Palo Alto Research Center

Empirical Estimates of Program Entropy

By Richard E. Sweet

XEROX

Jacobi

Empirical Estimates of Program Entropy

BY Richard E. Sweet

CSL-78-3 SEPTEMBER 1978

See abstract on next page.

KEY WORDS AND PHRASES

entropy, information theory, object code compression, parse tree, pattern matching, Markov source, non-uniform order, empirical study of programs, interpreter, programming style, Mesa

CR CATEGORIES

4.12, 4.13, 4.6, 5.6, 6.22

XEROX
PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road / Palo Alto / California 94304

Abstract

We wish to investigate compact representation of object programs, therefore we wish to measure *entropy*, the *average information content* of programs. This number tells how many bits, on the average, would be needed to represent a program in the best possible encoding. A collection of 114 MESA programs, comprising approximately a million characters of source text, is analyzed. For analysis purposes, the programs are represented by trees, obtained by taking the parse trees from the compiler before the code generation pass and merging some of the symbol table information into them.

A new definition is given for a Markov source where the concept of "previous" is defined in terms of the *tree structure*, and this definition is used to model the MESA program source. The lowest entropy value for these Markov models is 1.7 bits per tree node, assuming dependencies of each node on its grandfather, father, and elder brother (order 3). These numbers compare with an approximate 10 bits per node required for a naive encoding, and an equivalent of 3.2 bits per node of code generated by the existing compiler. Motivated by sample set limitations for higher order models, we derive an entropy formula in which the order is non-uniform.

The non-uniform entropy formulas are particularly suited to trees, where we can now speak of conditional probabilities in terms of *patterns*, or arbitrarily shaped contexts around a node. A method called *pattern refinement* is presented whereby patterns are "grown", i.e., the set of nodes matching an existing pattern is divided into those matching a larger pattern and those remaining. A proof is given that the process always leads to a lower estimate unless the old and new patterns induce exactly the same conditional probabilities. The result of applying this technique to the sample was an estimate of 1.6 bits per node. Further application would reduce this number even more.

Analytic solutions for the error bounds in approximating the entropy of a Markov source are very difficult to obtain, so an experimental approach is used to gauge a confidence figure for the estimate. These calculations suggest that a more accurate estimate would be 1.8 bits per node, with a standard deviation of 13%. This corresponds to an entropy of .54 bits per character of source program.

The methods of this thesis can be used both to define a bound for code compression and to evaluate existing object code.

Acknowledgments

I am grateful to my principal advisor, Cordell Green, for introducing me to the thesis problem, providing me with useful advice, and asking me thought-provoking questions about what I was doing.

Special thanks go to Don Knuth, my patient advisor for many years. Participation in his FORTRAN study in the summer of 1970 kindled my interest in the empirical study of programs. His contributions to this thesis were numerous: his intuition led me to the entropy formula for trees used in the thesis; his prodding inspired me to gain a deeper understanding of several theoretical areas, not all of which made it to the eventual thesis; his comments on earlier drafts were at a level of detail far deeper than is typically expected of a reading committee member.

I had several fruitful discussions with Tom Cover that improved my understanding of information theory. These were of particular help in defining a Markov source of trees and in estimating the error of my entropy calculations.

I appreciate the support of the Xerox Corporation, Palo Alto Research Center, and my many colleagues there who provided programs for analysis and who listened to presentations of early results. Special thanks go to Ed Satterthwaite and Chuck Geschke, who helped me to unravel the mysteries of the MESA compiler, and to Adele Goldberg and Dan Swinehart, who read and discussed early drafts of the thesis. I would also like to thank the text editor implementors, who gave me better than average service when I uncovered bugs.

Finally, I need to thank my wife Janet and my daughter Emilie, who inspired me to finish this thesis and who survived a year of rather unpleasant personality on my part. Thanks also to my mother-in-law, who came to hold the household together during three critical weeks near the end of my research.

Contents

Figures

1. Introduction

1.1. Measuring Program Entropy

The designer of a system should have a good model of how it will be used. For example, much effort in early compilers went into optimizing program constructs that seldom, if ever, occurred in real programs. The modern trend is toward finding techniques that lessen the likelihood of such misdirected effort. A statistical study of the programs that are actually written in a language is a valuable tool for a compiler designer. Its worth can be manifested in several ways: in a more efficient compiler, in a faster program, or in a more compact object program representation.

This thesis deals with estimating the *entropy,* or *average information content* of programs. Such a number will tell a compiler designer just how many bits of information an average program contains, i.e., how small the object program may be and still contain the full program specification. These numbers are usually given in terms of some easily computed dimension of the source programs, for example, length. Such knowledge of the *typical* source program can help the designer to design compact object programs, and can also provide a theoretical framework against which to measure progress.

In recent years, there has been dramatic reduction in the cost of computer hardware, sparked by advances in integrated circuit technology. The current trend is away from large time-shared computers and toward small single user machines. One component that still is a major cost item in computers is *memory.* Those of us accustomed to computers having tens of millions of bits of main storage must learn to "think small." Since most interesting computer tasks require large amounts of memory, it is not uncommon to have a *virtual memory,* in which some of the information treated as main memory is actually residing on secondary storage, such as a disk. In these cases, the running time of programs is largely influenced by the size of the *working set,* that portion of the virtual memory that is being actively referenced. Anything that reduces the size of programs can thereby also increase the speed.

Working set considerations aside, we often are in a position to trade execution speed for program size, and *vice versa*. One longstanding technique used by programmers to reduce size is the interpretation of instructions specially designed for a given task. This is sometimes thought of as "table driven programming", where a single concise table entry can cause a specialized interpreter to execute a large number of instructions. One does, nevertheless, pay a price; running programs interpretively is typically slower than running programs written directly in the machine code. However, exactly what is meant by machine code is becoming less distinct. For some time, the "machine code" that programmers have used has actually been interpreted by programs written for a lower level instruction set, called *microcode*. The task of writing such *microprograms* has been done by a few experts and typically stored in *read-only memory*. In recent years, however, manufacturers have provided computers that store their microprograms in *read-write* memories, allowing a language designer to tailor the perceived machine language to match the constructs of the source language. Using the formalism of information theory, we shall see how the statistical properties of programming language usage can be exploited in the design of an interpretive code.

1.2. Entropy of Programs

Entropy is a quantity associated with a *source*, a process that emits a sequence of symbols (*outputs*) according to some fixed probability laws. Typically, the probabilities associated with a given source output are influenced by the values of previous outputs. When dealing with experimental data, we cannot determine the exact probability laws, but must be content to build a *model* of the source and estimate the entropy of the model. We will carefully construct our model so that the entropy of the model is an overestimate of the true entropy; hence, we know that our lower bound for encoding is a conservative one.

There has been much activity in estimating the entropy of natural language. In an early paper on the subject [Shannon51], several models of English language sources were investigated. Some models had probability rules that depended on the value of the previous few characters or words; such models are called *Markov* sources. One can apply similar models to programming languages. With programming languages, however, we are dealing with a much more controlled *grammar,* so we need not consider the programs as mere strings of source text. We can exploit the *structure* of the statements in a straightforward manner.

One way that programming languages differ from natural languages is that sentences have an underlying nested structure that can be thought of as a *tree.* Modern programming language constructs such as "IF...THEN...ELSE" and "BEGIN...END" allow statements much more deeply nested than even German natural language sentences. The author once wrote a program *profile* facility which produced a neatly indented listing of a program together with statement counts. One of the early users of the system uncovered a "bug" when one of his program statements that was nested over 30 levels deep caused the placement procedure to "indent" the program completely off the right side of the page! Such programs are fairly uncommon, but they do help to point out the treelike nature of computer programs.

In this thesis, we will be concerned with finding representations for programs that allow the tree structure to be exploited when estimating the average information content of an atomic program symbol. The representation chosen is a tree as produced by the parser and clarified by means of the program semantics. What is meant by the entropy of such a model? Standard formulas for the entropy of Markov sources require knowing the probabilities of all m-tuples of source symbols and conditional probabilities based upon all $(m-1)$-tuples, numbers which are difficult to obtain for some m-tuples with any confidence due to the finite nature of our sample set. We will see an alternative formula in which the amount of history used in the calculation is context dependent rather than uniform. This formula has better error bound properties for estimating entropy from experimental data, and is readily generalized to an entropy formula for trees, allowing us to define the concept of a Markov source of trees and estimate its entropy with a reasonably sized empirical sample.

1.3. Survey of Related Work

Empirical Study of Programs

Historically, designers of compilers and languages have had little knowledge of how typical programmers were using a programming language. There has recently been a trend toward empirical study of programs. One of the better known works is by Knuth [Knuth71a], in which he reports on a collection of FORTRAN programs analyzed one summer at Stanford. His bibliography lists some of the earlier works in the field.

The results of Knuth's paper are divided into *static* and *dynamic* statistics. In this thesis, we shall be primarily concerned with static statistics, although Chapter 6 contains a few words about the dynamic case. Knuth's static counts were made by a program that read source programs and maintained a count of occurrences of the various reserved words and operators of the language. The most striking conclusion that can be drawn from these numbers is that actual programs are far simpler than had been previously believed. Compiler writers had prided themselves in generating efficient code for complicated expressions, while in practice, expressions had an average length of only two operands. Knuth found that 68% of the assignments involved no operator at all. Figure 1.1 shows the complexity of expressions in assignment statements where the operators + and - were given one point, * given five, and / given eight points:

Complexity	0	1	2	3	4	5	6	7	8	9
Number	56,751	14,645	1,124	106	267	2,436	1,988	562	2,359	552
Per cent	68.0	17.5	1.4	0.1	0.3	3.0	2.0	0.6	3.0	0.6

Figure 1.1. Complexity of expressions in Knuth's FORTRAN sample.

Further analyses showed that although FORTRAN allows arbitrarily complex *control flow* due to the presence of GOTO statements, all of the tested programs had reasonably simple flow graphs. Since the analysis programs were working with the surface strings of the language and not with parsed programs, they could not tell procedure calls from array accesses, so the interesting statistics about procedure parameters was not available.

Alexander and Wortman made similar studies of XPL programs [Alexander75]. Rather than write a routine to analyze source programs, they modified the compiler to count various statement and expression types. The XPL compiler is a BNF production-oriented one, so this task was easy. Like Knuth, they also analyzed the programs both statically and dynamically. The following enumeration from [Alexander75] shows statically obtained information they found useful for determining resource consumption:

- distribution of constants by type and value;
- distribution of variables by type, value, and point to declaration (lexical nesting);
- distribution of statements by type;
- complexity of expressions and use of operators in expressions;
- distribution of machine instruction emitted by the compiler, also adjacent pairs and triples of instructions;
- distribution of registers, operand addresses, and constants occurring in emitted instructions.

Another interesting item from this paper is a table showing the distribution of numeric constants in their sample programs. Figure 1.2 shows an excerpt from their table, which also indicates the number of bits necessary to represent numbers in the given range by the standard number representation techniques. Their compiler treated unary minus as an operator rather than as a part of a number, so all constants in their sample were positive.

Range (logarithmic)	*Number of bits*	*Number*	*Percentage*	*Cumulative Percentage*
ZERO	1	7762	15.6	15.6
[2**0,2**1)	1	8459	17.0	32.6
[2**1,2**2)	2	3952	7.9	40.5
[2**2,2**3)	3	2986	6.0	46.5
[2**3,2**4)	4	4747	9.5	56.0
[2**4,2**5)	5	4682	9.4	65.4
[2**5,2**6)	6	5908	11.9	77.3
[2**6,2**7)	7	4715	9.5	86.8
[2**7,2**8)	8	4037	8.1	94.9
[2**8,2**9)	9	1372	2.8	97.7

Figure 1.2. Distribution of numeric constants in Alexander's XPL sample.

Figure 1.2 provides the interesting result that more than half of the numeric constants of their sample could be represented in 4 bits. In the sample of programs analyzed below in Chapter 4, the results were even more dramatic: when one considers all the numeric constants either stated directly or resulting from "constant folding", the numbers 0, 1, 2, and 3 account for 52% of the total static usage.

The author made an informal study of programs written in an early version of the MESA language running on the PDP-10. This study was motivated by the trend toward personal computers that caused the effort on that language to be directed toward a small 16 bit machine. In order to conserve space, the language was to be compiled into a compact code that was then to be interpreted. Several studies were made of existing TENEX-MESA programs to give guidance to the designers of the interpreter. One of these studies, which was never formally written up, concerned the complexity of expressions and procedure calls.

The TENEX-MESA compiler parsed the programs into trees one statement at a time and then generated code from the tree in essentially a one pass style. Some of the operations normally associated with the code generation pass of classical compilers took place in the tree building process. For example, when building a tree for an iteration statement (FOR i...), the parser generated the standard trees of an assignment and an if-statement for the incrementation and testing of the control variable. It was easy to intercept the parse trees before the code generation and to "walk" through them, looking at various operators and their operands.

Of particular interest were the assignment statements. Of the 4818 assignments in the TENEX-MESA sample, 17% were of the form *variable ← number,* 10% were *variable ← variable,* and 18% were *variable ← procedure call.* Some 11% were increment or decrement statements, i.e., *variable ← same variable ± number.* Figures 1.3 and 1.4 show statistics taken on the arguments of procedure calls in the sample.

args	0	1	2	3	4	5	>5
count	873	2462	1175	387	98	34	4
per cent	17	49	23	8	2	1	-

Figure 1.3. Number of arguments in TENEX-MESA procedures.

# of args	*variable*	*number*	*field selection*	*array access*	*procedure call*	*other*	*total*
1	1601	329	224	19	153	136	2462
2	1464	368	187	11	100	220	2350
3	601	319	90	7	36	108	1161
4	187	90	61	-	5	49	392
5	62	45	27	1	7	28	170
>5	10	3	5	-	3	3	24
total	3925	1154	594	38	304	544	6559
%	60	18	9	1	5	8	

Figure 1.4. Description of arguments to TENEX-MESA procedures.

Figures such as the above are helpful when deciding on conventions for subroutine linkage and immediate operands, but they need a firmer mathematical foundation on which the designer can base decisions about the relative merit of competing features for the interpreter.

Probabilistic Grammars

In an early paper on the entropy of languages [Grenander67], Grenander discussed the shortcomings of the simple Markov models of language generation such as those of the previously mentioned Shannon paper, in this case, the first order Markov model:

> ... the probability that a certain word is generated given the string of words that precedes it, depends only upon the last word in the string. ... Some defects of this model are obvious. Although it allows for stochastic dependence, it does so only in a very special way *via* interaction of neighboring words. It is easy to exhibit examples in which the dependence has longer span. ... This defect of the model could be expressed by saying that the Markovian dependence is too linear; it attributes too much significance to the linear ordering of words making up the sentence.

Grenander continues with a discussion of the exponentially growing number of probabilities needed for higher order Markov sources, and concludes that is impractical to get estimates of reasonable accuracy for source models of sufficiently high order to capture the span of dependencies in natural language text. He concludes:

... The prospect of such models therefore appears gloomy if we require both a high level of approximation and that we be able to estimate its structure from real data. What is wrong with this approach is clearly that we have generalized in a too mechanical manner. We must introduce the dependence in a way tailored to fit the phenomenon we study.

He then continues by choosing to represent sentences by *derivation trees,* given a context free grammar, and calculate the entropy in terms of probabilities associated with the various productions used in the derivation of sentence. Rather than discuss his work in detail, we will consider two more recent papers in the field.

In a recent paper [Soule74] on entropies of probabilistic grammars, Soule defines a grammar G as a triple $(\mathbf{V}_N, \mathbf{V}_T, \mathbf{R})$ consisting of a set of nonterminal symbols $\mathbf{V}_N$, terminal symbols $\mathbf{V}_T$, and rules $\mathbf{R}$. A rule is written

$$p\colon s \rightarrow \alpha,$$

where $0 < p \leq 1$, s is in $\mathbf{V}_N$, and α is in $(\mathbf{V}_N \cup \mathbf{V}_T)^*$. Denote the set of rules that rewrite s by $\mathbf{R}_s$; for each s, we must have

$$\sum_{r \in \mathbf{R}_s} p_r = 1,$$

where p_r is the probability associated with rule r.

Soule denotes the language generated by G, with starting symbol s, by $L(G, s)$. He chooses to define the entropy in terms of a *derivation,* a (possibly countable) sequence of rule-names that specify the generation of either a sentence in $L(G, s)$, or (in bad cases) a countably long intermediate form. If more than one non-terminal is to be rewritten at some step in the derivation, we will choose to do so in a consistent order, such as always rewriting the leftmost non-terminal. The set of all derivations beginning with a rule in $\mathbf{R}_s$ is denoted by $\Omega(G, s)$. The probability of a derivation d, denoted $P(d)$, is the product of the probabilities associated with all rules in the derivation. The function

$$\mathrm{Gen}_s\colon \Omega(G, s) \rightarrow L(G, s)$$

simply maps a derivation into the word that results when s is rewritten by the each of the rules that make up that derivation.

The probability $P(w)$ of a word w in $L(G, s)$ is defined to be the sum of the probabilities of all derivations d such that $\text{Gen}_s(d) = w$.

The *derivational entropy* of G is a vector $H(G)$ of N components such that for each s in V_N,

$$H(G, s) = \sum_{d \in \Omega(G, s)} P(d) \log P(d).$$

Of more interest than the entropy of derivations is the entropy of words. Soule defines the *sentential entropy* to be the following:

$$Hs(G, s) = \sum_{w \in L(G, s)} P(w) \log P(w).$$

He then proves a theorem that $H_s(G, s) \leq H(G, s)$, with equality if and only if $L(G, s)$ is unambiguous. Soule gives several results relating the mean word length and sentential entropy to various information theoretical concepts such as information rate, and channel capacity.

Thompson and Booth [Thompson71a] had previously considered various schemes for encoding the languages generated by probabilistic grammars. For a probabilistic language L, they defined a *code* to be another probabilistic language C, with terminal alphabet typically {0, 1}, such that there is a mapping of L onto C. In general, one code word in C is the empty code word ε, where $w \in L$ is mapped onto ε if $P(w) = 1$ or 0, i.e., if w is impossible or certain. If $L_\varepsilon \subseteq L$ is defined as

$$L_\varepsilon = \{w \in L \mid w \text{ maps onto } \varepsilon\},$$

then the mapping of $L - L_\varepsilon$ onto $C - \varepsilon$ is a one-to-one mapping.

Thompson and Booth investigated four classes of coding automata that use the probability information to define the mapping from L onto C in such a way as to be optimal under various constraints. They are *character* encoding, *word* encoding, *grammar* encoding, and *parse* encoding. The character encoding is the standard Huffman encoding [Huffman52] obtained by calculating the probabilities of the various characters. The word encoding requires a finite language, but accommodates general ones by approximating the language by a finite one and including an "escape code". The grammar encoding uses the previous k characters of a sentence to determine the encoding of the next character. They show that if the language has a property called

LL(k), then the language **C** is a context-free probabilistic language, and, since there are means of determining the average word length for such languages, the codes can be compared. The parse encoding is similar to the entropy model of Soule; one encodes the sequence of productions from one possible derivation of the sentence to be encoded.

The grammar based models of language encoding are similar to the Markov model we shall be concerned with in later chapters of this thesis, restricted to uniform first order approximations in which only the father of a node in the parse tree affects the conditional probabilities of this node. Furthermore, the trees we shall deal with contain considerable semantic information.

Statistical Models

One of the early mentions of *program entropy* was in a set of unpublished notes written by McKeeman and Horning sometime around 1968-69 [McKeeman68]. Their principal concern was determining the appropriateness of a particular machine organization for a given programming language by seeing which machine yielded the smallest program. They also had some thoughts on theoretical bounds of encoding using an entropy-like formula. They considered how to deal with the problem of variable names:

> ... We clearly do not wish to distinguish, in machine code, programs that differ only by a systematic substitution of identifiers. We propose that we first tabulate all identifiers used in a program and then systematically replace them by a standard set (say X1, X2, ..., X100, etc.). We can reduce the set even more by taking into account block structure and the permissibility of duplicate use of names, or may have to do something more complex if the form of the identifier carries some semantic information (for instance the [FORTRAN] IJKLMN type convention). The essential point, reducing the variability of programs by using a consistent naming convention, is straightforward for any given conventional programming language.

Their means for estimating the entropy of a given program was essentially that used by Soule. Their methods did not take into account any influence of the probabilities of one production caused by the previous ones, although they pointed out that the higher order dependencies should be considered when defining entropy. At the time, they were only considering conventional machine architectures where such conditional probabilitites could not be exploited.

In his thesis [Hehner74], Hehner applied the framework of information theory to the design of computer hardware, both instruction encoding and data encoding. In a more recent paper [Hehner77], he was concerned primarily with the encoding of computer instructions. He found that for his sample, as much as 75% of the space taken by contemporary machine-language representations could be saved. One question that he addressed is the reliability implications of removing redundancy from object programs:

> 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 the error results in a legal instruction, it may be detected indirectly but escape identification, or it may escape detection. . . . 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.

Hehner also discussed the entropy of programs; he represented the program as a *sequence of tokens* in the conventional way (rather than as tree structure), and he determined an entropy estimate in terms of *bits per token*. If there is a large sample of programs, one can hope that the frequencies of the various tokens are representative of programs in general. This is also probably true for pairs of tokens, and, with decreasing confidence, for higher order m-tuples. He defined the entropy in terms of conditional probabilities much as we shall do in Chapter 3, but did not provide any numerical estimates from his sample for the entropy of the tokens.

Hehner then applied the same methodology to the object programs generated by the XPL compiler. The stream of instructions can be thought to have conditional probabilities based on previous instructions just as source tokens do. Since his measurements were all *static*, it is necessary to establish a known context after every label. One very interesting portion of his paper is an analysis of a process he calls *iterative pairing* wherein commonly occurring adjacent pairs of instructions are replaced by a single one. He gave criteria for deciding how much the information content of the program will decrease upon making the replacement (it may actually be increased by the replacement). He then discussed the encoding of instructions using their conditional probabilities based on *m* previous instructions. Figure 1.5 shows the numerical results from encoding his sample by various schemes. The "minimum redundancy" figure comes from applying the Huffman coding procedure to the "360-like" instructions.

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*	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

* bits per original operation, 4.85 bits per compound operation.

Figure 1.5. Code compression in Hehner's XPL sample.

Finally, Hehner showed that in the limit, conditional encoding of instructions is always better than grouping together common operations for code compactness. The principal differences between the work of Hehner and the author's work are in the source of tokens to be encoded, object instructions *vs.* parse tokens, and in the structure imposed upon them, linear *vs.* tree structure.

Program Oriented Encodings

The use of an interpreter of specialized codes has long been a practice by software designers to reduce the size of programs. Recently, there have been some papers on producing computers with architectures well suited to a particular language. Hehner discusses a number of the earlier works in his thesis [Hehner74], one of the better known examples being the Burroughs B5000 series of computers. Now that microprogramming has made computer architecture such a malleable concept, we can expect more work in the field.

Deutsch [Deutsch73] describes an architecture for a machine to execute LISP programs, called MicroLISP. He observes the fact that a typical LISP function references rather few functions and variables, and hence can be encoded by instructions with short address fields that refer to an external table, called the "local name table". In addition, there are a small number of commonly referenced functions stored in a "global name table". These techniques, plus others described in the paper allow object programs of one-third to one-fourth the size of the same programs compiled for the PDP-10 by more conventional means.

Deutsch also presents an argument for program oriented architectures concerning the ease of writing debugging aids:

> MicroLISP has been presented as a machine language, but slight additions would permit unambiguous decompilation into the original S-expression for editing. This approach is only feasible in general when the machine language closely resembles the source code: compilers for conventional machines must rearrange and suppress the original program structure extensively to achieve efficient execution. Interpretive systems, of course, generally do reconstruct the source text from an intermediate representation, often using their knowledge of the program structure to advantage (e.g. indenting to indicate depth of logical nesting).

Wade and Stigall [Wade75] analyze the potential cost savings of several encoding features. They first analyze the grouping of common sequences of instructions into new instructions. This is essentially the "iterative pairing" process of Hehner, and the criteria for improvement are essentially equivalent. They also analyze the situation where the instructions can be broken into several classes, with the interpreter having "modes". They give formulas for the code length improvement as a function of the average number of instructions executed before a "mode switch" instruction is needed.

Foster and Gonter [Foster71] describe a technique for conditionally encoding computer instructions. They observe the common sequences of instructions in object programs and make the following observations:

> ...Given that we have just executed instruction *x*, it is not the case that all possible instructions are equally likely to follow *x*. For example, "load accumulator" is very rarely (in most codes) followed by "enter accumulator." ... Suppose instead of complete generality we decide that in the normal course of events we will allow *N* different instructions to follow a given instruction. One of these must provide an escape mechanism to allow for the unusual program. If *N* is much less than the total repertoire of the machine, we can achieve considerable compression of the op-code field.

To try out their ideas, they took a collection of programs written in assembly language for the CDC-3600. Figure 1.6 shows the percentage of op-code transitions captured by the mechanism as a function of the size of the conditional op code field.

Number of Bits in Op-Code Field	*Number of Successors*	*% transitions captured by successors*
2	3	46.3
3	7	67.5
4	15	84.4
5	31	95.4
6	63	99.7

Figure 1.6. Conditional code efficiency from Foster and Gonter study.

The total number of possible instructions is 142. Clearly conditional encoding of the instructions would reduce the total space required. They discuss the difficulties of programming such a machine in assembly language, primarily in debugging. The modern trend is away from assembly language programming, so this is less of a problem; a debugger that operates on a source language level can be made smart enough to translate the conditional codes into something understandable.

Of course, computer programs do not consist entirely of instructions, they also contain data. For some languages, it is possible to effectively encode these data by techniques exploiting their statistical properties. For the language LISP, the line separating instructions and data is a fuzzy one; when developing programs, one tends to run them interpretively. Clark and Green [Clark77] describe an empirical study of LISP list structures. A computation that they perform that is most related to the content of this thesis is the *entropy* of **car** and **cdr** pointers (relativized to the address of the cell containing the pointer). The common LISP systems for the PDP-10 use 18 bits for each of the pointers. Clark and Green found considerably fewer bits of *information* in the pointers of the programs that they studied.

The statistics of that paper are all static, but in his thesis [Clark76], Clark also obtained dynamic statistics on distribution of values for **car**, **cdr**, and other language features. In addition, he gave algorithms for *linearizing* lists, i.e., rearranging them so that the **cdr** (**car**) points to the very next cell in memory whenever possible. Two thirds of the **car**'s and one fourth of the **cdr**'s do not point to lists at all, in which cases the algorithm does not apply. Nevertheless, one would predict that the entropy estimate of the affected pointer would go down after such an operation. In fact, the entropy estimate of both pointers usually goes down; Figure 1.7 shows the entropy of **car** and **cdr** in the original data, and after either **car** or **cdr** direction linearization. They chose to give separate values for each of five large sample programs.

sample	*original*	*data*		*car direction*	*linearization*		*cdr direction*	*linearization*	
number	*car*	*cdr*	*sum*	*car*	*cdr*	*sum*	*car*	*cdr*	*sum*
#1	9.05	4.81	13.86	7.90	2.37	10.27	9.15	1.13	10.27
#2	10.32	5.16	15.48	8.63	3.04	11.66	9.88	1.98	11.86
#3	9.73	5.08	14.80	8.35	2.38	10.73	9.56	1.28	10.84
#4	6.39	3.46	9.86	4.96	2.39	7.35	6.13	1.17	7.30
#5	9.93	3.33	13.26	8.16	2.07	10.23	9.19	.99	10.17

Figure 1.7. Pointer entropy from Clark and Green study.

The reason that the entropy estimate for **cdr** dropped more dramatically upon linearizations is that the **cdr** pointed to another list a much higher percentage of the time than did the **car**. Also, the **car** direction linearization tended to put the **cdr** cells nearby as well, so either form of linearization produced about the same numbers. Clark and Green point out that the actual entropy of the pointers should be dependent only on the semantics of the language and not on the particular representation chosen. Thus the smaller entropy numbers obtained by linearizing the lists more accurately reflects the entropy of the pointers.

1.4. Summary of Thesis Results

In this thesis, a large sample of programs, amounting to approximately a million characters of source code, is analyzed, using new techniques for analysis. In the first place, the representation chosen for study is a tree, for reasons described in Chapter 2; an *entropy* number is calculated in terms of *bits per node*, where the nodes are those making up the trees. Since there is a reasonable correspondence between input tokens and tree nodes, these numbers could also be stated in terms of more familiar dimensions of programs. By making several extensions and generalizations to the entropy formulas for sources of strings, the entropy of trees is defined in terms of the probability distribution of nodenames occurring at any given point in the tree. In most cases, these probabilities depend upon the values of various neighbors of the given node.

Several mathematical models are presented to capture the dependencies of nodes upon their neighbors. The first class consists of the *uniform Markov* models, in which there is a fixed collection of neighbors determining the conditional probabilities of a node. Figure 4.2 contains detailed results of the entropy estimates using these models; Figure 1.8 is a summary of the entropy values.

Neighbors Affecting Probabilities	*Entropy Estimate*
none	4.75
father	3.18
elder brother	3.16
grandfather, father	2.80
father, elder brother	2.05
grandfather, father, elder brother	1.66

Figure 1.8. Summary of uniform Markov entropy estimates.

The finite nature of the sample set causes problems when using these uniform formulas. For a model using m neighbors, the formulas require probabilities for all $(m+1)$-tuples, and conditional probabilities based on all m-tuples. For those m-tuples with only a few occurrences in the sample, there is a high likelihood of error in the probability estimate.

Dissatisfaction with the uniform models motivated the derivation of a new *non-uniform* entropy formula wherein nodes are conditioned on various collections of neighbors depending upon the context. The collection of neighbors that influence the probability distribution of a place in the tree is called a *pattern*. An initial set of patterns is generated by a modification of the uniform Markov entropy models. A procedure for obtaining a larger set of patterns, called *pattern refinement*, is given, with a proof that the entropy of the enlarged model is a better estimate of the true entropy, except in a specialized case, where it is not worse. Figure 1.9 shows the entropy estimate both before and after enlarging the pattern set. One cannot compare the size of the patterns exactly with the number of conditioning nodes in the uniform models since the patterns specify not only who the father is, but which son position a particular node occupies. Nevertheless, the number of nodenames specified in the pattern is given in the figure. The last two columns relate the estimate to more familiar quantities.

	number of patterns				*Source*	*Equivalent*
nodenames specified:	1	2	3	*entropy estimate*	*bits per token*	*bits per character*
initial set	154			2.1	3.1	.63
final set	152	41	75	1.6	2.4	.48

Figure 1.9. Non-uniform Markov entropy estimates.

The reduction in the estimate is not as dramatic as that of the uniform Markov estimates, partly because the initial set is already well chosen; furthermore, a conservative approach was taken of only increasing the order in those cases where there was a reasonable sample for estimating the probabilities.

Analytic solutions for the error bounds are very difficult to derive when dealing with Markov processes. In order to obtain some confidence in the probabilities that determine the entropy of the model, the sample was therefore divided into several pieces, with a portion of the sample, called a *training sample*, used to postulate an encoding for the trees. The remaining *test sample* can be encoded according to these codes and an *average code length* computed. If the statistics of the sample are uniform over the pieces, this average length will be close to the entropy. By applying this process repeatedly with each piece having a turn as the test sample, we obtain a more likely estimate of the entropy for the sample to be 1.8 ±13%.

1.5. Guide to the Reader

Chapter 2 discusses ways of representing a program as a tree structure. A sample program is given which is used throughout the thesis whenever examples of representation or encoding are needed. Details of the representation ultimately chosen are presented, together with diagrams of trees drawn from the representation of the sample program.

Chapter 3 contains most of the formal mathematics of the thesis. It contains definitions of all the information theory terms used, and a discussion of *trees* as outputs from a Markov source. The *non-uniform* formula for entropy of a Markov source is derived, and a generalization of the formula is made for trees, taking the structure of the trees into account when defining the concept of "previous". Finally, there is a proof that the *pattern refinement* procedure, described further in Chapter 4, always yields an improvement in the estimate (although increased experimental error may outweigh the improvement).

Chapter 4 contains the experimental results. The sample set, comprising approximately a million characters of source programs, is described. Results are given for the entropy of Markov source models of various orders, using the uniform formula for entropy. The patterns used in the more general entropy formula are described, and sample results are shown from the entropy estimate determined from the original set of patterns. The new methodology called *pattern refinement,* which obtains a lower estimate by enlarging the set of patterns, is discussed, together with sample output from the process applied to the sample set of programs. Finally, it is shown how the tools developed for the pattern refinement procedures can be used to evaluate the merit of various *program transformations* interactively.

Chapter 5 discusses the potential error in the entropy estimate. An experimental test of sample variability is described, together with results of applying the test to actual data. Another formula for the entropy estimate from a finite sample is presented which is faster and easier to compute than those of Chapter 3.

Chapter 6 discusses application for the methodology of the thesis and possible extensions and suggestions for further research.

Appendix A contains an alphabetical listing of the nodenames occurring in the program tree representations defined in Chapter 2.

Appendix B contains a few comments about how some of the non-obvious algorithms of the thesis were implemented, together with a few general pointers for anyone who wishes to apply similar analyses to other programming languages. It also contains a description of the algorithm used to format the tree diagrams that appear in various figures of the thesis.

Appendix C contains detailed statistics for both the initial pattern set and the final pattern set. They are shown sorted by pattern number, by entropy contribution, and by pattern description (alphabetically). These statistics reveal a great deal about the nature of "typical" programs, since they show the conditional probabilities of a wide variety of language constructs.

2. Deciding on a Data Representation

The programs analyzed in this thesis were written in MESA [Geschke77], a language under development at Xerox Palo Alto Research Center, being implemented on a 16 bit minicomputer. For the purposes of this thesis, one can consider MESA to be a dialect of PASCAL [Wirth71].

Our goal is to estimate a bound for the space required to hold an object program, which we will do by studying a collection of existing programs. First, we must decide what representation of the programs should be adopted for study. Any representation that retains enough information to generate the object program is a legitimate candidate, but some representations allow a lower estimate, or make more intuitive the interpretation of partial results. Therefore, let us consider several possibilities.

2.1. Representations not Employed

The work of Thompson and Booth involving probabilistic languages used the *grammar* of the language to draw conclusion about encoding efficiency and hence entropy. This approach has some attractiveness because of the large body of theory involving grammars. However, the existing grammar used by the parser in the existing compiler has many nonterminal symbols that are present only to allow its specific parsing method to work, and do not really bear information. This grammar is not the proper one for incorporation into such a theoretical framework. Moreover, there are program dependencies which are *semantically* based, which would not be considered if we deal only with syntax.

Hehner used a related representation by investigating the *sequence of tokens* that make up a program. This is an improvement over surface strings, but does not capture nested dependencies where there is no adjacency of source text. He also studied the program *object code*, a representation in which too much encoding has already taken place to judge the true entropy of the language.

2.2. Parsed Structure as Representation

In any language such as MESA (or ALGOL) that allows nested statements, a program can be viewed naturally as a *tree*. Such a representation allows the control structure and inter-statement dependencies to be easily manipulated by analysis programs. A tree representation has the advantage of being less tied to the surface structure of the particular programming language than most other choices. Parsed ALGOL programs and LISP programs look quite similar.

Fortunately, the MESA compiler is a ready source of trees for programs. The compiler is written in the classical manner, with multiple passes operating on a program. It represents the intermediate stages as a tree. The initial tree is constructed by an LALR parser, and succeeding passes modify the tree in light of increasing knowledge, or extract information into symbol tables, literal tables, etc. The final passes generate code from the entire collection of information. In the discussion below, references to "the compiler" are to this multipass system. Other compilers would not necessarily perform the same actions at the various passes.

Examples of language and encoding features discussed in the remainder of this thesis will be drawn from the following procedure, paraphrased from a stream input-output program. It is not really necessary to understand the operation of this program, we shall be concerned only with superficial aspects of its syntax.

```
WriteString: PROCEDURE [s: STRING] =
  BEGIN
  c: CHARACTER;
  i: INTEGER;
  FOR i IN [0..s.LENGTH) DO
    c ← s[i];
    WriteCharacter[c];
    ENDLOOP;
  IF s.LENGTH ≠ 0
    THEN StartofLine ← (c = CR);
  END;
```

Figure 2.1. A sample MESA program.

To aid readers in understanding the syntax, a few explanatory comments are in order.

- *id : type* is a declaration of *id*. The type of `WriteString` is a procedure that takes one parameter of type STRING. The "=" in the declaration equates `WriteString` to the following procedure body.
- *id . id'* is a field selection operation. If *id* is a POINTER, the expression refers to the *id'* field of the RECORD to which *id* points. If *id* is of type RECORD, the expression refers to the *id'* field of *id*. The predefined type STRING is a pointer to a record containing a LENGTH field and the actual string text. The construct `s.LENGTH` is the length of the string `s`.
- [*exp..exp'*) is a half-open interval from *exp* up to but not including *exp'*. The FOR statement is executed for `i` taking on each value in the range.
- Procedure arguments, like array indices and character selections, are enclosed in square brackets [].

`WriteString` is a simple procedure that writes a string by writing each character in turn. It finally checks to see whether the last character written is a carriage return, and if so, sets a BOOLEAN flag saying that the output device is currently at the beginning of a new line. Figure 2.2 shows the statement list of the above procedure body as parsed by the first pass of the compiler.

