, suitably buffered, which constituted the output of Phase 2. (If the note indicated for Phase 2 applies, then only a part of the entire output stream needs to be processed by Phase 3.)

Included in the declaration processor, as envisioned in the present architecture, is a partial parsing function for just those syntactic rules selected to declaration processing. Generally, this part of the syntax should constitute only a small subset of the total syntax of the source language. As declaration productions are identified, appropriate entries are made in the symbol table. Also, the symbol table may be structured according to the hierarchical nesting of declaration blocks, so that only appropriate portions need be core resident during parsing.

The space required for Phase 3 consists of that needed to contain the declaration processor programs and data structures (including the partial parsing functionality), together with I/O support programs and data structures, and additional space for input buffers and the symbol table. Note that no buffers are required for an output stream since Phase 3 does not generate any.

Phase 4. Functionally, Phase 4 consists entirely of that larger part of the parser dealing with the syntax of executable statements. In addition, I/O support functions are required. The input to Phase 3 includes the symbol table as modified by Phase 2, suitably organized hierarchically so that only appropriate portions need be core resident while parsing within nested procedure definition and block boundaries. In addition, Phase 4 inputs the sequence of symbols suitably blocked, constituting the output of Phase 2. If the note indicated for Phase 2

47

applies, then only a part of the entire output stream needs to be processed by Phase 4.

In the present architecture, the output of Phase 4 consists of an edited version of the input stream. The editing consists of the insertion of pairs of special syntactic parentheses together with appropriate state information, as described in Section 2.

The space required for Phase 4 consists of that needed to contain the executable statement parser programs and data structures, together with I/O support programs and data structures, and additional space for the symbol table, and input and output buffers. Note that the separation of the tree building function from the parsing phase removes the need to also hold the tree structure representation of the source language program in core concurrently with the above listed items.

Phase 5. Functionally, Phase 5 consists entirely of the tree builder function. (See Section 2.) In addition I/O support functions are required. The input to Phase 5 consists of the output stream, suitably buffered from Phase 4. Note that the symbol table is not a required input for the tree builder, as it is for the parsing function. Furthermore, the tree builder is a very simple function as compared with the parsing function. Input support functions are also required.

The output of the tree builder is the tree structured representation of the source language of program being compiled. State information is the input stream (as output by the parser) is placed into the appropriate nodes of the tree structure. Conceptually, at least, it is assumed that the entire tree structure remains in core when Phase 5 completes its work. However, in practice, the tree might be organized so that smaller portions can be sequentially processed by later phases provided excessive I/O for re-reading and re-writing the separate parts can be avoided.

The space required for Phase 5 consists of that needed to contain the tree builder and I/O support function programs and data structure, and additional space for I/O buffers and the tree structure output of the phase.

Phases 6, 7, and 8. The next three phases consists of the set/used analyzer, the flow analyzer, and the global machine independent optimizer respectively. These three functions are optionally used iteratively to perform machine-independent optimizations as described in Chapter 6 . The input to these phases is the tree structure output by Phase 5 together with the symbol table output by Phase 3. The result of the processing performed by Phases 6, 7, and 8 is a restructuring of the tree to reflect introduced optimizations, as well as appropriate indications in the symbol table of unused symbols following all introduced optimizations.

Each of the three Phases, 6, 7, and 8 require space for the symbol table and the tree structure. In addition, space is required for the respective functional programs and data structures. Note, however, that no intermediate file interfacing exists, and therefore no I/O support functions are used and no I/O buffers are required.

Phase 9. Functionally, Phase 9 consists entirely of the part of the storage allocation function dealing with user declared variables. (See Section 2.) The input to Phase 9 is the symbol table. The output to Phase 9 consists of entries in the symbol tab. to reflect the addresses within blocks of the storage allocated to various variables, and an additional table (SYMDEF's and SYMREF's) which will be output with the object code in a suitable format to provide linkage for external variables.

The space required for Phase 9 consists of that needed to contain the storage allocation programs and data structures for variable handling, as well as the symbol table and the SYMDEF and SYMREF table for external variables.

Phase 10. Functionally, Phase 10 consists of a very small part of the machine independent code generation function as described in Chapter 7 . This small function is described in Chapter 7 as the recursive Tree Walker (TW). Support output functions are also required.

49

Let A denote an arbitrary node of the tree, and B and C its left and right descendents respectively. (Note that either or both of B and C may be NULL nodes.) Phase 10 then consists of a single recursive call to TW with the base node of the entire tree as its single argument. TW then operates as follows:

- Output the contents of node A with code $\underline{1}$.
- If the pointer to node B is not NULL, then recursively call TW with the pointer to B as argument.
- Output the contents of node A, with code $\underline{2}$
- If the pointer to node C is not NULL, then recursively call TW with the pointer to node C as argument.
- Output the contents of node A with code $\underline{3}$.
- Return to caller.

It is clear that the entire effect of Phase 10 is to linearize the information contained in the tree in exactly the same sequence as would be available to the machine independent code generator as described in Chapter 7 .

This space required for Phase 10 consists of that needed for the TW program and data and support output programs and data structures, together with space for the tree and output buffers. Note that the symbol table need not be core resident during Phase 10.

Phase 11. Functionally, Phase 11 consists of the major part of the machine independent code generator. It consists of a scanning program which linearly scans the data from the nodes of the tree as output by Phase 10. For each node, a call is made to an appropriate node-type related program which consists of code to process the node for each of the tree codes that can accompany the node information. Thus, each node-type related program contains the functionality of three programs as described for the code generator in Chapter 7 . I/O support functions are also required.

50

There are two main functions performed by each node-type related program. One function is to maintain state information that characterizes the context (derived from ancestor nodes from the tree) that may effect the interpretation of the node information. The other function is to output the parameters which would have been included in the sequence of calls to a table driven machine dependent code generator interpreter. In Chapter 7, the machine dependent code generator calls this interpreter, but in the present architecture, the call is replaced by putting the parameters into an output stream to be processed by the machine dependent code generator in a separate phase.

The space required for Phase 11 consists of that needed to contain the scanning program, the various node-type related programs, I/O support programs, data structures for these programs, and space for the symbol table and input and output buffers. Note that the linearizing of the tree by Phase 10 removed the need for having the tree concurrently core resident with the large programs of Phase 11.

Phase 12. Functionally, Phase 12 consists of an input stream scanning function, the machine dependent code generator, a preliminary storage allocator for temporary variables, and a register allocator. Support I/O functions are also required. The input to Phase 12 is the symbol table and the sequence of parameter data, suitably buffered, which was the output of Phase 11. The output of Phase 12 is an output stream comprising the object code form of the source program being compiled.

The input stream scanning function performs some initialization of state variables, etc., and then scans the sequence of parameter data combinations in order. For each such combination, the table driven interpreter (which is the form used here for the machine dependent code generator) is called with the parameter list. The interpreter, as part of its functioning, will include register allocation function and a preliminary temporary variable storage allocation function.

51

In order to perform these functions, each register will be associated with an identifier for its contents within each straight line sequence of executable code. These sequences will have been identified during flow analysis. An algorithm, such as the least recently used algorithm, will be used to reassign a register to new contents for an intermediate result during a calculation, or for a compiler assigned temporary resulting from common expression elimination during global machine independent optimization (Phase 8). As each reassignment is made, code is generated to store the previous contents (if any since the beginning of a new straight line sequence) in the next available temporary location of the appropriate type (e.g. base register/pointer; index register/integer, floating point register/real, etc.). When the end of a straight line sequence is found, the bookkeeping tables are appropriately updated to represent the fact that the contents of registers are no longer available. This may also result in code being generated to store the present contents of registers in temporary storage locations. The final temporary storage allocation function is performed in Phase 13.

The space required for Phase 12 consists of that needed for the table driven interpreter and its data structures (including those for managing register allocation and storage allocation), the input stream scanning program and I/O support functions and their data structures, and additional space for the symbol table and input and output buffers.

Phase 13. This phase includes a number of general object code post-processing functions used for global machine dependent optimization. (See Chapter 6.) In particular, Phase 13 includes a function to make the final storage allocations for temporary variables. I/O support functions are also required.

The input to Phase 13 is the object code output by Phase 12. The output of Phase 13 (including all such post processing activities) is an edited form of the object code.

The final storage allocation for temporary variables works as follows. The object code is scanned to determine those occasions where the stored value in a temporary is not subsequently used. In such cases, the STORE instruction can be detected, and the storage for the temporary may be deleted if all such uses are eliminated.

The space required for Phase 13 consists of that required for the particular post processor functions and I/O support functions and their respective data structures (including bookkeeping tables such as, for example, for the storage allocation functions described above), together with space for input and output buffers.

CHAPTER 4

TABLE LOOK-UP ALGORITHMS

1. Overview

This chapter reviews the set of compiling algorithms which are classed as table look-up algorithms. In this class are included all those algorithms for organizing information into a number of different structures and accessing any specific entry only through the use of a single identifying attribute. Many such algorithms have been developed for use in assemblers, compilers, and filing systems, and many have been thoroughly modeled and analyzed mathematically. The structures used by these algorithms are not necessarily tables, but also include tree structures as well. In addition, the various structures may have many different internal organizations, and, consequently, they have definite ranges of performance characteristics which are dependent on the manner in which that information is accessed, altered, and processed, as well as on the attributes of the information which is stored into them.

The first sections of this chapter discuss the various computational structures which support table look-up operations, and these are di ussed independent of any specific application. The remaining sections explore the application environment for table look-up algorithms within the context of compilers. The concluding section summarizes and highlights some of the more interesting trade-off and performance aspects for the algorithms and the applications which are discussed. As is indicated in this last section, Section 5, the greater part of the information and data presented here is drawn from the Art of Computer Programming, Vol. III, by D. E, Knuth, and this survey is augmented with results from a number of reports and papers which have appeared since this book was published.

2. Categories of Table Look-up Algorithms

Table look-up algorithms fall primarily into four broad categories determined by the data access methods and storage organizations they employ.

54

1. Sequential list algorithms are the simplest with information taking on uncomplicated tabular or list forms. The searching techniques used are strictly serial, possibly taking advantage of orderings cn the table to gain advantages in performance.

2. Binary table algorithms operate on ordered tables and have average and worst case retrieval properties which make them attractive for certain applications.

3. Tree organizations allow great flexibility in providing rapid updating, and well defined worst case performance. Most tree structures are either the simple binary form, or the more complicated multi-way forms.

4. Hashing and algorithms built around hashing as a basic function have emerged as the most useful class of table look-up algorithms to date. They are particularly interesting in the mathematical problems involved in their analysis, and considerable work has been done in this area.

Each of these broad classes provide specific opportunities for the system designer to adapt a particular algorithm to his needs and, in some cases, to even mix the strategies of two or more such algorithms to synthesize a system with more desirable performance characteristics. Discussed in the sections which immediately follow are the basic data structures, and the operations performed on them, which characterize the above four categories.

Sequential Algorithms   The simplest algorithms involve straightforward sequential search of a tabular data structure. Each element of the table is examined in sequence starting with the first, until either the desired element is found or the end of the table is reached, indicating that the element is absent. The best case performance results when the desired element appears first in the table, and the search process terminates immediately. The worst case performance results when the desired element is nonexistent or appears last in the table, in which case all the elements of the table must be examined in turn. Given a table of N elements, each accessed with the same frequency, the average number of probes A, required for retrieval of an element is $A = N/2$. That is, the average number of probes grows linearly with the length of the table.

Some improvements in performance can be gained whenever there are differences in the frequency of retrieval requests for the elements. This

is achieved by ordering the elements in the table so that their positions correspond to their frequency of access. More frequently accessed information appears earlier in the table than less frequently accessed information, yielding a reduction in the average number of probes, with the reduction dependent on the frequency distribution of the elements. If the elements of a table are ordered on frequency, and the frequencies form, for example, the sequence $f_1 = 1/2$, $f_2 = 1/4$, ..., $f_j = 1/(2^j)$), then the average number of probes can be shown to be $A = 2-1/2^{N-1}$.

The deletion of a table element or a contiguous block of table elements can be achieved by marking them as non-existent; however, since these elements must be accessed examined during the retrieval of any elements which follow them in the table, they contribute to the cost of retrieving those elements. This cost may be reduced through a compaction of the table to reclaim the space taken up by elements marked for deletion but not physically removed. The cost of performing the compaction is simply the cost of moving the elements of the table into the vacated positions, and the decision to compact can be made whenever this cost is exceeded by the average cost of retrieving. For tables which are ordered on frequency of access, deletions and compactions preserve this ordering.

Insertions for tables without frequency ordering are achieved by adding the element to be inserted to the end of the table. Tables with such ordering require that the new element be placed within the table at the appropriate position, and this generally requires that all elements following it in the order be physically shifted down one position to make room. The cost of making an insertion into a frequency ordered table consists of the number of accesses required to find the appropriate position and the data movement required to displace the following entries down one position.

A dynamic frequency ordering can be achieved by treating the table as a least recently used stack. That is, each element is moved to the 'top' of the table whenever it is accessed and the topmost elements are pushed down, thus allowing the table to dynamically adjust to whatever access pattern exists at a particular time. More frequently accessed elements will tend to move to the top of the table, while elements which

tend to be less frequently used will move to the bottom of the table. Deletions and insertions in this scheme can be handled as in the unordered table, with the usage pattern of the newly added or remaining elements being determined by the pattern of accesses to them.

Binary search algorithms. Binary search algorithms operate on tables which have some content dependent ordering defined for them. This ordering is usually achieved by treating each element's identifying field as a numeric or alphanumeric field, and then sorting the table using this identifier as a primary key. For a table of length N, the search begins by first comparing the key of the table entry whose position is closest to $N/2$ with the key of the desired element. If the two keys agree, then the search is successful, and the desired element has been found. If the search key is greater than the table entry key, the desired element must be found in the half of the table having all elements with keys greater than that of the table entry. The converse applies if the search key is less than the table entry key. In either case, half of the table has been excluded from the search on the basis of this one comparison. The search then proceeds, treating the proper half of the table in the same manner as above. The search terminates either on locating the desired element, or on finally halving the table to just one entry. The maximal length search requires $\log_2 N$ comparisons and the average number of comparisons is $A = \log_2(N-1)$.

Insertions and deletions are achieved in the same manner as for frequency ordered sequentially searched tables. That is, an element inserted in the binary search table must be positioned so as to preserve the ordering defined by the key values. Elements following the one to be inserted must be moved down in the table to provide room. Deletions can be accomplished by marking elements as non-existent, and the table compacted when the cost of handling those deletions rises above some threshhold. A number of properties of the binary search algorithm make it attractive for use in table look-up applications. Among these there is a well-defined and easily calculable worst case performance measure which is very close to the average performance of the algorithm. In addition, the ordering defined on the table is an intuitive one, and the table can be brought into initial order with the use of standard sort

algorithms, without additional data on access patterns as required by frequencey ordered tables.

Tree searching algorithms. These algorithms operate on tree like data structures where each node in the tree represents a data element. Such algorithms have search patterns that begin with the top, or root node of the tree and successively isolate subtrees which are searched until the desired element is located or found to be absent. The most common tree form used is the binary tree in which each node can have at most two successor nodes as indicated in Figure 2.
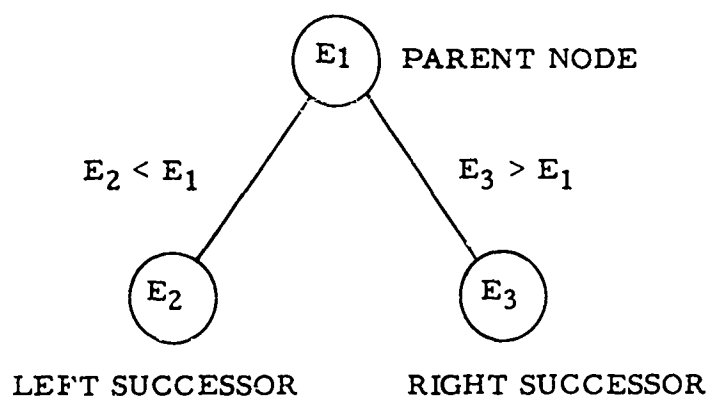
Figure 2 - Typical Node Relationships for a Tree Structured Search Table

Some tree forms allow a parent node to have many successors; others require that the data elements stored in the tree be situated at the lowest level, with the nodes having values which guide the search to the proper element at this level. In the interest of brevity we will discuss only the binary tree and some variations on this structure which improve performance of the algorithm for specific classes of use.

In the binary tree search, the key of the desired element is compared with the key of the root node. If the key is equal to that of the root node, the search is successfully terminated. If the key is greater than the key of this node, the search proceeds using the right successor node as a new root node. The converse of this applies if the key is less than that of the root node element. Eventually, the proper node is found, or the search

58

terminates at a node which has no successor pointer for the result of the comparison of the desired key with the key at that node. An example tree is shown in Figure 3 employing a set of keys entered in the order D, C, A, B, F, E, G.
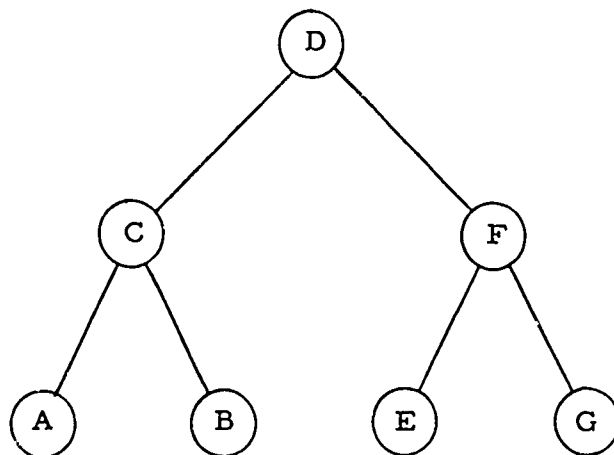
Figure 3 - A Typical Binary Tree

The resulting tree has the property of being partially ordered. That is, all nodes subordinate to right successor of a given node are all greater than that node, and conversely all nodes subordinate to the left successor of a given node are all less than that node. Thus in searching such a tree the pattern of accesses to nodes proceeds from level to level, isolating at each level successively smaller subtrees for examination. While this search pattern resembles closely that of a binary table search, the average and maximum number of comparisons required span a rather wide performance range. For a tree which is well balanced, the search length is proportional to log N, and for unbalanced trees the search length is proportional to N. (This worst case performance results from entering a set of elements in inverted order, with the greatest key first, and the least key last.)

Deletions in binary trees require some care in preserving whatever balance exists in the tree. Proper deletion strategies prevent any one path in the tree from growing much shorter than any other path. Insertions require that the tree be searched for an available position in much the same manner as a retrieval is performed; the new node becomes the left

59

or right successor of the appropriate node. Various specialized algorithms (such as the AVL, balanced tree scheme) have been developed to create and maintain a pseudo-random distribution of key values at the nodes, thereby bounding the average and worst case performance.

The binary searched tree is attractive for several of its retrieval and update characteristics. Among these are the similarity of its performance to that of the binary searched table. The tree form also facilitates the insertion and deletion of data elements without involved data movement or displacement of existing elements within the tree. To implement this algorithm, some sort of dynamic storage algorithm is necessary, and in determining cost of updating, the work required for managing the storage pool must be taken into account.

Hashing algorithms. Hashing algorithms have grown to be the method of choice in many of the compiling systems currently used. These algorithms must operate on a fixed sized table, say of length N, and treat each of the available locations as addressable by an integer between 0 and N-1. To locate the address of the position in the table where a given element is to be stored, the key associated with that element is mapped via some numerical function onto the integers 0, 1, 2, ..., N-1. The value of the function for each key is taken to be the address of the appropriate table position. If the function which is chosen to perform this mapping is fairly well behaved, most of these data elements would map to unique addresses, and a few would map to the same address. To resolve the conflict at those addresses which are assigned more than one element under the mapping, a secondary function must be applied to determine which elements to assign to those table positions which remain unoccupied. As with the actual mapping function, the conflict resolution may be performed in many different ways. A considerable amount of work has been performed on the behavior of both hashing functions and conflict resolution or overflow functions. Under the assumption of a fairly good hashing function, and assuming that the table is to be filled with K elements, where $K \leq N$, the average number of comparisons for a linear search overflow method is found to be $A = 1/2 \, (1 + 1/(1 - a))$, where $a = K/N$ represents the fraction of total positions that are occupied. Comparable results are obtained for other conflict resolution methods.

Insertions and deletions are constrained to preserve the size of the table, since this number is a parameter of the mapping functions and must remain unchanged if all the other elements are to remain addressable. Once, under successive insertions, the table becomes full, a new one must be created, in order to increase the size of the table. If under deletions, the ratio a grows very small, considerable effort is required in order to reclaim the unused space. Despite these minor disadvantages, the average cost of retrieving information from hash addressed tables makes them quite attractive in terms of their use in compilers. Empirical studies have shown that average retrieval costs for reasonable sized tables can be made to approach very close to one comparison at the expense of allowing unused storage in the table to keep the loading factor at somewhat less than one. Additional reductions in average retrieval costs can be gained by entering elements into the table in the order of decreasing frequency of usage.

3. Compiling Applications

All of the table look-up algorithms described in Section 2 have been employed in compiling systems to implement a wide variety of functions which require table look-up activity for their operation. The most familiar and well known of these applications are symbol table management and pattern recognition. Pattern recognition is used in compilers to identify particular constructions which can be reduced to simpler ones, or for which optimized code generation sequences can be formed. Such uses attempt to improve the object code produced by a compiler by reducing the space it occupies and/or by the run-time required to execute it. Some discussion of the usage of table look-up algorithms for optimization can be found in Chapter 6.

The most wide-spread use of table-look-up methods, however, has been in the area of symbol table management for compiling systems, and the following discussion presents some of the specific applications and various factors which affect performance of the algorithms in those applications.

Symbol tables are used to recognize and manage information about several classes of symbols which are encountered in source language programs. Some tables are used to identify special key words or tokens which have preassigned meanings; others are used to store user defined

symbols and their attributes; specialized tables are used to store whatever literals appear in a source program. The lexical analysis and declarations processing phases of compilation perform the translation of source program character strings first to language tokens, and then to symbol table locations. These can be used by the following parsing and code generation phases in retrieving whatever symbol attribute information they require. Of the three classes of symbols, key words, user symbols, and literals, key words present the least difficulty in symbol table management. Key words, such as 'IF', 'THEN', 'FOR', can all be organized into a single convenient table, and each name encountered in the program can be easily compared against entries in this table to determine if it is a key word. If key words are maintained in a separate area, they can be ordered on frequency, for more efficient searching. Alternatively, all the key words can be stored along with user defined symbols, generally without disturbing the structure of the table, allowing the same access routines to be used for both classes of symbols. Since their attributes are fixed, and predefined, such symbols present no insertion, or deletion problems, and are subject only to retrieval. Because key words have fixed attributes they are sometimes referred to as permanent symbols.

User symbols require both insertion and retrieval operations, and in some languages with block structure, they also require deletion operations or their equivalents. Insertions of user symbols are performed at their point of definition. In many languages user symbols are defined by their first usage, with their attributes being determined by the context of that usage, or by implicit spelling rules; this is true for example of FORTRAN. Other languages require explicit declarations of all user symbols before their usage, and certain of these languages restrict such declarations to all appear together in a declaration portion of the source program. However, since empirical studies have shown that user symbols, similar to key word symbols, are heavily biased toward retrieval, insertion operations represent a one time cost which can be significantly less than the total cost of retrieval over a compilation.

Literals such as '1', '3.1459', '-10' etc., are almost universally defined by their usage, and are usually maintained in a table organized pool. A compiler usually allocates a literal to a created 'constant' variable,

and will search this pool to avoid duplicate created constants. This form
of retrieval can be facilitated by transforming the literal to a bit string in
a constant form, and then using this string as an identifier for the table
look-up operation. Thus '1.E+01' and '10.0' would both have the same bit
string representation, and would identify the same literal.

4.   Language and Program Structure

The structure of a source language, and the form of source programs
written in it, can have some effect on both the choice and performance of
a particular table look-up algorithm for symbol table management. This
is particularly true of languages which permit block structure either with
or without the qualification of user symbols within a block by the block
name. The effect of block organization is to cause all symbols defined
within a block to refer only to storage associated with that block, even
though a containing or possibly separate block may also define symbols
with the same spelling. The effect of qualification is to permit within any
given block, explicit reference to symbols of another block, by simply
qualifying the symbol name with the name of the external block. Both of
these features require modifications to the simplified table look-up

algorithms which permit the compiling system to distinguish reference
to user symbols on the basis of block context. The general appearance
of such symbol table management is to create a separate symbol table for
a block when it is entered, use this to satisfy retrievals of symbols local
to the block, and then erase this symbol table when the block is passed.
Symbols which were defined previous to that block can then be retrieved
in a logically consistent manner.

One such scheme, employed by the AED compiler, uses a Master
Spelling Table to hold all the unique spellings of symbols encountered in all
the blocks of a program, and separate sub-tables (Block Attribute Tables),
each subordinate to a block, to hold all the attributes of the associated
symbols. Access to the Master Symbol Table is via hashing of the symbol
spelling name. Associated with each name in the table is a pointer to its
currently active attribute entry. For instance consider the following program
schema.

```
BLOCK B1 ;
    DEFINE A, B;
        BLOCK B2;
            DEFINE A, C;
            END B2;
    END B1;
```

At the point when the second 'BLOCK' definition is encountered the Master Spelling Table and Block Attributes Table resembles the configuration shown in Figures 4 and 5.

Figure 4 - Illustrative Master
Spelling Table -
Stage 1

Figure 5 - Illustrative Block
Attributes Table -
Stage 1

Now when the new definition of A is encountered, the Master Spelling Table entry for 'A' is altered to associate that spelling with the attribute entry subordinate to block B2, and a copy of the old association is placed in that entry.

The Master Spelling Table and Block Attribute Tables now resemble the configuration shown in Figures 6 and 7.

A	⑤
B	③
C	⑥

① ② ③

④	B1		B2
⑤	A		A
	∅		2
⑥	B		C
	∅		∅

Figure 6 - Illustrative Master Spelling Table - Stage 2

Block B1.　　　Block B2.

Figure 7 - Illustrative Block Attributes Table - Stage 2

When block B2 is exited, the copy is recovered, and replaced in the Master Spelling Table, restoring A's old association block B1.

Block qualification can also be handled transparently using hashing by appending to each symbol an identifier for the block which defines the symbol, thus achieving uniqueness, and then using this unique spelling as an argument to the hashing function.

Binary tree organizations are fairly adaptable to such wholesale deletions and insertions, provided those operations preserve random distribution of the symbols throughout the tree, and avoid serious degradation in performance.

5. Summary and Conclusions

The above material has summarized salient features of table look-up technology, and the compiling environments where table look-up algorithms are applied. The specific performance behavior of all these methods, or the impact of the compiling environment on this behavior has not been considered. As indicated in the introduction, much of the relevant analysis is contained in The Art of Computer Programming. Vol. 3, by D. E. Knuth, and the reader is directed there for specific details, proofs, and the like. Additional references are found at the end of this report.

The basic conclusions can be summarized briefly in the following statements, which have to do with trade-off and performance aspects of the methods described above.

- Each algorithm has performance characteristics which favor its use in particular applications. The following is a brief summary of the areas of performance trade-offs:

  - Very short fixed lists can be organized for sequential search table management with reasonable efficiency if the data elements are ordered in frequency of use. Moderately small variable bits might be better treated with sequential methods than with binary search methods, although some hashing methods will give improved performance unless block oriented processing is required.

  - Moderately large lists are amenable to binary tree representations, although variability in the tree introduced by block inserts and deletes will in general disturb the average performance characteristics of the search strategy.

  - Very large lists, or no block oriented processing for any but the smallest lists, are best treated using hashing methods. Proper choice of a hashing function, collision resolution method, and loading factor (either through analysis or empirical study) can achieve very attractive average performance characteristics. Block oriented processing requires some care in the implementation of search strategies to avoid introducing excessively large overhead costs.

CHAPTER 5

PARSING ALGORITHMS

1. Overview

This chapter presents the results of a review of parsing algorithms conducted during the study. The general conclusions reached from our study of parsing algorithms is as follows:

- For most parsing techniques in common use, differences in implementation overshadow differences in technique in impact on performance.

- For one technique category ("general techniques"), which is not commonly used in compilers, the performance expected would generally be poor due to the generality of the technique. This expected poor performance is probably the reason the technique is not commonly used in compilers.

- The reasons a particular technique is selected for use in a compiler are generally distinctly independent of performance considerations.

- Aside from performance considerations, it is possible to make some general statements about various advantages or disadvantages one parsing technique would be expected to have in comparison to the other techniques.

In Section 2, the categories of parsing techniques are summarized. Section 3 discusses the kinds of factors which influence the choice of parsing technique to be used in designing a language and/or a compiler. Section 4 summarizes some generalizations of non-performance related advantages/ disadvantages among the various techniques.

2. Categories of Parsing Techniques

Below are presented five categories of parsing techniques . For each category a brief characterization of the category is presented, together with a brief discussion of variations within the category currently in use.

General techniques. These techniques can handle any context-free grammar including non-deterministic and ambiguous grammars. Algorithms within this category are essentially the British Museum Algorithm, or some variant. Consequently, performance is expected to be poor.

Limited look-ahead bottom-up. This category includes the least restrictive deterministic techniques. This category also permits the algorithmic (automated) translation of a BNF description of the grammar rules into a table for driving the parsing algorithm. (This automated parser building property is a characteristic of most commonly used techniques.) These techniques all involve looking ahead a number, say k, of lexemes to determine if the top of the stack completes a new phrase, or if the next lexeme should be put on top of the stack. The most general algorithm of this category is the LR(R). Variations such as SLR(k) and LALR(k) attempt to simplify the bookkeeping tables in exchange for some loss of generality.

Precedence systems. Precedence systems are also "bottom up" techniques which attempt to construct phrases from simpler elements. All are equivalent to special cases of the previous category. There are many variations. Conceptually, one or more precedence tables are used to determine if the top of the stack completes a phase, or if the next lexeme should be put on top of the stack. The grammar rules may be directly represented by the precedence tables, rather (or in addition to) representing the grammar in BNF. Variations trade generality (in a number of different dimensions) for table size and complexity.

Deterministic top-down techniques. These techniques are all essentially recursive techniques where a phrase causes a recursive procedure to be invoked to break a phrase into smaller constituent phrases. Whereas the limited look-ahead bottom-up techniques are by their nature deterministic, the top-down techniques may require considerable designer effort to ensure its deterministic character.

Reductions analysis techniques. These techniques are conceptually similar to the bottom up techniques, except that the grammar is represented as a program for doing pattern matching on the stack configuration, rather as BNF.

3. Parser Selection Factors

The principal decision criterion in selecting a parsing technique for a compiler is the availability of support tools for converting a language grammar representation into a functional parser. If a compiler designer has, for example, only tools for LALR(1) grammar, then this is the technique of choice.

The reason the availability of tools is a primary importance is that a variety of "fixes" are available to handle those cases not directly encompassed by the parsing technique chosen. These "fixes" will be discussed below. However, it will be useful to first note that virtually all compilers use some "fixes" for situations that are by their nature beyond the reach of any of the parsing techniques in common use.

Virtually all languages have one or more aspects that are not context-free. For example, the constraints against duplicate or undefined symbols are handled outside the grammar and parser by means of auxiliary symbol table management mechanisms. Data type matching may also be handled in the same manner rather than expanding the grammar to include a much larger number of production rules. Lexical analysis is also generally handled outside the parser, although it is sometimes included by expanding the grammar with rules for symbol generation.

In view of the accepted approach to go outside the grammar when the technique does not fit some language requirement, the generality of a parsing technique is not a critically important selection criterion. However, "fixes" using auxiliary mechanisms generally take significant time and energy to work out, and reduce reliability by increasing compiler complexity. Consequently, if several parsing techniques are available, one should probably choose the most general technique.

Besides using an auxiliary mechanism there are other "fixes" possible to fix a mismatch between language requirements and the chosen technique. If the compiler design also has control of the language, then a slight change of a language construction may fix the mismatch problem. If the language is a "given", then the possibility of modifying the grammar rules for describing the language may fix the situation. The introduction of additional intermediate phrase constructions might be one such possible grammar level fix. If the language is

69

"given" and the grammar cannot be made to fit, then an auxiliary mechanism outside the parser might be used. Finally, changing the choice of parsing technique to a different and more general system (which is not immediately available) could be used as a last resort.

4. Advantages and Disadvantages of Various Parsing Techniques

Presented below is a summary of some generalizations concerning the advantages and disadvantages of the various parsing techniques discussed earlier.

General Techniques:

- are too slow.
- have poor error recovery.
- provide no help in design.
- might be useful in experimental environments with frequently changing language and grammars.

Limited Look-Ahead Bottom Up:

- are very general.
- permit fast performing implementations.
- have storage requirements that vary greatly with particular choice within category.
- are fairly complex.
- are amenable to good error recovery.

Deterministic Top Down Techniques:

- have excellent error recovery capability.
- are somewhat less general than the above techniques.
- permit excellent design support aids.
- permit very fast performing implementations.
- include some requirements for ad hoc design effort to guarantee deterministic behavior.

Precedence Systems:

- are very good for processing arithmetic expressions.
- also permit trade-offs of generality for table size.
- are relatively simple.
- have good error recovery possibilities.

70

- permit very fast performing implementations.
- are moderately general.

Reductions Analysis Techniques:

- are very general.
- require much ad hoc design effort.
- can pr⋅ ⋅de excellent error recovery.
- permit very fast performing implementations.
- are hard to debug.

CHAPTER 6
OPTIMIZATION ALGORITHMS

1. <u>Overview</u>

This chapter discusses work performed in this study in surveying
the literature on techniques for the optimization of compiler generated
object code. The range of optimizations covered is quite broad and
includes all identified areas which can possibly influence compiler
performance in any way. Specific algorithms employed in optimizing
compilers were not directly studied since their implementations are
generally quite diverse. Instead, an organized survey is presented of
classes of optimization which are currently used in commercially
available compilers, or which are currently being investigated in various
research environments. Each of these optimization classes is discussed
in terms of examples of the application of the techniques involved, and
what kinds of improvement they might produce in object code.

The examples presented are in the form of sequences of simple
statements. These may be shown with a representation of distinct linear
sequences of executable code linked with branching flow above and
reconvergence below. Each example consists of an "original" sequence
and an "optimized" sequence which reflects the effect of the particular
optimization being illustrated.

In lieu of exploring specific algorithms in detail, a thorough
discussion is presented of the kinds of preliminary analysis activity required
by a compiler in order to generate a sufficient body of information about
a program so that optimizations can be performed on them. The kinds
of actual data manipulation and processing required to perform the
program restructuring and code generation activities necessary for
optimization are also presented.

Thus, the discussion extends over a two dimensional conceptual
space with optimization classes as one coordinate and activity classes
as the other. The dependencies of optimizations on analysis are

represented in the form of two matrices in Section 6. The use of a matrix form is intended to clearly indicate the processing activity required to support the optimizations being discussed. Optimizations can be of the machine independent type, discussed in Section 3, and the machine dependent type, discussed in Section 4. An Optimization-Analysis matrix is provided in Section 6 for each type. Analysis classes can be oriented toward data gathering and information structuring or the actual performance of the optimization. Both cases are discussed in Section 5.

2. Categories of Optimization Methods

Optimizations fall into two fairly distinct categories: the machine independent type and the machine dependent type. Machine independent optimizations tend to require considerable manipulation and restructuring of the source program structure and logic. For this reason, most require an intermediate representation of source program structure. There are many such representations which are more or less equivalent, and for the purposes of the present discussion it is assumed, without appreciable loss of generality, that the intermediate representation used is a tree structure. Generally, machine dependent optimizations attempt to improve object code quality by adapting source programs to a particular computer hardware environment. Such optimizations generally do not invol·e modifications of the tree structure. On the other hand, machine independent optimizations are generally representable by specific changes in the tree structure representation. Section 3 discussed machine independent optimizations, and Section 4 discusses machine dependent optimizations.

3. Machine Independent Optimizations

Common expression elimination. Common expression elimination refers to the transformation of programs to minimize the repeated calculation of an expression which occurs in several distinct places. The net effect of applying this optimization is to compute a repeatedly used expression only once, store its value in a temporary location, and then for all later occurrences of that expression, substitute the value stored in the temporary.

73

There are essentially two different scopes for this optimization:

- Within single assignment statements.
- Across a multiplicity of assignments statements.

Both optimizations require modification of the tree structure representation of the source program being compiled, but the first requires only local examination and manipulation of this representation. The second necessitates more involved analysis. Since the expressions of interest span possibly a number of intervening assignments, a flow analysis and set/used analysis is required before constant values stored in temporaries can be substituted for occurrences of these expressions. Both optimizations rely heavily on pattern matching computations to identify occurrences of the same expression, and tables to hold expressions which potentially can be optimized. The pattern matching involves matching one part of the tree structure representation to another part. The specific implementation used for this data structure, the matching algorithms, table organization and table access methods all determine the performance of the common expression optimization processing within a compiler.

An example of common expression elimination is the following:

Original	Optimized
E = A * B	E = A * B
C = A * B + D $\longrightarrow$	C = E + D

Equivalent expression elimination. This optimization is for all practical purposes identical to common expression elimination. The most significant distinguishing characteristic is that it may be applied over expressions which, at the source code level, have different constituent variables. In terms of program logic (at the static level, without actually having to execute the program) the optimizer can determine that although such variables bear different names, they actually take values which render their respective containing expressions equivalent. One means of recognizing this kind of equivalent results from assignment propagation together with the appropriate substitutions

74

of values or variable references. Once two different expressions have
been so reduced, the machinery of common expression elimination can
be applied, and any potential optimizations then introduced. Assignment
propagation then is a precursor of some removal of redundancy in
expressions that differ at the source code level, but which are semantically
equivalent under the substitution of values or names for certain variables
within these expressions. Equivalent expression elimination can be
performed along with common expression evaluation, but is applicable
only after a preliminary transformation which propagates known values
through the program's tree structure. All considerations of scope bounds,
flow analysis, set/used analysis then apply in the same manner as for
common expression evaluation.

An example of equivalent expression elimination is the following:

Original	After Assignment Propagation	Optimized
E = A * B	E = A * B	E = A * B
F = B $\longrightarrow$	F = B $\longrightarrow$	F = B
C = A * F + D	C = A * B + D	C = E + D

Code motion out of parallel paths. Equivalent code can frequently
be displaced from parallel paths through a program. Such paths can be
established through an instance of an IF-THEN-ELSE construction in
which the THEN clause and ELSE clause constitute the program segments
which are parallel. Such parallel paths are also created through GOTO
statements involving variable indexed switches. Such statements cause
flow to diverge from a common point in a program (possibly rejoining
at a later point). An analysis of program flow can be performed which
identifies forking or reconverging situations, which create parallel
paths. Equivalent statements in such parallel paths may be moved up
above the forking point or below the point of reconvergence (if one exists).
The direction in which code can be moved is dictated by performing a
set/used analysis.

If no variables referenced as data sources within a statement are altered between the forking point and the occurrence of the statement, then that statement can be removed from the parallel paths and established above the forking point. Thus the work done by the statement is performed only once, rather than on each path. On the other hand, if variables set by a statement are not referenced below the statement up to the reconvergence point (if one exists), then that statement can be moved below the reconvergence point.

This optimization relies heavily upon flow analysis to identify the straight line sequences of executable code, the forking points and the reconvergence points. Once this has been done, pattern matching is used to isolate those constructions which potentially may be moved up or down from parallel paths. Whether such motion of code is performed depends on the results of set/used analysis. Code can actually be moved only if the conditions above are met and then only in the directions indicated.

An example of code motion out of parallel paths is the following:

| Original | Optimized |
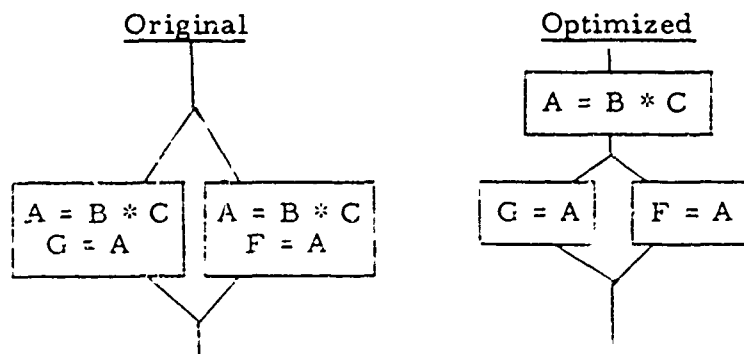
Invariant computations. Invariant computation optimizations also result in code motion, but of a special kind. This form of optimizations attempts to improve the object code resulting from iteratively controlled loops, and equivalent forms, by moving constant computations outside loops where they are evaluated only once. The requirement for performing this optimization is a flow analysis to identify and bound

the iterative loop, and an associated set/used analysis to determine those constructions which can be calculated above the loop, and then referenced as values of temporary locations. The same facilities employed in code motion optimization can be applied to move invariant code upward outside of loops.

Dead variable elimination. Dead variable elimination involves the recognition and elimination of variables which are set but not used, or which are declared but never referenced. Essentially, a variable which is never referenced can be deleted. Furthermore, the deletion of certain variables in a program may be sufficient to trigger the deletion of other variables which may have been used to produce a right hand side value for that variable in an assignment. A set/used analysis is sufficient to support this optimization. This set/used analysis can be performed either before other optimizations, or after all other optimizations. If the former, then each subsequent iteration of assignment propagation should update in an appropriate manner the results of the set/used analysis. The actual work of the optimization is the maintenance of the set/used data, usually in conjunction with the compiler's symbol table. Once this work is done, the optimization is made effective during the storage allocation function of the compiler. (A description of the storage allocation function is presented in Section 2 of Chapter 3.)

An example of dead variable elimination is the following:

Original	After Assignment Propagation	Optimized
A = 10	A = 10	
B = A $\longrightarrow$	B = 10 $\longrightarrow$	C = 10 + D
C = B + D	C = 10 + D	

Redundant statement elimination. This optimization involves the removal of statements which have no effect upon the execution of a program. Such statements fall into two classes:

- Those which are apparent at the source code level.
- Those which result from an assignment propagation pass.

This optimization is dependent only upon flow analysis and set/used information. Assignments of values to variables not used before they are reset with new values can be eliminated. Such operations are most easily performed on the tree structure representation.

Variable merging. Variable merging is a form of space saving optimization. The storage allocations to variables which are never used concurrently are overlayed. Flow analysis and set/used analysis is required to make the determination of the active or inactive state of a variable in various segments of a program. Inactive variables become candidates for storage sharing, provided their set/used information indicates their storage allocations can be safely reused. The optimization becomes effective during storage allocation.

Formula transformation. Formula transformation is one of the more complex categories of program optimization. There are many different transformation which can be applied, and almost any kind of transformation can be considered (short of actually changing the algorithm).

- Expression reordering is usually done as a space saving optimization by reducing the use of temporary locations required to evaluate an arbitrary expression. Some labeling, counting, and interrogation of the tree structure representation is required to determine reorderings that do indeed improve usage of temporary locations. However, the effect of the optimization is best seen in the resulting object code produced as illustrated in the example below. The following is an example of expression reordering:

Original   Optimized

$E = A + B + C * D \longrightarrow E = C * D + A + B$

LOAD	A		LOAD	C	
ADD	B		MPY	D	
STO	T		ADD	A	
LOAD	C	compiler	ADD	B	compiler
MPY	D	output	STO	E	output
ADD	T				
STO	E				

78

- **Factoring** involves essentially algebraic manipulation of the source program at the intermediate representation level. It relies heavily on pattern matching techniques, and the interrogation of fixed tables of transformations which may potentially be applied.

  The following is an example of factoring:

Original	Optimized
$D = A * B + A * C \longrightarrow$	$D = A * (B + C)$

- **Strength reduction** is primarily concerned with the optimizations of calculations within iterative loops. Operations involving iterative computation (e.g. multiplication, exponentiation, factorials, etc.) can be objects of this form of formula transformation optimization.

  The following is an example of strength reduction:

Original	Optimized
FOR I = 1, N	Z = K
BEGIN	FOR I = 1, N
L = K * I + M $\longrightarrow$	BEGIN
END	L = Z + M
	Z = Z + K
	END

- **Constant evaluation** generates values for expressions involving only constants at compile time. Simple pattern matching to isolate the constants and associated operators is sufficient to determine opportunities for this optimization.

  The following is an example of constant evaluation:

Original	Optimized
$R = 2. * 3. 1415926 \longrightarrow$	$R = 7.2831852$

- **Low power exponentiation** changes exponentiation by a low integer power into a sequence of multiplications. In the machine independent realization this optimization it is performed as follows. First, pattern matching in the tree structure representation is performed, then a transformation is made of the representation of the exponentiation expression into a representation for a sequence of multiplications.

79

The following is an example of low power exponentiation:

Original		Optimized
$X = Y ** 5$	$\longrightarrow$	$T1 = Y * Y$
		$T2 = T1 * T1$
		$X = T2 * Y$

Subroutine closing. Subroutine closing is similar to code motion. However, the straight line program segments affected need not be parallel. For this optimization, identical groups of statements which appear in distinct areas of a program are "extracted", and a single copy established as a closed subroutine. Calls to this closed subroutine replace the statement groups. Thus a space savings results as compared with the unoptimized program. This transformation is supported by flow analysis and pattern matching to isolate the code segments of interest. A major modification of the intermediate representation is required to physically restructure a program in this fashion.

The following is an example of subroutine closing.

Original $\qquad\qquad$ Optimized

$$
\begin{bmatrix} S_1 \\ S_2 \\ \vdots \\ S_n \end{bmatrix}
\longrightarrow
$$

$$
\begin{bmatrix} \text{CALL P} \end{bmatrix}
$$

$$
\begin{bmatrix} \text{CALL P} \end{bmatrix}
$$

$$
\begin{bmatrix} S_1 \\ S_2 \\ \vdots \\ S_n \end{bmatrix}
$$

$$
\left. \begin{bmatrix} S_1 \\ S_2 \\ \vdots \\ S_n \end{bmatrix} \right\} \text{This sequence becomes subroutine P.}
$$

80

Retargeting of jumps. Retargeting of jumps is one of the conceptually simpler optimizations, and it can be performed either as a machine dependent or machine independent optimization. For machine independent retargeting, a flow analysis is used to reveal those transfers of control which pass through one or more other branch statements to reach a particular program statement. Then the original branch statement is simply retargeted to the destination program statement to eliminate intervening branches. (Machine dependent retargeting is discussed under Section 4).

The following is an example of retargeting of jumps:

Original	Optimized
.	.
.	.
.	.
GOTO L1	GOTO L2
.	.
.	.
.	.
L1: GOTO L2 $\longrightarrow$	L1: GOTO L2
.	.
.	.
.	.
L2: ---	L2: ---

4. Machine Dependent Optimizations

Minimization of memory references. Minimization of memory references is a generalization of the register allocation problem. Registers on almost all machines can be referenced and altered at rates significantly faster than main memory locations. This, and the fact that the majority of machines require variables to be held in registers in order to be operated upon, make it very desireable to minimize references to memory in order to reduce execution time. The common approach is to allocate a target machine's registers in such a way as to minimize direct reference to main memory for the values of variables. For machines with only one or two operational registers, the whole problem of register allocation is of little consequence. However, for machines with many

operational registers, where such registers may have specific usage characteristics, such as indexing, fixed point, and floating point, the problem can be quite complex.

Several different forms of this optimization may constrain the register allocation process. One form merely attempts to keep as many registers busy as possible during the evaluation of complicated arithmetic expressions since this will reduce stores and retrievals of values for variables and temporaries. Another form attempts to bind certain variables to registers across loops, in order to keep frequently and periodically referenced values readily accessible in machine registers. Yet another form is concerned with addressing which is dependent on base register and index register usage.

Register allocation, as a means of minimizing memory references, is a complicated process resulting in algorithms which can become quite complex. However, simplified heuristics, such as assignments based on least recently used rules, can achieve results which are respectably close to optimal at a much lower computational cost. Most sophisticated algorithms, especially those which operate over loops, require flow analysis and set/used analysis to provide sufficient information about the source program to allow efficient register allocation.

Elimination of redundant stores. Elimination of unnecessary stores attempts primarily to isolate and remove store instructions which move values from registers to variables and temporary locations. There are two techniques for achieving this elimination. The first requires that the compiler have complete information about the reference and usage patterns of variables during the code generation operations on arithmetic or Boolean expressions. Using this information during the register allocation pass, it is possible to determine just when the value assigned to a register is no longer needed. Rather than force a store of the register's value to a temporary when that register must be freed for other use, the register is simply reassigned and the value forgotten. The second method involves using a simpler procedure requiring less complete knowledge of the reference patterns, and a correspondingly simpler register allocation algorithm. A heuristic approach is used for the allocation of registers, and, as the code

generation progresses, special usage tables are maintained which are sub-
sequently used in a post-processing pass to isolate and remove redundant
stores.

Retargeting of jumps. Retargeting of jumps can be performed
during object code post-processing. This required following chains
of transfer instructions, and then shortening the chains to direct
branches.

Low power exponentiation. Exponentiation by low integer powers
can be detected and efficiently processed during the machine dependent
code generation process. The exponentiation code generator can examine
the exponent expression, determine its form, and if appropriate,
generate an appropriate sequence of multiply instructions, thereby
avoiding the otherwise costly call to a library exponentiation routine.

Use of special instructions. Code generators which attempt to
use special instructions available on the target machine can achieve a
considerable improvement in the object code they produce, though such
improvements are generally quite local in their nature. This optimization
can be accomplished one of two ways. Either special machine dependent
code generators or tables can be used which recognize and respond to
the special situations in which special instructions can be used, or
object code pattern matching can be done during general object code
post-processing, and the appropriate object code substitutions made at
that time.

The following are two examples of the use of special instructions:

- Shifts and increments may be substituted for certain
  machine operations to more efficiently achieve a standard
  program operation, which would otherwise require more
  code or slower code to be generated. Fairly transparent
  examples include the use of SHIFT instructions to
  perform integer division and multiplication by powers of
  two, and INCREMENT instructions to achieve addition
  by one. Such substitutions are generally handled
  by special logic in the code generation process.

83

- Special operations and memory-to-memory operations available on some machines are largely unexplored in optimizations currently practices. However, some compilers do generate code for such machines. The identification of opportunities for such optimizations is easiest to implement by analysing the tree structure representation of the source program during machine dependent code generation.

Use of special data reference modes. Data organizations in current languages can be quite complex, with elementary items organized into data structures with possibly many hierarchical levels. Referencing such structures or parts of structures present an opportunity for optimization at both storage allocation and data referencing time.

The following are three examples of the use of special data reference modes:

- Variable precision is allocated by many compilers; they maintain sufficient information about the types and precision of data items to allow the storage allocation pass to determine whether or not an item should be stored in a byte, halfword, or double word location. For instance, some systems restrict loop and iteration variables to occupy half word locations unless explicitly declared to be of greater precision.

- Packing and unpacking of data items is simplified by machines which have facilities for accessing variable length bytes, or which allow referencing of partial word using special modes or masks to discard portions of words which are not needed.

- Immediate and indirect access modes are of great utility in compiler generated code, in accessing simple constants and referencing based or pointer referenced data.

5.   Optimization Activities

Optimization activity within compilers can be organized into two fairly general categories. The first involves all those algorithms, processing tasks, and data manipulations which are preparatory to the actual performance of optimizations such as we have discussed above. This includes the preliminary analysis of program structure, and the

construction and maintenance of necessary data and associated tables to support these optimizations. The second category includes those operations by which optimizations of a program are realized. These can be organized into two distinct classes: (1) those which require modification of the program at the level of the tree structure representation, and (2) those which can be achieved through the use of special machine dependent code generation elements or table entries. Most of the former are machine independent optimizations, which are considerably influenced by the actual instruction repertoire of the target machine.

Flow analysis. Flow analysis is performed on a program to develop several kinds of useful information. The primary result is a static representation of the straight line executable sequences and the dynamic control paths within a program. For the purposes of the present discussion, straight line executable sequences are called "blocks". Secondary information includes identification of dominance relationships among blocks, parallel control paths and the like.

Flow analysis is performed by taking into consideration the influence on the flow paths of various language elements. Some simple examples of such considerations are the following:

- FOR statements represent iterative loops.
- IF-THEN (-ELSE)constructions create parallel paths.
- Conditional GOTO's create either parallel paths or iterative loops. (The determination of which is well defined but somewhat complex.)
- Labeled statements which are the objects of GOTO's and which are executable in sequence from preceding statements represent reconvergence points of parallel paths, or possibly, loop constructions.
- BEGIN-END pairs bounding no GOTO's or IF-THEN-(ELSE) constructions, or labeled statements constitute boundaries of a block (or a constituent part of a block).

The results of such analysis is of great use in many optimizations that attempt to perform movement of statements or statement groups. The hierarchical dominance relationships among statement blocks can

be determined as well, allowing some optimizations of register allocation across block boundaries. This information is represented as block dominance graphs, and indicates what statement groups are subordinate to others within control flow paths. Optimizations proceed from innermost blocks to outermost blocks.

Flow analysis information is most conveniently represented in the form of directed graphs with blocks of statements as edges, and branches and control receiving points as nodes. For the purposes of determining the connectivity of blocks, matrix forms are also used. An initial adjacency matrix, giving those blocks which can be reached within a single control step, under repeated self multiplication is sufficient to determine which blocks are adjacent in any number of steps.

Set/used analysis. Set/used information is collected for each variable declared in a program. Uses are recorded whenever a variable appears on the right hand side of an assignment statement, as an argument in a subroutine call, or in an expression which must be evaluated subordinate to another statement. Sets are recorded for a variable whenever it appears on the left hand side of an assignment statement, as an argument in a subroutine call or as an iteration control variable. In program languages which allow pointers to refer to variables, any usage of a pointer to reference data also counts as set/ use references for all variables to which the pointer refers.

Set/used information can be collected relative to control flow structure, including block dominance relationships. Such information can be easily represented in list form, with each variable "owning" a list of source program statements in which it is set, and a corresponding list of statements in which it is used.

Tree structure representation pattern matching. Pattern matching on the tree structure representation presents a fairly difficult problem to the compiler designer. While such analysis could proceed easily using node by node comparison on a tree structure, considerations of efficiency require greater attention to organization and execution.

86

Pattern matching on tree structures can be of a local nature (constrained to a single arithmetic statement). Techniques for speeding up such matching and searching include the use of hashing techniques and the insertion of special information at certain nodes within the tree. It is also possible to take advantage of host machine characteristics in representing tree structures and searching for specific bit patterns during the matching process. For instance, using a single or double word to hold an operand-operator-operand structure can yield considerable improvement in search times.

Pattern matching at the tree structure representation is essential to carry out optimizations based on common expression or subexpression detection. Formula transformation also requires such techniques to isolate node structures which have simpler or more efficient expressions. Pattern matching can be applied directly to the original tree, or to the tree resulting from an iteration of assignment propagation.

Tree structure representation processing. Some additional processing of the tree structure can be of utility in facilitating certain optimizations. For expression reordering, labeling the tree for node depth is convenient in signaling when a construction can be rearranged to minimize the generation of temporary locations in its evaluation.

Object code pattern matching. Detecting special patterns in object code produced by a compiler may be performed either as part of a separate general object code post-processing pass, or in a limited fashion as part of a peephole object code post-processor. If part of a separate pass, then global occurrences of special instruction sequences are sought, as well as opportunities for reordering evaluations, constant evaluation, retargeting jumps, and many other special cases of machine independent optimizations. Such searches can be implemented as special tests, or they can be driven by a sophisticated finite state machine which can be constructed to detect rather complex object code patterns and trigger their reduction.

87

If the pattern matching is performed as part of a peephole object code post-processor, then the range of object code over which matching is attempted is limited to the contents of a relatively small core resident buffer.

Assignment propagation. Assignment propagation falls quite appropriately in the class of preparatory optimization work. It can potentially induce sufficient equivalence in form to allow pattern matching driven optimizations to proceed where, otherwise, they would have been blocked by superficial source program differences. (See discussion of equivalent expression elimination in Section 3.)

Iteration. Several iterative preparatory and optimization work passes are possible. For example, an iteration might involve an assignment propagation pass followed by code motion, redundant statement elimination, common expression elimination and dead variable elimination.

Tree restructuring. Restructuring of the intermediate representation is required to support many machine independent optimizations. Restructuring includes the operations of eliminating nodes in the tree structure, replacing node sequences with equivalent ones, altering node linkages and so on. Most of these restructurings are generally controlled by specific optimization algorithms and are local in nature, involving only a small number of nodes.

Major tree restructuring. Alternate of program structure on a major scale is rarely required, but in the instance of subroutine closing, considerable code motion and elimination may occur, which would have such an effect.

Space allocation adjustment. Allocation of storage to variables generally leaves the intermediate representation unchanges. Most alterations required by variable elimination and merging can be easily and simply indicated with a program's associated symbol table.

88

Object code post-processing. Post-processing of object code
generated by the compiler is performed for a number of reasons.
Simple examples include removal of redundant LOAD-STORE or
STORE-LOAD pairs, and the transformation of certain code sequences
to more efficient ones. In conjunction with pattern matching algorithms,
more complex optimizations can be performed, such as movement of
invariant code. However, code motion of any sort requires considerably
more work at this stage than at the global machine independent optimiza-
tion stage when the function operates on the tree structure representation.

Special code generators. Code generation routines, which are
engineered for optimizing certain local constructions for a given target
machine, fall into this class. The primary optimization is to employ
special machine instructions to achieve standard operations with
improvements in speed or space utilization. Simple examples include
the use of shifts for multiplication by powers of two, increments for
addition or subtraction by one, and immediate data modes to reference
constants. Such code generation functions can operate directly from
tree structure representations, or those representations can have been
augmented with special information to signal constructions which are
candidates for special techniques. This special information would have
been placed into the tree structure representation (or into auxiliary
data structures) during general machine independent optimization.

6. Matrices of Optimizations vs. Required Analysis

Two matrices are presented below. The first (Figure 8)
summarizes the preparatory work and optimization work required to bring
about the various machine independent optimizations discussed in
Section 3. The second matrix (Figure 9) presents a similar summary
with respect to the machine dependent optimizations discussed in
Section 4.

89

MACHINE INDEPENDENT OPTIMIZATIONS

	PREPARATORY WORK							OPTIMIZATION WORK				
	Iteration	Assignment Propagation	Flow Analysis	Set/Used Analysis	Intermediate Representation Pattern Matching	Labeling of Intermediate Representation	Object Code Pattern Matching	Restructure Intermediate Representation	Major Modification of Intermediate Representation	Adjust Space Allocation	Object Code Post Processing	Special Code Generators
Common Expressions Elimination												
Within Single Assignment			X	X	X			X				
Across Multiple Assignments		X	X	X	X			X				
Equivalent Expression Elimination												
Within Single Assignment			X	X	X			X				
Across Multiple Assignments		X	X	X	X			X				
Code Motion Across Parallel Segments												
Common Expression	X	X	X	X	X		X	X				
Equivalent Expressions		X	X	X	X			X				
Invariant Computations			X	X				X				
Assignment Propagation	X		X	X	X			X				
Dead Variable Elimination			X	X						X		
Redundant Statement Elimination												
Vacuous Statements	X	X	X	X	X			X				
Equivalent Statements	X	X	X	X	X			X				
Variable Merging			X	X				X				
Formula Transformation												
Expression Reordering					X	X	X	X				
Factoring			X		X	X		X				
Strength Reduction			X		X	X		X				
Constant Evaluation					X	X	X	X				
Low Power Exponentiation					X	X		X				
Subroutine Closing		X	X		X				X			
Retargeting of Jumps		X	X		X			X				

Figure 8.   Matrix of Preparatory and Optimization Work Required for Machine Independent Optimizations

90

PREPARATORY WORK

OPTIMIZATION WORK

MACHINE DEPENDENT OPTIMIZATIONS

	Flow Analysis	Set/Used Analysis	Labeling of Intermediate Representation	Special Tables	Object Code Pattern Matching	Register Assignment Heuristics	Object Code Post Processing	Special Code Generators
Minimization of Memory References	X	X				X		
Minimization of Redundant Stores			X	X			X	
Retargeting of Jumps			X				X	
Low Power Exponentiation								X
Use of Special Instructions Shifts for Multiply, Divide			X		X			X
Special Operations			X		X			X
Memory-Memory Instructions			X					X
Use of Special Data Reference Modes Byte, Halfword, Etc.				X	X		X	X
Packing and Unpacking				X	X		X	X

Figure 9.   Matrix of Preparatory and Optimization Work Required for Machine Dependent Optimizations.

CHAPTER 7

CODE GENERATION ALGORITHMS

1. Overview

This chapter discusses various methods of code generation used in compilers. The range of code generation methods used is quite small as compared with optimization, table look-up, or parsing algorithms. In fact, there are only three distinct methods used:

- Directly Programmed Code Generators.
- Macro Organized Code Generators.
- Table Driven Code Generators.

In Section 2, a paradigm architecture for code generation is described. This architecture describes an organization for the machine independent part of a code generating phase, which follows a parsing phase. The three methods of code generation indicated above constitute alternate methods of including the machine dependent parts of a code generator. The place in which each of the above three methods of code generation fits into the paradigm is also presented in Section 2. Then in Sections 3, 4, and 5, each of these three methods will be discussed from the point of view of overall advantages and disadvantages. One clear conclusion can be summarized here:

- There is no definite advantage or disadvantage with respect to compiler performance in comparing the three methods.

This conclusion for code generation is the same as that found for parsing algorithms and, to some extent, for table look-up algorithms. The choice of an algorithm in designing a compiler is generally made for other than compiler performance considerations. Consequently, compiler performance is best determined by direct measurement rather than by looking into its architecture and algorithms. With respect to optimization algorithms, for example (see Chapter 6), algorithms are generally selected to provide a desired balance between compiler performance and object code performance.

92

2. Paradigm for a Code Generator Architecture

The code generation architecture described below includes what is more-or-less the overall functional activities performed by a code generation phase of a compiler. Although there are unlimited variations possible, the essential elements would all have to be present in any architecture used. The purpose of describing a single architecture as a paradigm for code generation architectures generally, is to provide a basis of comparison of how the three specific methods indicated in Section 1 fit into a code generation architecture.

The architecture described assumes a separate code generation phase following a parsing phase. The parsing phase is assumed to produce a binary tree structure repsentation of the source language program, and the code generation phase walks through the nodes of the tree to gather information used to generate object code. In a one pass compiler, some state information available from the parser could be used by the code generation activity; in a multi-pass compiler, this information would normally have to be recreated if it were needed. Furthermore, in a one pass compiler, the parser does not generate nodes of a tree as output; rather the parser productions directly call upon the node type related code generating procedures (denoted below as PROG-A-1, PROG-A-2, and PROG-A-3) as the parsing activity discovers appropriate productions are to be applied. However, these distinctions are relatively unimportant from the point of view of providing a single context for comparing the three code generation methods listed in Section 1.

At the completion of the parsing phase, the following information is assumed to be included in the tree structure representation of the program being compiled. At each node of the tree there is specific information, more-or-less related to particular syntactic elem nts in the source language, as well as the linkages to the other nodes which define the tree structure. Let us refer to the node specific information simply as the node type. Let us denote an arbitrary node of the tree as A. Let B and C be the left and and right descendants of A respectively. (Note: either or both of B and C may be NULL nodes).

93

We assume a recursive tree walking program, TW which operates as follows. When TW is invoked, a pointer to a node, say A, is passed as an argument. TW thus performs the following sequence of steps:

- Call a program associated with the node type of A, say PROG-A-1. (PROG-A-1 may be NULL in which case this call is skipped).

- Extract the pointer to B. If not NULL, call TW with the pointer to B as argument.

- Call a second program associated with node type A, say PROG-A-2. (PROG-A-2 may be NULL, in which case this call is skipped).

- Extract the pointer to C. If not NULL, call TW with the pointer to C as argument.

- Call a third program associated with node type A, say PROG-A-3. (PROG-A-3 may be NULL, in which case this call is skipped).

The activity of the entire code generation phase can be considered the result of a single call to TW with a pointer to the base of the entire tree as argument.

Note that in the above sequence of steps, TW does not take into account any state information. All tests of state and changes of state are handled in the programs PROG-A-1, PROG-A-2 and PROG-A-3. Also note that there are relatively many distinct such programs, possibly as many as three for each node type, which would correspond to three for each syntactic element of the language. It is certainly most likely that the· will be at least one non-null program for each node type. We will next look at the functional elements of each such program.

Each node type related program consists of a machine independent part and a machine dependent part. The machine independent part interrogates information available at the node and state information; branches are taken and state information may be updated. Each flow path of the machine independent part may lead to a distinct machine dependent part. The machine dependent parts set up parameters specific to the outputting of object code which constitutes the final compiled version of the program being compiled. The specific object code that is generated depends on the state information set up by the machine independent part, as well as the specifics of the target machine for which the output code is intended. It is these machine dependent parts of each node type related program thatn can be implemented by means of one of the three methods listed in Section 1.

94

In summary, the machine independent part is (generally) implemented as direct code involving conditional branching, testing of state information, and updating state information.* The machine dependent part is implemented in one of three ways (direct code, macros, table driven interpreter). It tests state information and causes the desired output code to be generated.

3. Directly Programmed Code Generators

If the machine dependent parts of the node type related programs are implemented as direct code, then each such part of this code will consist of a sequence of steps as follows:

- Set up parameters
- Call Object Code Output primitive.
- Set up parameters.
- Call Object Code Output primitive.

.

.

.

