HALSTEAD'S COMPLEXITY MEASURE ON PASCAL PROGRAMS

by

SHOU-NAN WANG

B.S. Tamkang College, 1976

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1980

Approved by :

William J. Hankley

Acknowledgements

The writer is indebted to many persons whose assistance made possible the completion of this report.

Much appreciation is due William J. Hankley, the writer's major advisor. The help and encouragement from him made this report possible. Also, I would like to thank David A. Gustafson for his ideas contributed to this report.

I appreciate Kam Mok's help in the development of the plotting program.

Finally, I would like to thank Chun Mei Liou for her encouragement throughout this project and her assistance in preparing this document.

ILLEGIBLE DOCUMENT

THE FOLLOWING DOCUMENT(S) IS OF POOR LEGIBILITY IN THE ORIGINAL

THIS IS THE BEST COPY AVAILABLE

THIS BOOK CONTAINS NUMEROUS PAGES WITH DIAGRAMS THAT ARE CROOKED COMPARED TO THE REST OF THE INFORMATION ON THE PAGE.

THIS IS AS RECEIVED FROM CUSTOMER.

TABLE OF CONTENTS

LIST OF FIGURES

TABLE

CHAPTER I

INTRODUCTION

I.1 Introduction

There has always been a need to evaluate the quality and the complexity, as well as other properties of computer programs. This evaluation is beginning to be recognized in the field of computer science, and the mechanical measurement of programs is now a part of software engineering.

Programs used to be measured by the number of lines of code, which only roughly gives the size of a program, not the complexity of, nor the effort spent on a program. McCabe [McCabe 76] proposed that the complexity of a program be measured by the number of predicates in it. It makes sense that, if a program has more predicates, it is more complex to understand.

Maurice Halstead [Halstead 77] approached the problem differently. He developed and refined a theory called Software Science. It is an empirical method to measure the statistical properties of programs. Several interesting properties of programs can be obtained based upon counts of operators and operands in them. The four parameters used are n_1 (number of unique operators), n_2 (number of unique operands), N_1 (total number of operators), and N_2 (total number of operands). An operator typically represents an

operation, e.g., "+", "-", or a procedure name. Operands represent the objects; they are constants and variables. The properties that can be derived are program length, volume, level, etc. These terms will be explained in Chapter 2.

Halstead initiated a series of studies of programs to develope his theory several years ago. Later, in 1976, he published a book [Halstead 77], in which he put together the results from his studies, along with some justifications, to try to explain how and why his method works. Most of the studies were done for Assembly, COBOL, FORTRAN, and PL/I programs. None was made for programs written in PASCAL, which is now a popular language. Also, declaration and I/O statements were ignored in his model, whereas in PASCAL the declarations seem to comprise an integral part of a definition of the program.

I.2 Motivation for this Report

The motivation for this report was to measure the properties of PASCAL and to incorporate declarations in Halstead's model. This report presents a design and implementation of extensions to Hartman's S-PASCAL compiler [Hartmann 77] to measure an approximation to Halstead's parameters. In several experiments Halstead's measures are repeated for each procedure declaration, each procedure body, each procedure (declaration + body), all declarations, all bodies, and the

whole program. Results are reported for measurements on several programs.

Chapter II will summarize Halstead's theory. Results and discussions are in Chapter III. Chapter IV will give the implementation details.

CHAPTER II

PROGRAM MEASUREMENTS

II.1 Basic Parameters

Halstead based his theory on four measures of algorithms :

n_1 = no. of unique operators in the implementation

n_2 = no. of unique operands in the implementation

N_1 = total no. of all operators in the implementation

N_2 = total no. of all operands in the implementation.

An operator is a symbol that defines an operation or a pair of symbols that serve to group things together. It is obvious that the assignment symbol ":=", the arithmetic operation symbols "+", "-", "*", "/", the comparison symbols "=", "<", ">", etc. are operators. Since a procedure (subroutine) defines an operation, the referencing of the procedure name is an operator reference. Also, "()", "[]", "begin-end" are each counted as a single grouping operator. The arguments for this choice are: first, that the two components are statistically completely dependent, and second, that the language syntax can be so modified as to use only one of the symbol pair without changing other operator and operand counts. On page 10, Table 1 contains the definition of all operators in PASCAL.

The definition for operands is more obvious. Objects that are manipulated by operations are operands. The variable names, symbolic

constants, enumeration names, literal constants, etc. are operands. One problem in counting operands is how to count constants; this issue will be explored later in Chapter III.

The vocabulary n of an implementation is the total number of unique operators and operands in the implementation. The total number of occurrences of both operators and operands is the program length N of the implementation. The equations are:

$$n = n_1 + n_2$$
$$N = N_1 + N_2 \quad .$$

Based upon statistical arguments, Halstead justified an estimator of program length $\hat{N}$ as:

$$\hat{N} = n_1 \log_2 n_2 + n_2 \log_2 n_2 \quad .$$

II.2 Derived Equations

The volume V of a program is defined as:

$$V = N \log_2 n$$

This is the length in bits of the most compact encoding of the program. The intuition is that, for an implementation of vocabulary n, at least $\log_2 n$ bits will be needed to represent each

operator/operand. The volume is thus the minimal bit length per token multiplied by the total occurrences N.

The volume of a program changes if it is translated into another language. When the program is translated into a language of the highest level (for the particular application), it would be just a procedure call. The volume of such a program is called the potential volume V* and is defined as

$$\begin{aligned} V^* &= n^* \log_2 n^* \\ &= (n_1^* + n_2^*) \log_2 (n_1^* + n_2^*) \\ &= (2 + n_2^*) \log_2 (2 + n_2^*) \end{aligned}$$

where n_1^* and n_2^* are the potential operator count and the potential operand count respectively. In a procedure call only two operators are needed, namely, the procedure name and a grouping operator to group together the operands (arguments). n_2^* is the minimum number of arguments needed for the call. However, there are significant difficulties in determining n_2^*. For example, consider a module that works as a symbol table handler for a compiler. If we are to use it, we must supply the following arguments to the module: identifier length, character set for the identifiers, symbol table size, the identifier that is to be put in or searched for, and probably the scope depth, etc. All these arguments will be the potential operands of this module. Whereas in practice, some of these arguments are not obvious or are built into the module.

For a given algorithm with volume V and potential volume V^* the program level L of an implementation is defined as

$$L = V^*/V \quad \text{or} \quad V^* = LV \ .$$

For a fixed algorithm, as the program volume V goes up, the program level goes down. The level L is 1 when the volume is the potential volume. An approximation to the program level is

$$\hat{L} = (2/n_1)\ (n_2/N_2) \ .$$

The explanation is that, the lower the level, the more the number of operators, so $L \sim n_1^*/n_1 = 2/n_1$; and if an operand is repeated many times, the level is low, so $L \sim n_2/N_2$.

Halstead suggested that the product, $\lambda = LV^*$, called the language level, remains constant for any one language. Thus, in a given language algorithms with larger potential volumes would have lower program levels. The histogram of the language levels of several languages shown in his book is reproduced as Figure 1 on page 11.

II.3 Program Impurities

Halstead argued that the impurities in programs would affect the measurement. The impurities in a program do not mean that the program is good or bad. His observation was that a program written by a

sophisticated (experienced) programmer has fewer impurities. In the studies of the report, all impurities are ignored. For reference, Halstead's listing of impurities [Halstead 77] [Ottenstein 76] are summarized here:

1. Complementary operations - use of two complementary operators to the same operand, eg. A = B + C - C.

2. Ambiguous operands - use of the same operand to refer to different things at different places in the program. This problem occurs when we try to use the same operand over and over again to save memory space.

3. Synonomous operands - the opposite to the above impurity. The problem arises when we use two operands for the same thing.

4. Commom subexpressions - Whenever a subexpression (segment of code) is used more than once, an operand (procedure) should be assigned to (defined for) it.

5. Unwarranted assignments. An operand is assigned a value and then used only once. The operand should not be created in the first place.

6. Unfactored expressions. An example will make it clear: use

```
X = A ↑ 2 + 2 * A * B + B ↑ 2
```

instead of

```
X = (A + B) ↑ 2 .
```

7. The situation when a subscripted variable is used several times to refer to the same object. If A[i] is referenced in the same context several times with i unchanged; after the first occurrence, it should only be counted as one operand. However, in the first reference of A[i], there are two operands, A and i, and one operator "[]".

operators in PASCAL

array	".."
record	","
set	";"
const	":"
var	"()"
type	"[]" used for set
packed	"[]" used for array
univ	"@" or "^"
case	mod
not	"*"
"="	"/"
"<"	unary operator "+"
">"	unary operator "-"
"<>"	arith operator "+"
"<="	arith operator "-"
">="	bool operator "and"
begin-end	bool operator "or"
while-do	set operator "or"/"+"
repeat-until	"in"
with-do	"."
for-to, for-downto	":="
if-then, if-then-else	div

* note 1 : The reserved word "forward" is not counted as an operator because it is primarily used for implementation consideration.

* note 2 : "of" is not counted because it is used with "set" or "array" merely to make it English like.

* note 3 : the reserved word "program" is not counted in the measure since each program must have one and the word contributes nothing to the program.

* note 4 : "nil" is treated as a value rather than a reserved word.

TABLE 1 Operators in PASCAL

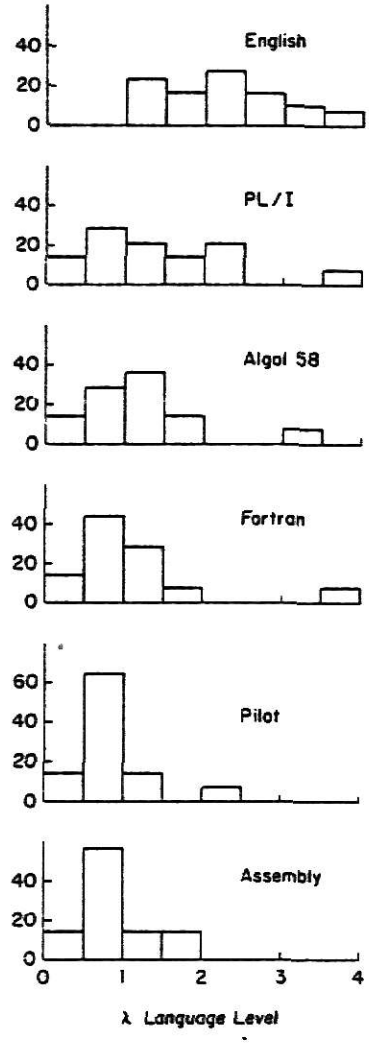


FIGURE 1 Frequency Distributions of λ

CHAPTER III

EXPERIMENTS AND RESULTS

III.1 General

This chapter presents measurements of $\hat{N}/N$ and λ for twelve different programs: the nine passes of the K-State implementation PASCAL compiler, KPASS1 to KPASS 9, the first two passes of Hartmann's compiler, HPASS1 and HPASS2, and the cross reference program XREF. The results are summarized in Fig. 2-16(page 23-37) at the end of this chapter. Only the values of the ratio of $\hat{N}$ to N and part of the values of the language level computed are shown in these figures. Before analyzing these results, two measurement issues are discussed: where to measure declarations, and how to treat constants.

III.2 On Measurement of Declarations

There is no doubt that constant names (symbolic constants) play an important role in the programming language PASCAL. Since a constant name is declared in the constant declaration part to equate itself to a literal constant, the question arises as to how to count each constant name. Should it be treated as an alias of the literal constant it represents, or better as an object by itself?

It is very easy to think that a constant name and the corresponding literal constant are the same and that the former merely provides a different way to represent the latter. In some cases it is true. An example is that we use the name "eol" to represent the ASCII code x'05' in a PASCAL program. However, in a lot of other cases, it is not so. As in the implementation of a compiler, we usually use integers for tokens; However, we may just as well use character strings for them. If a constant name is counted as an alias of the literal constant it represents, then the count for n_2 will probably vary between the two representations of tokens. The counts should not be different, since the logic of the program is not changed by changing the machine representations of the tokens. It is obvious that, in this example, the constant names are different from the literal constants represented by them. On the other hand, it seems so obvious that they should be treated differently. A token is a token by itself. We do not think it as an integer when it is encoded as an integer, and we do not think it as a character string when it is encoded as a character string.