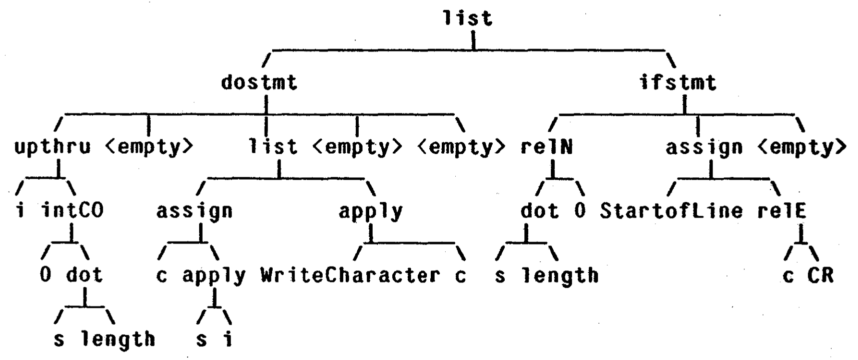


Figure 2.2. Portion of a parse tree generated by Pass 1 of the Compiler.

In the remainder of the thesis, nodenames taken from trees will be underlined. For those readers not familiar with parse trees, Figure 2.2 may be somewhat difficult to read at first, but the meaning is actually quite straightforward. The procedure body is a list of statements, in this case a dostmt and an ifstmt. The subtree rooted by dostmt is generated for the source construct "FOR i...". Notice that some of its sons are empty, denoted "<empty>". This is because the language allows several forms of iteration statement, with, for example, WHILE clauses at the beginning or end, or with nonstandard exits. The parser generates the same general tree for all such statements, merely leaving the subtree "empty" for those features not present in the given source statement. The ifstmt tree is easier to understand. It has three sons, the test, the "THEN" part and the "ELSE" part. In the example program, there is no "ELSE" part. Appendix A gives a complete list of nodenames, together with number of sons and the source constructs which give rise to the node.

Such a tree representation of the program clearly does not discard any important information. In fact, a quite successful program has been written to produce neatly indented listings that takes these trees and unparses them, using the program structure to control line breaks, etc. For analysis, however, it is best to wait until later passes of the compiler have had a chance to embellish the tree.

Knuth's study of FORTRAN programs used the surface strings as a representation [Knuth71]. We could produce similar results from Pass 1 trees, and would, in fact, have an easier task than he did, since statement typing and decomposition has already been done by the parser. The study of TENEX-MESA programs described in Chapter 1 was done using Pass 1 trees.

Experience with Pass 1 trees suggests several problems. Some syntactically similar program constructs must be differentiated by means of the semantics. Thus the string character selection and procedure call in the dostmt of Figure 2.2 are both parsed as apply by Pass 1. Later passes will convert the subtrees to seqindex and call, respectively.

The names of variables allow us to tell very little about variable usage. In a strongly typed language, two identifiers of the same name appearing in the same expression might well have entirely different semantics. The earliest time to obtain a tree in which variable names have been disambiguated is at the end of Pass 3, when each name is replaced by an index into the compiler's symbol table. This seems like a good source of trees for analysis. Indeed, the initial programs developed for this thesis manipulated Pass 3 trees.

Recall that we wish to encode programs using a minimum of space. As we shall see in Chapter 3, increasing redundancy can help to make more compact encodings. Programmers know that each program or procedure contains a few variables that are used frequently and some others that are used only a few times. Since the actual names of the variables do not affect the execution of the program, we can increase redundancy for analysis by renaming the variables of each procedure to be the same, ordering by frequency of reference. We are concerned with minimizing object code size, so static frequency is a sufficient measure.

One cannot tell the type of a variable from its name. This information is available elsewhere in the parse tree in the declarations, but such remote references are not very convenient for analysis. Similarly, some "variable" names actually refer to named constants, such as CR in the example, which is declared earlier to be equal to the ASCII code for carriage return. For analysis, we wish to replace such references with their values.

2.3. Tree Representation Employed in Analysis

It is not until Pass 4 of the MESA compiler that named constants are replaced by their value. As an added bonus, "constant folding" has taken place, i.e., arithmetic on constants has been performed at compile time. However, by this time, the declarations have been completely processed and removed from the trees. This poses a problem: if we are to benefit from the processing done by Pass 4, we need to encode some information from the symbol table. Otherwise, we would be lacking information about lengths of variables, etc.

Putting the Variable Type Information Back into the Trees

The symbol table entries for variables contain all the relevant information about type. One way to retain the information of the program would be to encode the parse tree and symbol table separately. This would be similar to the approach used with Pass 3 trees, since the declaration trees were easily separable. This would retain a shortcoming that was observed in the statistics obtained from the Pass 3 trees: we know what operations are executed on variables, but we don't know the types of variables involved. That is an oversimplification; the type information is in the tree, but is not located conveniently to the actual variable occurrences. We solve this problem by placing information from the symbol table back into the tree. Each instance of a variable in a tree is replaced by the collection of information necessary to generate a proper object program from the tree. This puts more information into the tree than is absolutely necessary, but it simplifies investigations of operations needed to execute actual programs.

Variables can be divided into four classes, depending upon where they are declared. For reasons related to the runtime structure, we will call the class name a *frame.*

1. `global` -- a variable declared outside of procedure bodies.
2. `local` -- a variable local to the given procedure.
3. `field` -- a named field of a record.
4. `entry` -- a procedure entry point.

In our discussion above, we saw that renaming the variables within each procedure increases redundancy. It is sufficient to assign sequence numbers to the variables, ordered by frequency of use. Fortunately, the compiler does just that in assigning addresses, so that we may simply use the *address* of each variable within its procedure frame as its sequence number.

Some variables, primarily `field` variables, begin at arbitrary places within a word. Thus, a variable must have a *bit offset.*

Not all variables occupy a single full word of memory. For example, the type BOOLEAN is defined as a single bit. A *length* is therefore needed for a variable. The compiler unpacks non-field variables into whole words, but for analysis purposes, the length used is the minimum number of bits necessary to hold a value of the given type. In a language supporting floating point operations, additional information would have to be included to distinguish what operations were used with the variable.

To summarize, each instance of a variable in the Pass 4 parse tree can be replaced by a tree with root var and four sons, corresponding to *frame*, *address*, *bit offset*, and *length*. For example, the variable c, an 8-bit character variable in the local frame, was assigned by the compiler to word 2 in the frame. References to c are replaced in the tree by

```
      var
   ____|____
  /    | |  \
local  2 0  8.
```

This makes the trees rather cumbersome to read, but it puts the pertinent variable information in a form easily manipulated by tree matching procedures.

Marking Literal Constants in the Trees

The nodes that provide type information for variables introduce a great quantity of numbers into the trees. We are interested in knowing what numbers are used by programmers as literal constants; therefore, two nonterminals were invented in the analysis programs for specifying numerical and string constants. All numerical constants are placed below num nodes, and all string constants are placed below str nodes. This mechanism provides a positive test (under num), instead of just a negative test (not under var) for determining the presence of a literal. It also helps to avoid fragmentation of common patterns, as we will see in our later discussion of patterns. Assuming that *i* is allocated the first location in the frame, the statement i ← 1 would compile to the tree

```
        assign
       ___|___
      /       \
     var      num
      |        |
   ___|___     |
  /   | | \    |
local 0 0 16   1.
```

Dealing with Arbitrarily Long Lists

Several schemes for dealing with the variable length list nodes were considered. Although the programs studied are written for a 16 bit computer, the analysis routines are based on a compiler that runs on a 36 bit machine. In this compiler, each non-terminal tree node has a field in the node header that gives the son count. This is redundant in all cases except list since all other nodes have a degree that is known *a priori*. Hence, only list nodes need an explicit means for determining son count. In the Pass 3 trees, the convention was an invented node listterm as the final son of each list node. This is equivalent to parentheses with list as the opening bracket and listterm as the closing one. One of the problems with that scheme was the difficulty of relating the length of lists to other context information. Several other, forgettable schemes were considered when the decision was made to use Pass 4 trees, but the one chosen allows the lengths to be easily related to context in the same manner as are other tree nodes: the first son of list is a count of the remaining sons. In the experimental data, 85% of the list nodes have fewer than 5 sons, so considerably fewer than 16 bits are required on the average for the son count. Results from the entropy studies were used by the compiler designers to decide on an encoding of list in versions of the compiler that run on the 16 bit machine.

Example of a Transformed Tree

Figure 2.3 is the transformed Pass 4 tree of the dostmt of Figure 2.2. Note that the compiler has used the semantics of s and WriteCharacter to convert the apply nodes of Figure 2.2 into the proper seqindex and call nodes.

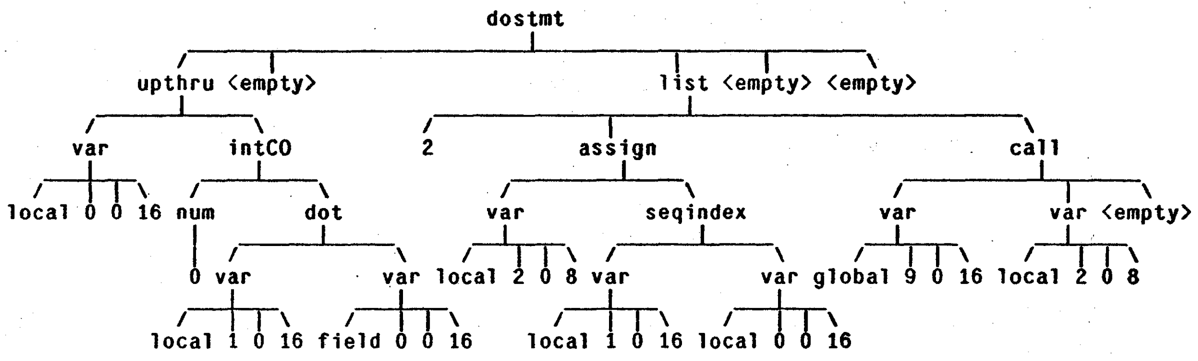


Figure 2.3. Portion of a converted Pass 4 tree.

In these trees, there is enough information to generate the same object code as the compiler does from the original program. One could not, however, recover the exact surface strings of the original, since the variable names are not in the tree. Furthermore, the user may define types that are used for compile time type consistency checking, but are the same at machine level. The information is moved into the trees after the compiler has done such checking.

2.4. Summary of Program Representation

The empirical study of programs described in the remainder of this thesis represents a program by the parse tree obtained from the compiler after Pass 4, during which declarations are removed and expressions involving only constant values are replaced with their computed value. The analysis routines further replace variables by subtrees giving their address and length, and mark literals by inserting nonterminal nodes num and str above them. Variable length list nodes are encoded by placing in their first son position a count of the remaining sons.

3. Finding an Entropy Model

3.1. Some Definitions from Information Theory

In order to discuss the methodology used for program analysis, it is necessary to present a few definitions and results from information theory. Let us consider a few definitions [Abramson63].

> **Definition:** *Let E be some event which occurs with probability $P(E)$. If we are told that the event E has occurred, then we say we have received*
>
> $$I(E) = \log \frac{1}{P(E)} = -\log P(E)$$
>
> *units of information.*

The unit of measure depends upon the base of the logarithm. Typically, $\log_2$ is used and the unit of measure is the *bit.*

> **Definition:** *Let $S = \{s_1, s_2, \ldots, s_q\}$ be a fixed finite source alphabet. A discrete information source is a process that emits a sequence of source symbols according to some fixed probability law.*

The simplest type of source in one in which successive symbols are statistically independent. Such a source is called a *zero-memory source* and is characterized by the alphabet S, and the probabilities

$$P(s_1), P(s_2), \ldots, P(s_q)$$

with which the symbols occur. We will often refer to a source by the name of its source alphabet when this can be done without danger of confusion.

> **Definition:** *Let S be a zero-memory source with a given probability distribution of the symbols. The entropy of S, denoted $H(S)$, is defined to be the average amount of information per source symbol:*

$$H(S) = -\sum_{i=1}^{q} P(s_i) \log P(s_i).$$

Entropy can be thought of as a measure of uncertainty. If a source usually emits the same symbol, there is little uncertainty, and also by the above definitions, the source has a low entropy. While the definition in terms of logarithms may seem somewhat arbitrary, it can be shown (see [Ash65] Section 1.2) that any uncertainty measure satisfying a particular set of reasonable axioms is equal to H within a constant multiplicative factor. We will later consider the definition of entropy for discrete information sources with more sophisticated probability laws.

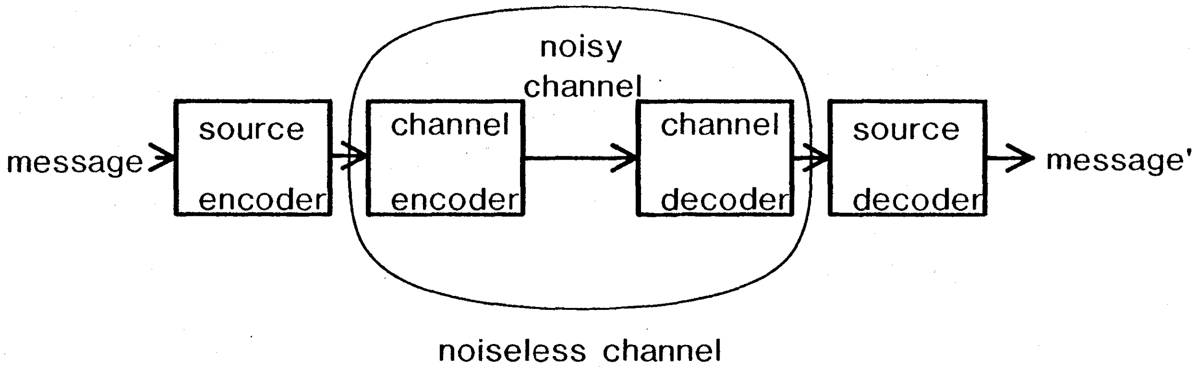


Figure 3.1. A model of information transmission.

The principle application of source information and entropy is to the problem of source *encoding*. We can envision a process where source messages are converted into some sort of *code*, transmitted across a *channel*, and decoded on the other side. There are generally two dissimilar kinds of encoding taking place: *source encoding* and *channel encoding*. In the former, the redundancy of the source output is being exploited to allow shorter average code lengths. In the latter, redundancy is often added so that errors induced by noise on the channel may be detected or corrected. As a simple example, parity is added to teletype characters. Often the channel encoding and decoding is "factored out" to yield an abstraction called a *noiseless channel*. Figure 3.1 is a block diagram of the transmission process. The criterion for source encoders and decoders is that no relevant information is lost between message and message'. Our goal is to investigate encoders for programming languages that minimize the average object code length. Therefore, the relevant information is that which is necessary to generate correct object programs.

A code in which each codeword can be deciphered as soon as all bits of it are received is called an *instantaneous* code. This is a property that we desire for object codes. A fundamental theorem of information theory [Shannon48], states that no instantaneous encoding of a source may be found which requires on the average fewer bits per source symbol than the entropy of the source. Therefore, we wish as small a value for the entropy as is possible. There are schemes that allow encodings which are quite close to the entropy in average code length [Huffman52] [Pasco76].

3.2. Markov Sources

For our purposes, it is too restrictive to consider only zero-memory sources. In many cases, the probability distribution of source outputs depends upon the values of recent previous outputs. Such sources are called Markov sources, and they require the concept of conditional probability for their analysis.

Let S be a source. We will need some notation to describe the sequential nature of the source outputs. We will let $p_{ij} = P\{s_i s_j\}$ denote the probability that the symbols s_i and s_j appear as successive outputs from S. This can be extended to higher order n-tuples, where

$$p_{i_1 i_2 \ldots i_n} = P\{s_{i_1} s_{i_2} \ldots s_{i_n}\}$$

denotes the probability that the n specified outputs occur together. We will use the notation $p_{j|i} = P\{s_j | s_i\} = p_{ij} \,/\, p_i$ for the *conditional probability* that symbol s_j will be output given that the previous output was s_i. For sources where there are dependencies on more than one previous symbol, we similarly let

$$p_{j|i_1 i_2 \ldots i_n} = P\{s_j | s_{i_1} s_{i_2} \ldots s_{i_n}\} = \frac{p_{i_1 i_2 \ldots i_n j}}{p_{i_1 i_2 \ldots i_n}}$$

denote the probability of s_j, given that the n preceding were as specified. We can now define the order of a Markov source.

Definition: *Let S be a source with alphabet $\{s_1, s_2, \ldots, s_q\}$ in which the occurrence of a source symbol s_i may depend upon at most m preceding symbols. Such a source is called a Markov source of order m. It is specified by giving the alphabet and the set of conditional probabilities*

$$p_{j|i_1i_2\ldots i_m} \text{ for } j = 1, 2, \ldots, q;\ i_t = 1, 2, \ldots, q.$$

3.3. Entropy of a Markov Source

It is possible to define the concepts of information and entropy for Markov sources in a fashion similar to that for zero-memory sources.

Definition: *Let S be a Markov source. The conditional entropy of S is defined by*

$$H(S|\, s_{i_1}s_{i_2}\ldots s_{i_n}) = -\sum_j p_{j|i_1i_2\ldots i_n} \log p_{j|i_1i_2\ldots i_n}.$$

The entropy of the source S of order m is defined by

$$H(S) = \sum_{i_1i_2\ldots i_m} p_{i_1i_2\ldots i_m} H(S|\, s_{i_1}s_{i_2}\ldots s_{i_m}).$$

There is another formula for the source entropy that is sometimes more convenient to use. It is obtained by rearranging the summation above and using the definition of conditional probability. In order to avoid notational overload, we will go through the rearrangement for $m = 1$, and simply state the result for general m.

$$\begin{aligned} H(S) &= \sum_i p_i\, H(S|\, s_i) \\ &= \sum_i p_i \sum_j -p_{j|i} \log p_{j|i} \\ &= -\sum_{ij} p_i p_{j|i} \log p_{j|i} \\ &= -\sum_{ij} p_{ij} \log p_{j|i}, \text{ by definition of conditional probability} \end{aligned}$$

In the case of general m, H is computed by a sum over all $(m+1)$-tuples:

$$H(S) = -\sum_{i_1i_2\ldots i_mj} p_{i_1i_2\ldots i_mj} \log p_{j|i_1i_2\ldots i_m}.$$

In the remainder of the thesis, we will have occasion to refer to either of the above formulas for $H(S)$ as *uniform* formulas. This name is chosen because the formulas uniformly use tuples of a given size. This is contrasted with formulas derived below that use an amount of history varying with context.

We will be dealing with experimental data in which we will never be sure of the order of the source. It is possible to perform an entropy calculation on a source as if it were of order less than its true order, so we want to know something about how this entropy estimate relates to the true entropy. This question is answered by the following theorem.

Theorem 3.1: *Let S be a Markov source of order m. Suppose that we assume an order $k < m$, and calculate an entropy value $H_k(S)$,*

$$H_k(S) = \sum_{i_1 i_2 \ldots i_k} p_{i_1 i_2 \ldots i_k} H(S \mid s_{i_1} s_{i_2} \ldots s_{i_k}).$$

The sequence of values $H_k(S)$ is monotone nonincreasing, i.e.,

$$H_k(S) \geq H_{k+1}(S) \geq H(S).$$

Note that by definition, $H_k(S) = H(S)$, for $k \geq m$.

Proof: The proof is straightforward, and found in any information theory book. Theorem 3.2, proved below, is a generalization of this theorem. For a explicit proof of Theorem 3.1, the reader is directed to Ash, Theorem 1.4.5. [Ash65].

Although we will not be concerned with any such sources, it is possible to have a Markov source of infinite order. In that case, the entropy of the source is defined to be the limit of the sequence $\{H_k(S)\}$.

3.4. Considering Trees as the Output of a Markov Source

Traversal Order

Consider a source that emits nodes from trees of the form described in Chapter 2. One method would be to traverse the tree in *preorder* (father, then sons from left to right), emitting each nodename as it is reached. All nodes except list have a fixed number of sons, and the first son of list is its son count, so clearly the tree structure could be reconstructed from this linearized representation. Given the sequential nature of most programming languages, such an order is well suited for encoding program trees. However, for the purposes of entropy definition, any well defined algorithm that eventually emits all nodes of a tree is sufficient. One general scheme would use an arbitrary collection of previously emitted nodes to determine the order in which to visit the sons of a node. A process reconstructing the tree would use the same amount of context to determine which son receives the incoming nodes. It should be obvious that *no* traversal will cause the most recently visited nodes to be those that most influence the probabilities at the next node. In the next section, we will deal with this problem.

The mathematical models employed in analyzing Markov sources assume a non-terminating supply of outputs. For tree analysis, we will assume that when our source reaches the end of one program tree, it starts sending another one. The space of possible programs is clearly infinite, but we will hopefully have a large enough sample to draw conclusions about the microscopic nature of programs in general.

Meaning of "Previous" for Trees

In Markov sources of strings, the concept of "previous" means just what one might expect: the output just seen. When we extend the concept to trees, we will mean not the node just traversed, but a "nearby" node at a location defined in terms of the tree structure. A small example will help us to see why the standard sense of *previous symbol* is not suitable for trees. The tree in Figure 3.2 represents the IF statement of the program from Figure 2.1.

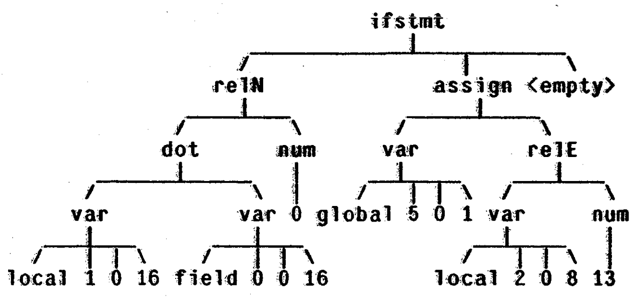


Figure 3.2. Tree representation of an IF statement

Consider the position of the assign node in the tree. From a preorder traversal, the preceding 6 outputs from the source are var, field, 0, 0, 16, num, and 0. Intuitively, the appearance of assign is more influenced by the fact that its father is ifstmt, and that it is the second son. There might also be an influence from the presence of a test for inequality (relN), but the most recent nodes, the expression on the right of the relN test, have little influence on the output. It seems best to chose the *previous* node from the set of chronologically previous ones in terms of the tree structure. For example, in Chapter 4, the sample data is used to model a source of order 3 with the probability distributions conditioned on the *elder brother*, the *father*, and the *grandfather*.

Experience with a Markov Model for Tree Entropy

The formalism described above was applied to a set of sample programs in order to estimate the entropy of a tree node. The results are summarized in Figure 4.2. A fixed traversal order of preorder was used in all cases. Various candidates for *previous* were tried from the set of siblings and ancestors of a node. Estimates were made for orders of 1, 2, and 3. The probabilities of the various $(m+1)$-tuples were estimated by their relative frequency in the sample. There were several shortcomings in this approach that led to the entropy formulation described later in this chapter.

- Any choice of *previous* seemed better in come contexts than in others. No set of previous positions was uniformly superior to all others.

- When the order of the approximation is m, we need estimates of the probability of $(m+1)$-tuples and conditional probabilities based on m-tuples. As m gets large, the number of occurrences of a given m-tuple in the sample becomes quite small. No formal error analysis of the results was achieved, but an uncomfortably large portion of the entropy estimate came from $(m+1)$-tuples that occurred only a few times.

Fortunately, the above problems can be ameliorated by recasting the entropy formula in a form that takes the context into consideration when determining the order.

3.5. Non-uniform Formula for Entropy of a Markov Source

Informal Statement of the Formalism

The order of a Markov source, say *m,* is the *maximum* number of previous outputs that influence the next output. In most actual sources, there are sequences of outputs where the next output is influenced by fewer than *m*. For example, in a simple model for an English language source, one might consider previous words only back to the beginning of the current sentence.

We wish to compute approximations $H_k(S)$ for increasing values of *k*, but our goals are best served by a formula that does not require extending *k*-tuples to *k*+1-tuples uniformly. Before deriving such a formula, it will be helpful to consider a small example.

Example

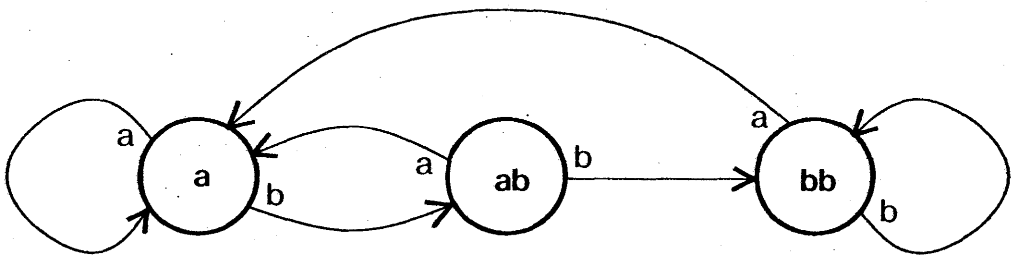


Figure 3.3. State diagram of a simple Markov source.

The state diagram of Figure 3.3 defines a simple Markov source. Associate probabilities with each of the arrows, subject to the constraint that the sum of the probabilities for all arrows leaving a circle is unity. The alphabet of S is $\{a, b\}$, and the order is 2. However, if the previous output was *a*, the probabilities are not affected by what the output was before the *a*. In other words, $P\{y| xa\} = P\{y| a\}$, for all $x,y \in S$. Clearly $H(S| xa) = H(S| a)$, for all $x \in S$. Consider the sum that defines $H(S)$,

$$\begin{aligned}H(S) &= \sum_{xy} p_{xy}\, H(S|\, xy)\\ &= p_{aa}H(S|\, aa) + p_{ba}H(S|\, ba) + p_{ab}H(S|\, ab) + p_{bb}H(S|\, bb)\\ &= p_{aa}H(S|\, a) + p_{ba}H(S|\, a) + p_{ab}H(S|\, ab) + p_{bb}H(S|\, bb)\\ &= (p_{aa} + p_{ba})H(S|\, a) + p_{ab}H(S|\, ab) + p_{bb}H(S|\, bb)\\ &= p_{a}H(S|\, a) + p_{ab}H(S|\, ab) + p_{bb}H(S|\, bb)\end{aligned}$$

which the reader can see is a sum over the *states* of the Markov source of the product of a state probability and the entropy given that state. Using such a formulation for the entropy requires us to deal with m-tuples only in those cases where all m previous outputs affect the value of the current output.

Entropy Formulation for a Markov Source

Let S be a Markov source of order m with source alphabet $\{s_1, s_2,\ldots, s_q\}$. In order to simplify our notation, we will let **boldface** letters denote vectors, typically of order m. If $\mathbf{j} = (j_1, j_2,\ldots, j_m)$, we shall use the abbreviation $s_{\mathbf{j}}$ for the m-tuple of alphabet characters $(s_{j_1}, s_{j_2},\ldots, s_{j_m})$. Let $p_{\mathbf{j}}$ be the probability of occurrence of the m-tuple $s_{\mathbf{j}}$, and let $p_{i|\mathbf{j}}$ be the conditional probability of s_i given that $s_{\mathbf{j}}$ occurs immediately before. The entropy of S is defined by

$$\begin{aligned}H(S) &= \sum_{\mathbf{j}} p_{\mathbf{j}}\, H(S|\, s_{\mathbf{j}})\\ &= -\sum_{\mathbf{j}} p_{\mathbf{j}} \sum_{i} p_{i|\mathbf{j}} \log (p_{i|\mathbf{j}}).\end{aligned}$$

Suppose now that dependencies of order m are not needed throughout the entire range of source output. That is, suppose that there is some p-tuple $\mathbf{k}$ such that for all $(m\text{-}p)$-tuples $\mathbf{r}$, and all $i \in \{1, 2,\ldots, q\}$,

$$p_{i|\mathbf{rk}} = p_{i|\mathbf{k}}.$$

The sum defining $H(S)$ can be partitioned into that portion of vectors $\mathbf{j}$ ending in $\mathbf{k}$, and all the rest. Consider now the portion containing $\mathbf{k}$.

$$\begin{aligned}
&\sum_{\mathbf{r}} p_{\mathbf{rk}}\, H(S|\, s_{\mathbf{rk}})\\
&= -\sum_{\mathbf{r}} p_{\mathbf{rk}} \sum_{i} p_{i|\mathbf{rk}} \log\, (p_{i|\mathbf{rk}})\\
&= -\sum_{\mathbf{r}} p_{\mathbf{rk}} \sum_{i} p_{i|\mathbf{k}} \log\, (p_{i|\mathbf{k}})\\
&= -\sum_{i} p_{i|\mathbf{k}} \log\, (p_{i|\mathbf{k}}) \sum_{\mathbf{r}} p_{\mathbf{rk}}\\
&= -p_{\mathbf{k}} \sum_{i} p_{i|\mathbf{k}} \log\, (p_{i|\mathbf{k}})\\
&= \ p_{\mathbf{k}}\, H(S|\, s_{\mathbf{k}})
\end{aligned}$$

Hence we see that in calculating the entropy of a source of order m, the sum over all m-tuples can be replaced by a sum over vectors of length$\leq m$.

3.6. A Non-uniform Entropy Formula for Trees

Generalization of the Non-uniform Entropy Formula

In the derivation of the non-uniform entropy formulation for Markov sources, the simplification came from considering a p-tuple $\mathbf{k}$, such that all m-tuples ending in $\mathbf{k}$ had the same conditional probabilities for their next output. The derivation actually made no use of the fact that $\mathbf{k}$ was contiguous, and at the end. Let us try to generalize those results.

Let $\gamma \notin S$ be allowed to match any source output. Engineers call such a variable a *don't care* condition. Let φ be an m-tuple over $S \cup \{\gamma\}$, which we will call a *pattern.* Define containment as follows: If $s_{\mathbf{j}}$ is an m-tuple over S, then $\varphi \subseteq s_{\mathbf{j}}$ if for each $\varphi_i \neq \gamma$, $\varphi_i = s_i$. Suppose that a set of patterns Φ has been chosen with the properties:

- The elements of Φ form a partition of S. That is, for every m-tuple $s_{\mathbf{j}}$, there is a unique $\varphi \in \Phi$, such that $\varphi \subseteq s_{\mathbf{j}}$ and φ has a minimal number of elements equal to γ.

- For all $\varphi \in \Phi$, if m-tuples $s_{\mathbf{j}}$ and $s_{\mathbf{k}}$ are both in the equivalence class defined by φ through the above partition,

  $$p_{i|\mathbf{j}} = p_{i|\mathbf{k}}, \text{ for all } i.$$

Then we can restate the entropy formula as

$$H(S) = \sum_{\varphi \in \Phi} p_{\varphi} \; H(S|\varphi),$$

where $p_{\varphi} = \sum\{p_j | s_j$ in the equivalence class defined by $\varphi\}$.

Tree Formula

Suppose that S is a source of tree nodes with a well defined traversal order. Let a pattern be a partial subtree, possibly with *don't care* elements, denoted *. The pattern has a node, denoted @, which is the last node of the pattern in traversal order. We will be interested in the probability distribution of nodenames that occur in this @ position, given that the rest of the pattern is matched. The location of the assign node in Figure 3.2 might correspond to the pattern

```
 ifstmt
   _|_
  /   \
relN   @
```

meaning the second son of an ifstmt whose first son is a relN test. Asterisks for the third son of ifstmt and the sons of relN are omitted. Suppose that we can find a set Π of patterns, such that for each $\pi \in \Pi$, the conditional probabilities of the values of @ are independent of any previous context other than that specified in π. Suppose further that for each node in a possible tree, there is some unique $\pi \in \Pi$, such that the node is in the @ position of π. If the patterns are allowed to overlap, we will require an effective algorithm for determining which pattern to associate with a given tree position. By reasoning similar to that of the previous section,

$$H(S) = \sum_{\pi \in \Pi} p_{\pi} \; H(S|\pi).$$

In order to estimate H, we need to find a traversal order, a suitable pattern set Π, and we need to estimate the probabilities p_{π}, and $p_{i|\pi}$. The comments above about overlapping patterns above might lead one to the conclusion that such situations are to be avoided; on the contrary, the pattern refinement procedure discussed below generates large numbers of nondisjoint patterns.

3.7. Pattern Refinement

In Section 4.4 we will describe a process called *pattern refinement,* in which the nodes matching a pattern π are partitioned into two sets: those also matching a new larger pattern ν, and the remaining ones. This section contains a proof that pattern refinement always leads to an improvement in the estimate unless ν has the same conditional probabilities as π. It is not even necessary for the refining pattern ν to have a lower entropy for its matching nodes than the original. In order to prove the theorem, a few elementary results from probability theory and information theory will be needed.

Lemma 3.1. *Let A, B, and C be events. Then*

$$P\{A \text{ and } B|C\} = P\{A|B \text{ and } C\}\cdot P\{B|C\}.$$

Proof: The result follows by two applications of the definition

$$P\{X|Y\} = P\{X \text{ and } Y\} \ / \ P\{Y\},$$

with $Y = B$ and C, then $Y = C$. See [Feller68], page 116. ∎

Lemma 3.2. *Let* $\{p_1,p_2,\ldots p_n\}$ *and* $\{q_1,q_2,\ldots q_n\}$ *be sets of probabilities. That is*

$p_i \geq 0$ *and* $q_i \geq 0$, *for all i, and*

$$\sum_i p_i = 1, \quad \sum_i q_i = 1.$$

Then the formula

$$-\sum_i p_i \log p_i \leq -\sum_i p_i \log q_i$$

holds with equality if and only if $p_i = q_i$, *for all i.*

Proof: This is a well known result, following from the inequality $\ln x \leq x-1$, with equality if and only if $x = 1$. See, for example, [Abramson63], formula 2-8b. ∎

Theorem 3.2. *Let π be a pattern, and ν a pattern containing π. Let π' be the pattern defined by "matching π, but not matching ν". Then the component of the entropy estimate contributed by the two disjoint patterns ν and π' does not exceed the component of the pattern π that they replace. In other words,*

$$p_\pi \ H(S|\pi) \geq p_\nu \ H(S|\nu) + p_{\pi'} \ H(S|\pi').$$

Furthermore, equality holds if and only if $P\{s_i|\nu\} = P\{s_i|\pi\}$ *for all i.*

Proof: The general approach of the proof is to divide the probabilities of various nodes into a ν component and a π' component. There will be some nodes for which one of the components will be zero. We will adopt the convention that 0 log 0 = 0 in order to simplify our notation.

Let $\alpha = P\{\nu|\pi\}$, $\beta = P\{\pi'|\pi\}$. Clearly $\alpha+\beta = 1$.

For each node $s_i \in S$, let $p_i = P\{s_i|\pi\}$. Decompose p_i as follows:

Let $p_i = q_i + r_i$, where $q_i = P\{s_i \text{ and } \nu|\pi\}$,

$r_i = P\{s_i \text{ and } \pi'|\pi\}$.

Clearly $\sum_i q_i = \alpha$, and $\sum_i r_i = \beta$. Also, by Lemma 3.1 and the fact that matching ν or π' implies matching π,

$$P\{s_i|\nu\} = \frac{q_i}{\alpha},$$

$$P\{s_i|\pi'\} = \frac{r_i}{\beta}.$$

Consider now the contribution of π to the entropy estimate.

$$\begin{aligned}
p_\pi\, H(S|\pi) &= -p_\pi \sum_i p_i \log p_i \\
&= -p_\pi \sum_i q_i \log p_i - p_\pi \sum_i r_i \log p_i \\
&= -\alpha p_\pi \sum_i \frac{q_i}{\alpha} \log p_i - \beta p_\pi \sum_i \frac{r_i}{\beta} \log p_i \\
&= -p_\nu \sum_i \frac{q_i}{\alpha} \log p_i - p_{\pi'} \sum_i \frac{r_i}{\beta} \log p_i \\
&\geq -p_\nu \sum_i \frac{q_i}{\alpha} \log \frac{q_i}{\alpha} - p_{\pi'} \sum_i \frac{r_i}{\beta} \log \frac{r_i}{\beta} \\
&= p_\nu\, H(S|\nu) + p_{\pi'}\, H(S|\pi').
\end{aligned}$$

From Lemma 3.2, equality holds if and only if $\frac{q_i}{\alpha} = \frac{r_i}{\beta} = p_i$ for all i. Since $\beta = 1-\alpha$ and $q_i+r_i = p_i$, $\frac{q_i}{\alpha} = p_i$ implies $\frac{r_i}{\beta} = p_i$. ▮

Although Theorem 3.2 is stated in terms of tree patterns, the concept is equally valid for any non-uniform entropy formula where a partition of the possible source output configurations is refined by splitting one of the equivalence classes into two disjoint pieces.

This chapter has contained most of the formal mathematics of the thesis; Chapter 5, on error analysis, also contains a few results.

4. Applying the Entropy Model to the Sample

4.1. Sample Description

Obtaining a Representative Sample

When it was decided to apply the framework of information theory to the empirical study of programs, the MESA language and its compiler were still under development, so there were very few source programs. Thus, the analysis programs were being developed as the language was maturing and as a collection of source programs were being written. Writing software that depends upon the internal workings of a compiler under development by others is a frustrating task, but eventually enough programs existed to provide a suitable sample for analysis. At this point the compiler, all support routines, and the entire set of available source programs were saved away on tape and this frozen universe was used for analysis and for refinement of the analysis tools. The sample lasted for almost 5 months, and for several iterations of the analysis programs. These were the Pass 3 trees described in Chapter 2. When Pass 4 trees were chosen for study, the changing language had stabilized, so a new larger sample was made from essentially all existing source programs.

The compiler and runtime system are all written in MESA itself. Since it is a young language, these comprise the majority of the programs written. The language is intended for system implementation, so these programs are probably representative of the ones which will be written in the future. The sample was 114 programs, of which approximately half were the multipass compiler. The remainder were runtime support and utility software.