- Set up parameters.
- Call Object Code Output primitive.

Note that there is a single primitive invoked several times with different parameters.

The principal advantages of this method is the simplicity of preparing the machine dependent parts of the program. There is no requirement to interface with a Macro processor; consequently, it is not necessary to be concerned with the special problems associated with the use of distinctly different Macro language (rather than the language in which the rest of the compiler is coded). Furthermore, the computer time required to run the compiler code through the Macro processor is saved. Overall, the direct code method should be less expensive to use in implementing a compiler and the resulting compiler should have a corresponding lower development cost.

The principal disadvantage of the direct code method is that the resulting compiler is much harder to retarget. Consequently, if several

---

* It is technically feasible to implement the machine independent part as a table driven interpreter, which behaves as a finite state machine. However, the study found no examples of such an architecture in actual use.

equivalent compilers are desired for different target machines, then the total development cost for the group of compilers will be considerably greater for the direct code method as compared with either of the other two methods.

4. Macro Organized Code Generators

If the machine dependent parts of the node type related programs are implemented as Macros, then each distinct occurrence of a set up parameters step followed by a call to the Object Code Output primitive, as presented in Section 3, is replaced by a specific Macro call. Thus, this method requires the creation of (generally) as many Macro functions as there are distinct invocations of the Object Code Output primitive in the direct code form of the code generator.

As far as the performance of a compiler using a Macro organized code generator, as compared with a compiler using a Directly Programmed Code Generator, there should be virtually no difference, since the resulting object forms of the compilers should be (almost) identical. The principal disadvantage of the Macro method is that the compiler implementor has to code in a Macro language to define the expansions of the Macros, and to run a Macro Processor to obtain the object form of a compiler. These additional requirements can result in increased debugging time and increased computer costs in developing the compiler.

The principal advantage is simply that all the machine dependent parts of the compiler are isolated into a file of Macro definitions. This can greatly facilitate retargeting the compiler.

5. Table Driven Code Generators

In a table driven code generator, the machine dependent parts of a node type related program are each implemented as a call to a table driven interpreter. That is, for each sequence of set up parameters, call Object Code Output primitive pairs, as presented in Section 3, which occurs in a distinct flow path of the machine dependent part, there is a single call to the interpreter. This interpreter uses the state information set up by the machine dependent part to refernce a fixed table. The entries in this table will be the required data to generate the appropriate sequence of output lines.

96

In principle, there might be some space/time trade-offs in the performance of a compiler using a table driven code generator, as compared with the other two methods. However, in practice, this choice of method should not result in a significant variation in performance. The principal advantage of the table driven code generator is that it is an alternative method for isolating the machine dependent part of the compiler to the Macro method. The Macro method uses a file of Macros definitions to represent the machine dependent requirements, and the table driven method uses a table. Consequently, one should consider only the comparison of the table driven and Macro methods. For table driven code generators, it is somewhat more difficult to plan in advance the form of Macro calls required for a variety of target computers with very different hardware architectures. However, in comparing the advantages and disadvantages of these two methods, the most general conclusion would likely be that the evaluation depends on the point of view of the implementor -- he will most likely prefer the method with which he has had more experience.

CHAPTER 8

HOW TO COMPARE COMPILERS IN THE SAME ENVIRONMENT

1. Introduction

In this chapter, a detailed discussion is presented of the results of our investigation of the same environment question:

> How can two compilers with the same features
> and operating in the same environment be
> compared?

This chapter will include:

- A discussion of each step taken in the investigation the same environment question.

- A discussion of desirable related work beyond the scope of the present study.

- A discussion of how the results of the present study, together with the indicated related work could be used to assign a dollar value to the differences in performance between two compilers due to differences in the compilers' architecture and algorithms.

Steps taken in this study. The following is a list of the steps taken in the study of the same environment question.

- A list of language elements was prepared for use in generating User Profiles. (See Section 5.)

- For each of five categories of language elements, methods were established for generating test programs to be used in evaluating compiler performance with respect to the language elements in the category. These methods are discussed in Chapter 9.

Each of these two steps will be discussed in further detail in Section 2.

Related work. The "desirable related work" to be discussed in Section 3 consists of:

- Investigating methods for collecting data to be used for generating static and dynamic User Profiles.

- Investigating methods of reducing such collected data by automatic means so as to facilitate the generation of static and dynamic User Profiles.

- Prepare test programs by the methods described in Chapter 9 and use these programs to generate Compiler Performance Profiles.

- Use User Profiles to establish "Gibson mix" for users.

Using the results. In Section 4, a discussion is presented of how the results of the investigation of the same environment question can be used to assign a dollar value various aspects of using a compiler.

2. Steps of the Study of the Same Environment Question

For each of the two steps taken in the study of the same environment question, a discussion is presented below. This discussion includes a description of the objectives for the step and the techniques used to fulfill the objectives.

Create list of language elements. The objectives that the list was intended to fulfill are listed below:

- To be as complete as possible (within the budgetary constraints of the project).

- To avoid an organization that will result in one element representing a very large fraction of language forms in users' programs.

- To have the list represent language elements which are not only useful for describing the manner in which high level language is used, but also those about which statistics comprising a User Profile can be collected by some relatively simple means, for example, instrumenting a compiler.

The techniques used to generate a list of language elements fulfilling the above objectives are listed below.

- Review papers and reports which discuss the relative frequency of occurrences of high level language syntactic forms.

- Organize the list into a number of convenient categories and sub-categories, so that very rarely occurring cases can be grouped together statistically.

- Break down highly used syntactic forms into a number of separate cases in order to have a more detailed characterization of a User Profile. Hopefully, the level of detail should be such that no single language element (syntactic form) should represent more than approximately ten percent of the total cases.

Generating methods for preparing test programs. The objects
to be satisfied by the methods for preparing test programs are listed
below.

- Isolate the effect of the language element for which the
  test programs are generated.

- Keep the number of test programs required to measure
  performance with respect to a single language to as
  few as possible.

- Keep each test program as simple structurally as
  possible. The generation of test programs could
  possibly become automated provided they are kept
  structurally simple.

- The methods should be easy to understand and to put
  into practice.

- The procedure to calculate a performance measure for
  the language element in terms of data collected from
  compiling and executing the test programs should be
  simple and straightforward.

The techniques used in developing the desired methods for
generating test programs are listed below. These techniques are
discussed in more detail in Chapter 9.

- Define a test program which can be used to measure a
  reference or base value for the language element.
  This base program will not use the language element
  under consideration. Then establish a measurement
  for 1 (or n) occurrences of the language element.
  The desired performance measurement is found by
  taking the difference (divided by n) of the measured
  performance between the second and base test
  programs.

- Establish a sequence of test programs containing n_1,
  n_2, ..., n_m occurrences of the language feature
  under consideration. The measured performance
  values for each test program are plotted as functions
  of n_k. The slopes calculated for the least mean
  square line fitting the test data will be used as the
  calculated performance values for the language
  feature.

- Same as the preceding, except that the intercept for $n = 0$
  is calculated. This is compared with the base performance
  value as determined above. If very different, this
  indicates gross non-linearity of performance with
  number of occurrences of the language element.

100

3. Desireable Related Work

There are a number of important problems related to the evaluation of compiler performance that are beyond the scope of the present study. Several such problems are discussed below.

Collecting data for User Profiles. There are several obvious methods which might be useful for collecting the data in terms of which a User Profile may be generated. Listed below are several methods that are worthwhile to investigate concerning the possibility that User Profile data can be collected cheaply by automatic methods.

Examples of methods which might be investigated for collecting static User Profile data are the following:

- Build instruments into selected modules of a compiler which can count the occurrences of syntactic forms processed by the module.

- For modules which are used for a single syntactic form, a trace mechanism could count the occurrences of calls to this module.

- Build simple finite state machine procedures to count the occurrences of specific language elements.

Examples of methods which might be investigated for collecting dynamic User Profile data are the following:

- Build into the code generator of a compiler the changes required so that each statement (or branch of a conditional) when executed causes a counter to be incremented. Also build instruments to collect static data statement by statement. The counts derived during execution can be used as weights to combine with statement level static data to produce a dynamic User Profile.

- Build into the compiler changes required to count each execution of a block of code. Also build instruments to collect static profile data for each such block. The counts and block profile data can be combined as above.

One further aspect of the language elements for User Profiles deserves special attention. It is obvious that executable statements nested within FOR loops will contribute by some multiplicity to the time of execution of the object code generated by the compiler. However,

101

such a statement still should contribute (approximately) the same as a
similar statement not in any FOR loop to the compilation time of a
program. If one is only interested in compiler execution performance,
then no effort is required to calculate or estimate an appropriate object
code execution multiplicity factor for statements within FOR loops. If
object code execution performance is of interest then it is necessary to
make such a calculation or estimate.

The only way a completely accurate multiplicity factor can be
determined is by collecting dynamically execution counts for contiguous
sequences of executable code as suggested above. It is easy to see that
the results of such an instrumentation method would readily permit a
highly accurate dynamic User Profile to be established.

Failing the availability of the above instrumentation in a compiler,
the list of language elements for User Profiles presented in Section 5
include the following:

* A histogram of the number of occurrences of nested
  FOR loops of depth $\underline{N}$.

* Histograms of the number of executable source
  language forms nested $\underline{N}$ levels deep within FOR
  loops.

At the present time no specific recommendation is made of how these static
data should be used to estimate the desired multiplicity factors. However,
as experience accumulates in collecting these data for actual static User
Profiles, methods for estimating the multiplicity factors should suggest
themselves.

Automatic generation of User Profiles. If the raw User Profile
can be collected automatically by one or more of the above methods,
the data could be stored in one or more files in a form suitable for later
processing. Then, the raw data as contained in the file could be
subsequently processed to calculate for each language element the
following:

* The average number of static occurrences of the element
  in a program comprising the user's environment.

102

- The average number of executions of the language element in the user's environment. (This average should take into account the relative frequency with which different programs or blocks are executed.)

Generate actual Compiler Performance Profiles. Up to this point, all the preceding discussion indicated how User Profiles might be generated for the language elements listed in Section 5. These suggestions are based solely on theoretical considerations. In order to determine the usefulness of the methods described in this chapter, actual Compiler Performance Profiles should be generated by the methods described in Chapter 9. Only from the accumulation of experience with actual Compiler Performance Profiles, can valuable insights be gained as to what changes are desirable in the language elements used for defining both User Profiles and Compiler Profiles, and ultimately, the performance values that are appropriate to assign to a particular compiler in a particular user application environment.

Other uses of a User Profile. Besides combining User and Compiler Performance Profiles, a User Profile might have other uses. One such other use that might be worth investigating is the possibility of using a User Profile as the basis for establishing a "Gibson mix" definition of the user's applications environment. This "user Gibson mix" would be used to evaluate the relative speed and size characteristics of different computers and operating systems for supporting the user's application programs to be executed.

4. Using the Results to Calculate Dollar Valuations of Compilers in the Same Environment

Introduction. It is assumed for the purposes of this section that one has compiled a User Profile by the methods discussed in this chapter, and that one has also written, compiled, and executed test programs by the methods described in Chapter 9 to obtain a Compiler Performance Profile. These results are then used as follows:

- Combine the User Profile and Compiler Performance Profile to obtain four figure of merit evaluations related to a "typical" user program. These evaluations constitute the Compiler Evaluation Profile.

103

- Combine these figures of merit with additional administrative data to calculate useful dollar valuations of the compiler.

The method to be used to combine the User Profile and Compiler Performance Profile is described below, together with an interpretation of the four resulting figures of merit. Finally, the calculation of dollar valuations is discussed.

Combining profiles. Ultimately, the combination of the User Profile and Compiler Performance Profile should give a figure of merit for each of four criteria for a "typical" user program. The four criteria are:

- Time to compile.
- Partition space required.
- Time to execute.
- Size of executable program.

The time to compile a "typical" user program is calculated in a simple, straightforward manner. For those elements whose time to compile performance data indicate a linear performance curve, simply multiply the Compiler Performance Profile time to compile coefficient by the number of occurrences indicated in the corresponding static User Profile element. For non-linear performance cases, the characterization of the non-linear curve is combined with the number of occurrences indicated. The grand total for all the language elements is the desired time to compile a "typical" user program.

The partition space required to compile a "typical" user program could be calculated in the same manner as above. However, it is likely that the average or typical space requirement is not as useful as, say, the ninetieth, ninety-fifth, and ninety-ninth percentile data. These later figures would indicate the space required to compile 90%, 95%, 99%, etc. of the user's programs. By means of these figures and knowledge of the actual quantity of space available, one could interpolate to determine approximately the fraction of user programs that could be

104

compiled in the available space. Presumably, the remaining fraction of programs would have to be reprogrammed into smaller programs in order for them to be compiled. The manner in which these percentile figures might be determined is another problem involving User Profiles that might be profitably investigated in a separate study.

The time to execute a "typical" user program would be determined in the same manner as the time to compile, except the dynamic User Profile data would be used instead of the static User Profile.

The size of executable program for the "typical" user program would also be determined from the dynamic User Profile combined with the Compiler Performance Profile, except that for each language element the "size of object program as assembled" data would be used in the Compiler Performance Profile.

Dollar Valuations. If we assume that the User Profile and Compiler Performance Profile data have been combined, then the results would include four figures of merit as follows:

- Time to compile a "typical" user program.
- Either:
  - Partition space required to compile a "typical" user program,

  Or:

  - Fraction of user programs that cannot be compiled in the available space.
- Time to execute a "typical" user program.
- Size of "typical" user program.

Now assume that the following additional administrative data is also available:

- Number of compilations expected annually (or for life of computer/compiler).
- Number of program executions expected annually (or for life of computer/compiler).
- Cost of unit of time of computer execution.
- Cost of unit of partition space.

- Cost of reprogramming a typical large user program.
- Operating system overhead.

The following dollar figures could then be calculated:

- Annual (or lifetime) dollar costs for compiling user programs.
- Annual (or lifetime) dollar costs for executing user programs.
- Expected cost (if any) for reprogramming existing user programs so that they can be compiled in the available space.

These dollar figures can be used in conjunction with procurement cost of the compilers under consideration to achieve the best cost/benefit result.

5. **A List of Language Elements to be Used for Generating a User Profile**

Introduction. This section presents a list of source language elements with respect to which a compiler's performance can be measured. Included are features which are relevant to many programming languages, but most importantly the AED language (since AED is representative of a good systems development language) and also to the JOVIAL/J3B language (since J3B is a representative of a user oriented language).

The list is organized into the following five categories:

- Lexical elements.
- Declarative elements.
- Scope definition elements.
- Program control elements.
- Data manipulation and compulational elements.

For each of these categories, an outline organized sub-section of language elements is presented. The listed elements are described in terms of the statistic that would be collected to represent this element in a User Profile. For each element, Chapter 9 presents a description of how test programs could be prepared to test compiler and object code performance with respect to the element. In combination, the results of accumulating data from these test compilation and object code executions comprise a Compiler Performance Profile.

I. Lexical elements.

    A.   Features involving lines, statements, characters applicable to comments, declarative, and executable text. (Features to be measured for each of comments, declaratives, and executable text separately.)

        1.   Number of Source Lines.

        2.   Number of Source Statements.

        3.   Number of Blank Lines.

        4.   Number of Blank Characters.

        5.   Number of Non-blank Characters.

        6.   Number of Blank Sequences of Length n (Histogram).

    B.   Token features applicable to declarative and executable text. (Measures to be made for declarative and executable text separately.)

        1.   Punctuation (number of references to each).

        2.   Keywords (number of references to each).

        3.   User declared tokens of Length n (Histogram of number. of references to token of length n).

    C.   Number of .INSERT (AED)/COMPOOL (J3B) Statements. (Note: These statements cause the compiler to editorially insert possibly large files into the compiler input stream. What is desired is to measure the compiler's performance in making this insertion as a function of file size.)

II. Declarative elements.   (Note: A through D constitute independent dimensions of element classes. That is, each element has an attribute in each category A through D. However, not all combinations in the Cartesian produce space of AxBxCxD are meaningful. Category E are special declarative elements that do not have all the attributes of A through D. Statics are to be collected for both the number of declarations of each meaningful combination, and the number of references to a variable with such a declaration.)

107

A. Scope and Storage Class.

    1. Static internal.

        a. Number declared at Scoping level n (Histogram).

        b. Number at level n replacing previously declaration for token at lower (numerical) level.

    2. Static external.

    3. Automatic (internal).

        a. Number declared at Scoping level n (Histogram).

        b. Number at level n replacing previously declaration for token at lower (numerical) level.

    4. "Based" (Beads and Bead Components in AED only).

    5. Parameters.

        a. By-value type*.

        b. By-reference type.

B. Structure Type.

    1. Scalar (That is: a simple variable).

    2. Array.

        a. 1-dimensional (AED and J3B).

        b. 2-dimensional (J3B only).

    3. Table (J3B only) (Histogram over number of items in Table) (Note: Items within the table have data types; the table itself does not.)

    4. Table Item (J3B only).

        a. Bit.

        b. Byte.

        c. Half word.

        d. Full word.

        e. Other.

    5. Bead (AED only) (Histogram over number of components in Bead) (Note: Components within a Bead have data types; the Bead itself does not.)

    6. Bead Component (AED only).

        a. Bit.

        b. Byte.

        c. Half word.

        d. Full word.

        e. Other.

*Note: Current versions of AED and J3B do not have by-value type parameters. However, they are planned for future versions.

C.   Data Type.

     1.   Integer of length $\underline{n}$ (Histogram). (In AED also used for Bit operations.)

     2.   Real.

     3.   Character String of length $\underline{n}$ (Histogram in J3B only -- in AED character string variables are always variable length.)

     4.   Bit String of length $\underline{n}$ (Histogram). (J3B only)

     5.   Boolean. (AED only)

     6.   Pointer. (AED only)

     7.   Label. (Parameter in AED only or in AED and J3B as a component of a switch declaration which is, in effect, an Array of Label variables. For switch declarations, a histogram should be collected of the number of switches having $\underline{n}$ labels.)

     8.   Procedure. (Parameter in AED only)

D.   Initial Values.

(Note: What is desired here is to determine the compiler performance (and the performance of the executable code) when declaring (and referencing) variables with initial values. For each data type, certain initial values are of particular interest since they are likely to occur with high frequency, and a compiler might be designed to handle these values with greater efficiency.)

     1.   Integer.

          a.   0 (Zero). (Note: Default initialization)

          b.   1.

          c.   Other.

     2.   Real.

          a.   0 (Zero). (Note: Default initialization)

          b.   1.

          c.   Other.

3. Character String.

   a.   Blanks. (Note: Default initialization)

   b.   Other.

4. Bit String.

   a.   0's. (Note: Default initialization)

   b.   Other.

5. Boolean. (AED only)

   a.   FALSE. (Note: Default initialization)

   b.   TRUE.

6. Pointer.

   a.   NULL. (Note: Default initialization)

   b.   Other.

E. Procedures and Functions.

(Note: In AED, internal procedures are not required to be declared. In J3B, an internal procedure must be declared before it is CALLED. Function declarations are "valued-procedures and have a data type (See C above). It is desired to measure the compilers performance for each type of procedure/function declaration. Histograms for each type are to be collected over the number of parameters of type by-value* or by-reference.)

1. Number of internal procedures with n parameters.

2. Number of internal procedures with n by-value parameters.

3. Number of internal procedures with n by-reference parameters.

4. Number of external procedures with n parameters.

5. Number of external procedures with n by-value parameters.

6. Number of external procedures with n by-reference parameters.

---

* See previous footnote for by-value parameters.

7. Number of internal functions with $\underline{n}$ parameters.

8. Number of internal functions with $\underline{n}$ by-value parameters.

9. Number of internal functions with $\underline{n}$ by-reference parameters.

10. Number of external functions with $\underline{n}$ parameters.

11. Number of external functions with $\underline{n}$ by-value parameters.

12. Number of external functions with $\underline{n}$ by-reference parameters.

III. Scope definition elements.

A. Number of DEFINE PROCEDURE...END statement pairs.

B. Number of BEGIN...END statement pairs. (Note: BEGIN is a scoping boundary in AED but not in J3B.)

C. FINI (AED)/TERM (J3B) -- single statement acting as logical end-of-file.

D. Maximum number of declaration levels (Note: Maximum at most 2 in J3B).

E. Histograms for n-th level of scoping* (The zero-th level is outside any procedure or BEGIN-END block - For each procedure or BEGIN-END block the level is incremented. by 1.)

1. Number of declaration statements at level $\underline{n}$.

2. Number of executable statements at level $\underline{n}$.

3. Number of user tokens declared at level $\underline{n}$.

4. Number of user tokens declared at level $\underline{n}$ which are already declared at a lower (numerically) level.

5. Number of BEGIN-END blocks initiating a new block at a level $\underline{n}$ (AED only).

6. Number of Procedure Definitions initiating a new block at level $\underline{n}$.

7. Number of labels defined at level $\underline{n}$.

---

*These histograms only meaningful for AED since J3B does not permit more than levels 0 and 1.

111

IV. Program control elements.

A. IF Constructions.

(Form of IF is:

IF <u>X</u> THEN S1 [ELSE S2]

<u>X</u> represents a Boolean (AED) or Bit (J3B) variable or expression.
Thus, <u>X</u> may be a variable, a primitive expression, or a com-
pound expression. It is desired to determine the distribution of
<u>X</u> forms among these three categories. The various detailed
forms of <u>X</u> are outlined in Section V under·Boolean Expressions.
S1 and S2 represent statements. It is desired to consider the
configuration of particular statement forms for S1 and S2
separately.)

1. General Form of <u>X</u>.

    a. Boolean (AED) or Bit (J3B) variable.

    b. Simple Predicate Form.
       (A predicate B. Predicates are =,¬=, >, <, >=, <=)

    c. Compound Boolean Expression.

112

2. Forms of S1.

    a. BEGIN.

    b. CALL.

    c. FOR.

    d. IF.

    e. GOTO.

    f. Assignment.

3. Forms of S2.

    a. BEGIN.

    b. CALL.

    c. FOR.

    d. IF.

    e. GOTO.

    f. Assignment.

B. FOR Constructions.

1. General Forms of FOR Statements.

    a1. FOR I=E Step F UNTIL G DO S (AED)

    a2. FOR I (E BY F WHILE X); S   (J3B)

    (Note: a2 more general than a1. If $\underline{X}$ has the form $I_{\neg}=G$, then these forms are equivalent.)

    b1. FOR I=E WHILE X DO S (AED)

    b2. FOR I (E WHILE X); S   (J3B)

    c. FOR I=E_1, E_2, ..., En DO S (AED)

    d. FOR I=I/Q_1/, /Q_2/, ..., /Qn/ DO S (AED)
       where:
          . n>1,
          . /Qi/ is either of the form Ei or of the form Ei STEP Fi until Gi, and
          . at least one /Qi/ is not of the form Ei.

    e1. WHILE X DO S (AED)

    e2. WHILE X;S   (J3B)

It is desired to determine the relative occurrence of each form, including histograms over $\underline{n}$ for forms c and d.

113

2. Forms of I.

   a. Simple Integer.

   (Note: The following are allowed only for AED)

   b. Subscripted Integer Array Element.

   c. Integer Bead Component.

   d. Simple Real Variable.

   e. Subscripted Real Array Element.

   f. Real Bead Component.

   g. Simple Pointer Variable.

   h. Subscripted Pointer Array Element.

   i. Pointer Bead Component.

   j. Simple Boolean Variable.

   k. Subscripted Boolean Array Element.

   l. Boolean Bead Component.

3. General Form of E (or Ei), F (or Fi), or G (or Gi).

   (Statistics for three forms to be collected separately)

   a. Single literal, or non-subscripted variable with the same data type as I. (AED or J3B - in J3B must be integer data type.)

   b. Other (AED or J3B - in J3B must be an integer expression.)

   (Note: Other forms are treated in detail as arithmetic or Boolean expressions in Section V.)

4. Form of X.

   a. Simple Boolean (AED) or Bit (J3B) variable.

   b. Other.

   (Note: The non-simple forms of X are treated in detail as Boolean expressions in Section V.)

5. Form of S.

   a. BEGIN.

   b. CALL.

   c. FOR.

   d. IF.

   e. GOTO.

   f. Assignment.

6. Histogram of the number of FOR loops nested $\underline{n}$ level deep.

114

C. Other (simple) program control elements

    1. Label Definition.

    2. GOTO.

        a. Simple label.

        b. Switch.

V. Data manipulation and computational elements.

A. Environmental Elements.

    1. Assignment Statement.

    2. E form in FOR Statement.

    3. F form in FOR Statement.

    4. G form in FOR Statement.

    5. X form in FOR Statement.

    6. X form in IF Statement.

    7. Subscript.

    8. Argument to Procedure or Function Call.

B. Nesting Histograms.

    1. Number of simple variable or literal references occurring at a FOR loop nesting depth of n.

    2. Number of references to parameters at nesting depth of n.

    3. Number of references to external data at nesting depth of n.

    4. Number of references to global internal data at nesting depth of n.

    5. Number of references to local static data at reference depth of n.

    6 Number of references to local automatic data at reference depth of n.

    7. Number of reference to automatic data at reference depth of n which is declared in a containing Begin or Procedure Definition block (AED only).

    8. Number of single subscripted array references at nesting depth of n.

    9. Number of double subscripted array references at nesting depth of n. (AED only)

10. Number of Bead component (AED) or Table Items (J3B) references at nesting depth of $\underline{n}$.

11. Number of arithmetic operations at nesting depth of $\underline{n}$.

    a. +

    b. -

    c. *

    d. /

12. Number of data type conversions at nesting depth of $\underline{n}$.

    a. To integer.

    b. To real.

    c. To pointer.

13. Number of function calls at nesting depth of $\underline{n}$.

    a. Functions with no arguments.

    b. Functions with single argument.

    c. Functions with two arguments.

    d. Functions with more than two arguments.

14. Number of procedure calls at nesting depth of $\underline{n}$.

    a. Procedures with no arguments.

    b. Procedures with single argument.

    c. Procedure with two arguments.

    d. Procedures with more than two arguments.

15. Total number of arguments to called functions or procedures at nesting dept of $\underline{n}$.

16. Number of assignment statements at nesting depth of $\underline{n}$.

17. Number of IF--THEN's at nesting depth of $\underline{n}$.

18. Number of ELSE's at nesting depth of $\underline{n}$.

19. Number of GOTO's at nesting depth of $\underline{n}$

    a. To label.

    b. To switch.

C. Form of Left Hand Side of Assignment Statement.

(Note: Details of expression forms used as subscripts are handled as arithmetic expressions.)

1. Simple variable.

2. Single subscripted variable with literal subscript.

116

3. Single subscripted variable with variable subscript.

4. Single subscripted variable with expression subscript.

(Note: 5-10 valid in AED only.)

5. Double subscripted variable with two literal subscripts.

6. Double subscripted variable with one literal and one variable subscript.

7. Double subscripted variable with two variable subscripts.

8. Double subscripted variable with one literal and one expression subscript.

9. Double subscripted variable with one variable and one expression subscript.

10. Double subscripted variable with two expression subscripts.

11. Bead component (AED) or Table Item (J3B).

D. Forms of Arithmetic Expressions.

1. Literal. (Special values depending on Data Type)

   a. 0 (integer, real).

   b. 1 (integer, real).

   c. TRUE. (Boolean - AED only)

   d. FALSE. (Boolean - AED only)

   e. NULL. (Pointer)

   f. 0's. (bit string - J3B only)

   g. Blanks (character string - J3B only - this case requires length of Blank string literal to equal the length of LHS variable in assignment statement).

   h. Other.

2. Variable (including subscripting, parameter, Bead component, table item).

   (Note: character of variables within reference and data type conversions are handled in detail in the histograms above -- see V. B. 12)

117

3. Function call.

4. Variable OPERATOR Literal.

    a. A+1.

    b. A-2.

    c. A*2.

    d. A/2.

    e. A**2.

    f. A OPERATOR Other Literal.

5. Literal OPERATOR A.

    a. 1+A.

    b. 1-A.

    c. 2*A.

    d. 1/A.

    e. e**A (e = 2.71828...).

    f. Other Literal OPERATOR A.

6. Variable OPERATOR Varaible.

    a. A+B.

    b. A-B.

    c. A*B.

    d. A/B.

    e. A**B.

7. Other expressions.

    a. Number of +'s.

    b. Number of -'s.

(Ignore unitary +) (Note occurrences of unitary minus separately from binary operator).

    c. Number of *'s.

    d. Number of /'s.

    e. Number of **'s.

    f. Number of Literals.

    g. Number of Variables.

    h. Number of function calls

    i. Number of occurrences of ABS function.

j.   Number of occurrences of .RS. (AED) or
       SHIFTR( ) (J3B).

k.   Number of occurrences of .LS. (AED) or
       SHIFTL( ) (J3B).

l.   Number of occurrences of .A. (AED only)

m.   Number of occurrences of .V. (AED only)

n.   Number of occurrences of .N. (AED only)

8. Histograms of expressions with more than 1 operator.

     a.   Number of expressions occurring with $\underline{n}$ operands.

     b.   Number of expressions occurring with $\underline{n}$ variables.

     c.   Number of expressions occurring with $\underline{n}$ literals.

     d.   Number of expressions occurring with $\underline{n}$ function
        calls.

     e.   Number of expressions occurring with $\underline{n}$ pairs of
        parenthesis (explicit only).

     f.   Number of expressions occurring with $\underline{n}$ pairs of
        parenthesis including implicit parenthesis for
        precedence changes. (For example: a*b+c*d has
        2 implicit pairs of parenthesis: (a*b) + (c*d).)

E.   Forms of Boolean Expressions.

(Note: Boolean Forms not involving Boolean OPERATORS
are considered <u>simple</u>. All others are called <u>compound</u>.)

1.   Literal.

     a.   TRUE.

     b.   FALSE.

2.   Variable (including subscripting, parameter, bead
     component, table item).
     (Note: Character of variables within reference and
     data type conversions are handled in detail in the
     histograms under section B above.)

3. Single Predicate Form.

    a. Variable <u>PREDICATE</u> Literal.

        (1) Predicates.

            (a) EQUAL.

            (b) NOT EQUAL.

            (c) LESS THAN.

            (d) GREATER THAN.

            (e) LESS THAN OR EQUAL.

            (f) GREATER THAN OR EQUAL.

        (2) Literals. (depending on Data Type of Variable)

            (a) 0 (integer, real).

            (b) 1 (integer, real).

            (Note: Literals (c) through (g) apply only to EQUAL and NOT EQUAL.)

            (c) Blanks. (character string, J3B only)

            (d) 00...0 B. (bit string, J3B only)

            (e) NULL. (pointer)

            (f) TRUE. (Boolean (AED) or Bit (J3B))

            (g) FALSE. (Boolean (AED) or Bit (J3B))

            (h) Other Integer or Real Literals.

            (i) Other non-numeric Literals (EQUAL and NOT EQUAL Only).

    b. Variable <u>PREDICATE</u> Variable.

        (Predicates as a. (1) above)

(Note: The above categories 1 through 3 are all simple expressions. The categories to follow (4 through 6 are all compound expressions.)

4. Use of Unary Boolean NOT.

    a. Preceding single Boolean Variable.

    b. Preceding simple Predicate Form.

    c. Preceding Variable within an expression.

    d. Preceding simple Predicate Form within an expression.

    e. Preceding an expression.

    f. Other.

5.   Binary Boolean Operators.

    a.   AND.

    b.   OR.

    c.   EQUIVALENT. (AED Only)

    d.   IMPLIES. (AED Only)

    e.   EXCLUSIVE OR. (J3B Only)

6.   Use of Boolean OPERATOR.

    a.   Variable OPERATOR TRUE.

    b.   Variable OPERATOR FALSE.

    c.   TRUE OPERATOR Variable.

    d.   FALSE OPERATOR Variable.

    e.   Variable OPERATOR Variable.

    f.   Simple Predicate OPERATOR TRUE.

    g.   Simple Predicate OPERATOR FALSE.

    h.   TRUE OPERATOR Simple Predicate.

    i.   FALSE OPERATOR Simple Predicate.

    j.   Variable OPERATOR Simple Predicate.

    k.   Simple Predicate OPERATOR Variable.

    l.   Simple Predicate OPERATOR Simple Predicate.

    m.   Compound Expression OPERATOR TRUE.

    n.   Compound Expression OPERATOR FALSE.

    o.   TRUE OPERATOR Compound Expression.

    p.   FALSE OPERATOR Compound Expression.

    q.   Compound Expression OPERATOR Variable.

    r.   Variable OPERATOR Compound Expression.

    s.   Compound Expression OPERATOR Simple Predicate.

    t.   Simple Predicate OPERATOR Compound Expression.

    u.   Compound Expression OPERATOR Compound Expression.

CHAPTER 9
HOW TO WRITE TEST PROGRAMS FOR GENERATING COMPILER
PERFORMANCE PROFILES

1. Overview

This chapter first describes the generalized approach to constructing and evaluating tests. In particular, the concept of isolating separable sub-costs associated with the cost of a statement is explained. This is the major goal of the series of tests presented in Sections 2 - 6.

Five separate series of tests designed to evaluate elements which could be processed differently by two compilers are outlined. These include:

- Lexical elements.
- Declarative elements.
- Scope definitions.
- Program control elements.
- Data manipulation and computational elements.

Separability of statement sub-costs. The prime objective of constructing test programs is to isolate separate statement sub-costs. There are three sub-costs associated with the cost of a statement containing one or more sub-expressions:

(1) The cost of each expression compiled and executed in isolation.

(2) The cost of the statement form itself (invariant, regardless of the form of expression used).

(3) The residual interactive cost of each expression as used in the context of the particular type of statement form.

A major goal of the test series is to determine items (1), (2), and (3) separately for various statement forms. However, the three sub-costs are, in general, not separable. In most cases, the total cost of a statement form is determined using the simplest possible expression forms, and as many as possible separable expression costs (sub-cost (1))

122

determined and subtracted from the total. This process provides a "base cost", which is then used for comparison purposes with other forms of expressions used with the statement being tested.

For example,

IF expression 1 THEN expression 2 ;

is the IF-statement form in AED. If F( ) (subroutine call) is used as expression 2, then, since F( ) is compilable and executable in isolation, sub-cost (1) is separable by subtracting the cost of the subroutine call from the cost of the total IF-statement. However, the cost of the interaction between the THEN and the subroutine call (F( ) as used in the context of THEN) cannot be determined this simply.

Some statements can, however, be made about sub-cost (3), after tests have been run on several different types of statement forms, each using the same set of sub-expressions. Specifically, the difference between the cost of one statement form using the same set of expressions can be compared to the comparable cost using another statement form. If this difference is negligible this indicates that sub-cost (3) is negligible for all statement types examined, since the type of statement in which the expression was embedded had no apparent effect on its cost. For example, consider the statement forms:

Set 1:
>   IF expression THEN (Call)   (a)
>   IF expression THEN (Assignment)   (b)
>   IF expression THEN (Loop)   (c)

Set 2:
>   FOR expression WHILE expression DO (Call)   (d)
>   FOR expression WHILE expression DO (Assign.)   (e)
>   FOR expression WHILE expression DO (Loop)   (f)

If the total cost difference between items (b) and (a) (or (c) and (a)) is the same as the cost difference between items (e) and (d) (or (f) and (d)), then the residual interactive cost (sub-cost (3)) is negligible for calls and assignment expressions (or calls and loop expressions) as used in IF and FOR statements.

123

Approach to test construction. To determine the actual cost of various language constructs, each test series in the total set begins with a skeleton program which represents a simple (minimal) set of statements. This is intended to allow as many overhead costs as possible to be subtracted from the observed test figures before plotting and analyzing them. The skeleton program contains just the source statements not related to the test to be performed (statements necessary to make the test program compilable or executable), plus all sub-expressions which are compilable in isolation (sub-cost 1). The skeleton test is compiled (and executed where necessary) in order to obtain a base cost figure to be subtracted from the results of the various test runs.

Succeeding test programs add or modify the basic setup, and performance figures are measured against the base values. It is anticipated that most (if not all) such comparison figures will be an incremental amount larger than the base figures, and that this increment is a measure of the difference between the test program and the base. In some instances, the test data values may be smaller than the base, showing that the base is not a true minimal case, but the performance measurement comparisons are still valid even in this unusual case.

Each set of tests is written as a series of programs in which the feature to be tested is repeated many times. The first test program in the set repeats the feature LOW times, the next program adds STEP repetition to the previous number, and so on until HIGH repetitions are reached in the final program in the test series. The integer val·es assigned to LOW, STEP, and HIGH depend upon the software and hardware measurement facilities available on the particular computer and operating system being used for the 'ests.

Evaluating test series results. After subtracting the base value from the test figures, the results of the test runs are plotted, and a slope which graphically shows the cost of the statement of interest is determined. The slope of this graph represents the basic cost of the feature, and the shape of the graph shows whether or not a fixed cost may be associated with each occurrence of the feature. If the

graph is linear, the cost for each occurrence of the form being tested
is a fixed amount; if the graph is non-linear, the cost is effected by the
number of repetitions which preceded it in the source program.

In the present study we have made the working hypothesis that
all such graphs should be of the linear type. If the results of the test
series are seen to be inconsistent with this hypothesis, then further study
would be required to determine the reasons underlying the failure of
our hypothesis of linearity.

Categories of efficiency. There are four basic compiler evaluation
efficiency measures of interest, broken down into two classes:

Class 1: Compiler Operation.

Measure 1: Computer Time Efficiency. (Processor Time)

Measure 2: Computer Memory Usage. (Core, Drums,
Disks, etc.)

Class 2: Resulting Object Code.

Measure 3: Execution Speed.

Measure 4: Execution Memory Usage.

In Measure 4, this figure includes the actual object code size as well as
additional space required for library support called in as a direct
result of the source code statements used (number conversions, sub-
routine call linkage mechanisms, etc.).

These four measurements can be taken either on "typical programs",
which use a mix of statements and operations typical of the application
for which the compiler is to be used, or on individual statement and
phrase forms, which require many more test programs, but which are
independent of the specific application. The test programs discussed
in this chapter are of the latter form; i.e., they test efficiency of
individual statement forms without regard for the relative importance
of each form or the relevant compiler application. While most of the
tests specified in this chapter concern primarily Measures 1 and 3,
the method presented is also directly applicable to Measures 2 and 4.

2. Lexical Elements

Introduction. The class of lexical elements tests exercises the lexical and table look-up functional elements of the compiler as well as input processing support in the CPU. These tests are designed to measure compile time efficiency and execution time efficiency in the following areas:

- Formatting features -- number of source lines, source statements, blank lines, blank and non-blank characters, lengths of blank character sequences.
- Punctuation, keywords, and tokens.
- External program elements.

Some tests are designed to be compiled only; others both compiled and executed. This difference is discussed with each set of tests. Also, skeleton programs resulting in base values to be subtracted from the test run figures are discussed with each set of tests.

Wherever appropriate, the tests are performed separately on three different types of statements:

- Comments.
- Non-executable declaration statements.
- Executable statements.

These three distinctions are not necessarily those known to actually represent differences in performance among different compilers (or different implementations of the same compiler) but, rather, are intended to provide a complete set of tests for lexical elements. The structure of the tests is simple enough to permit them to be generated by a program, if desired.

Basic overhead costs. A base or skeleton program is constructed and run to determine a "base cost" figure to be subtracted from the results of the various test runs. This program contains only those source statements which are necessary to make the program compilable (or executable, where noted) and which are not related to the test to be performed. The particular lexical elements tests consist of a series

126

of programs repeating the statement form to be tested. The statement is repeated LOW times, and the number of repetitions is increased by STEP until HIGH number of statements is processed.

The skeleton program for the tests which are compiled only in AED, takes the form:

```
BEGIN
    INTEGER DUMMY ;
    EXTERNAL DUMMY ;
    ---------------
END FINI
```

The dashed lines indicate the position where the additional test program statements are to be added.

Formatting features. This set of tests is designed to measure the overhead costs of formatting features:

- Number of source lines.
- Number of source statements.
- Number of blank lines.
- Number of blank and non-blank characters.
- Lengths of blank character sequences.

Each formatting feature is tested on comments, declaration statements, and executable statements. Some of these tests serve as skeleton tests for other features and are discussed where applicable.

Number of source lines. To evaluate the compiler's performance with a varying number of source lines, construct three separate sets of tests to process:

- Comments.
- Declaration statements.
- Executable statements.

First, examine the effect of a varying number of source lines in a comment. Construct a non-null comment and add it to the skeleton program. In AED comments take two forms:

1. COMMENT xxxxxxx ;

2. ... xxxxxxx //

The first of these two forms is referred to as a comment, the second as a remark. They represent however, equivalent ways to express the character string xxxxxxx as a piece of non-processing, descriptive text within an AED program.

After compiling the test program first with a one-line comment and then with a one-line remark and recording the processing time, then increase the length of each so that it extends over more than one line. Repeat this process until the desired number of tests has been compiled (by STEP until HIGH number is reached). After subtracting off the time required to compile the skeleton program, the results of these test runs can be plotted and analyzed.

Next, construct a representative set of declaration statements which are formatted one statement per source line, and add the set to the skeleton program. In AED, this could take the form:

INTEGER A ;

POINTER ARRAY B ;

BOOLEAN C ;

Again, compile the test program with the set of declaration statements, and record the processing time. Add replications of the same set, recompile, and record the processing time (by STEP until HIGH number is reached). After subtracting off the time required to compile the skeleton program, the results of these test runs can be plotted and analyzed.

Lastly, to evaluate the compiler's performance with respect to the number of source lines, construct a representative set of executable statements which are formatted one statement per source line, and add it to the skeleton program. Include a declaration of each different

128

variable contained in an executable statement. Perform the same repetitive process as just described and, again, plot the results. Note that even though the set of executable statements is repeated, the declaration statements need appear only once in the test program.

Number of source statements. To evaluate the compiler's performance with a varying number of source statements, construct three separate sets of tests to process:

- Comments.
- Declaration statements.
- Executable statements.

The tests for this series are designed so as to make the effects due to the number of lines as close as possible to those measured in the previous set, so that they may be factored out from the observed compilation times.

First, to examine the effects of a varying the number of comment source statements, construct a single source line which consists of as many comments (AED comments, remarks, or both) as will fit on the one line. Add the line to the skeleton program, compile it, and record the processing time. Again, repeat the process by increasing the number of replications of the commented line, and record the results. Alternatively, start with the largest "single comment" as used for the first set of source line tests. Gradually break it into successively smaller comments until each is packed at its maximum density. Again, record the results.

Next, examine the effects of a varying number of declaration source statements. Construct a representative set of declaration statements packed together as closely as possible. This may include several different declaration statements per line as, for example:

INTEGER A1, A2, A3 ; POINTER B1, B2, B3 ; INTEGER ARRAY
  C1, C2, C3 ;
BOOLEAN D1, D2, D3, D4 ; REAL E1, E2, E3, E4 ;

129

Add the set to the skeleton program, compile it, and record the processing time. Again, repeat the process by increasing the number of replications of the set of declarations and record the results.

Lastly, examining the effects of a varying the number of executable statements. Replace the set of tested features in the previous test series with a representative set of executable statements packed together as closely as possible. Perform the same repetitive testing process. Note that even though the set of executable statements is repeated, the declaration statements need appear only once in the test program

Number of blank lines. To evaluate the compiler's performance with a varying number of blank lines, construct four separate sets of tests to process blank lines occurring:

- Between statements.
- In comments.
- In declaration statements.
- In executable statements.

First, to examine the effects of a varying number of blank lines between statements (null statements), add LOW number of blank lines to the skeleton program. Compile it, subtract off the time required to compile only the skeleton program, and plot the results. Then add STEP number of additional blank lines, and repeat the process until HIGH number of replications is reached. The results which are plotted should indicate the effects of blank lines on a compiler's performance.

The last test program in this set will be referred to as an "empty sheet" program and will be used in other tests.

To examine the effects of a varying number of blank lines in comments, construct the same series of tests as described for a varying number of source comment lines but include only null comments. Repeat the process and record the results.

130

To examine the effects of a varying number of blank lines in declaration statements, add a single declaration statement to the skeleton program, compile it, and record the results. Then repetitively stretch the comment over several lines and repeat the process. For example:

```
INTEGER A1,A2,A3 ;              ... TEST 1 //
INTEGER A1                      ... TEST 2, LINE 1 //
  , A2,        A3               ... TEST 2, LINE 2 //
  ;                             ... TEST 2, LINE 3 //
```

Lastly, to examine the effects of a varying number of blank lines in executable statements, repeat the "stretched-out" process just described but replace the declaration statement with an executable statement.

Number of blank and non-blank characters. To determine the compiler's performance as a function of the percentage of the source language program devoted to blank and non-blank characters, construct three separate sets of tests to process the characters occurring:

- In comments.
- In declaration statements.
- In executable statements.

First, to examine the effect of a varying number of non-blank characters in comments, take the largest extended blank comment described for testing a varying number of blank lines in comments. This statement takes the form:

Line Number	Contents
1	COMMENT
2	

N	;

Consider this test to be the skeleton test. Now add a group of non-blank characters to the test, compile, and record the results. Continue this process, adding groups of non-blank characters until the desired number of repetitions is reached. If the language places an upper limit on the

131

number of non-blank characters in a string within a comment, start a
new string as soon as the maximum length is reached. The results of
the skeleton test should be subtracted from each test in this set and the
results plotted.

Next, examine the effect of a varying number of blank characters
in declaration statements by taking one of the densely-packed tests for
the number of declaration source statements as the skeleton program.
Now add a single blank at every position of the program at which inclusion
is permissible, compile the program, subtract off the time required to
compile the skeleton program, and record the results. Then repeat
this process, adding another group of blanks at the same places until
the desired number of replications is reached (STEP until HIGH), compile
the programs, subtract off the time to compile the skeleton program,
and plot the results. Drop statements at the end as necessary to maintain
the program size at the same number lines for each test.

Lastly, to examine the effects of blank characters in executable
statements, replace the declaration statement tests just described with
executable statements and perform the same testing process.

Lengths of blank character sequences. To determine the compiler's
performance with varying lengths of blank character sequences, construct
three separate sets of tests to process blank characters occurring:

- In comments.
- In declaration statements.
- In executable statements.

First, to examine the effect of a varying number of blank characters
in a comment, take the largest non-blank long comment statement tested
for the number of non-blank characters in comments and consider it to
be the skeleton test program. After each character in the comment
insert a blank, compile the program, subtract off the time to compile
the skeleton program, and plot the results. Then repeat this process,
adding another group of blanks at the same places until the desired
number of replications is reached (STEP until HIGH). Drop characters

132

from the end of the comment as needed to maintain the program size at a fixed number of lines.

Next, examine the effect of a varying number of blank characters in declaration statements, take the largest declaration source statements test program and consider it to be the skeleton test program. Place a single blank at every position of the program at which it is permissable, compile the program, subtract off the time to compile the skeleton program, and plot the results. Then, as in the comment test just described, repeat the process, adding another group of blanks at the same places until the desired number of replication is reached (STEP until HIGH). Drop statements off the end of the program as is necessary to maintain the program size at the same number of lines for each test.

Lastly, to examine the effects of blank characters in executable statements, replace the declaration statements tests just described with executable statements and perform the same testing process.

Punctuation, keywords, and tokens. This series of tests is designed to measure the performance of the compiler as a function of the number of references to specific vocabulary elements (punctuation and keywords) and to user-declared tokens of varying size. All tests in this series are to be constructed using the "empty sheet" test program earlier described. This "empty sheet" program serves as a skeleton program. It differs from the skeleton program described at the outset of this section in that the "empty sheet" contains a (generally) large, predetermined number of blank lines while the skeleton contains only the minimal amount of information necessary for compilation but no extra blank lines.

To evaluate the compiler's performance on punctuation characters, select one punctuation character, write a legal statement containing that character, and add the statement to the "empty sheet" program. Then either compile the program or compile and execute the program, depending on whether the statement is a declaration or an executable statement. After running the program, subtract off the time to run the "empty sheet" program and plot the results. Continue to perform this process

133

while increasing the number of usages of the punctuation character until the "empty sheet" is filled up. Then select another punctuation character and, again, repeat the testing process. Examples of AED punctuation characters include:

+ - * / = , $ $=$ // $/$

Next, to evaluate the compiler's performance on keywords, perform the same tests as just described but repeat usage of a keyword. Some examples of AED keywords are:

ABS
AND
COMMON
DO
POINTER
PRESET
UNTIL
WHILE

To evaluate the compiler's performance on <u>user-defined tokens</u>, construct two separate sets of tests to examine:

- Declaration statements.
- Executable statements.

For <u>user-defined tokens in declarations</u>, construct a test program consisting of declarations of symbols one character in length, compile the program, subtract off the time necessary to compile the "empty sheet" program which serves as a skeleton, and plot the results. Repeat this process, gradually increasing the symbol length to the maximum permitted by the language. Before constructing the first test, leave enought blank space at the end of the program so that as the symbol length increases, the program does not exceed the capacity of the "empty sheet."

To evaluate the effect of user-defined tokens in a program containing both declarations and executable statements, construct a test program consisting of the declaration of a symbol and a single executable statement referencing the symbol. In AED, this could take the form:

```
BEGIN
INTEGER A ;
EXTERNAL A ;
A = 2 ;
.......
END FINI
```

Compile and execute the program, subtract off the times necessary to compile and execute the "empty sheet" program which serves as a skeleton, and plot the results. Repeat the process, replicating the executable statement until the "empty sheet" is filled. Perform this test for symbols varying in length from one character to the maximum permitted by the language.

Inclusion of external program elements. This series of tests is designed to measure the performance of the compiler including external program elements. In AED, this take the form of a .INSERT file while in J3B this takes the form of a COMPOOL file. These tests evaluate:

- Number of included elements.
- Size of included element.

Construct a program containing a mix of both declaration and executable statements. In AED, this could take the form:

```
BEGIN
    INTEGER A1, A2 ;          ... LINE1 //
    EXTERNAL A1 ;             ... LINE2 //
    BOOLEAN B1 ;              ... LINE3 //
    B1 = TRUE ;               ... LINE4 //
    A1 = 4 ;                  ... LINE5 //
END FINI
```

Consider this program to be the skeleton program for this series. Compile it and record the time.

Now, to evaluate the <u>number of included elements</u>, move the
first line of the program after the BEGIN to a .INSERT file, calling it
LINE1. The test would take the appearance:

```
BEGIN
.INSERT LINE1 ;              ... LINE1 //
      EXTERNAL A1 ;          ... LINE2 //
      BOOLEAN B1 ;           ... LINE3 //
      B1 = TRUE ;            ... LINE4 //
      A1 = 4 ;               ... LINE5 //
END FINI
```

Compile the new program, subtract off the time necessary to compile
the skeleton, and plot the result. Repeat the testing process, placing
each additional line in a .INSERT file until every indented line in the
above example has .INSERT before it.

Next, evaluate the <u>size of the included element</u>. Use the same
skeleton test as above. Place the first line in a .INSERT file, calling
it LINE1 as above, for example. Then add the next line, LINE2, to the
same .INSERT file and repeat the testing process. Continue adding
successive lines to the <u>same</u> .INSERT file until all program lines ar
contained in the .INSERT file (LINE1 through LINE5) while the main
program contains only:

```
BEGIN
.INSERT LINE1 ;
END FINI
```

A similar series of tests may be constructed for the inclusion
of external elements into a J3B program by means of a COMPOOL file.
However, this series would have to conform with the J3B restrictions
that COMPOOL elements must contain only declaration statements and
that the program including them must contain only one COMPOOL
statement (it can specify several file names).

3. Declarative Elements

The class of declarative elements test exercises the declaration processing functional element of the compiler. This includes all forms of declarations as well as references to declared variables. The declaration portion includes declaration of other types of program elements such as procedures and labels. Reference tests are performed only on variables, not on references to other forms of program elements. (Procedure calls, GOTO's, etc., are tested elsewhere.)

There are four basic factors which influence the cost of declarative elements:

- **Scope and Class**: Includes local, global, or external scope; static, automatic, or "based" variable class; procedure argument scope and class.

- **Structure**: Includes single word, array, table, component, partial word component. (Switches are arrays of labels.)

- **Data Type**: Includes procedure, label, integer, real, Boolean, pointer, character types.

- **Initial Value**: Includes all forms of statements used to set the initial value of a variable (PRESET in AED).

In compilers featuring block structuring, the cost of declarations also depends upon whether or not the declarations override global declarations for the same spelling or built-in declarations. Similar costs are incurred on compilers without the block structure feature which allow re-declaration.

The tests displaying the costs of declarative elements are run on the compiler only, for declaration statements, and on both the compiler and the executable object code, for the reference tests.

Basic overhead costs. The term "Declaration Statement" as used here, includes all forms of statement used to set the four above-mentioned factors of a user-defined identifier. In any particular language, the source language used to set these factors may be contained in a single statement, or spread out among various seperate statements. The meaning of the statement may also depend upon its context. For example, the position in the source program at which a type declaration is placed

136

may determine its scope. It is the purpose of this set of tests to illustrate how each of the four basic declaration factors may be tested, using the AED source language for illustrations. Other source languages would contain similar tests.

To determine the actual cost of various constructs, each test series in the total set begins with a skeleton base program which represents a simple (minimal) set of declarations. Succeeding test programs add or modify the basic setup, and performance figures are measured against the base values. It is anticipated that most (if not all) such comparison figures will be an incremental amount larger than the base figures, and that this increment is a measure of the difference between the test program and the base. In some instances, the test data may be smaller than the base, showing that the base is not a true minimal case, but the performance measurement comparisons are still valid even in this unusual case.

Each declaration feature test is written as a series of programs in which the feature to be tested is repeated many times. The first test program in the set repeats the feature LOW times, the next program adds STEP repetition to the previous number, and so on until HIGH repetitions are reached in the final program in the test series. The integer values assigned to LOW, STEP, and HIGH depend upon the software and hardware measurement facilities available on the particular computer and operating system being used for the tests.

After subtracting the base value from the test figures, the results of the test compilations are plotted, and a slope determined. The slope of this line represents the basic cost of the feature, and the shape of the slope (linear or non-linear) shows whether or not a fixed cost may be associated with each occurrence of the feature.

Minimal base declaration program set. Construct a set of programs containing repetitions of the data type declaration statement:

INTEGER A_n ;

Include sufficient framework to permit the program to compile. In AED, this program takes the form:

```
BEGIN
INTEGER DUMMY ;
EXTERNAL DUMMY ;
INTEGER A1 ;
INTEGER A2 ;
...
INTEGER An ;
END FINI
```

Since these tests are for the compiler operation only (not object code efficiency), the program need not be executable, only compilable. The use of the variable DUMMY in the above program is intended to permit compilation, and has no other purpose.

Declaration string test. Construct a comparison set of programs in which all data types of the A's are declared in a single declaration ("INTEGER A1, A2, ... An") instead of in separate statements. Spread out the declaration list over the same number of lines as in the base program, so that input/output and lexical processing costs are essentially equivalent. Comparing the two sets of data shows the basic cost of processing the INTEGER declaration repeatedly versus a string of declarations in which the INTEGER mechanism is invoked only once and given a list of names.

Scope and class tests. Construct a series of test which is similar to the base skeleton program, but modifies the scope of the variable in the following ways:

- Insert a BEGIN - END around the list of repeated statements so that they lie within an inner block.

- Redo both test series, adding an EXTERNAL declaration statement listing all of the A's, thus increasing the scope of the variables.

138

- Construct a test series based upon the base program set, but making each declared "A" a procedure argument (e.g.,

  DEFINE PROCEDURE F(A1, A2, ..., An)

  WHERE INTEGER A1 ; INTEGER A2 ;

  ... INTEGER An TOBE ...)

- Construct a test series based on the previous set, but each "A" use automatic storage (e.g., in AED, DEFINE RECURSIVE PROCEDURE ...)

Structure tests . Construct a series of tests which show the declaration cost associated with different structures. Using the base program described above, replace the word "INTEGER" everywhere by the word "INTEGER ARRAY" and then by the word "INTEGER COMPONENT". Compile and plot these test results, after subtracting off the same base cost figures from both test series.

Redo the INTEGER COMPONENT test four times adding a packing statement for each "A", using the packing:

- Single bit.
- Byte.
- Half word.
- Other (non-standard machine sub-word configuration).

Redo the four sets of tests, replacing the multiple PACK statement by a single statement which lists all of the "A's". The comparison will show the cost of processing individual PACK statements versus processing the basic declaration data.

Lastly, redo the INTEGER COMPONENT tests with no PACK statements, but using the offset statement to assign the data to different word locations (the AED $=$ operator). First assign the component to "$=$0," then "$=$"1", and lastly to "$=$-1". It is assumed that all offsets greater than 1 will have a similar cost to the "$=$1" case.

139

Initial value tests.   In most compilers, all variables have an
initial value assigned to them when the object program is loaded into
memory in preparation for a run.   If no specific value is assigned in
the source program, a default value is used.   For example, in AED,
all integer and real variables are set to 0, all pointer variables are set
to NULL, and all Boolean variables are set to FALSE.   This series of
tests examines the compiler's performance in setting initial values
specified in the source program.

Construct a series of tests based upon the base program, but
adding the statement

PRESET An = 0 ;

for each "A" in the program.   Re-compile the tests using:

- PRESET A = 1 ; (non-zero)
- PRESET A = -1 ; (negative)
- PRESET A = 1 $/$2·; (partial word)

Other structure, scope, data type and initial value tests.   To
this point, the INTEGER data type has been tested with variations in
structure, scope, and class of variable.   After compiling the above tests,
some impression of the performance of the compiler regarding declaration
statement processing should emerge.

It does not appear obvious that these individual factors are
separable for measurement purposes.   For example, the cost of processing
identifiers which have external scope (availability outside the compilation)
may differ for various data types or structure, and the sub-costs due to
scope, class, structure, initial value and data type may not be easily
separable by source program testing alone (various internal compiler
modifications may be required to separate these costs).   Therefore, it
is recommended that various combinations of data type, scope, class,
initial value and structure be attempted and these test results compared
with previous tests to determine if patterns become evident which would

allow conclusions to be drawn regarding individual cost factors. The lists of factors under each declaration category are given at the beginning of this section, and can be used as a basis for these additional tests. The test sets should be patterned after the above INTEGER tests, using combinations of factors which illustrate compiler performance features of interest, and which are comparable to other test set results.

Referent tests. Previous tests in this series have considered various forms of declaration statements. The cost of accessing or referring to the data in various contexts is coupled to the declaration, and is considered here.

Unlike the declaration statements which are only of interest in relation to compilation costs, references to data have both a compilation and an object code execution cost. Therefore, all tests in this series are designed to be compiled and executed.

When considering how to construct a series of referent tests, it becomes clear that the costs of data references are in general not separable from the statements in which the references occur. For example in the statement "A=A+B ;", the cost of accessing the value for B is difficult to isolate. The cost of "A=A ;" can be determined by compiling this portion of the statement in isolation and subtracting the resultant cost data from the cost of the complete statement. However, this leaves the cost of "+B" which cannot be further broken down and compiled in isolation to determine the cost of the addition operator versus the cost of accessing the value of variable B.

, The series of referent lists are constructed in the form of a base program plus a series of test programs that build upon the base. Each test repeats the statement form of interest, increasing the number of repetitions with each test program, just as in the declaration series of tests discussed above. Each group of tests in the series varies some factor of the referent being tested, and a comparison of the test results measures performance as related to the base as well as related to the other tests in the series.

141

As a base program, construct a series of test programs which repeats the statement:

A=A ;

Repeat the program for LOW to HIGH repetitions of the statement. The cost of the compilation and execution of the object code of these tests provides the base figures to be subtracted from all referent tests.

The referent test programs are constructed as follows. Each test is built upon the base test by repeating a statement of the form:

A=A op (referent) ;

The tests for various referents include:

op	referent
+	integer and real literals
+	integer and real variables, arrays, and components
AND	Boolean literals
AND	Boolean variables, arrays, and components

For components, use all of the forms of packing as well as positive and negative offsets described under the declaration test set, above. Repeat all tests for the variables of the classes static, automatic, and procedure arguments (non-recursive as well as recursive).

To measure referent costs in the procedure call context, construct a test set which uses as a base, the program which repeats

A = F(B) ;

To construct the test series, replace the procedure argument B by all forms of permissable referent (integer, real, Boolean, component, procedure, etc.), including all array forms and packing and offset options. Also test the case where B is an argument of an output procedure, making this outer procedure first non-recursive and then recursive.

The output of these tests clearly includes several sub-costs, but the *difference* between comparable format tests measures changes in referent cost only, and therefore gives a measure of referent performance.

4.   Scope Definition Elements

Introduction. The class of scope definition elements tests exercises the parsing and the machine independent and dependent code generator and optimization functional elements of the compiler. These tests are designed to measure compile time efficiency and execution time efficiency in the following areas:

- Procedure definitions.
- Parallel scoping ranges.
- Compilation boundaries of multiple programs.
- Declaration levels.
- Embedded scoping ranges containing different types of statements.

Some tests are designed to be compiled only; others both compiled and executed. This difference is discussed with each set of tests. Also, skeleton programs resulting in base values to be subtracted from the test run figures are discussed with each set of tests.

All tests in this class are intended to be set up to have the same number of lines; blank lines should be inserted wherever necessary. This restriction is designed to neutralize lexical processing aspects of the compiler which are most strongly reflected as a "cost per line" of compilation. To make the restriction easy to follow, some of the tests are based on the "empty sheet" program introduced in the lexical test series.

Basic overhead costs. A base or skeleton program is constructed and run to determine a base cost figure to be subtracted from the results of the various test runs. This program contains only those source statements which are necessary to make the program compilable (or executable, where noted) and which are not related to the test to be performed. The particular scope definition elements tests consist of a

143

series of programs repeating the statement form to be tested. The
statement is repeated LOW times, and the number of repetitions is
increased by STEP until HIGH number of statements is processed.

The skeleton program for the tests which are compiled only
in AED, takes the form:

```
BEGIN
    INTEGER DUMMY ;
    EXTERNAL DUMMY ;
    -------------
END FINI
```

The dashed lines indicate the position where the additional test program
statements are to be added.

Procedure definitions. This set of tests is designed to measure
the overhead costs of a procedure definition. Construct a set of programs
containing repetitions of the AED null procedure definition:

DEFINE PROCEDURE A_n TOBE

Since, in AED, the above statement does not generate code, this set of
tests needs only to be compiled and takes the form:

```
BEGIN
    INTEGER DUMMY ;
    EXTERNAL DUMMY ;
    DEFINE PROCEDURE A1 TOBE ;
    ..........
    DEFINE PROCEDURE An TOBE ;
END FINI
```

Construct all procedure definitions tests such that each contains the same
number of lines, in conformance with the "empty sheet" concept
discussed in connection with the lexical elements tests.

Parallel scoping ranges. This set of tests is designed to measure the overhead costs of a scoping range. Construct a set of programs containing repetitions of the AED null scoping range. This feature takes the form of the following pair of statements:

BEGIN   END ;

Since, in AED, the above statements do not generate code, add repetitions of the BEGIN - END pair of statements to the compile-only skeleton program. Construct all parallel scoping range tests such that each contains the same number of lines, in conformance with the "empty sheet" concept discussed in connection with the lexical elements tests.

In certain other higher level programming languages, PL/I for example, BEGIN - END pairs of statements, in addition to procedure definitions, define the retention range for the compiler's allocation of automatic storage locations. In evaluating the performance of such a compiler, the set of tests constructed would have to be both compilable and executable, and both the compilation time and the efficiency of the generated code would have to be evaluated.

Compilation boundaries. This set of tests is designed to measure the overhead cost of splitting a single program into two or more separate compilations. Construct a set of programs containing repetitions of the AED empty program definition statements:

END FINI   BEGIN

As in the two previous sets of tests, the above statements do not generate code, add repetitions of the END FINI - BEGIN pair of statements to the compile-on , skeleton program so that it takes the form:

```
BEGIN                    ... PROGRAM A1 //
INTEGER DUMMY ;
EXTERNAL DUMMY ;
END FINI
. . . . . . . . . .
    BEGIN                ... PROGRAM An //
    INTEGER DUMMY ;
    EXTERNAL DUMMY ;
    END FINI
```

145

Construct all compilation boundaries tests such that each contains the
same number of lines, in conformance with the "empty sheet" concept
discussed in connection with the lexical elements tests.

The facility to compile a sequence of programs without reinitializing
the compiler is not present on all AED compilers. The test methods
described here is applicable to the AED compiler on the Control Data
6600 but not to AED compilers on either the Univac 1108 or the IBM 360.

Declaration levels. The two sets of tests described here are
designed to measure the overhead cost of embedded scoping ranges (or
declaration levels). The first set examines empty scoping ranges; the
second non-empty.

Construct a set of programs containing embedded repetitions of
the AED null scoping range pair of statements:

BEGIN   END ;

Again, because the above statements do not generate any code, add
repetitions of the BEGIN - END pair of statements to the compile-only
skeleton program so that it takes the form:

```
BEGIN
INTEGER DUMMY ;
EXTERNAL DUMMY ;
    BEGIN                       ... LEVEL 1 //
        BEGIN                   ... LEVEL 2 //
..........
            BEGIN               ... LEVEL n /.
            END ;               ... LEVEL n //
..........
        END ;                   ... LEVEL 2 //
    END ;                       ... LEVEL 1 //
END FINI
```

Construct all tests such that each contains the same number of lines, in
conformance with the "empty sheet" concept discussed in connection
with the lexical elements tests.