From the above discussion, we know that the constant declaration part serves as a mapping function; it links the constant names with their represented literal constants. The only reason for its existence is that the machine does not know how a constant name is represented. In order for the compiler to generate proper codes, it must know the primitive machine types of the constant names. Sometimes a constant name is used as an alias of some literal constant; sometimes it is not. To make the correct judgement, one

must analyze the program to determine which case is applicable to a constant name. This approach is not used in this project because of the enormous amount of work involved. Rather, we performed repeated experiments, alternately counting all constant names and the literal constants which they represent as the same operand, or as different ones. Also, the counts were repeated both with and without constant declarations (i.e., there are four different cases for each experiment).

As to measuring declarations, there are two different views. In a computer everything is represented in bits. One bit is not different from another. But, according to the rules built into compiled program codes, the machine knows when to use the proper interpretation for the value saved in each memory unit. A computer usually knows four interpretations: integers, floating-point numbers, character strings, and bits. When we are programming, not every object in the real environment is of these primitive types. We have to map the objects in our minds to the primitive representations available. This procedure is exactly what we do in the declaration part of a program. Of course, declarations also serve to avoid some errors, such as referencing a misspelled variable in FORTRAN.

In PASCAL a type declaration is used to link an abstract object type and its machine representation together. A variable declaration links an object and its representation; a constant declaration links

an object and its represented value. Since they merely set up an agreement between the user and the machine and do not contribute to the implementation of an algorithm, they should be ignored.

The alternative view is the following one: The Halstead measures for a particular program should reflect the characteristics of that program in the language in which it is written, not the characteristics of the underlying algorithm and not the characteristics of an underlying lower level target language. When a PASCAL program is compiled, the information from the declarations is distributed to the operations, whence a generic operator like "+" may become an "add-real" in the target language. Now, a target program (with operators like "add-real") will not have the same $n_1 \ldots N_2$ counts as its source version, even though it expresses the same algorithm. The body of a PASCAL source program (without its declarations) will have neither the same information as the full program, nor the same $n_1 \ldots N_2$ counts. If declarations were optional (for example, as are some declarations in BASIC and in FORTRAN), then we could conclude that they would be merely mental constructs which aid the programmers in understanding a program, but they would not be an integral part of the expression of the algorithm. We view comments in this light. On the other hand, since declarations are essential in PASCAL and since they contribute to the expressive power of PASCAL (by allowing programmer defined types), they are an integral part of the expression of algorithms in PASCAL. Hence the counts for PASCAL programs should includ the declarations.

Yet, we observe the "const" declarations serve a double purpose. First, they declare the name and the type of a symbolic constant. Second, they assign a literal value to the constant. We believe that the declared constants are truely parameters of a program in the broadest sense. That is, users may invoke instances of a program with different parameter values as may be desired-----even though most constant values will not be changed by users of a program. The assignment of values to symbolic constants may be viewed as part of the invocation of the program and not part of its implementation expression. Therefore, we conclude that--at minimum--the assignment part of constant declarations should not be counted as part of the $n_1 \ldots N_2$ measures. Rather, a declaration like "const A = 1" should be counted as "const A : integer". This was not done in the experiments, but sets of measurements were taken completely ignoring the const declarations.

The counts were made both with and without counting declarations.

III.3 Estimated Program Length

The primary measurement used in this study is program length. The reason for this is that it is hard to justify other parameters of Halstead's complexity measure. Yet it is very easy to see how good $\hat{N}$ is as an estimation to the real program length N. AS for the language

level λ, we can use some intuition to judge the result. The expectation is that the language level is not high.

For each program measured, the calculation is done both with and without constant declarations. In each of the figures 2-13 shown on page 23-34, if there are two numbers in each box, the upper one is the one obtained with constant declarations, the other one without. When there is only one number in a box, it means that the two ratios are the same.

As each figure shows, the counts were made for each procedure declaration, each procedure body, each procedure (declaration + body), all declarations, all bodies, and the whole program. There are two ways to count constants, i.e., (1) a constant name is counted as an alias of its corresponding literal constant, (2) a constant name is treated as an operand different from the literal constant it represents. Not all values of the language level computed are listed because only the four shown seem to be meaningful after some analysis.

In each figure, the ratios shown in the two columns titled "each decl" and "all decl's" are very high. This fact implies that the estimated value $\hat{N}$ is very inaccurate. This assertion is justified, for Halstead's measure is based on algorithms only, where a declaration only serves as a mapping function. It alone does not make an algorithm.

The ratios of $\hat{N}/N$ under the titles "each decl" and "each proc" are also very high. The reason for this is that a procedure is not a module. A module is a stand-alone program, which constitutes an algorithm by itself. A procedure usually is only a part of an algorithm; thus, the measure made on a partial algorithm is not accurate.

The rest of the ratios $\hat{N}/N$ is put together in figure 14 and plotted in figure 15. The mean for each category is computed. To see how close $\hat{N}$ is to N in each category, we compute the values in the row titled "closeness" using the equation $\frac{\sum_{i}^{m}(\hat{N}i/Ni - 1)}{m}$, where m = no of programs. The smaller the value, the better the approximation $\hat{N}$ is to N.

The program KPASS5 is a procedure that prints out messages for errors detected in the first four passes. In this program most of the operands are constant names, which are the error code names, and the character string literals, which are error message texts. The former operands appear only twice, once in the declaration, the other in the case statements as the case labels. Because most operands appear only once or twice in this program, the program length is over estimated by Halstead's method when we treat constant names differently from literal constants. The estimation of N is not bad when we count constant names as aliases of literal constants. Because of the over estimation of N for program KPASS5, we also count the mean and the closeness of $\hat{N}/N$ for each counting method without including KPASS5. The pursuit and the analysis as to why the estimation of N for program

KPASS5 is high and as to how Halstead's estimating equation can be modified to fit the measuring of such programs are beyond the scope of this report. When counting constant names as operands by themselves, the values of the language level computed for KPASS5 are also very high, compared to those obtained from other programs. The reason might have been that there are not many distinct operators referenced in the program KPASS5.

From figures 14 and 15(page 35-36) it is obvious that the results obtained based on counting methods (1), (4), and (6) are the best. The fact that two of the above three counting methods include declarations seems to contradict the first view (ignoring declarations) in section 2. As stated earlier, the manner as to how to treat declarations is an open question. The arguments in section 2 were just some intuitive remarks, which were not substantiated by any theory or fact, although they were by no means proved invalid. Since the prime purpose of this report is to provide a tool for Halstead's complexity measure, no attempt is made to try to expain why the results that include declarations are better.

III.4 The Language Level of PASCAL

The values of the language level for PASCAL obtained by using the three counting methods mentioned above are plotted in figure 16(page 37). As can be seen from the figure, except for program KPASS5, all λ's are within the range of 2. Compared to the distributions of the

λ's of other languages shown in figure 1(page 11), the distribution of the language level of PASCAL obtained in this report suggests that the K-State implementation of PASCAL is not more expressive than the other programming languages in figure 1. That is, it would take as many tokens to "speak" in PASCAL as in PL/I. (PASCAL advocates would argue that PASCAL has other semantic strength over PL/I.)

Some explanation about the language level of PASCAL follows. First, there are not many supporting functions or procedures in PASCAL, and there is no subprogram library support in the original K-State implementation of PASCAL; hence, each program must repeat the same set of procedures necessary for its operation. Had the programs been able to reference these procedures without having to define them, the language level probably will be higher. Second, PASCAL was designed to be a small and precise programming language. In order to be precise, we have to use more operations to achieve what can be done in other programming languages. The following code segment illustrates this point:

```
var  A : real;
     I : integer;
begin
   A := conv(I)
end;
```

where in PL/I, the conversion is done implicitly by saying "A=I". Since the language is small, we have to define a lot of functions by our own. As an example, if A and B are two arrays of the same type

and we are to equate B to A, the complexity of the implementation in PASCAL compared to that in PL/I is quite obvious :

PL/I	PASCAL
B = A;	for i:= low_bound to high_bound do B[i] := A[i];

However, there seems to be some powerful or good structures in PASCAL that are not revealed in Halstead's measurements, such as the with-statement and the variant record construct. Further study will be needed to incorporate these considerations into the model.

III.5 Validity of Program Measure

As noted in Chapter II, all questions of program impurities were ignored, whereas Halstead warns that impurities may invalidate the prediction equations. One seperate and indirect measure of the program purity is to examine the program's adherence to Zipf's law of language [Cherry 66]. This predicts that frequency of occurrence of each token (total number of occurrence) versus the rank order of occurrence of each token should be a linear curve plotted on log-log scale. (Halstead's explanation of the length equation also implies that Zipf's law should hold seperately for operators and operands.)

Plots of the frequency-rank distribution for operator tokens and operand tokens for some programs measured are shown in Figures 17-26

on page 38-47. Deviation of these curves from the ideal linear forms may indicate the extent to which the predictor equation may give spurious results. One pattern which is evident in these curves is the dropping off of the low frequency tokens. That is caused partially by the single occurrence of many tokens in the program. These are constants or variables which are declared or initailized just once and not again ever used.

III.6 Discussion

The decision as to how to count constant names can be avoided if, as in the language ADA, we can read the enumeration names directly. If an object is to be used as the alias of a literal constant, then define the former as a constant name; if it is to be used as a distinct operand, then define it in an enumeration. Having been able to do so, we could count all constant names as the aliases of some literal constants.

We believe that more time has to be spent to gain the insight into the nature of algorithms and programs. After knowing more about the nature of them, we would be able to adjust Halstead's model such that it will predict the complexity of programs, as well as the underline algorithms, more accurately.

	each decl	each body	each* proc	all decl's	all bodies	whole pgm
(1)	1.87 1.96	1.60	1.70 1.71	2.00 2.11	1.06	0.89 0.94
(2)	2.33 1.97	1.62	1.84 1.74	2.49 2.13	1.12	1.03 0.98

(A) $\hat{N}$ / N

	all bodies	whole pgm
(1)	1.09	0.78 0.85
(2)	1.22	1.09 0.94

(B) Language Level λ

* 'each proc' includes both the declaration and the body of a procedure

NOTE: (1) means 'count a constant name as an alias of its value'
(2) means 'count a constant name as an distinct operand'

FIGURE 2 Measured Results of Program XREF

	each decl	each body	each* proc	all decl's	all bodies	whole pgm
(1)	1.56 2.02	1.46	1.50 1.58	1.71 2.24	0.98	0.81 0.90
(2)	2.57 2.02	1.51	1.81 1.62	2.78 2.26	1.16	1.12 1.05

(A) $\hat{N}$ / N

	all bodies	whole pgm
(1)	0.95	0.67 0.77
(2)	1.35	1.34 1.09

(B) Language Level λ

* 'each proc' includes both the declaration and the body of a procedure

NOTE: (1) means 'count a constant name as an alias of its value'
(2) means 'count a constant name as an distinct operand'

FIGURE 3 Measured Results of Program KPASS1

	each decl	each body	each* proc	all decl's	all bodies	whole pgm
(1)	1.69 2.21	1.62	1.64 1.72	1.97 2.64	1.09	0.82 0.94
(2)	2.96 2.22	1.71	2.09 1.80	3.32 2.65	1.43	1.25 1.21

(A) $\hat{N}$ / N

	all bodies	whole pgm
(1)	1.09	0.72 0.88
(2)	1.91	1.73 1.52

(B) Language Level λ

* 'each proc' includes both the declaration and the body of a procedure

NOTE: (1) means 'count a constant name as an alias of its value'
(2) means 'count a constant name as an distinct operand'

FIGURE 4 Measured Results of Program KPASS2

	each decl	each body	each* proc	all decl's	all bodies	whole pgm
(1)	1.54 2.10	1.70	1.61 1.83	1.89 2.67	0.80	0.63 0.71
(2)	2.68 2.12	1.92	2.00 2.03	3.10 2.69	1.21	1.02 1.05