Sample Size

The physical characteristics of the set of sample programs are enumerated below. Since length of identifiers varies with programmers, a count of the tokens as returned by the lexical scanner of the compiler was also taken as a measure of source program size. Tokens included identifiers, operators, numbers, reserved words, etc. The count of lines was made as an afterthought and in some cases represents a count of lines in a later version of a given program.

Number of Programs: 114.
Lines of Source: approximately 37,000.
Characters of Source (excluding comments): 992,030.
Tokens of Source: 199,406.

The programs were written by 5 programmers, all of considerable experience. The size of the programs varied from small collections of shared definitions to large compiler modules. The smallest was 165 characters (28 tokens), and the largest was 31,696 characters (7119 tokens). The mean number of characters per program was 8090, with a standard deviation of 7150, indicating a large variability in program size. The following information is irrelevant to our further discussion, but is included for interest.

Average length of identifier (excluding reserved): 8.12 chars
Average proportion of comments: 7.9%

The programs are low on comments, but the programmers compensate by choosing long descriptive names for variables. The mean of 8.12 characters is quite large when one considers the relatively frequent use of short variables such as *i* and *j*. Some definitions modules had an average identifier length in excess of 13. These modules also had a higher percentage of comments, sometimes in the range of 30 to 40%.

Existing Encodings of the Sample

Our goal is to see how little code is necessary to represent a program. It is useful to have a benchmark against which to measure our ability. Since code compression comes often at the cost of increased complexity in the encoding and decoding processes, knowing the cost of a naive encoding would be helpful. Certainly the output of the existing compiler can serve as an upper bound on the amount of space needed to encode a program, but this does not reflect a naive encoding since an effort was made to have the compiler produce compact object code. The other representation of the programs that we have is the trees described in Chapter 2. The actual representation of these trees used by the analysis routines was designed for ease of incremental updating of various data bases, so the size of the set of trees is not the correct measure. We will see in the next section that a naive, but compact encoding of these trees would require at least 10 bits per node.

Code generated by the compiler: 944,512 bits.

Tree representation: 296,895 nodes·10 bits per node = 2,968,950 bits.

4.2. Applying Uniform Order Markov Model

Zero-memory Source Model

The simplest model for encoding the trees is to assume that the various tree nodes are independent. Each node i is assumed to occur with probability p_i. From the definitions of Chapter 2, we can calculate the entropy $H_0(S)$ by the formula

$$H_0(S) = -\sum_i p_i \log p_i.$$

This model provides some insight into the use of the language. Of the 296,895 nodes in the sample, there were 1037 distinct nodenames. While this number seems large, recall that it also includes programmer-specified numbers and strings. There were 556 nodes which occurred only once, 399 of them string literals. We will consider the effects of literals in the next section. The entropy estimate was 4.75 bits per node, quite a bit smaller than the slightly over 10 bits needed to represent 1037 symbols of equal frequency.

Figure 4.1 shows the 35 most frequent nodes together with their contribution to the entropy calculation. The most frequent nodes are those that were invented to put the information about variables into the tree. No meaningful conclusions can be drawn about use of numerical literals, since most of the 0's, 1's, and 16's are sons of var nodes, the node which was invented to put information from the symbol table back into the trees for analysis. We can see that the static usage of variables by frame location is ordered: local, global, field, and entry. The most popular statement nodes are call, assign, ifstmt, and return. Some of the list nodes refer to compound statements. The most popular expression nodes are: var, call, dot (field selection via pointer), dollar (field selection from variable), plus, relE (test for equality), and uparrow (obtaining value of cell from pointer to the cell). The node call can be either a statement or an expression; we cannot tell from these zero-memory statistics which is which. The first 35 nodes (3% of the total) accounted for 93% of the node usage and 84% of the total entropy estimate.

nodename	*count*	p	$-p \log p$	$\Sigma -p \log p$
0	54280	.183	.448	.448
var	40289	.136	.391	.839
16	26481	.089	.311	1.150
local	17580	.059	.241	1.392
empty	13917	.047	.207	1.599
global	12279	.041	.190	1.789
1	11293	.038	.179	1.968
num	10220	.034	.167	2.135
2	9230	.031	.156	2.291
field	7744	.026	.137	2.428
list	7425	.025	.133	2.561
14	7054	.024	.128	2.690
call	6616	.022	.122	2.812
assign	5826	.020	.111	2.923
dot	5726	.019	.110	3.033
3	5428	.018	.106	3.139
plus	4039	.014	.084	3.223
4	3309	.011	.072	3.295
dollar	2770	.009	.063	3.358
entry	2686	.009	.061	3.420
relE	2348	.008	.055	3.475
8	2245	.008	.053	3.528
5	2189	.007	.052	3.580
ifstmt	1810	.006	.045	3.625
return	1683	.006	.042	3.667
body	1513	.005	.039	3.706
6	1385	.005	.036	3.742
32	1341	.005	.035	3.778
uparrow	1270	.004	.034	3.811
7	1264	.004	.034	3.845
item	1214	.004	.032	3.877
9	1178	.004	.032	3.909
15	929	.003	.026	3.935
11	928	.003	.026	3.961
10	890	.003	.025	3.986

Figure 4.1. Excerpt from the zero memory entropy calculation.

Dealing with Programmer-defined Literals

In the Markov studies performed for this thesis, certain simplifying assumptions were made about literal constants. It was assumed that the numbers and string constants seen in the sample defined the range of all possible values for numbers and string constants. For numbers this was not too bad an assumption. A vast majority of the numbers fell into a small range, making the entropy estimate for those outside the range unimportant to the total estimate for numbers. With strings, however, this was not the case. Of the 449 literal strings in the sample, the most frequent occurred 4 times, and only 30 occurred more than once. In Chapter 5, on error analysis, a more conservative estimate of entropy for string constants is factored back into the final estimates in the more general entropy model (they increase the total estimate by 4%).

Uniform Markov Estimates

In Chapter 3, we saw that the standard definition of "previous" for Markov sources was not the proper definition to use for trees; the notion should be defined in terms of the tree structure. Figure 4.2 shows the results of applying a Markov source approximation of order m to the sample for various values of m, and for various definitions of "previous." The formula used to compute the entropy was

$$H(S) = -\sum_{\mathbf{k}j} p_{\mathbf{k}j} \log p_{j|\mathbf{k}} ,$$

where $\mathbf{k}$ denotes an m-tuple over the source alphabet. In order to obtain a number, one must approximate $p_{\mathbf{k}j}$ for all $(m+1)$-tuples which occur in the sample. For those nodes with a low frequency of occurrence, the relative error in the approximation is likely to be larger. In Figure 4.2, f is the frequency of the $(m+1)$-tuple in the sample. The last two columns show the percentage of $(m+1)$-tuples which occurred fewer than 6 times, and their contribution to the total entropy estimate. In the table, *brother* refers to the immediately preceding brother in the tree. For a node that is a first son, the value of its *brother* is null.

order m	definition of previous	H estimate	#of distinct $(m+1)$-tuples	% tuples with $f < 6$	%H from $f < 6$
0	--	4.752	1037	72	1.4
1	*father*	3.180	1945	60	2.3
1	*brother*	3.158	2652	61	3.4
2	*grandfather, father*	2.802	6507	66	6.4
2	*father, brother*	2.053	4002	62	4.9
3	*grandfather, father, brother*	1.662	11954	61	12.2

Figure 4.2. Results of uniform Markov estimates on trees.

The results for the zero memory approximation have also been included in Figure 4.2 for comparison. Another interesting number is the average number of bits of object code per node as generated by the existing compiler. There were 296,895 nodes in the trees and 944,512 bits of code generated, yielding an average of 3.18 bits per node. This shows that the Markov estimates and the current compiler are of similar magnitude. It also reflects the considerable thought that went into the design of the current object code in order to keep size to a minimum.

As the order of the Markov source approximation increases, the entropy estimate decreases. At the same time, however, the amount of the estimate which comes from low frequency terms increases. We also see that the choice of *previous* as well as the order affects the results. Figure 4.3 shows the 35 most frequent 3-tuples in the second order estimate with *father* and *brother* as previous nodes.

Several things can be learned by examining Figure 4.3. The most frequent nodes are not necessarily the ones which contribute the most to the entropy estimate. If the conditional probability $p_{j|\mathbf{k}}$ is close to unity, log $p_{j|\mathbf{k}}$ is small, offsetting the large $p_{\mathbf{k}j}$ value. If the 3-tuples were ordered by contribution to H instead of frequency, the total for the top 35 3-tuples would be .941 instead of .723. The 3-tuples (dot, var, var) and (dot, plus, var) each contribute zero to the entropy. This is true not because of the value of the *brother* (var and plus, respectively), but because dot has 2 sons, the second of which is always var. In 40% of the cases shown, the *brother* is null, telling us only that the node j is a first son. The other two entries with an entropy contribution of 0 result from the "boundary conditions" on the traversal: all trees begin body[list[....

father	*brother*	*j*	*count*	$p=p_{fbj}$	$p'=p_{j\mid fb}$	$-p \log p'$	$\Sigma -p \log p'$
var	0	16	26020	.088	.503	.087	.087
var	-	local	17580	.059	.436	.071	.158
var	-	global	12279	.041	.305	.071	.229
var	0	0	11970	.040	.231	.085	.314
var	-	field	7744	.026	.192	.062	.376
var	local	0	7230	.024	.411	.031	.407
var	1	0	6542	.022	.957	.001	.409
call	-	var	6370	.021	.963	.001	.410
var	0	14	6190	.021	.120	.064	.474
assign	-	var	4011	.014	.688	.007	.481
var	2	0	3959	.013	.890	.002	.483
var	local	1	3726	.013	.212	.028	.511
list	-	2	3599	.012	.485	.013	.524
plus	-	var	3396	.011	.841	.003	.527
var	3	0	3159	.011	.925	.001	.528
var	global	0	3033	.010	.247	.021	.548
dot	-	var	2965	.010	.518	.009	.558
dot	var	var	2965	.010	1.000	0.000	.558
plus	var	var	2720	.009	.801	.003	.561
var	-	entry	2686	.009	.067	.035	.596
var	field	0	2651	.009	.342	.014	.610
call	var	empty	2469	.008	.304	.014	.624
dot	-	plus	2352	.008	.411	.010	.634
dot	plus	var	2352	.008	1.000	0.000	.634
var	local	2	2260	.008	.129	.023	.657
num	-	0	2254	.008	.221	.017	.674
list	assign	assign	2179	.007	.506	.007	.681
var	4	0	2080	.007	.969	.000	.681
call	var	list	1786	.006	.220	.013	.694
num	-	1	1786	.006	.175	.015	.709
call	list	empty	1769	.006	.974	.000	.710
call	var	var	1620	.005	.200	.013	.722
-	-	body	1513	.005	1.000	0.000	.722
body	-	list	1513	.005	1.000	0.000	.722
var	5	0	1510	.005	.961	.000	.723

Figure 4.3. Excerpt from Markov source estimate of order 2.

There are definitely dependencies of the nodes upon the values of the neighbors within the tree. The high percentage of entropy coming from *m*-tuples of low frequency in the models above suggests that simple uniform attempts to capture the dependencies are not adequate. Below we will apply the non-uniform formulation for entropy discussed in Chapter 3.

4.3. Estimating the Entropy by the Non-uniform Entropy Formula

General Methodology

We wish to choose a set of patterns so that every node in a tree can be associated with a unique pattern, depending on nodes which appeared earlier in a traversal of the tree. For our investigations, we shall always be using preorder traversal. One can think of the patterns as defining a set of states of an encoding process. The ultimate goal is to have a set of patterns such that all previous context that significantly affects the value of a node is contained in the pattern associated with that node. If our current set of patterns does not have that property, our model is one which assumes a smaller local order than the actual local order of the source. This corresponds to an overestimate of the true entropy, but for practical purposes, we will not have a sufficient sample set to find all dependencies accurately. We will, however, have a bound for each pattern π and we can use it to decide whether it is useful to pursue larger patterns containing π.

Description of the Patterns Used

Recall from Chapter 3 that a patterns specifies a context reached by a partial traversal of the tree. The pattern has a distinguished node, denoted @, which is the next node to be visited in the traversal. As long as each point in the tree is associated with a unique pattern, we need not be limited to simple equal matches in the definition of our patterns. For example, one could have the pattern

```
{assign|assignx}
         |
        / \
       * var
          |
         / \
     local  @
```

which selects the address of local variables appearing as the right side of either assignment statements or assignment expressions. The patterns need not be disjoint. If π and ν are both patterns, with $\pi \subseteq \nu$, then there are some places within trees that match both patterns. In fact, all places matching ν also match π. We do not wish to associate a given node with more than one pattern, so we will think of the pattern π as matching all situations which do not also match ν. Since a list node can have an arbitrary number of sons, we need some way to deal with this freedom in our choice of patterns. This is accomplished by allowing a node in the pattern to be preceded by an arbitrary number of brothers. Thus the pattern

```
 list
   |
 / | \
* ... @
```

selects a node which is a son of list, but not the first son.

Notation

We will be talking about many patterns in the remainder of the chapter. While the tree diagrams are helpful, they are not very compact. We will adopt a "function-like" notation to specify patterns, which is clearly equivalent to the trees. In this notation, the above patterns would be written:

```
{assign|assignx}[*,var[local,@]]
list[*,...@].
```

Initial Pattern Set

In order to "grow" a set of patterns, we must chose a starting point. The most conservative approach would begin with a zero-memory model, i.e., the single pattern "`@`". But nodes seem to depend upon at least their father, so we shall start with a first order model. Such a set of patterns can be generated quickly and easily by a modification to the programs which produced the uniform Markov estimates.

The starting pattern set specifies for each node in the tree its *father* and the *son position* which the node occupies. For example, the node <u>`ifstmt`</u> has 3 sons, so the initial pattern set contains the three patterns:

`ifstmt[@]`, `ifstmt[*,@]`, and `ifstmt[*,*,@]`.

The variable length <u>`list`</u> node requires special handling. The first son is a count of the remaining sons, and should be treated differently from the later sons. In the initial pattern set, all non-initial sons of <u>`list`</u> are considered to be in the same pattern. Hence, there are the two patterns:

`list[@]`, and `list[*,...@]`.

Entropy Estimate from the Initial Pattern Set

For each pattern π, there is the set of nodes in the sample associated with π. These nodes, considered as the output of a zero memory source, have an entropy $H(S|\pi)$. If the probability that a node is associated with pattern π is p_π, then the pattern has a contribution to the total entropy estimate of $p_\pi H(S|\pi)$. Experimentally, we estimate p_π and $H(S|\pi)$ by assuming the probabilities of patterns and nodes to be their relative frequency in the sample.

```
(2)    var[*,@] (address of variables)
f = 40289 H =  3.680, p =  .136, H contr. =   .499, cumul. H =   .499
 154 cases (11 shown)
 0         12914   .32      4          2076   .05      9            833   .02
 1          6658   .17      8          1532   .04      7            825   .02
 2          4014   .10      5          1517   .04      10           590   .01
 3          3131   .08      6           946   .02      <<others>>  5253   .13

(5)    list[*,...@] (elements of lists: statements, expressions, etc.)
f = 22481 H =  3.763, p =  .076, H contr. =   .285, cumul. H =   .784
  72 cases (11 shown)
 assign     5085   .23      ifstmt     1528   .07      casetest     572   .03
 call       3573   .16      return     1409   .06      casestmt     509   .02
 num        2623   .12      item       1068   .05      dostmt       448   .02
 var        2420   .11      dot         582   .03      <<others>>  2664   .12

(4)    var[*,*,*,@] (length of variables)
f = 40289 H =  2.099, p =  .136, H contr. =   .285, cumul. H =  1.069
  67 cases (11 shown)
 16        26020   .65      15          611   .02      11           357   .01
 14         6502   .16      8           451   .01      4            347   .01
 1          1509   .04      2           414   .01      3            334   .01
 32         1168   .03      64          381   .01      <<others>>  2195   .01

(1)    var[@] (frame of variables)
f = 40289 H =  1.762, p =  .136, H contr. =   .239, cumul. H =  1.308
  4 cases
 local     17580   .44      field      7744   .19
 global    12279   .30      entry      2686   .07

(6)    num[@] (programmer specified numbers)
f = 10220 H =  5.132, p =  .034, H contr. =   .177, cumul. H =  1.485
 408 cases (11 shown)
 0          2254   .22      16383       397   .04      16           196   .02
 1          1786   .17      -1          345   .03      13           161   .02
 2           748   .07      65535       242   .02      5            143   .01
 3           537   .05      4           221   .02      <<others>>  3190   .31

(12)    assign[*,@] (right hand side of assignment statements)
f =  5826 H =  3.591, p =  .020, H contr. =   .070, cumul. H =  1.556
  42 cases (11 shown)
 call       1288   .22      plus        377   .06      assignx      153
.03
 num        1078   .19      <empty>     314   .05      minus        134
.02
 var         775   .13      dollar      261   .04      dindex       113
.02
 dot         580   .10      addr        208   .04      <<others>>   545
.09

(9)    call[*,@] (procedure call actual parameters)
f =  6616 H =  2.961, p =  .022, H contr. =   .066, cumul. H =  1.622
  27 cases (11 shown)
 list       1816   .27      dot         571   .09      addr         129   .02
 var        1739   .26      dollar      209   .03      plus          68   .01
 <empty>     760   .11      str         204   .03      dindex        56   .01
 num         686   .10      call        189   .03      <<others>>   189   .03

(7)    list[@] (length of lists)
f =  7425 H =  2.476, p =  .025, H contr. =   .062, cumul. H =  1.683
  38 cases (11 shown)
 2          3599   .48      5           322   .04      10            61   .01
 3          1139   .15      6           208   .03      0             60   .01
 1          1038   .14      7           113   .02      9             54   .01
 4           591   .08      8            80   .01      <<others>>   160   .02

(3)    var[*,*,@] (bit offset of variables)
f = 40289 H =   .336, p =  .136, H contr. =   .046, cumul. H =  1.729
  16 cases (3 shown)
 0         38841   .96      3           283   .01
 2           432   .01      <<others>>  733   .02
```

Figure 4.4. Excerpt from initial pattern entropy calculation.

In the initial pattern set, there are 154 patterns. The most frequent one occurs 40,289 times, and the least frequent one 2 times. There are 12 patterns which occur fewer than 25 times. The estimated entropy per node is 2.058 bits. This number is very close to the uniform Markov model of order 2, with *father* and *brother* as the previous nodes. This is not surprising, since we saw above that the *brother* in that model was often simply specifying the son position. Figure 4.4 contains a portion of the output from the entropy calculation, showing patterns, their contribution to the total, and the distribution of nodes associated with those patterns. For purposes of this figure, only the frequently occurring nodes are shown for a pattern. See Appendix C for a more complete listing.

A few words of explanation are needed for Figure 4.4. The number in parentheses is the *pattern number*. For the initial pattern set, they are assigned in order of decreasing frequency. As patterns are added to the set, they are given numbers sequentially. The description in parentheses tells what source constructs give rise to trees matching the pattern. The *H* figure given is the entropy estimate for the nodes associated with the pattern. The patterns are sorted by *H contr*, the product of *p* and *H*. The *cumul. H* value is the running total of *H contr.* for the patterns. The nodes matching the pattern are shown in three columns, giving the nodename, its frequency, and its relative frequency.

In the uniform Markov source models, it was difficult to relate the dependencies to the actual statements which appear in programs. The patterns show the correspondence in a more natural way. One should remember, however, that these are *static* counts, made on the basis of occurrence in programs, not *dynamic* counts taken of nodes as they occur during execution of a program.

From pattern 12, we can see that 32% of the right-hand-sides of assignments are simple, with the value assigned either a simple variable or a number. Procedure calls account for 22% of the right-hand-sides, and field selection (<u>dot</u> and <u>dollar</u>) account for 14%. The 5% figure for <<u>empty</u>> occurs in extract statements, e.g., `[v1,v2] ← f[]`. The compiler parses the left side of such statements into a <u>list</u> of empty assignments. The count for simple variables also includes subscripted variables with constant indices that can be "folded" into a simple variable.

In pattern 9, we see that only 27% of all procedure calls take more than 1 argument, and that 11% of the procedure calls are parameterless. Of the procedure calls with only one argument, 60% take a simple variable or a number.

Pattern 7 matches the length specification for list nodes. Short lists predominate. Even without other encoding, one could save space by defining new nodes list2, list3, etc. to handle the common cases and using just about any reasonable scheme for longer lists.

Pattern 3 matches the bit offset of variables. It is overwhelmingly zero. We will later see that only a small amount of further context suffices to separate all the nonzero instances.

Structuring the Sample to Facilitate Matching

A data structure was developed for the analysis programs that greatly simplifies the pattern match procedure, the refinement procedure, and the transformation procedures described below. With the first pattern matching facilities used, a problem developed; as the size of existing patterns grew, the computation required for finding all instances of a set of refinement patterns took 30 minutes to an hour of CPU time; and it was clear that some of the extensions of the analysis software could not be run at all. The solution to these problems is quite simple: in the sample trees, we store a value associated with each node that identifies which pattern that node matches. Since the analysis programs run on a 36 bit machine, there was enough unused room in a node to store a pattern sequence number without increasing the storage occupied by the sample. This is essentially the data processing technique of inverting a data base. Figure 4.5 shows the ifstmt tree from Figure 3.2, with the number of the pattern in the initial pattern set that each node matches.

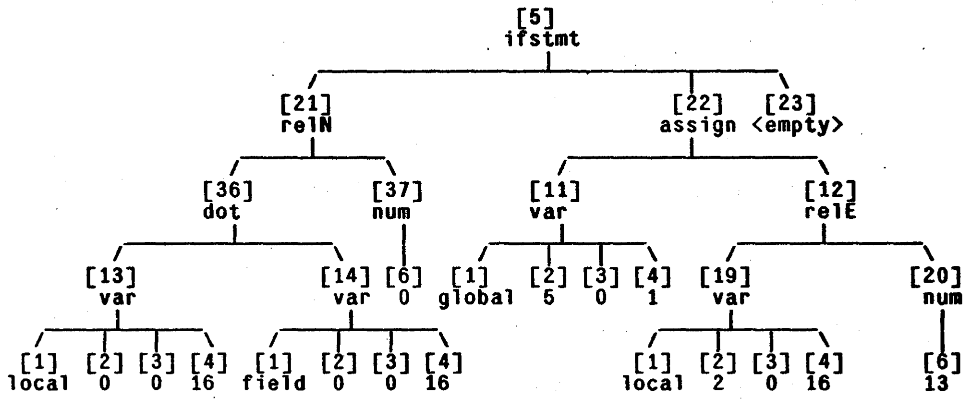


Figure 4.5. Parse tree showing pattern sequence numbers.

For the numbers to be useful, any procedure that places a node in a new pattern must update the pattern sequence field in the tree. This consistency maintenance is far simpler than general pattern matching. By writing all such changes to the sample onto a file, it is also possible to "undo" the effects of a new pattern.

4.4. Improving the Pattern Set

Finding Larger Patterns

The output from the entropy calculation gives the contribution of each pattern to the total entropy estimate. We can lower the estimate by finding, for a given pattern, situations where a larger context produces a distribution of nodes with a smaller entropy. For example, we see from pattern 7, Figure 4.4, that 48% of all lists are of length 2, and the entropy for the distribution of lengths is 2.476 bits. If however, we single out those lists which are parameter lists of call statements (approximately a quarter of all lists), 77% are of length 2, and the distribution of lengths has an entropy of only 1.085 bits.

Several schemes were used for finding such larger patterns. A more detailed description of the program finally used is given in Appendix B. Below is a brief listing of the features of the pattern match facility.

1. The entire collection of trees is traversed in preorder. Several auxiliary data structures keep a record of ancestors and siblings.

2. For an existing pattern, further restrictions may be made by specifying a list of predecessors to have given nodenames or son displacements.

3. For a pattern, perhaps restricted as in 2, the user can specify a collection of historical information (nodenames or son displacements of predecessors) on which he wishes to obtain statistics; they are automatically obtained for the nodes in the "@" position of the pattern.

4. One may also specify a collection of pairs of the items selected in item 3 above. For example, grandfather and "@" node.

5. As each node is encountered which matches the (possibly restricted) pattern, all requested information is stored in a hash table.

6. The output of the match gives the distribution of nodes or values as requested in item 3. For a pair of items (α,β), the distribution of items β is given for each value of α.

For patterns only one node larger than existing ones, the pair statistics from item 6 point out potential entropy reducers. For more complicated patterns, one can restrict an existing pattern by item 2 and iterate the match process. Other features of the match procedure allow several patterns to be investigated at once, allow the same statistics to be taken on a set of existing patterns, and allow several independent sets of statistics to be taken on a given pattern during a single traversal of the sample set.

Pattern Refinement

We will use the term *pattern refinement* to denote the process of "growing" patterns. A pattern π with a large entropy contribution is explored using the techniques of the previous section. A set of larger patterns ν_1, $\nu_2,\ldots$, ν_k, containing π, are discovered which have node distributions of lower entropy than that of π. The refinement procedure removes the nodes matching ν_i from those of π. This leaves a new pattern π', defined as those nodes matching π, but not ν_i. We know by Theorem 3.2 that this will lead to a lower entropy estimate.

A program related to the matching routines of the previous section allows incremental updating of the entropy estimate. The entire set of trees is traversed. When a node matching π is encountered, a test is made to see if it also matches ν_i. If so, the node is marked as matching ν_i, and the triple composed of π, ν_i, and the nodename is added to a hash table. In later sections of the thesis, we will refer to this triple, together with a count of occurrences, as a *transaction*. Another procedure processes the transactions, making changes to the node list of all relevant patterns. Figure 4.6 shows the transactions produced by the matching procedure and sample output from the update procedure when pattern 7, the length of lists, is refined.

```
TRANSACTIONS:
          old        new
        pattern    pattern  nodename   count
           7         218        2        80
           7         219        2      1390
           7         219        3       263
           7         219        4       121
           7         219        5        42

BEFORE:
(7)     list[@]
 f =  7425 H =  2.476, p =  .025, H contr. =   .062
  38 cases (11 shown)
 2          3599  .48      5             322  .04     10            61  .01
 3          1139  .15      6             208  .03     0             60  .01
 1          1038  .14      7             113  .02     9             54  .01
 4           591  .08      8              80  .01     <<others>>   160  .02

AFTER:
(7)     list[@]
 f =  5529 H =  2.786, p =  .010, H contr. =   .052
SEE ALSO:       f      H      p contr
 (218)       80 0.000   .000 0.000, arraydesc[list[@]]
 (219)     1816 1.085   .006  .007, call[*,list[@]]
  38 cases (11 shown)
 2          2129  .39      5             280  .05     10            61  .01
 1          1038  .19      6             208  .04     0             60  .01
 3           876  .16      7             113  .02     9             54  .01
 4           470  .09      8              80  .01     <<others>>   160  .03

(218)   arraydesc[list[@]]
 f =    80 H =  0.000, p =  .000, H contr. =  0.000
 2            80  1.00

(219)   call[*,list[@]]
 f =  1816 H =  1.085, p =  .006, H contr. =   .007
    4 cases
 2          1390  .77      4             121  .07
 3           263  .14      5              42  .02

delta H =   -.003
```

Figure 4.6. Example pattern refinement.

The description of results shown in Figure 4.6 are perhaps a little cryptic. In the "after" case, pattern 7 has been refined to mean all those lists that do not also satisfy patterns 218 and 219. Pattern 218 is particularly interesting. It says that a list under an `arraydesc` node has two sons (with probability 1), and no information need be encoded about the length of these lists.

Results of Pattern Refinement

The procedure for lowering the entropy estimate by pattern refinement is quite easy to characterize.

- The pattern matcher is used to find possible refinements of patterns with large entropy contributions.
- The refinement procedure produces a list of transactions on the pattern set.
- The entropy update procedure produces a new estimate for the affected patterns.
- The process is repeated.

Of course, as the Sorcerer's Apprentice learned, there needs to be some means for terminating the loop. The ridiculous extreme would be to find a huge set of patterns where nearly every pattern determines a unique member of the sample set. This approaches an encoding of programs by assigning them sequence numbers, an encoding that would be undefined on a program never seen before. The "rule of thumb" actually used was to consider a pattern π only if the number of instances of π was large compared to the range of values for the nodes associated with π. In Chapter 5, we will see a better stopping criterion in which the variability of the sample is gauged in order to decide on the advisability of a particular refinement.

Such a process could be automated, but would require some care in choosing the predecessors of a pattern when looking for larger patterns. For the purposes of this thesis, the refinement was done with a "programmer in the loop." This has its advantages, since some patterns can be produced by knowledge of the compiler.

The initial set of patterns contained 154 patterns, with the most "costly" pattern having an entropy estimate contribution of .499 bits. After applying the refinement procedure repeatedly, there were 268 patterns, with a highest contribution of .095 bits. The entropy estimate went from 2.058 to 1.642. This not only yielded a slightly lower value than produced by the uniform Markov approximations, it also reflects a conservative policy of only choosing refinements when the error introduced seems small. Section 5.8 contains an analysis of the potential inaccuracy of the estimate. Appendix C contains the actual data from the final set of patterns, including those additional modifications described in the next section.

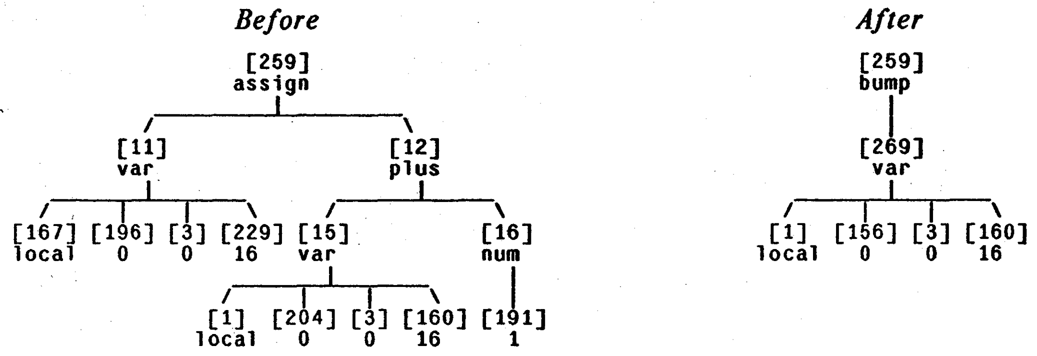
4.5. Program Transformations

Sometimes a problem becomes much simpler to solve when approached from a different direction. Mathematicians use this concept when they apply a "change of variables" to a problem. The same is true of program encoding. For example, when Clark and Green were studying the encoding of list structures in LISP [Clark77], they found little regularity in the absolute value of CDR cells. When, however, the pointers were made relative to the address of the list cell, it became apparent that a most of CDR's point nearby in memory, and that many point to the adjacent cell. This relativizing of pointers is what can be considered an *invertible transformation* of the program. Such a transformation leads to a lower entropy estimate, since the model of the source corresponding to this new representation better reflects the dependencies in the data.

With a slight modification to the entropy update procedure, we can provide machinery to allow rather general invertible transformations of the sample trees. Recall that a *transaction* is a triple of *old pattern, new pattern,* and *nodename*, together with a count. If we simply allow either the old pattern or new pattern field to be zero, we have the capability of adding or deleting nodes in the set of trees and incrementally updating the entropy estimate. A concrete example will help us to see what else is needed to facilitate transformations.

Transforming "bump" Trees

Every programmer knows that programs contain many statements of the form $\alpha \leftarrow \alpha+1$. In fact, quite a few computers have instructions for incrementing a memory location. It is not uncommon for a compiler's code generator to generate such an instruction when applicable. We might therefore wish to encode the statement more compactly than the standard tree representation of assign[α,plus[α,num[1]]. By suitable modifications to the matching procedure, we can find all such trees. Let us devise a scheme for transforming one into bump[α], and see what other tools are needed. One must be careful, of course, that α causes no side effects, such as a field pointed to by a procedure call. The "before" portion of Figure 4.7 shows the tree for such a statement. The numbers in square brackets are the patterns associated with the various nodes.



Glossary of pattern numbers

1. var[@]
3. var[*,*,@]
11. assign[@]
12. assign[*,@]
15. plus[@]
16. plus[*,@]
156. var[local,@]
160. var[local,*,*,@]
167. assign[var[@]]
191. plus[*,num[@]]
196. {assign|assignx}[var[local,@]]
204. plus[var[local,@]]
229. assign[var[local,*,*,@]]
259. item[*,list[*,...@]]
269. bump[@]

Figure 4.7. Example of a program transformation.

Ignore for the moment the pattern numbers given in brackets. In order to change the trees to reflect the new <u>bump</u> construct, we need only insert a single node, <u>bump</u>, in the tree, change the pointer that previously pointed to <u>assign</u>, and have the son of <u>bump</u> point to the old first son of <u>assign</u>. In addition to transforming the tree, we wish to update our pattern data base incrementally to reflect changes in the entropy estimate caused by such a transformation, and we wish to keep the pattern sequence numbers consistent with the patterns matched by the nodes. For the newly created bump node, the matching pattern is the same as that of the old assign node. The var node is assigned a new pattern number, that for son of bump. All the nodes of the subtree rooted by plus are discarded, so transactions are generated for them with a new pattern field of 0. The sons of the other var node are much more difficult to deal with because they match patterns involving the grandfather, which is assign in the original trees. Since we are taking away the assign node, we have to associate these sons with smaller patterns not involving the value of their grandfather. This is simplified by having each newly generated pattern remember which smaller pattern or patterns it refines. If we define the *depth* of a pattern to be the number of levels in its tree representation, we can describe a *pattern downgrading* procedure to deal with our problem.

- Begin the procedure with an allowable pattern depth of 1. Begin at the sons of the subtree to be downgraded.

- Until the pattern matching a node no longer exceeds the allowable depth, replace the pattern with the one from which it was refined. If there is more than one possibility, ask an "oracle" (in the current implementation, the programmer). Record a transaction of any pattern change for this node.

- Recursively apply this procedure to the sons of the node with the allowable depth increased by one.

In the sample set, there were 188 instances of the construct $\alpha \leftarrow \alpha+1$, of which 25 were assignment expressions. In most cases, α was a simple variable, but some involved field selection operations and pointer arithmetic. The results of the TENEX-MESA statistical study , Sec. 1.3, would predict a larger number of "bump" statements, but that study also counted the increment clauses of FOR statements. During the transformation phase, there were 1777_{10} transactions generated, which involved creating 2 new patterns, bump and bumpx, and adjusting the node distribution for 62 previously existing patterns. All of these changes reduced the entropy estimate by .0012 bits per node, although the total number of nodes fell from 296895 to 295093, giving an effective decrease of .011. Even so, such numbers indicate that "bump" operations do not really consume very much of the total program information when viewed statically.

5. Error Analysis

5.1. Need for Error Bounds

One of our goals is to find a lower bound on the size of an object program. The smaller the determined entropy, the more compact our encoding can be made. We saw in Chapter 4 with both the uniform and nonuniform Markov models, that as we increased the *order* of the source, the amount of history used, the number obtained for the entropy became smaller. It should be clear, however, that the possible error in the estimate increases with the order of the model. It does us little good to have an entropy estimate of 1.0 if this number is known only to within ± 10.

We can view a Markov source as a process that can be in one of a number of *states,* depending upon the context, i.e., the values of previous outputs. The entropy of the source is defined in terms of the probabilities of the various states, and the conditional probabilities for given states. When we estimate the entropy of a source, there are two sources of error: we may overlook some of the states of the process, or we may incorrectly estimate one of the probabilities. We know from Theorem 3.2 that in the former case, our estimate will be an overestimate. Since such errors are inevitable, we can loosely rationalize them as bounding the compression capability of an encoder with a given level of complexity. The other source of errors, incorrect probabilities, is more serious. Statisticians have shown that for zero-memory sources, estimating the probabilities by the relative frequencies produces an underestimate of the true entropy [Basharin59]. A Markov source can be viewed as a collection of zero-memory sources, one for each state [Cover76], so it seems likely that experimental data will produce an underestimate in the Markov case as well.

5.2. Notation

Let us try to understand how the estimation of probabilities from the experimental data might cause problems. To facilitate our discussion, we will define some notation. It will be instructive to talk first about a zero-memory source S. Let the possible source outputs be

$$s_1, s_2, \ldots, s_q$$

which occur with probabilities

$$p_1, p_2, \ldots, p_q$$

We don't know the values of the p_i, but we have empirical frequencies obtained from a "representative" sample. Denote them by

$$n_1, n_2, \ldots, n_q$$

where $n_1+n_2+\cdots+n_q = N$. We choose to estimate p_i by $\hat{p}_i = n_i/N$. Note that some of the n_i's could be 0, so we will continue our convention that 0 log 0 = 0 in the sums below, while in actual practice, we simply wouldn't have a term for that s_i.

5.3. Encoding Inefficiencies Induced by Improper Probability Estimates

Entropy is defined as the *average information content* of a source output. It is also a lower bound for the *average code length*. We can achieve this bound if can we give an output s_i a code of length $l(s_i) = -\log p_i$. While this is not always possible, assume for the moment that such a code can be found. We want to know how the average code length is affected when we devise a code, not using the p_i, but using our empirical $\hat{p}_i$. The true entropy of S is given by

$$H(S) = -\sum_{i=1}^{q} p_i \log p_i ,$$

we have approximated this by

$$\hat{H}(S) = -\sum_{i=1}^{q} \hat{p}_i \log \hat{p}_i .$$

The average code length, on the other hand, is defined by the sum

$$\textit{average code length} = -\sum_{i=1}^{q} p_i \log \hat{p}_i .$$

We know from Lemma 3.2 that the average code length will be larger than either H or

$\hat{H}$. The real difficulty occurs for those cases when $\hat{p}_i = 0$. This corresponds to possible outputs which did not occur at all in the sample. The quantity log 0 is undefined, so if we had assigned codes on the basis of $\hat{p}_i$, we would not be able to encode s_i at all.

