Information Content of Programs and Operation Encoding

ERIC C. R. HEHNER

University of Toronto, Toronto, Ontario, Canada

ABSTRACT The problem of determining the minimum representation of programs for execution by a computer is considered The methods of measuring space requirements suggest practical methods for encoding programs and for designing machine languages An analysis of the operation portion of instructions finds that the 47 operation codes used by a well-known compiler require, on average, fewer than two bits each

KEY WORDS AND PHRASES: redundancy, minimum redundancy encoding, machine language, operation codes

CR CATEGORIES: 5.6, 6.0

1. *Introduction*

In this paper we assume that a computer designer's goal is to design the best machine for the intended environment. Specifically, if the environment is represented by a class of programs to be run on the machine, the goal is to minimize the length of the machine-language representation of the class, or equivalently the average length of programs (weighted by frequency of program) in the class. Identifying and removing common dependencies and contributors to redundancy in machine language has two advantages. First, as much as 75 percent of the space taken by contemporary machine-language representations can be saved. Second, if some data path (between memory and processor, within a processor, or between memories) is a system bottleneck, then minimizing space will help to reduce execution time by making more efficient use of available bandwidth. The cost is an increase in hardware complexity; at current and projected prices, this is an attractive trade-off. Although the discussion in this paper is in terms of machine languages, it is equally applicable to any situation in which storage or transmission of information is a major expense.

One may object to the goal of minimizing redundancy on the grounds that it is needed for reliability. Some forms of redundancy allow the detection of some errors. If there are unused operation codes, or addresses that are illegal or inaccessible, and if the error happens to result in one of these illegal instructions, it can be detected and the offending instruction identified. If the error results in a legal instruction, it may be detected indirectly but escape identification, or it may escape detection. Other forms of redundancy, such as interinstruction dependencies, do not provide any error detection ability.

The use of accidental redundancy in machine-language instructions for error detection is at best a haphazard approach, and at worst a poor excuse for badly designed codes. The purpose of machine language is to specify a sequence of actions as succinctly as possible. Error detection ability is important enough to deserve its own separate mechanisms, specially designed for that purpose, such as parity bits or tag bits [2, 3, 9].

The information content of a program is an almost achievable target to aim for in its machine-language representation; redundancy is an absolute measure of performance in

This work was supported in part by the National Research Council of Canada

Author's address: Computer Systems Research Group, University of Toronto, Toronto, Ontario M5S 1A4, Canada.

this respect for any machine; methods for determining entropy suggest practical methods for improving a machine language. Definitions of these terms will now be presented and applied to the encodings of operations For other parts of instructions and for data the reader is referred to [6].

2. *Definitions*

In information theory [11, 1] the entropy of a set of n messages is defined as $H = -\sum p_i \log p_i$, where p_i is the probability or relative frequency of the ith message, $p_i > 0$, $\sum p_i = 1$. (All logarithms are to base 2, and all sums take subscripts over the range 1 to n. We may allow $p_i = 0$ by taking $0 \log 0 = 0$, although with a suitable adjustment of the message set, this is unnecessary.) The entropy of a set of messages represents the average amount of choice involved in sending a message, or the average amount of uncertainty on the part of the receiver as to which message will be sent. Entropy is independent of any representation of the messages. It is a close (within one bit) lower bound on the average length of messages (weighted by frequency of message), according to a "best" binary representation, i.e. a binary representation that minimizes the average length of a message.

We now define the information content of the ith program (or message) in a class (or set) of programs (or messages) as $I_i = -\log p_i$. It is the amount of information, measured in bits, that distinguishes it from the other members of the class. Like entropy, the information content of a program is independent of the representation and is a close (within one bit) estimate of the number of bits taken by a best binary representation. Entropy is then the average information content of a program.

In information theory, the redundancy of a set of messages is defined as $D = 1 -$ (relative entropy), where the relative entropy is the actual entropy divided by the maximum possible entropy of the set. For a set of n messages, entropy is at its maximum when $p_i = 1/n$ for each i. Redundancy is therefore $1 - H/\log n$. This definition is independent of representation and unsuitable for our purposes. We define the redundancy of a representation as $D' = 1 - H/$(average message length), where the length of a message depends on its representation. With this definition, redundancy measures the proportion of the representation that exceeds the entropy.

3. *Determination of Entropy*

If the entire class of programs to be run on the machine and the probability or frequency of each program are known, then calculating the information content of each program and the entropy of the class is straightforward. More than that, an algorithm for determining a best representation has been given by [7]. However, few machine designers are given the luxury of such information.