Next, to test embedded scoping ranges over non-null statements, construct a compile-only skeleton program with a predetermined number of declaration statements:

```
BEGIN
INTEGER DUMMY ;
EXTERNAL DUMMY ;
INTEGER A0 ;
INTEGER A1 ;
. . . . . . . . . . . .
INTEGER An ;
END FINI
```

Place a BEGIN - END pair of statements around the last declaration statement (INTEGER An ;), and recompile. Then place another BEGIN - END pair of statements so as to include both the next-to-last declaration and the pair just added, and recompile again. Continue this process until only the first declaration lies in the outermost BEGIN-END block of the program and takes the following form:

```
BEGIN
INTEGER DUMMY ;
EXTERNAL DUMMY ;
INTEGER A0 ;
    BEGIN                         ... LEVEL 1 //
    INTEGER A1 ;
    END ;
. . . . . . . . . . . .
    BEGIN                         ... LEVEL n //
    INTEGER An ;
    END ;
END FINI
```

Again, construct all tests such that each contains the same number of lines.

<u>Embedded scoping ranges containing different types of statements.</u>
These tests are designed to evaluate the performance of the compiler in
relation to processing embedded scoping ranges where the innermost
range contains different groups of the same type statement, some
executable and some only compilable. While the previous sets of tests
concentrate on the effect on the program of imposing additional scoping
levels, these tests attempt to isolate any effect of outer scoping levels
on statements at an inner level.

Specifically, these tests examine:

- Declaration statements.
- Executable statements.
- User tokens.
- User tokens redeclared.
- Nested procedure definitions.
- Labels.

These tests use, first, the skeleton program discussed earlier
under basic overhead costs. To this skeleton each set of tests adds,
successively, an increasing number of similar statements. Then one
or more (LOW as explained in Section 1) number of embedded levels
of declarations is added to the skeleton program, and an increasing
number of similar statements is again added. The number of levels of
declarations is increased (by STEP until HIGH number is reached, as
described earlier), and an increasing number of similar statements is
added. Thus, these tests vary first the number of similar statements
and then the number of embedded levels of declarations.

To test the compiler's performance with a varying <u>number of
declaration statements</u> within embedded scoping ranges, construct a
group of declaration statements of the form

        INTEGER A1 ;
        INTEGER A2 ;
        . . . .
        INTEGER Am ;

and add these to the innermost embedded scoping range of both the
skeleton test and the tests with increasing levels of embedded scoping
ranges. These tests need be compiled only, since no object code is
generated by the AED compiler for any construct tested.

To test the compiler's performance with a varying <u>number of
executable statements</u> within embedded scoping ranges, replace the
group of declaration statements in the previous test set with a group
of executable statements. These take the form:

A = A ;

Add repeated instances of the above assignment statement to the innermost
embedded scoping range of both the skeleton test and the tests with
increasing levels of embedded scoping ranges. The same statement

A = A ;

may be repeated as many times as desired since AED makes no restriction
on this language usage. It is however necessary to declare A and this
should be done in the outermost block. The last test in this set should
take the form:

```
BEGIN
INTEGER DUMMY ;
EXTERNAL DUMMY ;
INTEGER A ;
    BEGIN                   ... LEVEL 1 //
        BEGIN               ... LEVEL 2 //
        ......
            BEGIN           ... LEVEL n //
            A = A ;         ... ASSIGNMENT 1 //
            A = A ;         ... ASSIGNMENT 2 //
            .......
            A = A ;         ... ASSIGNMENT m //
            END ;           ... LEVEL n //
        ......
        END ;               ... LEVEL 2 //
    END ;                   ... LEVEL 1 //
END FINI
```

Since these tests contain executable statements, they must be compiled and executed, and both the compilation time and efficiency of the resultant object code evaluated.

To test the compiler's performance with a varying <u>number of user tokens</u> within embedded scoping ranges, replace the group of statements in the basic test set with a single declaration statement of increasing number of tokens (items being declared) of the form:

INTEGER A1, A2, ... Am ;

Repeat the process just described. These tests need only be compiled, since no ojbect code is generated by the AED compiler for any construct tested.

Note that this set of tests differs from the varying number of declarations statements tests in that the number of variables is considered significant, rather than the number of statements.

Construct a set of tests similar to the set just described but have each variable declared in both the innermost and outermost scoping ranges to evaluate the compiler's performance with a varying <u>number of redeclared user tokens.</u> Any observed differences between the previous set of tests and this set may be attributed to the fact that the variables had already been declared.

To test the compiler's performance with a varying <u>number of nested procedure definitions</u> add an increasing number of nested procedure definition statements to the skeleton test program. The last test in this set should take the form:

```
BEGIN
INTEGER DUMMY ;
EXTERNAL DUMMY ;
DEFINE PROCEDURE A1 TOBE BEGIN
    DEFINE PROCEDURE A2 TOBE BEGIN
    . . . . . . . . . . . .
        DEFINE PROCEDURE AN TOBE BEGIN
        END ;                        ... PROCEDURE An //
    . . . . . . . . . . . .
```

150

```
        END ;                        ... PROCEDURE A2 //
        END ;                        ... PROCEDURE A1 //
        END FINI
```

This set of tests needs only to be compiled, since no object code is generated by the AED compiler for any construct tested.

Lastly, to test the compiler's performance with a varying <u>number of labels</u>, perform the set of tests described for executable statements but assign a label to each statement. The features tested for inclusion to the skeleton test take the form:

```
        L1:  A = A ;

        L2:  A = A ;

        ..........

        Ln:  A = A ;
```

Compile and execute this set of tests and then compare the results with those from the executable statements tests. The differences in performance may be attributed to the presence of the labels.

5.   <u>Program Control Elements</u>

<u>Introduction.</u> The class of program control elements tests exercises the parsing and machine independent and dependent functional elements of the compiler. These tests are designed to be compiled and executed.

Program control elements play an important role in overall compiler performance. This role is more or less important depending upon the specific application for which the compiler is being used. For example, a "system" program (compiler program, file system program, etc.) in general makes heavier use of conditional expressions and Boolean operators than a "scientific" application program (numberical analysis and data reduction, avionics programs, etc.), which makes heavier use of loops and arithmetic operators. In all cases, however, program control statement efficiency is an important measure of compiler performance.

Basic overhead costs. To make the plot of the graphic output
more sensitive, as many overhead costs as possible are subtracted from
the test figures before plotting. To determine the overhead base cost
to be subtracted, an overhead skeleton program is constructed which
contains just the source statements not related to the test to be performed
(statements necessary to make the test program compilable), plus all
sub-expressions which are compilable in isolation (sub-cost (1)). These
skeleton programs are compiled and executed in order to obtain a "base
cost" figure, as discussed above.

For example, in AED, an overhead skeleton program might be:

```
BEGIN
DEFINE PROCEDURE MAIN TOBE
    BEGIN
    INTEGER A ;
    A=A ;
    A=A ;
     . . .
    A=A ;
    END
END FINI
```

The above skeleton corresponds to the test program:

```
BEGIN
DEFINE PROCEDURE MAIN TOBE
    BEGIN
    INTEGER A ;
    IF TRUE THEN A=A ;
    IF TRUE THEN A=A ;
     . . .
    IF TRUE THEN A=A ;
    END
END FINI
```

where the underlined phrases (simple IF -- THEN) are the forms being
tested.

152

<u>Types of IF tests</u>. IF statement tests are broken down into tests of the IF -- THEN and IF -- THEN -- ELSE statement forms, and tests of the various THEN and ELSE clause forms. The same set of tests is intended for testing both compiler and resulting object code efficiency, and therefore include whatever source statements are necessary to permit their convenient execution, as well as the basic syntax necessary for compilation.

The description of each test program contains three integers: LOW, STEP, and HIGH, which represent the number of times a particular construct is to be repeated in the test program. It is anticipated, depending upon the clock accuracy and consistency of results from repetition of the same test runs, that these LOW, STEP, and HIGH integers will need to be varied, depending upon the specific computer and operating system on which the testing is conducted.

In each case, the tests are designed to test the parser and code generator portions of the compiler. The costs of reading input, performing lexical analyses on the input characters, and writing compiler output are not of interest here, and these costs are included in the base cost determined from the skeleton program and subtracted from the test program results.

<u>Conditional statement forms (IF)</u>. This category covers the basic "conditional" class. AED statements are used as illustrations, below.

The form of an IF construction in AED is:

IF <u>X</u> THEN <u>S1</u> ELSE <u>S2</u>

where <u>X</u> is any Boolean variable or phrase, and <u>S1</u> and <u>S2</u> are variables, constants, or phrases whose type depends upon the particular form of IF construction used. If the IF is used in its "non-valued" form, <u>S1</u> and <u>S2</u> must have the type "statement"; if the IF is used in its "valued" form, <u>S1</u> and <u>S2</u> must be of the same type, and may take on whatever form is required by the larger statement in which the IF is nested. For

example, in AED, the following statements illustrate some of the "valued" uses of IF:

Pointer1 = IF Bool1 THEN Pointer2 ELSE Pointer3
GOTO IF Bool2 THEN Label1 ELSE Label2
A = A + IF Bool3 THEN 1 ELSE 2

In the tests described below, the "valued" form of IF is ignored, since it is not basically a program control construct and is not common among compiling languages. (For example, it is not available in J3B.) Only the non-valued IF form is examined here.

Tests for IF clause forms. The IF--THEN statement has two sub-expressions, with their associated sub-costs:

IF expression1 THEN expression2 ;

where expression1 is of type Boolean and expression2 is of type statement. Since expression2 is of type statement, it can be compiled in isolation, and therefore sub-cost (1) resulting from expression2 can be determined. However, expression1 cannot be compiled in isolation, and there is therefore no way to separate out its contributed sub-cost. Also, since the IF statement will not compile unless expression1 and expression2 are given, there is no way to determine the invariant cost of the IF-statement form (sub-cost (2)).

Consequently, the test procedure to be used in this case is as follows. First construct a program containing LOW number of statements of the form

IF TRUE THEN statement ;

where statement is any executable statement which will cause the IF statement to be compilable and executable (e.g., A=A). Construct a series of test programs, incrementing the number of these statements by STEP until a program containing HIGH number of statements. Compile and execute each test program thus generated. Repeat the same set of tests with the IF FALSE form, so that statement is never executed, and

154

all that is measured is the mechanism used to test and branch between statements. Of course, the overhead figure subtracted from this second set of test runs does not include the repetative statement execution.

Repeat the above test series, replacing the IF TRUE with (1) IF FALSE, and (2) IF Boolean variable, and then (3) with the expression var1 == var2 (simple EQL comparison). The test set need not be run using the other 5 forms of comparison operator, since it is assumed that the difference in efficiency between the various comparison operators is a separate topic covered by a separate set of tests, and is not coupled to the IF statement efficiency.

To complete the IF clause test set, compile and run the same series of tests using the compound expression form Bool1 AND Bool2 for the variable. Again, as with the individual comparison operators, it is assumed that differences in efficiency between operators and combinations of operators (OR, NOT, Bool1 AND NOT Bool2, etc.) can be separated from IF statement efficiency tests; the operator efficiency question is covered by another set of tests.

Goal of THEN and ELSE clause tests. The goal of this series of tests is to determine the interaction costs (sub-cost (3)) of the six permissable forms of THEN clause: BEGIN, CALL, FOR, IF, GOTO, assignment. A separate overhead skeleton program for each of the six permissable forms is generated, to subtract off the effect of the form of THEN- clause used and thus leave the effect of the IF -- THEN portion only. For example, the overhead skeleton for the CALL form is:

```
BEGIN
DEFINE PROCEDURE MAIN TOBE
     BEGIN
     CALL( ) ;
     CALL( ) ;
     ...
     CALL( ) ;
     END
END FINI
```

155

This skeleton corresponds to the test program:

```
BEGIN
DEFINE PROCEDURE MAIN TOBE
    BEGIN
    IF TRUE THEN CALL( ) ;
    IF TRUE THEN CALL( ) ;
    ...
    IF TRUE THEN CALL( ) ;
    END
END FINI
```

where the underlined portion is the construct to be tested.

THEN and ELSE clause tests. Construct a series of tests, beginning with LOW number of statements and increasing this number by STEP until HIGH for each of the six forms of THEN clause: BEGIN, CALL, FOR, IF, GOTO, and Assignment. Set the IF variable to TRUE. Compile and execute these tests, as well as the overhead skeleton program corresponding to each test program. Subtract the effect of the overhead from each test, to obtain the desired performance measure.

Repeat the same series of tests, replacing each statement of the form

IF variable THEN statement

by the statement

IF variable THEN statement ELSE statement

Note that the overhead skeleton programs need not be re-compiled or re-run for these second ELSE clause tests, since the desired overhead numbers can be obtained directly from the first set, based upon the number of statements executed. Run this set of tests twice, first with variable set to TRUE, and then with variable set to FALSE. This will show any differences in performance between the THEN and the ELSE portions of the statement. Use the same expression for both statements (THEN and ELSE).

156

FOR Statements. The FOR statement (looping statement) has the following forms in AED:

Form 1: FOR index variable = initial value WHILE continuation condition DO loop body ;

Form 2: FOR index variable = initial value STEP increment value UNTIL termination value DO loop body ;

Form 3: FOR index variable = value, value, value, ... , value DO loop body ;

Form 4: WHILE continuation condition DO loop body ;

An elaboration on Form 3 allows any value in the list to the right of the = to be any of the following three phrases:

- initial value WHILE continuation condition
- initial value STEP increment value UNTIL termination value
- initial value STEP increment value WHILE continuation condition

The goal in running the tests on FOR statements is to determine the efficiency of the FOR loop mechanism used to accomplish the various forms of loops. To do this, tests are constructed to determine the efficiency of each form of FOR statement, working with each permissable form of initial value, index variable, increment value, continuation condition, termination condition, and loop body.

Tests for Variations in Loop Statement Operands. Construct a set of tests with LOW to HIGH repetitions of each of the looping statement forms, using the simplest convenient form of operands. In AED this corresponds to the four sets of statements:

FOR I=I WHILE I LEQ 1 DO I=I+1 ;
FOR I=I STEP 1 UNTIL 1 DO I=I+1 ;
FOR I=I, I DO I=I+1 ;
WHILE I LEQ 1 DO I=I+1 ;

where I is declared to be of type INTEGER. Since the AED compiler initializes all integer variables to 0, each loop statement causes the DO portion (I=I+1) to be executed twice; thus each set of test results may be compared to each other set to determine relative efficiency of the equivalent construct.

157

Next, determine the base value to be subtracted from each set of
tests. This is done by compiling and executing a series of programs
containing $n*2$ repetitions of the statement "I=I+1" (the DO expression),
where n is the number of loop statements in the test program, plus n
repetitions of the statement "I=I" (the FOR expression). The STEP,
UNTIL, and WHILE expressions are not separable.

To test various other forms of operands, next vary each operand
according to the following sequence. Consider the statement forms:

FOR i=e STEP f UNTIL g DO s ;

FOR i=e WHILE x DO s ;

FOR i=$e_1, e_2, \ldots e_n$ DO s ;

WHILE x DO s ;

Each of the lower-case letter codes represents an operand class that is
to be varied to gather performance measurements:

Letter Code	Variations Tested
i	Structure and Data Type
e, f, g	Literal/Variable Structure Type
x	Boolean Data Type
s	Statement Type

Construct a series of tests for each of the four loop statement forms,
using the following variations, one at a time, in place of the correspond-
ing letter code shown above. For all but the single letter code being
examined, use the basic (simplest) case employed at the beginning
of this section.

Forms of "i" Structure and Data Types:

a. Simple integer.

b. Subscripted integer array element.

c. Integer bead component.

d. Simple real variable.

e. Subscripted real array element.

f. Real bead component.

158

g. Simple pointer variable.

h. Subscripted pointer array element.

i. Pointer bead component.

j. Simple Boolean variable.

k. Subscripted Boolean array element.

l. Boolean bead component.

General form of "e" (or "e_i"), "f" (or "f_i"), or "g" (or "g_i") literal or variable:

a. Single literal, or non-subscripted variable with the same data type as i.

b. Other.

Form of "x" boolean data types:

a. Simple Boolean.

b. Other.

Form of "s" statement types:

a. BEGIN.

b. CALL.

c. FOR.

d. IF.

e. GOTO.

f. Assignment.

Tests of loop execution efficiency.   The above test series examines the cost of the various forms of operands used with looping statements. The run-time cost of the loop mechanism itself is determined by constructing a series of tests containing a single loop statement, specifying an increasingly larger number of iterations around the loop (e.g., UNTIL 10, UNTIL 100, UNTIL 1000, etc.). Subtract the base value derived from the test run which contained the proper number of repetitions of the statement "I=I+1" to isolate the cost of the loop action. Run the tests for all four forms of loop statement, and for all forms of operand listed above. In this series, the only meaningful efficiency measure is on the code generator output (data resulting from the run, not the compilation).

159

<u>Procedure and function call mechanism tests.</u>   To test the
procedure and function call mechanisms of the compiler, construct the
following series of tests. Repeat the simplest form of procedure and
function call LOW to HIGH times, creating two series of test programs
and a resultant graph showing the slope of the individual test program
results. The "simplest" forms are

> F( ) ;

for a procedure, and

> A=F( ) ;

for a function call.

Use a program format containing only the minimum number of statements
required to permit compilation and execution of the test series, and
subtract the cost of the skeleton version of this program as a base value.
Compile and run the tests using the simple form and then the recursive
form of procedure and function definition. Rerun the test series using
all other permissable forms of calls (Fortran and COBOL compatible
forms, internal versus external forms, etc.).

More complex tests of function and procedure calls used with
various language constructs and environments are not considered here;
they are discussed in relation to each individual context (IF--THEN,
FOR, etc.). Also, tests of procedure and function argument and output
value transmission are convered under the test series discussed in
Section 6.

<u>Other program control tests.</u>   Unconditional transfer statements
comprise the final set of program control statements to be tested. These
include labels, transfer statements (GOTO in AED), and switches. Unlike
the conditional (IF) statement and the loop (FOR) statement which are
tied to Boolean phrases of various types, unconditional transfers are
completely separable from other language forms.

To construct an efficiency test for unconditional transfer statements, write a series of test programs as follows:

- Write a test program containing transfers to the next statement:

      GOTO L1 ;

  L1 : GOTO L2 ;

  L2 : GOTO L3 ;

      (etc.)

  Include enough of these to permit sufficient compiling and execution time to be measurable.

  Compile and run this test, subtracting the overhead skeleton cost. Continue to increase the number of transfers and run tests until a pattern is evident.

- Write a second series of tests based on the first, which includes a single executable statement (such as "A=A") before each transfer statement. (This should detect control sequence optimization.) Compiler and run these tests, subtracting off the overhead.

- Repeat the entire series of tests described above, replacing the simple transfer statements with switch calls which have the same logical effect. Use integer literals in the switch calls. For example, in AED, the repeated statements of the first test set would be:

  SWITCH W = L1, L2, L3, ...

      GOTO W(1) ;

  L1 : GOTO W(2) ;

  L2 : GOTO W(3) ;

      (etc)

- Repeat the preceding switch tests by replacing the integer literals by all other legal forms, such as integer components, arguments, etc.

- Construct a series of tests for transfer statements used as procedure arguments. This test is accomplished by defining two procedures and transferring control back and forth between the two programs. For example, in AED this would be:

```
DEFINE PROCEDURE MAIN TOBE
     BEGIN
     PROC(L1) ;
L1 : PROC(L2) ;
L2 : PROC(L3) ;
     (etc)
DEFINE PROCEDURE PROC(L) WHERE LABEL L TOBE
GOTO L ;
```

Include sufficient numbers of these call/transfer sequences
to provide significant timing data. Compile, execute, and
subtract overhead values from these tests. Rerun the
tests passing the target label through 2, 3, 4, etc., levels
of nest depth. Lastly, rerun the tests using recursive
procedure forms, rather than the simple procedure
mechanism.

Finally, it is desired to test the non-lexical performance features
of labels (lexical features are examined in Section 2). The major such
performance feature involves the code generator logic employed in its
machine register usage memory. In the simplest logic design, each
label occurrence causes all register memory in the compiler to be
cleared, so that code must be generated to load a register the next
time a particular value is needed in the register. A more sophisticated
code generator performs sufficient flow analysis to determine that
certain register memory need not be cleared, and, if a needed value
was already in a particular register, it need not be re-loaded because
of the occurrence of the label.

To test this code generator feature, construct a test program
which repeats the pattern

```
L0 : X = COMP(P) ;
L1 : X = COMP(P) ;
     (etc)
```

Repeat this labeled statement form LOW to HIGH time   ι ω : ι , ιnd
run this test series, and then re-compile and re-run the identi....
programs, except remove all statement labels. Subtract the ba⁻⌐

162

values obtained by compiling and running a program containing the same number of repetitions of the statement

        X = X ;

The difference between the labeled and non-labeled versions of the test illustrate the desired code generator performance measure.

6. Data Manipulation and Computation Tests

The class of data manipulation and computation tests exercise a portion of the parsing and code generation programs of the compiler. The tests measure performance in handling the various data manipulation operators, as used in various environment. All tests are designed to be compiled and executed; skeleton programs resulting in base values to be subtracted from the test run figures are discussed with each set of tests.

Arithmetic assignment operator. The simplest form of arithmetic data manipulation is the assignment statement A = B;. The first series of tests examines the efficiency of this simple assignment form. For this series of tests, we are interested only in the various means employed by the code generator for storing arithmetic data. In particular, we are interested in the interaction between the "=" operator and the right-hand side (B). Therefore, the series of tests is based upon variations of B which may cause variations in the code generated for the assignment operation, such as a "store zero" instruction to implement the statement A = 0;. (The left-hand side of the "=" is tested in a separate set of tests, described later in this section.)

The tests consist of a series of programs repeating the statement form to be tested. The statement is repeated LOW times, and the number of repetitions is increased by STEP until HIGH number of statements.

The first set of tests examines all forms of literals. The specific statement forms to be tested are:

- A=0 ; (integer)
- A=1 ; (integer)
- A= -1 ; (integer-negation)
- R=0 ; (real)
- R=1. ; (real)
- R= -1 ; (real-negation)
- B=TRUE ; (Boolean)

164

- B=FALSE ; (Boolean)
- P=NULL ; (Pointer)
- Other.

where the class "Other" includes any other special forms peculiar to the compiler being tested (e.g. 0's in bit strings, blanks in character strings, etc.).

A single overhead skeleton program is used for all of the above tests, which consists of the minimum required for compilation and execution. In AED, this is:

```
BEGIN
DEFINE PROCEDURE MAIN TOBE
BEGIN
INTEGER A ;
A = 0 ;
END ;
END FINI
```

Construct a second series of tests similar to the above set, except using all other permissible structure and data types of variable instead of literal on the right of the "=".

This includes the following forms:

- A=B (all types, integer, Boolean, real, pointer, etc.)
- A=AR(1) (all types of array, using both an integer variable and a literal for I, and including as many levels of subscripting as permitted)
- A=COMP(P) (bead or table item of all types, positive and negative offset and packing)

Test the above forms for B, AR, I, COMP, and P as local variables, arguments of procedures, and arguments of recursive procedures. For any single test, make only one item an argument, to isolate the effect of the mechanism.

165

Function call test. Construct a series of tests to examine the
cost of storing the result of a function call. This test is the same as the
above set, except that the form A=F ( ); is repeated in each test program,
and the base value subtracted from the test set includes the repeated
calls on F (F( ) ; F( ) ; etc.) to isolate just the cost of the A storage
mechanism. Use all legal types for A and F (integer, real, etc.) and
make A of the same type as F.

Right hand side binary operator test. Next, test the cost of binary
arithmetic operators used on the right hand side of the "=". Construct the
following series of tests, beginning with LOW number of repetitions and
increasing the number by adding STEP new repetitions until HIGH. Use the
minimum basic form of program which will allow the test to be compiled and
executed. In AED, this would be:

```
BEGIN
DEFINE PROCEDURE MAIN TOBE
BEGIN
INTEGER A ;
A = A op A ;
A = A op A ;
A = A op A ;
END
END FINI
```

The overhead skeleton program for these tests replaces the

    A = A op A ;

by the form

    A = A ;

This skeleton program base value subtracted from each test program
thus leaves the cost of op A, which includes the three sub-costs: the
cost of evaluating A, the invariant cost of the op, and any residual
interactive cost of using the particular form of op with the particular
type of A.

166

The specific test programs to be compiled and executed are:

- Five sets of tests, of the form A = A op A, with op being +, -, *, / and **. Use an integer variable name for A.

- A second set of five tests of the same form as above, using a real variable name for A instead of integer.

- A third set of tests of the same form, using other types of variables for A. In AED, this would include a small set of pointer arithmetic constructs; in other languages it might include bit-arithmetic, etc.

- A series of tests of the same form using other, special forms of binary operator for op. This set includes the operators for shifting (R.S. and L.S. in AED, SHIFTR and SHIFTL in J3B) and logical masking and combining (.A. for AND, .V. for OR, etc.)

- A set of tests of the form A = B op C, using special literals such as 1 and 2 for B and C. (The code generated for manipulating these particular literals may be quite unique).

Include tests for the following specific forms:

- A + 1, and 1 + A.
- A - 2, and 1 - A.
- A * 2, and 2 * A.
- A/2, and 1/A.
- A**2, and e**A (e=2.71828...).
- A op Other Literal, and Other Literal op A.

Use first integer and then real A and, lastly, any other type of variable which produces a legal construct in the language (e.g., P = P + 1, with P of type pointer).

In all of the above tests, the base value to be subtracted is obtained by creating a skeleton program which repeats the statement of the form A = A. Note that a different skeleton program must be generated for each data type used for A. In the final test series involving literals, the same base value is used for both permutations (e.g., the same base value derived by repeating A = A, is used for both A + 1 and 1 + A), thus illustrating the effect of inverting the order of the operands.

167

Left hand side of assignment statement. Construct a series of
tests to examine all legal forms of referent permitted on the left hand
side of an assignment statement, thus illustrating the parsing and code
generation actions performed for the "=" operator. Include the following
forms:

- Simple variable (A=).
- Subscripted Variable (AR(I) =, AR(Literal) =, AR(expression) =).
- Bead or Table Item Component (COMP(PTR) =).

If multiple subscripting is available, the subscripted variable form is
elaborated to show all possible combinations of variable, literal, or
expression for each subscript.

For each test in the set, repeat one of the above forms LOW number
of times, increasing by STEP until HIGH. For the right hand side of the
statement use the simplest form that permits compilation and execution
of the test (A=A, AR(I) =A, etc.).

The statement form left-expression = right-expression is not
separable, for purposes of obtaining a base value. That is, neither
left-expression nor right-expression can be compiled in isolation.
Therefore, the skeleton program for this set of tests consists of the
required compilation framework plus a single "A = A" statement to
permit the skeleton program to compile and execute. The skeleton
program, in AED, is:

```
BEGIN
DEFINE PROCEDURE MAIN TOBE
BEGIN
INTEGER A ;
A = A ;
END ;
END FINI
```

Repeat this set of tests for the following cases:

- All data types of variable, array, and component
  (integer, pointer, real, etc.).

168

- All forms of COMP offset and packing.
- All combinations of non-recursive and recursive procedure arguments (for AR, I, COMP, etc.).

If permitted by the language being tested, also test the forms

$$F =$$

and

$$F( ) =$$

for setting the values of functions.

Strings of arithmetic operations. So far, the test programs have examined only single occurrences of a particular operator. For example, the statement $A = A + A$ has been examined, but the effect of a string such as $A = A + A + A$, or $A = A + A + A + A$, etc. has not been examined. It is true that in some instances, using some particular computer hardware designs, a compiler might take advantage of the fact that a string of operators is specified, and thus compile a more efficient object code sequence. Indeed, sequences involving "*" and "/" quite frequently result in interesting patterns of object code, since products and dividends commonly occur in separate hardware registers, and the opportunities for compiler object code optimization are clearly evident, especially taking into account the commutivity of source code operator/operand sequences.

However, the set of tests to evaluate these types of operator strings and sequences is beyond the scope of the present effort, especially in light of the fact the "system-type" applications (compilers, file systems, etc.) make very little use of arithmetic operator strings, and the evaluation of their efficiency for system programming applications is of little importance compared to other forms of evaluation considered here. Arithmetic operator strings are therefore not included in the set of tests described here, but are left for future follow-on efforts to consider.

169

Boolean expression tests. The set of Boolean expression tests
closely parallels the arithmetic expression tests described above. The
Boolean tests are intended to examine the compiler's parsing and code
generation handling of Boolean forms of assignment and data manipulation,
using both literals and variables as operands. As in the arithmetic tests,
we are interested primarily in the handling of the Boolean operators,
both as a fixed cost for each operator and as the expense of the inter-
action between the operator and its environment. The cost of different
forms of data referencing and retrieval are not considered here; these
costs are covered in a separate set of tests.

As in the case of the arithmetic tests, a series of Boolean test
programs is constructed, where each program repeats a statement to
be tested a certain number of times. The first test contains LOW number
of repetitions, and these are increased by STEP number of repetitions for
each succeeding test, until HIGH number of repetitions is reached. A
"base value" is determined by writing one or more overhead skeleton
programs and compiling and executing these skeletons. The base value
thus derived is then subtracted from the figures determined from each
test compilation and run. A plot of these final figures then shows a slope
(expected to be linear in most cases) which characterizes the cost of
the feature being tested.

Use of single Boolean literals. The first series of tests evaluates
the cost of the Boolean literals TRUE and FALSE. To test these literals,
construct a series of tests containing LOW to HIGH repetitions of the
statement A = TRUE, where A is a Boolean variable. Since neither
A nor TRUE are compilable in isolation, the overhead skeleton
program for this series of tests consists of just the basic statements
necessary to permit compilation and execution. In AED, this is

```
BEGIN
DEFINE PROCEDURE MAIN TOBE
BEGIN
BOOLEAN A ;
A = TRUE ;
```

170

END ;

END FINI ;

Note that, in AED, at least one executable statement is required to permit compilation, and the skeleton program therefore, contains the single statement "A = TRUE".

Repeat the above test set, replacing A = TRUE by A = FALSE to determine the assignment of a Boolean variable to FALSE. The results of this test may be compared to the TRUE test, to determine any differences in cost (one of the two forms may take advantage of a "store zero" operation, for example).

Use of Boolean Variables. Repeat the above series of tests, replacing the repeated statement by A = B where B is a Boolean variable. Construct tests to show the cost of the assignment operator for each of the following forms of Boolean variable, B:

- A = B (simple variable).
- A = BA (I) (Boolean array element -- use variable and literal for I, and use as many subscripts as permitted.
- A = COMP (P) (Boolean component or table item),

Conduct series of tests for all legal data types for B, BA, and COMP (integer, real, pointer, etc.) and for B, BA, I, COMP, and P as simple and as recursive procedure arguments. Make only one item at a time an argument to isolate the desired cost. Use the same skeleton program base value as for A = Literal in all cases.

Boolean operator tests. The series of Boolean operator tests is designed to determine two costs: the invariant cost of the use of the Boolean operator itself, and the interactive cost of using the operator with its right and/or left context, in various environments. Use of the operator in conjunction with loops and conditionals (IF and FOR) is not covered here; those tests are discussed in Section 5.

171

The following forms of Boolean operator are covered by the tests:

- Predicates (e. g., A > 5).
- Unary operators (e. g., NOT A).
- Binary Operators (e. g., A AND B).

As in the case of arithmetic operators, strings of operators (e. g., A AND B OR C AND NOT D) are somewhat more common in system programs than arithmetic operator strings, they are still not of major importance, and the complex series of tests which would be required to test all possible strings of Boolean operators of interest, is clearly beyond the scope of the present study.

Common form for all boolean operator tests. All Boolean operator tests (predicate, unary, and binary operators forms) employ the same form of repetitive assignment statement, which is repeated LCW times in the first test program, and repeated an additional STEP times for each succeeding test until HIGH number of repetitions is reached. The repeated statement takes the form:

A = (form to be tested ;

and uses the repeated form:

A = A ;

for predicate forms, and for all other forms in the overhead skeleton program, to obtain the base value to be subtracted from the test results.

Predicate tests. The predicate form includes all comparisons between variables and literals of all types. In ALD, this class of statement takes the form:

A = A predicate literal;

or

A = A predicate variable;

172

By using the base value derived from a skeleton program containing repetitions of the statement A = A;, the test results give the cost of **predicate** literal or **predicate** variable. More specifically, this cost represents:

    (1)   The cost of accessing the literal or variable value.

    (2)   The invarient cost of the **predicate**.

    (3)   The cost of the interaction between the **predicate** and the literal or variable in the given context.

    (4)   The cost of the interaction between the left hand side and the **predicate** (the cost of the A **predicate** interaction) in the given context.

These four cost factors are not separable, since none of the portions of the statement can be further broken down and compiled in isolation.

To create a set of data results representing the sum of these four costs, construct a series of test programs containing from LOW to HIGH number of repetitions of the statement form:

A = A **predicate** (variable or literal);

Use the **predicate** forms:

- = = (EQL).
- ¬ = (NEQ).
- < (LES).
- > (GRT).
- > = (LEQ).
- > = (GEQ).

Construct a set of tests using all legal types of variable for A (integer, real, pointer, etc.) as compared to all legal literal forms (0, 1, NULL, TRUF FALSE, etc.) and all legal variable structure and data type forms. The variable forms include:

- A (Simple variable).
- AR(1) (array).
- COMP (P) (component or table item).

for all types of these variables (integer, real, etc.) and using the local procedure argument, and recursive procedure argument form for A, AR, I, COMP, and P.

Unary operators. The only Boolean unary operator is NOT. This series of tests is designed to test the parsing and code generator performance of the NOT operator in all of the various contexts in which it is properly used.

Since NOT always precedes a complete Boolean expression, it is possible to separate its cost from the remainder of the statement in which it appears. More specifically, consider the statement

A = NOT B;

Subtracting the effect of the statement

A = B;

leaves the following costs:

(1) The invariant effect of the use of NOT.
(2) The interactive cost of the NOT operator in its context between the "=" and the "B".

These two subcosts are not separable. However, if the use of NOT in several other contexts exhibits an essentially identical cost, the interactive context cost ((2), above), can be considered to be negligible.

To test the NOT operator, construct a series of tests beginning with LOW repetitions of the NOT statement to be examined and increasing the number of repetitions by STEP until HIGH. Use

A = expression;

for the repetitive statement, using the following expressions:

- NOT A (Preceding a Boolean variable).
- NOT AR(I) (Preceding an array element).
- NOT COMP (P) (Preceding a component or a table item).
- NOT (I = = J) (Preceding a predicate phrase).
- NOT B AND A (Preceding a variable within an expression).

174

- A AND NOT (B = = C) (Preceding a predicate in an expression).
- NOT (A AND B) (Preceding an expression).
- NOT F ( ) (Preceding a Boolean function).
- Any other legal construct using NOT.

For the array form, use both a variable and a literal for the index, and repeat the tests for multiple subscripts, using all combinations of variable and literal subscript values, up to the maximum number of subscripts permitted. Repeat the entire set of tests, using both non-recursive and recursive procedure arguments for each variable used in the phrase following the NOT.

Binary Operator Tests. The class of binary Boolean operators includes:

- AND.
- OR (Inclusive).
- XOR (Exclusive).
- EQV (Equivalence).
- IMF (Implication).

On a given compiler, not all of these may be available, but tests should be run on as many as are permitted.

To obtain the desired test data, compile and execute a set of repetitive statement is:

A = expression OPERATOR expression;

where OPERATOR is one of the above, and the expressions are various combinations of Boolean literal, variable, predicate, and compound expression forms designed to provide varying contexts for the use of OPERATOR. The overhead skeleton program repeats the statement:

A = TRUE;

to obtain a single base value to be subtracted from all tests. Using this base value, the test results, therefore, provide a total cost which includes:

175

(1) The incremental cost of evaluating the left hand <u>expression</u> as opposed to evaluating TRUE.

(2) The invariant cost of OPERATOR.

(3) The incremental cost of the interaction between the left-hand <u>expression</u> and OPERATOR.

(4) The incremental cost of the interaction between OPERATOR and the right-hand <u>expression</u>.

(5) The cost of evaluating the right hand <u>expression</u>.

To obtain these costs in a wide variety of environments compile and run a series of tests, using each of the follow repetative right-hand-side forms:

Variable <u>OPERATOR</u> TRUE.

Variable <u>OPERATOR</u> FALSE.

TRUE <u>OPERATOR</u> Variable.

FALSE <u>OPERATOR</u> Variable.

Variable <u>OPERATOR</u> Variable.

Simple Predicate <u>OPERATOR</u> TRUE.

Simple Predicate <u>OPERATOR</u> FALSE.

TRUE <u>OPERATOR</u> Simple Predicate.

FALSE <u>OPERATOR</u> Simple Predicate.

Variable <u>OPERATOR</u> Simple Predicate.

Simple Predicate <u>OPERATOR</u> Variable.

Simple Predicate <u>OPERATOR</u> Simple Predicate.

Compound Expression <u>OPERATOR</u> TRUE.

Compound Expression <u>OPERATOR</u> FALSE.

TRUE <u>OPERATOR</u> Compound Expression.

FALSE <u>OPERATOR</u> Compound Expression.

Compound Expression <u>OPERATOR</u> Variable.

Variable <u>OPERATOR</u> Compound Expression.

Compound Expression <u>OPERATOR</u> Simple Predicate.

Simple Predicate <u>OPERATOR</u> Compound Expression.

Compound Expression <u>OPERATOR</u> Compound Expression.

where "compound expression" includes the unary and binary operator forms.

For each "Variable", use the forms

- Variable name (real, integer, pointer, etc. where permissable in the language).
- AR (I) (all types of arrays and subscript forms).
- COMP (P) (all types of components and table elements).

Run the tests for static, procedure argument, and recursive procedure argument forms for "Variable".

Argument transmission tests. For modular software, procedure argument transmission performance is an important consideration. This series of tests examines the argument transmission performance of the compiler's parsing and code generation mechanism, for all forms of argument transmission. Argument accessing is not considered here; that topic is covered under data access tests, which examines all forms of data references.

The forms of argument transmission considered here include all forms of argument permitted in a procedure call statement. In AED, this statement takes the form:

F(arg1, arg2, ... , argn) ;

The cost associated with argument transmission includes:

- The cost of parsing the argument list.
- The cost of generating the basic argument transmission cost and register loading instructions.
- The cost of accessing the value of each argument.
- The cost of creating the data item to be passed for each argument and placing it in the transmission area.

These various costs are, in general, inseparable, and the tests described below make no attempt to separate them. The technique used to obtain performance data is the same technique used with other language features in this area: a series of programs is constructed, which repeats the language feature of interest, LOW to HIGH times. A base value is subtracted from the measured cost of each compilation and execution, to eliminate as much of the overhead as possible, and the resulting test data is plotted. The slope of the resulting graph is used to measure the performance of the feature being tested.

177

The base value for the argument transmission tests is obtained by compiling and running a test program of the form, in AED:

```
BEGIN
DEFINE PROCEDURE MAIN TOBE
   BEGIN
   F( ) ;
   F( ) ;
   ...
   F( ) ;
   END ;
END FINI
```

This test therefore represents the cost of passing no arguments, but includes the calling mechanism and other overhead costs.

To test the argument transmission feature, construct a series of test programs which repeat the statement:

```
F(arg) ;
```

where "arg" is set to the following forms, one at a time:

- Simple variable (integer, real, pointer, procedure, label, etc.).

- Structure type variations.

  AR(I) (array, for I as both a literal and a variable and using from one to the highest number of permissable subscripts).

  COMP(P) (based variable (component in AED)).

- Expression "A + B" (if this form is available in the compiler).

Use all permissible data types for A, B, AR, and COMP. For the COMP(P) form, use both packed and unpacked COMP forms, as well as positive and negative offsets from P.

It is anticipated that the small cost of parsing an argument string (arg1, arg2,...) versus the single argument form discussed above (f(arg)) is only of academic interest, and not worth the effort to construct a separate test series. That is, the parsing cost of processing the comma string is a very small portion of the four cost items discussed above.

178

It should be note that the "expression" form (A+B) is the simplest form which shows the cost of calculating an argument value, not just accessing a piece of data. This form includes additional code generator cost during compilation and during execution of the object program, and therefore exercises a portion of the code generator not tested elsewhere (except possibly in the case of a packed COMP argument). However, any more elaborate forms of expression merely add the cost of expression evaluation, and are not directly related to argument transmission costs.

CHAPTER 10

HOW TO EVALUATE ENVIRONMENTAL DIFFERENCES

1. How the Environment Equalizing Question was Studied

Introduction. In Chapter 2 an overview was presented of how the environment equalization question was studied. In this section, this topic will be discussed in further detail. For convenience, the full statement of the question is repeated below.

> If two compilers with the same features operate
> in different environments, how can their measured
> differences in performance be allocated to environ-
> mental differences vs. compiler differences?

The main thrust of the approach to this question was to test the feasibility of using the elements of a high level language in which compilers are written as a basis for establishing a "compiler Gibson mix". The investigation was very limited in scope. Static and dynamic Compiler Demand Profiles were constructed from the AED source language modules and execution behavior of two compilers: an AED compiler and a J3B compiler. The static Compiler Demand Profiles are presented in Chapter 11 and the dynamic profiles are presented in Chapter 12. Details of the methods used to generate these profiles will be presented later in this section.

Before the data had been generated from which the profiles were calculated, it had been expected that the static AED and J3B profiles might exhibit differences which would be reduced in the dynamic profiles. Had this turned out to be the case, this result would have been interpreted as a partial demonstration that compilers (perhaps of a specified class) all have substantially similar dynamic profiles, and, therefore, a typical profile can be established which would be used to define the desired "compiler Gibson mix". The results of the study produced static and dynamic profiles, both of which established an overall impression of great similarity between the AED and J3B profiles. However, the static profiles showed significantly greater overall similarity than the dynamic profiles. From this result we concluded that the particular method used to generate the

dynamic profile introduced statistical anomalies for reasons discussed in Chapter 12.

The principal conclusion drawn from the overall similarity between the profiles of the AED and J3B compilers adequately was that they satisfied the intended partial demonstration that compilers are essentially similar with respect to their Compiler Demand Profiles, and that a "compiler Gibson mix" can be defined on the basis of a suitably established "typical" profile. The profiles presented in Chapters 11 and 12 are used in Section 2 as an illustration of such a "typical" profile, and a "compiler Gibson mix" based on these profiles is defined as an example. In the absence of better data, based on a more elaborate study than the present one, it is recommended that the example "compiler Gibson mix" defined in Section 2 be used as a basis for "equalizing environments" until better data becomes available.

Steps used to generate Compiler Demand Profiles. The steps listed below were followed in generating static and dynamic Compiler Demand Profiles for the AED and J3B compilers.

1.   A list of AED language forms were prepared which were used more-or-less as a basis for the profiles. This list was approximately a subset of the list of language forms for a User Profile which was presented in Section 5 of Chapter 8.

2.   An AED compiler was modified to collect statistics on the occurrences of language forms in the modules being compiled. Due to the nature of this AED compiler, the placement of the instruments was most conveniently made in such a fashion that the statistics were actually gathered on the tree structure representation of the language forms, which were not exactly isomorphic with the forms in the list established during Step 1. Details of the instrumentation of the AED compiler are presented in Appendix 1.

3.   All AED source language modules of an AED and J3B compiler were compiled using the instrumented AED compiler created in Step 2. For each procedure of each module, a table was printed out containing counts of occurrences of the language forms found by the instrumented compiler. Appendix 3 presents samples of these raw data tables, together with an explanation of how the raw data is interpreted. (Note that some source language modules of the two compilers were written in assembly language, and these did not contribute to the statistics. Also, AED library routines used by both compilers were not included in the statistics.)

181

4. Two J3B source language programs were selected from a collection of such programs used for acceptance testing of the J3B compiler. These programs in combination exhibited most of the language forms in the list generated at Step 1. The programs were modified so that two corresponding AED programs could be prepared which were as syntactically identical to the J3B programs as we could make them. These four test programs (two AED and two J3B) are presented in Appendix 2.

5. Modified version of both the AED and J3B compilers were created which would count the number of calls executed to each procedure in each compiler.

6. The modified compilers created at Step 5 were used to compile the corresponding pairs of test programs produced at Step 4. The results of these compilations were counts of the number of times each procedure of each compiler was called in compiling each of two test programs. The counts generated from the pairs of test programs were added together to create weighting factors for each procedure of each compiler.

7. A program was written which would combine the tables generated at Step 3 with weighting factors. The elements of each table were multiplied by the weighting factor for the table (corresponding to a procedure). Then the resulting tables were added together to produce a summary table for each compiler. This program was used twice for each compiler: once with all weighting factors set to 1, and once using the weighting factors generated at Step 6. The resulting tables constituted the raw data for the static and dynamic Compiler Demand Profiles respectively.

8. The tables produced at Step 7 were analyzed manually to produce the tables and bar charts which represent the static and dynamic Compiler Demand Profiles in Chapters 11 and 12 respectively.

2. How to Generate a "Compiler Gibson Mix"

Introduction. In this section a methodology is established for defining a "compiler Gibson mix" from a typical Compiler Demand Profile. This methodology is illustrated by actually defining a "compiler Gibson mix" in terms of the static Compiler Demand Profiles for the AED and J3B compilers described in Chapter 11. The static profiles were used rather than the dynamic profiles because the AED and J3B static profiles were more similar than were the corresponding dynamic profiles. (Possible reasons for this result are suggested in Chapter 12.)

The form the "compiler Gibson mix" takes is a system of AED statements with weights specified for the following constituents:

- Individual statements.
- Individual statements within a category or sub-category of statements.
- Sub-categories of statements within a category of statements.
- Categories of statements.
- All statements.

The choice of individual statements, sub-categories, and categories and the weights assigned to these constituent elements, were based on data presented in the bar graphs of Figures 15 through 26 in Chapter 11.

In order to use the "compiler Gibson mix" as defined in this section, each constituent element must be assigned a performance value interpreted as the execution time of an "average" instance of the constituent in the computer environment being "equalized". The procedures to be used to generate actual performance timing values for individual statements are discussed in Section 3. How these values are combined with the assigned weights is described in this section. Ultimately, the result of this calculation of weights combined with values yields a performance value for the "all statements" constituent. This value thus represents the following single characteristic number for the environment:

- The execution time of an average compiler statement.

Main constituents of a "compiler Gibson mix". Presented below is a list of main categories which define the "compiler Gibson mix". For these main categories and for one individual statement, a single underlined letter is used to represent the measured or calculated performance value for the particular constituent.

- $\underline{Z}$: all statements.
- $\underline{A}$: assignment statements.
- $\underline{C}$: procedure calls.
- $\underline{G}$: GOTO statements.
- $\underline{F}$: FOR statements.

- $\underline{I}$: IF statements.
- $\underline{T}$: the single statement X = 0; (where X represents an integer variable).

The categories of statements represented by $\underline{A}$, $\underline{C}$, $\underline{G}$, $\underline{F}$, and $\underline{I}$ will be defined later in this section. Their definitions will be established in such a way that these variables will represent the execution time of an average number of the respective category plus the execution time represented by $\underline{T}$.

These categories are assigned the weights appearing as coefficients in the following equation:

$$\underline{Z} = .38\,\underline{A} + .25\,\underline{C} + .25\,\underline{I} + .10\,\underline{G} + .02\,\underline{F} - 1.00\,\underline{T} \tag{1}$$

The weights for $\underline{A}$, $\underline{C}$, $\underline{I}$, $\underline{G}$, and $\underline{F}$ were approximately determined from the bargraph of Figure 15. Note that the $\underline{I}$ weight is a combination of the IF-THEN and IF-THEN-ELSE frequencies. The weight -1.00 for $\underline{T}$ is chosen because each of the other constituents is defined to include a value of $\underline{T}$, and in combination contribute a combined value equal to the value of $\underline{T}$.

Thus, the variable $\underline{Z}$ can be calculated from Equation 1, provided values can be assigned to each of the six variables on the right hand side. However, as will be shown below, only the values for the variables $\underline{A}$, $\underline{C}$, $\underline{G}$ and $\underline{T}$ will be directly derivable from values for individual statements. The variables $\underline{I}$ and $\underline{F}$ are defined in terms of other constituent categories, including the category represented by $\underline{Z}$ and $\underline{I}$. $\underline{I}$ will be calculated by Equation 2, and $\underline{F}$ will be calculated by Equation 3, and both of these equations include the variables $\underline{Z}$ and $\underline{I}$ on the right hand side. Thus, to arrive at a value for $\underline{Z}$, Equations 1, 2, and 3 will have to be solved simultaneously.

$\underline{T}$: the statement X = 0. The assignment of values to individual statements is discussed in Section 3. Thus this individual statement is assigned a value by a method described in that section.

184

<u>A</u> : <u>assignment statements</u>. Assignment statements are represented by the 23 statements listed in Table 1. In these statements, X, Y, Z, and W all represent integer variables. For each statement a weight is presented. The sum of all weights is 800. Thus, the value of the variable <u>A</u> is calculated as 1/800 the weighted sum of the values of the 23 individual statements. The assignment of values to individual assignment statements is discussed in Section 3.

The assignment of weights to the 23 statements were based on the data presented in the bar charts of Figures 19, 20, and 23. (The literal 17 was arbitrarily chosen to represent a literal other than 1.) Since the data did not separately count "+" and "-", it was assumed that these operators should appear equally. The operators * and /were excluded because their frequency appeared to be relatively insignificant. The statements with a single operator are represented two ways:

- X = X ± (variable or literal).
- X = Y ± (variable or literal).

Since no counts were available to distinguish these forms, and since their values might be significantly different on some computers, equal weights were arbitrarily assigned to them.

One further observation is important to note. The weight assigned to X = 0 includes both the relative weight of 60 as a constituent representative of all assignment statements, but also an additional weight of 400 so that the total value assigned to <u>A</u> includes the average of all constituent assignments plus the single assignment represented by the variable <u>T</u>.

Table 1. Weights for Assignment Statements in "Compiler Gibscn Mix"

Assignment Form	Weight
X = 0	460
X = 17	100
X = Y	160
X = X + 1	5
X = Y + 1	5
X = X - 1	5
X = Y - 1	5
X = X + Y	5
X = Y + Z	5
X = X - Y	5
X = Y - Z	5
X = X + 17	4
X = Y + 17	4
X = X - 17	4
X = Y - 17	4
X = (Y + Z) + 1	2
X = (Y - Z) + 1	2
X = (Y + Z) - 1	2
X = (Y - Z) - 1	2
X = (Y + Z) + W	4
X = (Y - Z) + W	4
X = (Y + Z) - W	4
X = (Y - Z) - W	4
TOTAL	800

C : procedure calls. Procedure calls are represented by the 6 statements listed in Table 2. In these statements, Y_1, Y_2, Y_3, Y_4, and Y_5 represent integer variables, and F represents a procedure name. It is assumed that procedure F consists of the single executable statement X = 0, where X is an integer variable. In this way, the single execution of any of the 6 statements representing the procedure call category automatically includes execution once of the statement represented by the variable $\underline{T}$.

For each statement in Table 16, a weight is presented. The sum of all weights is 100. Thus the value of the variable $\underline{C}$ is calculated as as 1/100 of the weighted sum of the values of the 6 individual statements. The assignment of values to individual procedure call statements is discussed in Section 3.

The assignment of weights to the 6 statements was based on approximating the data presented in the bar graph of Figure 25.

Table 2.   Weights for Procedure Calls in a "Compiler Gibson Mix"

Procedure Call Form	Weight
F ( )	15
F (Y_1)	35
F (Y_1, Y_2)	25
F (Y_1, Y_2, Y_3)	15
F (Y_1, Y_2, Y_3, Y_4)	5
F $(Y_1, Y_2, Y_3, Y_4, Y_5)$	5
TOTAL	100

G : GOTO statement. The GOTO statement category is represented by the single statement GOTO labelname, where labelname is defined as the label of the executable statement $X = 0$. In this way, the value measured for the GOTO statement will also include the execution of the statement represented by the variable $\underline{T}$. The assignment of values to the individual GOTO statement is discussed in Section 3.

We observe that in principle the label used in the GOTO statement could be local, a switch, or a parameter of a procedure (non-local GOTO). However, no counts are available to distinguish these three cases in the data. Since a spot check of source language modules showed that the local label case was by far the most common, it was decided to ignore the contributions of the other two cases.

F : FOR statements. The FOR statement category is represented by a single example combined with values obtained for other categories. The single FOR statement is:

FOR I = 1 STEP 1 UNTIL 10 DO X = 0

where the number of iterations, 10, was chosen arbitrarily, since no data was available on the number of times FOR loops were iteratively executed. The number 10 was chosen because the number of iterations was thought to be significantly greater than 1, perhaps one order of magnitude, and there was no basis to assume that it might be as much as 100 (i.e. two orders of magnitude).

Let $\underline{D}$ represent the value calculated for the above statement by the methods described in Section 3. We then calculate $\underline{F}$ by the following equation.

$$\underline{F} = \underline{D} - 9\underline{T} + 10\,(.58\,\underline{B} + .02\,\underline{C} + .20\underline{A} + .20\underline{I}) \tag{2a}$$

where $\underline{B}$ represents the value of a BEGIN-END block of statements following "DO". The coefficients of $\underline{B}$, $\underline{C}$, $\underline{A}$ and $\underline{I}$ approximate the respective non-zero bars of both AED and J3B profiles in the bar chart of Figure 18. The factor 10 in the equation is the number of iterations spexified for the FOR loop. The term $-9\underline{T}$ reduces $\underline{D}$ by the proper

188

amount so that $\underline{F}$ includes in its value a contribution for one execution of the statement corresponding to $\underline{T}$.

The value of the variable $\underline{B}$ is calculated from the following equation:

$$\underline{B} = \underline{N} \times \underline{Z} \tag{2b}$$

where $\underline{N}$ represents the average number of executable statements in a BEGIN-END block following "DO". No specific data was available on the actual distribution of statement categories in such a BEGIN-END block, and it was arbitrarily assumed that they each may be represented by the average compiler statement with a corresponding value of $\underline{Z}$. $\underline{N}$ is calculated from the equation:

$$.58 \, \underline{N} + .42 = 3.31 \tag{2c}$$

The factor .58 and the term .42 are derived from the same considerations used to determine the coefficients within the parentheses of Equation 2a. The right hand side of 3.31 represents the average number of executable statements nested in FOR loops, as determined from the data in the bar graphs of Figures 13 and 14. $\underline{N}$ is found to be 4.8. Combining this result and Equations 2a and 2b yield the following single equation for evaluating the variable $\underline{F}$:

$$\underline{F} = \underline{D} - 9 \, \underline{T} + 19 \, \underline{Z} + .2 \, \underline{C} + 2\underline{A} + 2 \, \underline{I} \tag{2}$$

Note that in this equation, the only right hand side variables not directly derived from combinations of values for single statements are $\underline{Z}$ and $\underline{I}$.

$\underline{I}$: IF statements. The IF statement category is represented by two sub-categories:

- IF expression THEN U = 0
- IF expression THEN U = 0 ELSE V = 0

where U and V are integer variables.

Let the variables I_1 and I_2 represent the values calculated for these two sub-categories respectively. Then, the value for the variable I is calculated from the following equation:

$$I = .75 (I_1 + .48\underline{B} + .16\underline{C} + .16\underline{G} + .16\underline{A} + .04\underline{T}) \qquad (3a)$$
$$+ .25 (I_2 + .28\underline{B} + .12\underline{C} + .04\underline{G} + .20\underline{A} + .36\underline{I}) - .67\underline{T}$$

where $\underline{B}$ is interpreted as in Equations 2a and 2b. This interpretation of $\underline{B}$ assumes that the BEGIN-END block following "THEN" and "ELSE" contain the same average distribution of constituents as the BEGIN-END block following "DO". This assumption is made arbitrarily in the absence of any better data on the structures of BEGIN-END blocks in different contexts. The coefficients .75 and .25 are derived from the data for the IF-THEN bars and ELSE bars in the bar chart of Figure 15. The coefficients in the parentheses were derived from approximations of the data in Figures 16 and 17. The term -.67$\underline{T}$ is included to subtract off the net contribution for the statement X = 0 contained in the values of the variables $\underline{C}$, $\underline{G}$, $\underline{A}$ and $\underline{I}$.

Combining the value for $\underline{N}$ of 4.8 with Equations 2a, 2b, and 3a, and simplifying, the following equation results:

$$I = .75\,\underline{I}_1 + .25\,\underline{I}_2 - .67\underline{T} + .15\underline{C} + .13\underline{G} + .17\underline{A} + .12\underline{I} + 1.26\underline{Z} \qquad (3)$$

Note that in this equation the only right hand side variables not directly derivable from combinations of values for single statements are $\underline{Z}$ and $\underline{I}$.

The single statements that comprise the two sub-categories are formed by using one of the 17 Boolean expressions listed below in Table 3. For each Boolean expression a weight is given, which represents the weight to be assigned the IF statement in either sub-category using that Boolean expression. The sum of all the weights is 200. Thus each of the two variables I_1 and I_2 is calculated as 1/200 of the weighted sum of the values assigned to the 17 individual statements comprising their respective sub-categories.

Table 3. Weights for Boolean Expressions in IF
Statements in a "Compiler Gibson Mix"

Form of Boolean Expression Used in an IF Statement	Weight
B	50
X == Y	20
X ¬ = Y	20
X == 0	20
X ¬ = 0	20
X == 17	20
X ¬ = 17	20
X == Y OR Z == W	2
X == Y OR Z ¬ = W	2
X ¬ = Y OR Z == W	2
X ¬ = Y OR Z ¬ = W	2
X == Y AND Z == W	2
X == Y AND Z ¬ = W	2
X ¬ = Y AND Z == W	2
X ¬ = Y AND Z ¬ = W	2
X + Y = 17	7
X - Y = 17	7
TOTAL	200

191

In Table 3, B represents a Boolean variable, and X, Y, Z, and W represent integer variables. Weights assigned to each Boolean expression are based on approximations of the data in the bar charts of Figures 21 and 22.

The assignment of values to individual IF statements is discussed in Section 3. Note that in Table 3, the literal 17 is used to represent an arbitrary literal other than 0.

3.   How to Equalize Environments

Overview. In this section a discussion is presented on how a "compiler Gibson mix" (such as the one defined as an example in Section 2) might be used to "equalize" environments. We note that the "compiler Gibson mix" of Section 2 is based on 65 individual statements in the following categories:

- 23 assignment statements (including the special statement X = 0 alone comprising a main category).
- 6 procedure CALL statements.
- 1 GOTO statement.
- 1 FOR statement.
- 17 IF statements without an ELSE clause.
- 17 IF statements with an ELSE clause.

A procedure will be described below for assigning a performance value to a statement which is applicable to all 65 statements. The performance value will represent the CPU time required by a particular computer/operating system environment to execute the functionality represented by the statement. The 65 values derived, one for each statement, may then be combined in the manner described in Section 2 to arrive at a value for the variable Z. This derived value for Z will represent the CPU time required by the computer/operating system environment to execute the functionality represented by an "average" AED statement used in writing a compiler in the AED language.

Let us suppose that a series of test programs (as described in Chapter 9) have been compiled with compiler 1 in computer/operating system environment E_1, and the same test programs are compiled with

compiler 2 in environment E_2. Let us suppose further that the compiler
execution times of these test compilations have been measured, and that
the results have been organized into two Compiler Performance Profiles,
say P_1 and P_2 respectively, using the methods described in Chapter 8.
Then, the two Compiler Performance Profiles can be normalized for
their respective environments (i.e. their environments can be "equalized")
by dividing each component of P_1 by the $\underline{Z}$ value derived for environment
E_1, and dividing each component of P_2 by the $\underline{Z}$ value derived for environ-
ment E_2. The resulting normalized profiles now reflect the performance
of the compiler in a "standard" environment.

We note that the procedure described above "equalizes" environments
with respect to processor speed, instruction set (see below), and one
aspect of operating system support (see below). However, no account
has been taken of differences in memory size. In the course of this study
it became clear that the performance of compilers on similar computers
with different memory sizes was in general very complex, and would
require as much additional study as that undertaken in the present study
for 'equalizing" what is primarily processor speed and instruction set.
The investigation did suggest the following observations:

- A one-pass compiler either fits in core or it does not. If it
  does, then the available core not occupied by the compiler
  code and fixed data structures will place a limit on the
  size of program that can be compiled. There are no other
  performance trade-off considerations for a one-pass compiler.

- A multi-pass compiler, such as that described in Section 4
  of Chapter 3, can be organized to operate in a minimum
  quantity of core, by using a relatively large number of
  phases. In such an environment, the throughput of a
  compilation will depend on the I/O channel and device
  operational time used for overlays in addition to CPU
  time.

- It is theoretically possible to balance the space/time trade
  off in architecting a multi-pass compiler in fewer phases
  that are used in the architecture of Section 4 of Chapter 3.

Instruction set differences are taken into account by the method
in which performance values are assigned to individual statements.
This method is discussed below. The one aspect of operating system

support taken into account is the mechanism used for a standard calling sequence interface in procedure or function calls. Other aspects of operating system support will be discussed briefly in Section 4.

Assigning values to statements. The following method can be used to assign a performance value to a statement which will represent the CPU time required in a particular environment for executing the functionality represented by the statement. This method is applicable to all 65 statements listed above. However, certain special considerations are important to take into account for IF, GOTO, and call statements. These special considerations are discussed later in this section.

First choose two integers n and m, (say $m > n$) which will be used in the manner described below. Then have a good coder code the functionality represented in the statement in assembly language as efficiently as he possibly can. This use of a good coder and assembly language will enable the functionality to be executed using the full power of the available instruction set, thereby "equalizing" for instruction set as well as for processor speed. Next two assembly programs are produced which are identical except for the inclusion in one of n replications of the coding for the statement, and in the other of m replications of this coding. (If labels appear in the coding, then new labels will of course have to be introduced for each replication.) These assembly language programs will include whatever declarations and initializations of variable used that are required, and whatever other special assembly language elements are required to create programs that will assemble executable object code.

It is obvious that the difference in execution times of the two programs will reflect $m-n$ executions of the statement. Therefore $1/(m-n)$ of this difference is the performance value to be assigned to the statement. In order to get meaningful time measurements, m and n must be sufficiently large, and m must be sufficiently larger than n, for the granularity of the clock used in the measurements to be insignificant. Generally m should be at least $2\ n$, and probably $3\ n$ to $10\ n$ would be better. The reason two runs are used, one each for n and m replications,

194

rather than a single run — to eliminate any overhead appearing in the program for such functions as location of work space, initialization, etc.

IF statements. In assigning values to IF statements, several different pairs of runs should be made for each statement. Each of the pairs of runs (having $\underline{n}$ and $\underline{m}$ replications) would use different combinations of initializations for the variables B, X, Y, Z and W. Since no data is available as to the relative frequency, the Boolean expressions in IF statements evaluated to TRUE or FLASE, arbitrarily the combinations of initializations should be such that TRUE and FALSE evaluations occur equally often. The value assigned to an IF statement would be derived from the average of the values for each pair of runs.

GOTO statements. In order to avoid possible peculiar execution patterns resulting from the chaining of GOTO statements, each GOTO statement should specify a label of a statement of the form X = 0, as follows:

GOTO labelname;

labelname: X = 0;

It is the assembly language code for this pair of statements that is replicated $\underline{n}$ and $\underline{m}$ times. Then the value assigned to the GOTO statement is calculated as:

$$\underline{G} = (1/(\underline{m}-\underline{n})) \, (t_m - t_n)$$

where t_m and t_n represent the measured execution times for the programs with $\underline{m}$ and $\underline{n}$ replications, respectively. This definition includes the time to execute the statement corresponding to $\underline{T}$ in the valuation of $\underline{G}$.

CALL statements. In coding the call statements, the coder must take the following considerations into account.

- If a convention is established for calling sequences in procedure and function calls which is to be followed by the object code of a compiler operating in the environment being "equalized" then this convention must be followed in coding the six CALL statements listed in Section 2.

- For each CALL statement, a separate program must be coded. This program will consist of an assembly language version of a procedure with the appropriate number of arguments, following all established conventions for parameter passing, and consisting of the assembly language version of the single executable statement X = 0. Each occurrence of the procedure call in the pair of test programs (with n and m replications of the procedure call) will be a call to the special assembly language procedure described above.

4.   Timing Data and "Equalizing" Environments

Introduction. In addition to the development of static and dynamic Compiler Demand Profiles, as described in Chapters 11 and 12 and generated as described in Section 1, a parallel activity was followed to explore the possibility of establishing timing data as components of dynamic profiles. The results of this parallel activity were such that no useful timing comparisons were possible between the AED and J3B compilers, since certain bottleneck routines in the AED support library (used by both compilers) had been rewritten for J3B to be more efficient. The procedure followed in collecting the timing data is described below, and a few interesting observations on the resulting data are summarized at the end of this section.

Collecting timing data. One test program was compiled by modified versions of the AED and J3B compilers. These modified compilers used the computer's 13 microsecond clock to measure the total time spent in each procedure of the compilers and library routines used by the compilers. The test program was the smaller of the two programs used to generate dynamic Compiler Demand Profiles as described in Chapter 12. The listings of these test programs are included with Appendix 2 of this report.

The timing measurements were output to a file which were processed by a separate program to produce a suitably formatted report.

Interpreting the timing data. The reports generated in the manner described above were further analyzed by hand calculations. The following interesting patterns were found.

- Input and output related CPU processing of characters occupied about 40% of the total CPU time used to compile the test program by both the AED and J3B compilers. In the AED compiler, this broke down as 20% input and 20% output; in J3B, it was 15% input and 25% output.

- Dynamic core management used an additional 20% of the CPU time in AED, and 15% of the time in J3B.

- Intermediate input/output between compiler passes used only 2-3% of the total time.