5.4. Experimental Tests of Sample Variability

One means of determining the quality of our probability estimates $\hat{p}_i$ is to use them to devise an encoding and then try the encoding out with new data, obtaining an empirical average cost. We will refer to the original data used to calculate $\hat{p}_i$ as the *training sample*. If both the training sample and our new *test sample* are representative, then the average cost should be close to $\hat{H}$. For the purposes of estimating the error, it is not really necessary to produce the encoding, we can simply assume that we have devised a code with codeword lengths based on the probabilities.

In order to deal with the case of outputs not in the training sample, we will add to our encoding an "escape code". Whenever we see the escape code, we know that the next l bits denote the output in a less compact, but complete code. We will take a small quantity ε away from each of the probabilities $\hat{p}_i$ in order to create the escape code.

Let $\hat{\hat{p}}_i = (1-\varepsilon)\hat{p}_i$. Suppose that we can encode the test sample using a code with codeword lengths of $-\log \hat{\hat{p}}_i$, except for "new" outputs, which require a codeword of $-\log \varepsilon$, followed by l bits of a less compact code. Suppose that in the test sample, there are h "new" outputs, and that s_i occurs m_i times. For those s_i's whose frequencies contribute to h, let $m_i = 0$. Let us determine the total cost of encoding the test sample.

Let $m_1+m_2+\cdots+m_t+h = M$.

$$\begin{aligned} cost &= -\sum_i m_i \log \hat{\hat{p}}_i - h \log \varepsilon + hl \\ &= -\sum_i m_i \log (1-\varepsilon)\hat{p}_i - h \log \varepsilon + hl \\ &= -\sum_i m_i \log \hat{p}_i - (M-f) \log (1-\varepsilon) - h \log \varepsilon + hl \end{aligned}$$

As $\varepsilon \rightarrow 0$, the $-\log(1-\varepsilon)$ term approaches 0, but the $-\log \varepsilon$ term becomes large. Therefore, a careful choice of ε can reduce our encoding cost. The value of ε must be fixed before encoding a new sample, but we can make an *a posteriori* determination of the value of ε that would have minimized the cost. We can do this by computing the derivative

$$\frac{\partial\, cost}{\partial \varepsilon} = \frac{M-h}{1-\varepsilon} - \frac{h}{\varepsilon}$$

and setting the expression to 0. This yields a minimizing value of h/M, which can be given the unsurprising interpretation that the best choice of ε is the actual probability of a new output.

Although ε must be fixed, there is no reason that we cannot use several training samples in order to determine an empirical probability of encountering outputs not in previous training samples.

5.5 Applying Experimental Tests to the Markov Models

The problem of finding an optimal encoding scheme for a Markov source is considerably more difficult than finding one for a zero-memory source. Pasco [Pasco76] has a good discussion of the published literature in the field. Our principle concern, however, is to gauge the variability in our sample set. For this, it is not actually necessary to produce an encoding, but rather to use the empirical probabilities to hypothesize code lengths and see if the distribution of nodes in the new sample is well matched to these lengths.

A Markov Source Example

The techniques described above for zero-memory sources can be readily extended to Markov sources. To gain insight into the extension, it is instructive to look first at some actual experimental data for a simpler source. Consider again the Markov source defined by the state diagram in Figure 3.3. Assign probabilities to the transitions as follows:

state	*output*	*probability*
a	a	.3
a	b	.7
ab	a	.4
ab	b	.6
bb	a	.5
bb	b	.5

Figure 5.1. Conditional probabilities for the example.

If we assume that the source is stationary, we can calculate the steady state probabilities of the various states by a technique beyond the scope of this discussion (see [Ash65]). They are given in Figure 5.2.

state	*probability*
a	.3937
ab	.2756
bb	.3307

Figure 5.2. Steady state probabilities for the example.

Using the values from Figures 5.1 and 5.2, we can compute the entropy of the source to be .9452 bits per symbol. Figure 5.3 show results from a program that randomly generated a string of a's and b's based upon the probabilities from Figure 5.1. It is comforting to see that the empirical steady state probabilities are close to the theoretical ones.

state	*output*			*Sample*				*state*		
π	i	1	2	3	4	5	*total*	*total*	$\hat{p}_\pi$	$\hat{p}_{i\|\pi}$
a	a	13	9	19	3	10	54			.28
a	b	29	28	28	27	30	142	196	.39	.72
ab	a	14	11	13	9	13	60			.42
ab	b	15	18	15	18	18	84	144	.29	.58
bb	a	15	17	15	18	18	83			.52
bb	b	14	17	10	25	11	77	160	.32	.48

Figure 5.3. Randomly generated data.

A Computationally Simpler Formula for Entropy

The formulas for entropy are defined in terms of probabilities and their logarithms. Since we are using relative frequencies for approximations of the probabilities, perhaps there will be some cancellation of terms in the formula when we deal with the frequencies not as decimal numbers but as fractions.

$$\begin{aligned}\hat{H}(S) &= \hat{p}_a\hat{H}(S|a) + \hat{p}_{ab}\hat{H}(S|ab) + \hat{p}_{bb}\hat{H}(S|bb)\\ &= \frac{196}{500}\left(-\frac{54}{196}\log\frac{54}{196} - \frac{142}{196}\log\frac{142}{196}\right)\\ &\quad + \frac{144}{500}\left(-\frac{60}{144}\log\frac{60}{144} - \frac{84}{144}\log\frac{84}{144}\right)\\ &\quad + \frac{160}{500}\left(-\frac{83}{160}\log\frac{83}{160} - \frac{77}{160}\log\frac{77}{160}\right)\\ &= \frac{1}{500}\Big(196\log 196 + 142\log 142 + 160\log 160\\ &\qquad - 54\log 54 - 142\log 142 - 60\log 60\\ &\qquad - 84\log 84 - 83\log 83 - 77\log 77\Big)\\ &= .935 \text{ bits per symbol.}\end{aligned}$$

This example shows that the reduced formula is indeed much easier to compute, with only one division required at the end, rather than requiring all of the probabilities to be computed. It is easy to derive the corresponding formula for arbitrary sources; we will merely state the result. Let n_π denote the number of occurrences of pattern π in the sample. Let $n_{\pi i}$ denote the number of occurrences of the node s_i in a situation matching pattern π. If there are a total of N nodes in the sample,

$$\hat{H}(S) = \frac{1}{N}\Big(\sum_{\pi} n_\pi \log n_\pi - \sum_{\pi,i} n_{\pi i}\log n_{\pi i}\Big).$$

With the customary formula, it is necessary to know the frequency of occurrence of each pattern (n_π) in order to estimate the probabilities $p_{i|\pi}$. This implies that if we have only sequential access to the data, it is neccessary to make two passes over the data in order to calculate the entropy. The formula above does not have this restriction; we can use the frequency of each node in a pattern and accumulate pattern totals. One should ask, however, what roundoff error is introduced by the subtraction of the two potentially large sums. There does not seem to be a real problem; when the two formulas were used to calculate the entropy of the program tree sample (300,000 nodes), the two estimates differed by less than one part per million.

5.6. Computing Average Code Length

Suppose now that we have used a training sample to estimate the probabilities that characterize our source. We now wish to encode a new sample using code lengths determined by these probabilities. Let n^t_π and $n^t_{\pi i}$ denote the pattern and node frequencies for the training sample. Consider the formula for average code length, ignoring for the moment the possibility of nodes in the sample not present in the training sample:

$$\begin{aligned}
\textit{average code length} &= \frac{1}{N} \sum_{\pi,i} n_{\pi i} \left(-\log \hat{p}_{i|\pi} \right) \\
&= \frac{1}{N} \sum_{\pi,i} n_{\pi i} \left(-\log \left(n^t_{\pi i} / n^t_\pi \right) \right) \\
&= \frac{1}{N} \left(\sum_{\pi,i} n_{\pi i} \log n^t_\pi - \sum_{\pi,i} n_{\pi i} \log n^t_{\pi i} \right) \\
&= \frac{1}{N} \left(\sum_{\pi} n_{\pi} \log n^t_\pi - \sum_{\pi,i} n_{\pi i} \log n^t_{\pi i} \right)
\end{aligned}$$

One should note that if the training sample and the test sample are the same, the average code length formula reduces to the formula for $\hat{H}$, the estimated entropy. We can apply this formula to the data of Figure 5.3 using 80% of the data as a training sample and the remaining 20% as a test sample; Figure 5.4 show the relevant counts.

Example

state π	*output* i	$n_{\pi i}$	n_{π}	$n^t_{\pi i}$	n^t_{π}
a	a	10		44	
a	b	30	40	112	156
ab	a	13		47	
ab	b	18	31	66	113
bb	a	18		65	
bb	b	11	29	66	131

Figure 5.4. Node counts considering first 80% of data as a training sample.

We can readily calculate the average code length from the formula:

$$\begin{aligned} \textit{average code length} &= \frac{1}{100}\Big(40 \log 156 + 31 \log 113 + 29 \log 131 \\ &\quad - 10 \log 44 - 30 \log 112 - 13 \log 47 \\ &\quad - 18 \log 66 - 18 \log 65 - 11 \log 66\Big) \\ &= .921 \end{aligned}$$

If we repeat this process for each of the 5 samples, we get the average code lengths shown in Figure 5.5

missing sample	1	2	3	4	5	mean	standard deviation
length	.963	.918	1.037	.896	.921	.947	.056

Figure 5.5. Average code length for each sample in terms of the others.

Allowing for Outputs not in the Training Sample

As we have seen before, we must allow for nodes in the sample that occur in situations where they did not occur in the training sample. We do this by "stealing" a small probability ε_{π} from each of the probabilities $\hat{p}_{i|\pi}$. When computing the cost of the "surprise" nodes, we add a length of $-\log \varepsilon_{\pi}$ as an "escape code" plus a length of l_{π} for a naive encoding of the node. The average code length is a little more complicated:

$$average\ length = \frac{1}{N}\Big(\sum_{\substack{\pi,i\\ n^t_{\pi i}\neq 0}} n_{\pi i}\,(-\log((1-\varepsilon_\pi)\hat{p}_{i|\pi})) + \sum_{\substack{\pi,i\\ n^t_{\pi i}=0}} n_{\pi i}(-\log \varepsilon_\pi + l_\pi)\Big).$$

Let $g_\pi = \sum\{n_{\pi i} \mid n^t_{\pi i} \neq 0\}$,

$h_\pi = \sum\{n_{\pi i} \mid n^t_{\pi i} = 0\}$.

This allows us to restate the cost as the following visually complicated but computationally simple formula:

$$average\ length = \frac{1}{N}\Big(\sum_{\pi} g_\pi \log n^t_\pi - \sum_{\substack{\pi,i\\ n^t_{\pi i}\neq 0}} n_{\pi i}\log n^t_{\pi i} - \sum_{\pi} g_\pi \log(1-\varepsilon_\pi)$$
$$- \sum_{\pi} h_\pi \log \varepsilon_\pi + \sum_{\pi} h_\pi l_\pi\Big).$$

5.7. A Criterion for Deciding Whether to Use a Pattern Refinement

Adding a new pattern to the source model defines a zero-memory source: the nodes appearing in situations matching the pattern. We can use the techniques of the preceding section to estimate the variability of the sample set for the new pattern.

Suppose that our data contains q distinct nodes, each occurring n_i times for $1 \leq i \leq q$. Let $\sum_i n_i = N$. We can calculate an estimate for the entropy of this distribution using the formula:

$$\hat{H}(S) = \frac{1}{N}\Big(N \log N - \sum_i n_i \log n_i\Big)$$

When we compute the entropy of S from the finite sample, we make the assumption that n_i/N is a valid approximation to the probability that node s_i will appear in a similar situation in future programs. If, however, the "density" of s_i nodes is highly non-uniform across the sample, this is not a very good assumption. Let us now consider how to know if the assumption is worthwhile. Let the sample be divided into r roughly equal pieces, with total sizes of $N^{(k)}$, $1 \leq k \leq r$. Denote the count of s_i nodes in the kth by n^k_i. Using the formulas of Section 5.6, we can obtain r values for the average code length, using r-1 pieces of the large sample as the training sample as follows:

Total training sample size: $N - N^{(k)}$,
Training count for node s_i: $n_i - n_i^k$,
Total test sample size: $N^{(k)}$,
Test count for node s_i: n_i^k.

The only remaining quantity in the average code length formulas is ε, the "fudge factor" for nodes not in the training sample. For the purposes of deciding upon a refinement, we will simply use ε equal to the empirical probability of an unseen node (shown to be the best possible ε in Section 5.4).

It is instructive to look at the two patterns in Figure 5.6, taken from Appendix C. They each predict the *frame* of a variable (`local`, `global`, `field`, or `entry`, see Chapter 2). In the sample, only `local` and `global` variables occurred in these constructs. Pattern 180 predicts the frame of a variable on the right side of a relational operator. Pattern 185 predicts the frame of a variable that is a "subscript", either of an array or a string.

```
(180)  {relE|relG|relN|relL|relGE|relLE}[*,var[@]]
 f =   403 H =   .998, p =  .001, H contr. =   .001
  global        212   .53        local          191   .47

(185)  {index|dindex|seqindex}[*,var[@]]
 f =   289 H =   .615, p =  .001, H contr. =   .001
  local         245   .85        global          44   .15
```

Figure 5.6. Two patterns from the final pattern set.

Figure 5.7 shows the results of encoding approximately 20% of the sample using code lengths determined by the statistics of the remaining 80%. If the distribution of nodes is reasonably uniform across the sample, the average code length should be close for each of the 5 samples. For Pattern 180, this appears to be true; we can predict with some confidence the probability distribution for nodes in situations matching the pattern. With Pattern 185, however, the situation is much worse; we would need a much larger sample to decide the actual probabilities. Thus, if we were to automate the selection of refinement patterns, we would rule out pattern 185 as a viable candidate.

pattern		*Test Sample* 1	2	3	4	5	*mean*	*standard deviation*
180	count	69	88	95	63	88		
	ave. length	1.017	1.001	1.102	1.037	1.057	1.024	.021 = 2%
185	count	96	79	61	36	17		
	ave. length	2.579	.373	.309	.332	.255	1.080	1.498 = 139%

Figure 5.7. Average code length using codes based on remaining samples.

This procedure was devised after the results shown in Appendix C were obtained. There were a few obviously bad patterns like 185 above, but the typical pattern produced a standard deviation of 15% for the 5 sample pieces.

Once we have a criterion for deciding on the quality of the sample for a given pattern, we can use it to decide whether a particular pattern refinement is warranted. Figure 4.6 showed a pattern refinement where two new patterns were added. Consider now only one of them, pattern 219. Pattern 7 predicts the length of a list in the parse tree; pattern 219 predicts the length of a list which is a parameter list of a procedure call. Figure 5.8 shows the average code length calculation for pattern 7 before refinement.

pattern		*Test Sample* 1	2	3	4	5	*mean*	*standard deviation*
7	count	1727	1308	1518	1483	1389		
	ave. length	2.437	2.727	2.405	2.362	2.656	2.508	.142 = 5.7%

Figure 5.8. Variability of pattern 7 before refinement.

Recall that we are interested in lowering our entropy estimate. When we refine the pattern, we obtain two new patterns, and replace the contribution to the entropy of the original pattern by a linear combination of the entropy estimates of the two new patterns. Although the entropy estimate for the combined new patterns will never be higher than that of the original, the variability of the sample may well place us in a situation where the mean plus one standard deviation is higher after the refinement than before. This is a usable criterion for deciding on a refinement. Figure 5.9 shows the two patterns after refinement.

		Test Sample						*standard*
pattern		1	2	3	4	5	*mean*	*deviation*
7	count	1160	1084	1208	1092	1065		
	ave. length	2.896	2.945	2.590	2.669	2.859	2.789	.139 = 5.0%
219	count	567	224	310	391	324		
	ave. length	.920	1.012	1.257	.918	1.584	1.107	.253 = 23%
total	count	1727	1308	1518	1483	1389		
	ave. length	2.247	2.614	2.318	2.208	2.561	2.377	.163 = 6.9%

Figure 5.9. Variability of pattern 7 and pattern 219 after refinement.

If we use the mean code length figure for our estimate, the original estimate of 2.508 is replaced by the linear combination

$$\frac{5609}{7425} \cdot 2.789 + \frac{1816}{7425} \cdot 1.107 = 2.377.$$

The linear combination figure for the 5 samples has a standard deviation of .163. We note that 2.377+.163 is less than 2.508+.142, so the refinement is warranted.

5.8. Evaluating the Entropy Estimates for the Program Sample

In this section, we will reexamine the entropy estimate obtained in Chapter 4 for the final pattern set. Certain simplifying assumptions used in the estimates are removed in order to make a more realistic estimate.

String Literals

As we saw in Section 4.2, the string literals that occurred in the test sample were not suited to the same sort of encoding as were the other tree nodes. In the uniform Markov studies, strings were given no special treatment since they do not grossly affect the results. When we calculate the average code length for the program sample, we are in a position to treat the string literal nodes as a special case and use code lengths for them that more accurately reflect the actual number of bits required in an object program.

Quite a bit of work has been done on the problem of "compressing" character strings. Hehner discusses several possible schemes in his thesis [Hehner71]. Some of the more interesting methods for string compression are adaptive, using early parts of the string to encode later ones. The analysis programs for this thesis use a very conservative approach: the string length is given, followed by the characters of the string. In order to have the average code length for strings closer to the entropy, the individual characters are Huffman coded, requiring on the average 5.06 bits per character as opposed to the 7 bit ASCII codes used by the compiler.

Average Code Length for the Program Sample

It is straightforward to modify the average code length formulas of Section 5.6 to consider string literals as a special case. Recall that the entropy estimate for the sample was 1.642 bits per node. This corresponds to an average code length using the entire sample as a training sample, and setting $\varepsilon_{\pi} = 0$, for all π. When we simply add a more realistic cost for string literals to the computation, the entropy value goes to 1.702 bits per node, the 4% increase mentioned in Section 4.2.

More interesting numbers are obtained from the average code lengths for encoding part of the sample using code lengths based on the remaining sample. In this case, we have to be more conservative in our choice of ε_{π} than in the procedure for deciding upon a refinement. The procedure is as follows:

- The encoding process is run on some portion of the samples, with a count of missing nodes maintained for each pattern.

- For each pattern, a value of ε_{π} is chosen equal to the empirical probability of a missing node for that pattern.

- For patterns with a zero empirical probability, ε_π is set to a small positive constant (.001 in this case) unless it is known that indeed there can be no missing nodes. An example of the latter is the var node which is constructed with one of four possibilities as its first son. In these cases, a value of $\varepsilon_\pi = 0$ is warranted. If an empirical probability of 1 is obtained, some smaller value (.5 in this case) is used for ε_π.

Table 5.10 shows the results of encoding 20% of the sample using code lengths based on the remaining 80%. The values of ε_π were chosen by considering the first three samples as described above.

	Test Sample						*standard*
	1	2	3	4	5	*mean*	*deviation*
count	62322	59001	58638	56843	58289		
ave. length	1.940	1.978	1.726	1.729	1.799	1.837	.230 = 13%

Figure 5.10. Average code length and variability for program sample.

The mean code length from Figure 5.10 of 1.84 bits per node is a reasonable estimate of the entropy of the source mdoel defined by the final pattern set. The standard deviation figure gives a measure of confidence in the entropy estimate.

We must be careful when choosing a division of our sample into pieces for the above procedure. If there are few large pieces, the training sample is correspondingly reduced, yielding poor estimates of the probabilities. If there are many small samples, the nodes of the test sample are less likely to have a typical distribution. Figure 5.11 shows the procedure applied to the same sample broken into 10 pieces instead of 5.

	Test Sample									
	1	2	3	4	5	6	7	8	9	10
count	23664	21407	20056	28655	24096	29634	34974	32747	42266	37594
ave. length	1.862	2.060	1.941	1.969	2.002	1.653	1.630	1.534	1.937	1.943

mean = 1.840 *standard deviation* = .376 = 20%

Figure 5.11. Average code length procedure applied to 10 sample pieces.

The mean code length was almost the same for both applications of the procedure, indicating that the probability estimates from 80% of the samples were close to those from 90%. The smaller test sample size in Figure 5.11 results in a larger variability of the average code length, but within the factor of $\sqrt{2}$ expected by doubling the number of samples; the samples are still large enough to have typical distributions of nodes.

5.9. Conclusions

The error estimating procedures produced for this thesis give a reasonably good picture of the variability of the empirical sample. They do not, however give mathematically rigorous estimates of the error bounds. This is an area ripe for further study, presumably by someone with a strong grounding in theoretical statistics.

6. Conclusions

6.1. Applications

The entropy of a program source defines a lower bound for the size of object programs. The most obvious application of this is the evaluation of existing compilers. As we have seen, to the extent that additional program dependencies exist, we can lower the average code length for programs by using more complicated encoding schemes. More complicated encoding schemes, however, entail more complicated decoding schemes or interpreters in the case of object programs. A designer can look at the performance of his current encoding and compare it to the theoretical minimum. If the current performance is "close enough", then the potential increase in size of the interpreter may not warrant further work on encoding.

One assumption of the analysis routines is that the program sample is representative of programs that will be written in the future. Just as a present day programmer might choose ALGOL for a numerical analysis task, LISP for a list processing task, and COBOL for a business application, it makes sense for a given language to have a compiler that can generate several different object codes, each tailored to a type of program. For example, input-output device driver programs and compilers may not have the same instruction mix. One could easily have several data bases of programs on which to run the analysis routines, each containing programs of a given class.

While the patterns used to estimate the entropy do in fact define an encoding for the language, it is not one that is easy to implement. The analysis routines build the set of pattern incrementally and never have to recognize patterns in actual trees after the initial pattern set is created. If one were trying to "automate" the design of an object code, it would probably be worthwhile to limit the number of possible patterns (states of the encoder). One could intuitively think of states such as *statement, expression, string* *constant*, etc. The methodology of this thesis would allow evaluation of the merit of such a constrained set of patterns.

Another related application is the ability to answer questions about programming style. Appendix C.4 - C.6 is full of interesting information. For example, in Pattern 9, we see what sort of parameters are used in procedure calls. These statistics, together with patterns 219 (lengths of parameter lists), and 246 (elements of parameter lists) tell us quite a lot about how we should design procedure linkage. Figure 6.1 shows the percentage of all procedure calls in the sample having a given number of parameters.

parameters	0	1	2	3	4	5
count	760	4040	1390	263	121	42
percent	11%	61%	21%	4%	2%	1%

Figure 6.1. Number of parameters in procedure calls (static count).

These numbers point out an interesting phenomenon. The statistics taken on TENEX-MESA programs, discussed in Section 1.3, that were used to design the encoding of MESA programs indicated that most procedure calls had fewer than 6 parameters. While MESA allows an arbitrary number of parameters, it is more efficient if there are 5 or less. The programmers in the test sample were implementers of the language who knew this; they didn't use more than 5 parameters. This is a feedback situation that one must be careful about when defining new features.

Another use for these program statistics is in language design. Suppose that a designer is adding a feature to the language to simplify a common construct that previously took many symbols to specify. Knowledge of how the construct is used in a large collection of programs can guide the designer in deciding what should be the default values of various parameters to the new feature.

6.2. Extensions and Directions for Further Research

All of the counts of various program constructs studied in this thesis are *static* counts. That is, they reflect how often something appears in the program text rather than how often it occurs in the course of program execution. When Knuth made his study of FORTRAN programs [Knuth71a], he found that less than 4% of a typical program accounted for more than half of its running time. Thus the distribution of use of various statement types was different for the *dynamic* counts, those weighted by the number of times that they are executed. An obvious candidate for extension of the analysis programs of this thesis is to find a way to investigate dynamic statistics of MESA programs.

The first thing that one would need for studying dynamic statistics is a facility for producing them. Since the language was under development at the same time as the static analysis, it was not considered a good idea to add the additional complication of making the necessary modifications to allow statement counts. Presumably such facilities will be provided in the future, but they were not ready in time for this thesis. Most of the static analysis procedures could be extended to allow frequency weights on the trees, but some care would have to be chosen in defining *previous* in the dynamic case. One would probably have to use some global flow analysis, restrict dependencies to cases where the flow is well understood, or establish a known context at all labels.

Another use for the methodology is the ability to simulate and evaluate new features. For example, with a slight extension to the matching routines, one can ask such questions as "How many times is a variable the same as the previously mentioned variable?" Such questions are probably more interesting in the dynamic case, but are still of some merit in static counts.

Another extension mentioned briefly in the preceding Section is the problem of limiting the size and complexity of the interpreter. One interesting, but difficult, approach would be to specify a host machine for the interpreter, say one of the commercially available microprogrammable computers, with a fixed size of control store and try to produce an "optimal" encoding for the language subject to the constraint that the interpreter fit into the chosen machine. A simpler problem would be to limit the number of allowed patterns, thereby limiting the possible interpretations of code bits and hence the size of the interpreter.

As was mentioned in Chapter 5, very little published work is available on error bounds for estimating entropy of a Markov source from experimental data. For that matter, there are only a few papers on error analysis for zero-memory sources [Basharin59] [Pfaffelhuber71] [Nemetz72]. Hopefully, as information theory is applied more often to the analysis of programs and language constructs, there will be a motivation within the statistics and information theory community to provide these very difficult theoretical analyses.

Another building block needed for the "automatic generation of program codes" is a means for deciding what patterns to try for refinement of existing patterns. Such a program falls into the domain of artificial intelligence; a set of heuristics for pattern "growing" could perhaps be obtained by looking at the performance of several programmers doing the task by hand.

In conclusion, it should be noted that even without facilities for automating instruction set design, much useful guidance can be obtained by the ability to obtain program statistics and by the metric that information theory provides for gauging the relative importance of various constructs.

Appendix A. Nodenames Found in Program Parse Trees

Nodename	*Sons*	*Meaning*
abs	1	ABS operator
addr	1	address of variable
and	2	AND operator
apply	2	used by early passes for id [explist]
arraydesc	1	descriptor for ARRAY
assign	2	assignment statement
assignx	2	assignment expression
base	1	base of ARRAY
body	1	list of statements of program or procedure body
call	3	procedure call (proc, args, catch phrases)
caseexp	3	case expression (cv, list of cases, endcase)
casestmt	3	case statement (cv, list of cases, endcase)
caseswitch	3	case statement that can be implemented by a dispatch
casetest	2	case statement items with constant labels
catchmark	1	used to mark label for RETRY, etc.
catchphrase	2	SIGNAL catching statement list
construct	2	record constructor statement
constructx	2	record constructor expression
continue	0	CONTINUE statement
dindex	2	index using ARRAY DESCRIPTOR
div	2	/ operator
dollar	2	field select within son1 variable
dostmt	5	do statement
dot	2	field select with son1 variable pointer
dst	1	dump state (low level machine dependent)
downthru	2	down through range (cv, range)
enable	2	ENABLE statement (catchphrase, statementlist)
entry	0	*frame* = procedure entry point
error	3	ERROR statement
exit	0	EXIT statement
extract	2	extraction from record
fdollar	2	field extraction from procedure return record
fextract	2	extraction for procedure return record
field	0	*frame* = field of a record
forseq	3	sequence type FOR statement iteration
global	0	*frame* = global to procedure bodies
goto	1	GO TO statement
ifexp	3	if expression (boolean, then, else)
ifstmt	3	if statement (boolean, then, else)
in	2	IN relational
index	2	element of an ARRAY

Nodename	*Sons*	*Meaning*
inlinecall	3	call of INLINE code
intCC	2	interval closed on both ends
intCO	2	interval closed on left, open on right
intOC	2	interval open on left, closed on right
intOO	2	interval open on both ends
item	2	general pairing node with many uses
label	2	block with exit labels (stmt list, exit list)
length	1	LENGTH of ARRAY (from DESCRIPTOR)
list	?	only arbitrary length node in trees
local	0	*frame* = local to a procedure body
lst	1	load state (low level machine dependent)
lstf	1	load state and free (low level machine dependent)
loophole	2	treat variable as having another type
max	1	MAX operator (of list)
memory	1	contents of named memory address
min	1	MIN operator (of list)
minus	2	- operator
mod	2	MOD operator
mwconst	1	multiword constant (list of values)
new	3	NEW statement
not	1	NOT operator
nullstmt	0	null statement
num	1	numerical literal
openstmt	2	OPEN statement
or	2	OR operator
plus	2	+ operator
register	1	contents of named REGISTER
relE	2	= operator
relG	2	> operator
relGE	2	>= operator
relL	2	< operator
relLE	2	<= operator
relN	2	# operator (not equal)
resume	1	RESUME from SIGNAL handler
retry	0	RETRY statement
return	1	RETURN statement
row	1	list of value for ARRAY construction
rowcons	2	ARRAY constructor statement
rowconsx	2	ARRAY constructor expression
seqindex	2	character selection from STRING
signal	3	SIGNAL statement
start	3	START statement
stop	1	STOP statement
str	1	STRING literal
svc	1	call on operating system function
syserror	1	ERROR statement for unspecified ERROR
temp	0	associated with constructors
times	2	* operator
uminus	1	unary - operator
unionx	2	associated with constructor for variant records
uparrow	1	indirect address operator

Nodename	*Sons*	*Meaning*
upthru	2	FOR sequence up through range (cv, range)
var	4	variable (frame, address, bit address, length)
vconstruct	2	constructor statement for variant records
vconstructx	2	constructor expression for variant records

Appendix B. Implementation Description

B.1. General Comments

While the low level details of the implementation are not of general interest, there are a few principles and techniques which would be useful to someone implementing a similar system.

The analysis programs and their associated data structures underwent several evolutionary changes. There were two general forces that motivated these changes: increasing amounts of data, and a desire to facilitate incremental changes to the entropy estimate.

On the subject of increasing amounts of data, one should consider the following advice: *If you have a lot of data to process, you should consider using data processing techniques.* One of the more useful pieces of software used in the analysis was a general sort/merge package. It was used in three or four different contexts, simply by supplying it with different *input, output,* and *comparison* procedures in each case. This removed the necessity of keeping all information in the computer main memory at once. In early versions of the analysis routines, the trees were used exactly as obtained from the compiler and converted to the analysis form "on the fly"; for the final version, the trees were converted in a batch mode and all the trees stored on a large data file.

Another useful tool is a hash table package with a very easy-to-use interface. It allows the storage and retrieval of several types of items (1 and 2 word keys, strings, etc.) and allows counts to be associated with items. The set of routines to find and update patterns contains four separate instances of the hash table package.

B.2. Machine Considerations

The analysis programs are written in an earlier version of MESA that runs under the TENEX operating system [Bobrow72]. The machine used has a 36 bit word with 18 bits of address space. A most useful facility provided by TENEX is the mapping of disk file pages (512 words) into the user's address space. This greatly simplifies virtual memory management since "dirty" pages are automatically written back to the disk. The large word size allowed auxiliary information to be added to the trees "for free" when a potential was seen for improving the system performance by using this information. The storage management facilities of the MESA runtime system make it easy to obtain blocks of free storage for making linked data structures, or for providing space for hash tables.

B.3. Token Representation

The basic item under investigation is the tree node. In order to facilitate manipulation of nodenames, a uniform representation, called a *tokenhandle*, is used. A *tokenhandle* occupies 18 bits, or a half-word of memory. It has three fields: a *tokentype*, a *datavalue*, and a *dataflag*. In the final version of the analysis routines, the *tokentype* has one of the following four values:

`nodename`	a non-terminal produced by the compiler
`numlit`	a number
`strlit`	a string
`newnode`	an invented nodename, like `var`, or `global`.

Earlier versions had a larger collection of tokentypes. Interpretation of the *datavalue* field depends upon the *tokentype* and the *dataflag*.

tokentype	*dataflag*	*datavalue*
`nodename`	--	index into an array of strings
`numlit`	FALSE	a small positive or negative number
`numlit`	TRUE	a hash table index

`strlit`	--	a hash table index
`newnode`	--	a hash table index

The overflow numbers and the string constants are contained in the same hash table, which only requires 2000 words of storage for the entire 400,000 words of sample trees. For reasons which don't seem very strong in retrospect, the new nodenames are kept in their own (200 word) hash table.

B.4. Tree Representation

Experience with early versions of the analysis programs shows the inadvisability of having the samples trees share the same address space as the programs. In fact, there are so many trees in the sample that they require more than 18 bits to address when stored in a straightforward way. Nevertheless, it is helpful for the matching routines to have random access to the set of trees. Fortunately, TENEX allows easy maintenance of virtual memories. The data structures used to store the trees are defined using 21 bit pointers, more than enough for the amount of data available for analysis.

Since the total space taken up by the trees is not a critical problem, a very conventional representation is used for them. Each tree has a one word header that contains a *tokenhandle* called the *nodename*, and a *soncount*. It then contains as many more words as it has sons, each containing a *sonvalue*. A *sonvalue* has three fields:

termson	a BOOLEAN variable that says whether this son is terminal or nonterminal.
sonptr	if a terminal son, contains a *tokenhandle*, otherwise contains a 21 bit pointer to the root of the subtree.
patseq	contains the sequence number of the pattern matched by the node in this son position of this particular tree.

The uses of the *patseq* field are discussed in the following section.

B.5. Tree Matching and Pattern Refinement

To decide what larger patterns should be tried as refinements, and to actually accomplish the refinement, a facility for matching trees is required. The routines used for this thesis underwent several complete replacements, but none of the algorithms represent any advances in the art of tree matching. The first routines were very straightforward in design; they could be called "brute force" matching procedures. While slow and lacking in esthetics, the procedures worked well enough on the small sample set available at the time and provided insight for the design of the other analysis routines. When the size of the sample made the matching routines impractical, the *patseq* field was added to the *sonvalue* record to speed things up as discussed below.

For this thesis, we are interested in a specialized form of tree matching: we want to take an existing pattern and see how the distribution of nodes matching that pattern changes when the pattern is made larger. This takes two related forms:

1. For hypothesizing new patterns, we want to obtain statistics for all possible node values occupying a position defined by a particular structural relationship to the existing pattern.

2. For refining an existing pattern, we want to find each node matching the old pattern which also match the new one, note what node occurs in the match, and update the data base containing statistics on the node distribution for the various patterns.

The *patseq* field in the trees makes the above tasks simple. A traversal procedure "walks" through the forest of program trees, maintaining a list of ancestors and brothers. For each tree to be visited, the *patseq* field of the *sonvalue* pointing to that tree is used as an index into an array. If we are not interested in expanding that pattern, the entry is null, otherwise it points to a list of items, each specifying a structurally related node such as *father*, *brother*-3-back, etc, and one of the following actions for matching the tokenhandle which is the nodename in that node:

`simplename`	match a tokenhandle included in the item
`listofnames`	match one of a list of tokenhandles pointed to by a pointer in the item.
`freename`	match any tokenhandle, storing the value in a specified position in an array of free values.
`boundname`	match the tokenhandle stored in a specified position in the array of free values.

During the refinement procedure, whenever an instance of the new pattern is encountered as described in Section 4.4, a *transaction* (*old pattern, new pattern, nodename, count*) is added to a hash table (or the *count* of an existing one incremented), indicating that the node now is in the set of nodes matching the new pattern. At the same time, the *patseq* field of the *sonvalue* pointing to the node is updated to reflect the new pattern. Figure B.1 shows the same tree as Figure 4.5, but with the pattern sequence numbers corresponding to the final pattern set.

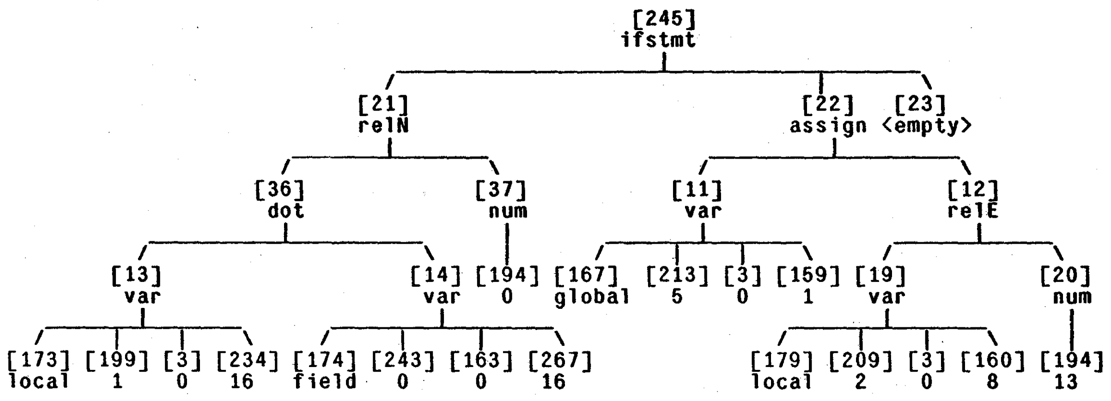


Figure B.1. Program tree showing pattern numbers from final set.

While the *patseq* scheme works satisfactorily, it is probably not the best. If disk space is not a concern, it would speed things up considerably to keep a completely inverted file with a list of tree address for each node matching each pattern, and making the pointers in the trees two-way pointers so that *father* nodes can be found without having to traverse the tree from the top.

B.6. Average Code Length Calculations

Recall from Section 4.5, dealing with program transformations, that we sometimes find it necessary to delete all of the nodes in a subtree from the data base of matched nodes. This is accomplished by a procedure that produces a set of transactions changing the matched pattern of each node to 0. If one applies the procedure to a collection of entire program trees and writes these transactions onto a file, the data can be given a different interpretation. The file contains a concise record of *pattern-nodename* pair counts for the portion of the sample that was used to produce the file. This is precisely the data needed for the entropy formula derived in Section 5.5. If we further sort the data so that all nodes matching a given pattern occur together, it becomes trivial to write a program estimating the entropy for this set of trees.