Normally one knows only that the programs to be run on the machine come from an infinite class, such as the set of all Algol programs, the set of all PL/1 programs, or the union of such sets. The definition of entropy can easily be extended to cover an infinite class, and, if the probability distribution is known, one can still compute the target for the average length of a program. According to the classical definition, the redundancy of any infinite class whose members appear with nonzero probabilities is 100 percent, but the redundancy of representation provides a performance measure even for infinite classes. With luck, one may find a good machine-language representation for the particular distribution, but there is no algorithm for determining a best representation for an arbitrary distribution over an infinite class.

Normally the probability distribution is not known. One is given a sample of programs from which a reasonable estimate of the distribution for the class must be derived. The sample is intended to be representative to some degree of the entire class. The answer to the question "Just how representative?" will provide the method of determining the class entropy.

It may be assumed that the relative frequencies of the language tokens in a large sample are representative of the frequencies in the class. Similarly it may be assumed that the relative frequencies of production numbers in a parse of the sample, or the frequencies of states in the parser, represent their frequencies over the entire class. Since a token sequence is a particular representation of programs, the entropy of the token set (average information content of a token) multiplied by the average number of tokens in a program (over the sample) will be an estimate of the entropy of the class of programs. Similarly the production sequence and parser state sequence provide estimates of the entropy. Unfortunately these estimates may be poor, and they depend rather heavily on the choice of representation.

Almost as certainly, the frequencies of token pairs, production pairs, or state pairs are representative. With decreasing confidence, it may be assumed that higher m-tuple frequencies are representative. In the following paragraphs, $\mathbf{j}$ will stand for an $(m-1)$-tuple of subscripts. If $P_{i\mathbf{j}}$ is the probability of the $i\mathbf{j}$th m-tuple of symbols from an alphabet of n symbols (tokens, productions, etc.) and N is the average length in symbols of a program, then the estimate of entropy based on nonoverlapping m-tuple frequencies is

$$G_m = -(N/m) \sum_{i\mathbf{j}} p_{i\mathbf{j}} \log p_{i\mathbf{j}}.$$

As m increases, these estimates form a monotonically decreasing sequence (Appendix 1, Theorem 1). Furthermore, as m becomes large, m-tuples become entire programs, and the sequence of estimates converges to the entropy of the sample. When the value of m is too small, not enough use is made of the sample, and the estimate is too high. When m is too large, too much use is made, and the estimate is low (unless the sample is the complete class, in which case the sample entropy is the class entropy).

More effective use can be made of m-tuple frequencies by considering the conditional entropy of the next symbol when the $m-1$ preceding symbols are known. If $p_{k|\mathbf{j}}$ is the probability of appearance of the kth symbol given that the $\mathbf{j}$th $(m-1)$-tuple precedes it, then the mth estimate of class entropy based on conditional symbol entropy is

$$F_m = -N \sum_{\mathbf{j}} p_{\mathbf{j}} \sum_{k} p_{k|\mathbf{j}} \log p_{k|\mathbf{j}},$$

where now the frequencies include overlapping tuples. Like G_m, F_m is a monotonically decreasing function of m (Appendix 1, Theorem 2), converging to the entropy of the sample, and $F_m \leq G_m$ (Appendix 1, Theorem 3).

The problem is to decide which m makes F_m or G_m the best estimate of H. It must be large enough to include all dependencies or statistical influences characteristic of the class and none that are due only to the finite size of the sample. This m is found by formulating an appropriate hypothesis and applying a statistical test to determine, to a given level of confidence, whether the sample is sufficient to support (or reject) the hypothesis. Suppose the sample were the complete class. Now $G_{m+1} = G_m$ implies $G_m = G_{m-1}$ (Appendix 1, Theorem 1). For most programming languages, pairs of tokens (productions, states) are not independent (usually not all pairs are legal); hence by Theorem 1, $G_2 \neq G_1$ and therefore $G_m \neq H$ for any m. The F estimates are more cooperative; F_m may converge to H for some finite m. Of course, when it does, $F_{m+1} = F_m$ ever after. Now $F_{m+1} = F_m$ if and only if $p_{k|\mathbf{j}} = p_{k|\mathbf{j}}$ (Appendix 1, Theorem 2). With a limited sample, one looks for the last m for which the hypothesis $p_{k|\mathbf{j}} = p_{k|\mathbf{j}}$ can be confidently rejected, i.e. the largest distance over which the sample indicates that a dependency exists. With limited computer resources, one may wish to make an additional assumption: that a dependency of order m implies a dependency of order $m-1$. This allows one to stop at the first m for which a dependency is not indicated.