Included in input/output of characters is time spent in conversion between internal and external formats and character transmission codes, as well as buffer management, etc. It was discovered that the reduced time for input in J3B was due to the rewriting for J3B of the library routines used to perform these functions in the AED compiler. The high fraction of time for output in J3B was due to an increased quantity of output generated by this compiler, including set/used listings, environment listings, etc., which are not output by the AED compiler.

Due to the significant rework of several support functions for J3B for compiler efficiency, it was therefore determined that the above data was not useful for "equalizing" environments considerations. However, the large amounts of time spent in character input/output, the significant savings that can be accomplished by using specially programmed support subroutines in these areas, and the relatively small time spent in intermediate (between passes) input/output, should be of interest to compiler designers and implementors. In this light, it should also be noted that the time spent in dynamic core management could be significantly less in a compiler with a fixed core utilization scheme, but this type of design leads to a large, fixed sized, rigid compiler.

One final observation is relevant here. Since there appears to be a significant function of time spent performing these functions, further study appears to be desireable to determine how differences in the operating system support of character string I/O and dynamic core management might be "equalized" between environments.

197

CHAPTER 11

STATIC COMPILER DEMAND PROFILE DATA

1. Overview

This chapter presents all static data gathered from the AED and
JOVIAL/J3B compilers. The next chapter entitled "Dynamic Compiler
Demand Profile Data" (Chapter 12) contains a parallel set of data in
which dynamic weighting factors are applied to the static information.
The two chapters taken together present a summary of all non-timing
data gathered during the project.

The information is grouped into three major sections as follows:

- Tables of the AED and J3B compilers' usage of AED
  language forms.

- Bar chart representations of various histograms of
  usage patterns.

- Bar chart of the relative percentage of usage of various
  AED language forms.

Each of the tables presents a logical group of data about the
types of and form of AED statements used in the two compilers. Numbers
presented in the tables were derived by compiling each individual source
program module of each compiler through a special "instrumented" AED
compiler which tested for certain predefined language constructs and
which generated output totals for each module. A second, summation
program was later run on these sub-totals to calculate grand totals
for each of the two compilers. (The details of the data collection and
reduction methods used are presented in Appendix 1.) Each compiler
contains about 450 procedures, with J3B being the larger of the two.
Library support routines are not included in the statistics.

Each of the tables presents the number of occurrences of various
statement and clause types in the context of other statement and clause
types. In addition, the number of occurrences of various arithmetic
and Boolean operators, and various forms of arithmetic and Boolean
expressions are also presented. The following is a list of the subjects
of the tables presented in Section 2.

- Statement types and total arithmetic and Boolean operators and labels.

- Assignment statement forms.

- Forms of statements used with conditionals and loops.

- Form of loop statement clauses.

- Forms of integers used in loops.

- Forms of arithmetic phrases.

- Forms of Boolean phrases.

The bar chart histograms of usage patterns are presented in Section 3. They consist of a series of histograms showing the number of occurrances in the compiler of a particular AED language form having n occurrences of a particular constituent component. The following is a list of the forms and constituents for the histograms:

- Number of procedure calls having n arguments.

- Number of assignment statements having n R.H.S. (right-hand-side) operators.

- Number of boolean expressions in IF clauses containing n operators.

- Number of FOR statements nested n deep.

- Number of executable statements nested n deep in FOR loops.

In Section 4, the relative percentage bar charts are presented. These bar charts show the relative percentage of occurrences of various statement and clause types, alone and in specified contexts. In particular, histograms are presented which show the percentage of occurrences of a statement or clause type with n occurrences of a specific constuent component. The following is a list of the subjects of the bar charts presented in Section 4.

- % of Use of Statement Types.

- % of Statement Types used following 'THEN'.

- % of Statement Types used following 'ELSE'.

- % of Statement Types used following 'DO'.

- % of Use of Arithmetic Operators.

- % of Use of Arithmetic Forms.

199

- % of Use of Boolean Operators.
- % of Use of Boolean Forms.
- % of Arithmetic Assignments Used with n RHS Operators.
- % of Boolean Expressions in IF Clauses used with n Operators.
- % of Procedure and Function Calls used with n Arguments.
- % of Executable Statements used Nested n Levels Deep in 'FOR' Loops.

Further details are given in Sections 2, 3, and 4. Conclusions and other textual discussion is presented with each individual table and bar chart, are also summarized in Chapter 14.

2. **Tables of Static Usage of AED Language Forms in the AED and J3B Compilers**

In this section, seven tables are presented which show the comparative static usage by the AED and J3B compilers of various statement and clause types, and the forms of arithmetic and Boolean expressions of the AED language. The subject matter with which the tables deal were listed in Section 1. With each table, interpretive information is presented to explain the contents of the table. In addition, observations are offered concerning the pattern of numerical results presented.

Statement types, operators, and labels. Table 4 summarizes for both the AED and J3B compilers. The number of occurrences of of each of five statement types (assignments, IF, FOR, calls of procedures or functions, and GOTO). Also, the total number of arithmetic operators and Boolean operators occurring and the total number of labels defined are also summarized.

The statement count was gathered by counting occurrences of the statement terminator symbol ";" (or the equivalent ("$, "). Since this statement terminator is not always required (such as preceeding the block-termination symbol "END"), the count is somewhat low, but nonetheless comparable between the two compilers. The Boolean operation count is also somewhat low, since it includes only those

200

Table 4. Summary of Static Usage of Statement Types, Operators, and Labels

	J3B	AED
ASSIGNMENT STATEMENTS	3046	3093
IF STATEMENTS	1295	1352
FOR STATEMENTS	53	70
CALL STATEMENTS AND FUNCTION CALLS	2165	2159
GOTO STATEMENTS	820	763
ARITHMETIC OPERATIONS	859	545
BOOLEAN OPERATIONS	1512	1392
LABELS	773	801

Boolean forms following an "IF", and not those forms used in Boolean assignment statements (e.g. "BOOL1 = BOOL2 <u>AND</u> BOOL3"). However, the great majority of the use of Boolean operators do follow an "IF", and the count presented should be quite close to the actual total.

It is evident from the numbers presented that the two compilers are remarkably close in all categories shown. Assignment statements and call statements are the most heavily used statement type, whereas looping ("FOR") statements are rarely used as compared with the conditional ("IF") forms. Also, arithmetic operations are used much less frequently than Boolean operations. This is to be expected with a "system program" such as a compiler. In scientific and engineering applications, on the other hand, Knuth has found that among a large number of FORTRAN programs, arithmetic forms are much more common than Boolean forms.

<u>Assignment statement forms</u>. Table 5 summarizes the static usages of selected sub-categories of assignment statements. It is particularly interesting to note the extremely heavy use of the form "A = B" as compared to all other forms. It should be noted that this sub-category includes all combinations of such constructions as "COMP1 (PTR) = COMP2 (PTR)", "COMP3 (PTR) = X" etc. These constructions are heavily used in both the AED and J3B compilers for manipulating the compiler's data structures. However, even in compilers without this pointer structure capability, the simple assignment category is expected to be very heavily used. In general, the profile of usage by sub-categories is remarkably similar between the two compilers.

<u>Statement types used with conditionals and loops</u>. Table 6 presents a summary of the static usages of five statement types (BEGIN, procedure call, FOR, IF, GOTO, assignment) in the three contexts immediately following the three AED keywords: THEN, ELSE, and DO. THEN and ELSE are, of course, keywords which begin conditionally executed clauses; DO establishes the iterative loop of a FOR statement.

202

Table 5. Summary of Static Usage of Assignment
Statement Forms

	J3B	AED
A = 0	176	218
A = 1	115	115
A = LITERAL (NOT 0 OR 1)	407	235
A = B	1262	1397
A = EXPRESSION	700	701
A = FUNCTION (.....)	386	427
TOTAL	3046	3093

203

Table 4. Summary of Static Usage of Statement Types Used with Conditionals and Loops

FORM OF STATEMENT FOLLOWING	THEN		ELSE		DO	
	J3B	AED	J3B	AED	J3B	AED
BEGIN	612	473	78	130	26	32
CALL	131	262	29	53	1	6
FOR	0	9	0	0	0	0
IF	43	73	165	109	3	23
GOTO	148	190	14	33	0	0
= (ASSIGNMENT)	143	177	82	71	20	9
TOTALS	1077	1184	368	396	50	70

The numbers presented in Table 6 again show the high degree of profile similarity between the AED and J3B compilers. The following is a list of observable attributes of the two profiles:

- Compound statements (BEGIN) most frequently follow THEN and DO in both profiles.

- The form ELSE IF shows significant prominence. It exceeds the number of occurrences of BEGIN in J3B (165/78) and approximates the BEGIN's in AED (109/130). This suggests the general popularity of using chains of IF - THEN - ELSE - IF combinations to express complex conditions. In comparison, the IF - THEN - IF form is used much less frequently, suggesting that Boolean phrases (e.g. BOOL1 <u>AND</u> BOOL2 <u>OR</u> BOOL3) are commonly used to select the precise THEN condition when needed.

- Little if any use is made of loops (FOR) following THEN ELSE or DO, although they may be hidden from our statistics by being written within BEGIN - END blocks.

<u>FOR statement sub-types</u>. Table 7 presents the number of occurrences of each of the following four sub-types of FOR statements:

- Using UNTIL clauses.

- Using WHILE clauses.

- Using multiple iteration lists.

- Other.

The AED and J3B profiles are again seen to very similar. Note that the UNTIL clause, which tests for a specific numeric value of the index variable, is the most popular form, even though the more general WHILE clause could be used to handle all UNTIL conditions, although with slightly more work by the user. Multiple iteration lists come in a poor third, showing that loops are seldom controlled by lists of multiple conditions.

<u>Integer forms in FOR statements</u>. Table 8 presents the number of occurrences of four forms that are used in each of three integer variable (or expression) positions that occur in FOR statement constructions. The four integer forms are:

Table 7. Summary of Static Usage of FOR Statement Sub-Types

	J3B	AED
UNTIL CLAUSES	33	36
WHILE CLAUSES	18	24
MULTIPLE ITERATION LISTS	2	10
OTHERS	0	0
TOTAL	53	70

Table 8. Summary of Static Usage of Integer Forms in FOR Statements

	"STEP" LEFT OF		RIGHT OF		"UNTIL" AFTER	
	J3B	AED	J3B	AED	J3B	AED
1	6	18	42	37	2	0
LITERAL (NOT 1)	25	18	1	2	5	14
A	9	2	0	0	16	20
EXPRESSION	4	1	1	0	10	2
TOTAL	44	39	44	39	33	36

- The literal 1.

- Literals other than 1.

- An integer variable (represented as "A").

- Integer expressions.

The three contexts with FOR statements in which the integer forms occur are the following:

- Left of STEP. This context specifies the first value of the index variable to be used in the loop.

- Right of STEP. This context specifies the increment (assumed positive) which is added to the index variable for each successive iteration of the loop.

- After UNTIL. This specifies the limit value of the index value used in the loop. (Note that a GREATER THAN test is made, and this limit value might be exceeded by the last increasing of the index value. Thus the limit value might not be used in the loop.

Here, the AED and J3B profiles show some clear differences, although there is also evident significant similarities. In J3B, the initial value of the index variable is most commonly a literal not equal to 1 (25/6), while in AED the split between 1 and other literals is even (18/18). In both compilers, the incrementing value (RIGHT OF STEP) is almost always 1. The terminating test value (AFTER UNTIL) is usually a variable (16 in J3B, 20 in AED). For the second most common form AFTER UNTIL, an expression (10), while AED makes more use of a literal (14).

Arithmetic forms. Table 9 presents the number of occurrences of ten arithmetic forms that appear in arithmetic expressions. Each form involves one of the following three groups of operators:

- + (plus) or - (minus),

- * (multipled by), or

- / (divided by).

The ten arithmetic forms are listed below. In these forms, "A" and "B" denote arithmetic variables. In collecting the data, the actual data type of the variables were ignored. However, in spot checking

208

Table 9. Summary of Static Usage of Arithmetic Forms

	+ OR -		*		/	
	J3B	AED	J3B	AED	J3B	AED
A OP (1) (2)	(1) 240	(1) 197	(2) 0	(2) 0	(2) 3	(2) 2
(1) (2) OP A	(1) 4	(1) 1	(2) 5	(2) 1	(1) 0	(1) 0
A OP LITERAL	126	155	10	2	13	5
LITERAL OP A	67	9	1	1	1	1
A OP B	280	161	12	0	6	0
EXPRESSION OP (1) (2)	(1) 53	(1) 26	(2) 0	(2) 0	(2) 1	(2) 1
(1) (2) OP EXPRESSION	(1) 0	(1) 1	(2) 0	(2) 0	(1) 0	(1) 0
EXPRESSION OP LITERAL	11	4	12	8	19	6
LITERAL OP EXPRESSION	6	3	3	1	0	0
OTHER	107	73	22	10	1	16
TOTAL	894	630	65	23	44	31

209

the source language programs we found only integers used in arithmetic forms. The ten forms are:

(1) A operator specific literal (1 or 2). The particular literal depends upon the operator used in the form: 1 is used with + and -; 2 is used with * and /.

(2) Specific literal (1 or 2) operator A. Here 1 is used with +, -, and /; 2 is used with *.

(3) A operator literal other than the specific literal used in (1) above.

(4) Literal (other than that used in (2) above) operator A.

(5) A operator B.

(6) Arithmetic expression operator specific literal (1 or 2). The literal values are as in (1) above.

(7) Specific literal (1 or 2) operator arithmetic expression. The literal values are as in (2) above.

(8) Arithmetic expression operator literal other than that used in (6) above.

(9) Literal (other than that used in (7) above) operator arithmetic expression.

(10) Other forms than the above.

The following three specific forms are seen to be by far the most frequently used:

- A ± 1,
- A ± literal other than 1, and
- A ± B.

The inverse form of the last two (1 + A or other literal +A) are rarely used, indicating that most programmers write expressions of the form "A ± 1" and A ± n rather than "1 ± A" and n ± A even though both forms are logically equivalent.

The fourth most frequently used form is the "OTHER" uses of the + operator. The "OTHER" form includes all forms not specified in forms (1) through (10). However, probably the form most frequently used in this category is:

- Arithmetic expression operator arithmetic expression (e.g. (A+B) * (C+D).)

The form EXPRESSION $\pm$ 1 also receives high usage, although usage of
the form literal (other than 1) $\pm$ A exceeds this in the J3B compiler.
Note also that * and / are rarely used as compares with + or -. This
is to be expected in system programs in contrast with scientific or
engineering programs.

Overall, is again evident that the AED and J3B profiles are
very similar.

<u>Boolean forms</u>. Table 10 presents the number of occurrences of
ten Boolean forms that appear in Boolean expressions within IF state-
ments. Each form involves one of the following four groups of predicates
or the Boolean connectives AND or OR:

- Equals (= =).
- Greater than, or not less than (> or >=).
- Less than, or not greater than (< or <=).
- Not equal (¬ =).

The ten Boolean forms are listed below. In these forms, "A" and "B"
denote variables. Spot checking of the use of variables with predicates
found no examples other than arithmetic integer data types. When
used with a Boolean connective, AND or OR, "A" and "B" denote Boolean
variables. Forms (1), (2), (6) and (7) are not applicable to the AND
and OR connectives. The ten forms are:

(1) Variable <u>predicate</u> 0.

(2) 0 <u>predicate</u> variable.

(3) Variable <u>predicate</u> literal other than 0, or Boolean
    variable <u>predicate</u> Boolean literal (TRUE or FALSE).

(4) Literal (other than 0) <u>predicate</u> variable, or Boolean
    literal (TRUE or FALSE) <u>predicate</u> Boolean variable.

(5) Variable <u>predicate</u> variable, or Boolean variable
    <u>connective</u> (AND or OR) Boolean variable.

(6) Expression <u>predicate</u> 0.

(7) 0 <u>predicate</u> expression.

(8) Expression <u>predicate</u> literal other than 0, or Boolean
    expression <u>connective</u> (AND or OR) Boolean literal
    (TRUE or FALSE).

Table 10. Summary of Static Usage of Boolean Forms

	=		> ≥		< ≤		¬=		AND		OR	
	J3B	AED	J3B	AED	J3B	AED	J3B	AED	J3B	AED	J3B	AED
A OP 0	62	122	15	11	70	13	48	89	0	0	0	0
0 OP A	0	0	0	0	0	0	0	0	0	0	0	0
A OP LITERAL	547	110	63	44	36	27	91	31	0	0	0	0
LITERAL OP A	0	0	0	0	0	0	0	0	0	0	0	0
A OP B	45	159	42	22	25	14	32	115	0	13	3	55
EXPRESSION OP 0	17	43	4	8	6	12	27	42	0	0	0	0
0 OP EXPRESSION	0	0	0	0	0	0	0	0	0	0	0	0
EXPRESSION OP LITERAL	44	6	6	6	2	3	8	0	0	0	0	0
LITERAL OP EXPRESSION	0	0	0	0	0	0	0	0	0	0	0	0
OTHER	15	48	29	15	9	10	28	28	103	188	176	110
TOTAL	730	488	159	106	148	79	234	305	103	201	179	165

212

(9) Literal (other than 0) <u>predicate</u> expression, or Boolean
     literal (TRUE or FALSE) <u>connective</u> (AND or OR)
     Boolean expression.

(10) Other forms than the above.

Forms (1), (3) and (5), especially using the predicate = = are
seen to be by far the most frequently used. However, J3B appears to
use the form A == literal (other than 0) much more frequently than
A == 0 or A == B, and AED uses these three forms in roughly the same
numbers. These results again indicate a stylistic preference for the
form A == 0 rather than the functionally equivalent 0 == A.

Form 10 (OTHER) involving predicates uses these predicates in
a roughly equivalent fashion (numbers range from 9 to 29), except for
an AED preference for the == predicate (48 occurrences). These
"OTHER" forms would probably be most frequently represented by the
form:

- Expression == expression;
  (e.g. (A+B) == (C+D)).

Forms (6) and (8) receive some slight use and again the use of
these forms with the four groups of predicates are all roughly equivalent
except for a few cases: AED prefers the forms expression == 0 (43
occurrences) and expression $\neg$ = 0 (42 occurrences). While J3B prefers
the form expression == literal other than 0 (44 occurrences), and to
some extent the form expression == 0 (27 occurrences).

In form 10, the uses of AND and OR are used roughly equally by
both AED and J3B; AED preferring AND (188 to 110) and J3B preferring
OR (176 to 103). J3B has almost no other use of these connective and
AED has a comparatively modest use with form (5). A spot check of the
source language code shows that the most frequent use of AND and OR
are in expressions like the following examples·

- A == B AND C == D,
- A == B OR C == D.

Other predicates than == occur in these forms, and B and/or D are frequently
replaced by literals.

213

By far the most frequently used predicate is == in both compilers with the second most frequent, ¬ =. The other two predicates are used somewhat less and are more-or-less used with approximately equal frequency.

The surprising lack of any usage whatsoever of certain forms prompted a review and retest of the data gathering tools to insure their proper functioning. The results of this activity confirmed that the tools were operating correctly and that the complete lack of usage of these forms is correctly reported. In particular, it was noted that the only use of the operators AND and OR out of all the forms tested (except form 10) was the form "variable OR variable" and "variable AND variable".

Overall, Table 10 again shows the AED and J3B profiles to have a great deal of similarity with a few differences of a stylistic character.

3.   Bar Charts of Histograms of Static Usage Patterns in the AED and J3B Compilers

In this section five histograms are presented which show the comparative static usage by the AED and J3B compilers of a variable number, n, of constituent components within five categories of statements. The subject matter with which these histograms deal were listed in Section 1. With each histogram interpretive information is presented to explain the data in the histogram. In addition, observations are offered concerning the patterns shown in the histograms.

Procedure and function calls using n arguments. This histogram (Figure 10) shows that the great majority of procedure calls have five or fewer arguments, with one argument being the most common. Calls to functions as well as procedures are included in the data. The overall impression is clearly one of great similarity between the AED and J3B profiles with respect to the use of arguments within procedure calls.
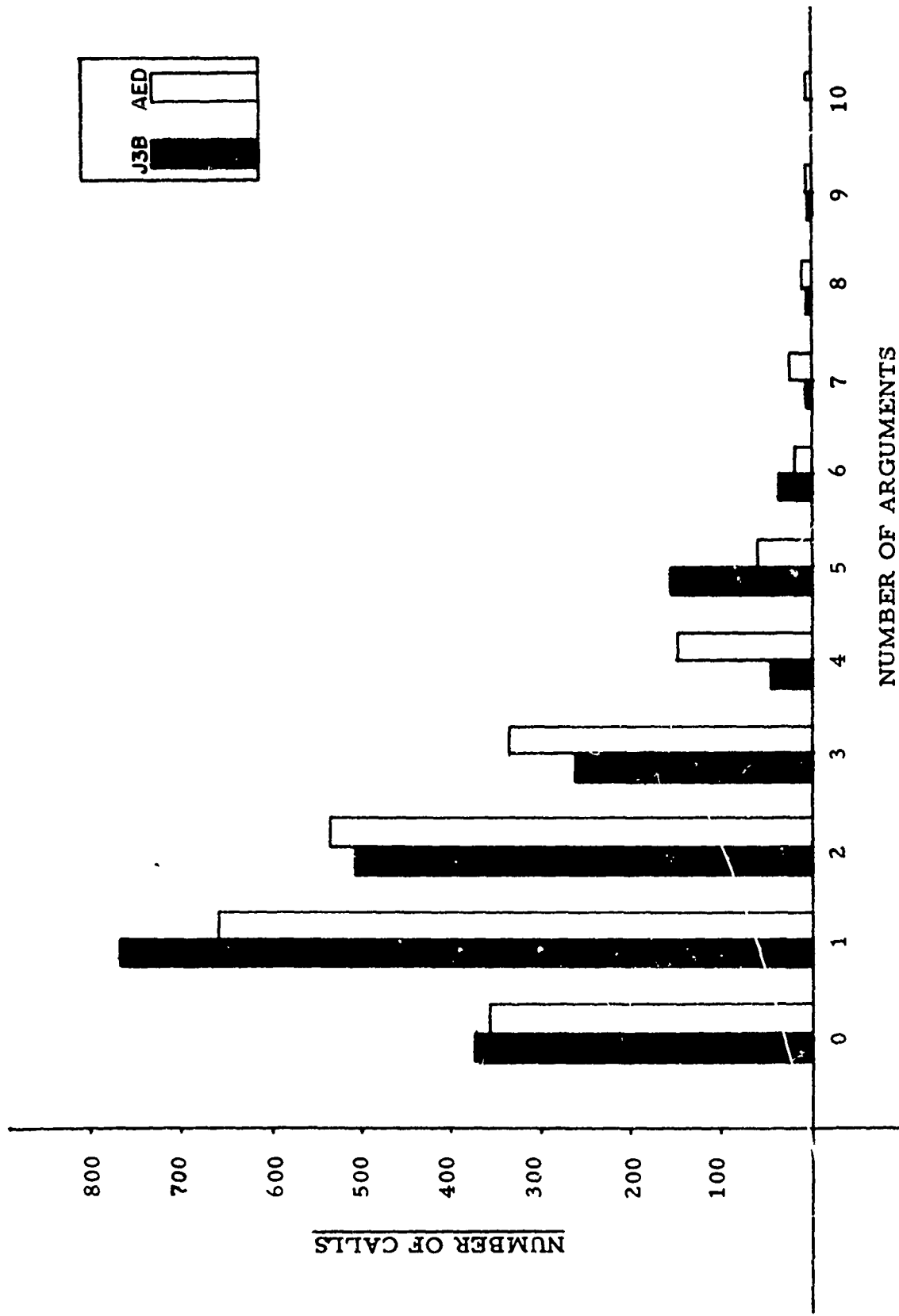
Figure 10. Histogram of Static Occurrences of Procedure and Function Calls Using n Arguments

215

Assignment statements with $\underline{n}$ right hand side operators. The
data presented in this histogram (Figure 11) counted only the arithmetic
operators (+, -, *, and /) in assignment statements. However, a spot
check of the source language code found very few examples of other
operators used in assignment statements. There were some occurrances
of the form:

- $A = F (...)$,

Where a function call is invoked on the right hand side (R. H. S.) and the
returned value is assigned to the variable on the left hand side. Occurrences
of this form would be countered as 0 R. H. S. operators, but they are very
rare compared to the forms:

- $A = $ literal, and
- $A = B$.

This histogram clearly shows the overwhelming preponderance
of simple assignment statements, having no R. H. S. operators. In
fact, very few statements have more than one operator. The two com-
pilers show very similar profiles with respect to this category.

Boolean expressions with $\underline{n}$ operators. The Boolean expressions
counted in the data shown in this histogram (Figure 12) were all in IF
clauses. However, a spot check of the source language programs
showed other occurrences of Boolean expressions (for example in
assignment statements) to be very rare in comparison to the counted
occurrences. The operators countered were six predicates (==, <,
<=, >, >=, ¬ =) and two connectives (AND, OR).

Here again the AED and J3B profiles are seen to be very similar.
An interesting pattern of heavier usage of odd numbers of operators
(1, 3, 5, etc.) is evident. This is reasonable, considering the fact that
many Boolean expressions are of the forms:

- IF A == B   (1 op)
- IF A == B OR C == D   (3 ops)
- IF A == B OR C == D OR E == F   (5 ops)
  etc.

216
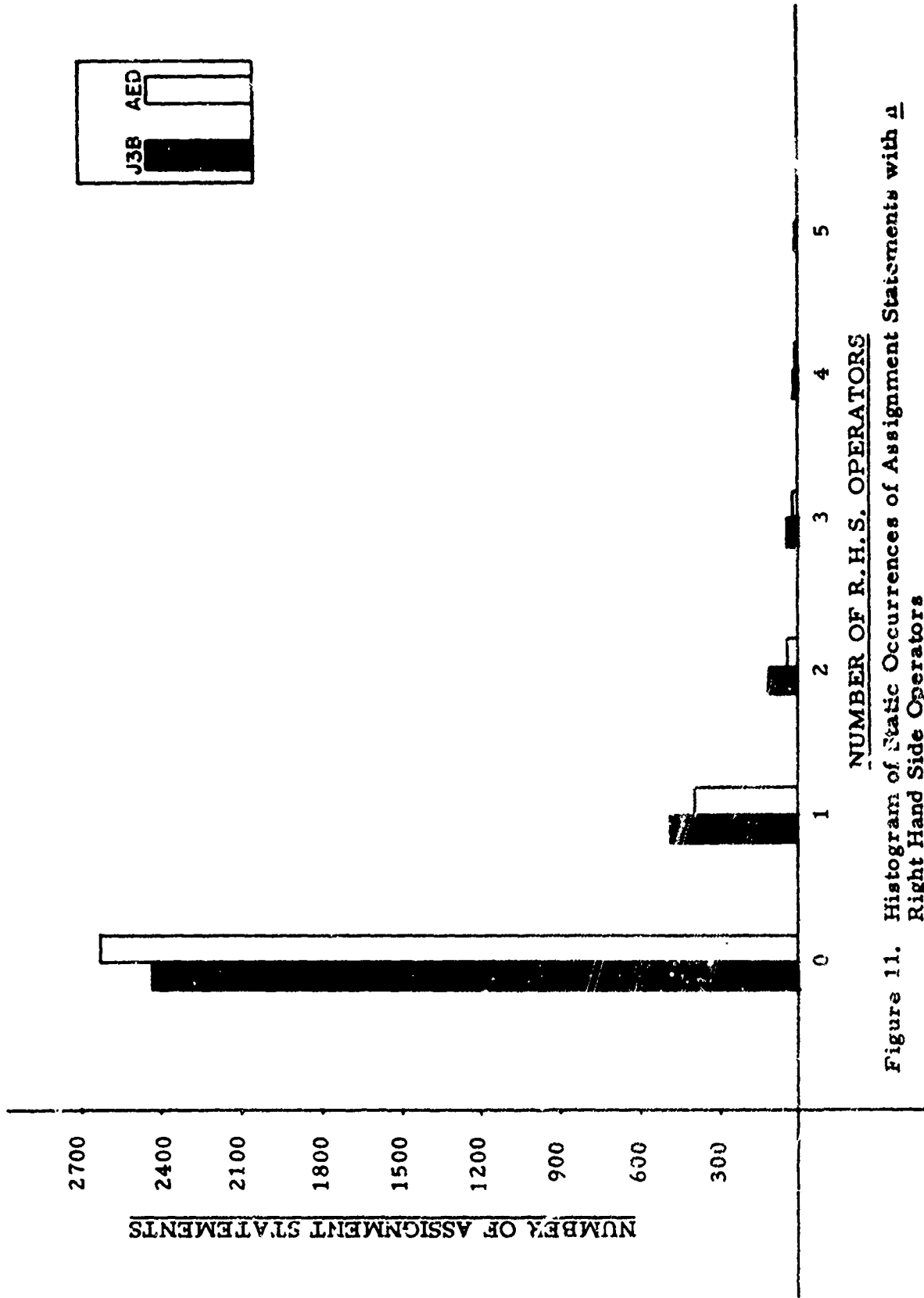
Figure 11. Histogram of Static Occurrences of Assignment Statements with a Right Hand Side Operators
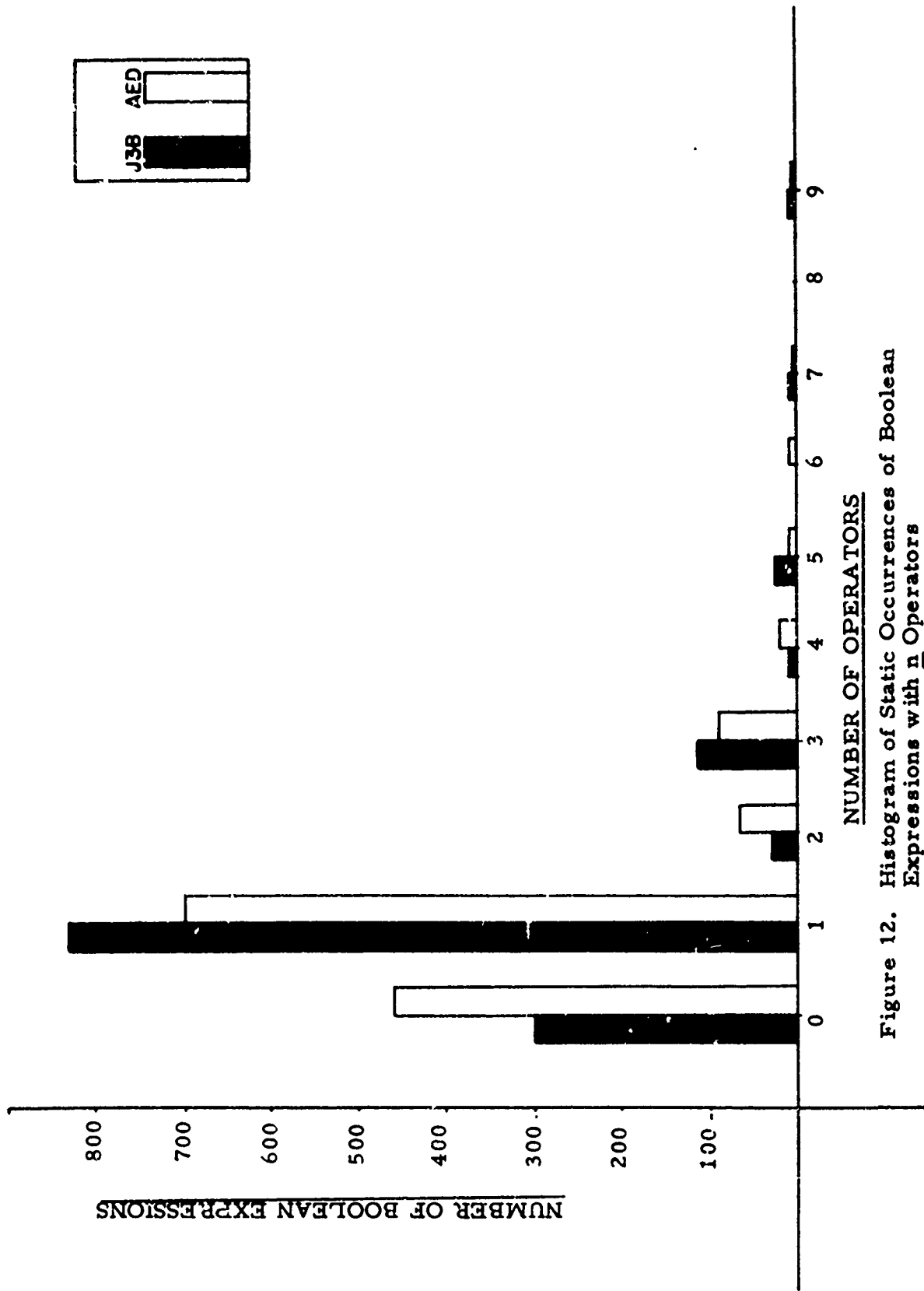
Figure 12. Histogram of Static Occurrences of Boolean Expressions with $\underline{n}$ Operators

The histograms also show that by far the greatest number of expressions are of the two forms:

- IF A   (0 operators)
- IF A OP B   (1 operator)

with the second form (1 operator) being the most common. From the data presented in Section 2, it is clear that the operator used here is a predicate, and most usually ==.

FOR loops and executable statements nested n deep. Both histograms dealing with FOR loops (Figures 13 and 14) show clearly that multiple nesting of loops is very rare in both compilers. Both J3B and AED contains a few nested loops of level 2; J3B contains one 3-level nest, and AED contains none. Another observation seen by comparing the two histograms with each is that there are about three executable statements on the average in a FOR loop.

Both histograms again show a great similarity in the AED and J3B profiles.

4.   Bar Charts of Frequency Histograms of Relative Static Usage
        of AED Language Forms

In this section twleve bar graphs are presented which show the comparative percentage of static usage for various AED language elements in the source language code of AED and J3B compilers. Four of the bar graphs (Figures 15, 16, 17, and 13) show the percentage of static usage among groups of statements and clause types alone and in contexts. The next four bar graphs (Figures 19, 20, 21, and 22) show the percentage of usage of arithmetic and Boolean operators and forms. The last four bar charts (Figures 23, 24, 25, and 26) present histograms of the percentage of static usage of four AED language contexts involving a variable number, n, of a specified constituent. The twelve subject matters with which these twelve bar charts deal were listed in Section 1.

219
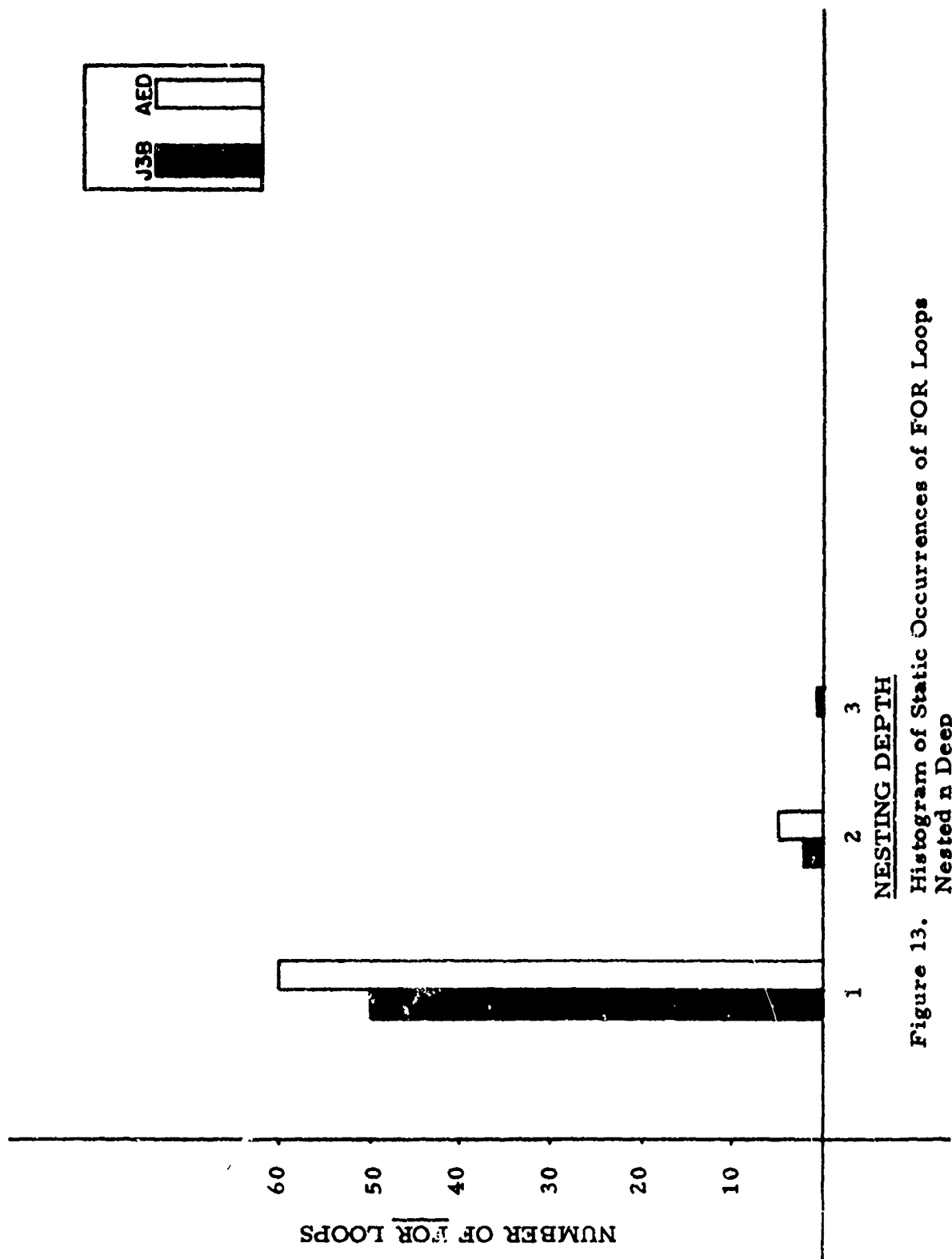
Figure 13. Histogram of Static Occurrences of FOR Loops Nested n Deep
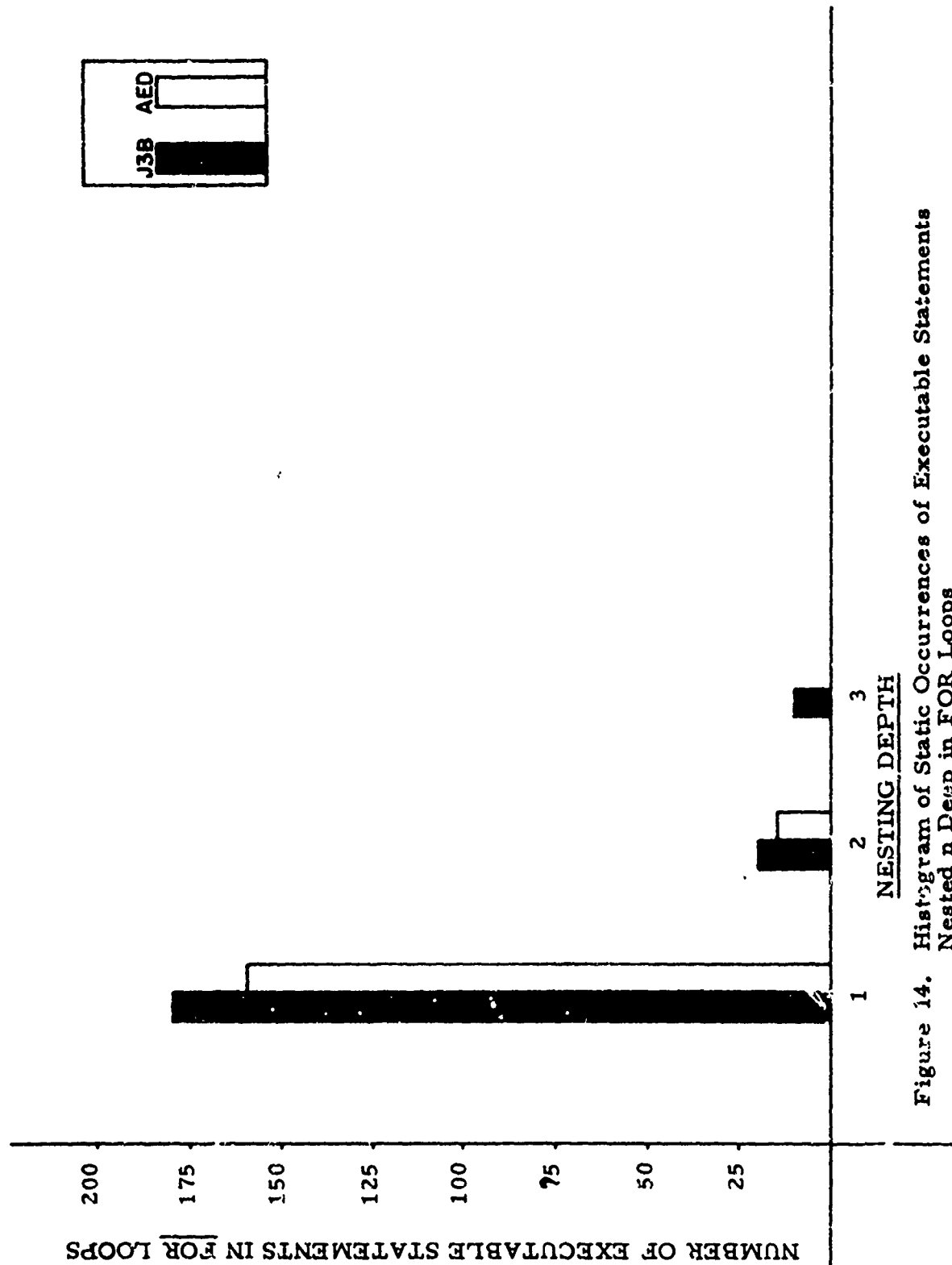
Figure 14. Histogram of Static Occurrences of Executable Statements Nested n Deep in FOR Loops
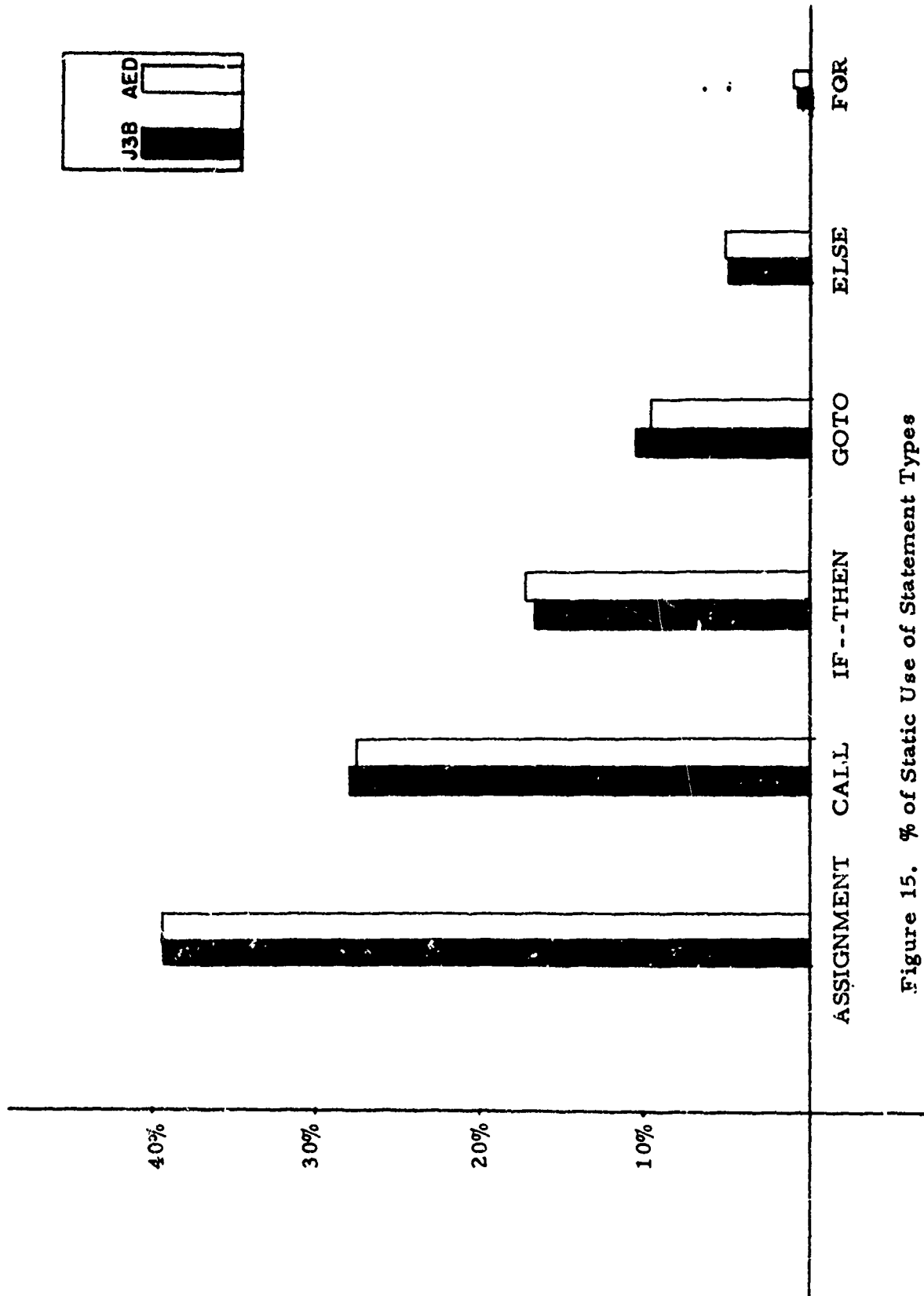
Figure 15. % of Static Use of Statement Types
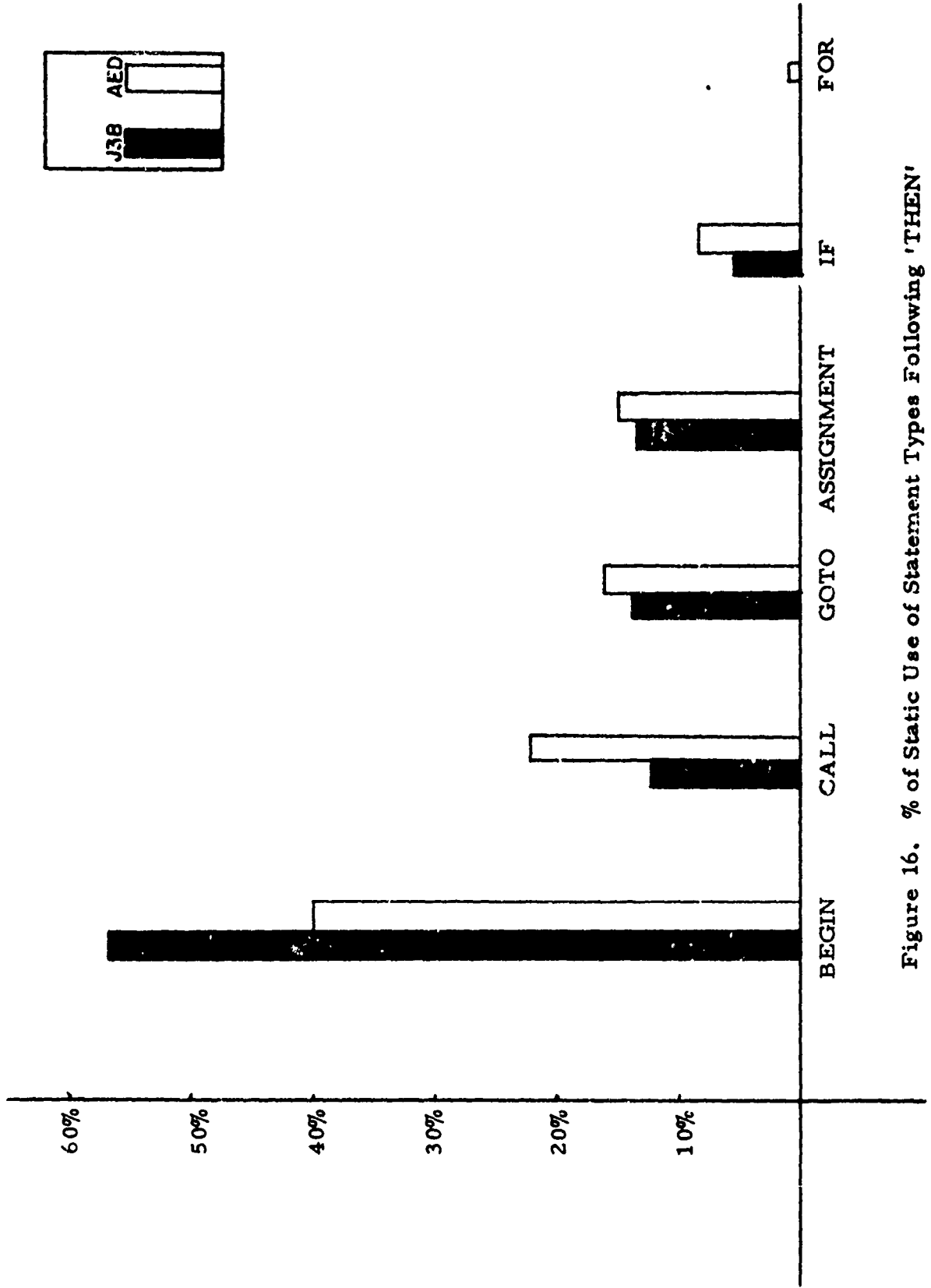
Figure 16. % of Static Use of Statement Types Following 'THEN'
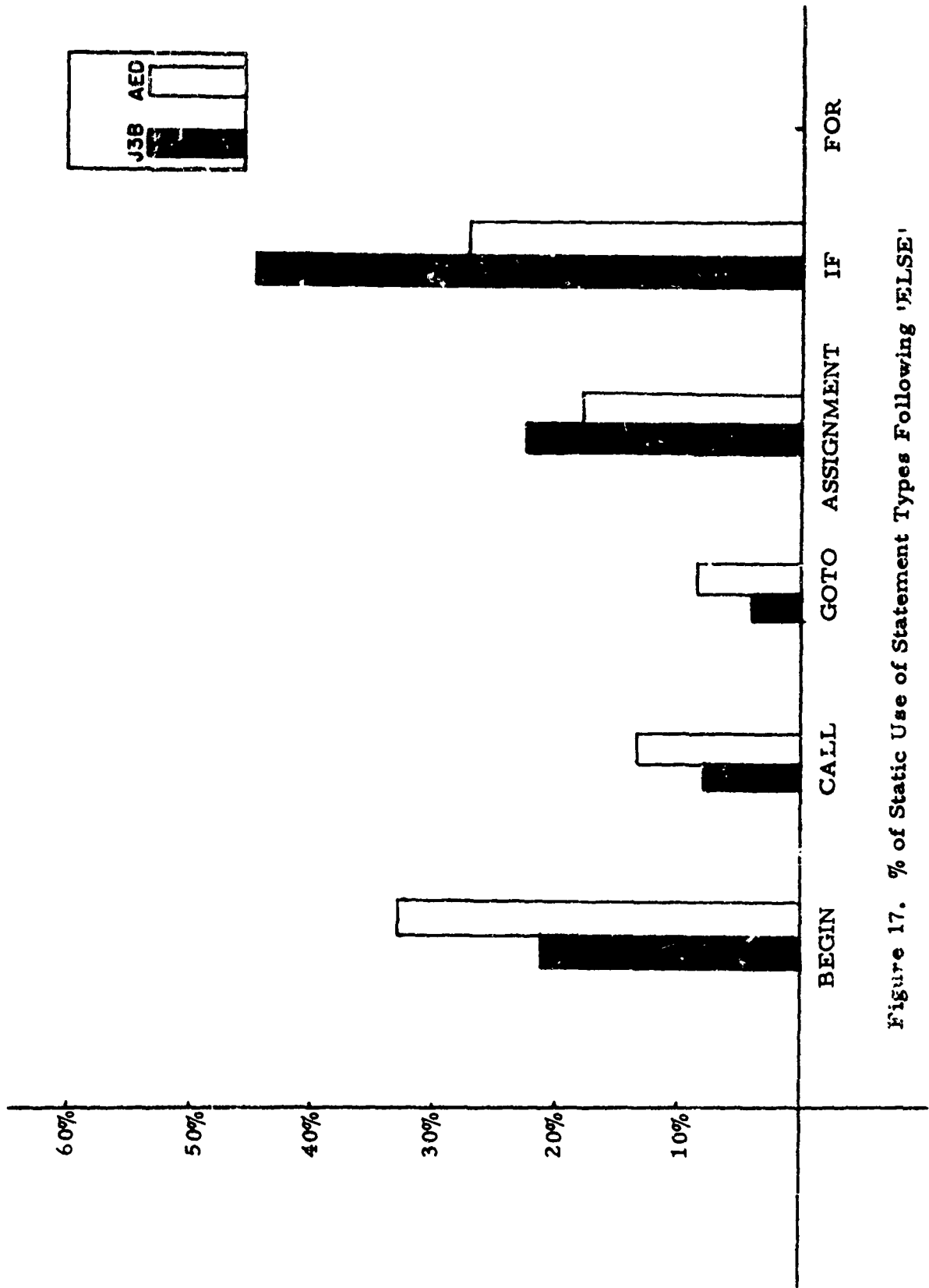
Figure 17. % of Static Use of Statement Types Following 'ELSE'

224
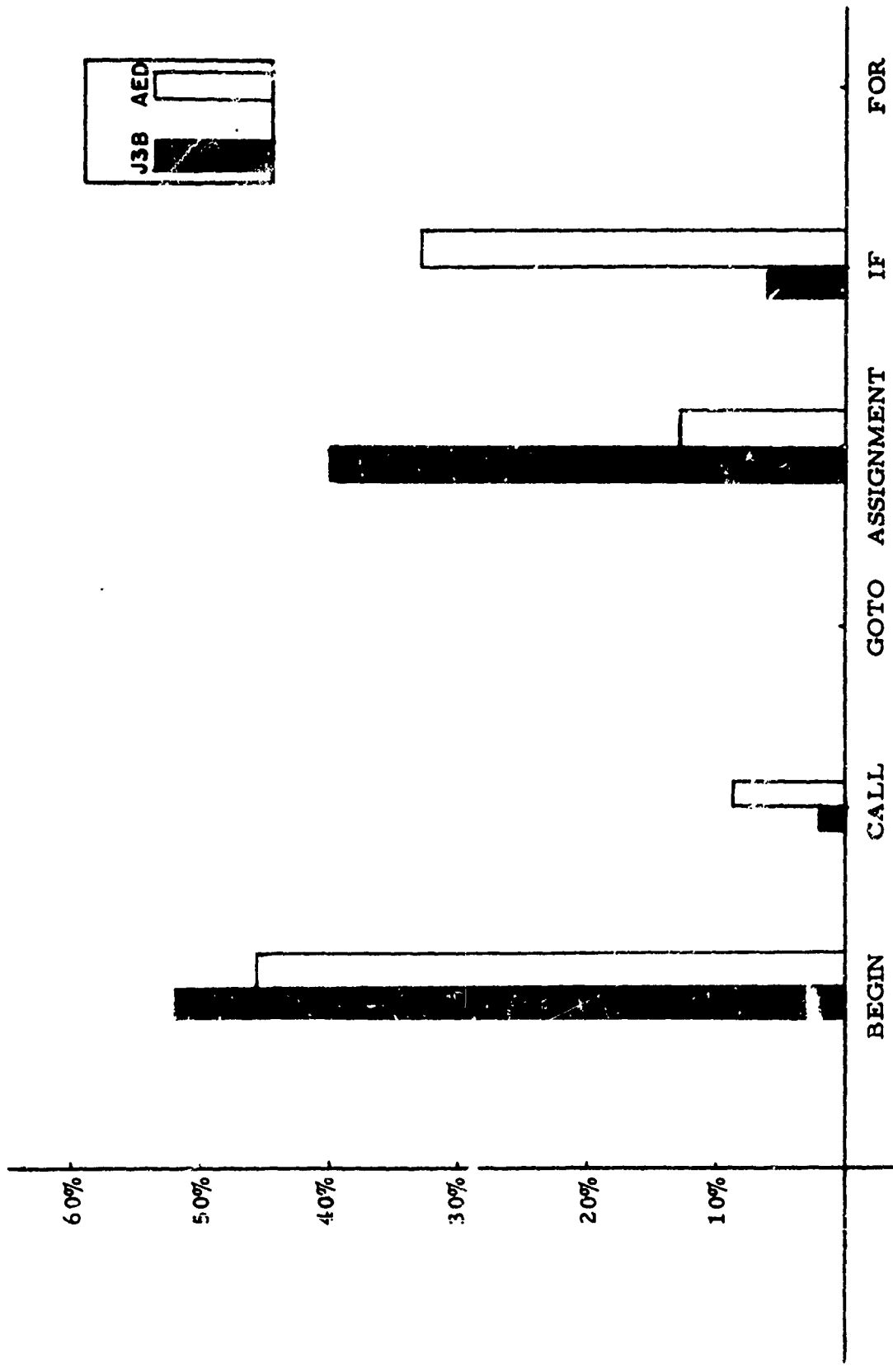
Figure 18. % of Static Use of Statement Types Following 'DO'

Statement types. This bar graph (Figure 15) shows the relative frequency of occurrence of the six statement types: assignment, procedure or function call, IF-THEN, GOTO, IF-THEN-ELSE, FOR. The IF-THEN constructions were counted separately from the IF-THEN-ELSE constructions. The overall impression is one of extreme matching between the AED and J3B profiles.

Statements following "THEN", "ELSE", or "DO". The bar charts (Figures 16 , 17 , and 18) show the relative frequency of six statement types (listed below) following one of the keywords, THEN, ELSE, or DO respectively. The keyword THEN occurs in both the IF-THEN and IF-THEN-ELSE constructions. DO occurs in FOR statements to introduce the loop to be iterated. The six statement types are:

- BEGIN (beginning a BEGIN-END block)
- CALL (a procedure call)
- GOTO
- Assignment
- IF
- FOR

After "THEN". As expected, a sequence of statements in a BEGIN-END block (BEGIN) usually follow "THEN". This statistic may be slightly exaggerated by the stylistic habit of some programmers who always use a BEGIN-END construction following a THEN, even though a single statement is used within the block (in which case the BEGIN-END is not needed). The use of this block form causes a different and more distinctive format of printout from the AED reformatting processor (PRALG), and is preferred by some. Also, if later corrections add statements following the THEN, the editing task is simplified if a BEGIN-END already exists in the source program.

The use of call, GOTO, and assignment statements following THEN appears to be used with about equal frequency, with AED showing more calls than J3B; IF statements are used somewhat less. FOR is almost never used in AED, and never in J3B.

226

After "ELSE". The statement types following ELSE (Figure 17) show a significant different in their usage pattern from the THEN case. Compound statements are much less dominant, and the IF form is much more popular. This is probably caused by the popular technique of testing for a condition in a THEN, and then continuing to refine the choice in the ELSE clause, resulting in IF-THEN-ELSE-IF sequences. Another usage difference shown in Figure 17 is that instead of being approximately equal in usage, assignments, calls, and GOTO's are used in that order of popularity.

Overall, the profiles shown in Figure 17 are seen to be very similar, but there are a few noticeable differences. J3B appears to have a higher preference for the IF-THEN-ELSE IF construction than AED uses, and AED has a preference for IF-THEN-ELSE BEGIN which is not used as frequently by J3B.

After "DO". In Figure 18 , it is seen that the statement types following "DO" show a very heavy use of assignments in J3B and a very heavy use of IF in AED. This lack of agreement in the AED and J3B profiles is probably because there are only a few FOR loops used in both compilers. Thus, a relatively small number of one form has a major effect on the percentage profile. If this explanation is correct, then the importance of the profile component shown in Figure 17 is relatively unimportant, and would contribute little to the overall performance factor calculated for an environment in which a compiler is being evaluated.

The complete absence of the use of FOR following DO shows that multiple nesting of loops is used only within a BEGIN-END (compound statement) block, and never as the simple loop-within-loop form.

Arithmetic operators. This bar chart (Figure 19) shows the relative use of the three categories of arithmetic operators:

- + or = (plus or minus),
- * (multiplied by), and
- / (divided by).

227
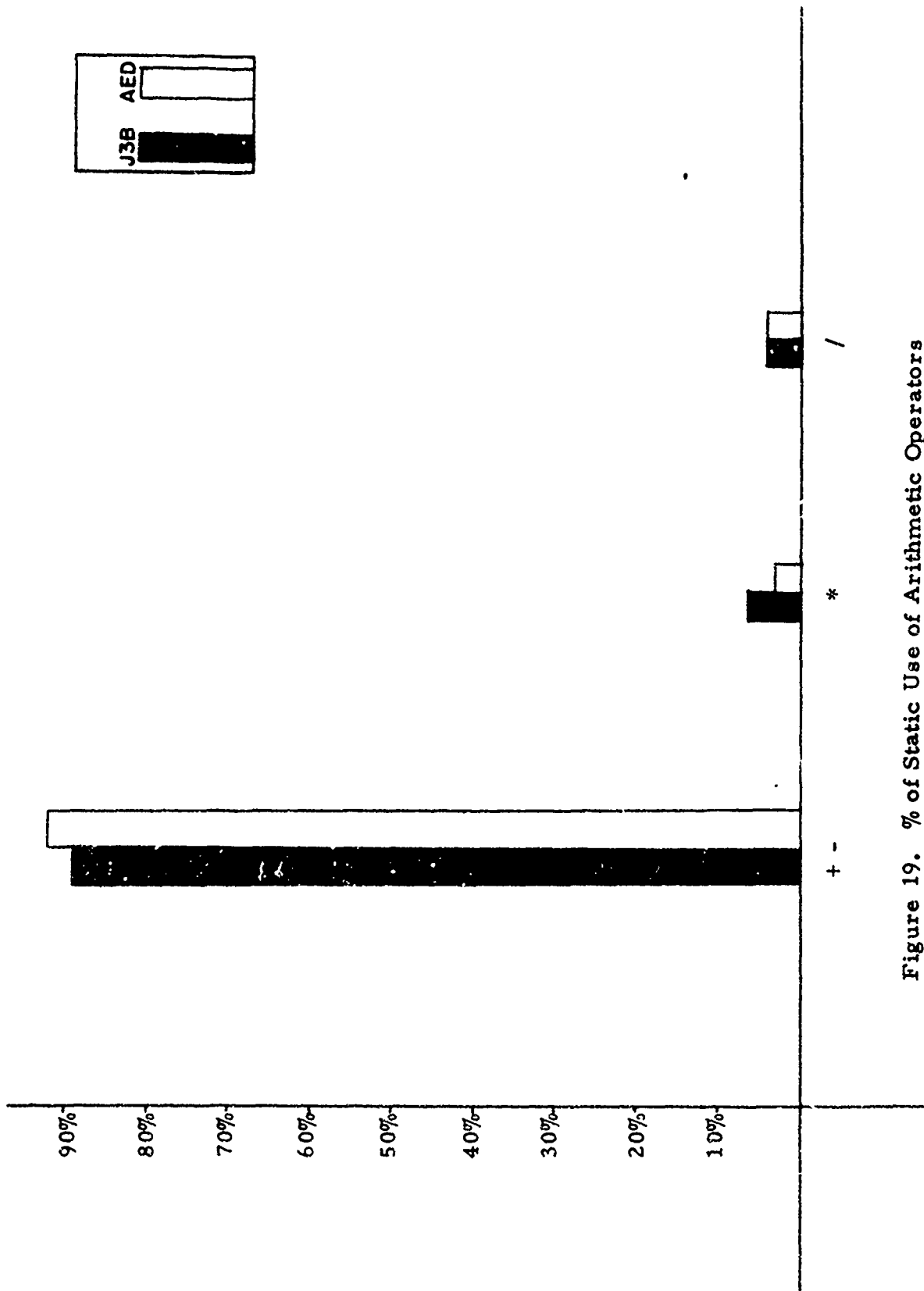
Figure 19. % of Static Use of Arithmetic Operators

The overall impression is one of extreme similarity between the AED and J3B profiles. Figure 19 also shows that * and / are very rare in comparison to + and -, and that * and / are used about equally.

Arithmetic forms. This bar chart (Figure 20) shows the relative use of the following six categories of arithmetic forms:

- A OP 1, 2 -- variable operator specific literal (1 or 2 depending on operator used -- see Section 2).

- A OP B -- variable operator variable.

- A OP L* -- variable operator literal other than 1 or 2 as specified above.

- E OP 1, 2 -- arithmetic expression operator specific literal (1 or 2 as above).

- E OP L* -- arithmetic expression operator literal other than 1 or 2 as specified above.

- OTHER -- all other arithmetic forms.

A spot check of the source language found that the OTHER category was almost exclusively the form

- Arithmetic expression operator arithmetic expression

such as (A+B) * (C+D), although a small number of other forms were also seen.