(A) $\hat{N} / N$

	all bodies	whole pgm
(1)	0.27	0.22 0.26
(2)	0.66	0.62 0.59

(B) Language Level λ

* 'each proc' includes both the declaration and the body of a procedure

NOTE: (1) means 'count a constant name as an alias of its value'
(2) means 'count a constant name as an distinct operand'

FIGURE 5 Measured Results of Program KPASS3

	each decl	each body	each* proc	all decl's	all bodies	whole pgm
(1)	1.42 2.04	1.64	1.54 1.80	1.70 2.50	0.83	0.64 0.73
(2)	2.66 2.07	1.98	2.06 2.10	3.00 2.54	1.38	1.11 1.16

(A) (A) $\hat{N}$ / N

	all bodies	whole pgm
(1)	0.30	0.24 0.28
(2)	0.91	0.78 0.76

(B) Language Level λ

* 'each proc' includes both the declaration and the body of a procedure

NOTE: (1) means 'count a constant name as an alias of its value'
(2) means 'count a constant name as an distinct operand'

FIGURE 6 Measured Results of Program KPASS4

	each decl	each body	each* proc	all decl's	all bodies	whole pgm
(1)	1.60 2.09	1.72	1.41 1.74	0.93 2.00	1.35	0.86 1.21
(2)	2.98 2.09	2.59	2.17 2.49	2.66 2.00	2.46	1.59 2.17

(A) $\hat{N}$ / N

	all bodies	whole pgm
(1)	3.05	1.42 2.21
(2)	10.17	4.96 7.22

(B) Language Level λ

* 'each proc' includes both the declaration and the body of a procedure

NOTE: (1) means 'count a constant name as an alias of its value'
(2) means 'count a constant name as an distinct operand'

FIGURE 7 Measured Results of Program KPASS5

	each decl	each body	each# proc	all decl's	all bodies	whole pgm
(1)	1.48 1.95	1.28	1.33 1.38	1.84 2.43	0.53	0.49 0.49
(2)	2.33 1.96	1.47	1.65 1.55	2.76 2.46	0.73	0.66 0.66

(A) $\hat{N}$ / N

	all bodies	whole pgm
(1)	0.21	0.17 0.19
(2)	0.42	0.39 0.37

(B) Language Level λ

'each proc' includes both the declaration and the body of a procedure

NOTE: (1) means 'count a constant name as an alias of its value'
(2) means 'count a constant name as an distinct operand'

FIGURE 8 Measured Results of Program KPASS6

	each decl	each body	each* proc	all decl's	all bodies	whole pgm
(1)	1.51 1.96	1.33	1.41 1.48	1.90 2.53	0.59	0.31 0.55
(2)	2.44 1.98	1.52	1.79 1.65	2.94 2.57	0.87	0.78 0.79

(A) $\hat{N}$ / N

	all bodies	whole pgm
(1)	0.16	0.15 0.16
(2)	0.38	0.38 0.36

(B) Language Level λ

* 'each proc' includes both the declaration and the body of a procedure

NOTE: (1) means 'count a constant name as an alias of its value'
(2) means 'count a constant name as an distinct operand'

FIGURE 9 Measured Results of Program KPASS7

	each decl	each body	each* proc	all decl's	all bodies	whole pgm
(1)	1.83 2.03	1.42	1.51 1.56	2.03 2.54	0.82	0.64 0.71
(2)	2.83 2.05	1.61	1.97 1.71	3.24 2.57	1.12	1.03 0.95

(A) $\hat{N}$ / N

	all bodies	whole pgm
(1)	0.41	0.31 0.34
(2)	0.79	0.84 0.65

(B) Language Level λ

* 'each proc' includes both the declaration and the body of a procedure

NOTE: (1) means 'count a constant name as an alias of its value'
(2) means 'count a constant name as an distinct operand'

FIGURE 10 Measured Results of Program KPASS8

	each decl	each body	each* proc	all decl's	all bodies	whole pgm
(1)	1.99 2.02	1.06	1.26 1.21	2.41 2.56	0.82	0.69 0.75
(2)	2.74 2.02	1.13	1.49 1.28	3.26 2.60	0.99	0.96 0.88

(A) $\hat{N}$ / N

	all bodies	whole pgm
(1)	0.58	0.45 0.50
(2)	0.85	0.90 0.72

(B) Language Level λ

* 'each proc' includes both the declaration and the body of a procedure

NOTE: (1) means 'count a constant name as an alias of its value'
(2) means 'count a constant name as an distinct operand'

FIGURE 11 Measured Results of Program KPASS9

	each decl	each body	each* proc	all decl's	all bodies	whole pgm
(1)	1.64 1.94	1.43	1.55 1.60	1.68 1.97	0.97	0.83 0.88
(2)	2.49 1.96	1.50	1.84 1.66	2.59 2.02	1.14	1.14 1.03

(A) $\hat{N}$ / N

	all bodies	whole pgm
(1)	1.03	0.61 0.65
(2)	1.46	1.22 0.92

(B) Language Level λ

* 'each proc' includes both the declaration and the body of a procedure

NOTE: (1) means 'count a constant name as an alias of its value'
(2) means 'count a constant name as an distinct operand'

FIGURE 12 Measured Results of Program HPASS1

	each decl	each body	each* proc	all decl's	all bodies	whole pgm
(1)	1.63 2.29	1.58	1.60 1.72	1.79 2.49	0.88	0.63 0.74
(2)	2.80 2.29	1.67	2.03 1.79	3.03 2.50	1.23	1.03 1.02

(A) $\hat{N}$ / N

	all bodies	whole pgm
(1)	0.68	0.40 0.53
(2)	1.42	1.18 1.07

(B) Language Level λ

* 'each proc' includes both the declaration and the body of a procedure

NOTE: (1) means 'count a constant name as an alias of its value'
(2) means 'count a constant name as an distinct operand'

FIGURE 13 Measured Results of Program HPASS2

Summary of Some $\hat{N}/N$ values

PROGRAM	(1)	(2)	(3)	(4)	(5)	(6)
XREF	1.06	1.12	0.89	1.03	0.94	0.98
KPASS1	0.98	1.16	0.81	1.12	0.90	1.05
KPASS2	1.09	1.43	0.82	1.25	0.94	1.21
KPASS3	0.80	1.21	0.63	1.02	0.71	1.05
KPASS4	0.83	1.38	0.64	1.11	0.73	1.16
KPASS5	1.35	2.46	0.86	1.59	1.21	2.17
KPASS6	0.53	0.73	0.49	0.66	0.49	0.66
KPASS7	0.59	0.87	0.31	0.78	0.55	0.79
KPASS8	0.82	1.12	0.64	1.03	0.71	0.95
KPASS9	0.82	0.99	0.69	0.96	0.75	0.88
HPASS1	0.97	1.14	0.83	1.14	0.88	1.03
HPASS2	0.88	1.23	0.63	1.03	0.74	1.02
mean*	0.93	1.33	0.70	1.10	0.83	1.16
variance*	0.17	0.31	0.13	0.16	0.15	0.24
closeness*	0.19	0.31	0.31	0.16	0.24	0.20
mean&	0.89	1.24	0.69	1.06	0.80	1.08
variance&	0.14	0.18	0.14	0.12	0.13	0.14
closeness&	0.18	0.20	0.33	0.12	0.24	0.12

(1) all bodies, constant names = literal constants
(2) all bodies, constant names <> literal constants
(3) whole program, constant names = literal constants, counting constant declarations
(4) whole program, constant names <> literal constants, counting constant declarations
(5) whole program, constant names = literal constants, not counting constant declarations
(6) whole program, constant names <> literal constants, not counting constant declarations

closeness -- defined by the equation $\frac{\sum_{1}^{m}(\hat{N}i/Ni - 1)}{m}$

the smaller the value,
the better the approximation

* calculation including KPASS5
& calculation not including KPASS5

FIGURE 14 Summary of some $\hat{N}/N$ values

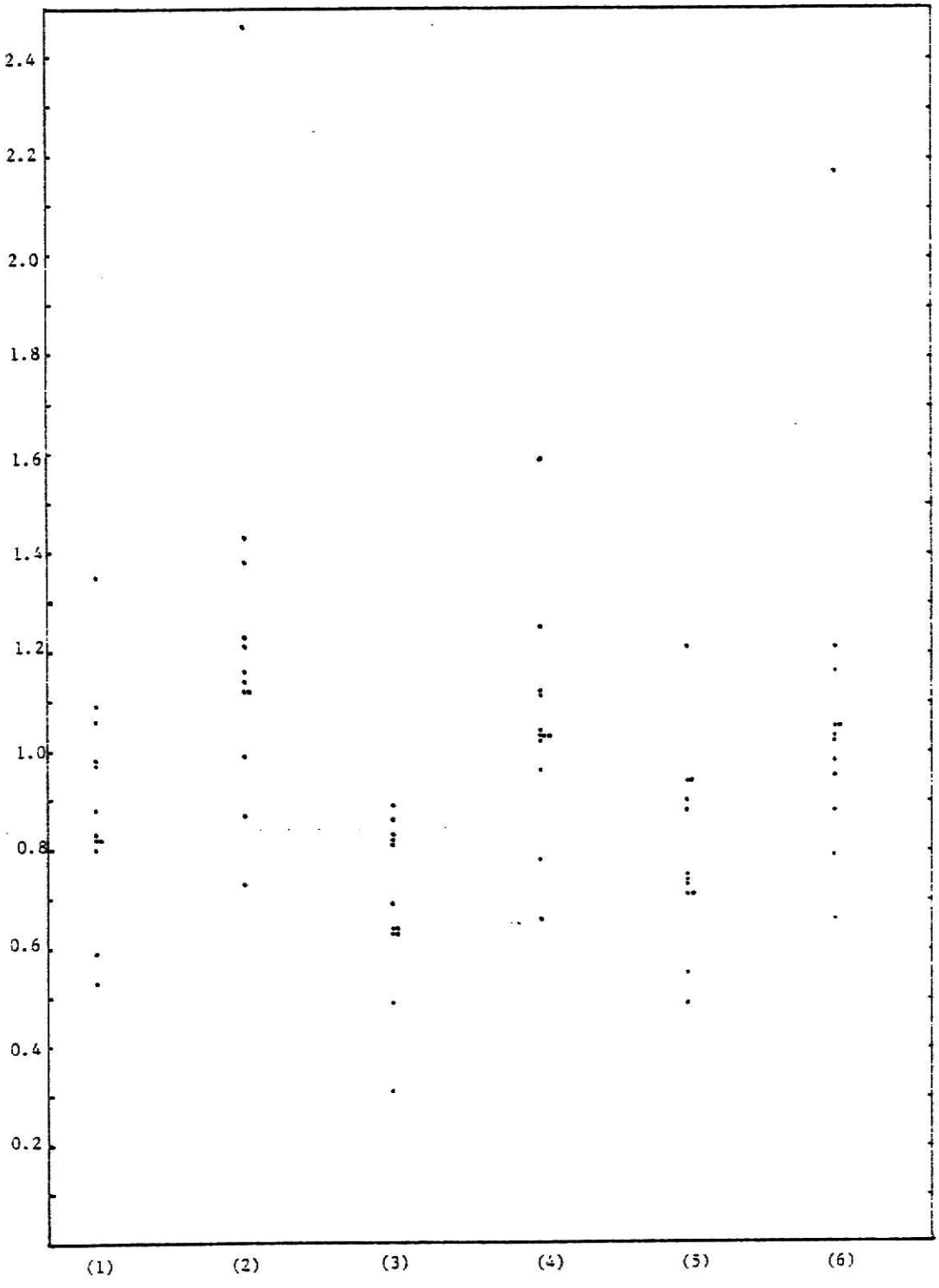
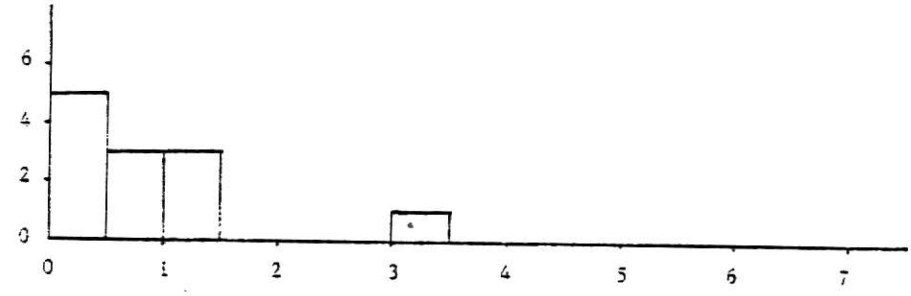
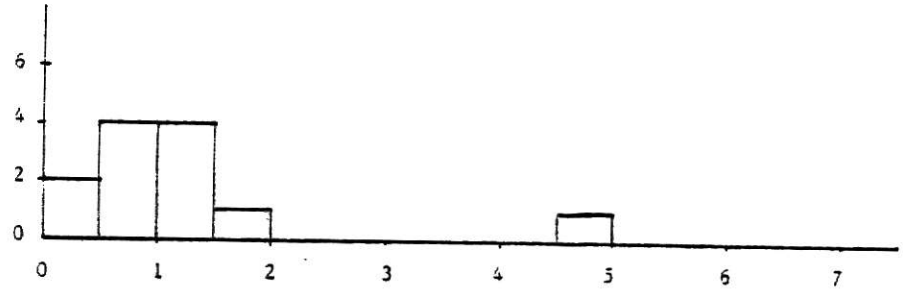