There are other methods of estimating entropy, for example, by calculating symbol entropy conditionally upon information contained in the parse stack during a parse of the

programs. The methods presented above lead, in Section 4, to practical suggestions for machine-language design.

4. *Application to Operation Codes*

Wortman was able to improve the operation set of the Student PL machine [12] by inventing new operations to replace certain pairs of old ones wherever they occur in a program. The pairs were chosen on the basis of high frequency and compatible semantics, i.e. pairs that fit "naturally" together; the result of the combinations was a reduction in the average space required for a program. This technique can be made algorithmic by replacing, wherever it occurs in the sample, that pair that reduces the information content the most (Appendix 2), then repeating until some prespecified limit is reached. We call this method "iterative pairing." As a heuristic for finding which pair reduces information content the most, the most common pair may be chosen. This heuristic is well suited to machines with fixed length fields since it tends to produce operations that have nearly equal frequencies. Unfortunately there may be no pair whose replacement reduces the information content; it may actually increase for any or all pairs. Furthermore, gaining the most benefit from a limited number of replacements may require choosing, at intermediate stages, pairs other than those that reduce information content the most or those that are most frequent.

If all pairs of operations are replaced by new operations, or in general all m-tuples by compound operations, then the space required for a minimum redundancy encoding of the m-tuples will not increase, and it will decrease if there are dependencies of order m or less (Appendix 1, Theorem 1). This coding method corresponds to the estimates of entropy based on m-tuple frequencies. Implementation considerations limit the size of the operation set, which may grow exponentially with m. In practice, however, the growth is much slower: In one study, of the 10^{21} possible 10-tuples of operations on the CDC3600, fewer than 7000 had nonzero frequencies, indicating that there are important dependencies of order 10 or less [5]. The advantage of iterative pairing, in spite of its uncertainty, is that it takes account of important high order dependencies before less important low order dependencies, and therefore gains more benefit from fewer new instructions.

Foster and Gonter have suggested a method of shortening the operation field that takes advantage of interinstruction dependencies [4]. In their method each of the n operations is allowed k successors plus an escape; that is, after any operation, only its k most frequent successor operations are given a code, plus one code for all $n - k$ other operations. The "operand" of the escape code is used to set the state of the machine as it would be if an operation which has the desired successor had just been performed. These codes are called "conditional" since an operation code can be decoded only by knowing the preceding operation. Since Foster and Gonter used a fixed length operation code field, their method can save space only if $k < n - 1$. With a variable length field, there is no advantage in keeping $k < n$, so we take $k = n$. We make use of the variable length fields by giving the operations a minimum redundancy encoding for each possible predecessor operation. Inclusion of a zero-frequency nonoperation in the set produces an open-ended code for future expansion, at a cost of one bit on the least frequent operation.

An immediate generalization of their method makes each operation code conditional upon the preceding $m - 1$ operations; given the m-tuple frequencies, the appropriate codes can be generated. This corresponds, in real coding terms, to the mth estimate of class entropy based on conditional symbol entropy. Like m-tupling, this method carries the guarantee which iterative pairing lacks: It will not increase the required space and will reduce it if there are dependencies of order m or less (Appendix 1, Theorem 2). For each m (greater than 1) this method is more effective in reducing space than compounding instructions into m-tuples (Appendix 1, Theorem 3). Furthermore, conditional

coding is more readily viewed as just a coding technique, not as an increase in the operation set.

5. *Experimental Results*

To see what gains may be made by these encoding techniques, a large program (XCOM—the compiler for the XPL language), written in a PL/1-like language (XPL), was compiled into a machine language that closely expresses the capabilities of the source language. Every commutative operator presents an opportunity to exchange the sequences of instructions that develop the two operands; to get maximum benefit from iterative pairing or conditional coding, commutative instructions were placed in a standard order We arbitrarily chose to place development of a complex expression operand before a single variable or constant operand, and we chose to place a variable operand before a constant operand. Within the category "variable operand," we placed procedure (function) calls ahead of arrays and arrays ahead of simple variables. We realize that the meaning of a program may be changed if development of an operand involves a call to a procedure with side effects, but we choose not to cater to such programming practices.