The overall impression from Figure 20 is that this aspect of the profiles again find significant similarity between the AED and J3B compilers. However, there are some noticeable stylistic differences. AED has a slight preference for the A OP 1, 2 form over the forms A OP B and A OP L which are used about equally. On the other hand, J3B uses the forms A OP B, A OP 1, 2, and A OP L in a definite decreasing pattern. The AED and J3B use of the OTHER form is similar, with J3B using it somewhat more. The use of the forms E OP 1, 2 and E OP L are comparatively rare in both AED and J3B, and are used about equally in both compilers.
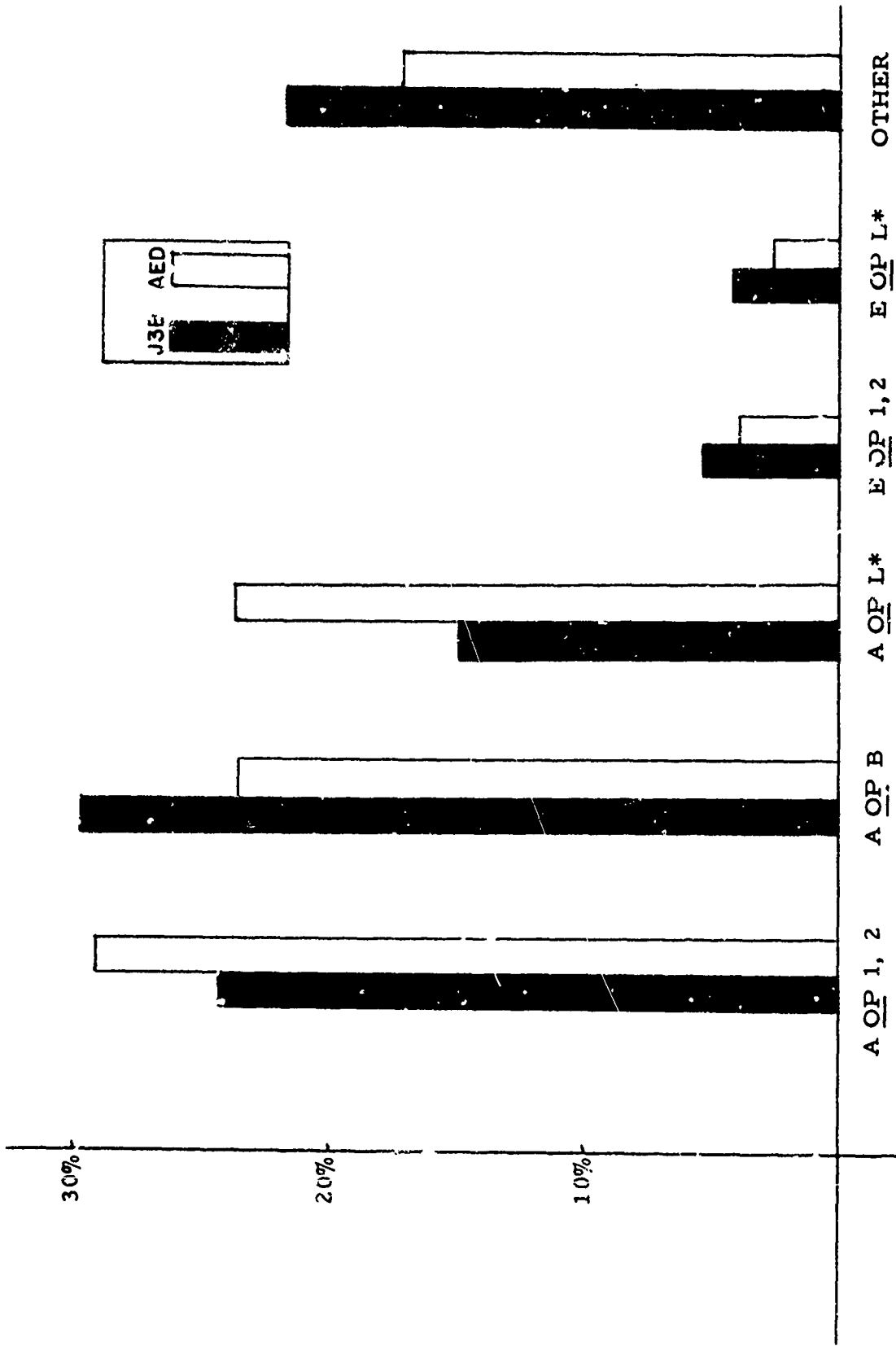
Figure 20. % of Static Use of Arithmetic Forms

Boolean operators. This bar chart (Figure 21) shows the
relative use of four categories of predicates (listed below) and the
Boolean connectives AND and OR. The predicate categories are:

- EQL (==)
- NEQ ($\neg$ =),
- GEQ+GRT (>= and >), and
- LEQ+LES (<= and <).

Figure 21 shows the EQL and NEQ (== and $\neg$ =) are the most
common Boolean operators, with EQL being far more popular than all
others. The remainder of the operators are used about equally.

The overall impression from Figure 21 is again that the AED
and J3B compilers have similar profiles.

Boolean forms. This bar chart (Figure 22) shows the relative
use of the following five categories of Boolean forms as used in IF
clauses:

- A PR L* -- variable predicate literal other than 0.
- A PR B -- variable predicate variable   te that a
  insignificant number of forms using AL   and OR and
  Boolean variables are included in this category).
- A PR 0 -- variable predicate 0 (literal).
- E PR L -- arithmetic expression predicate literal
- OTHER -- all other forms.

A spot check of the source language code showed that members of the
OTHER category are primarily of a form of which the following is a
representative example:

- A==B AND C==D

Here == often is replaced by $\neg$ =, and B and D are often literals.

In Figure 22 we note the high preference of J3B for the form
A PR L*, and the more moderate preference of AED for the form
A PR B not shared by J3B. However, the overall impression from
Figure 22 is rough similarity between the AED and J3B profiles. If

231
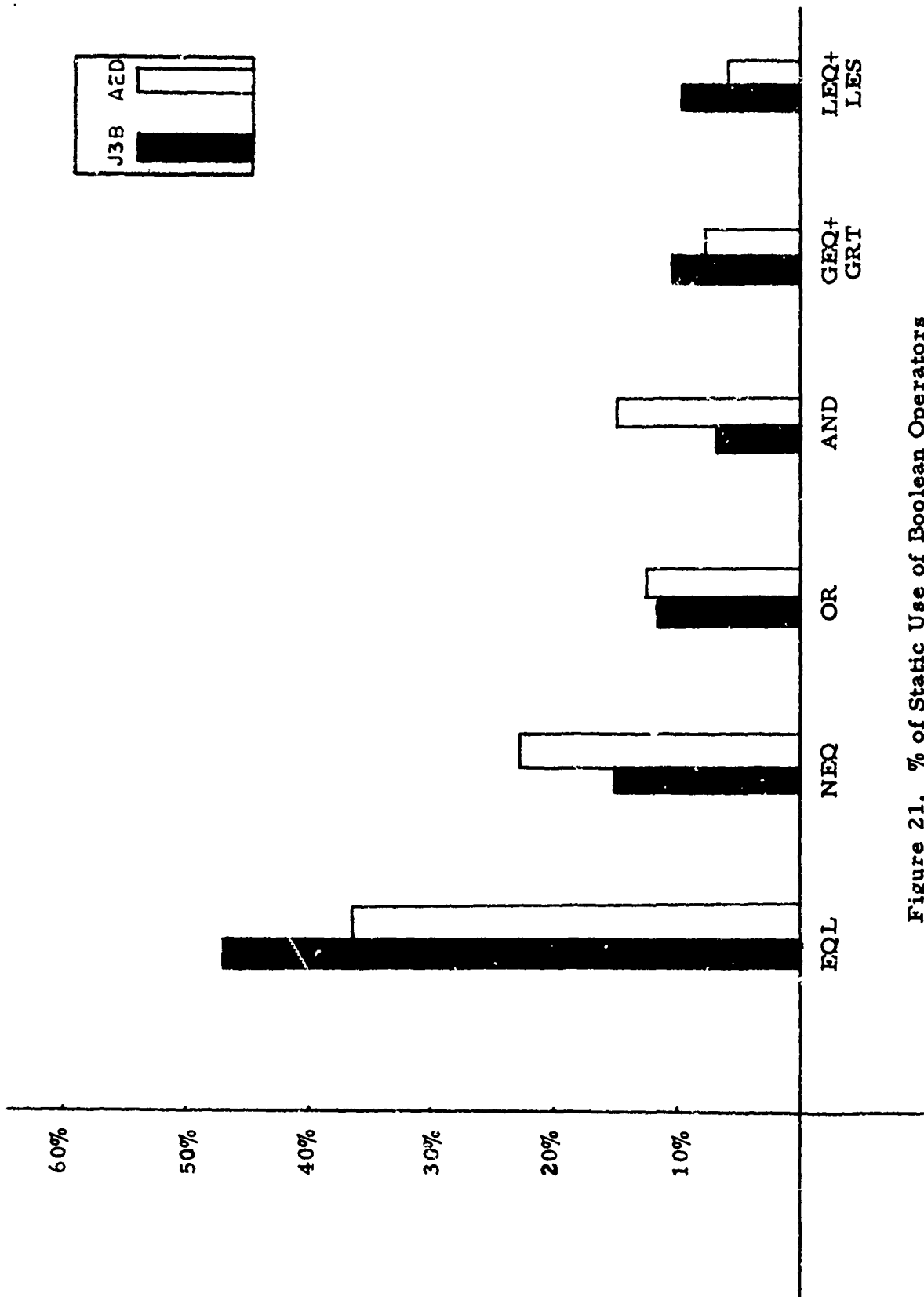
Figure 21. % of Static Use of Boolean Operators
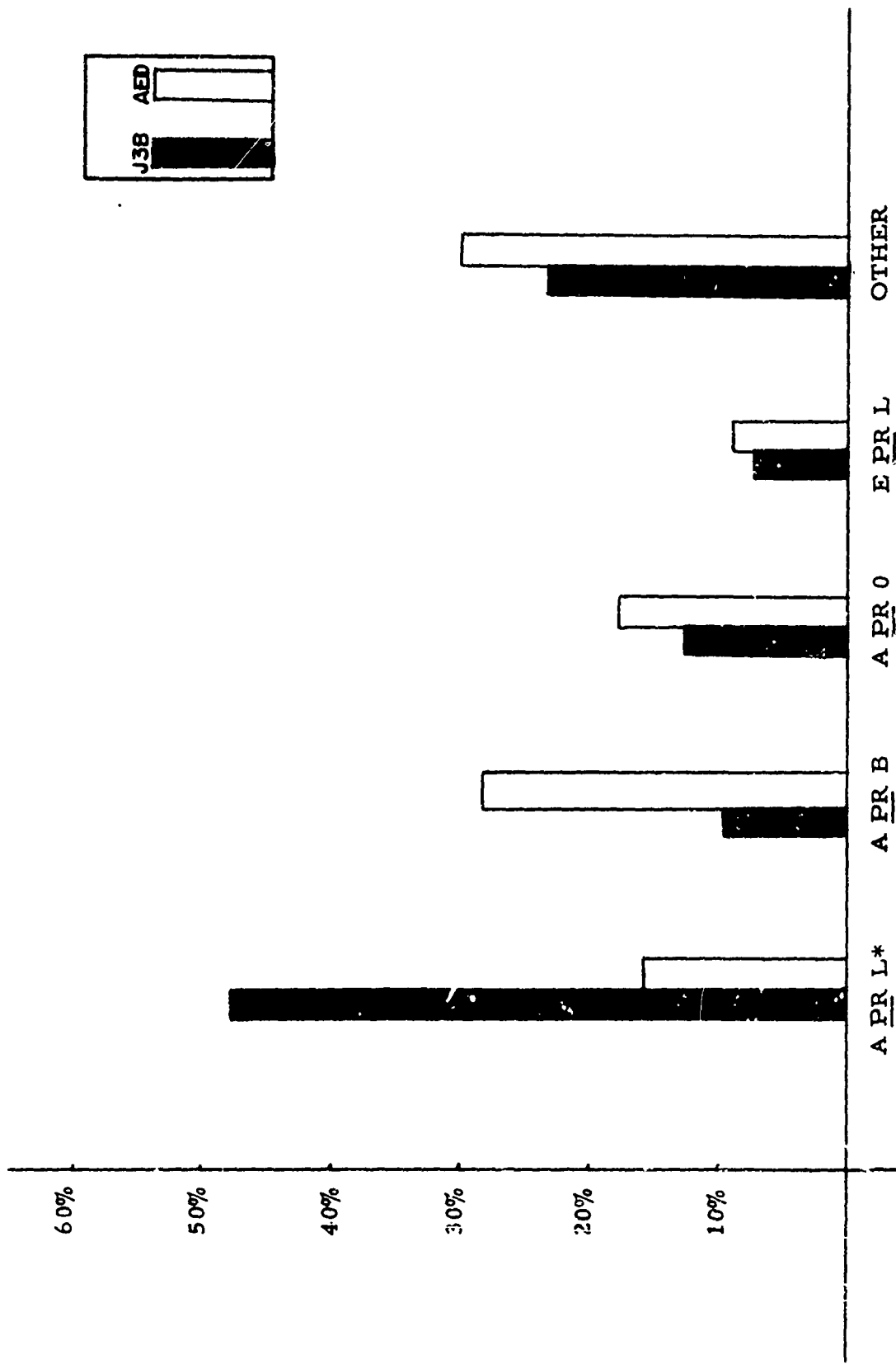
Figure 22. % of Static Use of Boolean Forms

the first three categories are combined, the resulting percentages would be approximately 70% for J3B and 62% for AED. The regrouped profiles using these values would then be very similar.

% of arithmetic operators with n R.H.S. operations. This histogram (Figure 23) shows that the great majority (roughly 80%) of R.H.S. forms have no operators. This result emphasizes the observation assignment statements are generally very simple in both compilers.

A spot check of the forms used in arithmetic assignment statements showed that they are typified by the following examples:

- A = B
- A = literal

Here, A or B are frequently bead components as well as declared integer variables. It should be noted that, through an oversight in the instrumented compiler used to gather the data, all Boolean assignment statements were erroneously added into the "0 - RHS" left-most column, (e.g. A = B AND C). Since all forms of Boolean assignment are very rare, it is qui's unlikely that this introduced any sizeable error in the data as shown. The spot check also found that the bulk of the remaining arithmetic assignments (roughly 20%) were of the forms typified by the following examples:

- A = B ± 1
- A = B ± literal other than 1
- A = B ± C

There is a clear overall impression from the histogram of Figure 23 that the AED and J3B profiles are extremely similar with respect to the use of operators in arithmetic assignment statements.

% of Boolean expressions with n operators. The Boolean expressions counted in generating the data comprising the histogram of Figure 24 were only those occurring in an IF clause. However, all other occurrences of Boolean expressions are extremely rare in the two compilers. The Boolean operators counted included six predicates ($==$, $\neg=$, $<$, $<=$, $>$, $>=$) and the two connectives AND and OR.
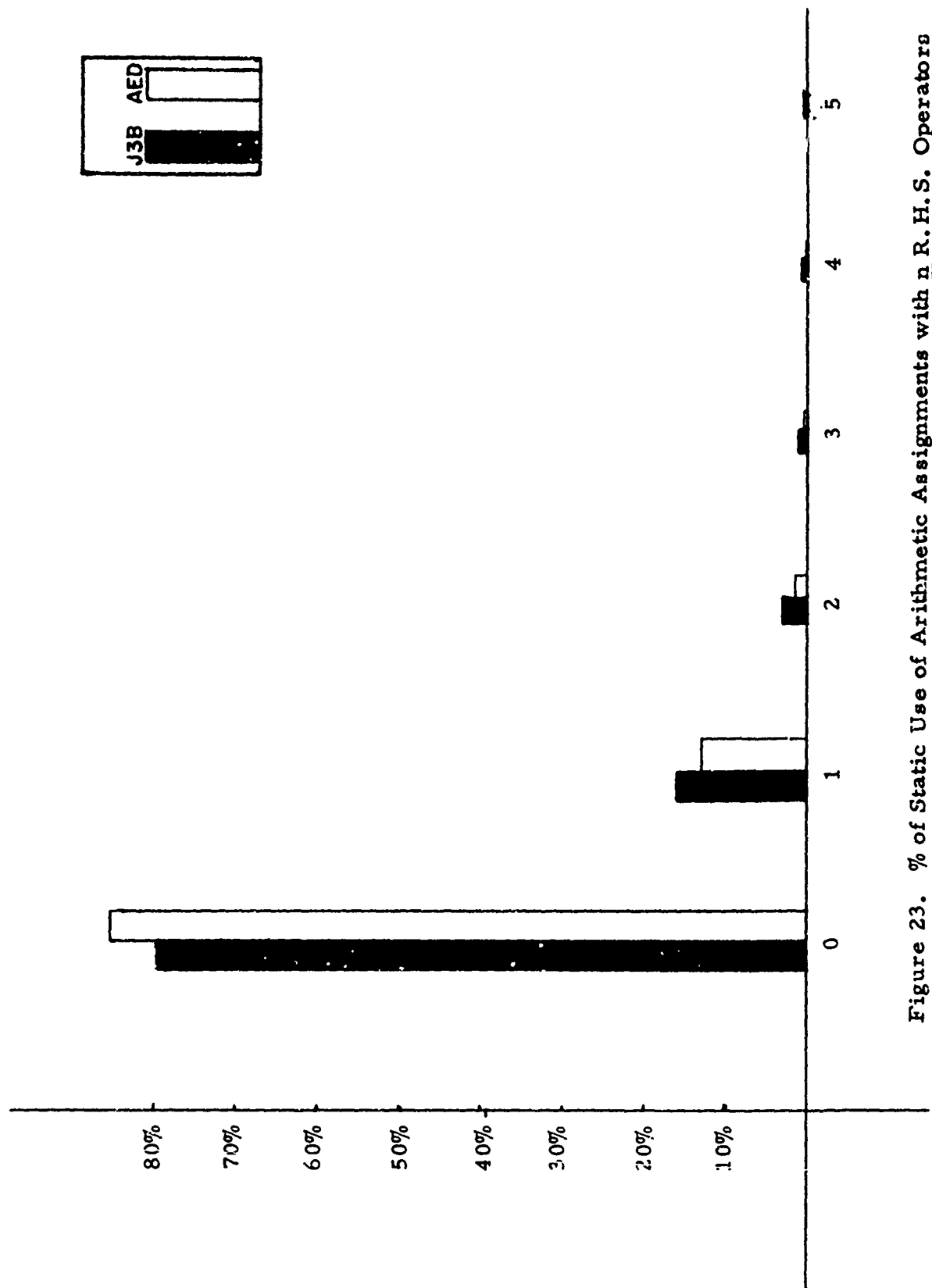
Figure 23. % of Static Use of Arithmetic Assignments with $\underline{n}$ R.H.S. Operators
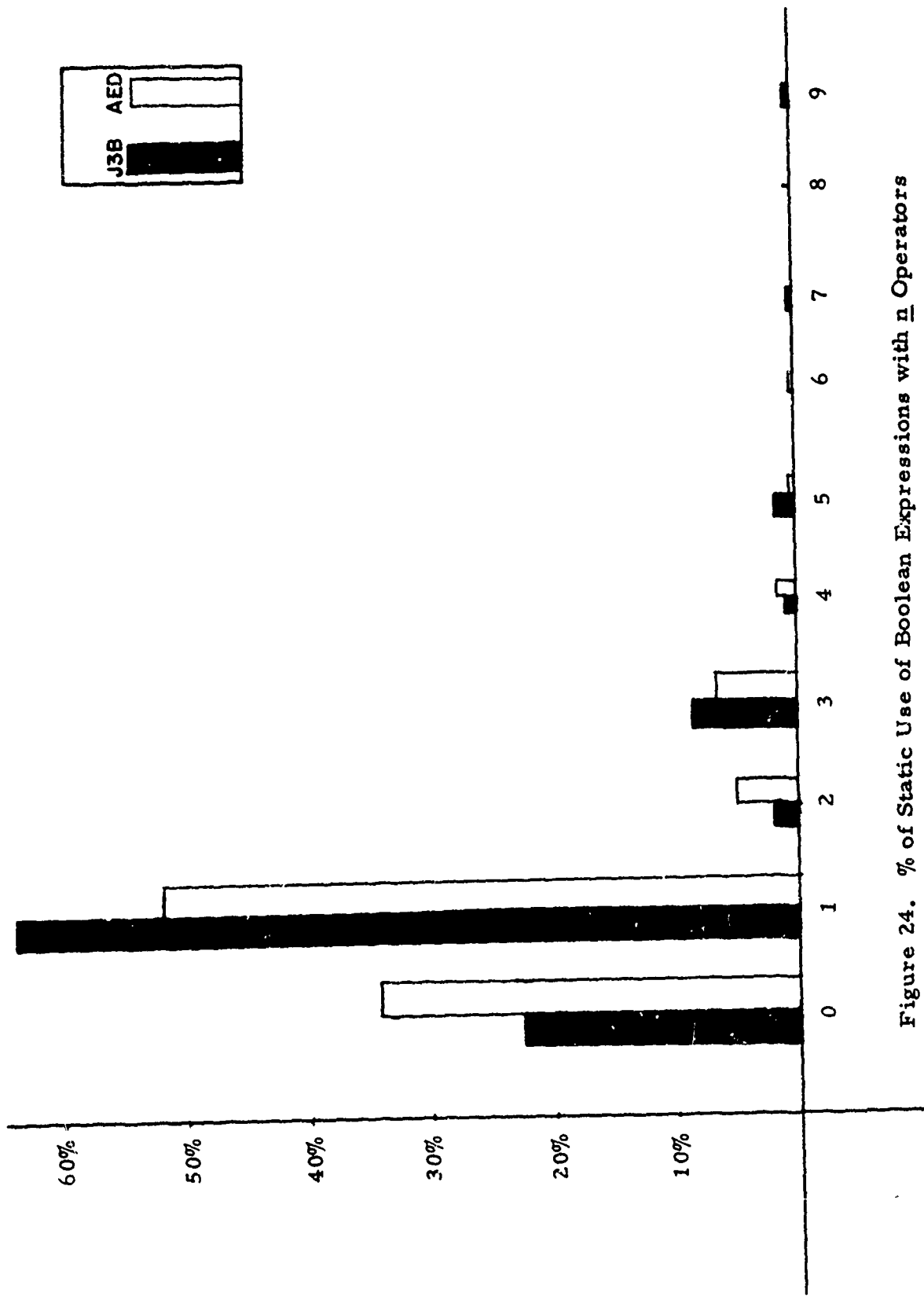
Figure 24. % of Static Use of Boolean Expressions with n Operators

Note that approximately 88% of all Boolean expressions in both compilers occur with 0 or 1 operators. A spot check of the source language programs found for the 0 operator case only the form IF A, where A was a Boolean variable. For the 1 operator case, the spot check found only the form IF A predicate B, where A and B were integer arithmetic variables, and predicate was usually == or ¬=. Boolean expressions involving 2 or more operators are seen to be very rare.

The overall impression from Figure 24 is again a significant degree of similarity between the AED and J3B profiles.

% of procedure and function calls with n arguments. The histogram presented in Figure 25 shows that approximately 90% of all procedure and function calls have three or fewer arguments for both the AED and J3B compilers. The overall impression in Figure 25 is again a high degree of similarity between the AED and J3B profiles.

% of executable statements nested n deep in FOR loops. The histogram of Figure 26 shows that almost all executable statements occur outside of FOR loops. This is easily understood since there are relatively few FOR statements used in the AED and J3B compilers, and it has been observed (Section 2) that on the average only three executable statements appear in each FOR loop. This result of course causes the AED and J3B profiles to be very similar with respect to this category of AED language usage.
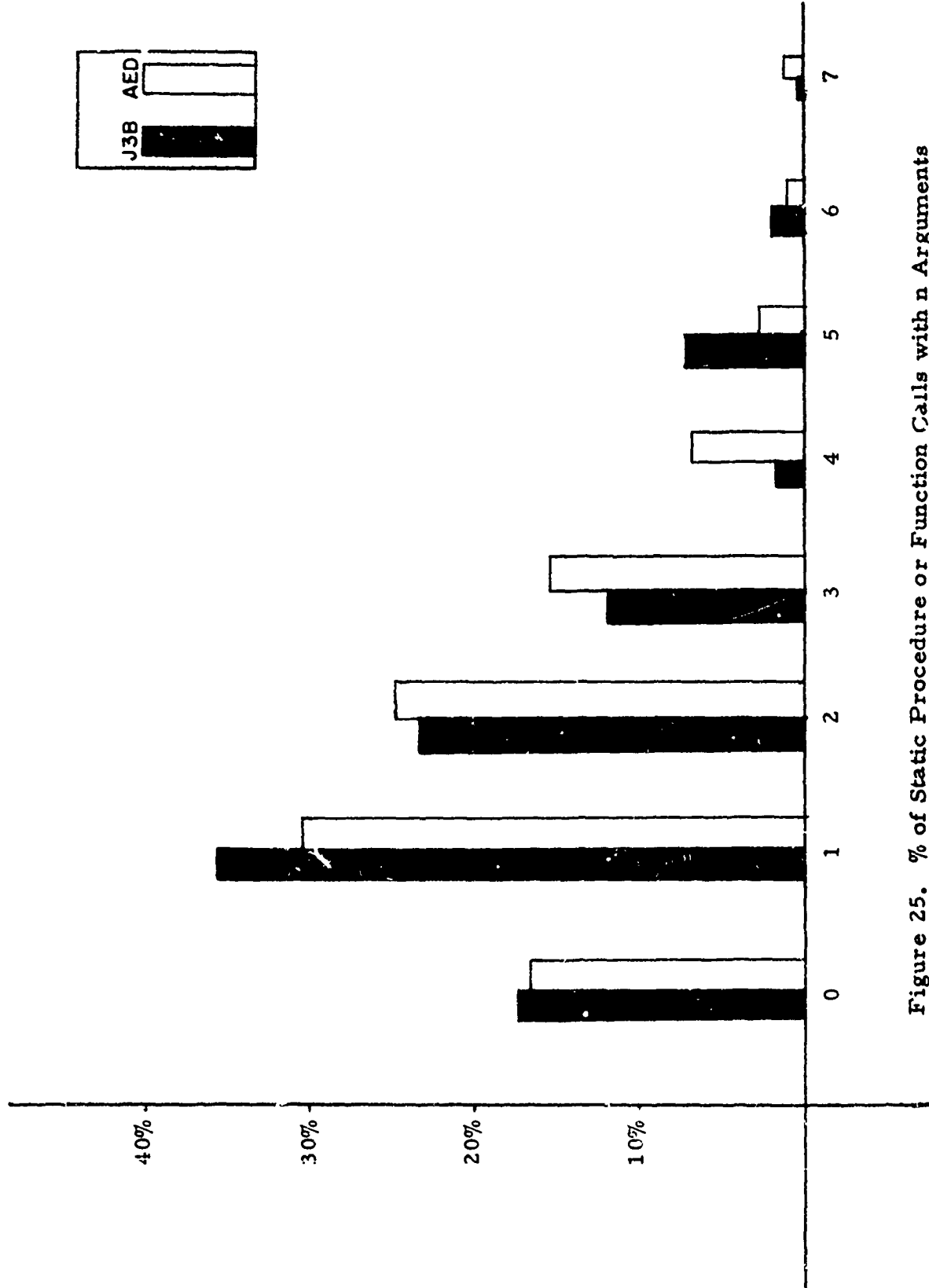
Figure 25. % of Static Procedure or Function Calls with n Arguments
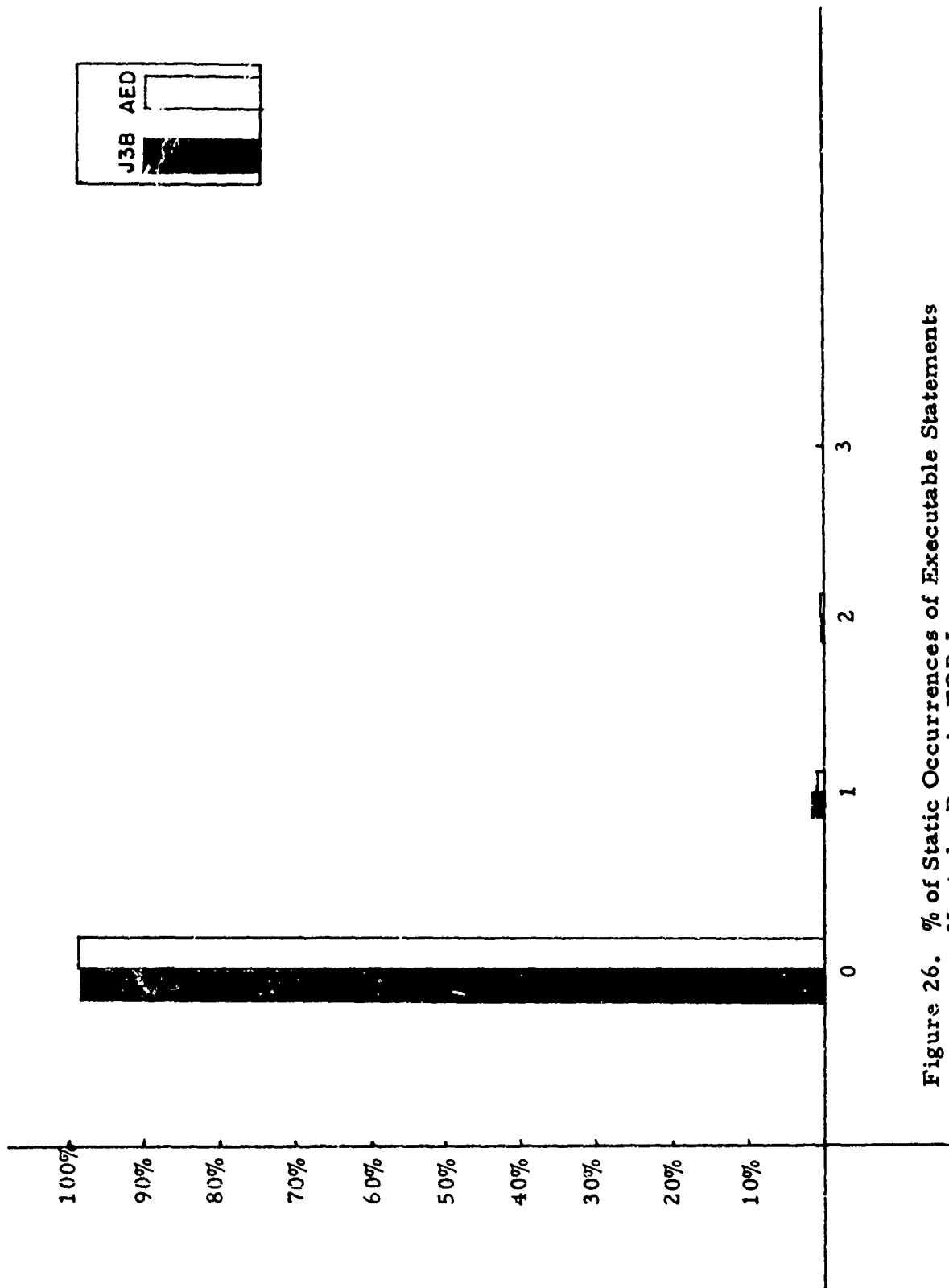
Figure 26. % of Static Occurrences of Executable Statements Nested n Deep in FOR Loops

CHAPTER 12
DYNAMIC COMPILER DEMAND PROFILE DATA

1. Overview

This chapter, together with the companion chapter entitled "Static Compiler Demand Profile Data" (Chapter 11), presents a summary of all non-timing data gathered during the project. Dynamic data is presented in this chapter in the same form as was the static data in Chapter 11. It is suggested that the reader should first become familiar with the static data discussion before preceding with a review of the information presented in this chapter.

The dynamic data was obtained using the results of the static data gathering effort as a basis. Static data counts for each source language procedure in each of the AED and J3B compilers were multiplied by a weighting factor to obtain comparable dynamic values. The weighted counts for each procedure were then summed, to obtain grand total weighted counts for each compiler.

The weighting factors applied to each procedure were derived by running a version of each compiler which was modified to keep a record of the number of times each program in the compiler was entered. Each compiler was run twice, using two specially devised test programs as compiler input. (These test programs are presented in Appendix 2.) The total number of calls for each compiler program was printed, and the sum of the number of calls for the two tests was calculated as the desired weighting factor. Thus, if procedure A were called forty times during test 1 and sixty times during test 2, and procedure B were called once during test 1 and never during test 2, then A would be assigned a weighting factor of 100, and B would be assigned a weighting factor of 1. Thus each static data count for procedure A would have 100 times the importance of procedure B in contributing to the grand totals of the dynamic compiler demand profiles.

It should be noted that this weighting factor technique has several features which influence the interpretation of the dynamic profile results:

- Frequently, not all statements of a program are executed once, each time it is executed. A loop may cause one set of statements to be executed several times for one execution of the procedure, or a GOTO RETURN, IF-THEN-ELSE (or other form of conditional statement) may cause a set of statements never to be executed at all. The weighting technique used does not take these factors into account.

- All error diagnostic, recovery, and print programs are eliminated from the statistics, since both test cases were designed to compile correctly with no errors, and t is caused the weighting factor to be 0 for all error-handling programs in each compiler.

- The weighting factor is a direct consequence of the style and contents of the two test cases. It is believed that a reasonable mix of statements and programming styles were used in writing the two tests, but it should be noted that any language feature which was not employed in either test and which is the only cause for calling a particular compiler subroutine, results in that particular subroutine to have a weight of 0.

In the following pages, the dynamic data is grouped into tables and bar charts in exactly the same form as the static data appears in Chapter 11. The reader should compare the corresponding static data table or bar chart with its dynamic counterpart to see how the results are effected by applying the dynamic weights. In general, the dynamic data shows the same general results as was seen in the static data, except as noted in the text accompanying each table or bar chart.

2.   Tables of Dynamic Usage of AED Language Forms in the AED and J3B Compilers

In this section seven tables are presented which are the dynamic data counterparts of the tables of static data presented in Section 2 of Chapter 11.

Statement types, operators, and labels. Table 11 is the dynamic counterpart of Table 4.

The relative dynamic usage of various statement and operation forms is very similar to the static figures. In both compilers, call statements dropped slightly in importance, while IF and GOTO statements

241

Table 11. Summary of Dynamic Usage of Statement Types, Operators, and Labels

	J3B	AED
ASSIGNMENT STATEMENTS	871345	891793
IF STATEMENTS	512979	406546
FOR STATEMENTS	21031	19522
CALL STATEMENTS AND FUNCTION CALLS	627429	402466
GOTO STATEMENTS	367676	367874
ARITHMETIC OPERATIONS	240114	171459
BOOLEAN OPERATIONS	629763	513438
LABELS	307804	302432
TOTAL	3578141	3015530

increased. Usage of Boolean operations remained far ahead of arithmetic operations. The increase in the importance of IF and GOTO in the dynamic evaluation tends to indicate that these forms are used more frequently in the repetitive portion of the compilation process, whereas they are less frequently used in the initialization and error reporting areas.

Assignment statement forms. Table 12 is the dynamic counterpart of Table 5. Here again, very small changes in relative importance are noted between the static and the dynamic data. In both compilers, the forms "A = B" and "A = EXPRESSION" assumed slightly more importance in the dynamic data, while "A = FUNCTION (...)" assumed less importance. The later effect verifies the slight drop in function call importance noted in Table 9 on dynamic usage of statement forms.

Statement types used with conditionals and loops. Table 13 is the dynamic counterpart of Table 6. Most of the variation from the static data introduced by the dynamic use of weighting factors is seen to be generally unimportant. Increases or decreases shown in one compiler are offset by the opposite trend in the other compiler. The exceptions are:

- The BEGIN category (compound statement form) decreased in both THEN and ELSE categories, but increased in the DO category. This was a trend in both compilers.

- GOTO increased in both the AED and J3B compilers for the THEN clause, and decreased slightly in the ELSE category.

- In the DO case, IF decreased and assignment statement usage increased in both compilers.

There is no apparent rationale for these slight changes, except for statistical anomalies that are possibly introduced by the factors listed in Section 1.

FOR statement sub-types. Table 14 is the dynamic counterpart of Table 7. The UNTIL and WHILE forms of loop termination varied dramatically between the static and dynamic figures. In the J3B compiler the relative importance of the two forms was reversed, with WHILE

Table 12.   Summary of Dynamic Usage of Assignment Statement Forms

	J3B	AED
A = 0	53128	43026
A = 1	31588	17505
A = LITERAL (NOT 0 OR 1)	130545	45337
A = B	370910	443229
A = EXPRESSION	205349	237624
A = FUNCTION (...)	79825	105072
TOTAL	871345	891793

244

Table 13. Summary of Dynamic Usage of Statement Types Used
With Conditionals and Loops

FORM OF STATEMENTS FOLLOWING	THEN		ELSE		DO	
	J3B	AED	J3E	AED	J3B	AFD
BEGIN	198201	124764	17569	35069	13481	9847
CALL	83600	67163	18621	7994	325	46
FOR	0	32	0	0	0	0
IF	8215	24864	56974	24102	3101	5344
GOTO	82987	63964	2799	9457	0	0
= (ASSIGNMENT)	59735	65103	16491	43572	4093	4285
TOTALS	432738	345890	112454	120194	21000	19522

Table 14. Summary of Dynamic Usage of FOR Statement Sub-Types

	J3B	AED
UNTIL CLAUSES	3305	12552
WHILE CLAUSES	18024	6820
MULTIPLE ITERATION LISTS	2	150
OTHERS	0	0
TOTAL	21031	19552

dominating the dynamic data and UNTIL dominating the static data. No such reversal was evident in AED. It can only be concluded that an insufficient number of FOR loops within too few source language modules were reflected in the weighting factors and that the results are therefore distorted by the inadequacy of the statistical sample.

Integer forms in FOR statements. Table 15 is the dynamic counterpart of Table 8.

The set of data shown in Table 15 also suffers from too small a sample to perform an adequate evaluation, as was also the care for the static data in Table 8. From the numbers shown, the dynamic results show substantially the same patterns as the static data of Table 15. That is, the initial index variable is usually set to a literal or variable; the step increment is almost always 1, and the terminating value is usually a literal (not 1) or a variable.

Arithmetic forms. Table 16 is the dynamic counterpart of Table 9.

Table 16 shows the same overwhelming use of + or -, and the rare use of "*" and "/" that was evident in Table 9. The very small total number of "*" and "/" operators in the two compilers makes this classification too small a sample for determining good static/ dynamic trend comparisons. In examining the individual forms of arithmetic phrases, no clear trend is obvious. Most increases in dynamic data for J3B are offset by decreases in the same category for AED and vice-versa. The category "A" OP B" remains the most commonly used form for the "+" and "-" operators; "EXPRESSION OP LITERAL" and "A OP LITERAL" are the most commonly used arithmetic forms using the "*" or "/" operators.

Table 15. Summary of Dynamic Usage of Integer Forms in FOR Statements

FORMS NEXT TO "STEP", "UNTIL"

	"STEP" LEFT OF		"STEP" RIGHT OF		"UNTIL" AFTER	
	J3B	AED	J3B	AED	J3B	AED
1	11	3514	12029	10788	20	0
LITERAL (NOT 1)	7042	5202	0	1766	1744	1908
A	4952	3838	0	0	795	10609
EXPRESSION	102	0	78	0	446	35
TOTAL	12197	12554	12107	12554	3005	12552

248

Table 16. Summary of Dynamic Usage of Arithmetic Forms

	+ OR -		*		/	
	J3B	AED	J3B	AED	J3B	AED
A OP (1) (2)	50334	50791	0	0	4401	0
(1) (2) OP A	48	2	46	1420	0	0
A OP LITERAL	41751	20970	4459	1606	338	1053
LITERAL OP A	10083	825	450	0	0	495
A OP B	64980	74041	0	0	450	0
EXPRESSION OP (1) (2)	20515	5381	0	0	0	0
(1) (2) OP EXPRESSION	0	0	0	0	0	0
EXPRESSION OP LITERAL	4865	2	477	1747	9461	1424
LITERAL OP EXPRESSION	65	1606	33	1420	0	0
OTHER	36621	38432	2108	3383	0	3383
TOTAL	229762	192050	7573	9576	14650	6355

Boolean forms. Table 17 is the dynamic counterpart of Table 10.

The dynamic data shows the EQL (==) operator still by far the most commonly used, with the other operators showing no significant trend in the dynamic versus static comparison. In comparing the dynamic use of various forms of Boolean expressions, the forms "A OP LITERAL" and "A OP B" are seen to be the most commonly used, as was also the case in the static data.

3.   Bar Charts of Histograms of Dynamic Usage Patterns in the AED and J3B Compilers

In this section five histograms are presented which are the dynamic counterparts of the histograms presented in Section 3 of Chapter 11.

Procedure and function calls using n arguments. The histogram presented in Figure 27 is the dynamic counterpart of the histogram presented in Figure 10.

The weighted statistics shown in the histogram of Figure 28 show that the majority of procedures are called with three of fewer arguments, as was also seen in the static histogram (Figure 10). The most noticeable change in the static versus dynamic histograms is a sizeable increase in the calls using 1 and 2 arguments in J3B, with an opposite, leveling out effect in AED. This indicates that procedures with fewer arguments are called in the heaviest used J3B compiler programs, while the opposite is true for AED. There is no apparent explanation for this result except for the statistical anomalies that are possibly introduced by the factors listed in Section 1.

Assignment statements with n right hand side operators. The histogram presented in Figure 28 is the dynamic counterpart of the histogram presented in Figure 11.

The number of assignment statements having no right hand side (R.H.S.) operators remains dominant using the weighted data. The static and dynamic histograms show very little effect in any area caused by the use of dynamic weighting factors.

Table 17. Summary of Dynamic Usage of Boolean Forms

	==		> >=		< <=		¬=		AND		OR	
	J3B	AED	J3B	AED	J3B	AED	J3B	AED	J3B	AED	J3B	AED
A OP 0	36131	48327	5151	192	21213	4953	27128	27918	0	0	0	0
0 OP A	0	0	0	0	0	0	0	0	0	0	0	0
A OP LITERAL	247800	49192	33423	17585	32439	5180	25457	14548	0	0	0	0
LITERAL OP A	0	0	0	0	0	0	0	0	0	0	0	0
A OP B	12458	79145	20896	3262	25296	7084	7435	44510	0	2575	5528	4880
EXPRESSION OP 0	13337	12540	3327	4415	10873	9843	6784	14061	0	0	0	0
0 OP EXPRESSION	0	0	0	0	0	0	0	0	0	0	0	0
EXPRESSION OP LITERAL	18933	5049	4630	1283	0	1768	615	0	0	0	0	0
LITERAL OP EXPRESSION	0	0	0	0	0	0	0	0	0	0	0	0
OTHER	9057	25928	11862	3106	1936	629	6802	3516	81206	70239	91229	18210
TOTAL	337716	220181	79289	29843	91757	29457	74221	104553	81206	72814	96757	23090

251
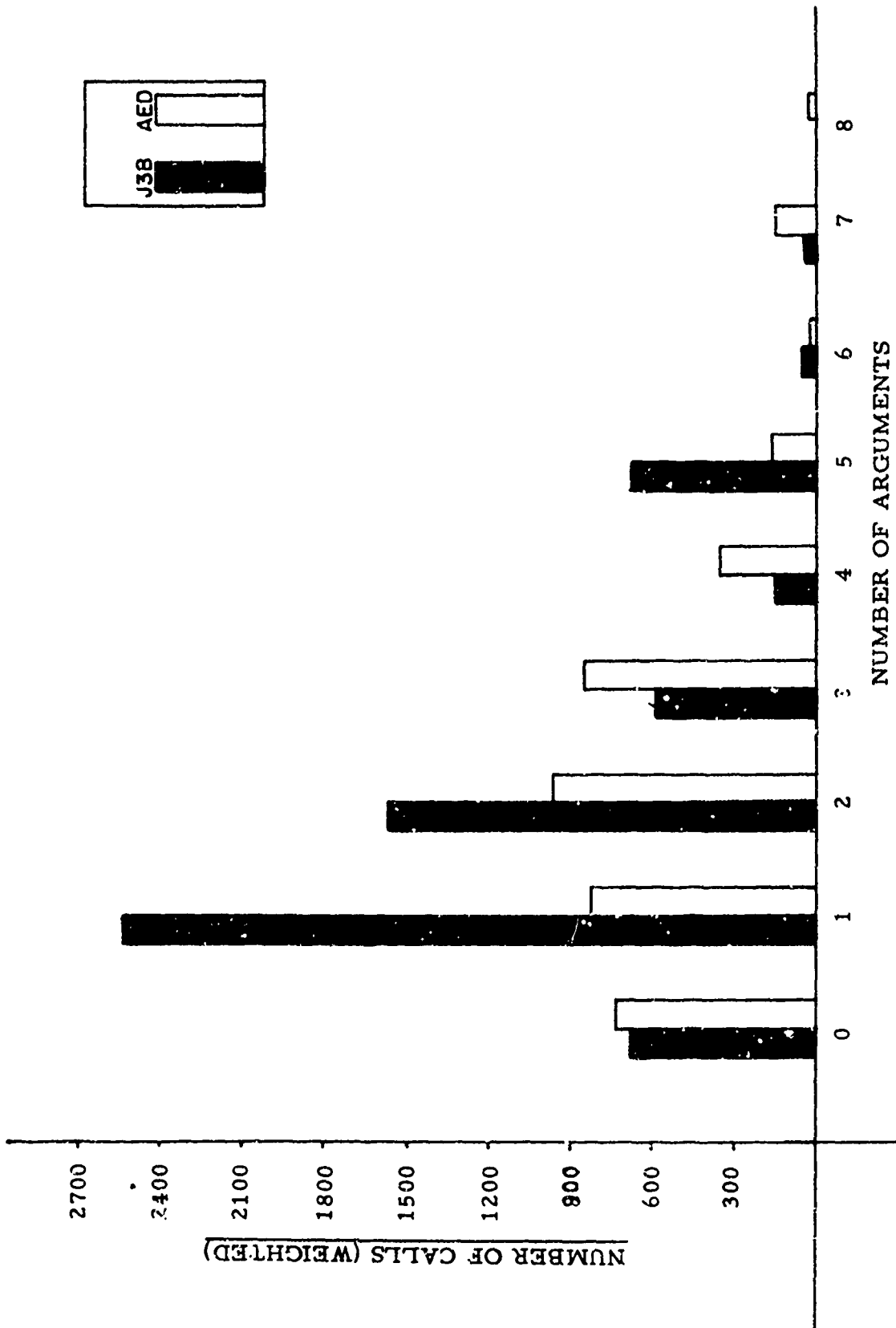
Figure 27. Histogram of Dynamic Occurrences of Procedure and Function Calls Using $\underline{n}$ Arguments
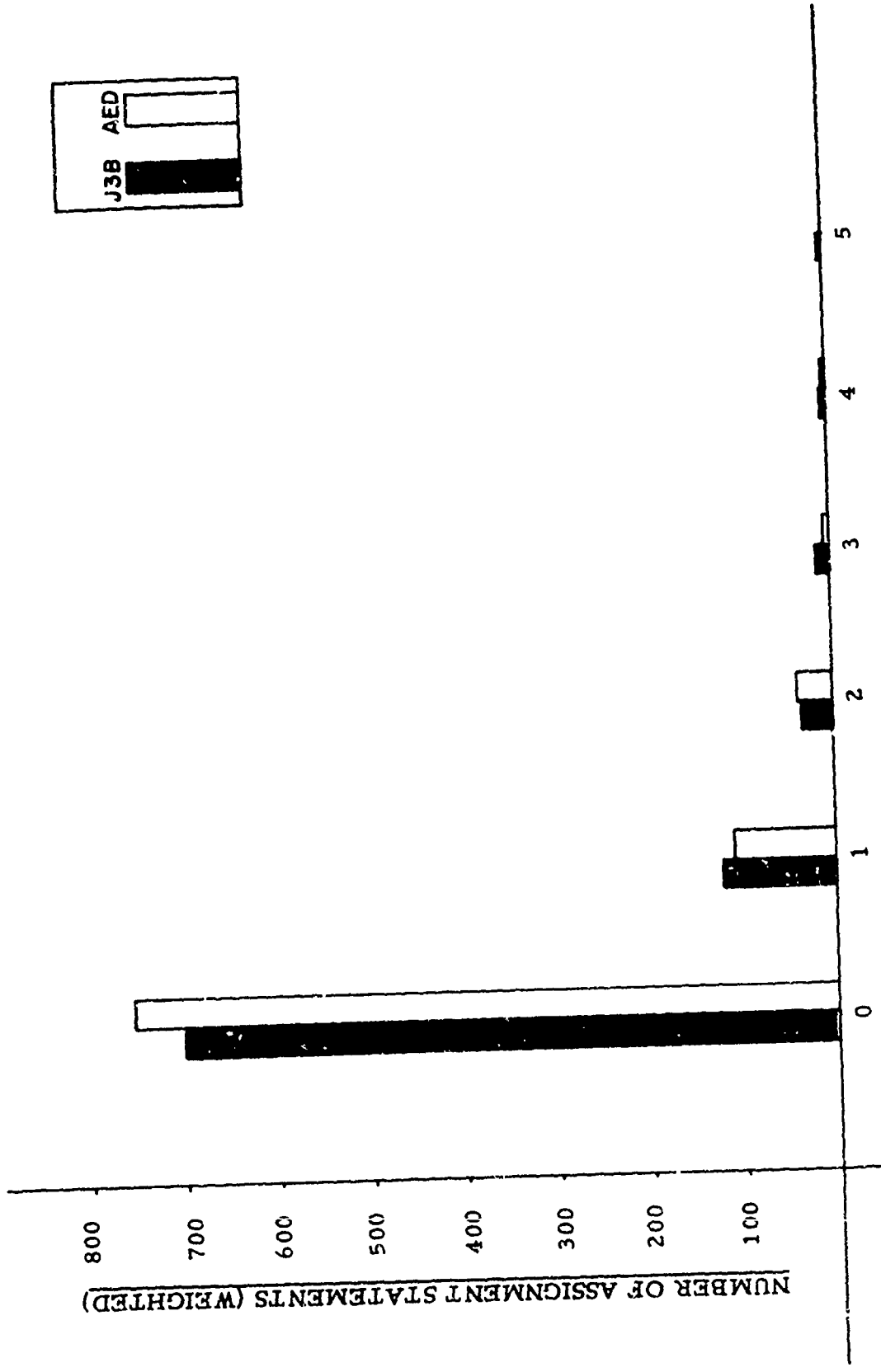
Figure 28. Histogram of Dynamic Occurrences of Assignment Statements With n Right Hand Side Operators

253

Boolean expressions with n operators. The histogram presented in Figure 29 is the dynamic counterpart of the histogram presented in Figure 12.

The application of the dynamic weights had very little effect on the data shown in this histogram, except that the 0 argument case was reduced in importance for the AED compiler. The 0 and 1 argument forms are seen to be dominant in the dynamic histogram, as was also the case in the static histogram.
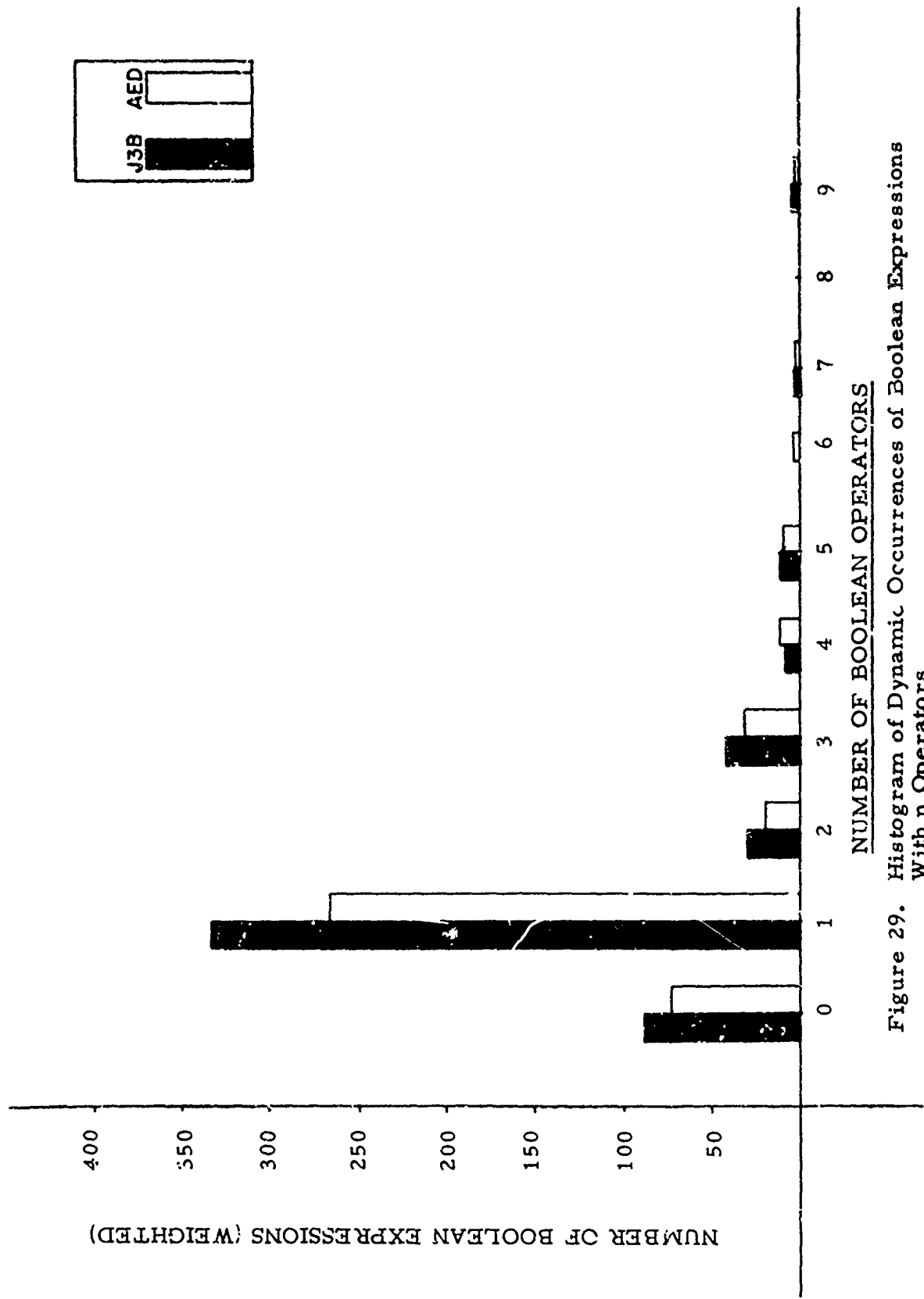
Figure 29. Histogram of Dynamic Occurrences of Boolean Expressions
With $\underline{n}$ Operators

255

FOR loops and executable statements nested <u>n</u> deep. The histograms presented in Figures 30 and 31 are the dynamic counterparts of the histograms presented in Figures 13 and 14 respectively.

The dynamic weights caused the AED statistic for use of FOR loops at a nesting depth of 2 to be greatly increased, while the J3B statistics were relatively unchanged due by the weighting effect. The loop at a nesting depth of 3 seen in the static data had its weighted count reduced to 0, since the program containing this loop was never entered in either of the two J3B test cases processed.

The reason for the odd change in the AED statistic is the fact that FOR loops are used rarely and that the few procedures in AED using FOR loops were called with a relatively high frequency, thereby resulting in a high weighted count.

In comparing Figures 31 and 14, we see that when the number of executable statements is taken into account, the unusual height of the AED 2 level bar in Figure 14 no longer shows up in the histogram of Figure 31. On the other hand, a new separation of the AED and J3B profiles is seen in the level 1 bars, where the J3B bar increases substantially as compared with the AED bar. The reasons for this odd behavior is probably again due to the inadequacy of the statistical sample for FOR loops.

NESTING DEPTH
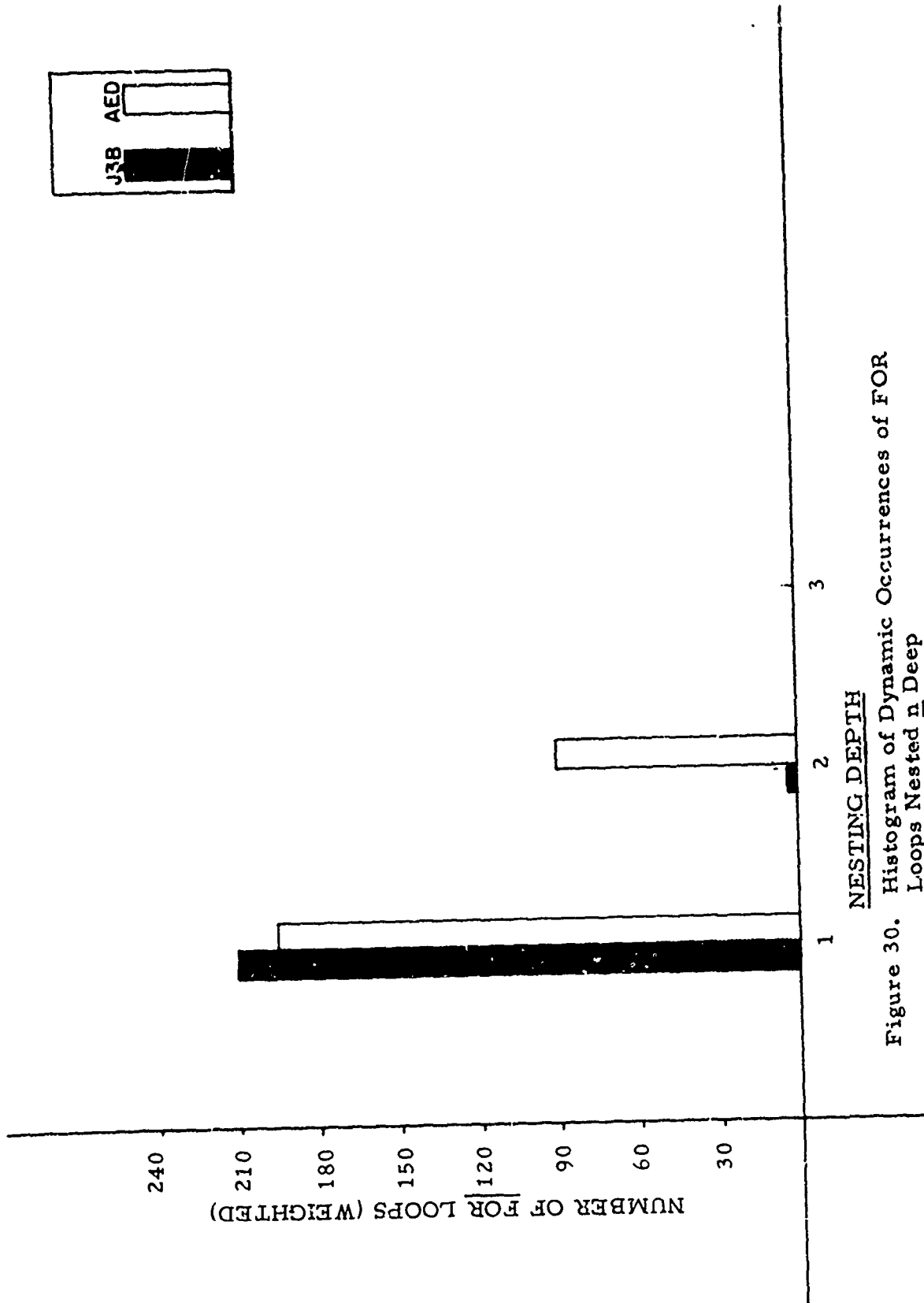
Figure 30. Histogram of Dynamic Occurrences of FOR
Loops Nested n Deep

257
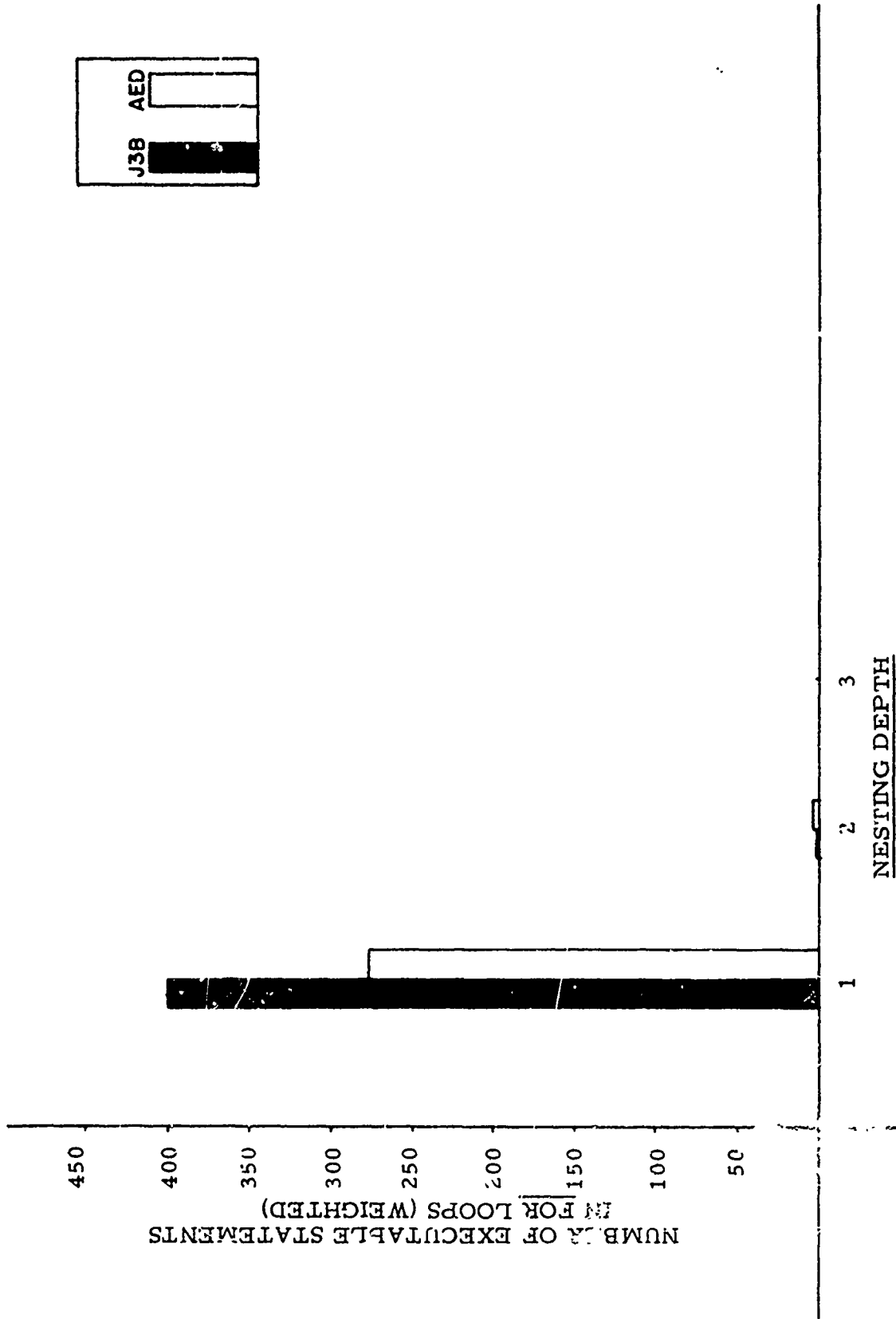
Figure 31. Histogram of Dynamic Occurrences of Executable
Statements Nested n Deep in FOR Loops

258

4.　　**Bar Graphs of Frequency Histograms of Relative Dynamic Usage**
　　　　of AED Language Forms

In this section twelve bar graphs are presented which are the dynamic counterparts of the bar charts of static data presented in Section 4 of Chapter 11.

Statement types. The bar graph presented in Figure 32 is the dynamic counterpart of the bar graph presented in Figure 15.

In Figure 32, the dynamic weighted data show a slight leveling effect in comparison with Figure 15, its static equivalent. Assignment and call statements, which comprised about two-thirds of all statements used statically, dropped to about sixty percent usage when the dynamic weights are included. Correspondingly, the IF, GOTO, and ELSE category which comprised about one-third of all statements used statically rose to about forty percent usage.
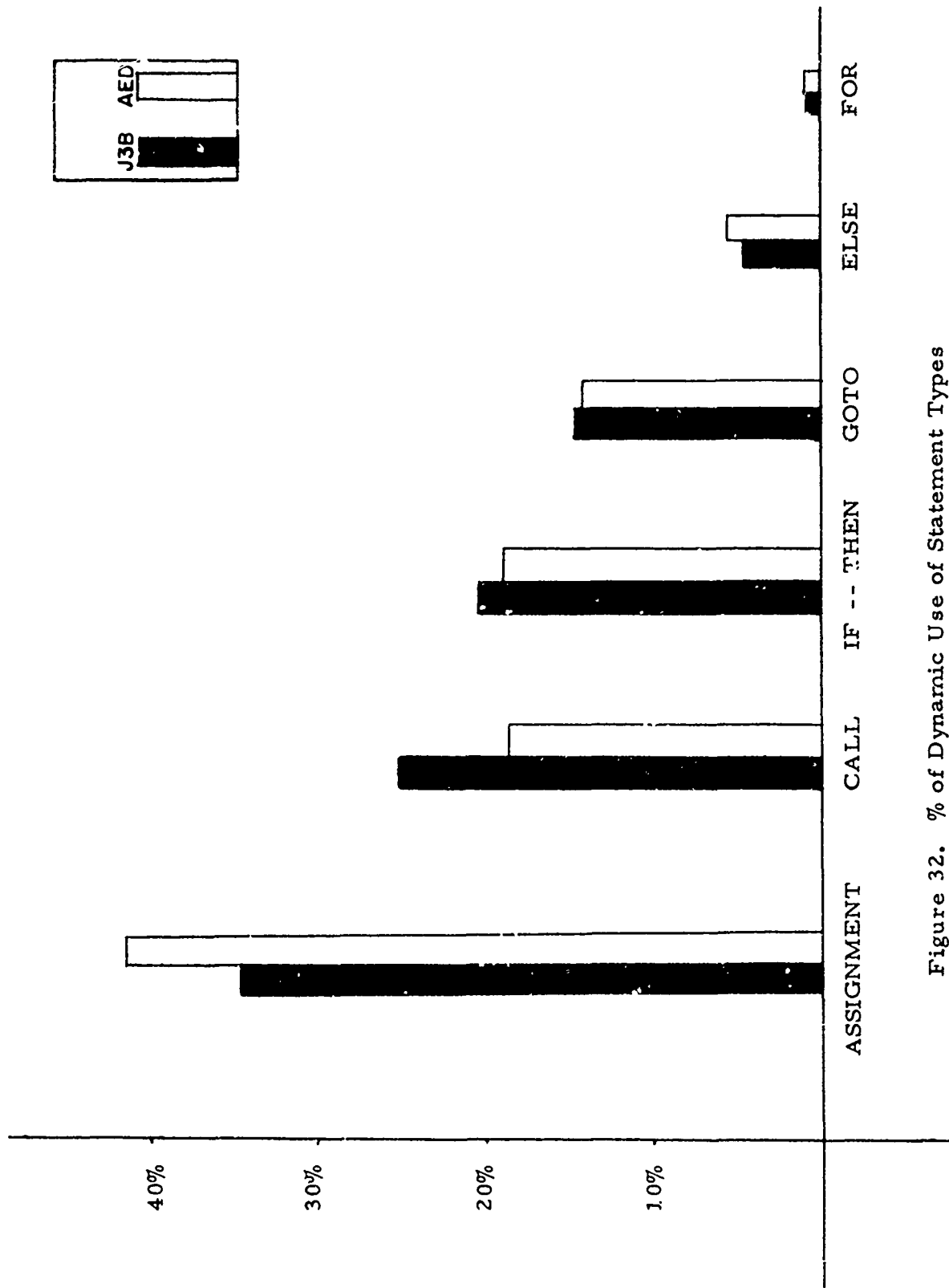
Figure 32. % of Dynamic Use of Statement Types

Statements following 'THEN', 'ELSE', and 'DO'. The bar graphs
presented in Figures 33, 34, and 35 are the dynamic counterparts of the bar
graphs presented in Figures 16, 17, and 18 respectively.