Of more importance is the *average code length* computation of Section 5.8. These are produced as follows:

- The number of sub-samples, *n,* is decided upon (in this case 5).
- Data files are produce by the *elimination* procedure for *n* collections of program trees of approximately the same magnitude, whose union is the entire sample set. A simple program transforms these data into records that contain:
  pattern sequence, nodename, sub-sample number, count.
- The data for all sub-samples are sorted according to the first three keys named above.
- Another simple program produces a file with *n*+3 word records, containing the following:
  pattern sequence, nodename, sample total, (*n*) *sub-sample totals.*
- It is now simple to calculate the average code cost of one sub-sample in terms of the others by using the sub-sample totals (some of which may be zero) and sample totals for the various patterns and nodes (the actual names of the nodes are irrelevant, though they are contained in the file).

These procedures were developed after most of the data analysis was performed for the thesis. If one were designing a new system to analyze a language, it would be worthwhile to consider how much of the computation can be done by such batch techniques.

B.7. Tree Printing

There are several figures in this thesis that contain diagrams of trees; these were primarily produced by a tree printing program. While there are some shortcomings to the program, there seems to be no published algorithm that does better. See, for example, Knuth's program for printing small binary trees [Knuth71b]. The program is based on a few easily stated principles:

- All nodes at a given level in the tree are at the same level on the page.
- Each nonterminal node is centered over the names of its sons.
- The width of the resulting tree is minimized.

There could be many implementations of such an algorithm. We will loosely discuss the program used for the thesis. The program relies heavily on the fact that all characters and blanks on the printing device are the same width, a restriction that is met by most currently available computer output devices. The program was also designed under the assumption of the availability of large amounts of data space in the computer memory. When we discuss the program, it will be useful to look at some sample output. Figure B.2 is a portion of the tree from Figure 3.2.

```
                                       relN
bracket line  →   .............  ____|____
arrow line    →   ............ /          \
name line     →   ........... dot          num

                         ______|_______    |
                        /              \   |
                      var              var 0
                    ___|____         ___|____
                   /   | |  \       /   | |  \
               local   1 0  16  field   0 0  16
```

Figure B.2. A portion of a printed tree.

The program represents the entire tree image in memory before printing any of it; the unit of representation is a record called a *pageentry.* This record contains one character on each of three lines, called the *bracket,* the *arrow,* and the *name.* The principle data structure is an array called *row.* Each entry of *row* has two fields, an integer named *nextavailable,* and an array of *pageentry's* named *column.* The interpretation of these variables is as follows:

- *row*[*i*]*.nextavailable* is the next free character position of the i^{th} row of the printed tree.

- *row*[*i*]*.column* is an array of *pageentry's,* each containing a single character on each of 3 adjacent lines on the output. *row*[*i*]*.column*[*j*] is the *pageentry* that describes the j^{th} character position of those lines.

- *row*[*i*]*.column*[*j*]*.bracket* is a blank or a character in the line below a nodename (| or _).

- *row*[*i*]*.column*[*j*]*.arrow* is a blank or a character pointing down to a nodename (/, |, or \).

- *row*[*i*]*.column*[*j*]*.name* is a blank or a character in a nodename.

Since trees come in assorted sizes and shapes, it is difficult to see what size to make the various arrays. In the program, the arrays are allocated at run time. The length of the array *row* is made equal to the maximum depth of the tree. The length of the array *row*[*i*].*column* is made equal to the rightmost character position occupied by a nonblank character in any of the three lines of the i^{th} row. The determination of the lengths for the *column* arrays is made by running the *placement* procedure described below without actually storing into the arrays.

In our discussion below, if t is a tree, let $t_1, t_2, \ldots, t_{k_t}$ be the sons of t. Let $t^{nodename}$ denote the nodename of the root of t, and let $|t^{nodename}|$ denote its length. In order to simplify the program, we will have each nodename contain a trailing blank, which is also reflected in its length. The constant *blankentry* is a *pageentry* with all three characters equal to blank.

The Placement Procedure

Most of the work of the program is done by a recursive procedure called *placement,* which is called with a pointer to a tree and a few other parameters and returns the character position of the nodename of its root.

Calling Sequence: *placement* [*t, leftmost, rownum*]
Input Parameters:

t -- a tree to be placed on the "page."

leftmost -- the leftmost character position allowed for its nodename.

rownum -- the number of the row on which to place the name.

Returned Value: *pos* -- position actually given to the nodename.
Side Effects: builds a representation of the tree in the array *row.*

Description of the procedure:

1. (*Make first approximation to placement.*)
   Set l = MAX{ *leftmost, row*[*rownum*].*nextavailable*}

2. (*Terminal node?*) If t has no sons ($k_t = 0$), set *pos* = l, and go to step 12.

```
3.  (Calculate widths.) Set $r = l + |t^{nodename}|$,
    set $totalsons = \sum_{i=1}^{k_t} |t_i^{nodename}|$.

4.  (Place first son.) Set $l = placement\ [\ t_1, (l+r)/2 - (totalsons-1)/2, rownum+1]$,
    set $al = l + |t_1^{nodename}|/2$.

5.  (Only one son?) If $(k_t = 1)$ ,
    set $row[rownum+1].column[al].arrow$ = "|",
    set $r = l + |t_1^{nodename}|$, and go to step 10.

6.  (Point down to first son.) Set $row[rownum+1].column[al].arrow$ = "/".

7.  (Place middle sons.) For $1 < i < k_t$,
    set $a = placement\ [t_i, 0, rownum+1] + |t_i^{nodename}|/2$,
    set $row[rownum+1].column[a].arrow$ = "|".

8.  (Place final son.) Set $r = placement\ [t_{k_t}, 0, rownum+1] + |t_{k_t}^{nodename}|$,
    set $ar = r - (|t_{k_t}^{nodename}|+1)/2$,
    set $row[rownum+1].column[ar].arrow$ = "\".

9.  (Draw bar between outside sons.)
    For $al < i < ar$, set $row[rownum+1].column[i].bracket$ = "_".

10. (Determine position for root.)
    Set $pos = \mathrm{MAX}\{\ row[rownum].nextavailable, (l+r)/2 - |t^{nodename}|/2\}$.

11. (Point down from root.) Set $ac = pos + |t^{nodename}|/2$,
    set $row[rownum+1].column[\mathrm{ac}].bracket$ = "|".

12. (Write root nodename.)
    For $row[rownum].nextavailable \leq i < pos$,
        set $row[rownum].column[i] = blankentry$.
    For $0 \leq \mathrm{i} < |t^{nodename}|$,
        set $row[rownum].column[pos+i].name = t^{nodename}[i]$,
        set $row[rownum].column[pos+i].bracket$ = blank,
        set $row[rownum].column[pos+i].arrow$ = blank.
    Set $row[rownum].nextavailable = pos + |t^{nodename}|$.
```

13. Return *pos*, the placement of *t*.

The Printing Program

The *placement* procedure builds a representation of the tree in the array *row*. Once this is done, it is straightforward to print the tree. This is done by procedures with knowledge of how many rows will fit on a page and the width of a page. The resulting output is a collection of pages which can be cut and pasted into a large tree diagram.

In the implementation of *placement*, any statements that actually write into the *column* arrays first test a BOOLEAN variable. Hence, *placement* is run, resulting in values for the *nextavailable* fields, which are then used to allocate *column* arrays of the proper size. Several fields were added to the tree nodes to facilitate printing. A *width* field saves computation when determining the width of the *nodename*, and allows the trailing blank to be synthesized by having the *width* be one more than the actual string width. A *value* field allows a number to be associated with a node and to be printed in brackets above the nodename. This is done by adding a *value* character field to the *pageentry* record. When the *value* is printed, the *width* field contains the maximum space needed for the *nodename* or *value*.

The final printing program is described below:

- Compute maximum depth, setting *width* fields.
- Allocate *row* array, with empty *column* elements.
- Call *placement* [*root*, 0, 0], with printing inhibited.
- Allocate *column* elements of proper size, resetting *nextavailable* fields.
- Call *placement* [*root*, 0, 0], with printing enabled.
- Produce printed output.

Experience with the Tree Printer

Another common way of printing trees is the "table of contents" listing. In this form, the structure is shown by indentation. All nodes at a given level in the tree are given the same indentation in the listing. Such listings often are more compact than tree diagrams, and show local structure adequately. They do not, however, show the global structure of the tree nearly as well as a diagram does.

The tree printing program proved very helpful when designing and debugging the analysis routines. There are several data structure that are treelike: programs trees, patterns, etc. Procedures were written to convert each of them into trees acceptable to the printer. For program trees, which tend to be large, the procedure provides an option of truncating the tree at a specified depth. One can set the *value* of the node to its address, its pattern sequence number, its "cost", or other useful information. Below we shall see some of the problems with the program.

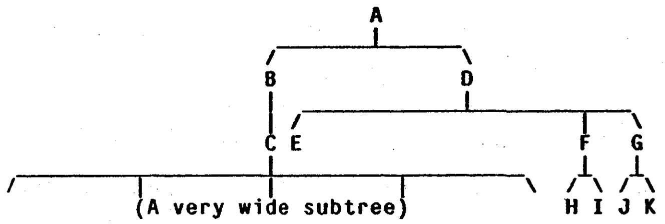


Figure B.3. Example tree showing a shortcoming of the printing algorithm.

The principal shortcoming of the algorithm is somewhat difficult to demonstrate in a small tree. Figure B.3 is a somewhat contrived example that points out the problem. Consider the sons of node D: they are E, F, and G. Since the subtree E is "shallow", it is placed quite far to the left. The nodes F and G must be placed considerably farther to the right in order to make room for *their* subtrees. With larger, wider trees, a shallow subtree such as E can be so far from its brothers that it gets "lost". Shallow subtrees in son positions other than the first lead to uneven spacing of the sons. One could probably add a pass to the algorithm that, once having established the rightmost subtree of a node, then reformats the other subtrees for compactness and even spacing.

The array structure used to represent the tree in memory was chosen in order to make the final, printing pass of the program very easy, and since it was written for a large computer with virtual memory, was a reasonable choice. For a smaller machine, some sort of list structure would probably be better, but would require a more sophisticated algorithm for slicing the diagram up into page-sized pieces for printing.

Appendix C. Detailed Pattern Data

This appendix contains the detailed statistics from the sample set. Both the initial and final pattern sets are shown, together with their contribution to the total entropy estimate. The patterns are shown sorted by various keys, but the detailed listing of nodes associated with a pattern is given only for the listing sorted by pattern number.

C.1. Initial Pattern Set -- Sorted by Pattern Number

In order to conserve space, the full set of nodes matching a given pattern is not always given. Any node whose conditional probability exceeds .005 is shown, however.

```
(1)    var[@]
 f = 40289 H =  1.762, p =  .136, H contr. =   .239
    4 cases
  local       17580   .44      field       7744   .19
  global      12279   .30      entry       2686   .07

(2)    var[*,@]
 f = 40289 H =  3.680, p =  .136, H contr. =   .499
   154 cases (18 shown)
  0           12914   .32      6            946   .02      13          316   .01
  1            6658   .17      9            833   .02      16          247   .01
  2            4014   .10      7            825   .02      15          221   .01
  3            3131   .08      10           590   .01      17          209   .01
  4            2076   .05      12           469   .01      <<others>> 2943   .07
  8            1532   .04      11           464   .01
  5            1517   .04      14           384   .01

(3)    var[*,*,@]
 f = 40289 H =   .336, p =  .136, H contr. =   .046
   16 cases
  0           38841   .96      4             71   .00      7            20   .00
  2             432   .01      6             63   .00      11           18   .00
  3             283   .01      12            56   .00      10           13   .00
  1             175   .00      5             54   .00      13            8   .00
  14            104   .00      15            33   .00
  8              90   .00      9             28   .00

(4)    var[*,*,*,@]
 f = 40289 H =  2.099, p =  .136, H contr. =   .285
   67 cases (13 shown)
  16          26020   .65      8            451   .01      3           334   .01
  14           6502   .16      2            414   .01      48          215   .01
  1            1509   .04      64           381   .01      0           211   .01
  32           1168   .03      11           357   .01      <<others>> 1769   .04
  15            611   .02      4            347   .01

(5)    list[*,...@]
 f = 22481 H =  3.763, p =  .076, H contr. =   .285
   72 cases (20 shown)
```

C.1. *Initial Pattern Set -- Sorted by Pattern Number*

```
 assign      5085   .23      dot          582   .03      addr          207   .01
 call        3573   .16      casetest     572   .03      <empty>       188   .01
 num         2623   .12      casestmt     509   .02      fextract      185   .01
 var         2420   .11      dostmt       448   .02      unionx        179   .01
 ifstmt      1528   .07      str          269   .01      construct     165   .01
 return      1409   .06      relE         233   .01      plus          127   .01
 item        1068   .05      dollar       219   .01      <<others>>    892   .04

(6)    num[@]
 f = 10220 H =  5.132, p =  .034, H contr. =   .177
 408 cases (23 shown)
 0           2254   .22      16           196   .02      9              73   .01
 1           1786   .17      13           161   .02      32767          73   .01
 2            748   .07      5            143   .01      15             59   .01
 3            537   .05      7            138   .01      12             58   .01
 16383        397   .04      6            113   .01      14             54   .01
 -1           345   .03      8             92   .01      255            54   .01
 65535        242   .02      10            92   .01      11             53   .01
 4            221   .02      32            82   .01      <<others>>   2249   .22

(7)    list[@]
 f =  7425 H =  2.476, p =  .025, H contr. =   .062
  38 cases (11 shown)
 2           3599   .48      5            322   .04      10             61   .01
 3           1139   .15      6            208   .03      0              60   .01
 1           1038   .14      7            113   .02      9              54   .01
 4            591   .08      8             80   .01      <<others>>    160   .02

(8)    call[@]
 f =  6616 H =   .238, p =  .022, H contr. =   .005
 var         6370   .96      dot          236   .04      dollar         10   .00

(9)    call[*,@]
 f =  6616 H =  2.961, p =  .022, H contr. =   .066
  27 cases (12 shown)
 list        1816   .27      dollar       209   .03      dindex         56   .01
 var         1739   .26      str          204   .03      ifexp          47   .01
 <empty>      760   .11      call         189   .03      <<others>>    142   .02
 num          686   .10      addr         129   .02
 dot          571   .09      plus          68   .01

(10)   call[*,*,@]
 f =  6616 H =   .112, p =  .022, H contr. =   .002
 <empty>     6517   .99      catchphrase   99   .01

(11)   assign[@]
 f =  5826 H =  1.413, p =  .020, H contr. =   .028
   9 cases
 var         4011   .69      uparrow      102   .02      seqindex       17   .00
 dot         1090   .19      index         79   .01      register        9   .00
 dollar       450   .08      dindex        63   .01      memory          5   .00

(12)   assign[*,@]
 f =  5826 H =  3.591, p =  .020, H contr. =   .070
  42 cases (18 shown)
 call        1288   .22      addr         208   .04      uparrow        36   .01
 num         1078   .19      assignx      153   .03      index          36   .01
 var          775   .13      minus        134   .02      mwconst        34   .01
 dot          580   .10      dindex       113   .02      times          31   .01
 plus         377   .06      inlinecall    72   .01      <<others>>    213   .04
 <empty>      314   .05      arraydesc     63   .01
 dollar       261   .04      ifexp         60   .01

(13)   dot[@]
 f =  5726 H =  1.400, p =  .019, H contr. =   .027
   9 cases
 var         2965   .52      dollar       110   .02      minus           7   .00
 plus        2352   .41      register      16   .00      call            7   .00
 dot          254   .04      num           11   .00      assignx         4   .00
```

C.1. *Initial Pattern Set -- Sorted by Pattern Number*

```
(14)   dot[*,@]
 f =  5726 H =  0.000, p =  .019, H contr. =  0.000
  var          5726  1.00

(15)   plus[@]
 f =  4039 H =  1.037, p =  .014, H contr. =   .014
   22 cases (11 shown)
  var          3396   .84        times         39    .01        plus           16    .00
  dot           357   .09        minus         25    .01        call           15    .00
  dollar         60   .01        register      22    .01        caseexp         5    .00
  num            49   .01        index         21    .01        <<others>>     34    .01

(16)   plus[*,@]
 f =  4039 H =  1.279, p =  .014, H contr. =   .017
   17 cases (11 shown)
  var          3085   .76        call          51    .01        caseexp         6    .00
  num           504   .12        times         18    .00        ifexp           5    .00
  dollar        182   .05        div           13    .00        minus           5    .00
  dot           149   .04        inlinecall    10    .00        <<others>>     11    .00

(17)   dollar[@]
 f =  2770 H =  2.051, p =  .009, H contr. =   .019
    8 cases
  var           968   .35        dollar       109    .04        call           18    .01
  uparrow       952   .34        index         77    .03        assignx         1    .00
  dot           568   .21        dindex        77    .03

(18)   dollar[*,@]
 f =  2770 H =  0.000, p =  .009, H contr. =  0.000
  var          2770  1.00

(19)   relE[@]
 f =  2348 H =  2.010, p =  .008, H contr. =   .016
   14 cases
  <empty>      1249   .53        call          50    .02        index           7    .00
  var           558   .24        inlinecall    15    .01        plus            5    .00
  dot           230   .10        dindex        12    .01        minus           2    .00
  dollar        136   .06        seqindex      11    .00        uparrow         1    .00
  assignx        64   .03        mod            8    .00

(20)   relE[*,@]
 f =  2348 H =   .819, p =  .008, H contr. =   .006
   15 cases
  num          1976   .84        call           4    .00        plus            1    .00
  var           298   .13        dindex         3    .00        addr            1    .00
  dot            41   .02        mwconst        3    .00        seqindex        1    .00
  dollar         10   .00        relN           2    .00        length          1    .00
  relE            5   .00        or             1    .00        register        1    .00

(21)   ifstmt[@]
 f =  1810 H =  3.047, p =  .006, H contr. =   .019
   16 cases
  relE          593   .33        or            86    .05        relLE          20    .01
  relN          370   .20        dot           57    .03        in             18    .01
  not           157   .09        call          48    .03        assignx         3    .00
  var           135   .07        relL          36    .02        dindex          3    .00
  and           132   .07        relGE         29    .02
  relG          101   .06        dollar        22    .01

(22)   ifstmt[*,@]
 f =  1810 H =  3.059, p =  .006, H contr. =   .019
   24 cases (15 shown)
  list          598   .33        signal        70    .04        resume         12    .01
  call          324   .18        ifstmt        40    .02        construct      11    .01
  assign        249   .14        syserror      37    .02        openstmt       10    .01
  return        196   .11        goto          33    .02        <<others>>     32    .02
  error          82   .05        dostmt        23    .01
  exit           75   .04        casestmt      18    .01
```

C.1. *Initial Pattern Set -- Sorted by Pattern Number*

```
(23)   ifstmt[*,*,@]
 f =  1810 H =  1.329, p =  .006, H contr. =   .008
  18 cases (11 shown)
 <empty>     1406   .78      ifstmt        45   .02      return        6   .00
 list         173   .10      casestmt      13   .01      goto          4   .00
 call          69   .04      openstmt      10   .01      fextract      2   .00
 assign        65   .04      dostmt         7   .00      <<others>>   10   .01

(24)   return[@]
 f =  1683 H =  2.521, p =  .006, H contr. =   .014
  30 cases (12 shown)
 <empty>      839   .50      caseexp       32   .02      relE         15   .01
 var          337   .20      constructx    31   .02      plus         15   .01
 num          148   .09      dollar        26   .02      <<others>>   47   .03
 list          74   .04      ifexp         24   .01
 call          73   .04      dot           22   .01

(25)   @
 f =  1513 H =  0.000, p =  .005, H contr. =  0.000
 body        1513  1.00

(26)   body[@]
 f =  1513 H =  0.000, p =  .005, H contr. =  0.000
 list        1513  1.00

(27)   uparrow[@]
 f =  1270 H =  1.191, p =  .004, H contr. =   .005
   7 cases
 plus         823   .65      dollar         4   .00      register      1   .00
 var          379   .30      minus          2   .00
 num           60   .05      dot            1   .00

(28)   item[@]
 f =  1214 H =   .936, p =  .004, H contr. =   .004
   9 cases
 relE        1021   .84      in            25   .02      relN          2   .00
 list          97   .08      relL           7   .01      relLE         2   .00
 lbl           52   .04      relG           7   .01      relGE         1   .00

(29)   item[*,@]
 f =  1214 H =  3.315, p =  .004, H contr. =   .014
  36 cases (19 shown)
 list         425   .35      num           26   .02      resume       11   .01
 assign       187   .15      dollar        20   .02      and          10   .01
 call         170   .14      goto          20   .02      exit         10   .01
 ifstmt       102   .08      openstmt      17   .01      dot           7   .01
 casestmt      42   .03      caseexp       14   .01      continue      7   .01
 nullstmt      41   .03      signal        12   .01      <<others>>   45   .04
 return        37   .03      error         11   .01

(30)   casestmt[@]
 f =   639 H =  1.543, p =  .002, H contr. =   .003
  11 cases
 dollar       450   .70      num           17   .03      uparrow       2   .00
 var           67   .10      assignx        4   .01      inlinecall    1   .00
 dot           55   .09      seqindex       3   .00      index         1   .00
 call          37   .06      minus          2   .00

(31)   casestmt[*,@]
 f =   639 H =  0.000, p =  .002, H contr. =  0.000
 list         639  1.00

(32)   casestmt[*,*,@]
 f =   639 H =  2.581, p =  .002, H contr. =   .006
  14 cases
 <empty>      269   .42      assign        22   .03      nullstmt      7   .01
 syserror     151   .24      return        22   .03      goto          4   .01
 list          61   .10      ifstmt        13   .02      casestmt      2   .00
```

C.1. *Initial Pattern Set -- Sorted by Pattern Number*

```
  signal        35   .05      exit          10   .02      openstmt        1   .00
  call          32   .05      error         10   .02

(33)   casetest[@]
 f =    572 H =   .597, p =  .002, H contr.  =   .001
  num          489   .85      list          83   .15

(34)   casetest[*,@]
 f =    572 H =  2.612, p =  .002, H contr.  =   .005
   15 cases
  list         194   .34      nullstmt      16   .03      openstmt        4   .01
  assign       140   .24      str           15   .03      exit            3   .01
  call         112   .20      return        11   .02      goto            2   .00
  num           41   .07      ifexp          6   .01      signal          2   .00
  ifstmt        20   .03      casestmt       5   .01      label           1   .00

(35)   addr[@]
 f =    570 H =  1.824, p =  .002, H contr.  =   .004
    6 cases
  var          339   .59      uparrow       68   .12      index          32   .06
  dot           84   .15      dollar        32   .06      dindex         15   .03

(36)   relN[@]
 f =    538 H =  2.411, p =  .002, H contr.  =   .004
   16 cases
  var          209   .39      dindex         6   .01      mod             2   .00
  dot          157   .29      seqindex       6   .01      fdollar         1   .00
  dollar        69   .13      plus           4   .01      uparrow         1   .00
  call          37   .07      index          3   .01      length          1   .00
  inlinecall    24   .04      <empty>        2   .00
  assignx       14   .03      minus          2   .00

(37)   relN[*,@]
 f =    538 H =  1.174, p =  .002, H contr.  =   .002
   10 cases
  num          430   .80      call           7   .01      inlinecall      2   .00
  var           53   .10      addr           5   .01      mwconst         1   .00
  dot           24   .04      seqindex       5   .01
  dollar         8   .01      uparrow        3   .01

(38)   str[@]
 f =    511 H =  8.735, p =  .002, H contr.  =   .015
  449 cases (11 shown)
  "Break"        4   .01      "VM."          3   .01      "error"         3   .01
  "Trace"        4   .01      " XXX"         3   .01      "Error # "      3   .01
  ".xm"          3   .01      "NIL"          3   .01      "New"           2   .00
  " -- "         3   .01      ".XM"          3   .01      <<others>>    477   .93

(39)   dostmt[@]
 f =    484 H =  1.610, p =  .002, H contr.  =   .003
    4 cases
  <empty>      219   .45      forseq        84   .17
  upthru       170   .35      downthru      11   .02

(40)   dostmt[*,@]
 f =    484 H =  1.777, p =  .002, H contr.  =   .003
   12 cases
  <empty>      277   .57      relL          10   .02      in              3   .01
  not          136   .28      and            5   .01      var             3   .01
  relN          28   .06      relG           5   .01      relLE           2   .00
  relE          11   .02      relGE          3   .01      or              1   .00

(41)   dostmt[*,*,@]
 f =    484 H =  2.224, p =  .002, H contr.  =   .004
   13 cases
  list         253   .52      openstmt       8   .02      inlinecall      1   .00
  assign        66   .14      label          7   .01      dostmt          1   .00
  ifstmt        54   .11      signal         3   .01      catchmark       1   .00
  casestmt      44   .09      nullstmt       3   .01
```

C.1. *Initial Pattern Set -- Sorted by Pattern Number*

```
  call           40   .08      enable         3   .01

(42)   dostmt[*,*,*,@]
 f =   484 H =   .133, p =  .002, H contr. =   .000
  <empty>       475   .98      item           9   .02

(43)   dostmt[*,*,*,*,@]
 f =   484 H =   .292, p =  .002, H contr. =   .000
    8 cases
  <empty>       468   .97      ifstmt         2   .00      goto           1   .00
  list            6   .01      return         2   .00      exit           1   .00
  assign          3   .01      call           1   .00

(44)   minus[@]
 f =   469 H =  2.672, p =  .002, H contr. =   .004
   18 cases
  var           224   .48      call          11   .02      uminus         3   .01
  dot            62   .13      div           10   .02      length         3   .01
  <empty>        45   .10      minus          8   .02      times          2   .00
  plus           38   .08      index          7   .01      uparrow        2   .00
  dollar         25   .05      dindex         4   .01      assignx        1   .00
  num            20   .04      ifexp          3   .01      abs            1   .00

(45)   minus[*,@]
 f =   469 H =  1.590, p =  .002, H contr. =   .003
   14 cases
  num           318   .68      minus          4   .01      ifexp          1   .00
  var            91   .19      times          4   .01      assignx        1   .00
  dot            17   .04      index          4   .01      div            1   .00
  dollar         15   .03      addr           2   .00      inlinecall     1   .00
  plus            8   .02      call           2   .00

(46)   dindex[@]
 f =   433 H =   .195, p =  .001, H contr. =   .000
  var           420   .97      dot           13   .03

(47)   dindex[*,@]
 f =   433 H =  2.450, p =  .001, H contr. =   .004
   11 cases
  var           131   .30      minus         21   .05      call           3   .01
  num           102   .24      dot            8   .02      uparrow        2   .00
  times          89   .21      assignx        4   .01      ifexp          1   .00
  plus           68   .16      dollar         4   .01

(48)   not[@]
 f =   382 H =  2.636, p =  .001, H contr. =   .003
   13 cases
  relE          118   .31      in            19   .05      and            1   .00
  var            97   .25      relL           5   .01      relN           1   .00
  call           50   .13      relG           5   .01      relGE          1   .00
  dot            47   .12      or             4   .01
  dollar         32   .08      ifexp          2   .01

(49)   assignx[@]
 f =   348 H =  1.308, p =  .001, H contr. =   .002
    7 cases
  var           246   .71      uparrow        7   .02      dindex         1   .00
  dot            69   .20      seqindex       4   .01
  dollar         18   .05      index          3   .01

(50)   assignx[*,@]
 f =   348 H =  3.349, p =  .001, H contr. =   .004
   21 cases (14 shown)
  call           73   .21      assignx       23   .07      register       6   .02
  num            71   .20      minus         19   .05      inlinecall     3   .01
  dot            40   .11      dollar        19   .05      seqindex       3   .01
  plus           34   .10      dindex        12   .03      relE           2   .01
  var            30   .09      addr           6   .02      <<others>>     7   .02
```

Appendix C: Detailed Pattern Data

C.1. *Initial Pattern Set -- Sorted by Pattern Number*

```
(51)   index[@]
 f =   338 H =   .965, p =  .001, H contr. =   .001
  var           265   .78    dollar        42   .12      dot             31   .09

(52)   index[*,@]
 f =   338 H =  2.325, p =  .001, H contr. =   .003
   11 cases
  var           106   .31    dollar        13   .04      inlinecall       2   .01
  times          85   .25    mod            4   .01      ifexp            1   .00
  num            85   .25    plus           3   .01      div              1   .00
  minus          36   .11    dot            2   .01

(53)   times[@]
 f =   317 H =  1.702, p =  .001, H contr. =   .002
   13 cases
  var           227   .72    plus           9   .03      length           2   .01
  minus          22   .07    inlinecall     4   .01      ifexp            1   .00
  call           18   .06    times          3   .01      abs              1   .00
  dot            16   .05    div            2   .01
  dollar         10   .03    uminus         2   .01

(54)   times[*,@]
 f =   317 H =   .535, p =  .001, H contr. =   .001
    5 cases
  num           292   .92    dot            8   .03      dollar           1   .00
  var            10   .03    call           6   .02

(55)   inlinecall[@]
 f =   262 H =  3.019, p =  .001, H contr. =   .003
   18 cases
  BITAND         86   .33    Stop           9   .03      PORTI            2   .01
  BITSHIFT       42   .16    BITXOR         7   .03      CONVERT          1   .00
  BITOR          39   .15    LDIVMOD        4   .02      PUSH             1   .00
  DIVMOD         26   .10    BLOCK          4   .02      USC              1   .00
  COPY           17   .06    NovaOutLd      3   .01      LongDiv          1   .00
  BITNOT         16   .06    NovaInLd       2   .01      LongMult         1   .00

(56)   inlinecall[*,@]
 f =   262 H =   .843, p =  .001, H contr. =   .001
    8 cases
  list          230   .88    var            7   .03      uparrow          1   .00
  num             8   .03    dot            4   .02      index            1   .00
  <empty>         7   .03    inlinecall     4   .02

(57)   and[@]
 f =   251 H =  3.104, p =  .001, H contr. =   .003
   14 cases
  relE           60   .24    call          12   .05      or               5   .02
  and            51   .20    dot            8   .03      relGE            3   .01
  relN           37   .15    relL           7   .03      in               3   .01
  var            32   .13    relG           6   .02      relLE            1   .00
  not            20   .08    dollar         6   .02

(58)   and[*,@]
 f =   251 H =  3.113, p =  .001, H contr. =   .003
   17 cases
  relE           85   .34    or            10   .04      caseexp          2   .01
  relN           42   .17    dot           10   .04      index            2   .01
  not            28   .11    in             8   .03      and              1   .00
  call           22   .09    dollar         7   .03      relGE            1   .00
  var            12   .05    relL           6   .02      fdollar          1   .00
  relG           11   .04    relLE          3   .01

(59)   ifexp[@]
 f =   211 H =  2.315, p =  .001, H contr. =   .002
   12 cases
  relE          102   .48    relN           6   .03      dollar           4   .02
  var            54   .26    relL           6   .03      or               3   .01
  in             14   .07    dot            6   .03      not              3   .01
```

C.1. *Initial Pattern Set -- Sorted by Pattern Number*

```
 relG           8   .04     and            4   .02     relGE          1   .00

(60)   ifexp[*,@]
 f =   211 H =  2.433, p =  .001, H contr. =   .002
   18 cases (11 shown)
 num          115   .55     dollar         9   .04     str            3   .01
 dot           25   .12     ifexp          6   .03     index          2   .01
 call          18   .09     plus           3   .01     memory         2   .01
 var           18   .09     minus          3   .01     <<others>>     7   .03

(61)   ifexp[*,*,@]
 f =   211 H =  3.005, p =  .001, H contr. =   .002
   21 cases
 num           95   .45     plus           6   .03     uminus         2   .01
 dot           20   .09     str            4   .02     addr           2   .01
 var           20   .09     caseexp        3   .01     relN           1   .00
 call          14   .07     index          3   .01     relL           1   .00
 ifexp         13   .06     dindex         3   .01     times          1   .00
 dollar         9   .04     min            3   .01     inlinecall     1   .00
 minus          7   .03     not            2   .01     constructx     1   .00

(62)   fextract[@]
 f =   196 H =  0.000, p =  .001, H contr. =  0.000
 list         196  1.00

(63)   fextract[*,@]
 f =   196 H =   .672, p =  .001, H contr. =   .000
 call         166   .85     inlinecall    28   .14     signal         2   .01

(64)   signal[@]
 f =   186 H =   .524, p =  .001, H contr. =   .000
 var          164   .88     dot           22   .12

(65)   signal[*,@]
 f =   186 H =  1.602, p =  .001, H contr. =   .001
    5 cases
 <empty>      104   .56     var           31   .17     num            2   .01
 list          44   .24     dollar         5   .03

(66)   signal[*,*,@]
 f =   186 H =  0.000, p =  .001, H contr. =  0.000
 <empty>      186  1.00

(67)   intCC[@]
 f =   183 H =   .483, p =  .001, H contr. =   .000
    4 cases
 num          168   .92     dot            2   .01
 var           12   .07     minus          1   .01

(68)   intCC[*,@]
 f =   183 H =  1.466, p =  .001, H contr. =   .001
    7 cases
 num          133   .73     plus           7   .04     dollar         1   .01
 var           19   .10     div            7   .04
 dot           10   .05     minus          6   .03

(69)   construct[@]
 f =   183 H =  1.817, p =  .001, H contr. =   .001
    6 cases
 var           90   .49     dot           26   .14     dollar         3   .02
 uparrow       52   .28     dindex        10   .05     index          2   .01

(70)   construct[*,@]
 f =   183 H =  0.000, p =  .001, H contr. =  0.000
 list         183  1.00

(71)   unionx[@]
 f =   179 H =  0.000, p =  .001, H contr. =  0.000
 var          179  1.00
```

C.1. *Initial Pattern Set -- Sorted by Pattern Number*

```
(72)   unionx[*,@]
 f =   179 H =  0.000, p =  .001, H contr. =  0.000
  list          179  1.00

(73)   upthru[@]
 f =   170 H =   .767, p =  .001, H contr. =   .000
  var           132   .78     <empty>      38   .22

(74)   upthru[*,@]
 f =   170 H =  1.220, p =  .001, H contr. =   .001
    4 cases
  intCC          83   .49     intOO         3   .02
  intCO          81   .48     intOC         3   .02

(75)   relG[@]
 f =   159 H =  2.109, p =  .001, H contr. =   .001
   11 cases
  var            94   .59     <empty>       8   .05        times            1   .01
  dot            21   .13     plus          7   .04        inlinecall       1   .01
  dollar         10   .06     length        5   .03        abs              1   .01
  assignx         9   .06     index         2   .01

(76)   relG[*,@]
 f =   159 H =  1.386, p =  .001, H contr. =   .001
    9 cases
  num           120   .75     dollar        6   .04        ifexp            1   .01
  var            14   .09     index         3   .02        mod              1   .01
  dot            11   .07     minus         2   .01        max              1   .01

(77)   lbl[@]
 f =   159 H =   .984, p =  .001, H contr. =   .001
    5 cases
  1             127   .80     3             4   .03        5                2   .01
  2              23   .14     4             3   .02

(78)   catchphrase[@]
 f =   130 H =  1.040, p =  .000, H contr. =   .000
  item           98   .75     <empty>      20   .15        list            12   .09

(79)   catchphrase[*,@]
 f =   130 H =  1.024, p =  .000, H contr. =   .000
    8 cases
  <empty>       109   .84     continue      4   .03        assign           1   .01
  goto            8   .06     list          2   .02        error            1   .01
  inlinecall      4   .03     call          1   .01

(80)   error[@]
 f =   129 H =   .065, p =  .000, H contr. =   .000
  var           128   .99     dot           1   .01

(81)   error[*,@]
 f =   129 H =  1.903, p =  .000, H contr. =   .001
    9 cases
  <empty>        59   .46     num           6   .05        ifexp            1   .01
  var            47   .36     dot           3   .02        plus             1   .01
  list            9   .07     str           2   .02        addr             1   .01

(82)   error[*,*,@]
 f =   129 H =   .065, p =  .000, H contr. =   .000
  <empty>       128   .99     catchphrase   1   .01

(83)   or[@]
 f =   127 H =  2.955, p =  .000, H contr. =   .001
   11 cases
  relE           38   .30     relG          8   .06        dot              5   .04
  relN           21   .17     var           8   .06        call             2   .02
  not            20   .16     and           7   .06        assignx          1   .01
  or             10   .08     relL          7   .06
```