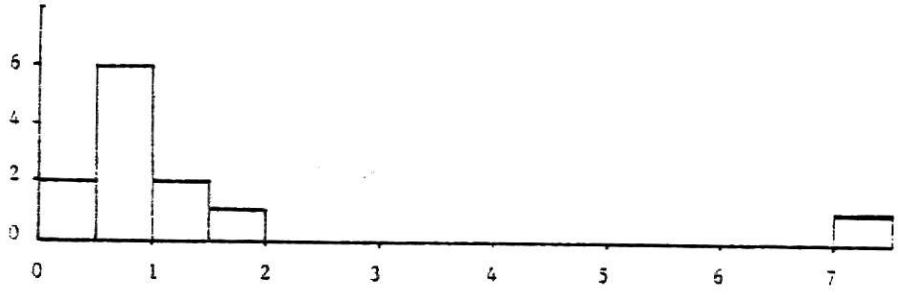
FIGURE 15 Distribution of $\hat{N}/N$



(A) obtained from category (1) of FIGURES 2(B)-13(B)



(B) obtained from category (4) of FIGURES 2(B)-13(B)



(C) obtained from category (6) of FIGURES 2(B)-13(B)

FIGURE 16 Frequency Distributions of λ Experimented

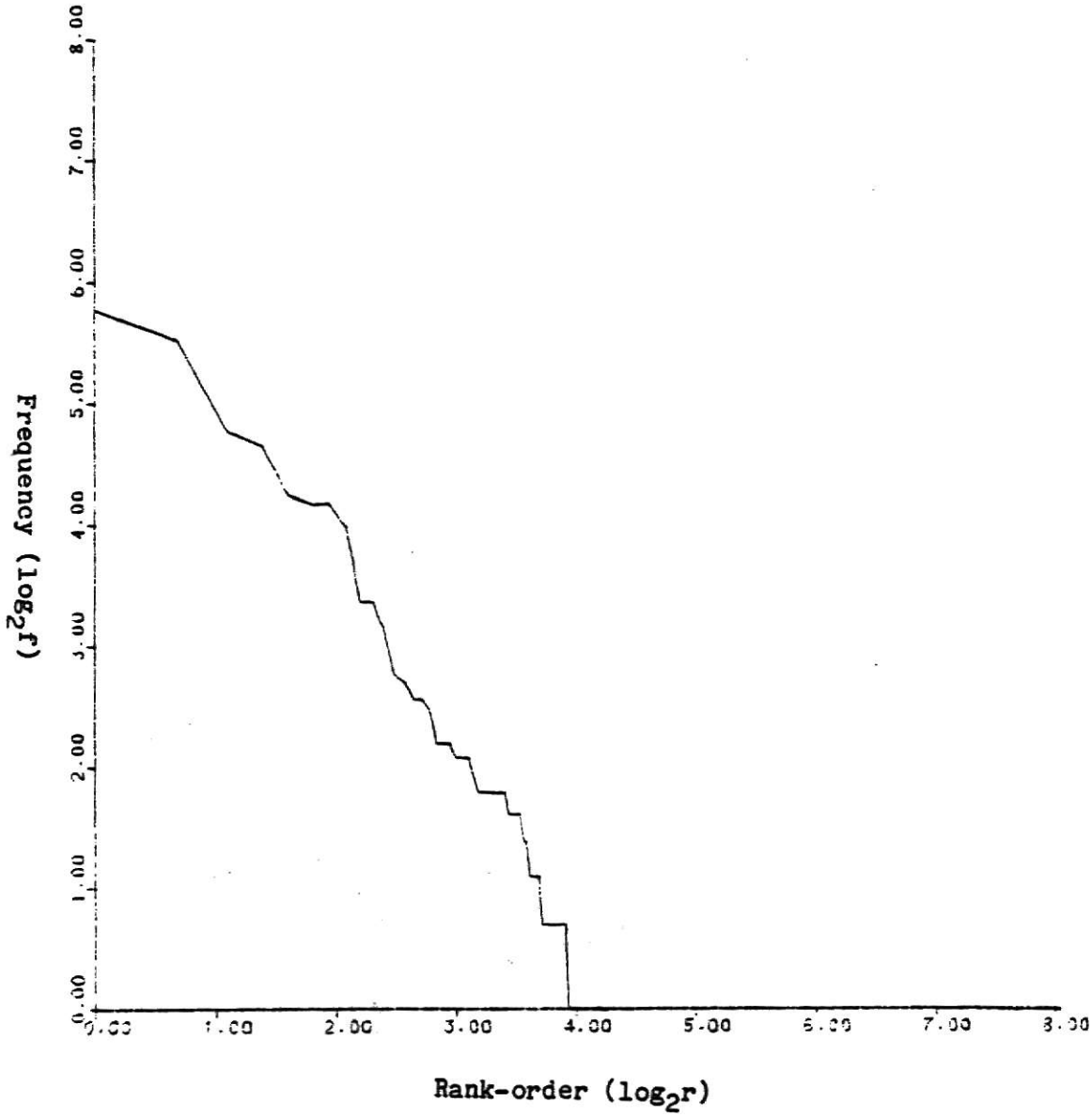


FIGURE 17 Rank-Frequency Distribution of Operators in the Bodies of the Program XREF

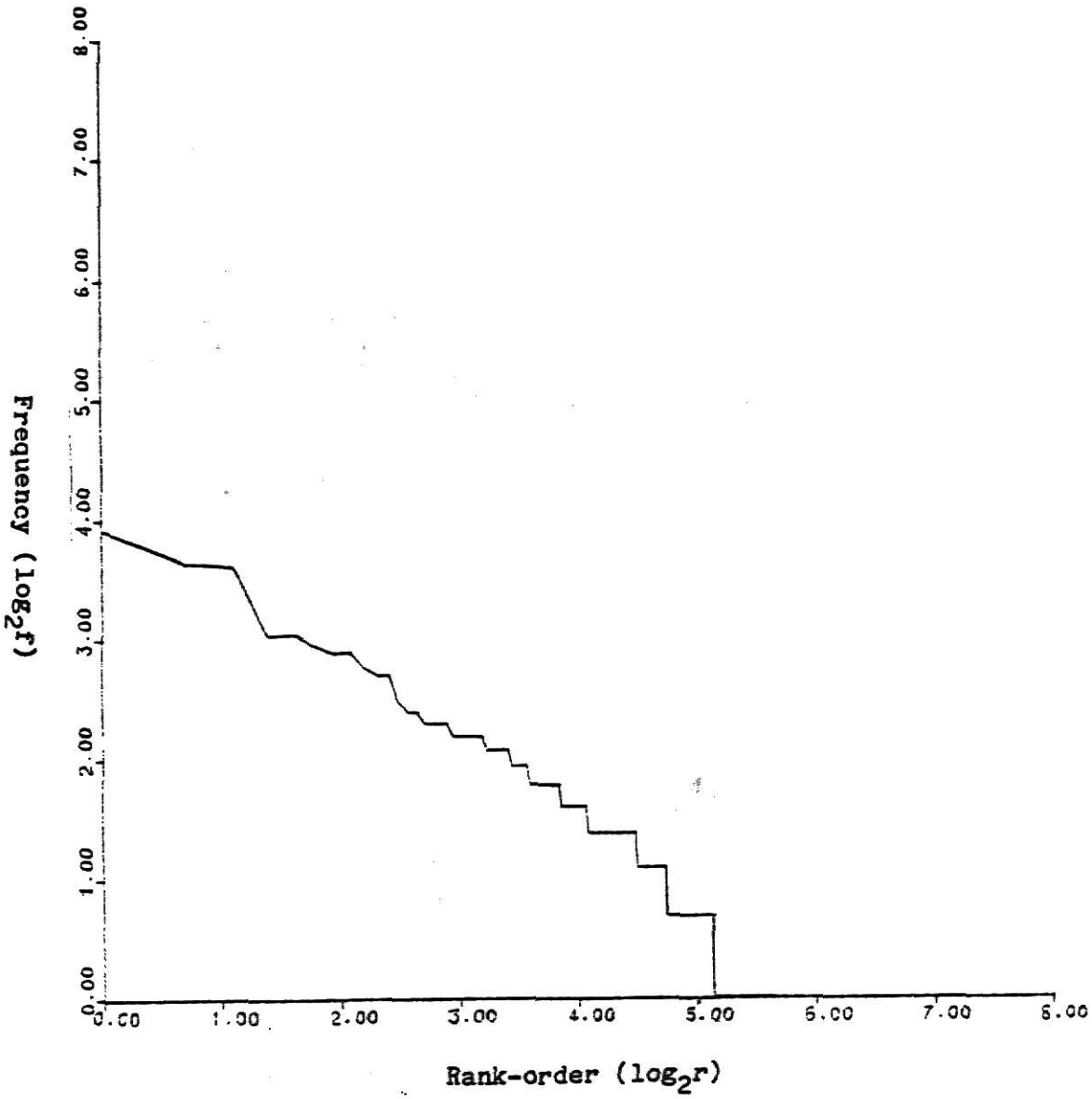


FIGURE 18 Rank-Frequency Distribution of Operands in the Bodies of the Program XREF