After 'THEN'. Figure 33 shows that the compound statement (BEGIN)
category remains the most heavily used, with the CALL, GOTO, and Assign-
ment statement group being about equally used, in second place. IF state-
ments are of less importance, and FOR almost never follows THEN. The
dynamic weighting increased the relative frequency of occurrence of the
CALL, GOTO, and assignment group, but overall left the picture very
similar to the static case.
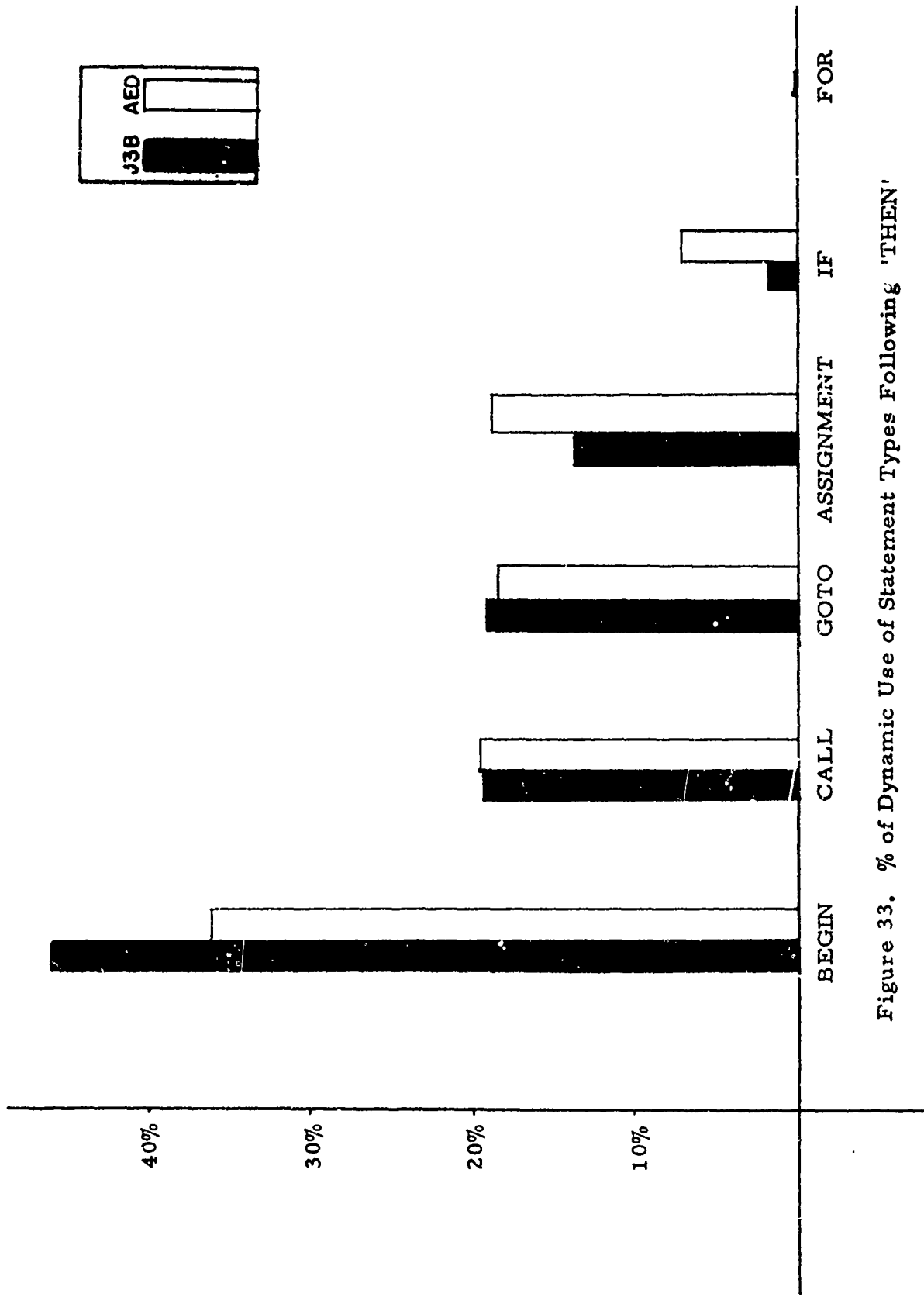
Figure 33. % of Dynamic Use of Statement Types Following 'THEN'

<u>After 'ELSE'.</u>   The dynamic bar graph (Figure 34) shows the same relative usage patterns that appeared in the static bar graph (Figure 17). Assignment statements in the AED compiler gained the most in relative frequency of occurrence due to the dynamic weighting.
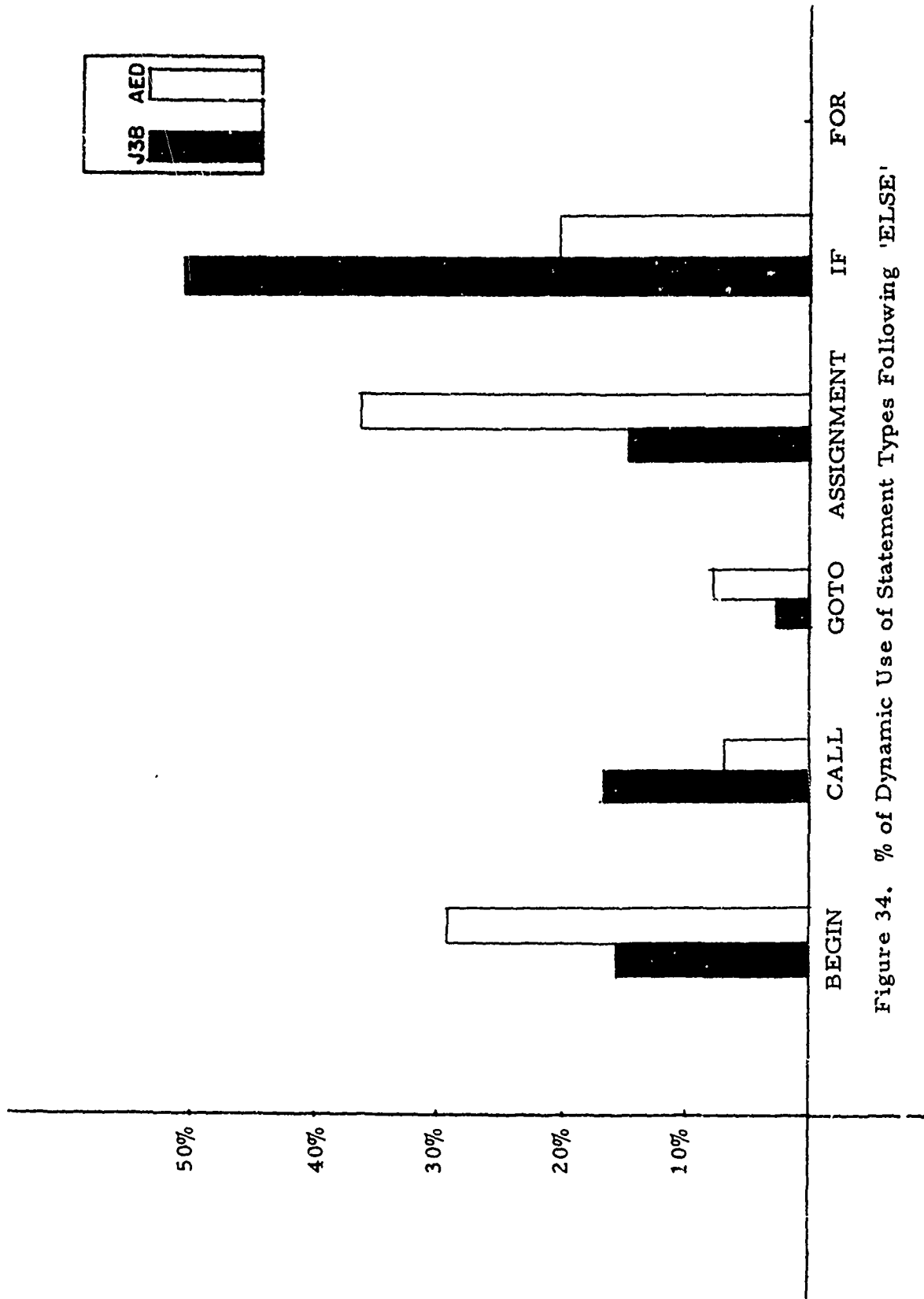
Figure 34. % of Dynamic Use of Statement Types Following 'ELSE'

<u>After 'DO'</u>. In Figure 35 it is seen that the major effect resulting from applying the weighting factors is to level off the differences between the AED and J3B profiles in the ASSIGNMENT and IF categories. For example, the fewer number of assignment statements in FOR loops are executed in the AED compiler more frequently than in the J3B compiler. It should be noted that the weights were determined by the number of times the procedure containing the FOR statement was <u>entered,</u> rather than by the the number of times the loop was actually executed; therefore, the number of statements actually executed inside loops is probably under represented in the data shown in the bar graph.
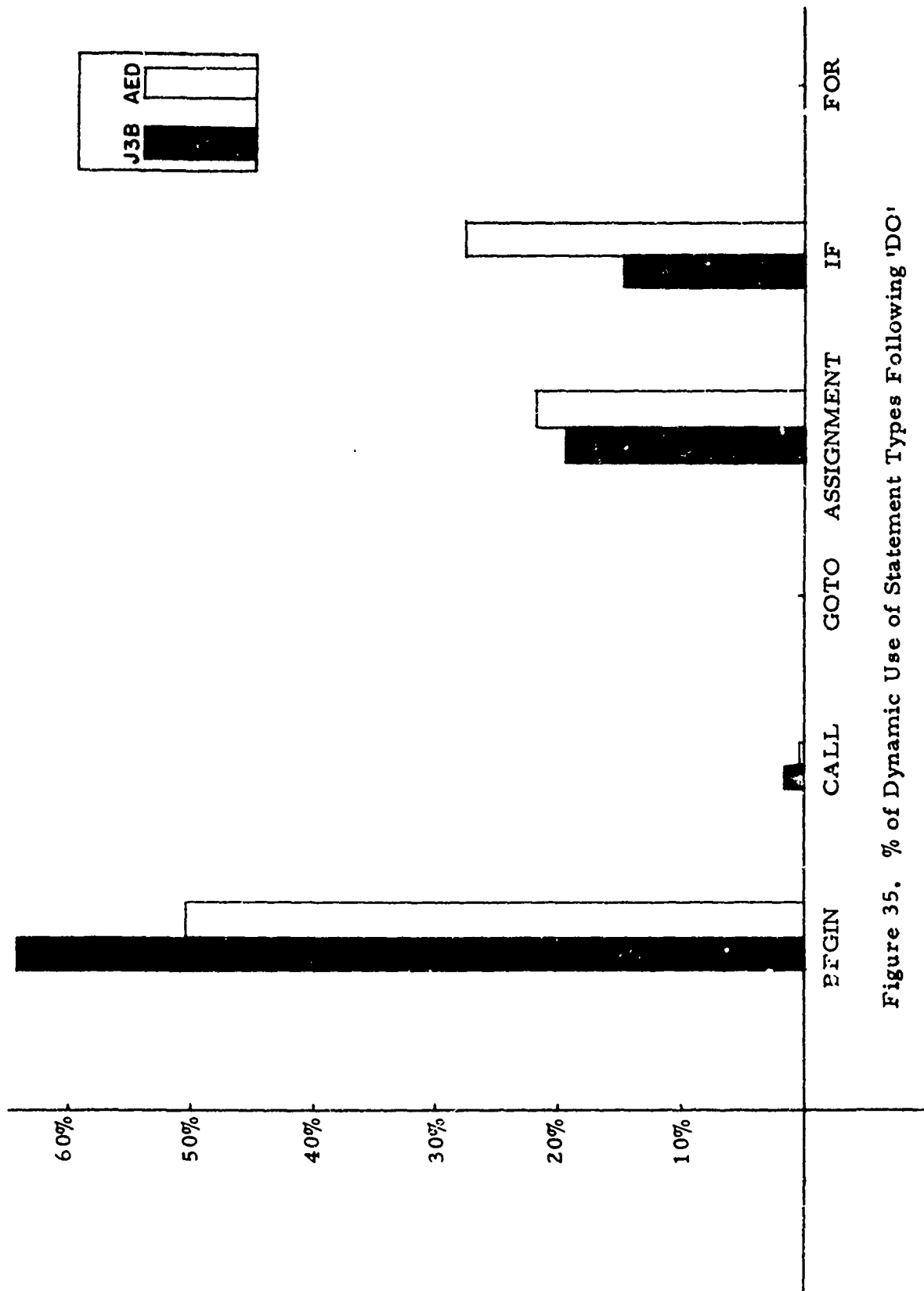
Figure 35. % of Dynamic Use of Statement Types Following 'DO'

266

Arithmetic operators. The bar graph presented in Figure 36 is the dynamic counterpart of the bar graph presented in Figure 19 .

The application of weighting factors produced very little effect in the relative frequencies of use of arithmetic operators. In both the dynamic bar graph (Figure 36) and the static bar graph (Figure 19), the operators "+" and "-" occur with a frequency of about ninety percent for both the AED and J3B profiles.
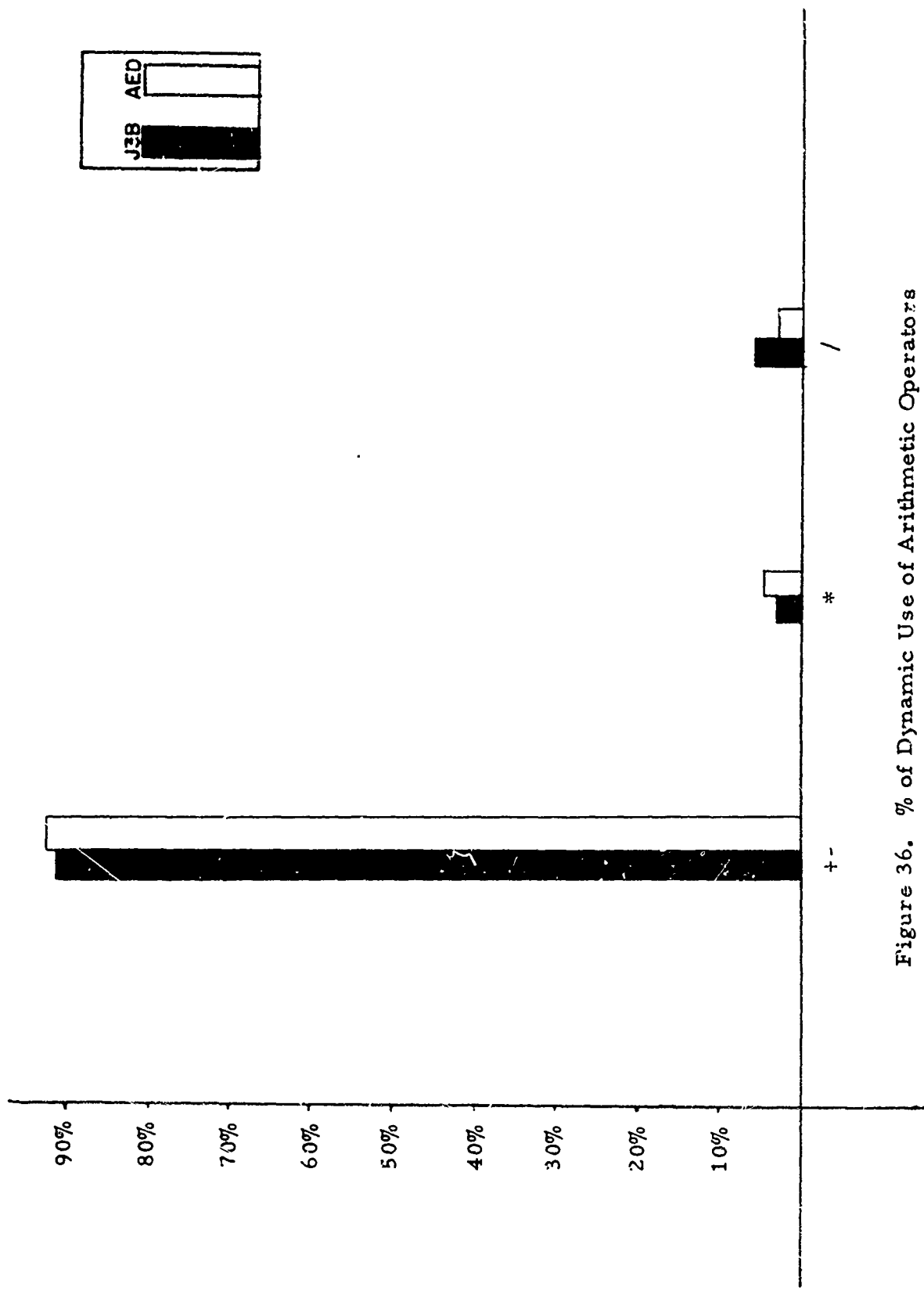
Figure 36. % of Dynamic Use of Arithmetic Operators

Arithmetic forms. The bar graph presented in Figure 37 is the dynamic counterpart of the bar graph presented in Figure 20 .

In Figure 37 the dynamic bar graph shows an increasing frequency of occurrence of the form A OP B, while the forms A OP 1,2 and A OP L* are reduced in frequency of occurrence as compared with the static bar graph of Figure 20 .

(See corresponding sub-section of Section 4 in Chapter 11 for an explanation of the notation used for labeling the bars of this bar graph.)
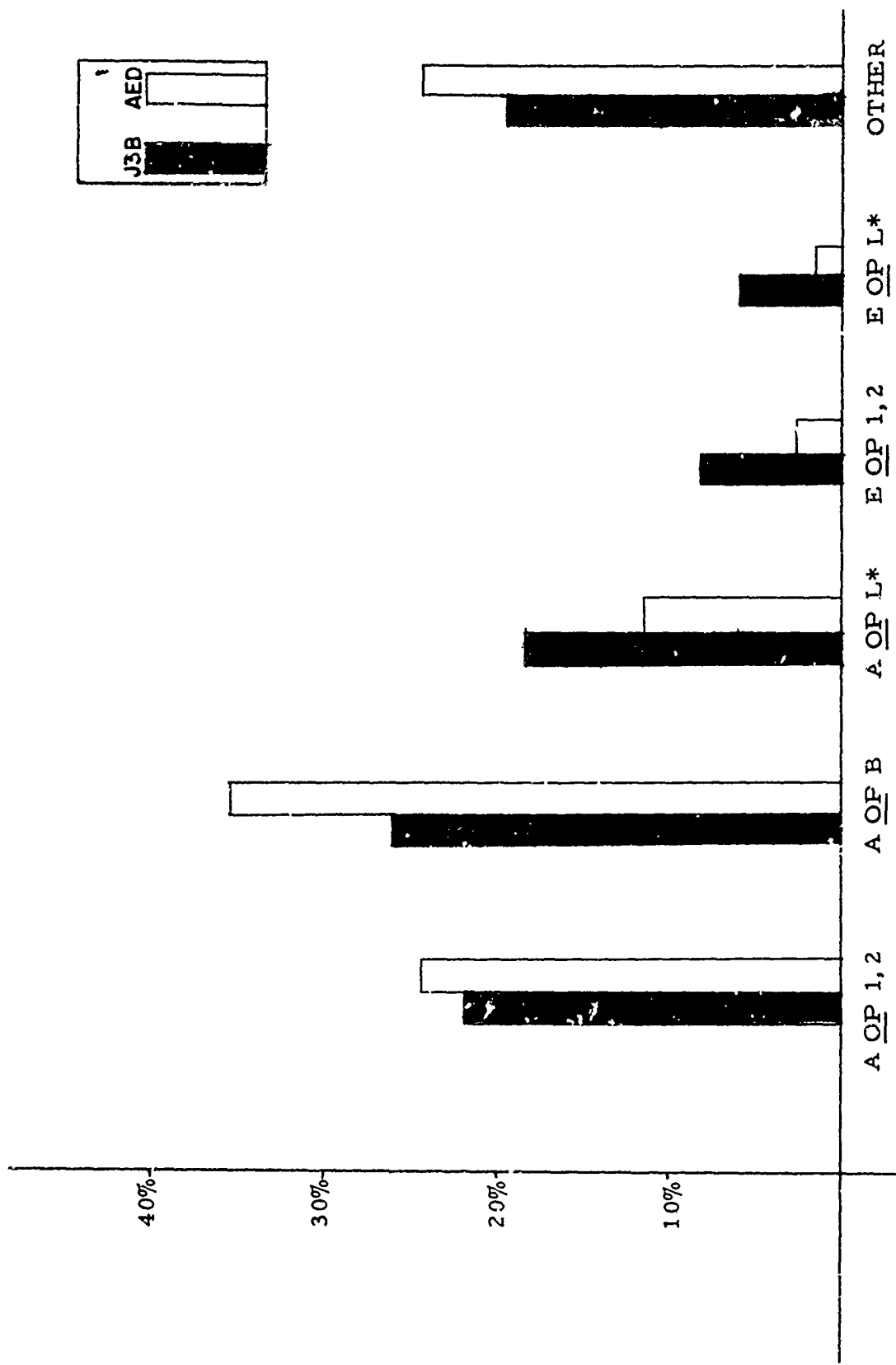
Figure 37. % of Dynamic Use of Arithmetic Forms

Boolean operators. The bar graph presented in Figure 38 is the dynamic counterpart of the bar graph presented in Figure 21 .

There are no noteworthy changes in the differences between the dynamic and static bar graphs of the use of Boolean operators.
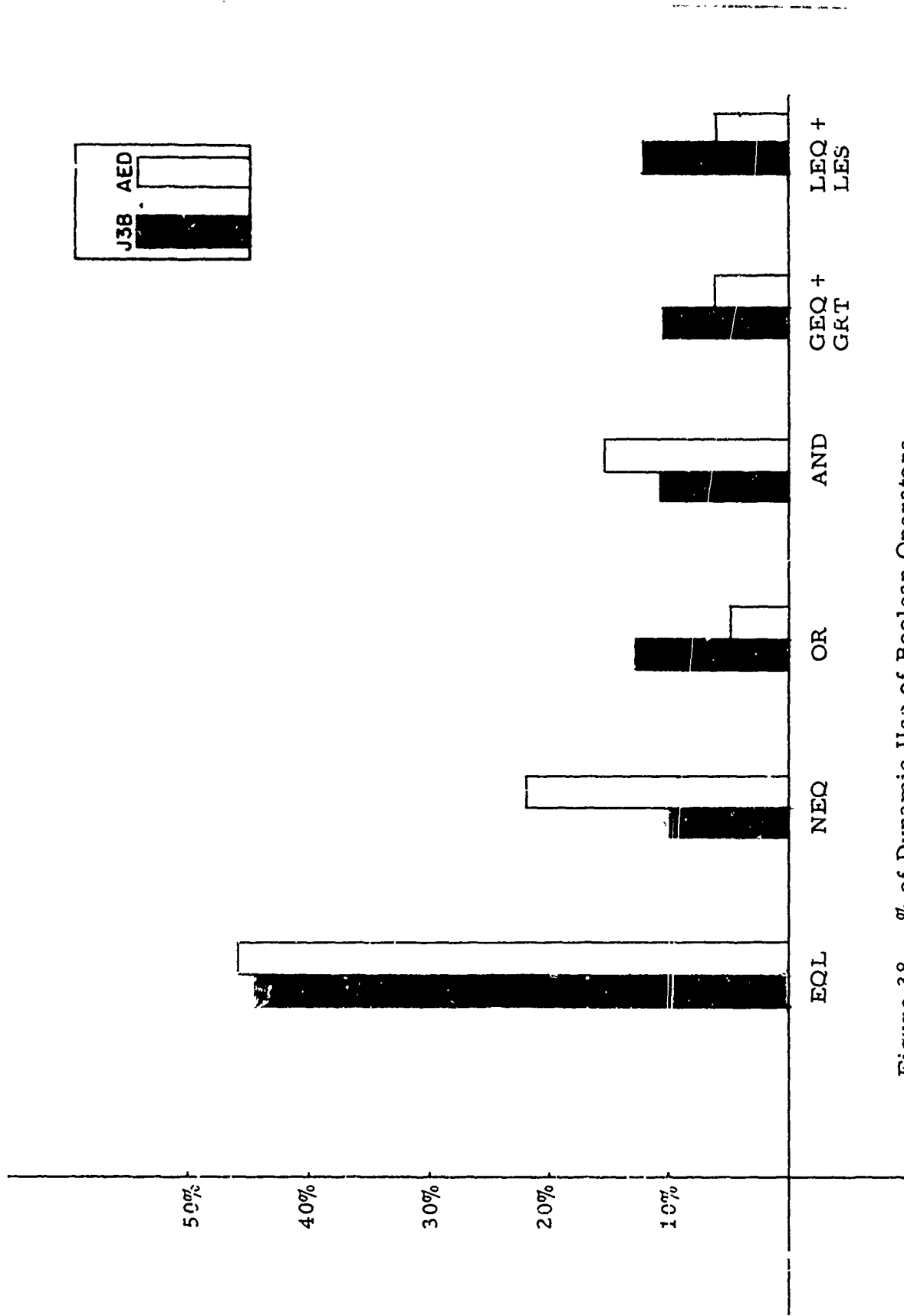
Figure 38. % of Dynamic Use of Boolean Operators

Boolean forms. The bar graph presented in Figure 39 is the dynamic counterpart of the bar graph presented in Figure 25 .

There are no noteworthy changes in the differences between the dynamic and static bar graphs of the use of Boolean forms.

(See corresponding sub-section of Section 4 in Chapter 11 for an explanation of the notation used for labeling the bars of this bar graph.)
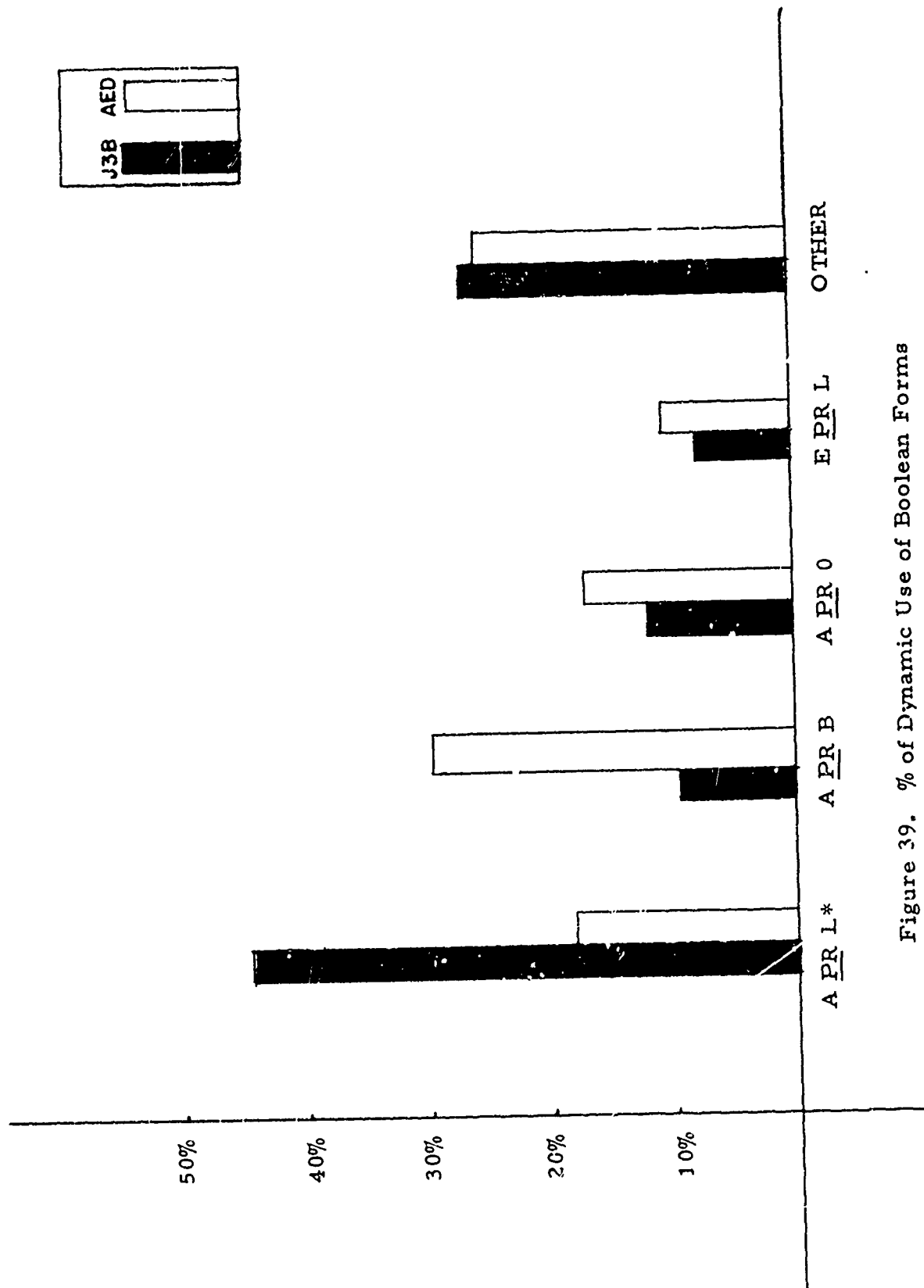
Figure 39. % of Dynamic Use of Boolean Forms

<u>% of Arithmetic operators with n R.H.S. operators.</u> The histogram presented in Figure 40 is the dynamic counterpart of the histogram presented in Figure 23 .

There are no noteworthy changes in the differences between the dynamic and static histograms of occurrences of arithmetic assignment statements with n arithmetic operators.
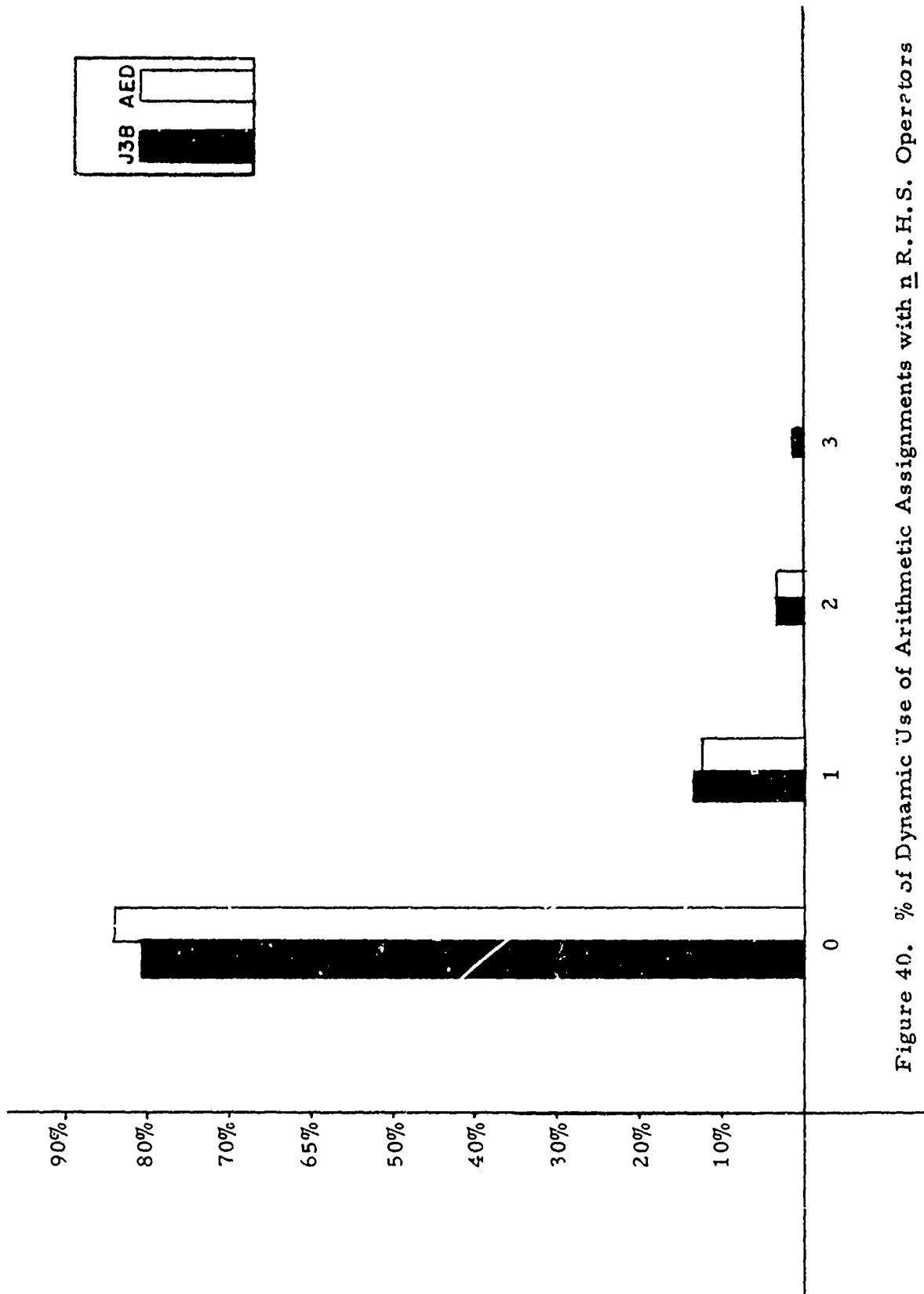
Figure 40. % of Dynamic Use of Arithmetic Assignments with n R.H.S. Operators

<u>% of Boolean expressions with n operators.</u> The histogram pre-
sented in Figure 41 is the dynamic counterpart of the histogram presented
in Figure 24.

The dynamic histogram shows both significant increase in the occur-
rences of Boolean expressions with 1 operator, and a reduction in the occur-
rences with 0 operators, as compared with the static histogram. This
indicates that forms such as IF A == B (1 operator) are used in procedures
that are called frequently, such as, for example, in input reading and code
generation procedures. On the other hand, the form IF A (0 operators)
occurs more frequently in rarely called procedures such as, for example,
initialization, error handling, etc.
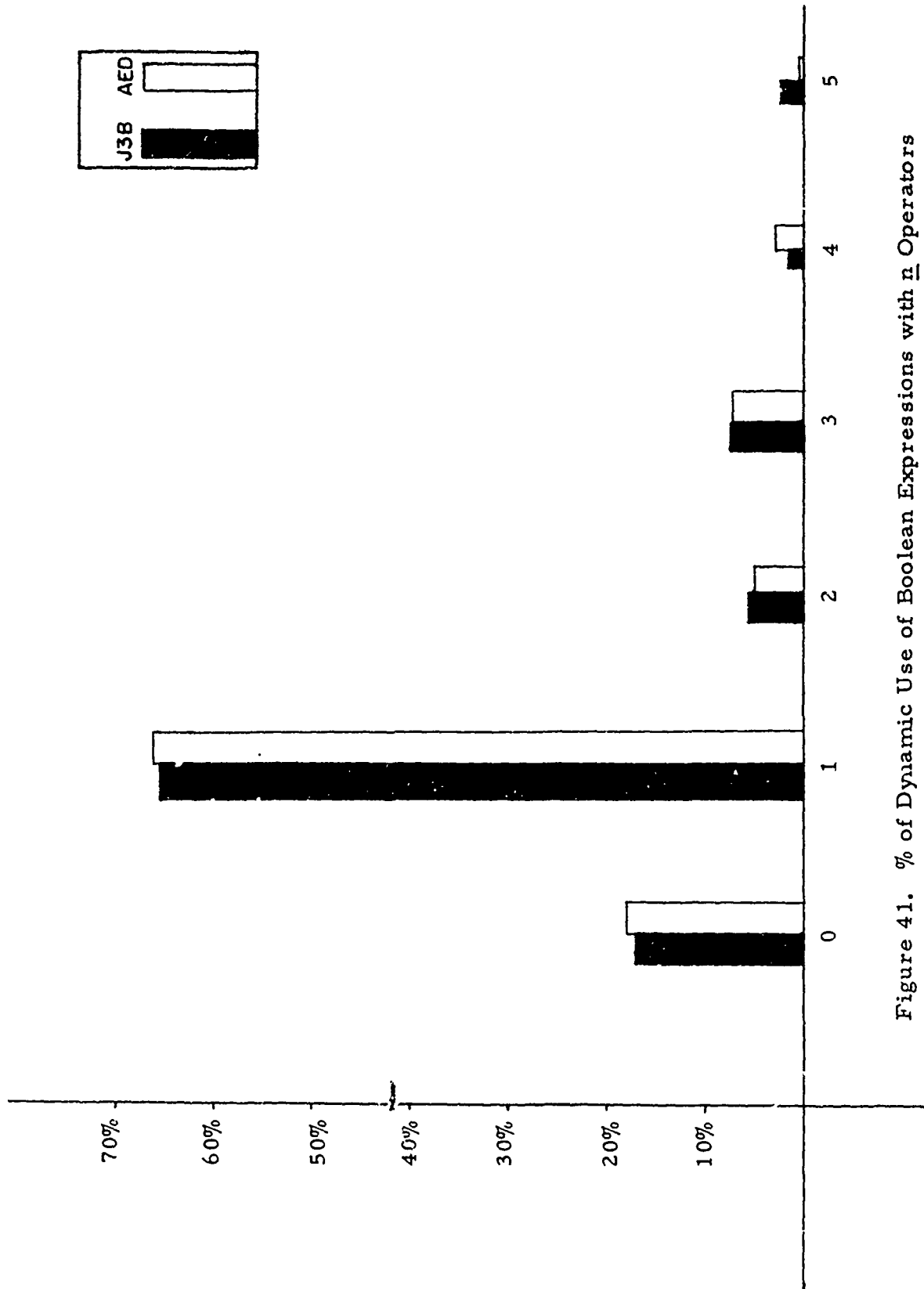
Figure 41. % of Dynamic Use of Boolean Expressions with n Operators

278

% of procedure and function calls with $\underline{n}$ arguments. The histogram presented in Figure 42 is the dynamic counterpart of the histogram presented in Figure 25 .

There are no noteworthy differences between the dynamic and static histograms of the occurrences of procedure or function calls with $\underline{n}$ arguments.
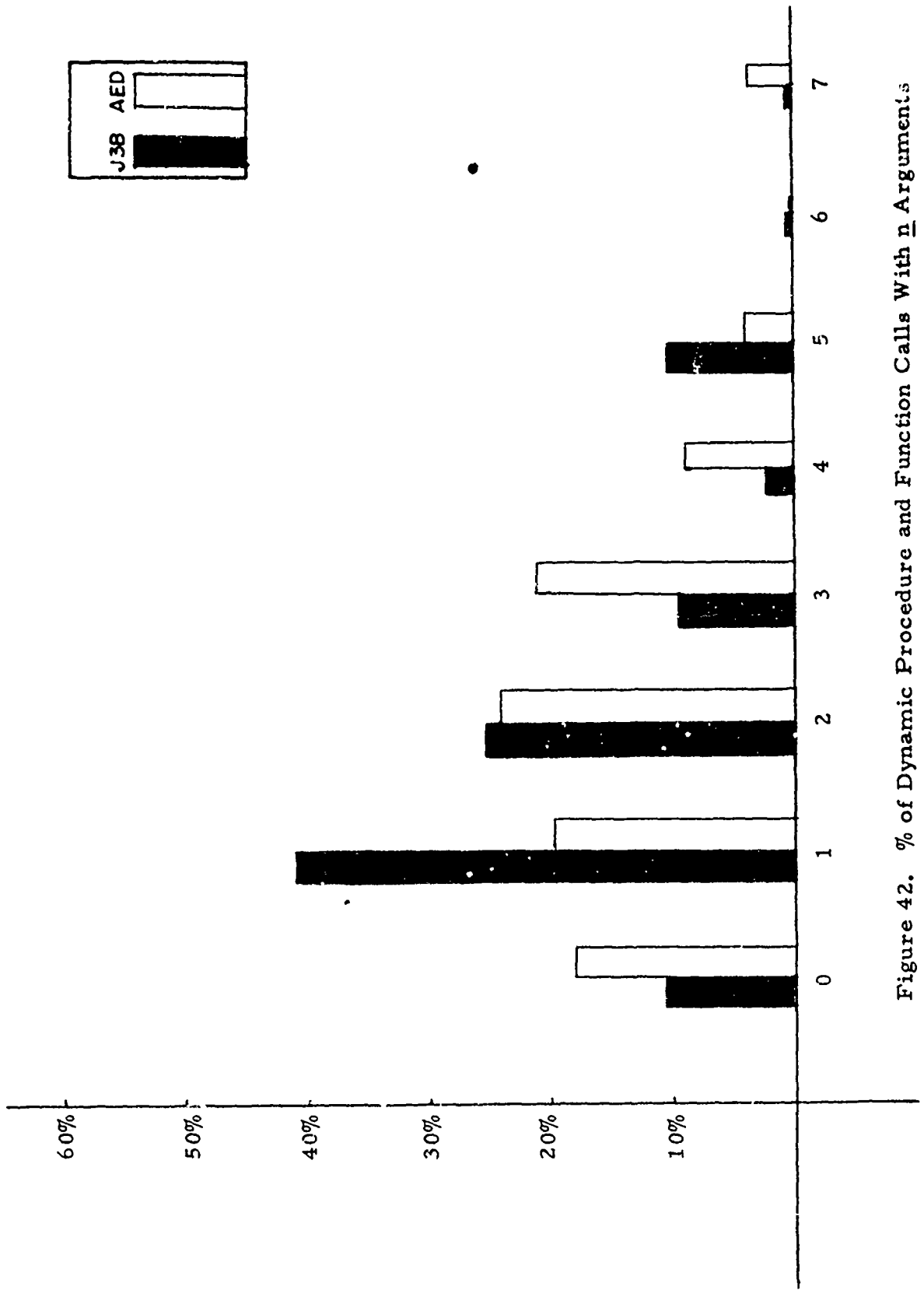
Figure 42. % of Dynamic Procedure and Function Calls With n Arguments

% of executable statements nested $\underline{n}$ deep in FOR loops. The histogram presented in Figure 43 is the dynamic counterpart of the histogram presented in Figure 26.

There are no noteworthy differences between the dynamic and static histograms of the occurrences of executable statements nested $\underline{n}$ deep in FOR loops.
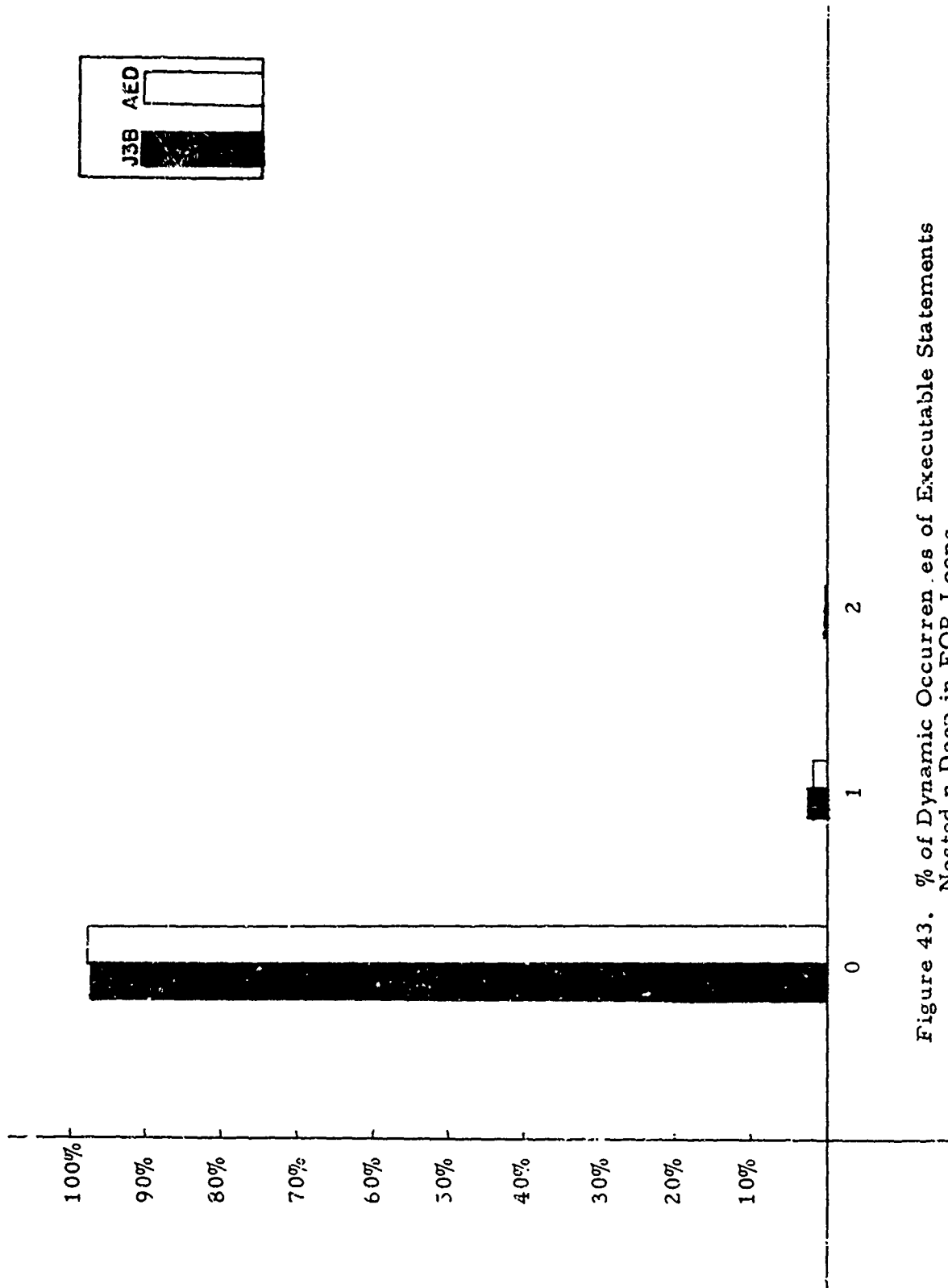
Figure 43. % of Dynamic Occurrences of Executable Statements Nested n Deep in FOR Loops

CHAPTER 13
HOW TO EVALUATE SPECIAL FEATURES

1. ## Introduction

In this chapter a brief discussion is given on how special features might be evaluated. A more detailed discussion would require an investigation beyond the scope of the present study. Section 2 discusses the special features question with respect to ease of use features, and Section 3 discusses the question with respect to case of maintenance features.

2. ## Ease of Use Features

Ease of use features are matters of human engineering -- that is, such features are included in a compiler in order to make a compiler easier to use.

Examples of such special features are:

- Diagnostics that are easier to use for debugging purposes.
- Error detectors in compiled code which are data dependent.
- Procedure interfaces that check for proper parameter/ argument data type matching.
- Detecting (nearly) all syntactic errors during one compilation. (This requires very good error recovery mechanisms during the parsing phase of computation.)
- Hooks for assisting the debugging of software:
  - Traces.
  - Breakpoints.
  - Symbolic debugging.
  - Patching.

The benefits to be derived from such features are:

- Reduce the number of debugging compilations per programming tasks.
- Reduce the programmer time required to generate debugged programs.

- Reduce the experience required of the programmers for obtaining equivalent work output.

- Increase reliability of programs believed to be debugged.

Each of these factors can, in principal, be given a dollar value, although for most of these factors, there are no techniques available to evaluate quantitatively these features with respect to the above benefits.

The only research method that suggests itself for assigning a dollar benefit valuation to these features is an elaborate series of psychological experiments. These experiments would be designed to determine quantitatively to what extend the benefits indicated above would actually occur from the presence of an ease of use feature in a compiler. Once this information can be obtained, then dollar values can be assigned to these features on the following basis.

- Each debugging compilation represents a dollar cost for the use of the computer facilities to perform the compilation, and in addition for the labor cost of the programmer in setting up and interpreting results from a debugging run.

- A reduction in programmer experience requirements is reflected in reduced programmer salaries.

- Increased program reliability reduces maintenance costs.

The above considerations indicate how future research might be directed at providing a basis for determining the dollar benefit to be associated with ease of use features. The cost of including these features in a compiler is much harder to evaluate. For this reason, our recommendation is to use a contractural approach to quantify the cost side of the analysis. In particular, we recommend that a vendor be required to provide separate prices for the inclusion of these features. However, it is inappropriate to make such requirements for the separate pricing of features before the dollar benefit data are available.

3. Ease of Maintenance Features

The following topics are concerned with special features of the compiler which facilitate maintenance functions.

- Portability (changing host machine).
- Retargetability (changing target machine).
- Maintenance (bug fixing).
- Enhancability (adding features).

It is our opinion that the simplest way to handle the cost/benefit evaluation of ease of maintenance features is by means of including the requirement of providing for options in Requests for Proposals. For example, if the buyer considers portability important, then the possibility of wanting an additional compiler to run on a specific second computer should be firmly established. Then, the RFP would require the vendor to provide an option price and option execution price for the second compiler. If the vendor normally uses an architecture that makes portability easy, the option price will likely be very low (perhaps zero) and the option execution price will reflect the degree of portability of his architecture. (That is, a poorly portable compiler would result in an option execution price similar to that of the original compiler -- an easily portable compiler would have a much smaller option execution price.) If the vendor would normally use a high performance architecture (to gain incentive profits), then the option price might be fairly expensive since vendor might choose a more portable architecture which would reduce his expectations of incentive profits.

Retargetability can be handled exactly in the same manner as portability. Enhancability can be handled by specifying options for specific features that are anticipated would be added later to an easily enhancible compiler. Maintenance (bug fixing) can be handled by means of warranties and options on warranties.

CHAPTER 14
CONCLUSIONS

1. Introduction

This chapter presents a summary of the conclusions reached as a result of the present study. For each indicated conclusion, cross references are provided to the relevant sections of this report which contain supporting material for the conclusion.

Section 2 presents conclusions for the study as whole. Sections 3, 4, 5, and 6 present the conclusions from studying respectively the architecture/algorithms question, the same environment question, the environment equalizing question, and the special features question. (A full statement of these four questions is presented in Chapter 1.)

2. Conclusions from the Study as a Whole

Listed below the conclusions reached from the study as a whole.

- The main technical objective for the study was to develop criteria that should be used in measuring the performance of compilers. This objective was achieved by the study. (See Chapter 15 for a summary of the criteria.)

- The criteria developed by the study satisfies the application objectives presented in Chapter 1. That is, the criteria are useful in selecting off-the-shelf compilers, for preparing RFP's for compilers and providing a basis for acceptance test design, and for choosing computer hardware where compilers are to be purchased either as a package with the hardware, or separately. (See Chapter 2 for an overview of these uses.)

3. Conclusions from Studying the Architecture/Algorithms Question

Listed below are the conclusions reached from studying the architecture/algorithms question.

- The specific secondary technical objective of the study was to determine the answer to two specific questions:

  - Is there a particular parsing scheme that is most efficient for all languages and user types, or is each language better suited by a unique parsing system? The study determined the answer to both parts to be No! (See Chapter 5.)

286

- Is there a relationship between table searching methods and the type of language which is being compiled? The study determined the answer to this question to be No! (See Chapter 4.)

- The full statement of the architecture/algorithms question is: Can analysis of a compiler's architecture and algorithms provide a basis for making valid judgements about the performance that should be expected from a compiler? The conclusion reached in the study is that this question should be answered No! (This conclusion is supported by the entire study of this question as described in Chapters 3, 4, 5, 6, and 7.)

- From the study of architectural choices in compiler design (Chapter 3), the following conclusions were reached:

  - With respect to compiler architectures, one-pass compilers are faster and larger than multi-pass compilers.

  - Multi-pass compilers permit more extensive optimizations, and therefore can produce more efficient object code.

  - Choices of algorithms for parsing and code generation are generally made for other than performance reasons. Generally, the reasons relate to cost of development of the compiler.

  - Some generalizations on the relative efficiency of table look-up algorithms are possible, but such generalizations are mainly related to specific internal architectural purposes for the table rather than external factors such as the language being compiled.

  - Optimization methods are highly varied, and no useful quantitative generalization was found which could relate compiler performance and object code quality for a particular individual or class of optimizations.

- From the study of table look-up algorithms (Chapter 4), it was concluded that each algorithm has performance characteristics that favor its use for particular types of compiler tables. The types of tables that match these special performance characteristics relate to areas of compiler activity rather than to types of languages.

- From the study of five categories of parsing techniques (Chapter 5), the following conclusions were reached:

- For most parsing techniques in common use, differences in implementation overshadow differences in technique in impact on performance.

- For one technique category ("general techniques"), which is not commonly used in compilers, the performance expected would generally be poor due to the generality of the technique. This expected poor performance is probably the reason the technique is <u>not</u> commonly used in compilers.

- The reasons a particular technique is selected for use in a compiler are generally distinctly independent of performance considerations.

- Aside from performance considerations, it is possible to make some general statements about various advantages or disadvantages one parsing technique would be expected to have in comparison to the other techniques. (See Section 4 of Chapter 5.)

- The study of optimization algorithms (Chapter 6) considered 19 machine independent optimizations and 9 machine dependent optimizations. This subject is so broad that no general conclusions seem appropriate. All optimizations are intended to trade an improvement in object code quality for a reduction in compiler speed. Some are applicable to single pass compilers and some to multi-pass compilers, and some to both. (See Chapter 3 discussion of optimizations and how they fit into architectures.)

- The study of code generation algorithms (Chapter 7) reached the general conclusion that there is no definite advantage or disadvantage in comparing three different methods of organizing and implementing code generators.

4.   Conclusions from Studying the Same Environment Question

Listed below are the conclusions reached from the study of the same environment question.

- In Chapter 8, criteria are defined for evaluating compilers in the same environment having the same special features. These criteria involve the use of User Profiles and Compiler Performance Profiles, both based on a list of language elements such as those listed in Section 5 of Chapter 8. These profiles can be combined to provide comparable dollar valuations of such compilers by methods discussed in Section 4 of Chapter 8.

288

*   The list of language elements for User Profiles presented in Section 5 of Chapter 8 satisfy the objectives for the list. These objectives are:

    *   The list is as complete as is reasonable to expect within scope of the study.

    *   The list incorporates sufficient detail so that it is reasonable to expect that no item in the list could likely occur with an excessively high frequency in user application programs. (The proof of this conclusion requires the actual generation of User Profiles.)

    *   It seems plausible that the collection of User Profile data for the list could be automated. (Several ideas on how this might be done are discussed in Section 3 of Chapter 8.)

*   Chapter 9 presents methods for generating test programs to be used in collecting Compiler Performance Profile data (for the elements of the list of User Profile language elements). These methods satisfy most of the objectives for such methods. These objectives are:

    *   The effects on compiler performance due to the individual elements of the list are reasonably well isolated from each other.

    *   The number of test programs which the methods require for the language elements should be small. However, for some language elements, the number becomes rather large. On the other hand, the methods are sufficiently simple that the generation of test programs could be automated.

    *   The methods are easy to understand and to put into practice.

    *   The measuring procedures specified (for calculating performance measured from the raw data collected from compiling and executing the test programs) are simple and straightforward.

5. Conclusions from the Study of the Environment Equalizing Question

Listed below are the conclusions reached from studying the environment equalizing question.

*   It is reasonable to expect that different compilers should have similar Compiler Demand Profiles. (See Chapters 11 and 12.)

289

- A typical Compiler Demand Profile can be used as a basis for defining a "compiler Gibson mix". (See Section 2 of Chapter 10.)

- A "compiler Gibson mix" (established by the methods described in Section 2 of Chapter 10) can be used to "equalize" environments. (See Section 3 of Chapter 10.)

6.   Conclusions from Studying the Special Features Question

The conclusions reached from studying the special features questions are listed below.

- Assigning dollar benefit valuations to ease of use features requires data not a present available. Psychological studies (described in Section 2 of Chapter 13) might be useful for developing these data.

- Assigning dollar cost valuations to ease of use features might be handled contracturally. (See Section 2 of Chapter 13.)

- The cost/benefit analysis of ease of maintenance factors could be facilitated by suitable use of contractural methods, provided a valid method of determining a dollar valuation of the performance of a compiler is developed. (See Section 3 of Chapter 13.)

CHAPTER 15
RECOMMENDATIONS

1. __General Recommendations__

Listed below is a summary of suggested general recommendations for using the results of this study.

- The criteria developed in this study for measuring the performance of compilers should be used as a basis for a number of compiler purchase activities. These activities are:

    - Purchasing off the shelf compilers.

    - Preparing RFP's for compilers and designing acceptance tests with respect to performance standards.

    - Choosing computer hardware and compilers as a package.

    - Choosing computer hardware when compilers are to be purchased separately.

(The criteria are summarized in Section 2. How these criteria can be used is discussed in Section 3.)

- The methods for measuring compiler performance (including effects due to environmental factors) should be used experimentally to establish their practical suitability and to determine how they should be modified to improve their usefulness. (These methods are described in Chapter 8.)

- The methods described in Chapter 10 should be used experimentally to normalize the performance measures derived by the Chapter 8 methods for environmental differences. That is, experimental use of the Chapter 10 methods should be undertaken to determine their practical suitability in "equalizing" environments.

- Further studies should be pursued to determine data needed to properly assign dollar benefit evaluations to ease of use features. (See Section 2 of Chapter 13.)

- Contractural methods should be developed to permit dollar cost/benefit analysis to be used in evaluating ease of maintenance features.

● A number of specific experimental studies should be
explored involving the use of the methods developed
in this study (Chapters 8 and 10) to refine the methods
and to further determine their usefulness. (These
suggestions for further studies are presented in
Section 3.)

2. Criteria Developed in the Study

The criteria for measuring the performance of compilers developed
in the present study are the following:

● The basic criteria to be used in evaluating a compiler
is the dollar cost of the compiler, and the dollar benefit
of the compiler. Other criteria presented below
constitute source information or derived combinations
of source information which contribute to an assignment
of a dollar benefit valuation to various aspects of a
compilers performance. (See Section 4 of Chapter 8.)

● The compiler's Compiler Performance Profile -- the
performance of a compiler with respect to elements
or constructions of the language operated on by the
compiler. Four performance measures ('the "directly
measureable factors") for each element are to be
determined:

  ● How the compiler's time to compile is effected
  by occurrences of the element in source programs
  being compiled.

  ● How the compilers space requirements (partition
  size) is effected by occurrences of the element.

  ● How much CPU time is required to execute the
  object code created by the compiler for an
  occurrence of the element.

  ● How much space is required for the object code
  resulting from an occurrence of the element.

● The user's applications programs' User Profile -- the
relative number of occurrences of an element or
construction of the language operated on by the
compiler in the user's application source language
programs. (This is the static User Profile.) Also,
the number of executions of occurrences of the
element is normal use of a user's application
programs. (This the dynamic User Profile.)

292

- The compiler's Compiler Evaluation Profile -- a combination of User Profile and Compiler Performance Profile representing performance measures for each of the four "directly measureable factors" with respect to a "typical" user program.

- The performance of a computer/operating system environment with respect to a "compiler Gibson mix". (See Chapter 10 for a detailed discussion of this criterion.)

- Administrative data defining the average degree of re-use of user applications programs. This information in combination with the Compiler Evaluation Profile can contribute to calculations of a number of useful dollar valuations for a compiler. (See Section 4 of Chapter 8.)

3. How the Criteria Can Be Used

Compiler acquisition situations. The results of our investigations should be useful in the acquisition of hardware and compilers in the following situations:

- Hardware selection.
- Buying off-the-shelf compilers.
- Preparing RFP's for compilers and designing acceptance tests for the delivered product.

With respect to hardware selection, there are two cases:

- Buying computer hardware with the intention of acquiring compilers separately.
- Buying computer hardware and compilers as a package.

Our investigation of the environment equalizing question (Chapter 10) supports the feasibility of establishing a "compiler Gibson mix". Such a "mix" is defined (Chapter 10) in terms of the static Compiler Demand Profiles for AED and J3B presented in Chapter 11. Further work is required to develop the assembly language representation of the elements of the "compiler Gibson mix" for one or more computers. Once all this work has been completed, the candidate computer systems could be compared in terms of their relative degree of support for compiler activity.

In addition to generating and using the "compiler Gibson mix",
a compiler purchaser should also generate and use a "user Gibson mix"
based on a User Profile, which would characterize the user programs
to be compiled and/or executed on the new installation. If it is known
what fraction of the time the installation will be performing compilations
as executions of compiled program, then the degree of compiler support
can be suitably weighted and combined with the support expected for
User Programs to give an overall dollar benefit valuation for the
computer.

If hardware and compiler are to be acquired as a package, the
results of our investigation of the same environment question are
directly applicable. A Compiler Performance Profile could be generated
for each hardware/compiler pair which, when combined with the User
Profile, would provide directly a Compiler Evaluation Profile for the
combination. This figure of merit would characterize the performance
of the combination with respect to compiling and executing a 'typical"
user program.

If different compilers under consideration have different features
which would contribute to ease of use and ease of maintenance, additional
information derived from considerations of the special feature question
would have to be taken into account.

With respect to buying off-the-shelf compilers, the results of
our investigation of the same environment question should also be
directly applicable. The generation of a Compiler Performance Profile
for each compiler, combined with the User Profile will provide directly
a Compiler Evaluation Profile for each compiler's performance with
respect to a "typical" user program.

Preparing RFP's constitues the most complex problem of
determining how to use the results of the present study. If a compiler
for the desired language already exists on a currently used machine,
different from the host machine, then a Compiler Evaluation Profile
could be used to define the performance for the current machine and

294

compiler. Then, the performance of the computer, operating system with respect to the defined "compiler Gibson mix" could be used to calculate how an "equiv lently performing compiler" would perform on the host machine. The result of such a calculation would provide a base line of performance to be expected from a compiler acquired in response to an RFP. These expectations could be included in the RFP specifica ons, and could also be used to determine whether the delivered product performed adequately well during acceptance testing. Contr ts entered into using these procedures should probably include incentives (and/or penalties) for performance better (or worse) than the base line as determined in the above manner. The possibility of using incentives is discussed further below.

Incentives. Consider the problem of specifying in an RFP the required performance of a compiler. If the requirements are too restrictive, vendors will be discouraged from bidding. If requirements are too loose, the compiler bought will not perform as well as may be possible with existing technology. Consequently, we believe the possibility of performance incentives should be considered. An approximate base-line for the expected performance can be derived in the manner discussed above. If a vendor supplies a compiler which performs better from this baseline, this difference in performance can be assigned a dollar value. For example, if a compiler runs ten percent faster than the baseline, this results in a ten percent savings in computer expenses for the useful life of the compiler. Some fraction of this saving could be returned to the vendor as a performance incentive. It will require some care in establishing the baseline and in the incentive formulas to use to get the best performing compilers for each dollar expended, but it is likely that this approach will improve this desired end result as experience is gained. With this approach, it is not critical that the baseline be as accurately determined as a set of specifications for a fixed price procurement without incentives. If the baseline is made looser, then one should expect that the base price bid will be smaller, since the vendor can expect to make higher profits from the incentives.

4.   Suggested Topics for Future Study

Listed below are some topics whose study would contribute to
the establishing of well defined procedures in evaluating and procuring
compilers.   These topics are offered in the light of the present study,
and results from exploring these topics are directly related to the
criteria established by the present study for evaluating compilers.

- More Compiler Demand Profiles should be determined
  (by the methods described in Chapters 10, 11, and 12).
  In developing these profiles, information related to
  more elements and constructions of the source language
  in which the compiler is written should be included.
  The purpose of the study should be the improvement
  of the "compiler Gibson mix" tentatively defined in
  Chapter 10, based on the present preliminary study of
  how such a "mix" should be defined.