As a basis for comparison, an IBM-360-like encoding [8] is included, along with a minimum redundancy encoding on the assumption of operation independence [7]. The former gives 8 bits to each operation code, and the latter gave an average of 3.6 bits per operation code.

The iterative pairing procedure was applied to the 47 original operations until the number of operations had increased to 178 (at this point our ability to give sensible names to the new combinations was exhausted). The pairs being combined must not include any pairs whose second member is the object of a branch. A stricter, but more convenient, condition is to disallow pairs of instructions that cross a statement boundary in the original source program. The result was an average of 4.85 bits per new operation code, or 1.8 bits per original operation.

The original operations were given a conditional coding, first in the context of one preceding operation, then in the context of the preceding pair, and finally in the context of the preceding triple of operations. Rather than maintaining context wherever possible, we established a standard context at every instruction that is the object of any branch or call. Our result is thus weakened, but easier to achieve. The chosen standard context was an arbitrary tuple of instructions. The results were 2.1, 1.7, and 1.6 bits per operation code on average. These results are summarized in Table I.

6. *Conclusions*

Some results in the encodings of operation have been presented. Results for other portions of instructions, and for data, show that similar space savings are possible [6]. With our sample (XPL compiler [10]), we trimmed 75 percent from the space taken by a contemporary machine representation (IBM 360) and 60 percent from a language directed machine representation that did not employ our techniques. As a bonus, the variable length encodings eliminate the overflow problem that results from trying to represent a large or infinite set by a fixed length encoding.

These encodings may be useful with current hardware for long term storage of programs. For execution, a hardware expander could transform instructions to a conventional fixed length representation. They may be useful whenever the transmission of information through a fixed bandwidth is a major expense or system bottleneck. Or, and this is our main interest, space-saving encodings could be the basis for a more economic computer design.

Appendix 1. *Some Information Theorems*

In this Appendix log denotes the binary (base 2) logarithm. All free subscripts are under-

TABLE I

Encoding	Bits per operation	Percent of (b)	Percent of (a)
(a) Like IBM 360	8		
(b) "Minimum redundancy"	3.6		45 3
(c) Iterative pairing	1.8[a]	49.9	22.6
(d) Conditional coding			
1 preceding	2.1	57 2	25.9
2 preceding	1 7	47.0	21 3
3 preceding	1 6	43.8	19 8

[a] Bits per original operation, 4 85 bits per compound operation

stood to be universally quantified over the range 1 to n, and all sums are over the range 1 to n. $\mathbf{j}$ stands for an $(m-1)$-tuple of subscripts. A subscripted p is a probability: p_i is the probability of appearance of the ith symbol in any position in a sequence of these symbols; $p_{i\mathbf{j}}$ is the probability of appearance of the $i\mathbf{j}$th m-tuple; $p_{k|\mathbf{j}}$ is the probability of appearance of the kth symbol given that the $\mathbf{j}$th $(m-1)$-tuple precedes it.

$$p_{\mathbf{j}} > 0, \quad p_{k|\mathbf{j}} = p_{\mathbf{j}k}/p_{\mathbf{j}},$$

$$\sum_{\mathbf{j}} p_{\mathbf{j}} = \sum_{k} p_{k|\mathbf{j}} = 1, \quad \sum_{i} p_{i\mathbf{j}} = \sum_{k} p_{\mathbf{j}k} = p_{\mathbf{j}}.$$

We may allow $p_{\mathbf{j}} = 0$ if we accept the convention $0 \log 0 = 0$, $0/0 = 0^0 = 1$.

Let $g_m = -(1/m) \sum_{i,\mathbf{j}} \log p_{i\mathbf{j}}$ be the average information content per symbol of an m-tuple of symbols. Then the estimate of entropy based on m-tuple frequencies is $G_m = Ng_m$, where N is the average number of symbols in a message.

Let $f_m = -\sum_{\mathbf{j}} p_{\mathbf{j}} \sum_k p_{k|\mathbf{j}} \log p_{k|\mathbf{j}}$ be the average information content per symbol, conditional upon the preceding $m-1$ symbols. Then the mth estimate of class entropy based on conditional symbol entropy is $F_m = Nf_m$. Define

$$d_{m+1} = -[1/(m+1)] \sum_{i\mathbf{j}k} p_{i\mathbf{j}k} \log (p_{i\mathbf{j}} p_{\mathbf{j}k}/p_{\mathbf{j}}).$$

LEMMA 1. *If $q_i > 0$ and $\sum q_i = 1$, then $-\sum p_i \log p_i \leq -\sum p_i \log q_i$ with equality iff $p_i = q_i$.*

PROOF. See [1].