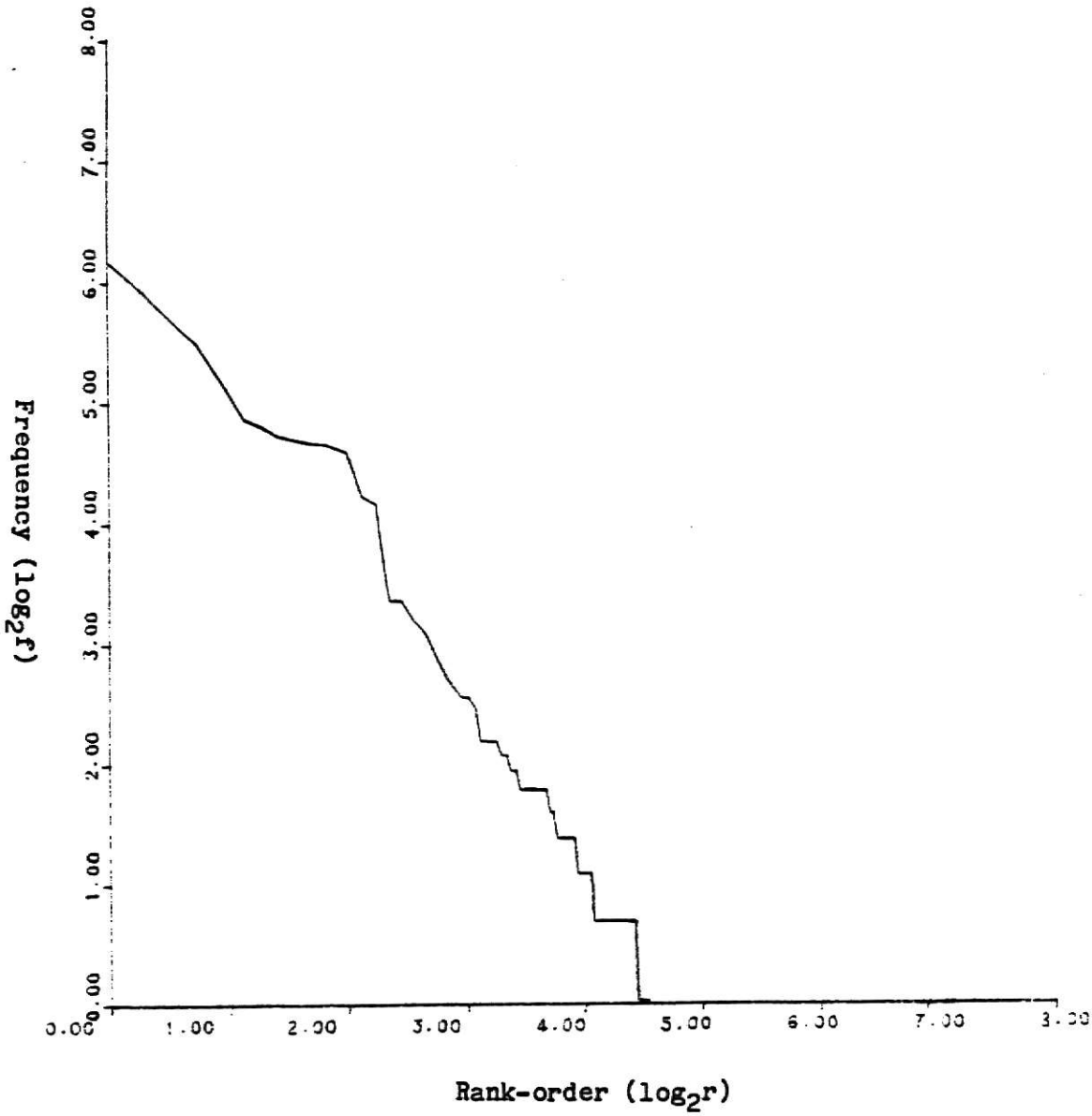


FIGURE 19 Rank-Frequency Distribution of Operators in the Whole Program XREF

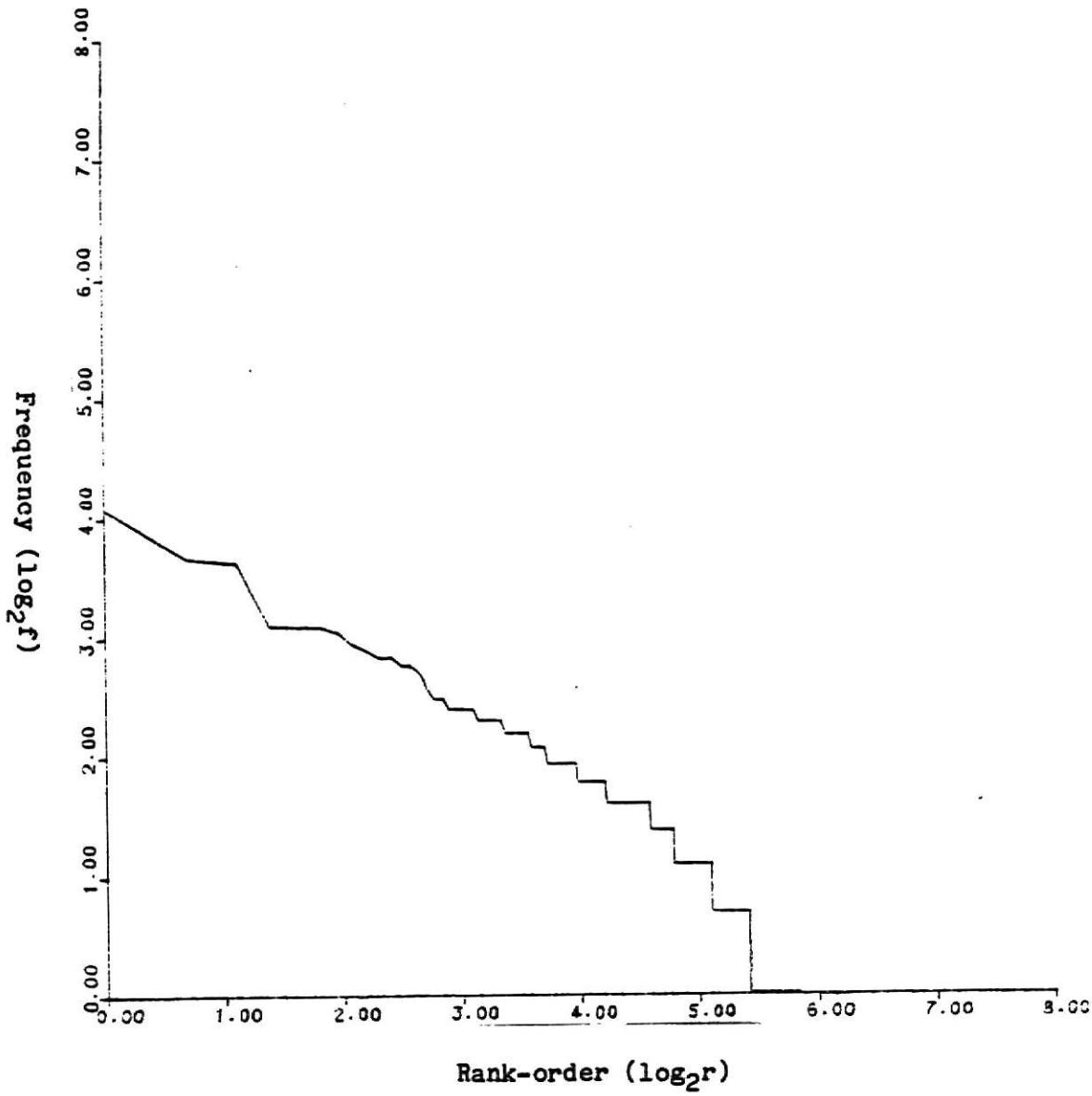


FIGURE 20 Rank-Frequency Distribution of Operands in the Whole Program XREF

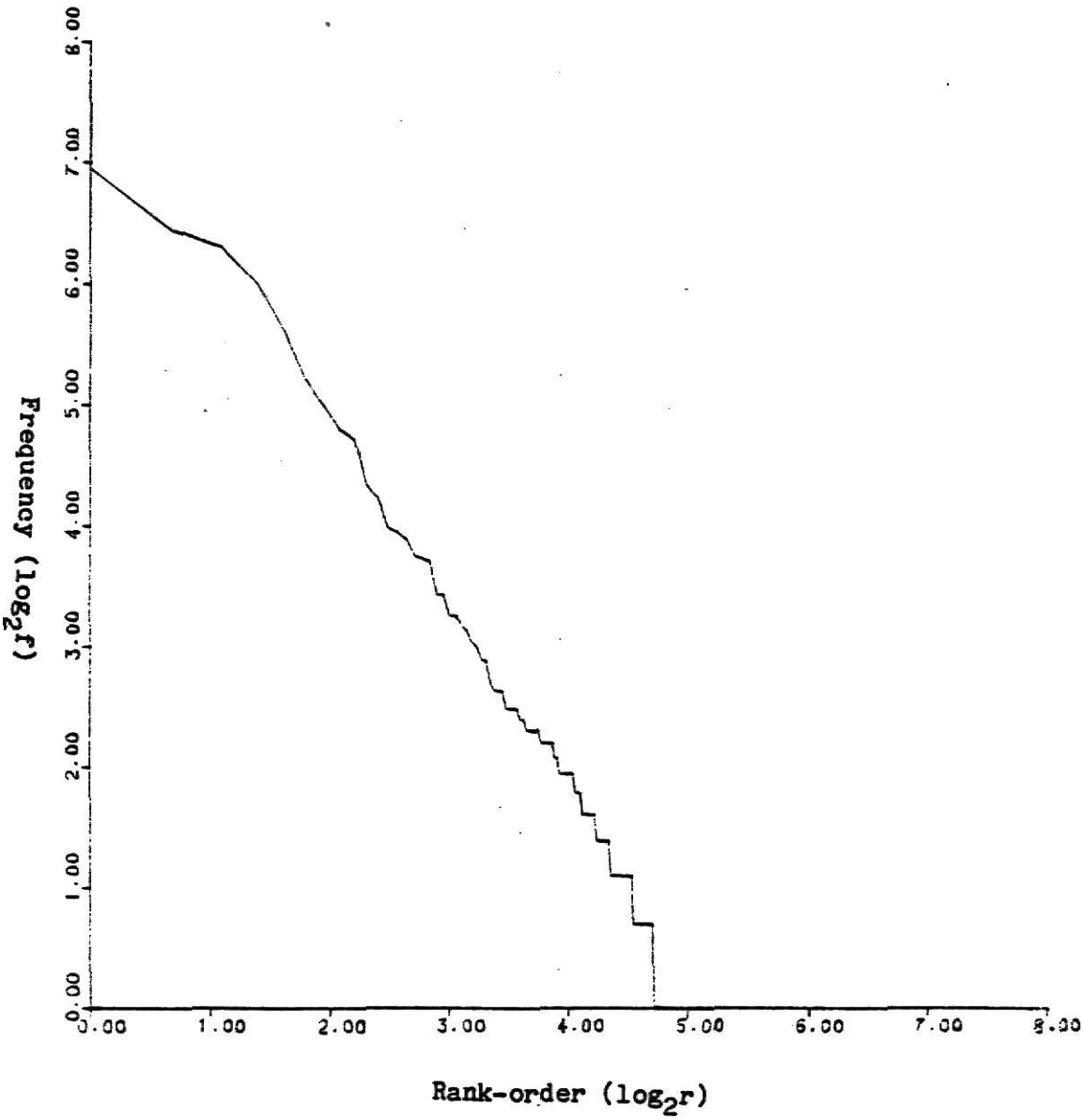


FIGURE 21 Rank-Frequency Distribution of Operators in the Bodies of the Program KPASS4