- The methods presented in Chapter 8 for automating
  the collection of User Profile data should be explored.
  Actual User Profiles should be developed. These
  profiles should be based on the list of language
  elements presented in Section 5 of Chapter 8 as a
  point of departure. Also, test programs for determining
  the performance of a compiler should be prepared
  using the methods presented in Chapter 5. These
  test programs should be compiled and executed,
  and Compiler Performance Profiles should be
  generated. The purpose of such studies would be to
  confirm the practical usefulness of the methods
  presented in the present study, and to acquire an
  accumulation of experience in the use of these
  methods that would provide the basis for improving
  them.

- Psychological studies, as discussed in Chapter 13,
  are necessary to develop data to provide the basis
  of assigning dollar benefit values to various ease
  of use features.

LIST OF REFERENCES

The references are organized below by chapters. For each chapter heading, the listed references were reviewed in conjunction with the study's activities that were related to the subject matter discussed in the chapter.

Chapter 4

Amble, Ole, and Knuth, D. E., Ordered Hash Tables, Stanford University Computer Science Department Report STAN-CS-73-767 (June 1973).

Batson, Alan, "The Organization of Symbol Tables," Comm. ACM 8, 4 (Feb 1965), 111-112.

Bell, J. R., et. al., "The Linear Quotient Hash Code." Comm. ACM 13, 11 (Nov 1970), 675-677.

Byrom, S. J. et.al., "Representation of Sets on Mass Storage Deviation Information Retrieval Systems," Proc. AFIPS Nat. Computer Conf., (1973), 245-250.

Clampett, H. A., Jr., "Randomized Binary Searching with Tree Structures," Comm. ACM 7, 3 (March 1964), 163-165.

Glass, Robert L., "An Elementary Discussion of Compiler/interputer Writing," Computing Surveys 1, 1 (March 1969), p. 55.

Gries, David, Compiler Construction for Digital Computers, Wiley, N.Y. (1971)

Knott, Gary D., "A Balanced Tree Storage and Regrieval Algorithm," Proc. 1971 Symp. on Information Storage and Retrieval, 175-196.

Knuth, D. E., The Art of Computer Programming, Vol 3: Searching and Sorting, Chapter 6, 309-569.

Lum, V. Y., "General Performance Analysis of Key-to-Address Transformation Methods Using an Abstract File Concept," Comm. ACM 16, 10 (Oct 1973), 603-612.

Lurie, D. and Vandoni, C., "Statistics for FORTRAN Identifiers and Scatter Storage Techniques," Software Practice and Experience 3, 2 (1973), 171-177.

Maurer, W. D., "An Improved Hash Code for Scatter Storage," Comm. ACM 11, 1 (Jan 1968), 35-38.

Morris, Robert, "Scatter Storage Techniques," Comm. ACM 11, 1 (Jan 1968), 38-43.

Peterson, W. W., "Addressing for Random Access Storage," IBM Journal (April 1957), 130-146.

Price, C. E., "Table Look-up Techniques," Computing Surveys 3, 2 (June 1971), 49-65.

Radke, C. E., "The Use of Quadratic Residue Research," Comm. ACM 13, 2 (Feb 1970), 103-105.

Scidmore, A. K. and Weinberg, B. L., "Storage and Search Properties of a Tree-organized Memory System," Comm. ACM 4, 1 (Jan 1963), 28-31.

Stanser, A. J., "Bracketing Technique in Elastic Matching," Computer Journal 16, 2 (May 1973), 132-134.

Sussenguth, E. H., Jr., "Use of Tree Structures for Processing Files," Comm. ACM 6, 5 (May 1963), 272, 279.

Chapter 5

Colmeraur, A., "Total Precedence Relations," Jour. ACM 17, 1 (January 1970), 14-30.

Conway, M. E., "Design of a Separable Transition-Diagram Compiler." Comm. ACM 6, 7 (July 1963), 396-408.

DeRemer, F. L., Practical Translators for LR (k) Languages, Project MAC M.I.T., (October 24, 1969), Available from DDC No. AD699501.

Eanes, R. S. and Goodenough, J B., Interim Report: Language Processing Technology, submitted to Frankford Arsenal under Contract DAAA 25-72C 0667, SofTech, (October 1, 1972).

Earley, J., "An Efficient Context-Free Parsing Algorithm," Comm. ACM 13, 2 (February 1970), 94-102.

Floyd, R. W., "Syntactic Analysis of Operator Precedence," Cour. ACM 10, 3 (July 1963), 316-333.

Gray, J. A. and Harrison, M. A., "Single Pass Precedence Analysis," Symp. Automata and Switching Theory, (1969).

Ichbia, J D. and Morse, S. P., "A Technique for Generating Almost Optimal Floyd-Evans Productions for Precedence Grammars," Comm. ACM 13, 8 (August 1970), 501-508.

Knuth, D. E., "On the Translation of Languages From Left to Right," Information and Control 8 (1965), 607-639.

Lalonde, W. R., An Efficient LALR Parser Generator, Technical Report CSRG-2, University of Toronto, (April 1971).

Lewis, P M. II and Stearnes, R. E., "Syntax Directed Transductions," Jour. ACM 15, 3 (July 1968), 465-488.

McKeeman, W. M. et.al., A Compiler Generator, Prentice Hall, (1970).

Wirth, N. and Weber, H., "EULER: A Generalization of ALGOL, and Its Formal Definitions: Part I," Comm. ACM 9, 1 (January 1966), 13-25.

Chapter 6

Aho, A. V. and Ullman, J. D., "Transformations on Straight Line Programs," Conf. Record Second Annual ACM Symp. on Theory of Computing, (May 1970), 136-140.

Allard, R W., Wolf, K. A., and Zemlin, R. A., "Some effects of the 6600 computer on language structures," Comm. ACM 7, 2 (Feb. 1964), 112-119.

Allen, F. E., "Program optimization," Annual Review in Automatic Programming Vol. 5, Pergamon, New York, (in press).

Allen, F. E., "Control Flow Analysis," ACM SIGPLAN Notices 5, (1970).

Allen, F. E. and Cocke, J., "Graph Theoretic Constructs for Program Control Flow Analysis," (Unpublished paper).

Anderson, J P., "A Note on Some Compiling Algorithms," Comm. ACM 7, 3 (March 1964), 149-150.

Apperson, Jerry L., Proposal for a Thesis on Optimal Evaluation Order for Expressions with Redundant Subexpressions, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA.

Bagwell, J. T., Jr., "Local Optimizations," ACM SIGPLAN Notices 5, 7 (1970).

Beatty, James C., "An Axiomatic Approach to Code Optimization for Expressions," Jour. ACM 19, 4 (October 1972), 613-640.

Busam, V. A. and Englund, D. E., "Optimization of Expressions in Fortran," Comm. ACM 12, 12 (December 1969), 666-674.

Cocke, John and Miller, Raymond, 'Some Analysis Techniques for Optimizing Computer Programs," Proc. Second Intl. Conf. of Systems Sciences, Hawaii, (Jan. 1969).

Cocke, John, "Global Common Subexpression Elimination," Proc. ACM Symp. on Compiler Optimization, (July 1970), 20-24.

Cocke, John and Schwartz, J. T., Programming Languages and their Compilers, Courant Institute of Mathematical Sciences, New York University, New York, (1970).

Day, W. H. E., "Compiler Assignment of Data Items to Registers," IBM Systems Journal 9, 4 (1970), 281-317.

Finkelstein, M., "A Compiler Optimization Technique," The Computer Journal 11, (1968), 22-26.

Gear, C. W., "High speed compilation of efficient object code," Comm. ACM 8, 8 (August 1965), 483-488.

Haynes, J R., An Optimizing Compiler for an Extended Version of the Floyd-Evans Production Language, TRM-12, Computation Center, The University of Texas, Austin, Texas, (March 1969).

Hecht, M S. and Ullman, J. D., "Flow Graph Reducibility," SIAM Jour. of Computing 1, (1971).

Heller, S. B., The Design of Software Systems by Iterative Optimization, TSN-3, Computation Center, The University of Texas, Austin, Texas, (March 1969).

Hill, V., Langmaack, J., Schwarz, H. R., and Seegmüller, G., "Efficient Handling of Subscripted Variables in ALGOL 60 Compilers," Proc. of the Rom Symp. on Symbolic Languages in Data Processing, Gordon and Breach, New York, (1962), 331-340.

Horwitz, Karp, Miller, and Winograd, "Index Register Allocation," Jour. ACM 13, 1 (January 1966), 43-61.

Juskey, H. D., and Wattenburg, W. H., "Compiling Techniques for Boolean expressions and conditional statements in ALGOL 60," Comm. ACM 4, 1 (January 1961), 70-75.

Johnson, R. K., A Survey of Register Allocation, Dept. of Computer Science, Carnegie-Mellon University, Available from DDC No. AD761529.

Kennedy, K., "A Global Flow Analysis Algorithm," Intl. Jour. of Computer Mathematics 3, (1971).

Kleir, R. L. and Ramamoorthy, C. V., "Optimization Strategies for Microprograms," IEEE Transactions on Computers C-20, (1971), 783-794.

Lowry, E., and Medlock, C. E., "Object Code Optimization," Comm. ACM 12, 1 (January 1969), 13-22.

Luccio, F., "A Comment of Index Register Allocation," Comm. ACM 10, 9 (September 1967), 572.

McKeeman, W. M., "Peephole Optimization," Comm. ACM 8, 7 (July 1965), 443-444.

Nakata, Ikuo, "On Compiling Algorithms for Arithmetic Expressions," Comm. ACM 10, 8 (August 1967), 494-92.

Nievergelt, J., "On the Automatic Simplification of Computer Programs," Comm. ACM 8, 6 (June 1965), 366-370.

Prosser, R. T., "Applications of Boolean matrices to the analysis of flow diagrams," Proc. Eastern Joint Computer Conf., Spartan Books, New York, (December 1959), 133-138.

Ramamoorthy, C. V., "Analysis of Graphs by Connectivity Considerations," Jour. ACM 13, 2 (1966), 211-222.

Redziejowski, R. R., "On Arithmetic Expressions and Trees," Comm. ACM 12, 2 (February 1969), 81-84.

Ryan, J. T., "A Direction-independent algorithm for determining the forward and backward compute point for a term of subscript during compilation," The Computer Journal 9, 2 (August 1966), 157-160.

Schneck, P. B., "Automatic Recognition of Parallel and Vector Operations in a Higher Level Language," Proc. ACM National Conf. (1972), 772-779.

Schneck, P. B. and Angel, E., "A FORTRAN to FORTRAN Optimizing Compiler," The Computer Journal 16, (Nov. 1973), 322-330.

Schneider, V., "On the Number of Registers Needed to Evaluate Arithmetic Expression," BIT 11, (1971).

Sethi, R. and Ullman, J. D., "The Generation of Optimal Code for Arith Arithmetic Expressions," Jour. ACM 17, 4 (October 1970), 715-728.

Yershov, A. P., "ALPHA - an automatic programming system of high efficiency," Jour. ACM 13, 1 (January 1966), 17-24.

Chapter 8

Fleiss, J. E. and Phillips, G.W., "A Statistics Gathering Package for the JOVIAL Language," Contract Report, Contract Number F30602-73-C-0062, RADC.

Knuth, D. E., "An Emperical Study of FORTRAN Programs", Software Practice and Experience 1. 2 (April-June 1971), 105-133.

Melkanoff, M.A. and Presser, L., "Software Measurements and Their Influence Upon Machine Language Design," AFIPS Conf. Proc. SJCC 34, (May 1969), 733-737.

Chapter 11

Knuth, D. E., "An Emperical Study of FORTRAN Programs," Software Practice and Experience 1, 2 (April-June 1971), 105-133.

APPENDIX 1

INSTRUMENTS USED TO GENERATE
COMPILER DEMAND PROFILES

As the term is used here, "instruments" are primarily counters of the specific forms of language elements and the associated logic to detect those forms. These instruments could have been installed in several different points within the AED compiling system: just after the lexical processing phase, at the parsing phase or at the code generation phase. Our choice was the code generation phase since, at this point, most of the bookkeeping for lexical analysis and parsing is out of the way, the semantics of the program have been determined, and most importantly, all of the significant processing passes through one major system module (COMPILE) which can be instrumented in a simple manner.

Since the source program has at this point been transformed into an intermediate representation, a few language elements such as parentheses and parentheses nesting are absent and can no longer be measured. Conversely, many other language elements become extremely simple to measure, especially those which have fairly local context.

To see why this is so we must examine somewhat more closely the form of the AED intermediate representation, which is a tree structure, bearing in mind that this form though widely used is not a universal one. The AED system module COMPILE is called within the hierarchy.

```
        ┌─────────────┐
        │    MAIN     │
        │  COMPILER   │
        │   CONTROL   │
        └─────────────┘
               │
               ▼
        ┌─────────────┐
        │    TREE-     │
        │   WALKER     │
        └─────────────┘
               │
               ▼
        ┌─────────────┐
        │   COMPILE    │
        │      O       │
        └─────────────┘
               ¦
               ▼
        ┌ ─ ─ ─ ─ ─ ─ ┐
          STATISTICS
        │ GATHERER    │
        └ ─ ─ ─ ─ ─ ─ ┘
```
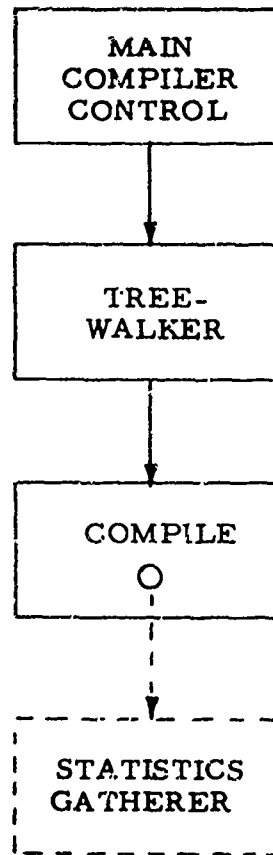
The compiler control box represents the code generation phase.
The tree walker represents the routines which systematically (recursively)
moves through tree structure representation, and calls COMPILE for each
node in the tree. The COMPILE module calls the instrumentational module
(the statistics gatherer), with appropriate parameters, each time it is
called, (once each time the tree walker passes a node). It is this node-
by-node processing which makes the collection of statistics on a local
context basis easy to implement. In addition, all relevant program logic
and variables for counters are maintained in the statistics gathering
routine and require little additional support from the AED compiler.
After the return from the statistics gatherer, COMPILE then dispatches
control to numerous routines which actually emit code for the node being
processed.

Without going into excessive detail, we can present a useful
description of the AED tree structure format, node types, branches, and
the traversal scheme used by the tree walker. It is easiest to do this
through actual examples of fairly simple program constructions. Consider
the phrase A + B + C. The AED compiler builds an interval tree for this
phrase, which may be represented as follows:

Legend:

- - -▸ minor pointer

/ left context pointer

\ right context pointer

/ \ major pointers

(OP) operator node
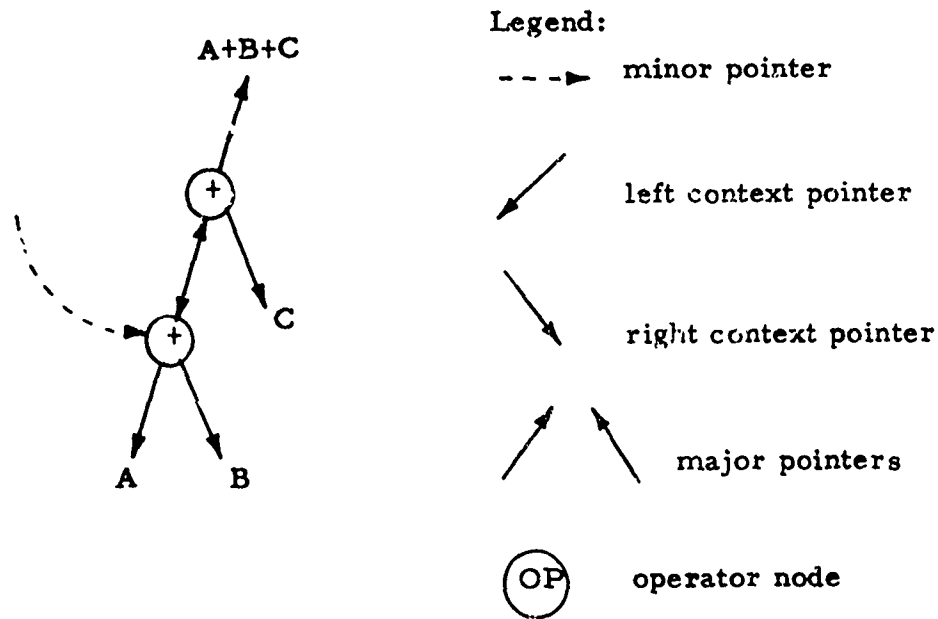
In the above diagram, the tree walker is brought to the lowest level node of the subtree for A+B+C via a _minor_ pointer originating at some node higher in the tree. This node is then passed to COMPILE which generates code for 'A+B' after first allowing the statistics gatherer to examine the node and classify it. The variables 'A' and 'B' are pointed respectively by the left and right context pointers of the '+' node. A return is made by COMPILE to the tree walker which then takes the major pointer to the next '+' node. The bi-directional arrow indicates that this major pointer for the lower '+' matches with the left context pointer of the higher '+'.

Another call is made to COMPILE (and subsequently to the statistics gatherer and code is generated to add 'C' to 'A+B'. On return, the major pointer for the upper node is used to reach the node next in the logical program sequence.

Consider the second example represented by the diagram below, for the statement A = 0:
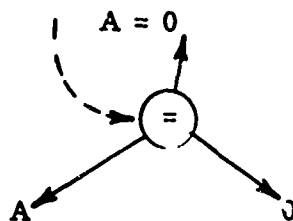
This rather simple subtree is reached via a minor pointer to the "=" node. The statistics gatherer (reached via COMPILE) examines the left and right context and then records an occurrence of an assignment statement.

As a third example, consider the statement
IF I EQL J THEN A = B ELSE A = C. This is represented by the following tree:

IF I EQL J THEN A = B ELSE A = C

ELSE

THEN

=

IF

=

A   C

NIL

EQL

A   B

I   · J
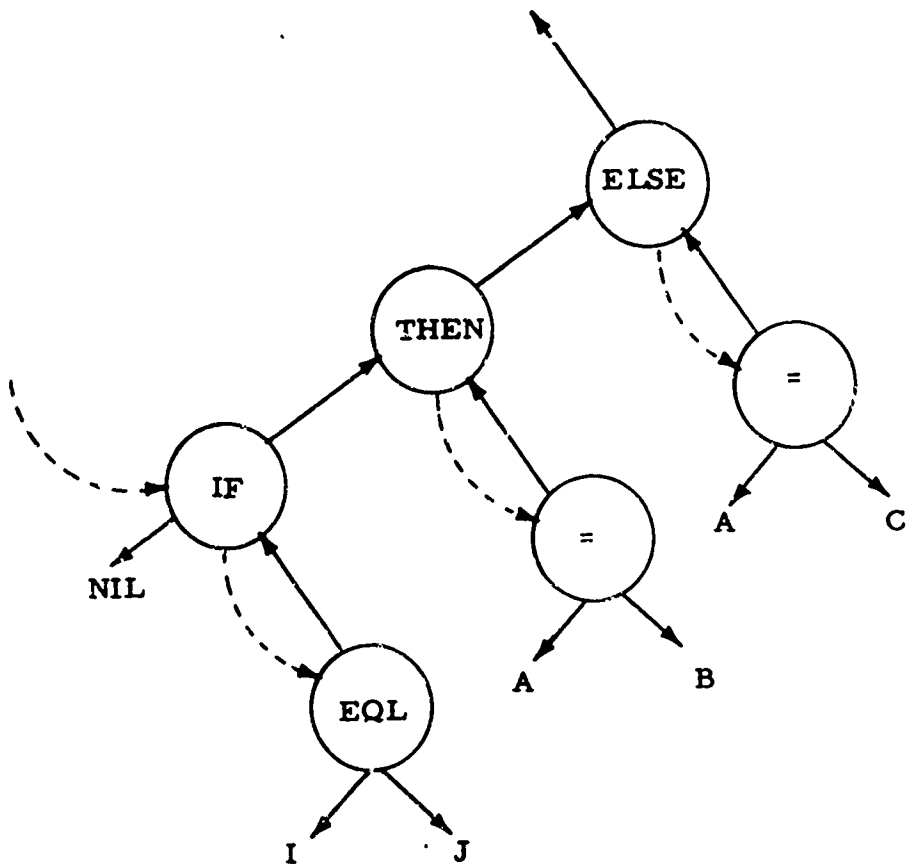
This third example has four minor pointers which are used to traverse the tree in the following sequence: IF_1, EQL, IF_2, $THEN_1$, =, $THEN_2$, $ELSE_1$, =, $ELSE_2$, where the subscript indicates the first or second passage of the node, and absence of any subscript signifies the only passage. The statistics gathered in the course of the tree traversal collects counts for the IF-THEN-ELSE construction and its constituent parts.

APPENDIX 2
TEST PROGRAMS USED TO GENERATE
COMPILER DEMAND PROFILES

1. Description of Tests

Section 2, below, presents a listing of the two test programs plus
all external data declaration files used by the two programs for use in
the dynamic compiler testing effort. Listings of both the J3B and the
AED versions of the programs and data files are included.

The two groups of test files are listed in the following order in
Section 2:

	File Name	Type	Data Files Referenced
Group 1 (J3B)			
1.	FILE N.KFNUP	J3B program	FILE NRBGNAV
2.	FILE C.PNHIT	J3B program	FILE CPNOWN
			FILE CNDDATA
			FILE NRBMAST
			FILE NRBAUX
			FILE NRBGNAV
			FILE CPNCONST
3.	FILE CPNOWN	J3B data declarations	(None)
4.	FILE CNDDATA	J3B data declarations	(None)
5.	FILE NRBMAST	J3B data declarations	(None)
6.	FILE NRBAUX	J3B data declarations	(None)
7.	FILE NRBGNAV	J3B data declarations	(None)
8.	FILE CPNCONST	J3B data declarations	(None)
Group 2 (AED)			
1.	N.KFNUP AED	AED program	NRBGNAV AED
2.	C.PNHIT AED	AED program	CPNOWN AED
			CNDDATA AED
			NRBMAST AED
			NRBAUX AED
			NRBGNAV AED
			CPNCONST AED

File Name	Type	Data Files Referenced

Group 2 (AED) (continued)

	File Name	Type	Data Files Referenced
3.	CPNOWN AED	AED data declarations	(None)
4.	CNDDATA AED	AED data declarations	(None)
5.	NRBMAST AED	AED data declarations	(None)
6.	NRBAUX AED	AED data declarations	(None)
7.	NRBGNAV AED	AED data declarations	(None)
8.	CPNCONST AED	AED data declarations	(None)

The two programs (N.KFNUP and C.PNHIT) were chosen to illustrate different types of compiler usage, as can be seen from a quick perusal of the listings. In particular, the following features may be observed:

Program	Features
N.KFNUP	1. Short program
	2. Heavy use of "FOR-loops"
	3. Heavy use of "IF-statements"
C.PNHIT	1. Long program, including several remotely inserted COMPOOL/INSERT files of data declarations.
	2. Heavy use of switches and GOTO's.
	3. Heavy use of logical bit-operations with masks (AND, OR, XOR, etc.)

All files have been carefully edited to prepare AED and J3B versions of the test programs which are as identical as possible. For example, the phrase "ELSE BEGIN" now occurs on the same line in both versions, whereas the original author of the J3B program had chosen to insert the two words on separate lines, and the original author of the AED version used a single line. Also, the same remarks and comments occur in both versions in the same format.

2. Programs and Data Files for J3B Version of Tests

```
FILE      NKKFNUP

START
    COMPOOL(NRBGNAV) !
    "                                                              "
    "            POSITION UPDATE PROGRAM                           "
    "                                                              "
    DEF PROC N.KFPNUP(!NKFPMATX,NKFKGANX) !
        BEGIN
    "                                                              "
    "            DEFINITION OF FORMAL PARAMETERS                   "
    "                                                              "
            ARRAY NKFPMATX(189) F ! "COVARIANCE MATRIX ARRAY"
            ARRAY NKFKGANX(38) F !   "KALMAN GAIN ARRAY"
    "                                                              "
  ` "        ENTRY TO POSITION UPDATE PROGRAM                      "
    "                                                              "
        FOR NKFI (0 BY 1 WHILE NKFI <= 1) ! "X AND Y UPDATE"
            BEGIN
            NKFS = NKFCOLMN(NKFI)+NKFI !
            NKFKGD = NKFPMATX(NKFS)+NKFRMATX !
            FOR NKFJ (NKFI BY 1 WHILE NKFJ <= NKFSTANO) !
                BEGIN            "CALCULATE KALMAN GAIN"
                NKFS = NKFCOLMN(NKFJ)+NKFI!
                NKFKGANX(NKFI+NKFJ) = NKFPMATX(NKFS)/NKFKGD !
                END !
              IF NKFI = 1 !       "KALMAN GAIN FOR Y UPDATE"
                  NKFKGANX(1+0) = NKFPMATX(1)/NKFKGD !
            FOR NKFJ (NKFI+1 BY 1 WHILE NKFJ <= NKFSTANO) !
                BEGIN            "UPDATE COVARIANCE MATRIX"
                NKFS = NKFCOLMN(NKFI)+NKFJ !
                FOR NKFH (NKFJ BY 1 WHILE NKFH <= NKFSTANO) !
                    BEGIN
                    NKFT = NKFCOLMN(NKFJ)+NKFH !
                    NKFPMATX(NKFT) = NKFPMATX(NKFT)-NKFKGANX
                                    (NKFI+NKFH)*NKFPMATX(NKFS) !
                    END !
                END !
            FOR NKFJ (NKFI BY 1 WHILE NKFJ <= NKFSTANO) !
                BEGIN            "UPDATE COL 1 OR 2 OF"
                                 "COVARIANCE MAT"
                NKFS = NKFCOLMN(NKFI)+NKFJ !
                NKFPMATX(NKFS) = NKFRMATX*NKFKGANX(NKFI+NKFJ) !
                END !
              IF NKFI <> 0 !     "X UPDATE"
                BEGIN            "UPDATE COL 1 FOR Y UPDATE"
                NKFPMATX(0) = NKFPMATX(0)-NKFPMATX(1)*NKFKGANX
                              (1+0) !
                FOR NKFJ (2 BY 1 WHILE NKFJ <= NKFSTANO) !
                    BEGIN
                    NKFPMATX(NKFJ) = NKFPMATX(NKFJ)-NKFKGANX
                                     (1+NKFJ)*NKFPMATX(1) !
                    END !
                NKFPMATX(1) = NKFKGANX(1+0)*NKFRMATX !
                END !
            END !
        RETURN !
        END !
TERM
```

```
FILE      C.PNHIT


START
      COMPOOL(CPNOWN,
              CNDDATA,
              NRBMAST,
              NRBAUX,
              NRBGNAV,
              CPNCONST);
      REF  PROC C.PNPACK   (S "FIRST BYTE",S "NO. BYTES") B 32 ;
                           "PACKED"
      REF  PROC C.PFXRCD   (S "FIXED PT VARBL",S "NO. BYTES") ;
      DEF PROC C.PNHIT() ;
      BEGIN
            "DEFINE THE SWITCH NUMBER BRANCH"
            SWITCH CPNBRNCH=(
                " 0:"HITCOUNT,
                " 1:"SLU,
                " 2:"SLU,
                " 3:"SLU,
                " 4:"SLU,
                " 5:"SLU,
                " 6:"HITCOUNT,
                " 7:"FLYTO,
                " 8:"ALTSELCT,
                " 9:"DSPSELCT,
                "10:"STRMODE,
                "11:"FASTFWD,
                "12:"FORWARD,
                "13:"REVERSE,
                "14:"FASTREV,
                "15:"HITCOUNT,
                "16:"ACCEPT,
                "17:"REJECT,
                "18:"ALTELEV,
                "19:"AUTOMAN,
                "20:"LANDSEA,
                "21:"INS1SEL,
                "22:"INS1DIT,
                "23:"INS2SEL,
                "24:"INS2DIT,
                "25:"DEADSLCT,
                "26:"ADDRDR,
                "27:"HITCOUNT,
                "28:"HITCOUNT,
                "29:"INS1ENBL,
                "30:"INS2ENBL,
                "31:"HITCOUNT);
            "DEFINE ALT/ELEV ALT CAL MODE BRANCH"
            SWITCH CPNAEBN=(
                " 0:"ACM0AND5,
                " 1:"ACM1AND2,
                " 2:"ACM1AND2,
                " 3:"ACM3,
                " 4:"ACM4,
                " 5:"ACM0AND5);
            "ARE THERE NO MORE HITS"
```

310

```
FILE      C.PNHIT


ANYHITS : IF CPNHIT(0) = 0    :
                  RETURN :
            ELSE BEGIN "OBTAIN THE NEXT HIT"
                  CPNTI1=CPNHIT(CPNHIT(0)) :
                  CPNSWCH=CPNTI1 :
                  "IS THE NEW STATUS OFF"
                  IF CPNTI1 < 0 :
                      "IS THE HIT A POSITION SWITCH"
                      IF CPNTI1 < -28
                      OR (CPNTI1 >= -14 AND CPNTI1 <= -11)
                      OR CPNTI1 >= -5 :
                          CPNTI1=-CPNTI1 :
                      ELSE "IF NOT, IGNORE"
                          GOTO HITCOUNT :
                  "NOW BRANCH ON THE SWITCH NUMBER"
                  GOTO CPNBRNCH(CPNTI1) :
FLYTO :           "IF THE FLY TO REQUEST IS AN OAP...  "
                  IF CMXTYPE(CPSXPTR) = 6 :
                      GOTO HITCOUNT : "IGNORE THE REQUEST"
                  ELSE BEGIN "TURN ON THE FLY TO LIGHT"
                          CPNOUT13=CPNOUT13 OR KMASK19 :
                          CPNFLYTO = 1 : "SET FLYTO FLAG"
                          GOTO ZEROCLOK :
                      END :
ALTSELCT :        "REVERSE HSL AND HR LIGHT STATUS"
                  CPNOUT13=CPNOUT13  XOR  X'00006000' :
                  CPNHSL=CPNOUT13 AND X'00006000' : "SET HSL/HR FLAG"
                  GOTO ZEROCLOK :
SLU :             "SET RESET INDICATED SLU STATUS "
                  CPNSLU(CPNTI1)=CPNSWCH :
                  GOTO HITCOUNT :
DSPSELCT :        "CYCLE THE LIGHT TEMPLATE"
                  CPNDIS=SHIFTL(CPNDIS,1) :
                  IF CPNDIS AND KMASK15 :
                      CPNDIS = KMASK19 :
                  IF CPNDIS AND KMASK17 : "IS THE NEW MODE NV"
                      "STOP THE BLINK TIMER"
                      CPNDCLOK = 0 :
                  ELSE "RE-START THE BLINK TIMER"
                      CPNDCLOK = 1 :
                  CPNOUT14 = (CPNOUT14 AND X'FFFF0FFF') OR CPNDIS :
                  "LIGHT APPROPRIATE SEGMENT AND SET THE"
                  "MODE FLAG"
                  CPNDSLR = SHIFTR(CPNDIS,13) :
                  GOTO ZEROCLOK :
STRMODE :         "REVERSE STEER MODE LIGHTS AND FLAG"
                  CPNOUT13 = CPNOUT13 XOR X'00000C00' :
                  CPNSTRB = SHIFTR(CPNOUT13 AND X'00000C00',11)       :
                  GOTO ZEROCLOK :
FASTFWD :         "SET FWD/REV REQUEST FLAG"
                  CPNFWDRV = 10 :
                  GOTO FWDRVOFF :
FORWARD :         CPNFWDRV = 1 :
                  GOTO FWDRVOFF :
REVERSE :         CPNFWDRV = -1 :
```

311

```
FILE      C.PNHIT


                    GOTO FWDRVOFF $
FASTREV :           CPNFWDRV = -10 $
FWDRVOFF :          "IF THE NEW STATUS IS OFF, OR IF THE
                     XHAIR MODE IS ID..."
                    IF CPNSWCH < 0 OR CPSID <> 0 $
                        CPNFWDRV = 0 $ "REMOVE FWD/REV REQUEST FLAG"
                    GOTO ZEROCLOK $
ACCEPT :            "IGNORE UNLESS ALTITUDE CALIBRATION"
                    "MODE IS 4"
                    IF CPNACM = 4 $
                        BEGIN
                            "SET ALT CAL COMPUTE FLAG,AND PREPARE
                            TO RECEIVE KALMAN VERDICTS"
                            CPNACC = CPNACS $
                            CPNACM = 5 $
                            NKFARJTM = 0 $
                            NKFARJTA = 0 $
                        END $
                    GOTO HITCOUNT $
REJECT :            "IGNORE IF ALT CAL MODE IS 5, OTHERWISE"
                    "END ALT CAL"
                    IF CPNACM = 5 $
                    GOTO HITCOUNT $
                    ELSE BEGIN
                            CPNOUT14 = CPNOUT14 AND X'FFF9FFFF' $
                            CPNOUT9 = CPNOUT9 OR X'00FFFFFF' $
                            CPNACM = 0 $
                            CPNHAC = 0 $
                        END $
                    GOTO HITCOUNT $
ALTELEV :           "BRANCH ON ALT CAL MODE"
                    GOTO CPNAEBN (CPNACM) $
ACM1AND2:           "TURN OUT ELEV LIGHT,PULL HIGH ALT CAL"
                    "FLAG DOWN"
                    CPNOUT14 = CPNOUT14 AND KRMASK13 $
ACM4 :              CPNHAC = 0 $
                    CPNACM = 3 $
                    IF NCRALTOK = 0$ "ARE THE RADAR ALTIMETERS DOWN"
                        "IS THE FLR UNAVAILABLE"
                        IF CPFLRON = 0 $
                            BEGIN "TERMINATE ALT CAL"
                                CPNOUT9 = CPNOUT9 OR X'00FFFFFF' $
                                CPNOUT14 = CPNOUT14 AND KRMASK14 $
                                CPNACM = 0 $
                            END $
                        ELSE BEGIN "SET HIGH ALT CAL FLAG AND ZERO
                                    THE DELTA BUFFER"
                            CPNHAC = 1 $
                            CPNDELTA = 0 $
                            END $
                    GOTO HITCOUNT $
ACM3 :              "RECORD WHICH NAV SYSTEM WAS USED FOR"
                    "ALT CAL"
                    CPNACS = CPNNMS $
                    CPNOUT14 = CPNOUT14 OR KMASK14 $ "ADVANCE TO MODE 4"




                              312
```

```
FILE      C.PNHIT


                    CPNACM = 4 ;
ACMOAND5 ;          GOTO HITCOUNT ;
AUTOMAN ;           "REVERSE LIGHTS AND MODE. INITIALIZE"
                    "SYSTEM REQUEST"
                    CPNOUT14=CPNOUT14 XOR X'0C000000' ;
                    CPNMANB=SHIFTR(CPNOUT14 AND X'0C000000',27) ;
                    CPNNMSB=NCPRNV ;
                    GOTO HITCOUNT ;
LANDSEA ;           "REVERSE LAND/SEA MODE"
                    CPNLANDB = CPNLANDB XOR X'60000000' ;
                    GOTO HITCOUNT ;
INS1SEL ;           "REQUEST INS1"
                    CPNNMS = 4 ;
                    GOTO HITCOUNT ;
INS1DIT ;           "IF IN MANUAL MODE, CYCLE LIGHT AND
                    TENTATIVE REQUEST. RESET DISPLAY CLOCK
                    TO ALLOW ONE SECOND MODE SET DELAY"
                    IF CPNMAN = 0 ;
                         BEGIN
                              CPNMDIT=SHIFTL(CPNMDIT,1);
                              IF CPNMDIT AND KMASK27 ;
                                   CPNMDIT=KMASK31 ;
                              CPNOUT4=CPNOUT4 AND X'FFFFF8FF'
                                      OR SHIFTL(CPNMDIT,8) ;
                              CPNCLOCK=13 ;
                         END ;
                    GOTO HITCOUNT ;
INS2SEL ;           CPNNMS = 2 ;
                    GOTO HITCOUNT ;
INS2DIT ;           IF CPNMAN=0 ;
                         BEGIN
                              CPNADIT = SHIFTL(CPNADIT,1)       ;
                              IF CPNADIT AND KMASK27 ;
                                   CPNADIT = KMASK31 ;
                              CPNOUT0 = CPNOUT0 AND X'F8FFFFFF'
                                      OR SHIFTL(CPNADIT,24) ;
                              CPNCLOCK = 13 ;
                         END ;
                    GOTO HITCOUNT ;
DEADSLCT ;          "SET REQUEST FLAG TO DEAD RECKON"
                    CPNNMS = 1 ;
                    GOTO HITCOUNT ;
ADDRDR ;            "REVERSE DR/ADDR REQUEST FLAG"
                    CPNDDSB = CPNDDSB XOR X'00000003'        ;
                    GOTO HITCOUNT ;
INS1ENBL ;          "SET INS ENABLE FLAG TO APPROPRIATE"
                    "VALUE"
                    CPNINS1 = CPNSWCH ;
                    GOTO HITCOUNT ;
INS2ENBL ;          CPNINS2 = CPNSWCH ;
                    GOTO HITCOUNT ;
                    "ALL BRANCHES RETURN HERE"
ZEROCLOK ;          " SET THE DISPLAY CLOCK TO ZERO "
                    CPNCLOCK = 0 ;
HITCOUNT ;          CPNHIT(0) =  CPNHIT(0) - 1       ;
```

313

FILE   CNDDATA


```
          ARRAY CPNHIT (7, S 31 $ "THE NAV PROGRAM HIT TABLE"
          ITEM CPSID        S 31$      "ID MODE FLAG                  "
          ITEM CPFLRON      S 31$      "FORWARD LOOKING RADAR ON      "
          ITEM CMALTCAL     S 31$      "NEXT DEST. AN ACT CAL         "
          ITEM CPKALTCL     S 31$      "IKB ALT CAL REQUEST           "
          ITEM CPKTNELV     F    $      "IKB TERRAIN ELEVATION         "
          ITEM CPSNUIP      S 31$      "POSITION UPDATE IN PROGRESS   "
          ITEM CMCSTP       S 31$      "STEER POINT FLAG              "
          ITEM CMCSPID      S 31$      "STEER POINT TYPE              "
          ITEM CMCSPSN      S 31 $ "STEER PT. SEQ. NO."
          ITEM CAFFDTF      B 32$      "FLT OR ENGD/AUTO TF FLAG      "
          ITEM CAFSMODE     B 32$      "FLIGHT DIRECTOR MODE          "
          ARRAY CPHRD(7)    B 32$      "BCD ARRAY                     "
          ARRAY CMXLAT      (7) F    $
          ARRAY CMXLONG     (7) F    $
          ARRAY CMXELEV     (7) F    $
          ARRAY CMXQUAL     (7) S 31$
          ARRAY CMXSQNO     (7) S 31$
          ARRAY CMXTYPE     (7) S 31$
          ITEM CPSXPTR      S 31$      "POINTER INTO CMXHAIR          "
          ITEM NCRALTOK     S 31$      "RADAR ALTITUDE OK  "
          ITEM NCRALT       F    $      "RADAR ALTITUDE.FEET"
          ITEM NCALTP       F    $      "PRIME ALTITUDE.FT."
          ITEM NSITG        F    $      "TIME TO DESTINATION.SEC"
          ITEM NSRNG        F    $      "RANGE TO DESTINATION"
          ITEM NLAT         F    $      "DISPLAY LAT.RADIANS"
          ITEM NLONG        F    $      "DISPLAY LONG. RADIANS"
          ITEM NVGND        F    $      "DISPLAY GND SPEED.FT/SEC"
          ITEM NHGT         F    $      "GROUND TRACK.RADIANS"
          ITEM NTHEAD       F    $      "TRUE HEADING.RADIANS"
          ITEM NWHEAD       F    $      "WIND HEADING.RADIANS"
          ITEM NWIND        F    $      "WIND SPEED. FT/SEC  "
          ITEM NHDISP       F    $      "HEIGHT ABOVE SEA LEVEL.FT"
          ITEM NDRMODE      S 31$      "DEAD RECKON MODE "
          ITEM NCDOPCUT     S 31$      "DOPPLER CUT-OUT FLAG "
          ITEM NCAALCPT     S 31$      "INS2 UP"
          ITEM NCMALCPT     S 31$      "INS1 UP"
          ITEM NVFSTRT      S 31$      "DEAD RECKON RESTART FLAG"
          ITEM NKFARJTM     S 31$      "INS1 ALT CAL REJECT FLAG"
          ITEM NKFARJTA     S 31$      "INS2 ALT CAL REJECT FLAG"
          ITEM NCAVAILM     B 32$      "INS1 DIT AVAILABLE FLAG"
          ITEM NCAVAILA     B 32$      "INS2 DIT AVAILABLE FLAG "
          ITEM NCMDIT       B 32$
          ITEM NCADIT       B 32$
          ITEM NCPRNV       B 16$      "PRIME SYSTEM FLAG"

                  GOTO ANYHITS $
             END $
          END $ "OF HIT PROCEDURE DEFINITION"
   TERM
```

FILE   CPNOWN


```
          ITEM CPNMTIME S 31 : "MISSION TIME IN MAJOR FRAMES"
          ITEM CPNTIMA  H 32 : "BIT NAME FOR MISSION TIME "
          ITEM CPNCLOCK S 31 : "DISPLAY CLOCK--A MOD 16 COUNTER"
          ITEM CPNCLOKR H 32 : "BIT NAME FOR THE DISPLAY CLOCK"
          ITEM CPNHSL   B 32 : "HR/HSL FLAG"
          ITEM CPNDCLOK S 31 : "DISPLAY SELECT BLINK TIMER"
          ITEM CPNDIS   H 32 : "DISPLAY SELECT TEMPLATE"
          ITEM CPNACM   S 31 : "ALT CAL MODE "
          ITEM CPNMDIT  H 32 : "INS1 DIT REQUEST FLAG"
          ITEM CPNADIT  H 32 : "INS2 DIT REQUEST FLAG"
          ITEM CPNTI1   S 31 : "TEMP1"
          ITEM CPNTB1   B 32 : "TEMP1"
          ITEM CPNTF1   F    : "TEMP1"
          ITEM CPNTI2   S 31 : "TEMP2"
          ITEM CPNTB2   B 32 : "TEMP2"
          ITEM CPNTF2   F    : "TEMP2"
       "VARIABLES LOCAL TO C.PNHIT "
          ITEM CPNSWCH    S 31:"ACTIVE HIT SWITCH NO./STATUS"
          ITEM CPNACS     S 31:"NAV SYSTEM USED FOR ALT CAL "
       "DATA LOCAL TO C.PNDISO"
          ITEM CPNPPDOK   S 31:"PRESENT POSITION DATA OK FLAG       "
          ITEM CPNSELNO   S 31:"SELECTED PT. SEQ NO.                "
          ITEM CPNUPRJ    H 32:"UPDATE REJECT LIGHT TEMPLATE        "
          ITEM CPNMAG1    S 31:"SELECTED POINT MAGWHEEL TIMER       "
          ITEM CPNSLTYP   B 32:"SELECTED POINT TYPE                 "
          ITEM CPNTNELV   F   :"TERRAIN ELEV..FEET                  "
          ITEM CPNSELON   F   :"SELECTED POINT LONG .RADIANS        "
          ITEM CPNSELAT   F   :"SELECTED POINT LAT  .RADIANS        "
          ITEM CPNSLELV   F   :"SELECTED POINT ELEV.FEET            "
          ITEM CPNSTRNO   S 31:"STEER POINT SEQ NO.                 "
          ITEM CPNMAG2    S 31:"STEER POINT MAGWHEEL TIMER          "
          ITEM CPNSTTYP   S 31:"STEER POINT TYPE                    "
       ARRAY CPNSLPOS(1) F:"SELECTED POINT POSITION              "
       ARRAY CPNTAR   (1) F:"TIME AND RANGE TO SELECTED PT        "
       ARRAY CPNMCODE(9) B 32:
                  "THE FOLLOWING ARE B 32 NAMES FOR THE ABOVE
                   OUTPUTS TO THE PANEL"
          ITEM CPNOUT0       B 32 :
          ITEM CPNOUT1       B 32 :
          ITEM CPNOUT2       B 32 :
          ITEM CPNOUT3       B 32 :
          ITEM CPNOUT4       B 32 :
          ITEM CPNOUT5       B 32 :
          ITEM CPNOUT6       B 32 :
          ITEM CPNOUT7       B 32 :
          ITEM CPNOUT8       B 32 :
          ITEM CPNOUT9       B 32 :
          ITEM CPNOUT10      B 32 :
          ITEM CPNOUT11      B 32 :
          ITEM CPNOUT12      B 32 :
          ITEM CPNOUT13      B 32 :
          ITEM CPNOUT14      B 32 :
          ITEM CPNOUT15      B 32 : " "
       "THE FOLLOWING IS AN ARRAY NAME FOR THE
       OUTPUT TO THE PANELS"
```

FILE   CPNOWN

```
        ARRAY CPNOUTA(15)    B 32 ;  " "
        "THE FOLLOWING VARIBLES ARE REFERENCED BY OTHER PROGRAMS"
        ITEM CPNSTEER   S 31;      "STEER MODE 0--TRK 1--DIR      "
        ITEM CPNSTRR    B 32;      "BIT NAME FOR STEER MODE       "
        ITEM CPNFWDRV   S 31;      "FORWARD REVERSE REQUEST       "
        ITEM CPNROLL    S 31;      "FORWARD REVERSE COMMAND       "
        ITEM CPNACC     S 31;      "ALT CAL COMPUTE               "
        ITEM CPNHAC     S 31;      "HIGH ALT CAL REQUEST          "
        ITEM CPNMAN     S 31;      "AUTO/MAN MODE FLAG            "
        ITEM CPNMANR    B 32;      "BIT NAME FOR  AUTO/MAN FLAG   "
        ITEM CPNLAND    S 31;      "LAND/SEA FLAG                 "
        ITEM CPNLANDR   B 32;      "BIT NAME FOR LAND/SEA FLAG    "
        ITEM CPNNMS     S 31;      "NAV SYSTEM SELECT             "
        ITEM CPNNMSR    B 32;      "BIT NAME FOR NMS              "
     \  ITEM CPNDDS     S 31;      "DR/ADDR REQUEST               "
        ITEM CPNDDSR    B 32;      "BIT NAME FOR DR/ADDR          "
        ITEM CPNINS1    S 31;      "INS1 ENABLE/DISABLE           "
        ITEM CPNINS2    S 31;      "INS2 ENABLE/DISABLE           "
        ITEM CPNSLUA    S 31;      "AFT                           "
        ITEM CPNSLUL    S 31;      "LEFT PYLON                    "
        ITEM CPNSLUR    S 31;      "RIGHT PYLON        SLU ENABLE "
        ITEM CPNSLUI    S 31;      "INTRMD                        "
        ITEM CPNSLUF    S 31;      "FORWARD                       "
        ITEM CPNDELTA   F   ;      "ALT CAL DELTA BUFFER          "
        ARRAY CPNSLU(5) S 31 ;     "THIS IS THE SLU STATUS TABLE.
                                    IT IS 'OVERLAYED' WITH THE SLU
                                    STATUS WORDS ABOVE"
        ITEM CPNFLYTO   S 31;      "FLY TO FLAG                   "
        ITEM CPNDSL     S 31;      "DISPALY SELECT FLAG           "
        ITEM CPNDSLR    B 32;      "BIT NAME FOR CPNDS2           "
        "OVERLAY DECLARATIONS FOR CPNOWN"
        OVERLAY    CPNSTEER   =CPNSTRR  ;
        OVERLAY    CPNMAN     =CPNMANR  ;
        OVERLAY    CPNLAND    =CPNLANDR ;
        OVERLAY    CPNNMS     =CPNNMSR  ;
        OVERLAY    CPNDDS     =CPNDDSR  ;
        OVERLAY    CPNSLU     =CPNSLUA,
                              CPNSLUL,
                              CPNSLUR,
                              CPNSLUI,
                              CPNSLUF ;
        OVERLAY    CPNMTIME   =CPNTIMR  ;
        OVERLAY    CPNCLOCK   =CPNCLOKB;
        OVERLAY    CPNTI1     =CPNTF1   =CPNTB1  ;
        OVERLAY    CPNTI2     =CPNTB2   =CPNTF2 ;
        OVERLAY    CPNDSL     =CPNDSLR  ;
```

316

FILE   NRBMAST

```
        ITEM NILATM       F $   "LATITUDE(RADIANS)"
        ITEM NILONM       F $   "LONGITUDE(RADIANS)"
        ITEM NIALTM       F $   "ALTITUDE(FEET)"
        ITEM NIVNM        F $   "VELOCITY NORTH(FT/SEC)"
        ITEM NIVEM        F $   "VELOCITY EAST(FT/SEC)"
        ITEM NIHDOTM      F $   "ALTITUDE RATE(FT/SEC)"
        ITEM NIPSITM      F $   "TRUE HEADING(RADIANS)"
        ITEM NIPSIMM      F $   "MAGNETIC HEADING(RADIANS)"
        ITEM NIGRAVM      F $   "LOCAL GRAVITY(FT/SEC**2)"
        ITEM NCMPITC      F $   "PITCH(RADIANS)"
        ITEM NCMROLL      F $   "ROLL(RADIANS)"
        ITEM NCMYAW       F $   "YAW(RADIANS)"
        ITEM NCMPDOT      F $   "P CH RATE(RAD/SEC)"
      , ITEM NCMRDOT      F $   "ROLL RATE(RAD/SEC)"
        ITEM NCMYDOT      F $   "YAW RATE(RAD/SEC)"
        "                                                    "
        " NAVIGATION DATA COMMON TO MASTER AND AUXILIARY     "
        "                                                    "
        ITEM NIALFAM      F $   "WANDER ANGLE(RADIANS)"
        ITEM NIVXM        F $   "X PLATFORM VELOCITY(FT/SEC)"
        ITEM NIVYM        F $   "Y PLATFORM VELOCITY(FT/SEC)"
        ITEM NIVZM        F $   "Z PLATFORM VELOCITY(FT/SEC)"
        ITEM NCMAX        F $   "X ACCELEROMETER(FT/SEC/FRAME)"
        ITEM NCMAY        F $   "Y ACCELEROMETER(FT/SEC/FRAME)"
        ITEM NCMAZ        F $   "Z ACCELEROMETER(FT/SEC/FRAME)"
        ITEM NIDVAXM      F $   "DELTA X VELOCITY(FT/SEC/FRAME)"
        ITEM NIDVAYM      F $   "DELTA Y VELOCITY(FT/SEC/FRAME)"
        ITEM NIDVAZM      F $   "DELTA Z VELOCITY(FT/SEC/FRAME)"
        ITEM NIC12M       F $   "DIRECTION COSINE C(1,2)"
        ITEM NIC22M       F $   "DIRECTION COSINE C(2,2)"
        ITEM NIC32M       F $   "DIRECTION COSINE C(3,2)"
        ITEM NIC13M       F $   "DIRECTION COSINE C(1,3)"
        ITEM NIC23M       F $   "DIRECTION COSINE C(2,3)"
        ITEM NIC33M       F $   "DIRECTION COSINE C(3,3)"
        ITEM NIC12DM      F $   "DERIVATIVE OF C(1,2)"
        ITEM NIC22DM      F $   "DERIVATIVE OF C(2,2)"
        ITEM NIC32DM      F $   "DERIVATIVE OF C(3,2)"
        ITEM NIC13DM      F $   "DERIVATIVE OF C(1,3)"
        ITEM NIC23DM      F $   "DERIVATIVE OF C(2,3)"
        ITEM NIC33DM      F $   "DERIVATIVE OF C(3,3)"
        ITEM NIOMEGXM     F $   "EARTH RATE ABOUT X(RAD/SEC)"
        ITEM NIOMEGYM     F $   "EARTH RATE ABOUT Y(RAD/SEC)"
        ITEM NIOMEGZM     F $   "EARTH RATE ABOUT Z(RAD/SEC)"
        ITEM NIRHOXM      F $   "CRAFT RATE ABOUT X(RAD/SEC)"
        ITEM NIRHOYM      F $   "CRAFT RATE ABOUT Y(RAD/SEC)"
        ITEM NIPRXM       F $   "PLATFORM RATE ABOUT X(RAD/SEC)"
        ITEM NIPRYM       F $   "PLATFORM RATE ABOUT Y(RAD/SEC)"
        ITEM NIAXCM       F $   "CORRECTED X ACCEL(FT/SEC/FRAME)"
        ITEM NIAYCM       F $   "CORRECTED Y ACCEL(FT/SEC/FRAME)"
        ITEM NIAZCM       F $   "CORRECTED Z ACCEL(FT/SEC/FRAME)"
        ITEM NIDAXCM      F $   "X ACCEL NOISE EST(FT/SEC/FRAME)"
        ITEM NIDAYCM      F $   "Y ACCEL NOISE EST(FT/SEC/FRAME)"
        ITEM NIDAZCM      F $   "Z ACCEL BIAS EST(FT/SEC/FRAME)"
        "                                                    "
        "            KALMAN FILTER DATA                      "
```

317

FILE   NRBMAST

```
                  "                                                  "
     ITEM NKFPRJTM        F $   "POSITION FIX REJECT FLAG"
     ARRAY NAINSM( 50)    F $
     ITEM NKP0101M        F $   "COVARIANCE ELEMENT P(1,1)"
     ITEM NKP0201M        F $   "COVARIANCE ELEMENT P(2,1)"
     ARRAY NKFPMATM(189)F $   "INS 1 COVARIANCE MATRIX"
     ARRAY NKXHATM(18)    F $   "INS 1 STATE VECTOR"
     ARRAY NKYMSVM(4)     F $   "INS 1 MEASUREMENT VECTOR"
```

318

FILE   NRRAUX

```
       ITEM NILATA        F $    "LATITUDE(RADIANS)"
       ITEM NILONA        F $    "LONGITUDE(RADIANS)"
       ITEM NIALTA        F $    "ALTITUDE(FEET)"
       ITEM NIVNA         F $    "VELOCITY NORTH(FT/SEC)"
       ITEM NIVEA         F $    "VELOCITY EAST(FT/SEC)"
       ITEM NIHDOTA       F $    "ALTITUDE RATE(FT/SEC)"
       ITEM NIPSITA       F $    "TRUE HEADING(RADIANS)"
       ITEM NIPSIMA       F $    "MAGNETIC HEADING(RADIANS)"
       ITEM NIGRAVA       F $    "LOCAL GRAVITY(FT/SEC** )"
       ITEM NCAPITCH      F $    "PITCH(RADIANS)"
       ITEM NCAROLL       F $    "ROLL(RADIANS)"
       ITEM NCAYAW        F $    "YAW(RADIANS)"
       ITEM NCAPDOT       F $    "PITCH RATE(RAD/SEC)"
       ITEM NCARDOT       F $    "ROLL RATE(RAD/SEC)"
       ITEM NCAYDOT       F $    "YAW RATE(RAD/SEC)"
       "                                                        "
       " NAVIGATION DATA COMMON TO MASTER AND AUXILIARY         "
       "                                                        "
       ITEM NIALFAA       F $    "WANDER ANGLE(RADIANS)"
       ITEM NIVXA         F $    "X PLATFORM VELOCITY(FT/SEC)"
       ITEM NIVYA         F $    "Y PLATFORM VELOCITY(FT/SEC)"
       ITEM NIVZA         F $    "Z PLATFORM VELOCITY(FT/SEC)"
       ITEM NCAAX         F $    "X ACCELEROMETER(FT/SEC/FRAME)"
       ITEM NCAAY         F $    "Y ACCELEROMETER(FT/SEC/FRAME)"
       ITEM NCAAZ         F $    "Z ACCELEROMETER(FT/SEC/FRAME)"
       ITEM NIDVAXA       F $    "DELTA X VELOCITY(FT/SEC/FRAME)"
       ITEM NIDVAYA       F $    "DELTA Y VELOCITY(FT/SEC/FRAME)"
       ITEM NIDVAZA       F $    "DELTA Z VELOCITY(FT/SEC/FRAME)"
       ITEM NIC12A        F $    "DIRECTION COSINE C(1,2)"
       ITEM NIC22A        F $    "DIRECTION COSINE C(2,2)"
       ITEM NIC32A        F $    "DIRECTION COSINE C(3,2)"
       ITEM NIC13A        F $    "DIRECTION COSINE C(1,3)"
       ITEM NIC23A        F $    "DIRECTION COSINE C(2,3)"
       ITEM NIC33A        F $    "DIRECTION COSINE C(3,3)"
       ITEM NIC12DA       F $    "DERIVATIVE OF C(1,2)"
       ITEM NIC22DA       F $    "DERIVATIVE OF C(2,2)"
       ITEM NIC32DA       F $    "DERIVATIVE OF C(3,2)"
       ITEM NIC13DA       F $    "DERIVATIVE OF C(1,3)"
       ITEM NIC23DA       F $    "DERIVATIVE OF C(2,3)"
       ITEM NIC33DA       F $    "DERIVATIVE OF C(3,3)"
       ITEM NIOMEGXA      F $    "EARTH RATE ABOUT X(RAD/SEC)"
       ITEM NIOMEGYA      F $    "EARTH RATE ABOUT Y(RAD/SEC)"
       ITEM NIOMEGZA      F $    "EARTH RATE ABOUT Z(RAD/SEC)"
       ITEM NIRHOXA       F $    "CRAFT RATE ABOUT X(RAD/SEC)"
       ITEM NIRHOYA       F $    "CRAFT RATE ABOUT Y(RAD/SEC)"
       ITEM NIPRXA        F $    "PLATFORM RATE ABOUT X(RAD/SEC)"
       ITEM NIPRYA        F $    "PLATFORM RATE ABOUT Y(RAD/SEC)"
       ITEM NIAXCA        F $    "CORRECTED X ACCEL(FT/SEC/FRAME)"
       ITEM NIAYCA        F $    "CORRECTED Y ACCEL(FT/SEC/FRAME)"
       ITEM NIAZCA        F $    "CORRECTED Z ACCEL(FT/SEC/FRAME)"
       ITEM NIDAXCA       F $    "X ACCEL NOISE EST(FT/SEC/FRAME)"
       ITEM NIDAYCA       F $    "Y ACCEL NOISE EST(FT/SEC/FRAME)"
       ITEM NIDAZCA       F $    "Z ACCEL BIAS EST(FT/SEC/FRAME)"
       "                                                        "
       "            KALMAN FILTER DATA                          "
```

FILE   NRBAUX

```
"                                                          "
ITEM NKFPRJTA         F .$   "POSITION FIX REJECT FLAG"
ARRAY NAINSA( 50)     F $
ITEM NKPO101A         F $    "COVARIANCE ELEMENT P(1,1)"
ITEM NKPO201A         F $    "COVARIANCE ELEMENT P(2,1)"
ARRAY NKFPMATA(189)F $       "COVARIANCE MATRIX ARRAY"
ARRAY NKXHATA(18)  F $       "STATE VECTOR "
ARRAY NKYMSVA(4)   F $       "MEASUREMENT VECTOR"
```

```
ITEM NKFTMP1 F $       "TEMPORARY STORAGE"
ITEM NKFTMP2 F $       "TEMPORARY STORAGE"
ITEM NKFTMP3 F $       "TEMPORARY STORAGE"
ITEM NKFNDSET S 31 $   "NAV TABLE DATA INDEX"
ITEM NCCPOVL B 32 $    "POSITION FIX TYPE FLAG"
ITEM NKFI S   31 $     "LOOP INDEX"
ITEM NKFJ S   31 $     "LOOP INDEX"
ITEM NKFH S   31 $     "LOOP INDEX"
ITEM NKFS S   31 $     "SUBSCRIPT"
ITEM NKFT S   31 $     "SUBSCRIPT"
ITEM NKFKGD F $        "DENOMINAT OF KALMAN GAIN"
ITEM NKFSTANO S 31 $   "ELEMENTS IN STATE VECTOR"
                       "LESS 1"
ITEM NKFRMATX F $      "POSITION MEASUREMENT"
                       "NOISE(FT**2)"
ITEM NKFAN F $         "NORTH COMPONENT OF POS"
                       "ERROR(FT)"
ITEM NKFAE F $         "EAST COMPONENT OF POS"
                       "ERROR(FT)"
ITEM NKFEX F $         "X TERM IN POS TEST"
ITEM NKFEY F $         "Y TERM IN POS TEST"
ITEM NCPRIME S 31 $    "PRIME DATA FLAG"
ITEM CSNPDN F $        "NORTH POS ERROR(FT)"
ITEM CSNPUE F $        "EAST POS ERROR(FT)"
ITEM NCLATP F $        "PRIME LATITUDE(RADIANS)"
ITEM NCLONP F $        "PRIME LONGITUDE(RADIANS)"
ITEM NLATC F $         "OVERFLY CHECKPOINT"
                       "LATITUDE(RAD)"
ITEM NLONC F $         "OVERFLY CHECKPOINT"
                       "LONGITUDE(RAD)"
ITEM CMFPQUAL S 31 $   "CHECKPOINT QUALITY INDEX"
ITEM NDALT     F $     "DEAD RECKONING ALTITUDE"
ARRAY NKFKGAIN(38)F $  "KALMAN GAIN MATRIX"
ARRAY NKFCOLMN( 18) S 31 $ "SYMMETRIC MATRIX POINTERS"
ARRAY KRPMATX ( 2)F $  "POSITION NOISE(FT**2)"
ARRAY NKINIT2 ( 18)F $ "INIT VALUE OF COV MATRIX"
ITEM KNINE F $         "POITION TEST CONST"
ITEM KERADIUS F $      "EARTH RAD AT EQUATOR(FT)"
ITEM KDELTAT   F $     "FRAME TIME(SEC)"
ITEM KGRAV0    F $     "GRAVITATIONAL CONSTANT"
ITEM KGRAV1    F $     "GRAVITATIONAL CONSTANT"
ITEM KGRAV2    F $     "GRAVITATIONAL CONSTANT"
ITEM KELPTCTY  F $     "EARTH ELLIPTICITY "
ITEM K2FORE    F $     "2*KELPTCTY/KERADIUS"
ITEM K1ORE     F $     "1/KERADIUS"
ITEM KEARTHR   F $     "EARTH RATE(RAD/SEC)"
ITEM KNK1      F $     "INERTIAL CONSTANT"
ITEM KNK2      F $     "INERTIAL CONSTANT"
ITEM KONE      F $     "ONE"
```

FILE   CPNCONST

```
            CONSTANT   CPNK1       F.= 57.29581      " OEGREES/RADIAN"
            CONSTANT   CPNK2       F=.5921      ;     "KNOTS/(FT/SEC) "
            CONSTANT   KMASK0      B 32 =     X'80000000'       ;
            CONSTANT   KMASK1      B 32 =     X'40000000'       ;
            CONSTANT   KMASK2      B 32 =     X'20000000'       ;
            CONSTANT   KMASK3      B 32 =     X'10000000'       ;
            CONSTANT   KMASK4      B 32 =     X'08000000'       ;
            CONSTANT   KMASK5      B 32 =     X'04000000'       ;
            CONSTANT   KMASK6      B 32 =     X'02000000'       ;
            CONSTANT   KMASK7      B 32 =     X'01000000'       ;
            CONSTANT   KMASKA      B 32 =     X'00800000'       ;
            CONSTANT   KMASK9      B 32 =     X'00400000'       ;
            CONSTANT   KMASK10     B 32 =     X'00200000'       ;
            CONSTANT   KMASK11     B 32 =     X'00100000'       ;
            CONSTANT   KMASK12     B 32 =     X'00080000'       ;
            CONSTANT   KMASK13     B 32 =     X'00040000'       ;
            CONSTANT   KMASK14     B 32 =     X'00020000'       ;
            CONSTANT   KMASK15     B 32 =     X'00010000'       ;
            CONSTANT   KMASK16     B 32 =     X'00008000'       ;
            CONSTANT   KMASK17     B 32 =     X'0C004000'       ;
            CONSTANT   KMASK18     B 32 =     X'00002000'       ;
            CONSTANT   KMASK19     B 32 =     X'00001000'       ;
            CONSTANT   KMASK20     B 32 =     X'00000800'       ;
            CONSTANT   KMASK21     B 32 =     X'00000400'       ;
            CONSTANT   KMASK22     B 32 =     X'00000200'       ;
            CONSTANT   KMASK23     B 32 =     X'00000100'       ;
            CONSTANT   KMASK24     B 32 =     X'00000080'       ;
            CONSTANT   KMASK25     B 32 =     X'00000040'       ;
            CONSTANT   KMASK26     B 32 =     X'00000020'       ;
            CONSTANT   KMASK27     B 32 =     X'00000010'       ;
            CONSTANT   KMASK28     B 32 =     X'00000008'       ;
            CONSTANT   KMASK29     B 32 =     X'00000004'       ;
            CONSTANT   KMASK30     B 32 =     X'00000002'       ;
            CONSTANT   KMASK31     B 32 =     X'00000001'       ;
            CONSTANT   KRMASK0     B 32 =     X'7FFFFFFF'       ;
            CONSTANT   KRMASK1     B 32 =     X'BFFFFFFF'       ;
            CONSTANT   KRMASK2     B 32 =     X'DFFFFFFF'       ;
            CONSTANT   KRMASK3     B 32 =     X'EFFFFFFF'       ;
            CONSTANT   KRMASK4     B 32 =     X'F7FFFFFF'       ;
            CONSTANT   KRMASK5     B 32 =     X'FBFFFFFF'       ;
            CONSTANT   KRMASK6     B 32 =     X'FDFFFFFF'       ;
            CONSTANT   KRMASK7     B 32 =     X'FEFFFFFF'       ;
            CONSTANT   KRMASK8     B 32 =     X'FF7FFFFF'       ;
            CONSTANT   KRMASK9     B 32 =     X'FFBFFFFF'       ;
            CONSTANT   KRMASK10    B 32 =     X'FFDFFFFF'       ;
            CONSTANT   KRMASK11    B 32 =     X'FFEFFFFF'       ;
            CONSTANT   KRMASK12    B 32 =     X'FFF7FFFF'       ;
            CONSTANT   KRMASK13    B 32 =     X'FFFBFFFF'       ;
            CONSTANT   KRMASK14    B 32 =     X'FFFDFFFF'       ;
            CONSTANT   KRMASK15    B 32 =     X'FFFEFFFF'       ;
            CONSTANT   KRMASK16    B 32 =     X'FFFF7FFF'       ;
            CONSTANT   KRMASK17    B 32 =     X'FFFFBFFF'       ;
            CONSTANT   KRMASK18    B 32 =     X'FFFFDFFF'       ;
            CONSTANT   KRMASK19    B 32 =     X'FFFFEFFF'       ;
            CONSTANT   KRMASK20    B 32 =     X'FFFFF7FF'       ;
```