LEMMA 2. *$g_{m+1} \leq d_{m+1}$ with equality iff $p_{k|i\mathbf{j}} = p_{k|\mathbf{j}}$.*

PROOF.

$$p_{\mathbf{j}k}/p_{\mathbf{j}} > 0, \quad \text{and} \quad \sum_{i\mathbf{j}k} (p_{i\mathbf{j}} p_{\mathbf{j}k}/p_{\mathbf{j}}) = \sum_{i\mathbf{j}} (p_{i\mathbf{j}} p_{\mathbf{j}}/p_{\mathbf{j}}) = 1.$$

Therefore, by Lemma 1, $g_{m+1} \leq d_{m+1}$ with equality iff $p_{i\mathbf{j}k} = p_{i\mathbf{j}} p_{\mathbf{j}k}/p_{\mathbf{j}}$, i.e. iff $p_{i\mathbf{j}k}/p_{\mathbf{j}} = p_{\mathbf{j}k}/p_{\mathbf{j}}$, hence the result.

THEOREM 1. *$g_m \geq g_{m+1}$, $g_m = g_{m+1}$ implies $g_{m-1} = g_m$. That is, g may begin flat, but after the first decease, it is a strictly monotonically decreasing function of m.*

PROOF. By induction on m:

Induction basis:

$$\begin{aligned} g_1 &= -\sum p_i \log p_i \\ &= -\frac{1}{2}\sum_{ik} p_{ik} \log p_i - \frac{1}{2}\sum_{ik} p_{ik} \log p_k \\ &= -\frac{1}{2}\sum_{ik} p_{ik} \log (p_i p_k) \\ &\geq -\frac{1}{2}\sum_{ik} p_{ik} \log p_{ik} \quad \text{by Lemma 1} \\ &= g_2. \end{aligned}$$

Note. $g_1 = g_2$ iff $p_{ik} = p_i p_k$, i.e. the symbol probabilities are independent.

Induction step: First note that

$$\begin{aligned}(m+1)d_{m+1} &= -\sum_{ijk} p_{ijk} \log (p_{ij}p_{jk}/p_j) \\ &= -\sum_{ijk} p_{ijk}\log p_{ij} - \sum_{ijk} p_{ijk} \log p_{jk} + \sum_{ijk} p_{ijk} \log p_j \\ &= 2m\, g_m - (m-1)g_{m-1};\end{aligned}$$

$$\begin{aligned}(m+1)(g_m - g_{m+1}) &= (m-1)(g_{m-1} - g_m) + 2m\, g_m - (m-1)g_{m-1} - (m+1)g_{m+1} \\ &= (m-1)(g_{m-1} - g_m) + (m+1)(d_{m+1} - g_{m+1}) .\end{aligned}$$

The first term is nonnegative by the induction hypothesis. The second term is nonnegative by Lemma 2. Hence the left side is nonnegative, and the induction is complete. If the left side is zero, the first term on the right must also be zero; hence the result.

THEOREM 2.

$$f_m \geq f_{m+1}, \qquad f_m = f_{m+1} \quad \textit{iff}\ p_{k|ij} = p_{k|j}.$$

PROOF.

$$\begin{aligned}f_m &= -\sum_j p_j \sum_k (p_{jk}/p_j)\log(p_{jk}/p_j) \\ &= -\sum p_{jk}\log p_{jk} + \sum p_j \log p_j \\ &= mg_m - (m-1)g_{m-1}.\end{aligned}$$

$$\begin{aligned}f_m - f_{m+1} &= mg_m - (m-1)g_{m-1} - (m+1)g_{m+1} + mg_m \\ &= (m+1)(g_m - g_{m+1}) - (m-1)(g_{m-1} - g_m).\end{aligned}$$

From the proof of Theorem 1 we have

$$(m+1)(g_m - g_{m+1}) = (m-1)(g_{m-1} - g_m) + (m+1)(d_{m+1} - g_{m+1});$$

therefore $f_m - f_{m+1} = (m+1)(d_{m+1} - g_{m+1}) \geq 0$ by Lemma 2, with equality iff $p_{k|ij} = p_{k|j}$.

THEOREM 3. $f_m \leq g_m$.