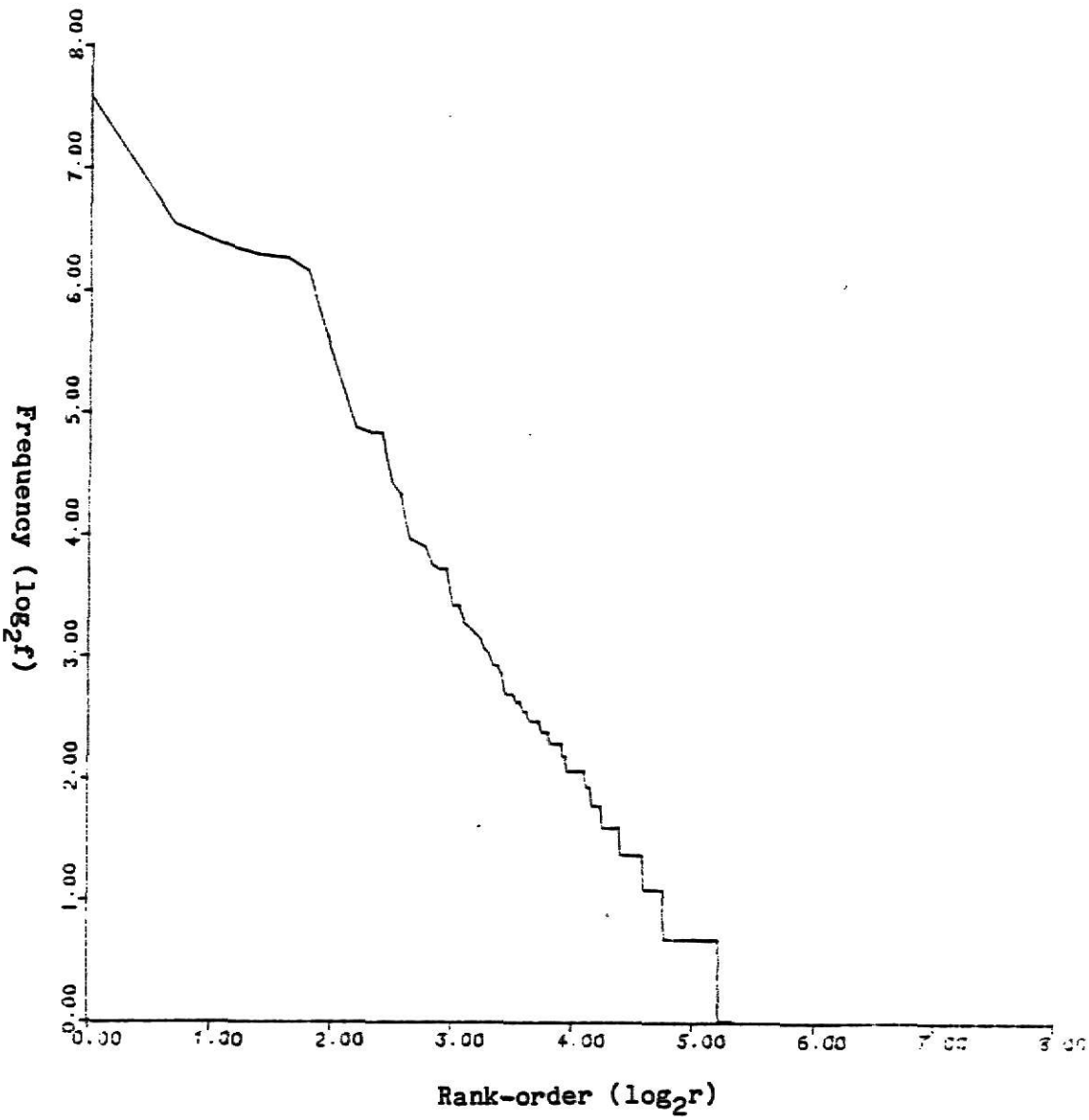


FIGURE 22 Rank-Frequency Distribution of Operators in the Whole Program KPASS4

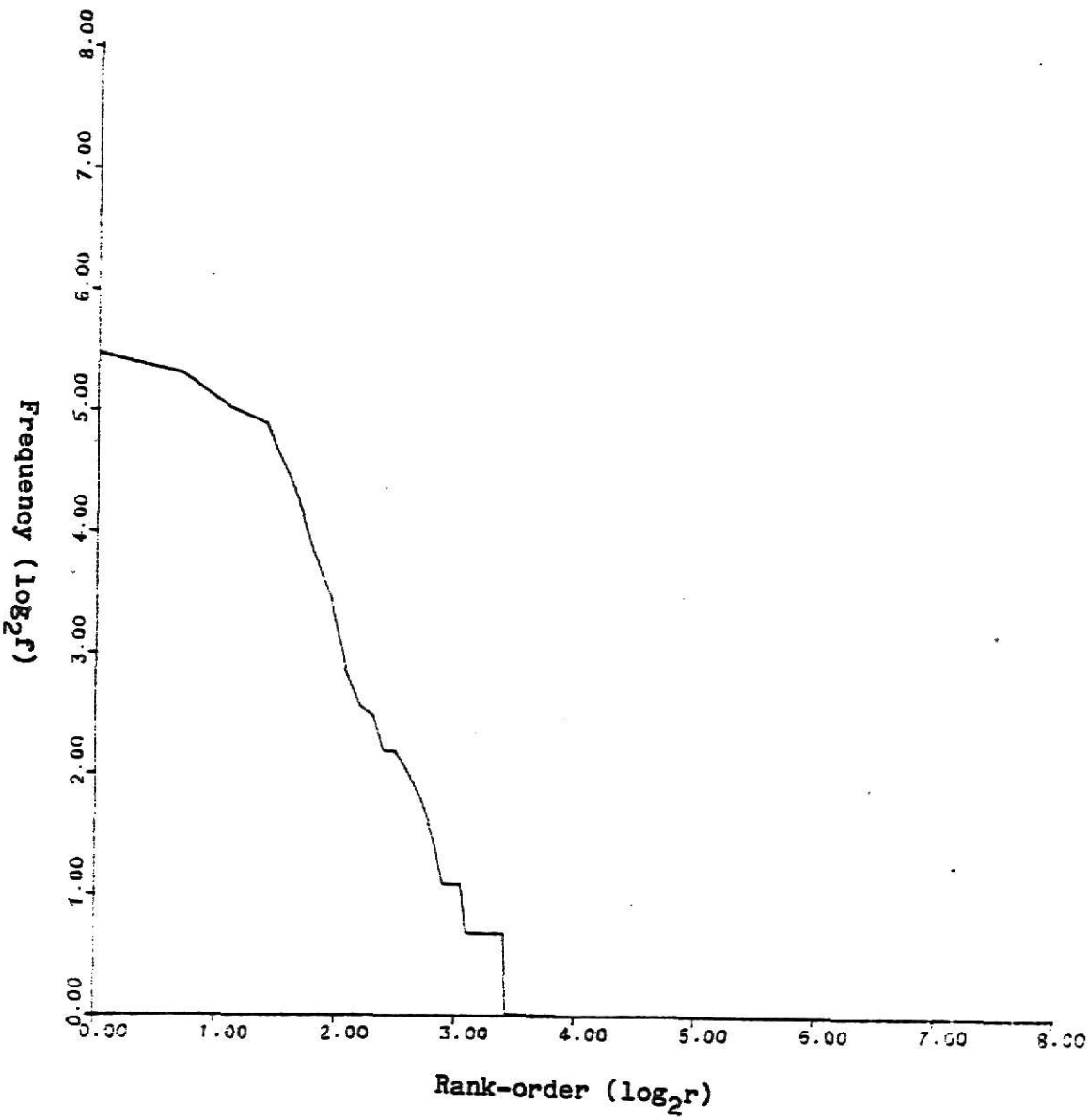


FIGURE 23 Rank-Frequency Distribution of Operators in the Bodies of the Program KPASS5

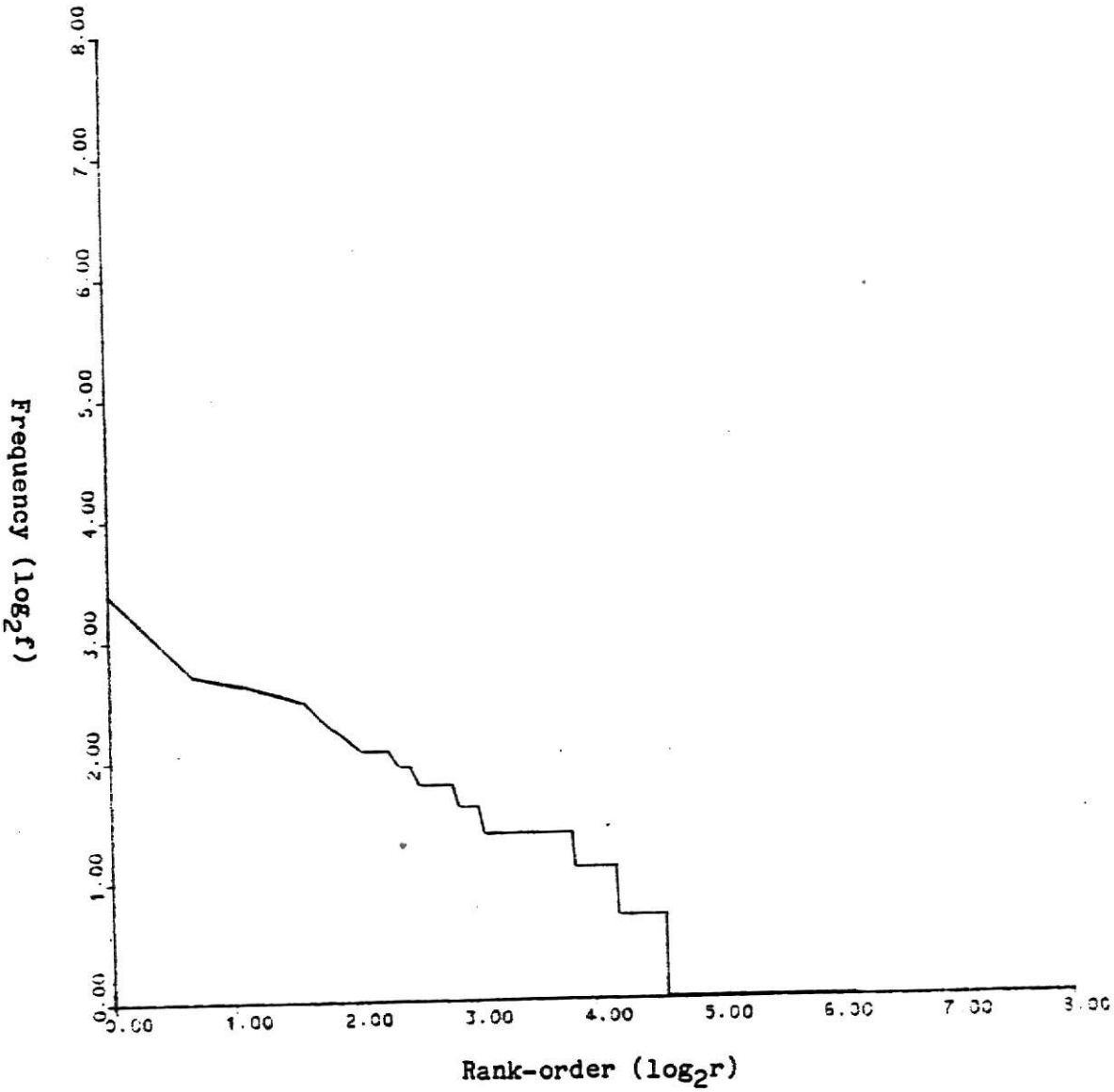


FIGURE 24 Rank-Frequency Distribution of Operands in the Bodies of the Program KPASS5