C.1. *Initial Pattern Set -- Sorted by Pattern Number*

```
(84)   or[*,@]
 f =   127 H =  2.856, p =  .000, H contr. =   .001
   14 cases
  relE          43   .34      relG          5   .04      dollar        2   .02
  and           25   .20      dot           4   .03      relGE         1   .01
  relN          21   .17      call          3   .02      relLE         1   .01
  not           10   .08      caseexp       2   .02      in            1   .01
  var            7   .06      relL          2   .02

(85)   goto[@]
 f =   105 H =  0.000, p =  .000, H contr. =  0.000
  lbl          105  1.00

(86)   in[@]
 f =   102 H =  1.492, p =  .000, H contr. =   .001
    6 cases
  var           57   .56      minus         5   .05      plus          1   .01
  <empty>       35   .34      dollar        3   .03      call          1   .01

(87)   in[*,@]
 f =   102 H =   .323, p =  .000, H contr. =   .000
  intCC         96   .94      intCO         6   .06

(88)   intCO[@]
 f =    94 H =   .849, p =  .000, H contr. =   .000
    4 cases
  num           78   .83      dot           4   .04
  var           11   .12      length        1   .01

(89)   intCO[*,@]
 f =    94 H =  2.573, p =  .000, H contr. =   .001
    9 cases
  var           38   .40      plus         10   .11      min           3   .03
  length        15   .16      dollar        7   .07      div           2   .02
  dot           12   .13      minus         5   .05      call          2   .02

(90)   relL[@]
 f =    93 H =  2.197, p =  .000, H contr. =   .001
    9 cases
  var           51   .55      assignx       9   .10      call          2   .02
  dollar        10   .11      dot           6   .06      index         2   .02
  <empty>        9   .10      plus          3   .03      div           1   .01

(91)   relL[*,@]
 f =    93 H =  2.141, p =  .000, H contr. =   .001
    9 cases
  num           46   .49      dollar        6   .06      times         1   .01
  var           20   .22      length        4   .04      index         1   .01
  dot           12   .13      div           2   .02      min           1   .01

(92)   openstmt[@]
 f =    91 H =  0.000, p =  .000, H contr. =  0.000
  <empty>       91  1.00

(93)   openstmt[*,@]
 f =    91 H =  1.052, p =  .000, H contr. =   .000
    8 cases
  list          76   .84      casestmt      2   .02      ifstmt        1   .01
  enable         5   .05      inlinecall    1   .01      dostmt        1   .01
  label          4   .04      assign        1   .01

(94)   div[@]
 f =    87 H =  2.183, p =  .000, H contr. =   .001
    7 cases
  var           37   .43      dollar        6   .07      minus         1   .01
  plus          23   .26      call          6   .07
  dot           11   .13      times         3   .03
```

C.1. *Initial Pattern Set -- Sorted by Pattern Number*

```
(95)   div[*,@]
 f =     87 H =   .572, p =  .000, H contr. =   .000
    4 cases
  num           79   .91      dot            3   .03
  var            4   .05      times          1   .01

(96)   register[@]
 f =     87 H =  0.000, p =  .000, H contr. =  0.000
  num           87  1.00

(97)   caseexp[@]
 f =     84 H =  1.343, p =  .000, H contr. =   .000
    7 cases
  dollar        62   .74      call           2   .02      num            1   .01
  var           12   .14      inlinecall     2   .02
  dot            4   .05      dindex         1   .01

(98)   caseexp[*,@]
 f =     84 H =  0.000, p =  .000, H contr. =  0.000
  list          84  1.00

(99)   caseexp[*,*,@]
 f =     84 H =   .885, p =  .000, H contr. =   .000
    7 cases
  num           73   .87      dot            2   .02      var            1   .01
  call           3   .04      str            2   .02
  ifexp          2   .02      dindex         1   .01

(100)  forseq[@]
 f =     84 H =  0.000, p =  .000, H contr. =  0.000
  var           84  1.00

(101)  forseq[*,@]
 f =     84 H =  2.496, p =  .000, H contr. =   .001
   10 cases
  var           30   .36      index          4   .05      minus          1   .01
  dot           25   .30      num            4   .05      div            1   .01
  call          10   .12      dindex         3   .04
  dollar         4   .05      plus           2   .02

(102)  forseq[*,*,@]
 f =     84 H =  1.745, p =  .000, H contr. =   .000
    5 cases
  dot           49   .58      var           10   .12      minus          3   .04
  call          14   .17      plus           8   .10

(103)  seqindex[@]
 f =     82 H =   .592, p =  .000, H contr. =   .000
  var           72   .88      dot            9   .11      dindex         1   .01

(104)  seqindex[*,@]
 f =     82 H =  1.736, p =  .000, H contr. =   .000
    6 cases
  var           52   .63      dot            7   .09      assignx        4   .05
  minus         10   .12      num            7   .09      plus           2   .02

(105)  caseswitch[@]
 f =     81 H =  1.175, p =  .000, H contr. =   .000
  minus         45   .56      <empty>       33   .41      plus           3   .04

(106)  caseswitch[*,@]
 f =     81 H =  0.000, p =  .000, H contr. =  0.000
  num           81  1.00

(107)  caseswitch[*,*,@]
 f =     81 H =  0.000, p =  .000, H contr. =  0.000
  list          81  1.00

(108)  arraydesc[@]
```

C.1. *Initial Pattern Set -- Sorted by Pattern Number*

```
 f =    80 H =  0.000, p =  .000, H contr. =  0.000
  list          80  1.00

(109)  constructx[@]
 f =    77 H =  0.000, p =  .000, H contr. =  0.000
  temp          77  1.00

(110)  constructx[*,@]
 f =    77 H =  0.000, p =  .000, H contr. =  0.000
  list          77  1.00

(111)  length[@]
 f =    50 H =   .529, p =  .000, H contr. =   .000
  var           44   .88      dot             6   .12

(112)  row[@]
 f =    47 H =  0.000, p =  .000, H contr. =  0.000
  list          47  1.00

(113)  rowcons[@]
 f =    47 H =  0.000, p =  .000, H contr. =  0.000
  var           47  1.00

(114)  rowcons[*,@]
 f =    47 H =  0.000, p =  .000, H contr. =  0.000
  row           47  1.00

(115)  mwconst[@]
 f =    44 H =  0.000, p =  .000, H contr. =  0.000
  list          44  1.00

(116)  resume[@]
 f =    44 H =   .774, p =  .000, H contr. =   .000
    4 cases
  <empty>       38   .86      dot             2   .05
  var            3   .07      list            1   .02

(117)  relGE[@]
 f =    41 H =  1.778, p =  .000, H contr. =   .000
    6 cases
  var           23   .56      dot             3   .07      <empty>         1   .02
  assignx       10   .24      dollar          3   .07      plus            1   .02

(118)  relGE[*,@]
 f =    41 H =  2.022, p =  .000, H contr. =   .000
    7 cases
  num           18   .44      dot             2   .05      max             1   .02
  var           13   .32      assignx         1   .02
  length         5   .12      dollar          1   .02

(119)  label[@]
 f =    41 H =  1.883, p =  .000, H contr. =   .000
    7 cases
  list          25   .61      ifstmt          3   .07      assign          1   .02
  enable         5   .12      catchmark       2   .05
  casestmt       4   .10      call            1   .02

(120)  label[*,@]
 f =    41 H =   .281, p =  .000, H contr. =   .000
  item          39   .95      list            2   .05

(121)  uminus[@]
 f =    35 H =  1.391, p =  .000, H contr. =   .000
    5 cases
  var           25   .71      call            3   .09      dindex          1   .03
  dollar         4   .11      dot             2   .06

(122)  vconstruct[@]
 f =    32 H =  0.000, p =  .000, H contr. =  0.000
```

C.1. *Initial Pattern Set -- Sorted by Pattern Number*

```
 dollar        32  1.00

(123)  vconstruct[*,@]
 f =     32 H =  0.000, p =  .000, H contr. =  0.000
  list          32  1.00

(124)  relLE[@]
 f =     30 H =  1.826, p =  .000, H contr. =   .000
    8 cases
  var           20   .67     abs            2   .07     dollar         1   .03
  <empty>        2   .07     times          1   .03     index          1   .03
  assignx        2   .07     dot            1   .03

(125)  relLE[*,@]
 f =     30 H =  1.505, p =  .000, H contr. =   .000
    5 cases
  num           20   .67     dot            2   .07     dollar         1   .03
  var            5   .17     index          2   .07

(126)  enable[@]
 f =     29 H =  0.000, p =  .000, H contr. =  0.000
  catchphrase   29  1.00

(127)  enable[*,@]
 f =     29 H =   .431, p =  .000, H contr. =   .000
  list          27   .93     ifstmt         1   .03     dostmt         1   .03

(128)  mod[@]
 f =     27 H =  2.203, p =  .000, H contr. =   .000
    6 cases
  var           12   .44     assignx        3   .11     inlinecall     3   .11
  plus           5   .19     dot            3   .11     dollar         1   .04

(129)  mod[*,@]
 f =     27 H =   .979, p =  .000, H contr. =   .000
  num           20   .74     length         6   .22     var            1   .04

(130)  min[@]
 f =     27 H =  0.000, p =  .000, H contr. =  0.000
  list          27  1.00

(131)  stringinit[@]
 f =     27 H =  0.000, p =  .000, H contr. =  0.000
  num           27  1.00

(132)  max[@]
 f =     25 H =  0.000, p =  .000, H contr. =  0.000
  list          25  1.00

(133)  catchmark[@]
 f =     24 H =  1.908, p =  .000, H contr. =   .000
    5 cases
  call          10   .42     enable         6   .25     ifstmt         1   .04
  assign         6   .25     fextract       1   .04

(134)  base[@]
 f =     22 H =   .439, p =  .000, H contr. =   .000
  var           20   .91     dot            2   .09

(135)  memory[@]
 f =     13 H =  1.884, p =  .000, H contr. =   .000
    4 cases
  var            5   .38     plus           2   .15
  num            4   .31     minus          2   .15

(136)  fdollar[@]
 f =     11 H =   .845, p =  .000, H contr. =   .000
  call           8   .73     inlinecall     3   .27
```

C.1. *Initial Pattern Set -- Sorted by Pattern Number*

```
(137)  fdollar[*,@]
 f =    11 H =  0.000, p =  .000, H contr. =  0.000
  var           11  1.00

(138)  downthru[@]
 f =    11 H =  0.000, p =  .000, H contr. =  0.000
  var           11  1.00

(139)  downthru[*,@]
 f =    11 H =   .946, p =  .000, H contr. =   .000
  intCO          7   .64      intCC           4   .36

(140)  dst[@]
 f =    10 H =  0.000, p =  .000, H contr. =  0.000
  var           10  1.00

(141)  lstf[@]
 f =    10 H =  0.000, p =  .000, H contr. =  0.000
  var           10  1.00

(142)  extract[@]
 f =     9 H =  0.000, p =  .000, H contr. =  0.000
  list           9  1.00

(143)  extract[*,@]
 f =     9 H =  1.224, p =  .000, H contr. =   .000
  call           6   .67      uparrow         2   .22       dollar           1   .11

(144)  lst[@]
 f =     9 H =  0.000, p =  .000, H contr. =  0.000
  var            9  1.00

(145)  abs[@]
 f =     8 H =  1.299, p =  .000, H contr. =   .000
  var            5   .62      call            2   .25       dot              1   .12

(146)  start[@]
 f =     8 H =   .811, p =  .000, H contr. =   .000
  var            6   .75      dot             2   .25

(147)  start[*,@]
 f =     8 H =  0.000, p =  .000, H contr. =  0.000
  <empty>        8  1.00

(148)  start[*,*,@]
 f =     8 H =   .544, p =  .000, H contr. =   .000
  <empty>        7   .87      catchphrase     1   .12

(149)  intOO[@]
 f =     3 H =  0.000, p =  .000, H contr. =  0.000
  var            3  1.00

(150)  intOO[*,@]
 f =     3 H =   .918, p =  .000, H contr. =   .000
  var            2   .67      plus            1   .33

(151)  intOC[@]
 f =     3 H =  0.000, p =  .000, H contr. =  0.000
  var            3  1.00

(152)  intOC[*,@]
 f =     3 H =  0.000, p =  .000, H contr. =  0.000
  var            3  1.00

(153)  stop[@]
 f =     3 H =  0.000, p =  .000, H contr. =  0.000
  <empty>        3  1.00

(154)  svc[@]
```

C.1. *Initial Pattern Set -- Sorted by Pattern Number*

```
f =      2 H =  0.000, p =   .000, H contr. =  0.000
 num             2  1.00
```

C.2. Initial Pattern Set -- Sorted by Entropy Contribution

pattern number	*freq.*	*H*	*H contr.*	*cumul. H*	*pattern*
2.	40289	3.680	.499	.499	var[*,@]
5.	22481	3.763	.285	.784	list[*,...@]
4.	40289	2.099	.285	1.069	var[*,*,*,@]
1.	40289	1.762	.239	1.308	var[@]
6.	10220	5.132	.177	1.485	num[@]
12.	5826	3.591	.070	1.556	assign[*,@]
9.	6616	2.961	.066	1.622	call[*,@]
7.	7425	2.476	.062	1.683	list[@]
3.	40289	.336	.046	1.729	var[*,*,@]
11.	5826	1.413	.028	1.757	assign[@]
13.	5726	1.400	.027	1.784	dot[@]
17.	2770	2.051	.019	1.803	dollar[@]
22.	1810	3.059	.019	1.822	ifstmt[*,@]
21.	1810	3.047	.019	1.840	ifstmt[@]
16.	4039	1.279	.017	1.858	plus[*,@]
19.	2348	2.010	.016	1.873	relE[@]
38.	511	8.735	.015	1.888	str[@]
24.	1683	2.521	.014	1.903	return[@]
15.	4039	1.037	.014	1.917	plus[@]
29.	1214	3.315	.014	1.930	item[*,@]
23.	1810	1.329	.008	1.939	ifstmt[*,*,@]
20.	2348	.819	.006	1.945	relE[*,@]
32.	639	2.581	.006	1.951	casestmt[*,*,@]
8.	6616	.238	.005	1.956	call[@]
27.	1270	1.191	.005	1.961	uparrow[@]
34.	572	2.612	.005	1.966	casetest[*,@]
36.	538	2.411	.004	1.970	relN[@]
44.	469	2.672	.004	1.975	minus[@]
50.	348	3.349	.004	1.979	assignx[*,@]
28.	1214	.936	.004	1.982	item[@]
41.	484	2.224	.004	1.986	dostmt[*,*,@]
47.	433	2.450	.004	1.990	dindex[*,@]
35.	570	1.824	.004	1.993	addr[@]
48.	382	2.636	.003	1.996	not[@]
30.	639	1.543	.003	2.000	casestmt[@]
40.	484	1.777	.003	2.003	dostmt[*,@]
55.	262	3.019	.003	2.005	inlinecall[@]
52.	338	2.325	.003	2.008	index[*,@]
58.	251	3.113	.003	2.011	and[*,@]
39.	484	1.610	.003	2.013	dostmt[@]
57.	251	3.104	.003	2.016	and[@]
45.	469	1.590	.003	2.018	minus[*,@]
10.	6616	.112	.002	2.021	call[*,*,@]
61.	211	3.005	.002	2.023	ifexp[*,*,@]
37.	538	1.174	.002	2.025	relN[*,@]
53.	317	1.702	.002	2.027	times[@]
60.	211	2.433	.002	2.029	ifexp[*,@]
59.	211	2.315	.002	2.030	ifexp[@]
49.	348	1.308	.002	2.032	assignx[@]
83.	127	2.955	.001	2.033	or[@]
84.	127	2.856	.001	2.034	or[*,@]
33.	572	.597	.001	2.035	casetest[@]
75.	159	2.109	.001	2.037	relG[@]
69.	183	1.817	.001	2.038	construct[@]
51.	338	.965	.001	2.039	index[@]
65.	186	1.602	.001	2.040	signal[*,@]
68.	183	1.466	.001	2.041	intCC[*,@]
81.	129	1.903	.001	2.042	error[*,@]
89.	94	2.573	.001	2.042	intCO[*,@]
56.	262	.843	.001	2.043	inlinecall[*,@]
76.	159	1.386	.001	2.044	relG[*,@]
101.	84	2.496	.001	2.045	forseq[*,@]
74.	170	1.220	.001	2.045	upthru[*,@]

C.2. *Initial Pattern Set -- Sorted by Entropy Contribution*

pattern number	*freq.*	*H*	*H contr.*	*cumul. H*	*pattern*
90.	93	2.197	.001	2.046	relL[@]
91.	93	2.141	.001	2.047	relL[*,@]
94.	87	2.183	.001	2.047	div[@]
54.	317	.535	.001	2.048	times[*,@]
77.	159	.984	.001	2.048	lbl[@]
86.	102	1.492	.001	2.049	in[@]
102.	84	1.745	.000	2.049	forseq[*,*,@]
104.	82	1.736	.000	2.050	seqindex[*,@]
43.	484	.292	.000	2.050	dostmt[*,*,*,*,@]
78.	130	1.040	.000	2.051	catchphrase[@]
79.	130	1.024	.000	2.051	catchphrase[*,@]
63.	196	.672	.000	2.052	fextract[*,@]
73.	170	.767	.000	2.052	upthru[@]
97.	84	1.343	.000	2.053	caseexp[@]
64.	186	.524	.000	2.053	signal[@]
93.	91	1.052	.000	2.053	openstmt[*,@]
105.	81	1.175	.000	2.053	caseswitch[@]
67.	183	.483	.000	2.054	intCC[@]
46.	433	.195	.000	2.054	dindex[@]
118.	41	2.022	.000	2.054	relGE[*,@]
88.	94	.849	.000	2.055	intCO[@]
119.	41	1.883	.000	2.055	label[@]
99.	84	.885	.000	2.055	caseexp[*,*,@]
117.	41	1.778	.000	2.055	relGE[@]
42.	484	.133	.000	2.056	dostmt[*,*,*,@]
128.	27	2.203	.000	2.056	mod[@]
124.	30	1.826	.000	2.056	relLE[@]
95.	87	.572	.000	2.056	div[*,@]
121.	35	1.391	.000	2.056	uminus[@]
103.	82	.592	.000	2.056	seqindex[@]
133.	24	1.908	.000	2.057	catchmark[@]
125.	30	1.505	.000	2.057	relLE[*,@]
116.	44	.774	.000	2.057	resume[@]
87.	102	.323	.000	2.057	in[*,@]
111.	50	.529	.000	2.057	length[@]
129.	27	.979	.000	2.057	mod[*,@]
135.	13	1.884	.000	2.057	memory[@]
127.	29	.431	.000	2.057	enable[*,@]
120.	41	.281	.000	2.057	label[*,@]
143.	9	1.224	.000	2.057	extract[*,@]
139.	11	.946	.000	2.057	downthru[*,@]
145.	8	1.299	.000	2.057	abs[@]
134.	22	.439	.000	2.057	base[@]
136.	11	.845	.000	2.058	fdollar[@]
80.	129	.065	.000	2.058	error[@]
82.	129	.065	.000	2.058	error[*,*,@]
146.	8	.811	.000	2.058	start[@]
148.	8	.544	.000	2.058	start[*,*,@]
150.	3	.918	.000	2.058	intOO[*,@]
14.	5726	0.000	0.000	2.058	dot[*,@]
18.	2770	0.000	0.000	2.058	dollar[*,@]
25.	1513	0.000	0.000	2.058	@
26.	1513	0.000	0.000	2.058	body[@]
31.	639	0.000	0.000	2.058	casestmt[*,@]
62.	196	0.000	0.000	2.058	fextract[@]
66.	186	0.000	0.000	2.058	signal[*,*,@]
70.	183	0.000	0.000	2.058	construct[*,@]
71.	179	0.000	0.000	2.058	unionx[@]
72.	179	0.000	0.000	2.058	unionx[*,@]
85.	105	0.000	0.000	2.058	goto[@]
92.	91	0.000	0.000	2.058	openstmt[@]
96.	87	0.000	0.000	2.058	register[@]
98.	84	0.000	0.000	2.058	caseexp[*,@]
100.	84	0.000	0.000	2.058	forseq[@]
106.	81	0.000	0.000	2.058	caseswitch[*,@]

C.2. *Initial Pattern Set -- Sorted by Entropy Contribution*

pattern number	*freq.*	*H*	*H contr.*	*cumul. H*	*pattern*
107.	81	0.000	0.000	2.058	caseswitch[*,*,@]
108.	80	0.000	0.000	2.058	arraydesc[@]
109.	77	0.000	0.000	2.058	constructx[@]
110.	77	0.000	0.000	2.058	constructx[*,@]
112.	47	0.000	0.000	2.058	row[@]
113.	47	0.000	0.000	2.058	rowcons[@]
114.	47	0.000	0.000	2.058	rowcons[*,@]
115.	44	0.000	0.000	2.058	mwconst[@]
122.	32	0.000	0.000	2.058	vconstruct[@]
123.	32	0.000	0.000	2.058	vconstruct[*,@]
126.	29	0.000	0.000	2.058	enable[@]
130.	27	0.000	0.000	2.058	min[@]
131.	27	0.000	0.000	2.058	stringinit[@]
132.	25	0.000	0.000	2.058	max[@]
137.	11	0.000	0.000	2.058	fdollar[*,@]
138.	11	0.000	0.000	2.058	downthru[@]
140.	10	0.000	0.000	2.058	dst[@]
141.	10	0.000	0.000	2.058	lstf[@]
142.	9	0.000	0.000	2.058	extract[@]
144.	9	0.000	0.000	2.058	lst[@]
147.	8	0.000	0.000	2.058	start[*,@]
149.	3	0.000	0.000	2.058	intOO[@]
151.	3	0.000	0.000	2.058	intOC[@]
152.	3	0.000	0.000	2.058	intOC[*,@]
153.	3	0.000	0.000	2.058	stop[@]
154.	2	0.000	0.000	2.058	svc[@]

C.3. Initial Pattern Set -- Alphabetical by Pattern

pattern number	*freq.*	*H*	*H contr.*	*pattern*
25.	1513	0.000	0.000	@
145.	8	1.299	.000	abs[@]
35.	570	1.824	.004	addr[@]
57.	251	3.104	.003	and[@]
58.	251	3.113	.003	and[*,@]
108.	80	0.000	0.000	arraydesc[@]
11.	5826	1.413	.028	assign[@]
12.	5826	3.591	.070	assign[*,@]
49.	348	1.308	.002	assignx[@]
50.	348	3.349	.004	assignx[*,@]
134.	22	.439	.000	base[@]
26.	1513	0.000	0.000	body[@]
8.	6616	.238	.005	call[@]
9.	6616	2.961	.066	call[*,@]
10.	6616	.112	.002	call[*,*,@]
97.	84	1.343	.000	caseexp[@]
98.	84	0.000	0.000	caseexp[*,@]
99.	84	.885	.000	caseexp[*,*,@]
30.	639	1.543	.003	casestmt[@]
31.	639	0.000	0.000	casestmt[*,@]
32.	639	2.581	.006	casestmt[*,*,@]
105.	81	1.175	.000	caseswitch[@]
106.	81	0.000	0.000	caseswitch[*,@]
107.	81	0.000	0.000	caseswitch[*,*,@]
33.	572	.597	.001	casetest[@]
34.	572	2.612	.005	casetest[*,@]
133.	24	1.908	.000	catchmark[@]
78.	130	1.040	.000	catchphrase[@]
79.	130	1.024	.000	catchphrase[*,@]
69.	183	1.817	.001	construct[@]
70.	183	0.000	0.000	construct[*,@]
109.	77	0.000	0.000	constructx[@]
110.	77	0.000	0.000	constructx[*,@]
46.	433	.195	.000	dindex[@]
47.	433	2.450	.004	dindex[*,@]
94.	87	2.183	.001	div[@]
95.	87	.572	.000	div[*,@]
17.	2770	2.051	.019	dollar[@]
18.	2770	0.000	0.000	dollar[*,@]
39.	484	1.610	.003	dostmt[@]
40.	484	1.777	.003	dostmt[*,@]
41.	484	2.224	.004	dostmt[*,*,@]
42.	484	.133	.000	dostmt[*,*,*,@]
43.	484	.292	.000	dostmt[*,*,*,*,@]
13.	5726	1.400	.027	dot[@]
14.	5726	0.000	0.000	dot[*,@]
138.	11	0.000	0.000	downthru[@]
139.	11	.946	.000	downthru[*,@]
140.	10	0.000	0.000	dst[@]
126.	29	0.000	0.000	enable[@]
127.	29	.431	.000	enable[*,@]
80.	129	.065	.000	error[@]
81.	129	1.903	.001	error[*,@]
82.	129	.065	.000	error[*,*,@]
142.	9	0.000	0.000	extract[@]
143.	9	1.224	.000	extract[*,@]
136.	11	.845	.000	fdollar[@]
137.	11	0.000	0.000	fdollar[*,@]
62.	196	0.000	0.000	fextract[@]
63.	196	.672	.000	fextract[*,@]
100.	84	0.000	0.000	forseq[@]
101.	84	2.496	.001	forseq[*,@]
102.	84	1.745	.000	forseq[*,*,@]

C.3. *Initial Pattern Set -- Alphabetical by Pattern*

pattern number	*freq.*	*H*	*H contr.*	*pattern*
85.	105	0.000	0.000	goto[@]
59.	211	2.315	.002	ifexp[@]
60.	211	2.433	.002	ifexp[*,@]
61.	211	3.005	.002	ifexp[*,*,@]
21.	1810	3.047	.019	ifstmt[@]
22.	1810	3.059	.019	ifstmt[*,@]
23.	1810	1.329	.008	ifstmt[*,*,@]
86.	102	1.492	.001	in[@]
87.	102	.323	.000	in[*,@]
51.	338	.965	.001	index[@]
52.	338	2.325	.003	index[*,@]
55.	262	3.019	.003	inlinecall[@]
56.	262	.843	.001	inlinecall[*,@]
67.	183	.483	.000	intCC[@]
68.	183	1.466	.001	intCC[*,@]
88.	94	.849	.000	intCO[@]
89.	94	2.573	.001	intCO[*,@]
151.	3	0.000	0.000	intOC[@]
152.	3	0.000	0.000	intOC[*,@]
149.	3	0.000	0.000	intOO[@]
150.	3	.918	.000	intOO[*,@]
28.	1214	.936	.004	item[@]
29.	1214	3.315	.014	item[*,@]
119.	41	1.883	.000	label[@]
120.	41	.281	.000	label[*,@]
77.	159	.984	.001	lbl[@]
111.	50	.529	.000	length[@]
7.	7425	2.476	.062	list[@]
5.	22481	3.763	.285	list[*,...@]
144.	9	0.000	0.000	lst[@]
141.	10	0.000	0.000	lstf[@]
132.	25	0.000	0.000	max[@]
135.	13	1.884	.000	memory[@]
130.	27	0.000	0.000	min[@]
44.	469	2.672	.004	minus[@]
45.	469	1.590	.003	minus[*,@]
128.	27	2.203	.000	mod[@]
129.	27	.979	.000	mod[*,@]
115.	44	0.000	0.000	mwconst[@]
48.	382	2.636	.003	not[@]
6.	10220	5.132	.177	num[@]
92.	91	0.000	0.000	openstmt[@]
93.	91	1.052	.000	openstmt[*,@]
83.	127	2.955	.001	or[@]
84.	127	2.856	.001	or[*,@]
15.	4039	1.037	.014	plus[@]
16.	4039	1.279	.017	plus[*,@]
96.	87	0.000	0.000	register[@]
19.	2348	2.010	.016	relE[@]
20.	2348	.819	.006	relE[*,@]
117.	41	1.778	.000	relGE[@]
118.	41	2.022	.000	relGE[*,@]
75.	159	2.109	.001	relG[@]
76.	159	1.386	.001	relG[*,@]
124.	30	1.826	.000	relLE[@]
125.	30	1.505	.000	relLE[*,@]
90.	93	2.197	.001	relL[@]
91.	93	2.141	.001	relL[*,@]
36.	538	2.411	.004	relN[@]
37.	538	1.174	.002	relN[*,@]
116.	44	.774	.000	resume[@]
24.	1683	2.521	.014	return[@]
112.	47	0.000	0.000	row[@]
113.	47	0.000	0.000	rowcons[@]
114.	47	0.000	0.000	rowcons[*,@]

C.3. *Initial Pattern Set -- Alphabetical by Pattern*

pattern number	*freq.*	*H*	*H contr.*	*pattern*
103.	82	.592	.000	seqindex[@]
104.	82	1.736	.000	seqindex[*,@]
64.	186	.524	.000	signal[@]
65.	186	1.602	.001	signal[*,@]
66.	186	0.000	0.000	signal[*,*,@]
146.	8	.811	.000	start[@]
147.	8	0.000	0.000	start[*,@]
148.	8	.544	.000	start[*,*,@]
153.	3	0.000	0.000	stop[@]
38.	511	8.735	.015	str[@]
131.	27	0.000	0.000	stringinit[@]
154.	2	0.000	0.000	svc[@]
53.	317	1.702	.002	times[@]
54.	317	.535	.001	times[*,@]
121.	35	1.391	.000	uminus[@]
71.	179	0.000	0.000	unionx[@]
72.	179	0.000	0.000	unionx[*,@]
27.	1270	1.191	.005	uparrow[@]
73.	170	.767	.000	upthru[@]
74.	170	1.220	.001	upthru[*,@]
1.	40289	1.762	.239	var[@]
2.	40289	3.680	.499	var[*,@]
3.	40289	.336	.046	var[*,*,@]
4.	40289	2.099	.285	var[*,*,*,@]
122.	32	0.000	0.000	vconstruct[@]
123.	32	0.000	0.000	vconstruct[*,@]

C.4. Final Pattern Set -- Sorted by Entropy Contribution

In the refined pattern set, some patterns are defined in terms of those larger patterns "removed" from them. These patterns are denoted by an asterisk (*) in the "was refined" column. See the listing sorted by pattern number to find the set of patterns refining the given pattern.

pattern number	*was refined*	*freq.*	*H*	*H contr.*	*cumul. H*	*pattern*
6.	*	4850	5.828	.096	.096	num[@]
12.		5663	3.574	.069	.164	assign[*,@]
9.		6616	2.961	.066	.231	call[*,@]
155.	*	5332	3.411	.062	.292	var[global,@]
245.		6617	2.639	.059	.352	body[list[*,...@]]
214.		4016	4.241	.058	.409	call[var[global,@]]
243.		4570	3.441	.053	.463	dot[*,var[field,@]]
160.	*	6685	2.327	.053	.515	var[local,*,*,@]
159.		12217	1.114	.046	.561	var[global,*,*,@]
194.		2610	4.744	.042	.603	{relL|relLE|relE|relN|relGE|relG}[*,num[@]]
1.	*	10800	1.141	.042	.645	var[@]
267.		4570	2.548	.039	.685	dot[*,var[field,*,*,@]]
158.		2686	4.039	.037	.721	var[entry,@]
246.		4263	2.425	.035	.756	call[*,list[*,...@]]
196.		3296	2.995	.033	.790	{assign|assignx}[var[local,@]]
5.	*	3096	3.130	.033	.823	list[*,...@]
7.	*	2874	3.163	.031	.853	list[@]
268.		2709	3.287	.030	.884	dollar[*,var[field,*,*,@]]
11.		5663	1.423	.027	.911	assign[@]
13.		5697	1.401	.027	.938	dot[@]
156.	*	2335	3.153	.025	.963	var[local,@]
242.		2709	2.433	.022	.985	dollar[*,var[field,@]]
229.		3114	2.078	.022	1.007	assign[var[local,*,*,@]]
171.		6370	.992	.021	1.029	call[var[@]]
17.		2754	2.050	.019	1.048	dollar[@]
22.		1810	3.093	.019	1.067	ifstmt[*,@]
21.		1810	3.047	.019	1.085	ifstmt[@]
205.		3027	1.795	.018	1.104	plus[*,var[local,@]]
201.		1909	2.798	.018	1.122	list[*,...var[local,@]]
174.		5697	.855	.017	1.138	dot[*,var[@]]
19.		2348	2.023	.016	1.155	relE[@]
164.		2754	1.713	.016	1.171	dollar[*,var[*,*,@]]
216.		944	4.912	.016	1.186	dot[*,var[global,@]]
16.		3847	1.180	.015	1.202	plus[*,@]
38.		511	8.735	.015	1.217	str[@]
24.		1683	2.521	.014	1.231	return[@]
29.		1214	3.315	.014	1.245	item[*,@]
237.		1909	2.104	.014	1.258	list[*,...var[local,*,*,@]]
15.		3847	1.032	.013	1.272	plus[@]
163.		5697	.657	.013	1.285	dot[*,var[*,*,@]]
213.		752	4.743	.012	1.297	assign[var[global,@]]
259.		1355	2.625	.012	1.309	item[*,list[*,...@]]
186.		1078	3.056	.011	1.320	assign[*,num[@]]
198.		1445	2.275	.011	1.331	call[*,var[local,@]]
167.		3885	.752	.010	1.341	assign[var[@]]
173.		2941	.939	.009	1.350	dot[var[@]]
199.		1929	1.384	.009	1.359	dot[var[local,@]]
23.		1810	1.333	.008	1.367	ifstmt[*,*,@]
215.		1009	2.251	.008	1.375	dot[var[global,@]]
209.		853	2.607	.008	1.383	{relL|relLE|relE|relN|relGE|relG}[var[local,@]]
232.		1445	1.463	.007	1.390	call[*,var[local,*,*,@]]
255.		828	2.427	.007	1.397	dostmt[*,*,list[*,...@]]
254.		670	2.948	.007	1.403	{construct|constructx}[*,list[*,...@]]
219.		1816	1.085	.007	1.410	call[*,list[@]]
20.		2348	.819	.007	1.417	relE[*,@]

C.4. *Final Pattern Set -- Sorted by Entropy Contribution*

pattern number	*was refined*	*freq.*	*H*	*H contr.*	*cumul. H*	*pattern*
233.		840	2.252	.006	1.423	dollar[var[local,*,*,@]]
197.		610	2.949	.006	1.429	{assign|assignx}[*,var[local,@]]
32.		639	2.590	.006	1.435	casestmt[*,*,@]
8.		6616	.238	.005	1.440	call[@]
34.		572	2.612	.005	1.445	casetest[*,@]
27.		1260	1.184	.005	1.450	uparrow[@]
257.		489	3.016	.005	1.455	inlinecall[*,list[*,...@]]
172.		1739	.806	.005	1.460	call[*,var[@]]
36.		538	2.411	.004	1.464	relN[@]
223.		598	2.132	.004	1.469	ifstmt[*,list[@]]
44.		469	2.672	.004	1.473	minus[@]
230.		590	2.026	.004	1.477	assign[*,var[local,*,*,@]]
28.		1214	.936	.004	1.481	item[@]
252.		600	1.863	.004	1.485	casetest[*,list[*,...@]]
41.		484	2.269	.004	1.488	dostmt[*,*,@]
50.		323	3.313	.004	1.492	assignx[*,@]
47.		433	2.450	.004	1.495	dindex[*,@]
35.		570	1.824	.004	1.499	addr[@]
48.		382	2.636	.003	1.502	not[@]
184.		757	1.317	.003	1.506	{index|dindex|seqindex}[var[@]]
30.		639	1.543	.003	1.509	casestmt[@]
207.		264	3.453	.003	1.512	{index|dindex|seqindex}[var[local,@]]
227.		425	2.096	.003	1.515	item[*,list[@]]
191.		316	2.807	.003	1.518	plus[*,num[@]]
204.		321	2.684	.003	1.521	plus[var[local,@]]
40.		484	1.777	.003	1.524	dostmt[*,@]
262.		391	2.113	.003	1.527	openstmt[*,list[*,...@]]
55.		262	3.019	.003	1.530	inlinecall[@]
52.		338	2.325	.003	1.532	index[*,@]
58.		251	3.113	.003	1.535	and[*,@]
39.		484	1.610	.003	1.537	dostmt[@]
57.		251	3.104	.003	1.540	and[@]
190.		318	2.446	.003	1.543	minus[*,num[@]]
220.		639	1.208	.003	1.545	casestmt[*,list[@]]
161.	*	428	1.749	.003	1.548	var[field,*,*,@]
45.		469	1.590	.003	1.550	minus[*,@]
168.		775	.959	.003	1.553	assign[*,var[@]]
10.		6616	.112	.003	1.555	call[*,*,@]
192.		292	2.387	.002	1.558	times[*,num[@]]
175.		964	.682	.002	1.560	dollar[var[@]]
217.		164	3.938	.002	1.562	signal[var[global,@]]
157.	*	428	1.491	.002	1.564	var[field,@]
61.		211	3.005	.002	1.567	ifexp[*,*,@]
37.		538	1.174	.002	1.569	relN[*,@]
264.		686	.882	.002	1.571	row[list[*,...@]]
208.		245	2.364	.002	1.573	{index|dindex|seqindex}[*,var[local,@]]
210.		191	3.006	.002	1.575	{relL|relLE|relE|relN|relGE|relG}[*,var[local,@
221.		253	2.208	.002	1.577	dostmt[*,*,list[@]]
202.		199	2.803	.002	1.578	minus[var[local,@]]
53.		317	1.702	.002	1.580	times[@]
179.		955	.549	.002	1.582	{relE|relG|relN|relL|relGE|relLE}[var[@]]
60.		211	2.433	.002	1.584	ifexp[*,@]
165.		179	2.818	.002	1.586	unionx[var[*,*,@]]
59.		211	2.315	.002	1.587	ifexp[@]
49.		323	1.353	.001	1.589	assignx[@]
260.		187	2.311	.001	1.590	label[list[*,...@]]
188.		168	2.569	.001	1.592	intCC[num[@]]
206.		162	2.598	.001	1.593	times[var[local,@]]
244.		160	2.536	.001	1.594	arraydesc[list[*,...@]]
180.		403	.998	.001	1.596	{relE|relG|relN|relL|relGE|relLE}[*,var[@]]
211.		355	1.130	.001	1.597	uparrow[var[local,@]]
83.		127	2.955	.001	1.598	or[@]
84.		127	2.856	.001	1.600	or[*,@]
231.		182	1.963	.001	1.601	assignx[var[local,*,*,@]]
248.		982	.363	.001	1.602	casestmt[*,list[*,...@]]