PROOF. From the proof of Theorem 2 we have $f_m = mg_m - (m-1)g_{m-1}$; therefore $g_m - f_m = (m-1)(g_{m-1} - g_m) \geq 0$ by Theorem 1.

Appendix 2. Iterative Pairing

Suppose that in a representative sequence S the symbol pair a_1a_2 is replaced with the new symbol a_0. Let p_i' be the probability of occurrence of a_i in the new sequence S'. Then

$$p_0' = p_{12}/(1-p_{12}), \quad p_1' = (p_1 - p_{12})/(1-p_{12}), \quad p_2' = (p_2 - p_{12})/(1-p_{12}),$$
$$\text{and} \quad p_i' = p_i/(1-p_{12})$$

for $i = 3$ to n. The average information content of a symbol in S is $I = -\sum p_i \log p_i$. The average information content of a symbol in S' per symbol in S is $I' = -(1-p_{12})\sum_{i=0}^{n} p_i' \log p_i'$. So the change in information content (per symbol in S) is

$$I' - I = \log \frac{p_1^{p_1} p_2^{p_2} (1-p_{12})^{1-p_{12}}}{(p_1 - p_{12})^{p_1-p_{12}}(p_2 - p_{12})^{p_2-p_{12}} p_{12}^{p_{12}}}.$$

Note. $0 \leq p_{12} \leq p_1, p_2 \leq 1$; so the fraction is always positive.

If p_1 or $p_2 = 0$ or 1, then $I' = I$. If $p_{12} < p_1, p_2 \ll 1$, then by applying a binomial expansion we find that, to first order, $I' - I = p_{12}\log(p_1p_2/p_{12})$. If $p_{12} = p_1p_2$, then to first order $I' = I$. If a_1a_1 is the symbol pair replaced, then

$$I' - I = \log \frac{p_1^{p_1}(1 - p_{11})^{1-p_{11}}}{(p_1 - 2p_{11})^{p_1-2p_{11}} p_{11}^{p_{11}}}$$

or, to first order,

$$I' - I = p_{11}\log(p_1^2/p_{11}) .$$

In iterative pairing, that pair for which $I' - I$ is greatest is combined at each stage. Perhaps a good heuristic is choosing that pair $a_i a_j$ for which p_{ij} is greatest. If the symbols a_1 and a_2 appear only in the pair $a_1 a_2$, then $p_1 = p_2 = p_{12}$ and

$$I' - I = p_1 \log p_1 + (1 - p_1)\log(1 - p_1) < 0 \quad \text{for} \quad 0 < p_1 < 1.$$

Therefore such pairs should always be combined.

ACKNOWLEDGMENTS. This work has benefited from discussions with several people, particularly D.B. Wortman, J.J. Horning, W.M. McKeeman, and R.N.S. Horspool, and from the constructive criticisms of the referees.

REFERENCES

1 ASH, R *Information Theory* Interscience Tracts in Pure and Applied Mathematics, No 19, Wiley, New York, 1965
2. BURROUGHS CORP Burroughs B6700 Information Processing Systems Reference Manual. Detroit, Mich , 1969
3. FEUSTAL, E.A The Rice Research Computer—a tagged architecture Proc AFIPS 1972 SJCC, Vol 40, AFIPS Press, Montvale, N J , pp 369-377.
4. FOSTER, C C , AND GONTER, R H Conditional interpretation of operation codes *IEEE Trans. Computers C-20* (Jan. 1971), 108-111
5. FOSTER, C C. Private communication.
6 HEHNER, E C R Computer design to minimize memory requirements *Computer 9,* 8 (Aug 1976), 65-70
7 HUFFMAN, D A A method for the construction of minimum redundancy codes *I R.E 40,* 9 (Sept 1952), 1098-1101
8. IBM CORP IBM System/360 Principles of Operation Form A22-6821-x, IBM Syst. Develop Div , Poughkeepsie, N Y , 1968
9 ILIFFE, J K *Basic Machine Principles* American Elsevier, New York, 1968
10 MCKEEMAN, W M , HORNING, J J , AND WORTMAN, D B *A Compiler Generator.* Prentice-Hall, Englewood Cliffs, N J., 1970
11 SHANNON, C.E , AND WEAVER, W *The Mathematical Theory of Communication* U of Illinois Press, Urbana, Ill , 1949
12 WORTMAN, D B A study of language directed machine design Ph D Th , Comptr. Sci. Dep , Stanford U , Stanford, Calif , 1972

RECEIVED OCTOBER 1975, REVISED AUGUST 1976