FILE   CPNCONST

```
CONSTANT KRMASK21   B 32 =   X'FFFFFBFF'      ;
CONSTANT KRMASK22   B 32 =   X'FFFFFDFF'      ;
CONSTANT KRMASK23   B 32 =   X'FFFFFEFF'      ;
CONSTANT KRMASK24   B 32 =   X'FFFFFF7F'      ;
CONSTANT KRMASK25   B 32 =   X'FFFFFFBF'      ;
CONSTANT KRMASK26   B 32 =   X'FFFFFFDF'      ;
CONSTANT KRMASK27   B 32 =   X'FFFFFFEF'      ;
CONSTANT KRMASK28   B 32 =   X'FFFFFFF7'      ;
CONSTANT KRMASK29   B 32 =   X'FFFFFFFB'      ;
CONSTANT KRMASK30   B 32 =   X'FFFFFFFD'      ;
CONSTANT KRMASK31   B 32 =   X'FFFFFFFE'      ;
```

3. Programs and Data Files for AED Version of Tests

N.KFNUP AED

```
BEGIN
.INSERT NRHGNAV $
...                                                        //
...        POSITION UPDATE PROGRAM                         //
...                                                        //
DEFINE PROCEDURE N.KFNUP(NKFPMATX,NKFKGANX)
...                                                        //
...        DEFINITION OF FORMAL PARAMETERS                 //
...                                                        //
 WHERE REAL ARRAY NKFPMATX $   ... COVARIANCE MATRIX ARRAY //
REAL ARRAY NKFKGANX $          ... KALMAN GAIN ARRAY //
...                                                        //
...        ENTRY TO POSITION UPDATE PROGRAM                //
...                                                        //
 TORE BEGIN
     FOR NKFI = 0 STEP 1 WHILE NKFI <= 1 ... X AND Y UPDATE //
     DO   BEGIN
          NKFS = NKFCOLMN(NKFI)+NKFI $
          NKFKGD = NKFPMATX(NKFS)+NKFRMATX $
          FOR NKFJ = NKFI STEP 1 WHILE NKFJ <= NKFSTANO
          DO   BEGIN         ... CALCULATE KALMAN GAIN //
               NKFS = NKFCOLMN(NKFJ)+NKFI $
               NKFKGANX(NKFI+NKFJ) = NKFPMATX(NKFS)/NKFKGD $
               END $
          IF NKFI == 1        ... KALMAN GAIN FOR Y UPDATE //
          THEN NKFKGANX(1+0) = NKFPMATX(1)/NKFKGD $
          FOR NKFJ = NKFI+1 STEP 1 WHILE NKFJ <= NKFSTANO
          DO   BEGIN         ... UPDATE COVARIANCE MATRIX //
               NKFS = NKFCOLMN(NKFI)+NKFJ $
               FOR NKFH = NKFJ STEP 1 WHILE NKFH <= NKFSTANO
               DO   BEGIN
                    NKFT = NKFCOLMN(NKFJ)+NKFH $
                    NKFPMATX(NKFT) = NKFPMATX(NKFT)-NKFKGANX
                    (NKFI+NKFH)*NKFPMATX(NKFS) $
                    END $
               END $
          FOR NKFJ = NKFI STEP 1 WHILE NKFJ <= NKFSTANO
          DO   BEGIN         ... UPDATE COL 1 OR 2 OF
                                       COVARIANCE MAT //
               NKFS = NKFCOLMN(NKFI)+NKFJ $
               NKFPMATX(NKFS) = NKFRMATX*NKFKGANX(NKFI+NKFJ) $
               END $
          IF NKFI -= 0        ... X UPDATE //
          THEN BEGIN          ... UPDATE COL 1 FOR Y UPDATE //
               NKFPMATX(0) = NKFPMATX(0)-NKFPMATX(1)*NKFKGANX
               (1+0) $
               FOR NKFJ = 2 STEP 1 WHILE NKFJ <= NKFSTANO
               DO   BEGIN
                    NKFPMATX(NKFJ) = NKFPMATX(NKFJ)-NKFKGANX
                    (1+NKFJ)*NKFPMATX(1) $
                    END $
               NKFPMATX(1) = NKFKGANX(1+0)*NKFRMATX $
               END $
          END $
     GOTO RETURN $
     END $
END FINI
```

324

```
BEGIN
.INSERT CPNOWN  $
.INSERT CNDDATA  $
.INSERT NRBMAST  $
.INSERT NRBAUX  $
.INSERT NRBGNAV  $
.INSERT CPNCONST  $
INTEGER PROCEDURE C.PNPACK  $  ... (FIRST BYTE, NO. BYTES)
                                B 32 PACKED //
PROCEDURE C.PFXHCD  $        ... (FIXEDPTVARBL, NO. BYTES) //
DEFINE PROCEDURE C.PNHIT TOBE
        ... DEFINE THE SWITCH NUMBER BRANCH //
        BEGIN
        SWITCH CPNBRNCH =     ... DEFINE THE SWITCH NUMBER BRANCH //
         HITCOUNT,            ...  0 //
         SLU,                 ...  1 //
         SLU,                 ...  2 //
         SLU,                 ...  3 //
         SLU,                 ...  4 //
         SLU,                 ...  5 //
         HITCOUNT,            ...  6 //
         FLYTO,               ...  7 //
         ALTSELCT,            ...  8 //
         DSPSELCT,            ...  9 //
         STRMODE,             ... 10 //
         FASTFWD,             ... 11 //
         FORWARD,             ... 12 //
         REVERSE,             ... 13 //
         FASTREV,             ... 14 //
         HITCOUNT,            ... 15 //
         ACCEPT,              ... 16 //
         REJECT,              ... 17 //
         ALTELEV,             ... 18 //
         AUTOMAN,             ... 19 //
         LANDSFA,             ... 20 //
         INS1SEL,             ... 21 //
         INS1DIT,             ... 22 //
         INS2SEL,             ... 23 //
         INS2DIT,             ... 24 //
         DEADSLCT,            ... 25 //
         ADDRDR,              ... 26 //
         HITCOUNT,            ... 27 //
         HITCOUNT,            ... 28 //
         INS1ENAI,            ... 29 //
         INS2ENHI,            ... 30 //
         HITCOUNT  $          ... 31 //
        ...       DEFINE ALT/ELEV ALT CAL MODE BRANCH       //
        SWITCH CPNAEBN =
         ACMOAND5,            ...  0 //
         ACM1AND2,            ...  1 //
         ACM1AND2,            ...  2 //
         ACM3,                ...  3 //
         ACM4,                ...  4 //
         ACM0AND5  $          ...  5 //
        ...       ARE THERE NO MORE HITS       //
```

```
C.PNHIT   ASD


ANYHITS  : IF CPNHIT(0) == 0
           THEN GOTO RETURN
           ELSE BEGIN            ... OBTAIN THE NEXT HIT //
                CPNTI1 = CPNHIT(CPNHIT(0)) ;
                CPNSWCH = PNTI1 ;
                ...      IS THE NEW STATUS OFF     //
                IF CPNTI1    0
                ... IS THE HIT A POSITION SWITCH //
                THEN IF CPNT    < -28
                     OR (CPNTI1 >= -14 AND CPNTI1 <= -11)
                      OR CPNTI1 >= -5
                     THEN CPNTI1 = -CPNTI1
                     ELSE       ... IF NOT, IGNORE //
                       GOTO HITCOUNT ;
                ...       NOW BRANCH ON THE SWITCH NUMBER      //
                GOTO CPNBRNCH(CPNTI1) ;
FLYTO  :                 ... IF THE FLY TO REQUEST IS AN OAP... //
         IF IXTYPE(CPSXPTR) == 6
         THEN GOTO HITCOUNT ... IGNORE THE REQUEST //
         ELSE BEGIN      ... TURN ON THE FLY TO LIGHT //
              CPNOUT13 = CPNOUT13 .V. KMASK19 ;
              CPNFLYTO = 1 ; ... SET FLYTO FLAG //
              GOTO ZEROCLOK ;
              END ;
ALTSELCT :                  ... REVERSE HSL AND HR LIGHT STATUS //
         CPNOUT13 = CPNOUT13 .X. "00006000" ;
         CPNHSL = CPNOUT13 .A. "00006000" ; ... SET HSL/HR FLAG //
         GOTO ZEROCLOK ;
SLU  :                      ... SET RESET INDICATED SLU STATUS //
         CPNSLU(CPNTI1) = CPNSWCH ;
         GOTO HITCOUNT ;
DSPSELCT :                  ... CYCLE THE LIGHT TEMPLATE //
         CPNDIS = CPNDIS .LS. 1 ;
         IF CPNDIS .A. KMASK15 ~= 0
         THEN CPNDIS = KMASK19 ;
         IF CPNDIS .A. KMASK17 ~= 0 ... IS THE NEW MODE NV //
         THEN CPNDCLOK = 0 ... STOP THE BLINK TIMER //
         ELSE            ... RE-START THE BLINK TIMER //
          CPNDCLOK = 1 ;
         CPNOUT14 = (CPNOUT14 .A. "FFFF0FFF") .V. CPNDIS ;
                            ... LIGHT APPROPRIATE SEGMENT AND SET THE
                                MODE FLAG //
         CPNDSLR = CPNDIS .RS. 13 ;
         GOTO ZEROCLOK ;
STRMODE  :                  ... REVERSE STEER MODE LIGHTS AND FLAG //
         CPNOUT13 = CPNOUT13 .X. "00000C00" ;
         CPNSTMR = (CPNOUT13 .A. "00000C00") .RS. 11 ;
         GOTO ZEROCLOK ;
FASTFWD  :                  ... SET FWD / REV REQUEST FLAG //
         CPNFWDRV = 10 ;
         GOTO FWDRVOFF ;
FORWARD  :  CPNFWDRV = 1 ;
         GOTO FWDRVOFF ;
REVERSE  :  CPNFWDRV = -1 ;
         GOTO FWDRVOFF ;
```

```
C.PNHIT   AED


FASTREV  :    CPNFWDRV = -10 ;
FWDRVOFF :                    ... IF THE NEW STATUS IS OFF, OR IF THE
                                  XHAIR MODE IS ID... //
              IF CPNSWCH < 0 OR CPSID ¬= 0
              THEN CPNFWDRV = 0 ; ... REMOVE FWD / REV REQUEST FLAG //
              GOTO ZEROCLOK ;
ACCEPT   :                    ... IGNORE UNLESS ALTITUDE CALIBRATION
                                  MODE IS 4 //
              IF CPNACM == 4
              THEN BEGIN
              ... SET ALT CAL COMPUTE FLAG,AND PREPARE
                  TO RECEIVE KALMAN VERDICTS //
                  CPNACC = CPNACS ;
                  CPNACM = 5 ;
                  NKFARJTM = 0 ;
                  NKFARJTA = 0 ;
                  END ;
              GOTO HITCOUNT ;
PEJECT   :                    ... IGNORE IF ALT CAL MODE IS 5, OTHERWISE
                                  END ALT CAL //
              IF CPNACM == 5
              THEN GOTO HITCOUNT
              ELSE BEGIN
                  CPNOUT14 = CPNOUT14 .A. "FFF9FFFF" ;
                  CPNOUT9 = CPNOUT9 .V. "00FFFFFF" ;
                  CPNACM = 0 ;
                  CPNHAC = 0 ;
                  END ;
              GOTO HITCOUNT ;
ALTELEV  :                    ... BRANCH ON ALT CAL MODE //
              GOTO CPNAEBN(CPNACM) ;
ACM1AND2 :                    ... TURN OUT ELEV LIGHT, PULL HIGH ALT CAL
                                  FLAG DOWN //
              CPNOUT14 = CPNOUT14 .A. KRMASK13 ;
ACM4     :    CPNHAC = 0 ;
              CPNACM = 3 ;
              IF NCRALTOK == 0 ... ARE THE RADAR ALTIMETERS DOWN //
                              ... IS THE FLR UNAVAILABLE //
              THEN IF CPFLRON == 0
                  THEN BEGIN ... TERMINATE ALT CAL //
                      CPNOUT9 = CPNOUT9 .V. "00FFFFFF" ;
                      CPNOUT14 = CPNOUT14 .A. KRMASK14 ;
                      CPNACM = 0 ;
                      END
                  ELSE BEGIN ... SET HIGH ALT CAL FLAG AND ZERO
                                  THE DELTA BUFFER //
                      CPNHAC = 1 ;
                      CPNDELTA = 0 ;
                      END ;
              GOTO HITCOUNT ;
ACM3     :                    ... RECORD WHICH NAV SYSTEM WAS USED FOR
                                  ALT CAL //
              CPNACS = CPNNMS ;
              CPNOUT14 = CPNOUT14 .V. KMASK14 ; ... ADVANCE TO MODE 4 //
              CPNACM = 4 ;
```

327

```
C.PNHIT   AED


ACMOANDS  :    GOTO HITCOUNT !
AUTOMAN   !                      ... REVERSE LIGHTS AND MODE. INITIALIZE
                                     SYSTEM REQUEST //
      .         CPNOUT14 = CPNOUT14 .X. "0C000000" !
                CPNMANH = (CPNOUT14 .A. "0C000000") .RS. 27 !
                CPNNMSB = NCPRNV !
                GOTO HITCOUNT !
LANDSEA   :                      ... REVERSE LAND / SEA MODE //
                CPNLANDB = CPNLANDB .X. "60000000" !
                GOTO HITCOUNT !
INS1SEL   :                      ... REQUEST INS1 //
                CPNNMS = 4 !
                GOTO HITCOUNT !
INS1DIT   :                      ... IF IN MANUAL MODE, CYCLE LIGHT AND
                                     TENTATIVE REQUEST.RESET DISPLAY CLOCK
                                     TO ALLOW ONE SECOND MODE SET DELAY //
                IF CPNMAN == 0
                THEN BEGIN
                    CPNMDIT = CPNMDIT .LS. 1 !
                    IF CPNMDIT .A. KMASK27 ¬= 0
                    THEN CPNMDIT = KMASK31 !
                    CPNOUT4 = CPNOUT4 .A. "FFFFF8FF"
                    .V. (CPNMDIT .LS. 8) !
                    CPNCLOCK = 13 !
                    END !
                GOTO HITCOUNT !
INS2SEL   :     CPNNMS = 2 !
                GOTO HITCOUNT !
INS2DIT   :     IF CPNMAN == 0
                THEN BEGIN
                    CPNADIT = CPNADIT .LS. 1 !
                    IF CPNADIT .A. KMASK27 ¬= 0
                    THEN CPNADIT = KMASK31 !
                    CPNOUT0 = CPNOUT0 .A. "F8FFFFFF"
                    .V. (CPNADIT .LS. 24) !
                    CPNCLOCK = 13 !
                    END !
                GOTO HITCOUNT !
DEADSLCT  :                      ... SET REQUEST FLAG TO DEAD RECKON //
                CPNNMS = 1 !
                GOTO HITCOUNT !
ADDRDR    :                      ... REVERSE DR / ADDR REQUEST FLAG //
                CPNDDSB = CPNDDSB .X. "00000003" !
                GOTO HITCOUNT !
INS1ENBL  :                      ... SET INS ENABLE FLAG TO APPROPRIATE
                                     VALUE //
                CPNINS1 = CPNSWCH !
                GOTO HITCOUNT !
INS2ENBL  :     CPNINS2 = CPNSWCH !
                GOTO HITCOUNT !
    ...        ALL BRANCHES RETURN HERE      //
ZEROCLOK  :                      ... SET THE DISPLAY CLOCK TO ZERO //
                CPNCLOCK = 0 !
HITCOUNT  :     CPNHIT(0) = CPNHIT(0) - 1 !
                GOTO ANYHITS !
                END !
        END !                    ... OF HIT PROCEDURE DEFINITION //
    END FINI
```

```
        INTEGER CPNMTIME $     ... MISSION TIME IN MAJOR FRAMES //
        INTEGER CPNTIMB $      ... HIT NAME FOR MISSION TIME //
        INTEGER CPNCLOCK $     ... DISPLAY CLOCK--A MOD 16 COUNTER //
        INTEGER CPNCLOKB $     ... HIT NAME FOR THE DISPLAY CLOCK //
        INTEGER CPNHSL $       ... HR/HSL FLAG //
        INTEGER CPNDCLOK $     ... DISPLAY SELECT BLINK TIMER //
        INTEGER CPNDIS $       ... DISPLAY SELECT TEMPLATE //
        INTEGER CPNACM $       ... ALT CAL MODE //
        INTEGER CPNMDIT $      ... INS1 DIT REQUEST FLAG //
        INTEGER CPNADIT $      ... INS2 DIT REQUEST FLAG //
        INTEGER CPNTI1 $       ... TEMP1 //
        INTEGER CPNTR1 $       ... TEMP1 //
        REAL CPNTF1 $          ... TEMP1 //
        INTEGER CPNTI2 $       ... TEMP2 //
        INTEGER CPNTR2 $       ... TEMP2 //
        REAL CPNTF2 $          ... TEMP2 //
                               ... VARIABLES LOCAL TO C.PNHIT //
        INTEGER CPNSWCH $      ... ACTIVE HIT SWITCH NO. / STATUS //
        INTEGER CPNACS $       ... NAV SYSTEM USED FOR ALT CAL //
                               ... DATA LOCAL TO C.PNDISP //
        INTEGER CPNPPDOK $     ... PRESENT POSITION DATA OK FLAG //
        INTEGER CPNSELNO $     ... SELECTED PT. SEQ NO. //
        INTEGER CPNUPRJ $      ... UPDATE REJECT LIGHT TEMPLATE //
        INTEGER CPNMAG1 $      ... SELECTED POINT MAGWHEEL TIMER //
        INTEGER CPNSLTYP $     ... SELECTED POINT TYPE //
        REAL CPNTNELV $        ... TERRAIN ELEV. ,FEET //
        REAL CPNSELON $        ... SELECTED POINT LONG ,RADIANS //
        REAL CPNSELAT $        ... SELECTED POINT LAT ,RADIANS //
        REAL CPNSLELV $        ... SELECTED POINT ELEV,FEET //
        INTEGER CPNSTRNO $     ... STEER POINT SEQ NO. //
        INTEGER CPNMAG2 $      ... STEER POINT MAGWHEEL TIMER //
        INTEGER CPNSTTYP $     ... STEER POINT TYPE //
        REAL ARRAY CPNSLPOS(1) $ ... SELECTED POINT POSITION //
        REAL ARRAY CPNTAR(1) $ ... TIME AND RANGE TO SELECTED PT //
        INTEGER ARRAY CPNMCODE(9) $ ... MAGWHEEL DRIVER CODE //
                               ... THE FOLLOWING ARE B 32 NAMES FOR THE ABOVE
                                   OUTPUTS TO THE PANEL //
        INTEGER CPNOUT0 $
        INTEGER CPNOUT1 $
        INTEGER CPNOUT2 $
        INTEGER CPNOUT3 $
        INTEGER CPNOUT4 $
        INTEGER CPNOUT5 $
        INTEGER CPNOUT6 $
        INTEGER CPNOUT7 $
        INTEGER CPNOUT8 $
        INTEGER CPNOUT9 $
        INTEGER CPNOUT10 $
        INTEGER CPNOUT11 $
        INTEGER CPNOUT12 $
        INTEGER CPNOUT13 $
        INTEGER CPNOUT14 $
        INTEGER CPNOUT15 $     ... //
                               ... THE FOLLOWING IS AN ARRAY NAME FOR THE
                                   OUTPUT TO THE PANELS //
```

```
CPNOWN    AEO

    INTEGER ARRAY CPNOUTA(15) ;
    ... THE FOLLOWING VARIABLES ARE REFERENCED BY OTHER PROGRAMS //
    INTEGER CPNSTEER ;   ... STEER MODE 0--TRK 1--DIR //
    INTEGER CPNSTRB ;    ... BIT NAME FOR STEER MODE //
    INTEGER CPNFWDRV ;   ... FORWARD REVERSE REQUEST //
    INTEGER CPNROLL ;    ... FORWARD REVERSE COMMAND //
    INTEGER CPNACC ;     ... ALT CAL COMPUTE //
    INTEGER CPNHAC ;     ... HIGH ALT CAL REQUEST //
    INTEGER CPNMAN ;     ... AUTO/MAN MODE FLAG //
    INTEGER CPNMANB ;    ... BIT NAME FOR AUTO/MAN FLAG //
    INTEGER CPNLAND ;    ... LAND/SEA FLAG //
    INTEGER CPNLANDB ;   ... BIT NAME FOR LAND/SEA FLAG //
    INTEGER CPNNMS ;     ... NAV SYSTEM SELECT //
    INTEGER CPNNMSB ;    ... BIT NAME FOR NMS //
    INTEGER CPNDDS ;     ... DR/ADDR REQUEST //
    INTEGER CPNDDSB ;    ... BIT NAME FOR DR/ADDR //
    INTEGER CPNINS1 ;    ... INS1 ENABLE/DISABLE //
    INTEGER CPNINS2 ;    ... INS2 ENABLE/DISABLE //
    INTEGER CPNSLUA ;    ... AFT //
    INTEGER CPNSLUL ;    ... LEFT PYLON //
    INTEGER CPNSLUR ;    ... RIGHT PYLON SLU ENABLE //
    INTEGER CPNSLUI ;    ... INTRMD //
    INTEGER CPNSLUF ;    ... FORWARD //
    REAL CPNDELTA ;      ... ALT CAL DELTA BUFFER //
    INTEGER ARRAY CPNSLU(5) ; ... THIS IS THE SLU STATUS TABLE.
                              IT IS 'OVERLAYED' WITH THE SLU
                              STATUS WORDS ABOVE //
    INTEGER CPNFLYTO ;   ... FLY TO FLAG //
    INTEGER CPNDSL ;     ... DISPALY SELECT FLAG //
    INTEGER CPNDSLB ;    ... BIT NAME FOR CPNDS2 //
                         ... OVERLAY DECLARATIONS FOR CPNOWN //
    CPNSTRB $=$ CPNSTEER ;
    CPNMANB $=$ CPNMAN ;
    CPNLANDB $=$ CPNLAND ;
    CPNNMSB $=$ CPNNMS ;
    CPNDDSB $=$ CPNDDS ;
    CPNSLUA $=$ CPNSLU(1) ;
    CPNSLUL $=$ CPNSLU(2) ;
    CPNSLUR $=$ CPNSLU(3) ;
    CPNSLUI $=$ CPNSLU(4) ;
    CPNSLUF $=$ CPNSLU(5) ;
    CPNTIMB $=$ CPNMTIME ;
    CPNCLOKB $=$ CPNCLOCK ;
    CPNTB1 $=$ CPNTF1 $=$ CPNTI1 ;
    CPNTB2 $=$ CPNTF2 $=$ CPNTI2 ;
    CPNDSLB $=$ CPNDSL ;
```

```
    INTEGER ARRAY CPNHIT(7) $ ... THE NAV PROGRAM HIT TABLE //
  . INTEGER CPSID $         ... ID MODE FLAG //
    INTEGER CPFLRON $       ... FORWARD LOOKING RADAR ON //
    INTEGER CMALTCAL $      ... NEXT DEST. AN ACT CAL //
    INTEGER CPKALTCL $      ... IKB ALT CAL REQUEST //
    REAL CPKTNELV $         ... IKB TERRAIN ELEVATION //
    INTEGER CPSNUIP $       ... POSITION UPDATE IN PROGRESS //
    INTEGER CMCSTP $        ... STEER POINT FLAG //
    INTEGER CMCSPID $       ... STEER POINT TYPE //
    INTEGER CMCSPSN $       ... STEER PT. SEQ. NO. //
    INTEGER CAFFDTF $       ... FLT DR ENGD / AUTO TF FLAG //
    INTEGER CAFSMODE $      ... FLIGHT DIRECTOR MODE //
    INTEGER ARRAY CPBCD(7) $ ... BCD ARRAY //
    INTEGER ARRAY CMXLAT(7) $
    INTEGER ARRAY CMXLONG(7) $
    INTEGER ARRAY CMXELEV(7) $
    INTEGER ARRAY CMXQUAL(7) $
    INTEGER ARRAY CMXSQNO(7) $
    INTEGER ARRAY CMXTYPE(7) $
    INTEGER CPSXPTR $       ... POINTER INTO CMXHAIR //
    INTEGER NCRALTOK $      ... RADAR ALTITUDE OK //
    REAL NCRALT $          ... RADAR ALTITUDE,FEET //
    REAL NCALTP $          ... PRIME ALTITUDE,FT . //
    REAL NSTTG $           ... TIME TO DESTINATION,SEC //
    REAL NSRNG $           ... RANGE TO DESTINATION //
    REAL NLAT $            ... DISPLAY LAT,RADIANS //
    REAL NLONG $           ... DISPLAY LONG, RADIANS //
    REAL NVGND $           ... DISPLAY GND SPEED,FT / SEC //
    REAL NHGT $            ... GROUND TRACK,RADIANS //
    REAL NTHEAD $          ... TRUE HEADING,RADIANS //
    REAL NWHEAD $          ... WIND HEADING,RADIANS //
    REAL NWIND $           ... WIND SPEED, FT / SEC //
    REAL NHDISP $          ... HEIGHT ABOVE SEA LEVEL,FT //
    INTEGER NDRMODE $       ... DEAD RECKON MODE //
    INTEGER NCDOPCUT $      ... DOPPLER CUT-OUT FLAG //
    INTEGER NCAALCPT $      ... INS2 UP //
    INTEGER NCMALCPT $      ... INS1 UP //
    INTEGER NVFSTRT $       ... DEAD RECKON RESTART FLAG //
    INTEGER NKFARJTM $      ... INS1 ALT CAL REJECT FLAG //
    INTEGER NKFARJTA $      ... INS2 ALT CAL REJECT FLAG //
    INTEGER NCAVAILM $      ... INS1 DIT AVAILABLE FLAG //
    INTEGER NCAVAILA $      ... INS2 DIT AVAILABLE FLAG //
    INTEGER NCMDIT $
    INTEGER NCADIT $
    INTEGER NCPRNV $        ... PRIME SYSTEM FLAG //
```

```
        REAL NILATM      I  ... LATITUDE(RADIANS) //
        REAL NILONM      I  ... LONGITUDE(RADIANS) //
        REAL NIALTM      I  ... ALTITUDE(FEET) //
        REAL NIVNM       I  ... VELOCITY NORTH(FT/SEC) //
        REAL NIVEM       I  ... VELOCITY EAST(FT/SEC) //
        REAL NIHDOTM     I  ... ALTITUDE RATE(FT/SEC) //
        REAL NIPSITM   · I  ... TRUE HEADING(RADIANS) //
        REAL NIPSIMM     I  ... MAGNETIC HEADING(RADIANS) //
        REAL NIGRAVM     I  ... LOCAL GRAVITY(FT/SEC**2) //
        REAL NCMPITCH    I  ... PITCH(RADIANS) //
        REAL NCMROLL     I  ... ROLL(RADIANS) //
        REAL NCMYAW      I  ... YAW(RADIANS) //
        REAL NCMPDOT     I  ... PITCH RATE(RAD/SEC) //
        REAL NCMRDOT     I  ... ROLL RATE(RAD/SEC) //
        REAL NCMYDOT     I  ... YAW RATE(RAD/SEC) //
...                                                      //
...     NAVIGATION DATA COMMON TO MASTER AND AUXILIARY //
...                                                      //
        REAL NIALFAM     I  ... WANDER ANGLE(RADIANS) //
        REAL NIVXM       I  ... X PLATFORM VELOCITY(FT/SEC) //
        REAL NIVYM       I  ... Y PLATFORM VELOCITY(FT/SEC) //
        REAL NIVZM       I  ... Z PLATFORM VELOCITY(FT/SEC) //
        REAL NCMAX       I  ... X ACCELEROMETER(FT/SEC/FRAME) //
        REAL NCMAY       I  ... Y ACCELEROMETER(FT/SEC/FRAME) //
        REAL NCMAZ       I  ... Z ACCELEROMETER(FT/SEC/FRAME) //
        REAL NIDVAXM     I  ... DELTA X VELOCITY(FT/SEC/FRAME) //
        REAL NIDVAYM     I  ... DELTA Y VELOCITY(FT/SEC/FRAME) //
        REAL NIDVAZM     I  ... DELTA Z VELOCITY(FT/SEC/FRAME) //
        REAL NIC12M      I  ... DIRECTION COSINE C(1,2) //
        REAL NIC22M      I  ... DIRECTION COSINE C(2,2) //
        REAL NIC32M      I  ... DIRECTION COSINE C(3,2) //
        REAL NIC13M      I  ... DIRECTION COSINE C(1,3) //
        REAL NIC23M      I  ... DIRECTION COSINE C(2,3) //
        REAL NIC33M      I  ... DIRECTION COSINE C(3,3) //
        REAL NIC12DM     I  ... DERIVATIVE OF C(1,2) //
        REAL NIC22DM     I  ... DERIVATIVE OF C(2,2) //
        REAL NIC32DM     I  ... DERIVATIVE OF C(3,2) //
        REAL NIC13DM     I  ... DERIVATIVE OF C(1,3) //
        REAL NIC23DM     I  ... DERIVATIVE OF C(2,3) //
        REAL NIC33DM     I  ... DERIVATIVE OF C(3,3) //
        REAL NIOMEGXM    I  ... EARTH RATE ABOUT X(RAD/SEC) //
        REAL NIOMEGYM    I  ... EARTH RATE ABOUT Y(RAD/SEC) //
        REAL NIOMEGZM    I  ... EARTH RATE ABOUT Z(RAD/SEC) //
        REAL NIRHOXM     I  ... CRAFT RATE ABOUT X(RAD/SEC) //
        REAL NIRHOYM     I  ... CRAFT RATE ABOUT Y(RAD/SEC) //
        REAL NIPRXM      I  ... PLATFORM RATE ABOUT X(RAD/SEC) //
        REAL NIPRYM      I  ... PLATFORM RATE ABOUT Y(RAD/SEC) //
        REAL NIAXCM      I  ... CORRECTED X ACCEL(FT/SEC/FRAM) //
        REAL NIAYCM      I  ... CORRECTED Y ACCEL(FT/SEC/FRAM) //
        REAL NIAZCM      I  ... CORRECTED Z ACCEL(FT/SEC/FRAM) //
        REAL NIDAXCM     I  ... X ACCEL NOISE EST(FT/SEC/FRGM) //
        REAL NIDAYCM     I  ... Y ACCEL NOISE EST(FT/SEC/FRAM) //
        REAL NIDAZCM     I  ... Z ACCEL BIAS EST9FT/SEC/FRAME) //
...                                                      //
...             KALMAN FILTER DATA //
```

332

NRBMAST   AED

```
    ...
        REAL NKFPRJTM          ! ... POSITION FIX REJECT FLAG //
        REAL ARRAY NGIMSM( 50) !
        REAL NKPO101M           !  ... COVARIANCE ELEMENT P(1,1) //
        REAL NKPO201M           !  ... COVARIANCE ELEMENT P(2,1) //
        REAL ARRAY NKFPHATM(189) ! ... INS 1 COVARIANCE MATRIX //
        REAL ARRAY NKXHATM(18) !   ... INS 1 STATE VECTOR //
        REAL ARRAY NKYMSVM(4) !    ... INS 1 MEASUREMENT VECTOR //
```

```
      REAL NILATA      $  ... LATITUDE(RADIANS) //
      REAL NILONA      $  ... LONGITUDE(RADIANS) //
      REAL NTALTA      $  ... ALTITUDE(FEET) //
      REAL NIVNA       $  ... VELOCITY NORTH(FT/SEC) //
      REAL NIVEA       $  ... VELOCITY EAST(FT/SEC) //
      REAL NIHDOTA     $  ... ALTITUDE RATE(FT/SEC) //
      REAL NIPSITA     $  ... TRUE HEADING(RADIANS) //
      REAL NIPSIMA     $  ... MAGNETIC HEADING(RADIANS) //
      REAL NIGRAVA     $  ... LOCAL GRAVITY(FT/SEC** ) //
      REAL NCAPITCH    $  ... PITCH(RADIANS) //
      REAL NCAROLL     $  ... ROLL(RADIANS) //
      REAL NCAYAW      $  ... YAW(RADIANS) //
      REAL NCAPDOT     $  ... PITCH RATE(RAD/SEC) //
      REAL NCARDOT     $  ... ROLL RATE(RAD/SEC) //
      REAL NCAYDOT     $  ... YAW RATE(RAD/SEC) //
      ...                                                      //
      ...  NAVIGATION DATA COMMON TO MASTER AND AUXILIARY      //
      ...                                                      //
      REAL NIALFAA     $  ... WANDER ANGLE(RADIANS) //
      REAL NIVXA       $  ... X PLATFORM VELOCITY(FT/SEC) //
      REAL NIVYA       $  ... Y PLATFORM VELOCITY(FT/SEC) //
      REAL NIVZA       $  ... Z PLATFORM VELOCITY(FT/SEC) //
      REAL NCAAX       $  ... X ACCELEROMETER(FT/SEC/FRAME) //
      REAL NCAAY       $  ... Y ACCELEROMETER(FT/SEC/FRAME) //
      REAL NCAAZ       $  ... Z ACCELEROMETER(FT/SEC/FRAME) //
      REAL NIDVAXA     $  ... DELTA X VELOCITY(FT/SEC/FRAME) //
      REAL NIDVAYA     $  ... DELTA Y VELOCITY(FT/SEC/FRAME) //
      REAL NIDVAZA     $  ... DELTA Z VELOCITY(FT/SEC/FRAME) //
      REAL NIC12A      $  ... DIRECTION COSINE C(1,2) //
      REAL NIC22A      $  ... DIRECTION COSINE C(2,2) //
      REAL NIC32A      $  ... DIRECTION COSINE C(3,2) //
      REAL NIC13A      $  ... DIRECTION COSINE C(1,3) //
      REAL NIC23A      $  ... DIRECTION COSINE C(2,3) //
      REAL NIC33A      $  ... DIRECTION COSINE C(3,3) //
      REAL NIC12DA     $  ... DERIVATIVE OF C(1,2) //
      REAL NIC22DA     $  ... DERIVATIVE OF C(2,2) //
      REAL NIC32DA     $  ... DERIVATIVE OF C(3,2) //
      REAL NIC13DA     $  ... DERIVATIVE OF C(1,3) //
      REAL NIC23DA     $  ... DERIVATIVE OF C(2,3) //
      REAL NIC33DA     $  ... DERIVATIVE OF C(3,3) //
      REAL NIOMEGXA    $  ... EARTH RATE ABOUT X(RAD/SEC) //
      REAL NIOMEGYA    $  ... EARTH RATE ABOUT Y(RAD/SEC) //
      REAL NIOMEGZA    $  ... EARTH RATE ABOUT Z(RAD/SEC) //
      REAL NIRHOXA     $  ... CRAFT RATE ABOUT X(RAD/SEC) //
      REAL NIRHOYA     $  ... CRAFT RATE ABOUT Y(RAD/SEC) //
      REAL NIPRXA      $  ... PLATFORM RATE ABOUT X(RAD/SEC) //
      REAL NIPRYA      $  ... PLATFORM RATE ABOUT Y(RAD/SEC) //
      REAL NIAXCA      $  ... CORRECTED X ACCEL(FT/SEC/FRAME) //
      REAL NIAYCA      $  ... CORRECTED Y ACCEL(FT/SEC/FRAME) //
      REAL NIAZCA      $  ... CORRECTED Z ACCEL(FT/SEC/FRAME) //
      REAL NIDAXCA     ?  ... X ACCEL NOISE EST(FT/SEC/FRAME) //
      REAL NIDAYCA     $  ... Y ACCEL NOISE EST(FT/SEC/FRAME) //
      REAL NIDAZCA     $  ... Z ACCEL BIAS EST(FT/SEC/FRAME) //
      ...                                                      //
      ...          KALMAN FILTER DATA                          //
```

```
                                                              //
      ...
      REAL NKFPRJTA         !  ...  POSITION FIX REJECT FLAG //
      REAL ARRAY NAINSA( 50) !
      REAL NKP0101A         !  ...  COVARIANCE ELEMENT P(1,1) //
      REAL NKP0201A         !  ...  COVARIANCE ELEMENT P(2,1) //
      REAL ARRAY NKFPHATA(189) ! ... COVARIANCE MATRIX ARRAY //
      REAL ARRAY NKXHATA(18) !    ... STATE VECTOR //
      REAL ARRAY NKYHSVA(4) !      ... MEASUREMENT VECTOR // .
```

```
REAL NKFTMP1 ;                   ... TEMPORARY STORAGE //
REAL NKFTMP2 ;                   ... TEMPORARY STORAGE //
REAL NKFTMP3 ;                   ... TEMPORARY STORAGE //
INTEGER NKFNDBST ;               ... NAVIGATION TABLE DATA INDEX //
INTEGER NCCPOVL ;                ... POSITION FIX TYPE FLAG //
INTEGER NKFI ;                   ... LOOP INDEX //
INTEGER NKFJ ;                   ... LOOP INDEX //
INTEGER NKFH ;                   ... LOOP INDEX //
INTEGER NKFS ;                   ... SUBSCRIPT //
INTEGER NKFT ;                   ... SUBSCRIPT //
REAL NKFKGD ;                    ... DENOMINATIG OF KALMAN GAIN //
INTEGER NKFSTAND ;               ... ELEMENTS IN STATE VECTOR
                                     LESS 1 //
REAL NKFRMATX ;                  ... POSITION MEASUREMENT
                                     NOISE(FT**2) //
REAL NKFAN ;                     ... NORTH COMPONENT OF POS
                                     ERROR(FT) //
REAL NKFAE ;                     ... EAST COMPONENT OF POS
                                     ERROR(FT) //
REAL NKFEX ;                     ... X TERM IN POS TEST //
REAL NKFEY ;                     ... Y TERM IN POS TEST //
INTEGER NCPRIME ;                ... PRIME DATA FLAG //
REAL CSNPDN ;                    ... NORTH MEASURED POS ERROR(FT) //
REAL CSNPDE ;                    ... EAST MEASURED POS ERROR(FT) //
REAL NCLATP ;                    ... PRIME LATITUDE(RADIANS) //
REAL NCLONP ;                    ... PRIME LONGITUDE(RADIANS) //
REAL NLATC ;                     ... OVERFLY CHECKPOINT
                                     LATITUDE(RAD) //
REAL NLONC ;                     ... OVERFLY CHECKPOINT
                                     LONGITUDE(RAD) //
INTEGER CMFPQUAL ;               ... CHECKPOINT QUALITY INDEX //
REAL NDALT ;                     ... DEAD RECKONING ALTITUDE //
REAL ARRAY NKFKGAIN(38) ;        ... KALMAN GAIN MATRIX //
INTEGER ARRAY NKFCOLMN(18) ;     ... SYMMETRIC MATRIX POINTERS //
REAL ARRAY KRPMATX(2) ;          ... POSITION NOISE(FT**2) //
REAL ARRAY NKINITP(18) ;         ... INITIAL VALUE OF COV MATRIX //
REAL KNINE ;                     ... POITION TEST CONST //
REAL KERADIUS ;                  ... EARTH RADIUS AT EQUATOR(FT) //
REAL KDFLTAT ;                   ... FRAME TIME(SEC) //
REAL KGRAV0 ;                    ... GRAVITATIONAL CONSTANT //
REAL KGRAV1 ;                    ... GRAVITATIONAL CONSTANT //
REAL KGRAV2 ;                    ... GRAVITATIONAL CONSTANT //
REAL KELPTCTY ;                  ... EARTH ELLIPTICITY //
REAL K2FORE ;                    ... 2*KELPTCTY/KERADIUS //
REAL K1ORE ;                     ... 1/KERADIUS //
REAL KEARTHR ;                   ... EARTH RATE(RAD/SEC) //
REAL KNK1 ;                      ... INERTIAL CONSTANT //
REAL KNK2 ;                      ... INERTIAL CONSTANT //
REAL KONE ;                      ... ONE //
```

336

```
SYNONYMS 57.2958 = CPNK1 ;        ... DEGREES/RADIAN //
SYNONYMS .5921 = CPNK2 ;          ... KNOTS/(FT/SEC) //
SYNONYMS "80000000"  = KMASK0 ;
SYNONYMS "40000000"  = KMASK1 ;
SYNONYMS "20000000"  = KMASK2 ;
SYNONYMS "10000000"  = KMASK3 ;
SYNONYMS "08000000"  = KMASK4 ;
SYNONYMS "04000000"  = KMASK5 ;
SYNONYMS "02000000"  = KMASK6 ;
SYNONYMS "01000000"  = KMASK7 ;
SYNONYMS "00800000"  = KMASK8 ;
SYNONYMS "00400000"  = KMASK9 ;
SYNONYMS "00200000"  = KMASK10 ;
SYNONYMS "00100000"  = KMASK11 ;
SYNONYMS "00080000"  = KMASK12 ;
SYNONYMS "00040000"  = KMASK13 ;
SYNONYMS "00020000"  = KMASK14 ;
SYNONYMS "00010000"  = KMASK15 ;
SYNONYMS "00008000"  = KMASK16 ;
SYNONYMS "00004000"  = KMASK17 ;
SYNONYMS "00002000"  = KMASK18 ;
SYNONYMS "00001000"  = KMASK19 ;
SYNONYMS "00000800"  = KMASK20 ;
SYNONYMS "00000400"  = KMASK21 ;
SYNONYMS "00000200"  = KMASK22 ;
SYNONYMS "00000100"  = KMASK23 ;
SYNONYMS "00000080"  = KMASK24 ;
SYNONYMS "00000040"  = KMASK25 ;
SYNONYMS "00000020"  = KMASK26 ;
SYNONYMS "00000010"  = KMASK27 ;
SYNONYMS "00000008"  = KMASK28 ;
SYNONYMS "00000004"  = KMASK29 ;
SYNONYMS "00000002"  = KMASK30 ;
SYNONYMS "00000001"  = KMASK31 ;
SYNONYMS "7FFFFFFF"  = KRMASK0 ;
SYNONYMS "BFFFFFFF"  = KRMASK1 ;
SYNONYMS "DFFFFFFF"  = KRMASK2 ;
SYNONYMS "EFFFFFFF"  = KRMASK3 ;
SYNONYMS "F7FFFFFF"  = KRMASK4 ;
SYNONYMS "FBFFFFFF"  = KRMASK5 ;
SYNONYMS "FDFFFFFF"  = KRMASK6 ;
SYNONYMS "FEFFFFFF"  = KRMASK7 ;
SYNONYMS "FF7FFFFF"  = KRMASK8 ;
SYNONYMS "FFBFFFFF"  = KRMASK9 ;
SYNONYMS "FFDFFFFF"  = KRMASK10 ;
SYNONYMS "FFEFFFFF"  = KRMASK11 ;
SYNONYMS "FFF7FFFF"  = KRMASK12 ;
SYNONYMS "FFFBFFFF"  = KRMASK13 ;
SYNONYMS "FFFDFFFF"  = KRMASK14 ;
SYNONYMS "FFFEFFFF"  = KRMASK15 ;
SYNONYMS "FFFF7FFF"  = KRMASK16 ;
SYNONYMS "FFFFBFFF"  = KRMASK17 ;
SYNONYMS "FFFFDFFF"  = KRMASK18 ;
SYNONYMS "FFFFEFFF"  = KRMASK19 ;
SYNONYMS "FFFFF7FF"  = KRMASK20 ;
```

337

CPNCONST AED

```
SYNONYMS "FFFFFBFF"  = KRMASK21 ;
SYNONYMS "FFFFFDFF"  = KRMASK22 ;
SYNONYMS "FFFFFEFF"  = KRMASK23 ;
SYNONYMS "FFFFFF7F"  = KRMASK24 ;
SYNONYMS "FFFFFFBF"  = KRMASK25 ;
SYNONYMS "FFFFFFDF"  = KRMASK26 ;
SYNONYMS "FFFFFFEF"  = KRMASK27 ;
SYNONYMS "FFFFFFF7"  = KRMASK28 ;
SYNONYMS "FFFFFFFB"  = KRMASK29 ;
SYNONYMS "FFFFFFFD"  = KRMASK30 ;
SYNONYMS "FFFFFFFE"  = KRMASK31 ;
```

APPENDIX 3

SAMPLE RAW STATIC AND DYNAMIC DATA

1.   <u>Meaning of Raw Data Matrix Elements</u>

      Each of the 25 features counted by the instrumented compiler was assigned a unique row in a 25 x 25 raw data output matrix. The meaning of each column in the row depends upon the form of data being gathered, and includes histogram as well as specific language forms and constructs. The following table presents the meaning of each row and column in the data matrix. (An "X" indicates that the element has no meaning.) A further explanation of each row is given following the table. Column 0 is the sum of all the other columns.

Table 18. Format of Raw Data Output From Instrumented Compiler

COLUMNS

LABEL FOR ROWS	0	1	2	3	4	5	6	7	8	9	10
IF	X	TOTAL	BEGIN	CALL	DO	IF	GOTO	=	X	X	X
ELSE	TOTAL	X	BEGIN	CALL	DO	IF	GOTO	=	X	X	X
FOR	X	TOTAL	UNTIL	WHILE	OTHER	MULT.	X	X	X	X	X
L-STEP	TOTAL	LIT=1	LIT≠1	VAR.	EXP.	X	X	X	X	X	X
R-STEP	TOTAL	LIT=1	LIT≠1	VAR.	EXP.	X	X	X	X	X	X
UNTIL	TOTAL	LIT=1	LIT≠1	VAR.	EXP.	X	X	X	,	X'	X
FORMW	TOTAL	X	BEGIN	CALL	DO	IF	GOTO	=	X	X	X
PM	TOTAL	A±B	A±LIT	A±1	LIT±A	1±A	E±LIT	E±1	LIT±E	1±E	OTHER
M	TOTAL	A*B	A*LIT	A*2	LIT*A	2*A	E*LIT	E*2	LIT*E	2*E	OTHER
D	TOTAL	A/B	A/LIT	A/2	LIT/A	1/A	E/LIT	E/2	LIT/E	1/E	OTHER
A	TOTAL	A=0	A=1	A=LIT	A=B	A=E	A=f	X	X	X	X
HIST	TOTAL	0ops.	1ops.	2ops.	3ops.	4ops.	5ops.	6ops.	7ops.	8ops.	9ops.
BOOL	TOTAL	0ops.	1ops.	2ops.	3ops.	4ops.	5ops.	6ops.	7ops.	8ops.	9ops.
EQL	TOTAL	A==B	A==LIT	A==0	LIT==A	0==A	E==LIT	E==0	LIT==E	0==LIT	OTHER
GEQ	TOTAL	A>=B	A>=LIT	A>=0	LIT>=A	0>=A	E>=LIT	E>=0	LIT>=E	0>=LIT	OTHER
GRT	TOTAL	A> B	A> LIT	A> 0	LIT> A	0> A	E> LIT	E> 0	LIT> E	0> LIT	OTHER
LEQ	TOTAL	A<=B	A<=LIT	A<=0	LIT<=A	0<=A	E<=LIT	E<=0	LIT<=E	0<=LIT	OTHER
LES	TOTAL	A< B	A< LIT	A< 0	LIT< A	0< A	E< LIT	E< 0	LIT< E	0< LIT	OTHER
NEQ	TOTAL	A¬=B	A¬=LIT	A¬=0	LIT¬=A	0¬=A	E¬=LIT	E¬=0	LIT¬=E	0¬=LIT	OTHER
AND	TOTAL	A.A.B	A.A.LIT	A.A.0	LIT.A.A	0.A.A	E.A.LIT	E.A.0	LIT.A.E	0.A.LIT	OTHER
OR	TOTAL	A.V.B	A.V.LIT	A.V.0	LIT.V.A	0.V.A	E.V.LIT	E.V.0	LIT.V.E	0.V.LIT	OTHER
LBL,GOTO	TOTAL	LABELS	GOTO's	SWITCHES	X	λ	X	X	X	X	X
PROC	X	TOTAL	0ops.	1ops.	2ops.	3ops.	4ops.	5ops.	6ops.	7ops.	X
DO.DPT	TOTAL	SINGLE	DOUBLE	TRIPLE	X	X	X	X	X	X	X
DO.STA	TOTAL	LEVEL1	LEVEL2	LEVEL3	X	X	X	X	X	X	X

Row Label	Explanation
IF	Columns represent forms of statements following the THEN in IF-THEN or IF-THEN-ELSE statements. Only columns 1 through 7 have meaning.
ELSE	Same as for IF, above, except following ELSE.
FOR	Columns represent forms of FOR statement (FOR-STEP-UNTIL, FOR-WHILE, etc.) The column labeled "Mult" represents the number of "FOR-lists". Only columns 1 through 5 have meaning.
L-STEP	Columns represent the form of FOR-loop starting value (FOR var = value). Only columns 0 through 4 are meaningful.
R-STEP	Same as L-STEP, except for the FOR-loop increment (FOR var = value STEP inc.)
UNTIL	Same as L-STEP and R-STEP, except for the FOR-loop iteration limit (FOR var = value STEP inc UNTIL limit).
FORMW	Columns represent the form of statement following the FOR (FOR statement). Only columns 0 and 2 through 7 have meaning.
PM	Columns display the use of various forms of operands used with the operators "+" and "—". Columns 10 through 11 are meaningful. In addition to the numbers 1 and 2, the following codes have been used for operand forms in the above matrix:

A	variable
B	variable
LIT	literal
E	expression
F	function call

Row Label	Explanation
M	Same as PM, except for the multiplication operator.
D	Same as for PM and M, except for the division operator
A	Columns represent all forms of assignment statement. Only columns 0 through 6 have meaning.
HIST	Columns are a histogram of the number of operators appearing on the right-hand side of an assignment statement. As in all histogram forms, 25 columns are meaningful in addition to column 0.
BOOL	Columns are a histogram of the number of operators appearing in Boolean expressions used within IF-statements. Boolean forms used within other statements are not included (e.g. within a WHILE clause, in a Boolean assignment, etc.).
EQL, GEQ, etc.	Columns show the use of the comparison operators with various specific forms of operand. Columns 0 through 10 are meaningful.

Row Label	Explanation
LBL, GOTO	Columns represent the number of labels, GOTO's and switches. Only columns 2 through 3 are meaningful.
PROC	Columns are a histogram of the number of arguments given in a procedure or function call.
DO.DPT	Columns are a histogram of the nest depth of FOR-loops. Although 25 columns were potentially meaningful, only the first 4 columns showed non-zero values.
DO.STA	Columns are a histogram of the number of statements used within the FOR-loop nest depths measured by DO.DPT.

2. Examples of Raw Data Matrix Output

The two example programs below, while not of the size of most AED or JOBIAL J3B compiler modules, do permit us to present some concrete and specific illustrations of some of the compiler instrumentation. The two brief programs are virtually identical, consisting for the most part of one 'FOR' statement with two subordinate assignments. In the first program the two assignment statements are simple; in the second program one of the assignments involves somewhat more computations. These programs and statistics can be examined below.

Counters relevant to a particular statement type are collected into the array described above and printed out here in row form, with each row bearing its explanatory label. Data for the form of the 'FOR' statement appears in the rows labeled 'FOR', 'L-STEP', 'R-STEP', 'UNTIL' and 'FORMW'. Column 1 of the FOR row simply counts the number of 'FOR' statements. Column 0 of the L-STEP, R-STEP arrays count the occurrence of the forms of initial value and increment of the step variable. Both tables have counts of '1' in these positions since both source programs have the same form of the FOR statement. Column 2 of the FOR, UNTIL, and FORMW rows have a '1', since the UNTIL form is used, the terminating value is a literal (not equal to 1), and a BEGIN follows the DO, respectively.

Similarly, the assignment statistics appear in the row labeled 'A'; in both examples a count of 2 appears in column 1 which counts the assignments of variables. The row labelled 'HIST' contains a histogram of the number of operands in assignment statements. Here, the two examples differ in since in the first example both assignments have only one operand, whereas in the second example one of the assignments has five operands. The reader is invided to verify the remainder of the data output by examining the matrix description.

For each of the two examples, the source language is presented first, followed by the raw data matrix output produced by the instrumented compiler.

343

EXAMPLE 1

```
BEGIN

DEFINE PROCEDURE MAIN TOBE
      BEGIN
      INTEGER A,B,C ;
      INTEGER ARRAY D(10),E(10) ;
      BOOLEAN Z ;
      FOR A = 1 STEP 1 UNTIL 10
      DO   BEGIN
             A = B ;
             A = C ;
             END ;
      END
END FINI
```

MAIN											
IF	0	0	0	0	0	0	0	0	0	0	0
ELSE	0	0	0	0	0	0	0	0	0	0	0
FOR	2	1	1	0	0	0	0	0	0	0	0
L-STEP	1	1	0	0	0	0	0	0	0	0	0
R-STEP	1	1	0	0	0	0	0	0	0	0	0
UNTIL	1	0	1	0	0	0	0	0	0	0	0
FORMN	1	0	1	0	0	0	0	0	0	0	0
PM	0	0	0	0	0	0	0	0	0	0	0
M	0	0	0	0	0	0	0	0	0	0	0
D	0	0	0	0	0	0	0	0	0	0	0
A	2	0	0	0	2	0	0	0	0	0	0
HIST	2	2	0	0	0	0	0	0	0	0	0
BOOL	0	0	0	0	0	0	0	0	0	0	0
EOL	0	0	0	0	0	0	0	0	0	0	0
GEQ	0	0	0	0	0	0	0	0	0	0	0
GRT	0	0	0	0	0	0	0	0	0	0	0
LEQ	0	0	0	0	0	0	0	0	0	0	0
LES	0	0	0	0	0	0	0	0	0	0	0
NEQ	0	0	0	0	0	0	0	0	0	0	0
AND	0	0	0	0	0	0	0	0	0	0	0
OR	0	0	0	0	0	0	0	0	0	0	0
LBL,GOTO	0	0	0	0	0	0	0	0	0	0	0
PROC	0	0	0	0	0	0	0	0	0	0	0
DO.OPT	1	1	0	0	0	0	0	0	0	0	0
DO.STA	2	2	0	0	0	0	0	0	0	0	0

EXAMPLE 2

```
BEGIN

DEFINE PROCEDURE MAIN TOBE
      BEGIN
      INTEGER A,B,C ;
      INTEGER ARRAY D(10),E(10)
      BOOLEAN Z ;
      FOR A = 1 STEP 1 UNTIL 10
      DO    BEGIN
            A = B*B+C*C+10 ;
            A = C ;
            END ;
      END
END FINI
```

MAIN											
IF	0	0	0	0	0	0	0	0	0	0	0
ELSE	0	0	0	0	0	0	0	0	0	0	0
FOR	2	1	1	0	0	0	0	0	0	0	0
L-STEP	1	1	0	0	0	0	0	0	0	0	0
R-STEP	1	1	0	0	0	0	0	0	0	0	0
UNTIL	1	0	1	0	0	0	0	0	0	0	0
FORMN	1	0	1	0	0	0	1	0	0	0	1
PM	2	0	0	0	0	0	0	0	0	0	0
M	2	2	0	0	0	0	0	0	0	0	0
D	0	0	0	0	0	0	0	0	0	0	0
A	2	0	0	0	1	1	0	0	0	0	0
HIST	2	1	0	0	0	1	0	0	0	0	0
BOOL	0	0	0	0	0	0	0	0	0	0	0
EQL	0	0	0	0	0	0	0	0	0	0	0
GEQ	0	0	0	0	0	0	0	0	0	0	0
GRT	0	0	0	0	0	0	0	0	0	0	0
LEQ	0	0	0	0	0	0	0	0	0	0	0
LES	0	0	0	0	0	0	0	0	0	0	0
NEQ	0	0	0	0	0	0	0	0	0	0	0
AND	0	0	0	0	0	0	0	0	0	0	0
OR	0	0	0	0	0	0	0	0	0	0	0
LBL,GOTO	0	0	0	0	0	0	0	0	0	0	0
PROC	0	0	0	0	0	0	0	0	0	0	0
DO.OPT	1	1	0	0	0	0	0	0	0	0	0
DO.STA	2	2	0	0	0	0	0	0	0	0	0

3.   Dynamic Raw Data

In order to weigh the static data based upon the sample test runs, two additional special compiler versions were constructed (one for AED and one for J3B) which output statistics showing how many times each compiler procedure was entered during the test run. Samples of the raw data obtained by these means is given below. In each case, only the first page of output is presented as a sample, whereas the actual output included about 8 pages for each compiler. The first example page of data is from the AED compiler, and the second is from the J3B compiler.

AED

PROCEDURE CALLS STATISTICS AT EXIT FROM PHASE3

```
NUMBER OF PROCEDURES TRACED   =     380
NUMBER OF CALLS TRACED        =   103272
NUMBER OF NORMAL RETURNS      =    95933
```

MAIN	AT LOCATION	12000 CALLED	1,RETURNED	0
AEDINIT	AT LOCATION	1B3F8 CALLED	1,RETURNED	1
AEDINI	AT LOCATION	1CC28 CALLED	1,RETURNED	1
COMARG	AT LOCATION	186B0 CALLED	4,RETURNED	4
PHASE1	AT LOCATION	20390 CALLED	1,RETURNED	1
FSET	AT LOCATION	1CC40 CALLED	1,RETURNED	1
FGSMEM	AT LOCATION	1B848 CALLED	128,RETURNED	128
FREMEM	AT LOCATION	1CB98 CALLED	128,RETURNED	128
FRET	AT LOCATION	1709C CALLED	1504,RETURNED	1504
FRED	AT LOCATION	16B18 CALLED	19,RETURNED	19
SHORTS	AT LOCATION	14D48 CALLED	1,RETURNED	1
SETSPL	AT LOCATION	23D48 CALLED	1,RETURNED	1
CEOLS	AT LOCATION	2B640 CALLED	14,RETURNED	14
COMPAR	AT LOCATION	2C038 CALLED	18,RETURNED	0
SETMMS	AT LOCATION	27A48 CALLED	1,RETURNED	1
INTPH1	AT LOCATION	256A0 CALLED	1,RETURNED	1
SETFIRS	AT LOCATION	1E634 CALLED	2,RETURNED	2
SETSTK	AT LOCATION	223F4 CALLED	1,RETURNED	1
SETRMK	AT LOCATION	2185C CALLED	1,RETURNED	1
SHRTSP	AT LOCATION	140C4 CALLED	2710,RETURNED	2710
FREC	AT LOCATION	16DDC CALLED	2713,RETURNED	2716
FREE	AT LOCATION	16EC8 CALLED	2988,RETURNED	2988
SETCHR	AT LOCATION	24360 CALLED	1,RETURNED	1
SETITM	AT LOCATION	2A8F8 CALLED	1,RETURNED	1
TARGT2	AT LOCATION	21080 CALLED	1,RETURNED	1
COPYC	AT LOCATION	180F8 CALLED	24,RETURNED	24
TARGTS	AT LOCATION	26AB0 CALLED	1,RETURNED	1
HOST1	AT LOCATION	15928 CALLED	1,RETURNED	1
ISITEM	AT LOCATION	27824 CALLED	7,RETURNED	7
STATE	AT LOCATION	2E8A8 CALLED	7,RETURNED	7
INSERT	AT LOCATION	183DC CALLED	914,RETURNED	914
RGTC	AT LOCATION	1421C CALLED	989,RETURNED	989
S.SEARC	AT LOCATION	23F30 CALLED	13,RETURNED	1
S.COPY	AT LOCATION	24100 CALLED	12,RETURNED	12
S.RNEXT	AT LOCATION	23E20 CALLED	12,RETURNED	12
S.SNEXT	AT LOCATION	23ED8 CALLED	24,RETURNED	24
PASS1	AT LOCATION	249A8 CALLED	1,RETURNED	1
INTPRE	AT LOCATION	13E74 CALLED	4,RETURNED	4
GENRB	AT LOCATION	2D190 CALLED	3,RETURNED	3
FREZ	AT LOCATION	16D1C CALLED	226,RETURNED	226
RBSET	AT LOCATION	2476C CALLED	1,RETURNED	1
INITIA	AT LOCATION	16SC8 CALLED	1,RETURNED	1
I.SEARC	AT LOCATION	14940 CALLED	958,RETURNED	221
I.COPY	AT LOCATION	1475C CALLED	653,RETURNED	653
I.RFIRS	AT LOCATION	145F8 CALLED	162,RETURNED	154
IDENT	AT LOCATION	1B088 CALLED	297,RETURNED	297

PROCEDURE CALLS STATISTICS AT EXIT FROM P.EXC$

NUMBER OF PROCEDURES TRACED = 258
NUMBER OF CALLS TRACED = 116500
NUMBER OF NORMAL RETURNS = 116461

MAIN	AT LOCATION	1464C	CALLED	1.RETURNED	0
AEDINIT	AT LOCATION	55A3A	CALLED	1.RETURNED	1
AEDINI	AT LOCATION	5C470	CALLED	1.RETURNED	1
I.COMP	AT LOCATION	2789C	CALLED	1.RETURNED	1
U.IDAT	AT LOCATION	27F4C	CALLED	1.RETURNED	1
FREMEM	AT LOCATION	56160	CALLED	1.RETURNED	1
RETMEM	AT LOCATION	19F98	CALLED	1.RETURNED	1
I.FIL	AT LOCATION	22964	CALLED	1.RETURNED	1
D.INTP	AT LOCATION	127AC	CALLED	2.RETURNED	2
GENRB	AT LOCATION	53D00	CALLED	1.RETURNED	1
U.FREZ	AT LOCATION	1A9B4	CALLED	1415.RETURNED	1415
U.FREF	AT LOCATION	1A528	CALLED	1827.RETURNED	1827
U.FHLP	AT LOCATION	19F44	CALLED	7.RETURNED	7
U.FRET	AT LOCATION	1A740	CALLED	1790.RETURNED	1790
OPNFIL	AT LOCATION	53F0C	CALLED	7.RETURNED	7
RPOPEN	AT LOCATION	50F58	CALLED	11.RETURNED	11
MAKEFL	AT LOCATION	59FF8	CALLED	11.RETURNED	11
RDWROP	AT LOCATION	5A004	CALLED	11.RETURNED	11
OPNDSK	AT LOCATION	599AC	CALLED	11.RETURNED	11
COPYC	AT LOCATION	52220	CALLED	12.RETURNED	12
OPNCMS	AT LOCATION	5C4F0	CALLED	11.RETURNED	11
U.FREC	AT LOCATION	1A46C	CALLED	20.RETURNED	20
SFTFMT	AT LOCATION	5A22C	CALLED	11.RETURNED	11
PUSH.FI	AT LOCATION	53F64	CALLED	7.RETURNED	7
SPSITM	AT LOCATION	54900	CALLED	14.RETURNED	14
F.OPTS	AT LOCATION	2730C	CALLED	1.RETURNED	1
COMARG	AT LOCATION	55F90	CALLED	1.RETURNED	1
U.SCDA	AT LOCATION	1A004	CALLED	1.RETURNED	1
I.PROG	AT LOCATION	27724	CALLED	1.RETURNED	1
I.LEX	AT LOCATION	15A84	CALLED	1.RETURNED	1
GETHMS	AT LOCATION	535D8	CALLED	2.RETURNED	2
I.PRG2	AT LOCATION	15F7C	CALLED	1.RETURNED	1
F.TITLE	AT LOCATION	14260	CALLED	1.RETURNED	1
LINES.	AT LOCATION	140FC	CALLED	1.RETURNED	1
SFTOUT	AT LOCATION	536B0	CALLED	1.RETURNED	1
I.TABS	AT LOCATION	1B164	CALLED	1.RETURNED	1
I.COND	AT LOCATION	27954	CALLED	9.RETURNED	9
L.CSYM	AT LOCATION	2790C	CALLED	1.RETURNED	1
I.LODS	AT LOCATION	16DE4	CALLED	62.RETURNED	62
I.UHAS	AT LOCATION	16CF4	CALLED	1199.RETURNED	1199
L.FNDS	AT LOCATION	16474	CALLED	3106.RETURNED	3106

MISSION
of
Rome Air Development Center

RADC is the principal AFSC organization charged with planning and executing the USAF exploratory and advanced development programs for electromagnetic intelligence techniques, reliability and compatibility techniques for electronic systems, electromagnetic transmission and reception, ground based surveillance, ground communications, information displays and information processing. This Center provides technical or management assistance in support of studies, analyses, development planning activities, acquisition, test, evaluation, modification, and operation of aerospace systems and related equipment.

Source AFSCR 23-50, 11 May 70