C.4. *Final Pattern Set -- Sorted by Entropy Contribution*

pattern number	*was refined*	*freq.*	*H*	*H contr.*	*cumul. H*	*pattern*
224.		173	2.051	.001	1.603	ifstmt[*,*,list[@]]
249.		159	2.228	.001	1.604	casestmt[*,*,list[*,...@]]
176.		2754	.126	.001	1.606	dollar[*,var[@]]
33.		572	.597	.001	1.607	casetest[@]
75.		159	2.138	.001	1.608	relG[@]
69.		183	1.817	.001	1.609	construct[@]
195.		148	2.227	.001	1.610	return[num[@]]
51.		338	.965	.001	1.611	index[@]
166.		339	.958	.001	1.612	addr[var[@]]
65.		186	1.602	.001	1.613	signal[*,@]
68.		183	1.466	.001	1.614	intCC[*,@]
263.		160	1.636	.001	1.615	return[list[*,...@]]
212.		129	2.023	.001	1.616	upthru[var[local,@]]
81.		129	1.903	.001	1.617	error[*,@]
89.		94	2.573	.001	1.618	intCO[*,@]
256.		354	.663	.001	1.618	fextract[list[*,...@]]
203.		90	2.595	.001	1.619	minus[*,var[local,@]]
181.		337	.692	.001	1.620	return[var[@]]
56.		262	.843	.001	1.621	inlinecall[*,@]
76.		159	1.386	.001	1.622	relG[*,@]
222.		196	1.099	.001	1.622	fextract[list[@]]
101.		84	2.496	.001	1.623	forseq[*,@]
74.		170	1.220	.001	1.624	upthru[*,@]
90.		93	2.197	.001	1.624	relL[@]
91.		93	2.141	.001	1.625	relL[*,@]
193.		87	2.221	.001	1.626	register[num[@]]
94.		87	2.183	.001	1.626	div[@]
241.		129	1.400	.001	1.627	upthru[var[local,*,*,@]]
185.		289	.615	.001	1.628	{index|dindex|seqindex}[*,var[@]]
54.		317	.535	.001	1.628	times[*,@]
269.		163	1.027	.001	1.629	bump[@]
77.		159	.984	.001	1.629	lbl[@]
169.		223	.688	.001	1.630	assignx[var[@]]
86.		102	1.492	.001	1.630	in[@]
265.		88	1.680	.001	1.631	signal[*,list[*,...@]]
102.		84	1.745	.000	1.631	forseq[*,*,@]
104.		82	1.736	.000	1.632	seqindex[*,@]
43.		484	.292	.000	1.632	dostmt[*,*,*,*,@]
187.		85	1.606	.000	1.633	index[*,num[@]]
78.		130	1.040	.000	1.633	catchphrase[@]
79.		130	1.024	.000	1.634	catchphrase[*,@]
63.		196	.672	.000	1.634	fextract[*,@]
73.		170	.767	.000	1.635	upthru[@]
225.		230	.549	.000	1.635	inlinecall[*,list[@]]
226.		97	1.190	.000	1.635	item[list[@]]
97.		84	1.343	.000	1.636	caseexp[@]
258.		243	.422	.000	1.636	item[list[*,...@]]
182.		372	.268	.000	1.636	uparrow[var[@]]
64.		186	.524	.000	1.637	signal[@]
93.		91	1.052	.000	1.637	openstmt[*,@]
105.		81	1.175	.000	1.637	caseswitch[@]
67.		183	.483	.000	1.638	intCC[@]
46.		433	.195	.000	1.638	dindex[@]
118.		41	2.022	.000	1.638	relGE[*,@]
200.		84	.963	.000	1.639	forseq[var[local,@]]
117.		41	1.954	.000	1.639	relGE[@]
88.		94	.849	.000	1.639	intCO[@]
119.		41	1.883	.000	1.639	label[@]
99.		84	.885	.000	1.640	caseexp[*,*,@]
42.		484	.133	.000	1.640	dostmt[*,*,*,@]
128.		27	2.203	.000	1.640	mod[@]
247.		120	.495	.000	1.640	caseexp[*,list[*,...@]]
124.		30	1.826	.000	1.640	relLE[@]
95.		87	.572	.000	1.641	div[*,@]
121.		35	1.391	.000	1.641	uminus[@]

C.4. *Final Pattern Set -- Sorted by Entropy Contribution*

pattern number	*was refined*	*freq.*	*H*	*H contr.*	*cumul. H*	*pattern*
103.		82	.592	.000	1.641	seqindex[@]
133.		24	1.908	.000	1.641	catchmark[@]
125.		30	1.505	.000	1.641	relLE[*,@]
189.		78	.477	.000	1.641	intCO[num[@]]
170.		30	1.159	.000	1.641	assignx[*,var[@]]
116.		44	.774	.000	1.642	resume[@]
87.		102	.323	.000	1.642	in[*,@]
111.		50	.529	.000	1.642	length[@]
129.		27	.979	.000	1.642	mod[*,@]
135.		13	1.884	.000	1.642	memory[@]
183.		132	.156	.000	1.642	upthru[var[@]]
127.		29	.431	.000	1.642	enable[*,@]
239.		50	.242	.000	1.642	seqindex[*,var[local,*,*,@]]
270.		25	.482	.000	1.642	bumpx[@]
120.		41	.281	.000	1.642	label[*,@]
143.		9	1.224	.000	1.642	extract[*,@]
139.		11	.946	.000	1.642	downthru[*,@]
145.		8	1.299	.000	1.642	abs[@]
134.		22	.439	.000	1.642	base[@]
136.		11	.845	.000	1.642	fdollar[@]
80.		129	.065	.000	1.642	error[@]
82.		129	.065	.000	1.642	error[*,*,@]
146.		8	.811	.000	1.642	start[@]
148.		8	.544	.000	1.642	start[*,*,@]
150.		3	.918	.000	1.642	intOO[*,@]
154.		2	0.000	0.000	1.642	svc[@]
153.		3	0.000	0.000	1.642	stop[@]
152.		3	0.000	0.000	1.642	intOC[*,@]
151.		3	0.000	0.000	1.642	intOC[@]
149.		3	0.000	0.000	1.642	intOO[@]
147.		8	0.000	0.000	1.642	start[*,@]
3.	*	31424	0.000	0.000	1.642	var[*,*,@]
14.		5697	0.000	0.000	1.642	dot[*,@]
18.		2754	0.000	0.000	1.642	dollar[*,@]
25.		1513	0.000	0.000	1.642	@
26.		1513	0.000	0.000	1.642	body[@]
31.		639	0.000	0.000	1.642	casestmt[*,@]
62.		196	0.000	0.000	1.642	fextract[@]
66.		186	0.000	0.000	1.642	signal[*,*,@]
70.		183	0.000	0.000	1.642	construct[*,@]
71.		179	0.000	0.000	1.642	unionx[@]
72.		179	0.000	0.000	1.642	unionx[*,@]
85.		105	0.000	0.000	1.642	goto[@]
92.		91	0.000	0.000	1.642	openstmt[@]
96.		87	0.000	0.000	1.642	register[@]
98.		84	0.000	0.000	1.642	caseexp[*,@]
100.		84	0.000	0.000	1.642	forseq[@]
106.		81	0.000	0.000	1.642	caseswitch[*,@]
107.		81	0.000	0.000	1.642	caseswitch[*,*,@]
108.		80	0.000	0.000	1.642	arraydesc[@]
109.		77	0.000	0.000	1.642	constructx[@]
110.		77	0.000	0.000	1.642	constructx[*,@]
112.		47	0.000	0.000	1.642	row[@]
113.		47	0.000	0.000	1.642	rowcons[@]
114.		47	0.000	0.000	1.642	rowcons[*,@]
115.		44	0.000	0.000	1.642	mwconst[@]
122.		32	0.000	0.000	1.642	vconstruct[@]
123.		32	0.000	0.000	1.642	vconstruct[*,@]
126.		29	0.000	0.000	1.642	enable[@]
130.		27	0.000	0.000	1.642	min[@]
131.		27	0.000	0.000	1.642	stringinit[@]
132.		25	0.000	0.000	1.642	max[@]
137.		11	0.000	0.000	1.642	fdollar[*,@]
138.		11	0.000	0.000	1.642	downthru[@]
140.		10	0.000	0.000	1.642	dst[@]

C.4. *Final Pattern Set -- Sorted by Entropy Contribution*

pattern number	*was refined*	*freq.*	*H*	*H contr.*	*cumul. H*	*pattern*
141.		10	0.000	0.000	1.642	lstf[@]
142.		9	0.000	0.000	1.642	extract[@]
144.		9	0.000	0.000	1.642	lst[@]
162.		2686	0.000	0.000	1.642	var[entry,*,*,@]
177.		128	0.000	0.000	1.642	error[var[@]]
178.		164	0.000	0.000	1.642	signal[var[@]]
218.		80	0.000	0.000	1.642	arraydesc[list[@]]
228.		44	0.000	0.000	1.642	signal[*,list[@]]
234.		1929	0.000	0.000	1.642	dot[var[local,*,*,@]]
235.		45	0.000	0.000	1.642	error[*,var[local,*,*,@]]
236.		113	0.000	0.000	1.642	ifstmt[var[local,*,*,@]]
238.		58	0.000	0.000	1.642	seqindex[var[local,*,*,@]]
240.		355	0.000	0.000	1.642	uparrow[var[local,*,*,@]]
250.		572	0.000	0.000	1.642	caseswitch[*,*,list[*,...@]]
251.		254	0.000	0.000	1.642	casetest[list[*,...@]]
253.		42	0.000	0.000	1.642	catchphrase[list[*,...@]]
261.		133	0.000	0.000	1.642	mwconst[list[*,...@]]
266.		32	0.000	0.000	1.642	vconstruct[*,list[*,...@]]

C.5. Final Pattern Set -- Alphabetical by Pattern

Pattern that have been refined are denoted by an asterisk (*) in the "was refined" column. See Section C.5 to find the set of patterns refining the given pattern.

pattern number	*was refined*	*freq.*	*H*	*H contr.*	*pattern*
25.		1513	0.000	0.000	@
145.		8	1.299	.000	abs[@]
35.		570	1.824	.004	addr[@]
166.		339	.958	.001	addr[var[@]]
57.		251	3.104	.003	and[@]
58.		251	3.113	.003	and[*,@]
108.		80	0.000	0.000	arraydesc[@]
218.		80	0.000	0.000	arraydesc[list[@]]
244.		160	2.536	.001	arraydesc[list[*,...@]]
11.		5663	1.423	.027	assign[@]
12.		5663	3.574	.069	assign[*,@]
186.		1078	3.056	.011	assign[*,num[@]]
168.		775	.959	.003	assign[*,var[@]]
230.		590	2.026	.004	assign[*,var[local,*,*,@]]
167.		3885	.752	.010	assign[var[@]]
213.		752	4.743	.012	assign[var[global,@]]
229.		3114	2.078	.022	assign[var[local,*,*,@]]
49.		323	1.353	.001	assignx[@]
50.		323	3.313	.004	assignx[*,@]
170.		30	1.159	.000	assignx[*,var[@]]
169.		223	.688	.001	assignx[var[@]]
231.		182	1.963	.001	assignx[var[local,*,*,@]]
134.		22	.439	.000	base[@]
26.		1513	0.000	0.000	body[@]
245.		6617	2.639	.059	body[list[*,...@]]
269.		163	1.027	.001	bump[@]
270.		25	.482	.000	bumpx[@]
8.		6616	.238	.005	call[@]
9.		6616	2.961	.066	call[*,@]
10.		6616	.112	.003	call[*,*,@]
219.		1816	1.085	.007	call[*,list[@]]
246.		4263	2.425	.035	call[*,list[*,...@]]
172.		1739	.806	.005	call[*,var[@]]
198.		1445	2.275	.011	call[*,var[local,@]]
232.		1445	1.463	.007	call[*,var[local,*,*,@]]
171.		6370	.992	.021	call[var[@]]
214.		4016	4.241	.058	call[var[global,@]]
97.		84	1.343	.000	caseexp[@]
98.		84	0.000	0.000	caseexp[*,@]
99.		84	.885	.000	caseexp[*,*,@]
247.		120	.495	.000	caseexp[*,list[*,...@]]
30.		639	1.543	.003	casestmt[@]
31.		639	0.000	0.000	casestmt[*,@]
32.		639	2.590	.006	casestmt[*,*,@]
249.		159	2.228	.001	casestmt[*,*,list[*,...@]]
220.		639	1.208	.003	casestmt[*,list[@]]
248.		982	.363	.001	casestmt[*,list[*,...@]]
105.		81	1.175	.000	caseswitch[@]
106.		81	0.000	0.000	caseswitch[*,@]
107.		81	0.000	0.000	caseswitch[*,*,@]
250.		572	0.000	0.000	caseswitch[*,*,list[*,...@]]
33.		572	.597	.001	casetest[@]
34.		572	2.612	.005	casetest[*,@]
252.		600	1.863	.004	casetest[*,list[*,...@]]
251.		254	0.000	0.000	casetest[list[*,...@]]
133.		24	1.908	.000	catchmark[@]
78.		130	1.040	.000	catchphrase[@]
79.		130	1.024	.000	catchphrase[*,@]
253.		42	0.000	0.000	catchphrase[list[*,...@]]

C.5. *Final Pattern Set -- Alphabetical by Pattern*

pattern number	*was refined*	*freq.*	*H*	*H contr.*	*pattern*
69.		183	1.817	.001	construct[@]
70.		183	0.000	0.000	construct[*,@]
109.		77	0.000	0.000	constructx[@]
110.		77	0.000	0.000	constructx[*,@]
46.		433	.195	.000	dindex[@]
47.		433	2.450	.004	dindex[*,@]
94.		87	2.183	.001	div[@]
95.		87	.572	.000	div[*,@]
17.		2754	2.050	.019	dollar[@]
18.		2754	0.000	0.000	dollar[*,@]
176.		2754	.126	.001	dollar[*,var[@]]
164.		2754	1.713	.016	dollar[*,var[*,*,@]]
242.		2709	2.433	.022	dollar[*,var[field,@]]
268.		2709	3.287	.030	dollar[*,var[field,*,*,@]]
175.		964	.682	.002	dollar[var[@]]
233.		840	2.252	.006	dollar[var[local,*,*,@]]
39.		484	1.610	.003	dostmt[@]
40.		484	1.777	.003	dostmt[*,@]
41.		484	2.269	.004	dostmt[*,*,@]
42.		484	.133	.000	dostmt[*,*,*,@]
43.		484	.292	.000	dostmt[*,*,*,*,@]
221.		253	2.208	.002	dostmt[*,*,list[@]]
255.		828	2.427	.007	dostmt[*,*,list[*,...@]]
13.		5697	1.401	.027	dot[@]
14.		5697	0.000	0.000	dot[*,@]
174.		5697	.855	.017	dot[*,var[@]]
163.		5697	.657	.013	dot[*,var[*,*,@]]
243.		4570	3.441	.053	dot[*,var[field,@]]
267.		4570	2.548	.039	dot[*,var[field,*,*,@]]
216.		944	4.912	.016	dot[*,var[global,@]]
173.		2941	.939	.009	dot[var[@]]
215.		1009	2.251	.008	dot[var[global,@]]
199.		1929	1.384	.009	dot[var[local,@]]
234.		1929	0.000	0.000	dot[var[local,*,*,@]]
138.		11	0.000	0.000	downthru[@]
139.		11	.946	.000	downthru[*,@]
140.		10	0.000	0.000	dst[@]
126.		29	0.000	0.000	enable[@]
127.		29	.431	.000	enable[*,@]
80.		129	.065	.000	error[@]
81.		129	1.903	.001	error[*,@]
82.		129	.065	.000	error[*,*,@]
235.		45	0.000	0.000	error[*,var[local,*,*,@]]
177.		128	0.000	0.000	error[var[@]]
142.		9	0.000	0.000	extract[@]
143.		9	1.224	.000	extract[*,@]
136.		11	.845	.000	fdollar[@]
137.		11	0.000	0.000	fdollar[*,@]
62.		196	0.000	0.000	fextract[@]
63.		196	.672	.000	fextract[*,@]
222.		196	1.099	.001	fextract[list[@]]
256.		354	.663	.001	fextract[list[*,...@]]
100.		84	0.000	0.000	forseq[@]
101.		84	2.496	.001	forseq[*,@]
102.		84	1.745	.000	forseq[*,*,@]
200.		84	.963	.000	forseq[var[local,@]]
85.		105	0.000	0.000	goto[@]
59.		211	2.315	.002	ifexp[@]
60.		211	2.433	.002	ifexp[*,@]
61.		211	3.005	.002	ifexp[*,*,@]
21.		1810	3.047	.019	ifstmt[@]
22.		1810	3.093	.019	ifstmt[*,@]
23.		1810	1.333	.008	ifstmt[*,*,@]
224.		173	2.051	.001	ifstmt[*,*,list[@]]
223.		598	2.132	.004	ifstmt[*,list[@]]

C.5. *Final Pattern Set -- Alphabetical by Pattern*

pattern number	*was refined*	*freq.*	*H*	*H contr.*	*pattern*
236.		113	0.000	0.000	ifstmt[var[local,*,*,@]]
86.		102	1.492	.001	in[@]
87.		102	.323	.000	in[*,@]
51.		338	.965	.001	index[@]
52.		338	2.325	.003	index[*,@]
187.		85	1.606	.000	index[*,num[@]]
55.		262	3.019	.003	inlinecall[@]
56.		262	.843	.001	inlinecall[*,@]
225.		230	.549	.000	inlinecall[*,list[@]]
257.		489	3.016	.005	inlinecall[*,list[*,...@]]
67.		183	.483	.000	intCC[@]
68.		183	1.466	.001	intCC[*,@]
188.		168	2.569	.001	intCC[num[@]]
88.		94	.849	.000	intCO[@]
89.		94	2.573	.001	intCO[*,@]
189.		78	.477	.000	intCO[num[@]]
151.		3	0.000	0.000	intOC[@]
152.		3	0.000	0.000	intOC[*,@]
149.		3	0.000	0.000	intOO[@]
150.		3	.918	.000	intOO[*,@]
28.		1214	.936	.004	item[@]
29.		1214	3.315	.014	item[*,@]
227.		425	2.096	.003	item[*,list[@]]
259.		1355	2.625	.012	item[*,list[*,...@]]
226.		97	1.190	.000	item[list[@]]
258.		243	.422	.000	item[list[*,...@]]
119.		41	1.883	.000	label[@]
120.		41	.281	.000	label[*,@]
260.		187	2.311	.001	label[list[*,...@]]
77.		159	.984	.001	lbl[@]
111.		50	.529	.000	length[@]
7.	*	2874	3.163	.031	list[@]
5.	*	3096	3.130	.033	list[*,...@]
201.		1909	2.798	.018	list[*,...var[local,@]]
237.		1909	2.104	.014	list[*,...var[local,*,*,@]]
144.		9	0.000	0.000	lst[@]
141.		10	0.000	0.000	lstf[@]
132.		25	0.000	0.000	max[@]
135.		13	1.884	.000	memory[@]
130.		27	0.000	0.000	min[@]
44.		469	2.672	.004	minus[@]
45.		469	1.590	.003	minus[*,@]
190.		318	2.446	.003	minus[*,num[@]]
203.		90	2.595	.001	minus[*,var[local,@]]
202.		199	2.803	.002	minus[var[local,@]]
128.		27	2.203	.000	mod[@]
129.		27	.979	.000	mod[*,@]
115.		44	0.000	0.000	mwconst[@]
261.		133	0.000	0.000	mwconst[list[*,...@]]
48.		382	2.636	.003	not[@]
6.	*	4850	5.828	.096	num[@]
92.		91	0.000	0.000	openstmt[@]
93.		91	1.052	.000	openstmt[*,@]
262.		391	2.113	.003	openstmt[*,list[*,...@]]
83.		127	2.955	.001	or[@]
84.		127	2.856	.001	or[*,@]
15.		3847	1.032	.013	plus[@]
16.		3847	1.180	.015	plus[*,@]
191.		316	2.807	.003	plus[*,num[@]]
205.		3027	1.795	.018	plus[*,var[local,@]]
204.		321	2.684	.003	plus[var[local,@]]
96.		87	0.000	0.000	register[@]
193.		87	2.221	.001	register[num[@]]
19.		2348	2.023	.016	relE[@]
20.		2348	.819	.007	relE[*,@]

C.5. *Final Pattern Set -- Alphabetical by Pattern*

pattern number	*was refined*	*freq.*	*H*	*H contr.*	*pattern*
117.		41	1.954	.000	relGE[@]
118.		41	2.022	.000	relGE[*,@]
75.		159	2.138	.001	relG[@]
76.		159	1.386	.001	relG[*,@]
124.		30	1.826	.000	relLE[@]
125.		30	1.505	.000	relLE[*,@]
90.		93	2.197	.001	relL[@]
91.		93	2.141	.001	relL[*,@]
36.		538	2.411	.004	relN[@]
37.		538	1.174	.002	relN[*,@]
116.		44	.774	.000	resume[@]
24.		1683	2.521	.014	return[@]
263.		160	1.636	.001	return[list[*,...@]]
195.		148	2.227	.001	return[num[@]]
181.		337	.692	.001	return[var[@]]
112.		47	0.000	0.000	row[@]
264.		686	.882	.002	row[list[*,...@]]
113.		47	0.000	0.000	rowcons[@]
114.		47	0.000	0.000	rowcons[*,@]
103.		82	.592	.000	seqindex[@]
104.		82	1.736	.000	seqindex[*,@]
239.		50	.242	.000	seqindex[*,var[local,*,*,@]]
238.		58	0.000	0.000	seqindex[var[local,*,*,@]]
64.		186	.524	.000	signal[@]
65.		186	1.602	.001	signal[*,@]
66.		186	0.000	0.000	signal[*,*,@]
228.		44	0.000	0.000	signal[*,list[@]]
265.		88	1.680	.001	signal[*,list[*,...@]]
178.		164	0.000	0.000	signal[var[@]]
217.		164	3.938	.002	signal[var[global,@]]
146.		8	.811	.000	start[@]
147.		8	0.000	0.000	start[*,@]
148.		8	.544	.000	start[*,*,@]
153.		3	0.000	0.000	stop[@]
38.		511	8.735	.015	str[@]
131.		27	0.000	0.000	stringinit[@]
154.		2	0.000	0.000	svc[@]
53.		317	1.702	.002	times[@]
54.		317	.535	.001	times[*,@]
192.		292	2.387	.002	times[*,num[@]]
206.		162	2.598	.001	times[var[local,@]]
121.		35	1.391	.000	uminus[@]
71.		179	0.000	0.000	unionx[@]
72.		179	0.000	0.000	unionx[*,@]
165.		179	2.818	.002	unionx[var[*,*,@]]
27.		1260	1.184	.005	uparrow[@]
182.		372	.268	.000	uparrow[var[@]]
211.		355	1.130	.001	uparrow[var[local,@]]
240.		355	0.000	0.000	uparrow[var[local,*,*,@]]
73.		170	.767	.000	upthru[@]
74.		170	1.220	.001	upthru[*,@]
183.		132	.156	.000	upthru[var[@]]
212.		129	2.023	.001	upthru[var[local,@]]
241.		129	1.400	.001	upthru[var[local,*,*,@]]
1.	*	10800	1.141	.042	var[@]
2.	*	0	0.000	0.000	var[*,@]
3.	*	31424	0.000	0.000	var[*,*,@]
4.	*	0	0.000	0.000	var[*,*,*,@]
158.		2686	4.039	.037	var[entry,@]
162.		2686	0.000	0.000	var[entry,*,*,@]
157.	*	428	1.491	.002	var[field,@]
161.	*	428	1.749	.003	var[field,*,*,@]
155.	*	5332	3.411	.062	var[global,@]
159.		12217	1.114	.046	var[global,*,*,@]
156.	*	2335	3.153	.025	var[local,@]

C.5. *Final Pattern Set -- Alphabetical by Pattern*

pattern number	*was refined*	*freq.*	*H*	*H contr.*	*pattern*
160.	*	6685	2.327	.053	var[local,*,*,@]
122.		32	0.000	0.000	vconstruct[@]
123.		32	0.000	0.000	vconstruct[*,@]
266.		32	0.000	0.000	vconstruct[*,list[*,...@]]
197.		610	2.949	.006	{assign|assignx}[*,var[local,@]]
196.		3296	2.995	.033	{assign|assignx}[var[local,@]]
254.		670	2.948	.007	{construct|constructx}[*,list[*,...@]]
185.		289	.615	.001	{index|dindex|seqindex}[*,var[@]]
208.		245	2.364	.002	{index|dindex|seqindex}[*,var[local,@]]
184.		757	1.317	.003	{index|dindex|seqindex}[var[@]]
207.		264	3.453	.003	{index|dindex|seqindex}[var[local,@]]
180.		403	.998	.001	{relE|relG|relN|relL|relGE|relLE}[*,var[@]]
179.		955	.549	.002	{relE|relG|relN|relL|relGE|relLE}[var[@]]
194.		2610	4.744	.042	{relL|relLE|relE|relN|relGE|relG}[*,num[@]]
210.		191	3.006	.002	{relL|relLE|relE|relN|relGE|relG}[*,var[local,@]]
209.		853	2.607	.008	{relL|relLE|relE|relN|relGE|relG}[var[local,@]]

C.6. Final Pattern Set -- Sorted by Pattern Number

```
(1)    var[@]
 f = 10800 H =  1.141, p =  .037, H contr. =   .042
SEE ALSO:      f     H     p contr
  (166)      339  .958  .001  .001, addr[var[@]]
  (167)     3885  .752  .013  .010, assign[var[@]]
  (168)      775  .959  .003  .003, assign[*,var[@]]
  (169)      223  .688  .001  .001, assignx[var[@]]
  (170)       30 1.159  .000  .000, assignx[*,var[@]]
  (171)     6370  .992  .022  .021, call[var[@]]
  (172)     1739  .806  .006  .005, call[*,var[@]]
  (173)     2941  .939  .010  .009, dot[var[@]]
  (174)     5697  .855  .019  .017, dot[*,var[@]]
  (175)      964  .682  .003  .002, dollar[var[@]]
  (176)     2754  .126  .009  .001, dollar[*,var[@]]
  (177)      128 0.000  .000 0.000, error[var[@]]
  (178)      164 0.000  .001 0.000, signal[var[@]]
  (179)      955  .549  .003  .002, {relE|relG|relN|relL|relGE|relLE}[var[@]]
  (180)      403  .998  .001  .001, {relE|relG|relN|relL|relGE|relLE}[*,var[@]]
  (181)      337  .692  .001  .001, return[var[@]]
  (182)      372  .268  .001  .000, uparrow[var[@]]
  (183)      132  .156  .000  .000, upthru[var[@]]
  (184)      757 1.317  .003  .003, {index|dindex|seqindex}[var[@]]
  (185)      289  .615  .001  .001, {index|dindex|seqindex}[*,var[@]]
    4 cases
  local       6816    .63      field        209    .02
  global      3632    .34      entry        143    .01

(2)    var[*,@]
 f =      0 H =  0.000, p = 0.000, H contr. =  0.000
SEE ALSO:      f     H     p contr
  (155)     5332 3.411  .018  .062, var[global,@]
  (156)     2335 3.153  .008  .025, var[local,@]
  (157)      428 1.491  .001  .002, var[field,@]
  (158)     2686 4.039  .009  .037, var[entry,@]

(3)    var[*,*,@]
 f = 31424 H =  0.000, p =  .106, H contr. =  0.000
SEE ALSO:      f     H     p contr
  (163)     5697  .657  .019  .013, dot[*,var[*,*,@]]
  (164)     2754 1.713  .009  .016, dollar[*,var[*,*,@]]
  (165)      179 2.818  .001  .002, unionx[var[*,*,@]]
  0          31424  1.00

(4)    var[*,*,*,@]
 f =      0 H =  0.000, p = 0.000, H contr. =  0.000
SEE ALSO:      f     H     p contr
  (159)    12217 1.114  .041  .046, var[global,*,*,@]
  (160)     6685 2.327  .023  .053, var[local,*,*,@]
  (161)      428 1.749  .001  .003, var[field,*,*,@]
  (162)     2686 0.000  .009 0.000, var[entry,*,*,@]

(5)    list[*,...@]
 f =  3096 H =  3.130, p =  .010, H contr. =   .033
SEE ALSO:      f     H     p contr
  (244)      160 2.536  .001  .001, arraydesc[list[*,...@]]
  (245)     6617 2.639  .022  .059, body[list[*,...@]]
  (246)     4263 2.425  .014  .035, call[*,list[*,...@]]
  (247)      120  .495  .000  .000, caseexp[*,list[*,...@]]
  (248)      982  .363  .003  .001, casestmt[*,list[*,...@]]
  (249)      159 2.228  .001  .001, casestmt[*,*,list[*,...@]]
  (250)      572 0.000  .002 0.000, caseswitch[*,*,list[*,...@]]
  (251)      254 0.000  .001 0.000, casetest[list[*,...@]]
  (252)      600 1.863  .002  .004, casetest[*,list[*,...@]]
  (253)       42 0.000  .000 0.000, catchphrase[list[*,...@]]
  (254)      670 2.948  .002  .007, {construct|constructx}[*,list[*,...@]]
  (255)      828 2.427  .003  .007, dostmt[*,*,list[*,...@]]
  (256)      354  .663  .001  .001, fextract[list[*,...@]]
```

C.6. *Final Pattern Set -- Sorted by Pattern Number*

```
  (257)      489 3.016  .002  .005, inlinecall[*,list[*,...@]]
  (258)      243  .422  .001  .000, item[list[*,...@]]
  (259)     1355 2.625  .005  .012, item[*,list[*,...@]]
  (260)      187 2.311  .001  .001, label[list[*,...@]]
  (261)      133 0.000  .000 0.000, mwconst[list[*,...@]]
  (262)      391 2.113  .001  .003, openstmt[*,list[*,...@]]
  (263)      160 1.636  .001  .001, return[list[*,...@]]
  (264)      686  .882  .002  .002, row[list[*,...@]]
  (265)       88 1.680  .000  .001, signal[*,list[*,...@]]
  (266)       32 0.000  .000 0.000, vconstruct[*,list[*,...@]]
   44 cases (17 shown)
  assign       966   .31      num           82   .03      signal       25   .01
  call         901   .29      dostmt        73   .02      dot          23   .01
  ifstmt       265   .09      casestmt      63   .02      unionx       23   .01
  var          168   .05      fextract      57   .02      dollar       22   .01
  return       140   .05      construct     43   .01      exit         18   .01
  <empty>       85   .03      bump          40   .01      <<others>>  102   .03

(6)    num[@]
 f =  4850 H =  5.828, p =  .016, H contr. =   .096
SEE ALSO:      f      H     p contr
  (186)     1078 3.056  .004  .011, assign[*,num[@]]
  (187)       85 1.606  .000  .000, index[*,num[@]]
  (188)      168 2.569  .001  .001, intCC[num[@]]
  (189)       78  .477  .000  .000, intCO[num[@]]
  (190)      318 2.446  .001  .003, minus[*,num[@]]
  (191)      316 2.807  .001  .003, plus[*,num[@]]
  (192)      292 2.387  .001  .002, times[*,num[@]]
  (193)       87 2.221  .000  .001, register[num[@]]
  (194)     2610 4.744  .009  .042, {relL|relLE|relE|relN|relGE|relG}[*,num[@]]
  (195)      148 2.227  .001  .001, return[num[@]]
  388 cases (25 shown)
  0            930   .19      65535         91   .02      11           33   .01
  1            606   .12      16383         85   .02      256          30   .01
  2            318   .07      6             80   .02      40           29   .01
  3            195   .04      7             75   .02      17           27   .01
  4            152   .03      8             60   .01      12           26   .01
  13           114   .02      10            58   .01      14           26   .01
  16           104   .02      32            56   .01      63           26   .01
  5            103   .02      9             44   .01      <<others>> 1450   .30
  -1            93   .02      255           39   .01

(7)    list[@]
 f =  2874 H =  3.163, p =  .010, H contr. =   .031
SEE ALSO:      f      H     p contr
  (218)       80 0.000  .000 0.000, arraydesc[list[@]]
  (219)     1816 1.085  .006  .007, call[*,list[@]]
  (220)      639 1.208  .002  .003, casestmt[*,list[@]]
  (221)      253 2.208  .001  .002, dostmt[*,*,list[@]]
  (222)      196 1.099  .001  .001, fextract[list[@]]
  (223)      598 2.132  .002  .004, ifstmt[*,list[@]]
  (224)      173 2.051  .001  .001, ifstmt[*,*,list[@]]
  (225)      230  .549  .001  .000, inlinecall[*,list[@]]
  (226)       97 1.190  .000  .000, item[list[@]]
  (227)      425 2.096  .001  .003, item[*,list[@]]
  (228)       44 0.000  .000 0.000, signal[*,list[@]]
   36 cases (15 shown)
  2            839   .29      7             79   .03      12           20   .01
  1            511   .18      8             65   .02      13           15   .01
  3            482   .17      0             60   .02      16           15   .01
  4            279   .10      10            53   .02      <<others>>   61   .02
  5            184   .06      9             44   .02
  6            137   .05      11            30   .01

(8)    call[@]
 f =  6616 H =   .238, p =  .022, H contr. =   .005
  var         6370   .96      dot          236   .04      dollar       10   .00

(9)    call[*,@]
 f =  6616 H =  2.961, p =  .022, H contr. =   .066
```

```
   27 cases (12 shown)
  list         1816   .27      dollar       209   .03      dindex        56   .01
  var          1739   .26      str          204   .03      ifexp         47   .01
  <empty>       760   .11      call         189   .03      <<others>>   142   .02
  num           686   .10      addr         129   .02
  dot           571   .09      plus          68   .01

(10)   call[*,*,@]
 f =  6616 H =   .112, p =  .022, H contr. =   .003
  <empty>      6517   .99      catchphrase   99   .01

(11)   assign[@]
 f =  5663 H =  1.423, p =  .019, H contr. =   .027
    9 cases
  var          3885   .69      uparrow      100   .02      seqindex      17   .00
  dot          1065   .19      index         79   .01      register       9   .00
  dollar        440   .08      dindex        63   .01      memory         5   .00

(12)   assign[*,@]
 f =  5663 H =  3.574, p =  .019, H contr. =   .069
   43 cases (18 shown)
  call         1288   .23      addr         208   .04      uparrow       36   .01
  num          1078   .19      assignx      151   .03      index         36   .01
  var           775   .14      minus        134   .02      mwconst       34   .01
  dot           580   .10      dindex       113   .02      times         31   .01
  <empty>       314   .06      inlinecall    72   .01      <<others>>   215   .04
  dollar        261   .05      arraydesc     63   .01
  plus          214   .04      ifexp         60   .01

(13)   dot[@]
 f =  5697 H =  1.401, p =  .019, H contr. =   .027
    9 cases
  var          2941   .52      dollar       109   .02      minus          7   .00
  plus         2349   .41      register      16   .00      call           7   .00
  dot           253   .04      num           11   .00      assignx        4   .00

(14)   dot[*,@]
 f =  5697 H =  0.000, p =  .019, H contr. =  0.000
  var          5697  1.00

(15)   plus[@]
 f =  3847 H =  1.032, p =  .013, H contr. =   .013
   22 cases (11 shown)
  var          3243   .84      times         39   .01      plus          16   .00
  dot           331   .09      minus         25   .01      call          15   .00
  dollar         49   .01      register      22   .01      caseexp        5   .00
  num            49   .01      index         21   .01      <<others>>    32   .01

(16)   plus[*,@]
 f =  3847 H =  1.180, p =  .013, H contr. =   .015
   17 cases (11 shown)
  var          3083   .80      call          51   .01      caseexp        6   .00
  num           316   .08      times         18   .00      ifexp          5   .00
  dollar        181   .05      div           13   .00      minus          5   .00
  dot           148   .04      inlinecall    10   .00      <<others>>    11   .00

(17)   dollar[@]
 f =  2754 H =  2.050, p =  .009, H contr. =   .019
    8 cases
  var           964   .35      dollar       106   .04      call          18   .01
  uparrow       944   .34      index         77   .03      assignx        1   .00
  dot           567   .21      dindex        77   .03

(18)   dollar[*,@]
 f =  2754 H =  0.000, p =  .009, H contr. =  0.000
  var          2754  1.00

(19)   relE[@]
 f =  2348 H =  2.023, p =  .008, H contr. =   .016
   15 cases
```

C.6. *Final Pattern Set -- Sorted by Pattern Number*

```
  <empty>      1249   .53      call          50   .02      index          7   .00
  var           558   .24      inlinecall    15   .01      bumpx          6   .00
  dot           230   .10      dindex        12   .01      plus           5   .00
  dollar        136   .06      seqindex      11   .00      minus          2   .00
  assignx        58   .02      mod            8   .00      uparrow        1   .00

(20)   relE[*,@]
 f =  2348 H =   .819, p =  .008, H contr. =   .007
   15 cases
  num          1976   .84      call           4   .00      plus           1   .00
  var           298   .13      dindex         3   .00      addr           1   .00
  dot            41   .02      mwconst        3   .00      seqindex       1   .00
  dollar         10   .00      relN           2   .00      length         1   .00
  relE            5   .00      or             1   .00      register       1   .00

(21)   ifstmt[@]
 f =  1810 H =  3.047, p =  .006, H contr. =   .019
   16 cases
  relE          593   .33      or            86   .05      relLE         20   .01
  relN          370   .20      dot           57   .03      in            18   .01
  not           157   .09      call          48   .03      assignx        3   .00
  var           135   .07      relL          36   .02      dindex         3   .00
  and           132   .07      relGE         29   .02
  relG          101   .06      dollar        22   .01

(22)   ifstmt[*,@]
 f =  1810 H =  3.093, p =  .006, H contr. =   .019
   25 cases (16 shown)
  list          598   .33      signal        70   .04      resume        12   .01
  call          324   .18      ifstmt        40   .02      construct     11   .01
  assign        239   .13      syserror      37   .02      openstmt      10   .01
  return        196   .11      goto          33   .02      bump          10   .01
  error          82   .05      dostmt        23   .01      <<others>>    32   .02
  exit           75   .04      casestmt      18   .01

(23)   ifstmt[*,*,@]
 f =  1810 H =  1.333, p =  .006, H contr. =   .008
   19 cases (11 shown)
  <empty>      1406   .78      ifstmt        45   .02      return         6   .00
  list          173   .10      casestmt      13   .01      goto           4   .00
  call           69   .04      openstmt      10   .01      fextract       2   .00
  assign         64   .04      dostmt         7   .00      <<others>>    11   .01

(24)   return[@]
 f =  1683 H =  2.521, p =  .006, H contr. =   .014
   30 cases (12 shown)
  <empty>       839   .50      caseexp       32   .02      relE          15   .01
  var           337   .20      constructx    31   .02      plus          15   .01
  num           148   .09      dollar        26   .02      <<others>>    47   .03
  list           74   .04      ifexp         24   .01
  call           73   .04      dot           22   .01

(25)   @
 f =  1513 H =  0.000, p =  .005, H contr. =  0.000
  body         1513  1.00

(26)   body[@]
 f =  1513 H =  0.000, p =  .005, H contr. =  0.000
  list         1513  1.00

(27)   uparrow[@]
 f =  1260 H =  1.184, p =  .004, H contr. =   .005
    7 cases
  plus          822   .65      dollar         4   .00      register       1   .00
  var           372   .30      minus          2   .00
  num            58   .05      dot            1   .00

(28)   item[@]
 f =  1214 H =   .936, p =  .004, H contr. =   .004
    9 cases
```