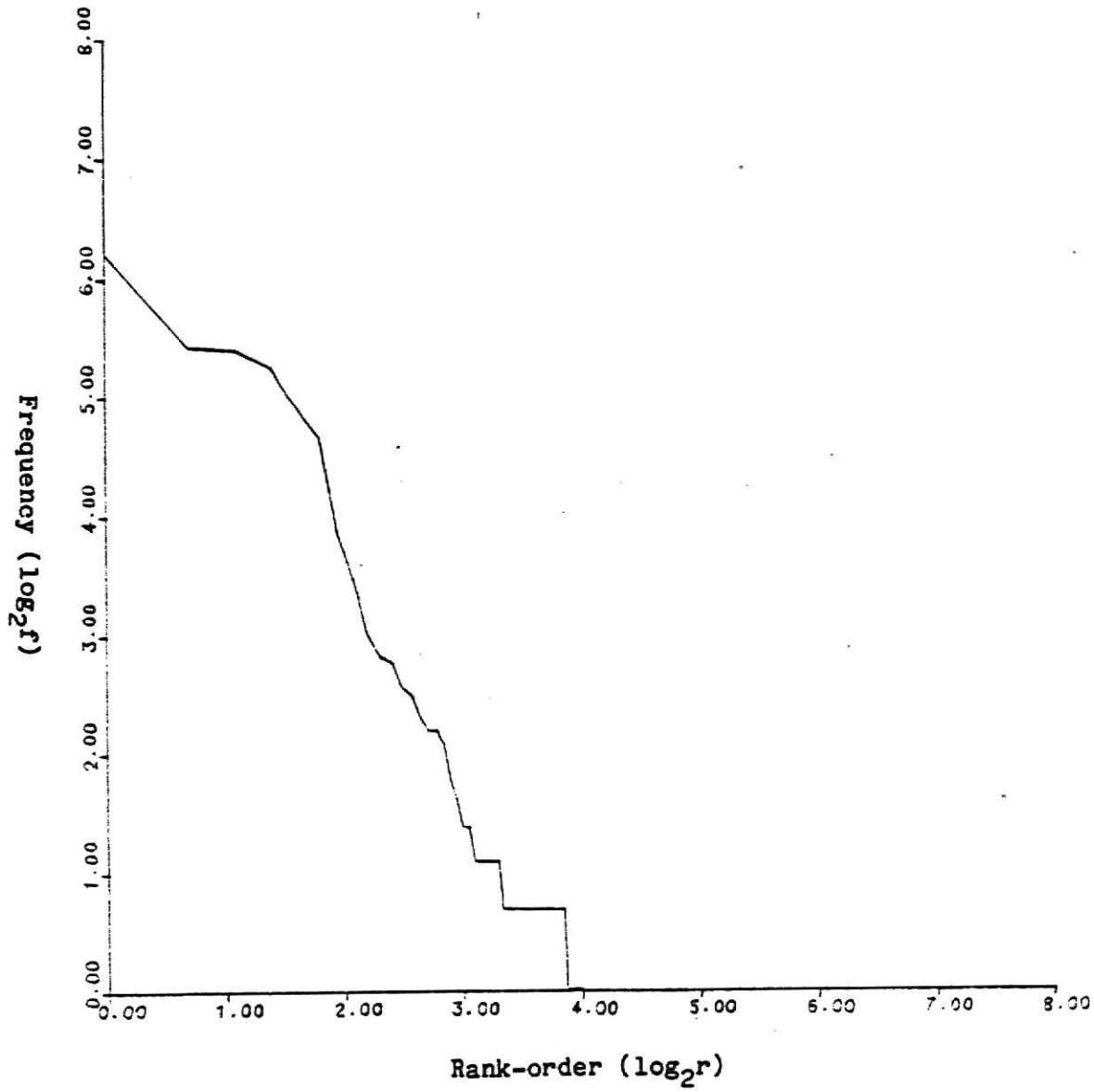


FIGURE 25 Rank-Frequency Distribution of Operators in the Whole Program KPASS5

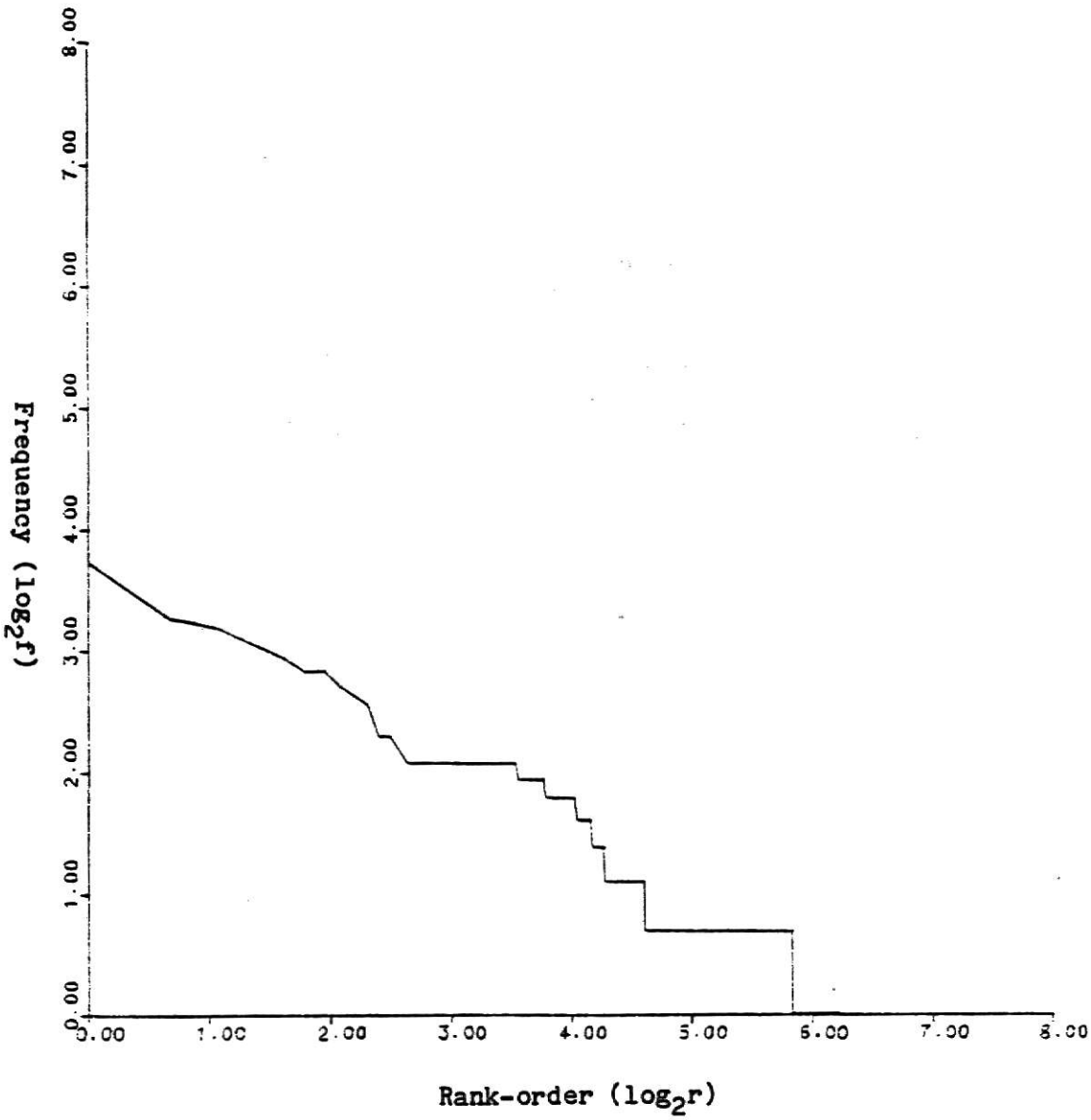


FIGURE 26 Rank-Frequency Distribution of Operands in the Whole Program KPASS5

CHAPTER IV

IMPLEMENTATION

IV.1 Overview

The implementation of the S-PASCAL compiler at the Computer Science Department of K-State consists of nine passes. The first four passes are lexical analysis, syntax analysis, name analysis, and declaration analysis, in that order. In order to automatically acquire information for the Halstead measure for PASCAL programs, pass 1, 3, and 4 were modified such that the output from pass 4 contains the frequency counts for all operators and most operands. Constant names and literal constants are output as intermediate code messages as they are encountered in the source program. The reason for not counting them in pass 4 is primarily due to memory size consideration.

The output from pass 4 must be copied onto a tape. Several programs written in PL/I then rearrange the data, analyze the constant names and literal constants, accumulate the counts for each operator and operand for different experiment cases, and finally report the statistical results. A program written in FORTRAN plots the log-log scale figure for the operator and operand frequency counts.

The whole processing sequence is shown in Fig. 27-28 on page 56-57.

IV.2 Why the Compiler Was Modified

In a program, the same name can be defined in different places to represent different objects. Each of these objects is a distinct operand. The scope rule makes it clear as to when the name represents which object. Since the scope analysis is done in pass 3 (name analysis), the output from this pass is used for the operator and operand counts.

Because of the following two reasons, we could not obtain all the information from the output of pass 3 :

1. Some basic operators are completely gone, and it is impossible to analyze the intermediate code to find where they were. For example, since a multi-dimensional array is represented as a one-dimensional array of one-dimensional arrays, there is no way of knowing which representation the user actually expressed.

2. Constant declarations are gone by pass 3, each reference of a constant name has been replaced either by the value it represents or by a constant number generated by the compiler. Since the use of constant names is a significant feature in PASCAL, we want to treat them differently from the literal constants.

An easy way to solve the first problem is to output a message as

each basic operator is encountered in the source code. This can be done in either pass 1 or pass 2. Pass 1 was chosen to modify because it was easy to modify.

For the second problem, the identification of constant names must be done in pass 3, since constant names need to be scope analyzed also. The approach is detailed in the next section.

IV.3 How the Compiler Passes Were Modified

In pass 1, the lexical analyzer, a unique message is output for each of the four operands, namely, "integer", "boolean", "real", and "char", and for each operator shown in Table 1 on page 10, except for the following. Since no semantic analysis is done in pass 1, there is no way of knowing if a "-" is a unary operator or a binary operator. Similar reason applies to the operator "+". Also, whether the operators "and" and "or" ("+", "-") are used as boolean operators (arithmetic operators) or as set operators is not known. The identification of these operators is thus delayed until the modified pass 4.

The determination of whether a pair of square brackets "[]" is used for array subscription or for set representation is easy. In pass 1, if the open square bracket "[" is preceded by an identifier, it is used for array subscription; if it is preceded by an operator (":=", "+", "=", "and", "or"), it is used for set bracketing.

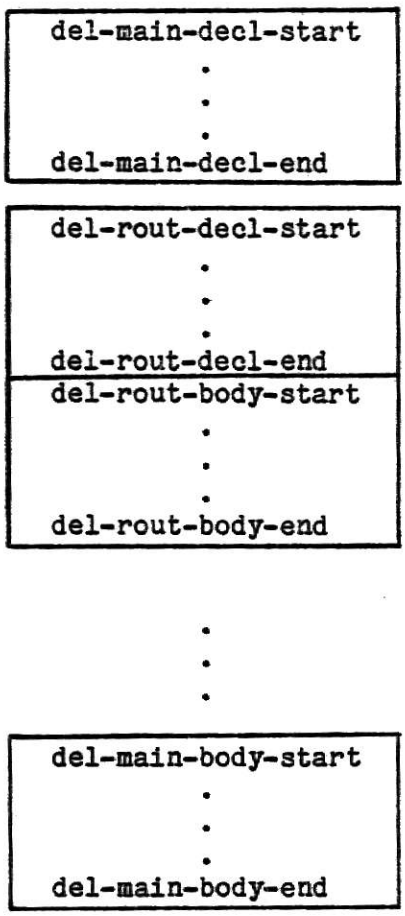
In pass 3 all constant declarations are absorbed. In the output a constant name is treated the same way as a literal constant. The modification made for this work is to output a token (in the form of a message) whenever a constant name is defined, to output a different token (also in message form) when the constant name is referenced. In both cases the token contains the unique number assigned to the constant name. Also, each literal constant in a declaration is output as encountered.

The modified pass 4 does not do any declaration analysis or address calculation. It accumulates the counts of operators and operands other than constants for each single declaration part and for each single body part. The counts are output at the end of the declaration or the body. Constant names and literal constants are output as they are read in. The analysis of constants is left to the next step. Since the compiler introduces many internal names in pass 3, one task in pass 4 is to recognize which names are declared by the user. After some analysis it was found that, from the output of pass 3 each type declaration is user-defined if it is followed by the token "type-def". Because of this fact, all types defined by the user can be identified and counted. Other types are system-induced and thus ignored. As to variables, those that are not declared explicitly are system-induced.

The problem of counting ambiguous operators not resolved in pass 1 is handled easily in pass 4 since the types are known. In the intermediate code input to pass 4 each name token has two fields, the

name index, and the type index of the name. From the type index we can determine whether a name (operand) is of set type or not. For each of the four operators "and", "or", "+", and "-", it is easy to identify which operations the operator represents. The operators are used for set operation if the types of the operands are of set types; otherwise, they are used for boolean or arithmetic operations. The distinction between unary and binary operators of the tokens "+" and "-" is solved in the original pass 3.

The output from the new pass 4 consists of two kinds of tokens: one kind works as delimiters, the other contributes to the counts. For each procedure the declaration part is enclosed by two distinct delimiters. Another two delimiters are used to group the body part. Four other delimiters are used to group the global declaration and the main body. Between each pair of delimiters are the tokens for constants followed by the counts for operators and other operands. The following shows the structure of the output from the modified pass 4:



The output tokens from pass 4 are the following: const_name_definition, const_name_reference, integer_value, real_value, string_value, char_value, boolean_value, nil, and the counts for other operands and all operators.

IV.4 How The Measured Parameters Were Obtained or Calculated

All the rest of the programs are run on IBM/370. The plotting program is written in FORTRAN, others are written in PL/I.

The first process is to move the main program body up such that, like all procedures, the main declaration and the main body parts are paired together.

The second process is to assign a unique number to each distinct literal constant. Each literal constant is replaced by a new token, the const_no, which includes the unique number assigned to the literal constant. Another token is also generated for each constant name in this process. This token is const_name_const_no_link. It contains a constant name and the constant number assigned to the literal constant to which the name is equated. The purpose of introducing this token is merely to ease the effort in the counting process.

In the counting process, the parameters n_1, n_2, N_1, N_2 are calculated and output to a file. The four parameters are calculated for each procedure declaration, each procedure body, each procedure, all declarations, all bodies, and the whole program. The counting is done both with and without constant declarations according to the parameter specified in the JCL statement. The tokens const_name_const_no_link generated in the previous process is used when constant names are treated as aliases of literal constants.

The last process, called the report process, computes the estimated program length $\hat{L}$, the language level λ, and other parameters for each group of the parameters (n_1, n_2, N_1, N_2) obtained from the counting process and prints them. The estimated program level $\hat{L}$ is computed using the equation

$$\hat{L} = (2/n_2)(n_2/N_2) .$$

Since the potential volume V^* is defined as

$$V^* = LV$$

the language level λ is defined as

$$\lambda = LV^* ,$$

the language level λ is computed using the equation

$$\lambda = LV^* = L^2V \simeq \hat{L}^2V,$$

where

$$V = N \log_2 n .$$

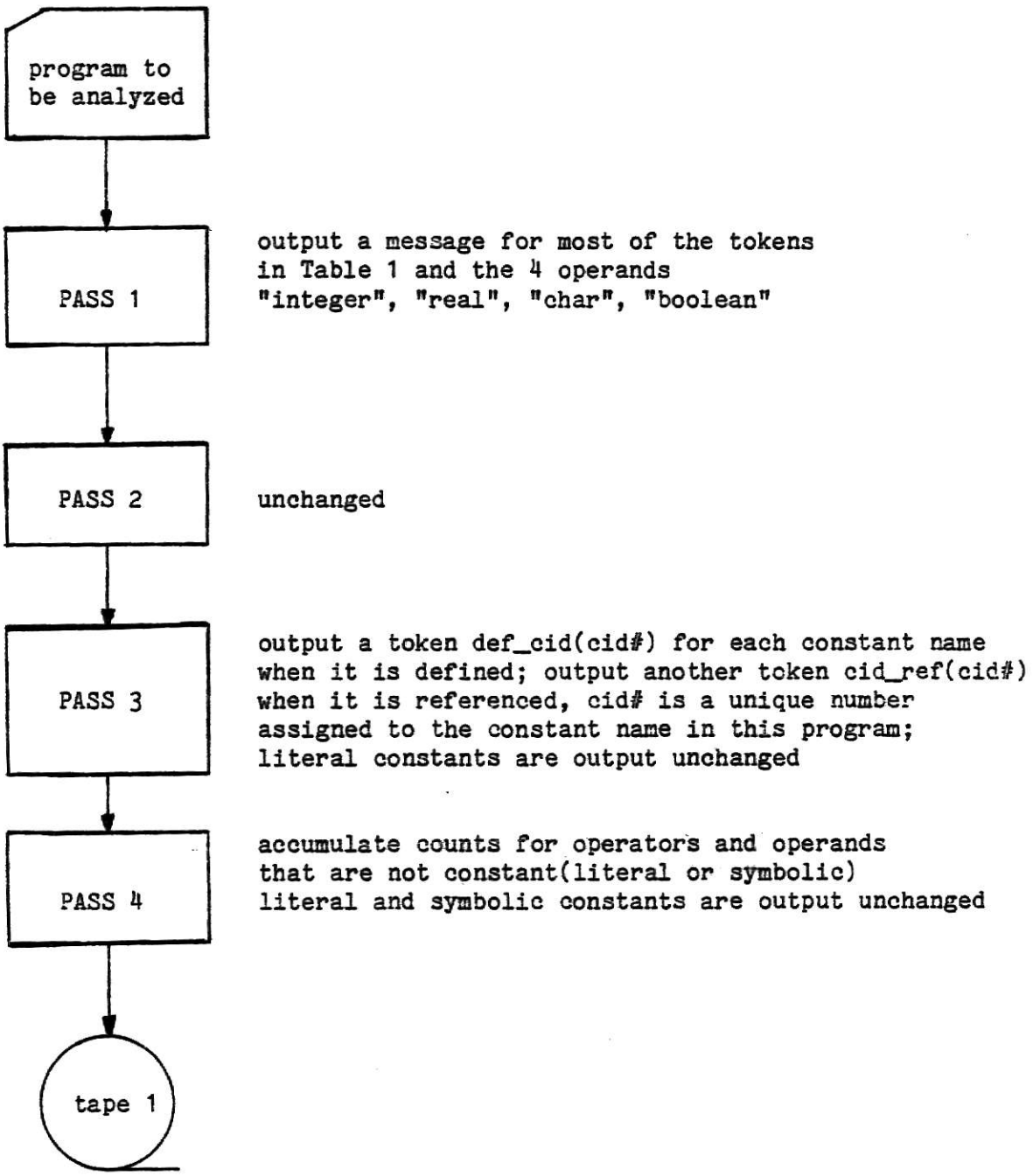


FIGURE 17 Step 1 of the Implementation

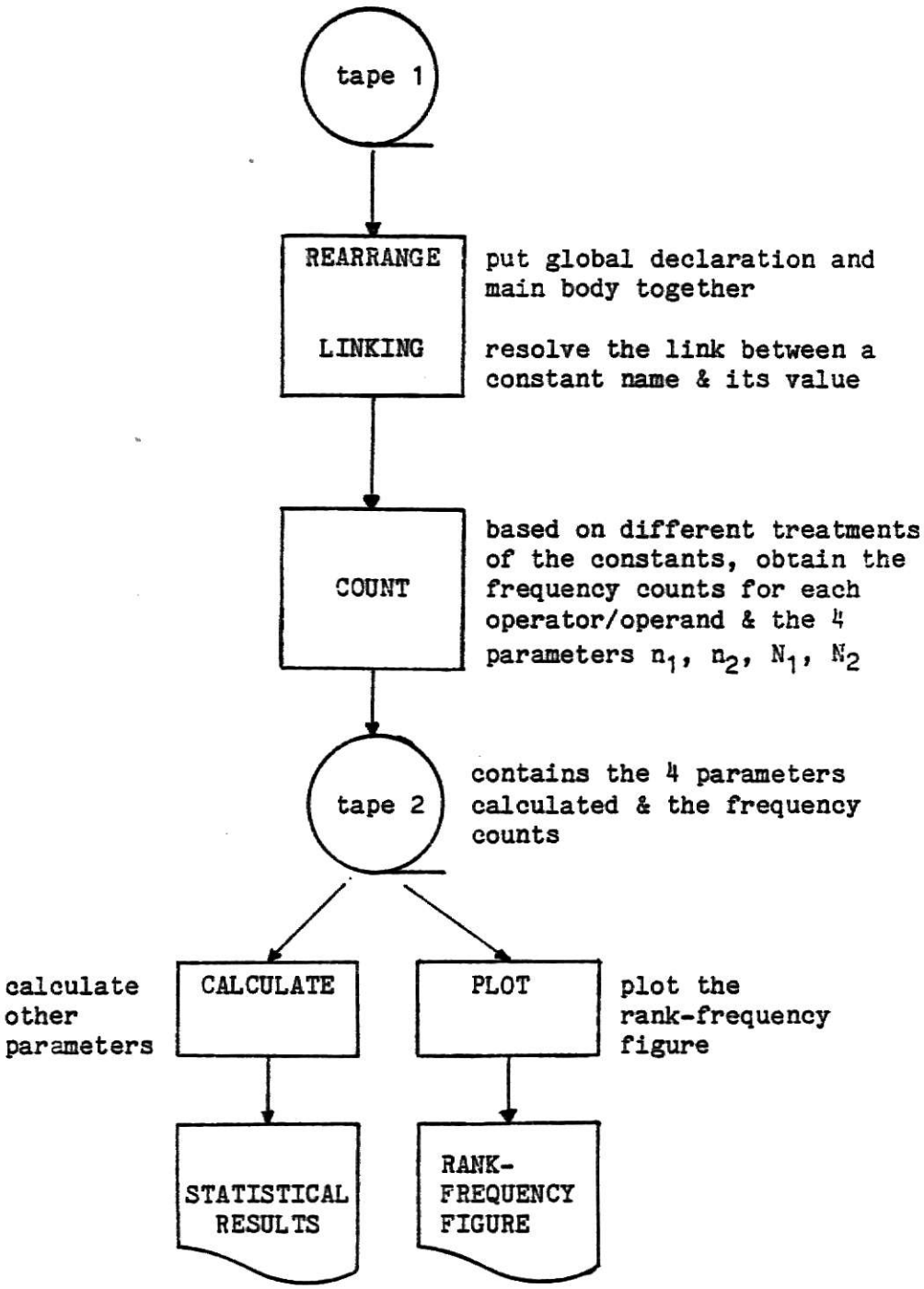


FIGURE 28 Step 2 of the Implementation

CHAPTER V

CONCLUSION

In Halstead's model, the parameters n_1 , n_2, N_1, N_2 are obtained directly from the physical representation of a program. But it treats the unary operator "-" and the binary operator "-" as two different operators. This distinction seems to be based on semantic consideration. The question arises as whether we should take into account all semantic meanings in a program when we use the model. If so, the same operator may be counted as different operators when it is applied to operands of different types. As an example, the operator "+" can be applied to both integer and real numbers; it can also be applied to operands of type "money" or "hours". Whether the operators +'s applied to operands of the four different types are of four different kinds or of one kind is not clear in the model.

Another issue of great interest is the relationship between algorithms and programs. An algorithm is a method used to solve some problem. A program realizes an algorithm in the form of some programming language. In Halstead's model, when we say "complexity", we mean the complexity of a program. But the complexity of an algorithm seems to be of more interest, since it directly reflects the difficulty of the algorithm, or that of the mental process. Though Halstead tried to estimate the time needed to understand a program, the result was not very convincing.

Measuring the complexity of computer programs is a very new field in computer science. There are a lot of unknowns to be answered. Though Halstead's model has some shortcomings to be overcome, it does provide a direction as to how to measure programs.

SELECTED BIBLIOGRAPHY

[Cherry 66] Cherry, Colin, "On Human Communication", MIT Press, 1966.

[Halstead 77] Halstead, Maurice H., "Element of Software Science", Elsevier North-Holland 1977.

[Hartmann 77] Hartmann, Alfred C., "A Concurrent Pascal Compiler for Minicomputers", Lecture Notes in Computer Science 50, Springer-Verlag 1977.

[McCabe 76] McCabe, Thomas J., "A Complexity Measure", IEEE Trans. on Software Engineering, vol. SE-2, NO.4, December 1976, pp. 308-320.

[Ottenstein 76] Ottenstein, Karl J., "A Program to Count Operators and Operands for ANSI-FORTRAN Modules", Computer Science Department Technical Report 196, Purdue University, June 1976.

[Zweben 73] Zweben, Stuart H., "Software Physics : Resolution of an Ambiguity in the Counting Procedure", Computer Science Department Technical Report 93, Purdue University 1973.

HALSTEAD'S COMPLEXITY MEASURE ON PASCAL PROGRAMS

by

SHOU-NAN WANG

B.S. Tamkang College , 1976

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1980

Maurice H. Halstead has developed several measures of program and language complexity, size, content, etc; all of which depend upon four primary parameters, n_1(unique operator count), n_2(unique operand count), N_1(total operators), N_2(total operands), which are static statistics of programs.

This report presents a design and implementation of extensions to Hartman's S-PASCAL compiler to measure an approximation to Halstead's parameters. These parameter counts are repeated for each procedure, and for the whole program. Results are reported for measurements on several programs.