C.6. *Final Pattern Set -- Sorted by Pattern Number*

```
 relE        1021   .84      in          25   .02      relN          2   .00
 list          97   .08      relL         7   .01      relLE         2   .00
 lbl           52   .04      relG         7   .01      relGE         1   .00

(29)   item[*,@]
 f =  1214 H =  3.315, p =  .004, H contr. =   .014
   36 cases (19 shown)
 list         425   .35      num         26   .02      resume       11   .01
 assign       187   .15      dollar      20   .02      and          10   .01
 call         170   .14      goto        20   .02      exit         10   .01
 ifstmt       102   .08      openstmt    17   .01      dot           7   .01
 casestmt      42   .03      caseexp     14   .01      continue      7   .01
 nullstmt      41   .03      signal      12   .01      <<others>>   45   .04
 return        37   .03      error       11   .01

(30)   casestmt[@]
 f =   639 H =  1.543, p =  .002, H contr. =   .003
   11 cases
 dollar       450   .70      num         17   .03      uparrow       2   .00
 var           67   .10      assignx      4   .01      inlinecall    1   .00
 dot           55   .09      seqindex     3   .00      index         1   .00
 call          37   .06      minus        2   .00

(31)   casestmt[*,@]
 f =   639 H =  0.000, p =  .002, H contr. =  0.000
 list         639  1.00

(32)   casestmt[*,*,@]
 f =   639 H =  2.590, p =  .002, H contr. =   .006
   15 cases
 <empty>      269   .42      return      22   .03      nullstmt      7   .01
 syserror     151   .24      assign      21   .03      goto          4   .01
 list          61   .10      ifstmt      13   .02      casestmt      2   .00
 signal        35   .05      exit        10   .02      openstmt      1   .00
 call          32   .05      error       10   .02      bump          1   .00

(33)   casetest[@]
 f =   572 H =   .597, p =  .002, H contr. =   .001
 num          489   .85      list        83   .15

(34)   casetest[*,@]
 f =   572 H =  2.612, p =  .002, H contr. =   .005
   15 cases
 list         194   .34      nullstmt    16   .03      openstmt      4   .01
 assign       140   .24      str         15   .03      exit          3   .01
 call         112   .20      return      11   .02      goto          2   .00
 num           41   .07      ifexp        6   .01      signal        2   .00
 ifstmt        20   .03      casestmt     5   .01      label         1   .00

(35)   addr[@]
 f =   570 H =  1.824, p =  .002, H contr. =   .004
    6 cases
 var          339   .59      uparrow     68   .12      index        32   .06
 dot           84   .15      dollar      32   .06      dindex       15   .03

(36)   relN[@]
 f =   538 H =  2.411, p =  .002, H contr. =   .004
   16 cases
 var          209   .39      dindex       6   .01      mod           2   .00
 dot          157   .29      seqindex     6   .01      fdollar       1   .00
 dollar        69   .13      plus         4   .01      uparrow       1   .00
 call          37   .07      index        3   .01      length        1   .00
 inlinecall    24   .04      <empty>      2   .00
 assignx       14   .03      minus        2   .00

(37)   relN[*,@]
 f =   538 H =  1.174, p =  .002, H contr. =   .002
   10 cases
 num          430   .80      call         7   .01      inlinecall    2   .00
 var           53   .10      addr         5   .01      mwconst       1   .00
```

C.6. *Final Pattern Set -- Sorted by Pattern Number*

```
  dot           24   .04      seqindex       5   .01
  dollar         8   .01      uparrow        3   .01

(38)   str[@]
 f =    511 H =  8.735, p =  .002, H contr. =    .015
  449 cases (11 shown)
  "Break"        4   .01      "VM."          3   .01       "error"        3   .01
  "Trace"        4   .01      " XXX"         3   .01       "Error # "     3   .01
  ".xm"          3   .01      "NIL"          3   .01       "New"          2   .00
  " -- "         3   .01      ".XM"          3   .01       <<others>>   477   .93

(39)   dostmt[@]
 f =    484 H =  1.610, p =  .002, H contr. =    .003
    4 cases
  <empty>      219   .45      forseq        84   .17
  upthru       170   .35      downthru      11   .02

(40)   dostmt[*,@]
 f =    484 H =  1.777, p =  .002, H contr. =    .003
   12 cases
  <empty>      277   .57      relL          10   .02       in             3   .01
  not          136   .28      and            5   .01       var            3   .01
  relN          28   .06      relG           5   .01       relLE          2   .00
  relE          11   .02      relGE          3   .01       or             1   .00

(41)   dostmt[*,*,@]
 f =    484 H =  2.269, p =  .002, H contr. =    .004
   14 cases
  list         253   .52      openstmt       8   .02       enable         3   .01
  assign        62   .13      label          7   .01       inlinecall     1   .00
  ifstmt        54   .11      bump           4   .01       dostmt         1   .00
  casestmt      44   .09      signal         3   .01       catchmark      1   .00
  call          40   .08      nullstmt       3   .01

(42)   dostmt[*,*,*,@]
 f =    484 H =   .133, p =  .002, H contr. =    .000
  <empty>      475   .98      item           9   .02

(43)   dostmt[*,*,*,*,@]
 f =    484 H =   .292, p =  .002, H contr. =    .000
    8 cases
  <empty>      468   .97      ifstmt         2   .00       goto           1   .00
  list           6   .01      return         2   .00       exit           1   .00
  assign         3   .01      call           1   .00

(44)   minus[@]
 f =    469 H =  2.672, p =  .002, H contr. =    .004
   18 cases
  var          224   .48      call          11   .02       uminus         3   .01
  dot           62   .13      div           10   .02       length         3   .01
  <empty>       45   .10      minus          8   .02       times          2   .00
  plus          38   .08      index          7   .01       uparrow        2   .00
  dollar        25   .05      dindex         4   .01       assignx        1   .00
  num           20   .04      ifexp          3   .01       abs            1   .00

(45)   minus[*,@]
 f =    469 H =  1.590, p =  .002, H contr. =    .003
   14 cases
  num          318   .68      minus          4   .01       ifexp          1   .00
  var           91   .19      times          4   .01       assignx        1   .00
  dot           17   .04      index          4   .01       div            1   .00
  dollar        15   .03      addr           2   .00       inlinecall     1   .00
  plus           8   .02      call           2   .00

(46)   dindex[@]
 f =    433 H =   .195, p =  .001, H contr. =    .000
  var          420   .97      dot           13   .03

(47)   dindex[*,@]
 f =    433 H =  2.450, p =  .001, H contr. =    .004
```

C.6. *Final Pattern Set -- Sorted by Pattern Number*

```
   11 cases
  var          131   .30      minus         21   .05      call          3   .01
  num          102   .24      dot            8   .02      uparrow       2   .00
  times         89   .21      assignx        4   .01      ifexp         1   .00
  plus          68   .16      dollar         4   .01

(48)   not[@]
 f =   382 H =  2.636, p =  .001, H contr. =   .003
   13 cases
  relE         118   .31      in            19   .05      and           1   .00
  var           97   .25      relL           5   .01      relN          1   .00
  call          50   .13      relG           5   .01      relGE         1   .00
  dot           47   .12      or             4   .01
  dollar        32   .08      ifexp          2   .01

(49)   assignx[@]
 f =   323 H =  1.353, p =  .001, H contr. =   .001
    7 cases
  var          223   .69      uparrow        7   .02      dindex        1   .00
  dot           68   .21      seqindex       4   .01
  dollar        17   .05      index          3   .01

(50)   assignx[*,@]
 f =   323 H =  3.313, p =  .001, H contr. =   .004
   22 cases (14 shown)
  call          73   .23      minus         19   .06      register      6   .02
  num           71   .22      dollar        19   .06      inlinecall    3   .01
  dot           40   .12      dindex        12   .04      seqindex      3   .01
  var           30   .09      plus           9   .03      relE          2   .01
  assignx       22   .07      addr           6   .02      <<others>>    8   .02

(51)   index[@]
 f =   338 H =   .965, p =  .001, H contr. =   .001
  var          265   .78      dollar        42   .12      dot          31   .09

(52)   index[*,@]
 f =   338 H =  2.325, p =  .001, H contr. =   .003
   11 cases
  var          106   .31      dollar        13   .04      inlinecall    2   .01
  times         85   .25      mod            4   .01      ifexp         1   .00
  num           85   .25      plus           3   .01      div           1   .00
  minus         36   .11      dot            2   .01

(53)   times[@]
 f =   317 H =  1.702, p =  .001, H contr. =   .002
   13 cases
  var          227   .72      plus           9   .03      length        2   .01
  minus         22   .07      inlinecall     4   .01      ifexp         1   .00
  call          18   .06      times          3   .01      abs           1   .00
  dot           16   .05      div            2   .01
  dollar        10   .03      uminus         2   .01

(54)   times[*,@]
 f =   317 H =   .535, p =  .001, H contr. =   .001
    5 cases
  num          292   .92      dot            8   .03      dollar        1   .00
  var           10   .03      call           6   .02

(55)   inlinecall[@]
 f =   262 H =  3.019, p =  .001, H contr. =   .003
   18 cases
  BITAND        86   .33      Stop           9   .03      PORTI         2   .01
  BITSHIFT      42   .16      BITXOR         7   .03      CONVERT       1   .00
  BITOR         39   .15      LDIVMOD        4   .02      PUSH          1   .00
  DIVMOD        26   .10      BLOCK          4   .02      USC           1   .00
  COPY          17   .06      NovaOutLd      3   .01      LongDiv       1   .00
  BITNOT        16   .06      NovaInLd       2   .01      LongMult      1   .00

(56)   inlinecall[*,@]
 f =   262 H =   .843, p =  .001, H contr. =   .001
```

C.6. *Final Pattern Set -- Sorted by Pattern Number*

```
   8 cases
  list         230   .88      var           7   .03      uparrow       1   .00
  num            8   .03      dot           4   .02      index         1   .00
  <empty>        7   .03      inlinecall    4   .02

(57)    and[@]
 f =    251 H =   3.104, p =  .001, H contr. =   .003
   14 cases
  relE          60   .24      call         12   .05      or            5   .02
  and           51   .20      dot           8   .03      relGE         3   .01
  relN          37   .15      relL          7   .03      in            3   .01
  var           32   .13      relG          6   .02      relLE         1   .00
  not           20   .08      dollar        6   .02

(58)    and[*,@]
 f =    251 H =   3.113, p =  .001, H contr. =   .003
   17 cases
  relE          85   .34      or           10   .04      caseexp       2   .01
  relN          42   .17      dot          10   .04      index         2   .01
  not           28   .11      in            8   .03      and           1   .00
  call          22   .09      dollar        7   .03      relGE         1   .00
  var           12   .05      relL          6   .02      fdollar       1   .00
  relG          11   .04      relLE         3   .01

(59)    ifexp[@]
 f =    211 H =   2.315, p =  .001, H contr. =   .002
   12 cases
  relE         102   .48      relN          6   .03      dollar        4   .02
  var           54   .26      relL          6   .03      or            3   .01
  in            14   .07      dot           6   .03      not           3   .01
  relG           8   .04      and           4   .02      relGE         1   .00

(60)    ifexp[*,@]
 f =    211 H =   2.433, p =  .001, H contr. =   .002
   18 cases (11 shown)
  num          115   .55      dollar        9   .04      str           3   .01
  dot           25   .12      ifexp         6   .03      index         2   .01
  call          18   .09      plus          3   .01      memory        2   .01
  var           18   .09      minus         3   .01      <<others>>    7   .03

(61)    ifexp[*,*,@]
 f =    211 H =   3.005, p =  .001, H contr. =   .002
   21 cases
  num           95   .45      plus          6   .03      uminus        2   .01
  dot           20   .09      str           4   .02      addr          2   .01
  var           20   .09      caseexp       3   .01      relN          1   .00
  call          14   .07      index         3   .01      relL          1   .00
  ifexp         13   .06      dindex        3   .01      times         1   .00
  dollar         9   .04      min           3   .01      inlinecall    1   .00
  minus          7   .03      not           2   .01      constructx    1   .00

(62)    fextract[@]
 f =    196 H =   0.000, p =  .001, H contr. =  0.000
  list         196  1.00

(63)    fextract[*,@]
 f =    196 H =    .672, p =  .001, H contr. =   .000
  call         166   .85      inlinecall   28   .14      signal        2   .01

(64)    signal[@]
 f =    186 H =    .524, p =  .001, H contr. =   .000
  var          164   .88      dot          22   .12

(65)    signal[*,@]
 f =    186 H =   1.602, p =  .001, H contr. =   .001
    5 cases
  <empty>      104   .56      var          31   .17      num           2   .01
  list          44   .24      dollar        5   .03

(66)    signal[*,*,@]
```

C.6. *Final Pattern Set -- Sorted by Pattern Number*

```
 f =   186 H =  0.000, p =  .001, H contr. =  0.000
  <empty>       186  1.00

(67)   intCC[@]
 f =   183 H =   .483, p =  .001, H contr. =   .000
    4 cases
  num          168   .92       dot            2   .01
  var           12   .07       minus          1   .01

(68)   intCC[*,@]
 f =   183 H =  1.466, p =  .001, H contr. =   .001
    7 cases
  num          133   .73       plus           7   .04      dollar         1   .01
  var           19   .10       div            7   .04
  dot           10   .05       minus          6   .03

(69)   construct[@]
 f =   183 H =  1.817, p =  .001, H contr. =   .001
    6 cases
  var           90   .49       dot           26   .14      dollar         3   .02
  uparrow       52   .28       dindex        10   .05      index          2   .01

(70)   construct[*,@]
 f =   183 H =  0.000, p =  .001, H contr. =  0.000
  list         183  1.00

(71)   unionx[@]
 f =   179 H =  0.000, p =  .001, H contr. =  0.000
  var          179  1.00

(72)   unionx[*,@]
 f =   179 H =  0.000, p =  .001, H contr. =  0.000
  list         179  1.00

(73)   upthru[@]
 f =   170 H =   .767, p =  .001, H contr. =   .000
  var          132   .78       <empty>       38   .22

(74)   upthru[*,@]
 f =   170 H =  1.220, p =  .001, H contr. =   .001
    4 cases
  intCC         83   .49       intOO          3   .02
  intCO         81   .48       intOC          3   .02

(75)   relG[@]
 f =   159 H =  2.138, p =  .001, H contr. =   .001
   12 cases
  var           94   .59       assignx        8   .05      times          1   .01
  dot           21   .13       plus           7   .04      inlinecall     1   .01
  dollar        10   .06       length         5   .03      abs            1   .01
  <empty>        8   .05       index          2   .01      bumpx          1   .01

(76)   relG[*,@]
 f =   159 H =  1.386, p =  .001, H contr. =   .001
    9 cases
  num          120   .75       dollar         6   .04      ifexp          1   .01
  var           14   .09       index          3   .02      mod            1   .01
  dot           11   .07       minus          2   .01      max            1   .01

(77)   lbl[@]
 f =   159 H =   .984, p =  .001, H contr. =   .001
    5 cases
  1            127   .80       3              4   .03      5              2   .01
  2             23   .14       4              3   .02

(78)   catchphrase[@]
 f =   130 H =  1.040, p =  .000, H contr. =   .000
  item          98   .75       <empty>       20   .15      list          12   .09

(79)   catchphrase[*,@]
```

C.6. *Final Pattern Set -- Sorted by Pattern Number*

```
 f =    130 H =  1.024, p =  .000, H contr. =   .000
   8 cases
  <empty>      109   .84      continue      4   .03      assign        1   .01
  goto           8   .06      list          2   .02      error         1   .01
  inlinecall     4   .03      call          1   .01

(80)   error[@]
 f =    129 H =   .065, p =  .000, H contr. =   .000
  var          128   .99      dot           1   .01

(81)   error[*,@]
 f =    129 H =  1.903, p =  .000, H contr. =   .001
   9 cases
  <empty>       59   .46      num           6   .05      ifexp         1   .01
  var           47   .36      dot           3   .02      plus          1   .01
  list           9   .07      str           2   .02      addr          1   .01

(82)   error[*,*,@]
 f =    129 H =   .065, p =  .000, H contr. =   .000
  <empty>      128   .99      catchphrase   1   .01

(83)   or[@]
 f =    127 H =  2.955, p =  .000, H contr. =   .001
  11 cases
  relE          38   .30      relG          8   .06      dot           5   .04
  relN          21   .17      var           8   .06      call          2   .02
  not           20   .16      and           7   .06      assignx       1   .01
  or            10   .08      relL          7   .06

(84)   or[*,@]
 f =    127 H =  2.856, p =  .000, H contr. =   .001
  14 cases
  relE          43   .34      relG          5   .04      dollar        2   .02
  and           25   .20      dot           4   .03      relGE         1   .01
  relN          21   .17      call          3   .02      relLE         1   .01
  not           10   .08      caseexp       2   .02      in            1   .01
  var            7   .06      relL          2   .02

(85)   goto[@]
 f =    105 H =  0.000, p =  .000, H contr. =  0.000
  lbl          105  1.00

(86)   in[@]
 f =    102 H =  1.492, p =  .000, H contr. =   .001
   6 cases
  var           57   .56      minus         5   .05      plus          1   .01
  <empty>       35   .34      dollar        3   .03      call          1   .01

(87)   in[*,@]
 f =    102 H =   .323, p =  .000, H contr. =   .000
  intCC         96   .94      intCO         6   .06

(88)   intCO[@]
 f =     94 H =   .849, p =  .000, H contr. =   .000
   4 cases
  num           78   .83      dot           4   .04
  var           11   .12      length        1   .01

(89)   intCO[*,@]
 f =     94 H =  2.573, p =  .000, H contr. =   .001
   9 cases
  var           38   .40      plus         10   .11      min           3   .03
  length        15   .16      dollar        7   .07      div           2   .02
  dot           12   .13      minus         5   .05      call          2   .02

(90)   relL[@]
 f =     93 H =  2.197, p =  .000, H contr. =   .001
   9 cases
  var           51   .55      assignx       9   .10      call          2   .02
  dollar        10   .11      dot           6   .06      index         2   .02
```

C.6. *Final Pattern Set -- Sorted by Pattern Number*

```
 <empty>          9   .10      plus          3   .03      div            1   .01

(91)   relL[*,@]
 f =     93 H =  2.141, p =  .000, H contr. =   .001
    9 cases
  num            46   .49      dollar        6   .06      times          1   .01
  var            20   .22      length        4   .04      index          1   .01
  dot            12   .13      div           2   .02      min            1   .01

(92)   openstmt[@]
 f =     91 H =  0.000, p =  .000, H contr. =  0.000
 <empty>         91  1.00

(93)   openstmt[*,@]
 f =     91 H =  1.052, p =  .000, H contr. =   .000
    8 cases
  list           76   .84      casestmt      2   .02      ifstmt         1   .01
  enable          5   .05      inlinecall    1   .01      dostmt         1   .01
  label           4   .04      assign        1   .01

(94)   div[@]
 f =     87 H =  2.183, p =  .000, H contr. =   .001
    7 cases
  var            37   .43      dollar        6   .07      minus          1   .01
  plus           23   .26      call          6   .07
  dot            11   .13      times         3   .03

(95)   div[*,@]
 f =     87 H =   .572, p =  .000, H contr. =   .000
    4 cases
  num            79   .91      dot           3   .03
  var             4   .05      times         1   .01

(96)   register[@]
 f =     87 H =  0.000, p =  .000, H contr. =  0.000
  num            87  1.00

(97)   caseexp[@]
 f =     84 H =  1.343, p =  .000, H contr. =   .000
    7 cases
  dollar         62   .74      call          2   .02      num            1   .01
  var            12   .14      inlinecall    2   .02
  dot             4   .05      dindex        1   .01

(98)   caseexp[*,@]
 f =     84 H =  0.000, p =  .000, H contr. =  0.000
  list           84  1.00

(99)   caseexp[*,*,@]
 f =     84 H =   .885, p =  .000, H contr. =   .000
    7 cases
  num            73   .87      dot           2   .02      var            1   .01
  call            3   .04      str           2   .02
  ifexp           2   .02      dindex        1   .01

(100)  forseq[@]
 f =     84 H =  0.000, p =  .000, H contr. =  0.000
  var            84  1.00

(101)  forseq[*,@]
 f =     84 H =  2.496, p =  .000, H contr. =   .001
   10 cases
  var            30   .36      index         4   .05      minus          1   .01
  dot            25   .30      num           4   .05      div            1   .01
  call           10   .12      dindex        3   .04
  dollar          4   .05      plus          2   .02

(102)  forseq[*,*,@]
 f =     84 H =  1.745, p =  .000, H contr. =   .000
    5 cases
```

C.6. *Final Pattern Set -- Sorted by Pattern Number*

```
 dot            49   .58       var           10   .12      minus            3   .04
 call           14   .17       plus           8   .10

(103)  seqindex[@]
 f =     82 H =   .592, p =  .000, H contr. =    .000
 var            72   .88       dot            9   .11      dindex           1   .01

(104)  seqindex[*,@]
 f =     82 H =  1.736, p =  .000, H contr. =    .000
   6 cases
 var            52   .63       dot            7   .09      bumpx            4   .05
 minus          10   .12       num            7   .09      plus             2   .02

(105)  caseswitch[@]
 f =     81 H =  1.175, p =  .000, H contr. =    .000
 minus          45   .56       <empty>       33   .41      plus             3   .04

(106)  caseswitch[*,@]
 f =     81 H =  0.000, p =  .000, H contr. =  0.000
 num            81  1.00

(107)  caseswitch[*,*,@]
 f =     81 H =  0.000, p =  .000, H contr. =  0.000
 list           81  1.00

(108)  arraydesc[@]
 f =     80 H =  0.000, p =  .000, H contr. =  0.000
 list           80  1.00

(109)  constructx[@]
 f =     77 H =  0.000, p =  .000, H contr. =  0.000
 temp           77  1.00

(110)  constructx[*,@]
 f =     77 H =  0.000, p =  .000, H contr. =  0.000
 list           77  1.00

(111)  length[@]
 f =     50 H =   .529, p =  .000, H contr. =    .000
 var            44   .88       dot            6   .12

(112)  row[@]
 f =     47 H =  0.000, p =  .000, H contr. =  0.000
 list           47  1.00

(113)  rowcons[@]
 f =     47 H =  0.000, p =  .000, H contr. =  0.000
 var            47  1.00

(114)  rowcons[*,@]
 f =     47 H =  0.000, p =  .000, H contr. =  0.000
 row            47  1.00

(115)  mwconst[@]
 f =     44 H =  0.000, p =  .000, H contr. =  0.000
 list           44  1.00

(116)  resume[@]
 f =     44 H =   .774, p =  .000, H contr. =    .000
   4 cases
 <empty>        38   .86       dot            2   .05
 var             3   .07       list           1   .02

(117)  relGE[@]
 f =     41 H =  1.954, p =  .000, H contr. =    .000
   7 cases
 var            23   .56       dollar         3   .07      plus             1   .02
 bumpx           8   .20       assignx        2   .05
 dot             3   .07       <empty>        1   .02
```

C.6. *Final Pattern Set -- Sorted by Pattern Number*

```
(118)  relGE[*,@]
 f =     41 H =  2.022, p =  .000, H contr. =   .000
   7 cases
  num           18   .44     dot           2   .05     max            1   .02
  var           13   .32     assignx       1   .02
  length         5   .12     dollar        1   .02

(119)  label[@]
 f =     41 H =  1.883, p =  .000, H contr. =   .000
   7 cases
  list          25   .61     ifstmt        3   .07     assign         1   .02
  enable         5   .12     catchmark     2   .05
  casestmt       4   .10     call          1   .02

(120)  label[*,@]
 f =     41 H =   .281, p =  .000, H contr. =   .000
  item          39   .95     list          2   .05

(121)  uminus[@]
 f =     35 H =  1.391, p =  .000, H contr. =   .000
   5 cases
  var           25   .71     call          3   .09     dindex         1   .03
  dollar         4   .11     dot           2   .06

(122)  vconstruct[@]
 f =     32 H =  0.000, p =  .000, H contr. =  0.000
  dollar        32  1.00

(123)  vconstruct[*,@]
 f =     32 H =  0.000, p =  .000, H contr. =  0.000
  list          32  1.00

(124)  relLE[@]
 f =     30 H =  1.826, p =  .000, H contr. =   .000
   8 cases
  var           20   .67     abs           2   .07     dollar         1   .03
  <empty>        2   .07     times         1   .03     index          1   .03
  assignx        2   .07     dot           1   .03

(125)  relLE[*,@]
 f =     30 H =  1.505, p =  .000, H contr. =   .000
   5 cases
  num           20   .67     dot           2   .07     dollar         1   .03
  var            5   .17     index         2   .07

(126)  enable[@]
 f =     29 H =  0.000, p =  .000, H contr. =  0.000
  catchphrase   29  1.00

(127)  enable[*,@]
 f =     29 H =   .431, p =  .000, H contr. =   .000
  list          27   .93     ifstmt        1   .03     dostmt         1   .03

(128)  mod[@]
 f =     27 H =  2.203, p =  .000, H contr. =   .000
   6 cases
  var           12   .44     dot           3   .11     bumpx          3   .11
  plus           5   .19     inlinecall    3   .11     dollar         1   .04

(129)  mod[*,@]
 f =     27 H =   .979, p =  .000, H contr. =   .000
  num           20   .74     length        6   .22     var            1   .04

(130)  min[@]
 f =     27 H =  0.000, p =  .000, H contr. =  0.000
  list          27  1.00

(131)  stringinit[@]
 f =     27 H =  0.000, p =  .000, H contr. =  0.000
  num           27  1.00
```

C.6. *Final Pattern Set -- Sorted by Pattern Number*

```
(132)  max[@]
 f =     25 H =  0.000, p =  .000, H contr. =  0.000
  list           25  1.00

(133)  catchmark[@]
 f =     24 H =  1.908, p =  .000, H contr. =   .000
    5 cases
  call           10   .42     enable         6   .25       ifstmt         1   .04
  assign          6   .25     fextract       1   .04

(134)  base[@]
 f =     22 H =   .439, p =  .000, H contr. =   .000
  var            20   .91     dot            2   .09

(135)  memory[@]
 f =     13 H =  1.884, p =  .000, H contr. =   .000
    4 cases
  var             5   .38     plus           2   .15
  num             4   .31     minus          2   .15

(136)  fdollar[@]
 f =     11 H =   .845, p =  .000, H contr. =   .000
  call            8   .73     inlinecall     3   .27

(137)  fdollar[*,@]
 f =     11 H =  0.000, p =  .000, H contr. =  0.000
  var            11  1.00

(138)  downthru[@]
 f =     11 H =  0.000, p =  .000, H contr. =  0.000
  var            11  1.00

(139)  downthru[*,@]
 f =     11 H =   .946, p =  .000, H contr. =   .000
  intCO           7   .64     intCC          4   .36

(140)  dst[@]
 f =     10 H =  0.000, p =  .000, H contr. =  0.000
  var            10  1.00

(141)  lstf[@]
 f =     10 H =  0.000, p =  .000, H contr. =  0.000
  var            10  1.00

(142)  extract[@]
 f =      9 H =  0.000, p =  .000, H contr. =  0.000
  list            9  1.00

(143)  extract[*,@]
 f =      9 H =  1.224, p =  .000, H contr. =   .000
  call            6   .67     uparrow        2   .22       dollar         1   .11

(144)  lst[@]
 f =      9 H =  0.000, p =  .000, H contr. =  0.000
  var             9  1.00

(145)  abs[@]
 f =      8 H =  1.299, p =  .000, H contr. =   .000
  var             5   .62     call           2   .25       dot            1   .12

(146)  start[@]
 f =      8 H =   .811, p =  .000, H contr. =   .000
  var             6   .75     dot            2   .25

(147)  start[*,@]
 f =      8 H =  0.000, p =  .000, H contr. =  0.000
  <empty>         8  1.00
```

C.6. *Final Pattern Set -- Sorted by Pattern Number*

```
(148)  start[*,*,@]
 f =      8 H =   .544, p =  .000, H contr. =   .000
  <empty>         7   .87      catchphrase    1   .12

(149)  int0O[@]
 f =      3 H =  0.000, p =  .000, H contr. =  0.000
  var             3  1.00

(150)  int0O[*,@]
 f =      3 H =   .918, p =  .000, H contr. =   .000
  var             2   .67      plus           1   .33

(151)  int0C[@]
 f =      3 H =  0.000, p =  .000, H contr. =  0.000
  var             3  1.00

(152)  int0C[*,@]
 f =      3 H =  0.000, p =  .000, H contr. =  0.000
  var             3  1.00

(153)  stop[@]
 f =      3 H =  0.000, p =  .000, H contr. =  0.000
  <empty>         3  1.00

(154)  svc[@]
 f =      2 H =  0.000, p =  .000, H contr. =  0.000
  num             2  1.00

(155)  var[global,@]
 f =  5332 H =  3.411, p =  .018, H contr. =   .062
SEE ALSO:      f      H     p  contr
  (213)      752 4.743  .003  .012, assign[var[global,@]]
  (214)     4016 4.241  .014  .058, call[var[global,@]]
  (215)     1009 2.251  .003  .008, dot[var[global,@]]
  (216)      944 4.912  .003  .016, dot[*,var[global,@]]
  (217)      164 3.938  .001  .002, signal[var[global,@]]
  134 cases (12 shown)
  0             2396    .45      5           149    .03      44            31    .01
  1              925    .17      6           111    .02      12            27    .01
  2              361    .07      7            75    .01      <<others>>   722    .14
  3              236    .04      40           59    .01
  4              198    .04      -10          42    .01

(156)  var[local,@]
 f =  2335 H =  3.153, p =  .008, H contr. =   .025
SEE ALSO:     f      H     p  contr
  (196)     3296 2.995  .011  .033, {assign|assignx}[var[local,@]]
  (197)      610 2.949  .002  .006, {assign|assignx}[*,var[local,@]]
  (198)     1445 2.275  .005  .011, call[*,var[local,@]]
  (199)     1929 1.384  .007  .009, dot[var[local,@]]
  (200)       84  .963  .000  .000, forseq[var[local,@]]
  (201)     1909 2.798  .006  .018, list[*,...var[local,@]]
  (202)      199 2.803  .001  .002, minus[var[local,@]]
  (203)       90 2.595  .000  .001, minus[*,var[local,@]]
  (204)      321 2.684  .001  .003, plus[var[local,@]]
  (205)     3027 1.795  .010  .018, plus[*,var[local,@]]
  (206)      162 2.598  .001  .001, times[var[local,@]]
  (207)      264 3.453  .001  .003, {index|dindex|seqindex}[var[local,@]]
  (208)      245 2.364  .001  .002, {index|dindex|seqindex}[*,var[local,@]]
  (209)      853 2.607  .003  .008, {relL|relLE|relE|relN|relGE|relG}[var[local,@]]
  (210)      191 3.006  .001  .002, {relL|relLE|relE|relN|relGE|relG}[*,var[local,@]]
  (211)      355 1.130  .001  .001, uparrow[var[local,@]]
  (212)      129 2.023  .000  .001, upthru[var[local,@]]
  27 cases (15 shown)
  0              580    .25      7           116    .05      12            14    .01
  1              503    .22      6            63    .03      11            13    .01
  2              379    .16      8            40    .02      10            12    .01
  3              217    .09      9            35    .01      <<others>>    41    .02
  4              160    .07      14           19    .01
  5              127    .05      13           16    .01
```

C.6. *Final Pattern Set -- Sorted by Pattern Number*

```
(157)  var[field,@]
 f =    428 H =  1.491, p =  .001, H contr. =    .002
SEE ALSO:       f     H      p contr
  (242)     2709 2.433  .009  .022, dollar[*,var[field,@]]
  (243)     4570 3.441  .015  .053, dot[*,var[field,@]]
   6 cases
  5             193  .45      6               52   .12      3                1   .00
  0             179  .42      4                2   .00      7                1   .00

(158)  var[entry,@]
 f =   2686 H =  4.039, p =  .009, H contr. =   .037
   54 cases (24 shown)
  1             651  .24      10              67   .02      20              29   .01
  2             350  .13      11              57   .02      21              29   .01
  3             267  .10      14              54   .02      18              24   .01
  4             190  .07      13              53   .02      23              22   .01
  5             155  .06      12              52   .02      22              19   .01
  6             130  .05      16              39   .01      24              14   .01
  7             111  .04      15              34   .01      <<others>>      98   .04
  9              91  .03      17              32   .01
  8              88  .03      19              30   .01

(159)  var[global,*,*,@]
 f = 12217 H =  1.114, p =  .041, H contr. =    .046
   43 cases (11 shown)
  16          10530  .86      11              60   .00      1584            48   .00
  14            362  .03      12              59   .00      64              44   .00
  32            354  .03      3               54   .00      15              42   .00
  1             237  .02      144             48   .00      <<others>>     379   .03

(160)  var[local,*,*,@]
 f =   6685 H =  2.327, p =  .023, H contr. =    .053
SEE ALSO:       f     H      p contr
  (229)     3114 2.078  .011  .022, assign[var[local,*,*,@]]
  (230)      590 2.026  .002  .004, assign[*,var[local,*,*,@]]
  (231)      182 1.963  .001  .001, assignx[var[local,*,*,@]]
  (232)     1445 1.463  .005  .007, call[*,var[local,*,*,@]]
  (233)      840 2.252  .003  .006, dollar[var[local,*,*,@]]
  (234)     1929 0.000  .007 0.000, dot[var[local,*,*,@]]
  (235)       45 0.000  .000 0.000, error[*,var[local,*,*,@]]
  (236)      113 0.000  .000 0.000, ifstmt[var[local,*,*,@]]
  (237)     1909 2.104  .006  .014, list[*,...var[local,*,*,@]]
  (238)       58 0.000  .000 0.000, seqindex[var[local,*,*,@]]
  (239)       50  .242  .000  .000, seqindex[*,var[local,*,*,@]]
  (240)      355 0.000  .001 0.000, uparrow[var[local,*,*,@]]
  (241)      129 1.400  .000  .001, upthru[var[local,*,*,@]]
   34 cases (12 shown)
  14           3135  .47      8               95   .01      3               53   .01
  16           2168  .32      11              94   .01      176             39   .01
  1             231  .03      48              69   .01      <<others>>     269   .04
  15            210  .03      4               65   .01
  32            199  .03      9               58   .01

(161)  var[field,*,*,@]
 f =    428 H =  1.749, p =  .001, H contr. =   .003
SEE ALSO:       f     H      p contr
  (267)     4570 2.548  .015  .039, dot[*,var[field,*,*,@]]
  (268)     2709 3.287  .009  .030, dollar[*,var[field,*,*,@]]
   7 cases
  0             179  .42      1                5   .01      14               1   .00
  16            145  .34      3                5   .01
  32             90  .21      15               3   .01

(162)  var[entry,*,*,@]
 f =   2686 H =  0.000, p =  .009, H contr. =  0.000
  16           2686 1.00

(163)  dot[*,var[*,*,@]]
 f =   5697 H =   .657, p =  .019, H contr. =    .013
```

C.6. *Final Pattern Set -- Sorted by Pattern Number*

```
  15 cases
 0          5256   .92      14              47   .01      10             11   .00
 1            74   .01      4               36   .01      15             11   .00
 8            62   .01      2               35   .01      7               9   .00
 3            53   .01      9               24   .00      11              5   .00
 6            50   .01      5               23   .00      12              1   .00

(164)  dollar[*,var[*,*,@]]
 f =  2754 H =  1.713, p =  .009, H contr. =   .016
  15 cases
 0          1904   .69      12              50   .02      6              12   .00
 2           364   .13      4               31   .01      11             11   .00
 3           186   .07      5               28   .01      7               9   .00
 1            56   .02      8               21   .01      13              6   .00
 14           53   .02      15              21   .01      9               2   .00

(165)  unionx[var[*,*,@]]
 f =   179 H =  2.818, p =  .001, H contr. =   .002
  16 cases
 1            45   .25      5                3   .02      12              2   .01
 3            43   .24      7                2   .01      13              2   .01
 0            32   .18      8                2   .01      6               1   .01
 2            32   .18      9                2   .01      15              1   .01
 4             4   .02      10               2   .01
 14            4   .02      11               2   .01

(166)  addr[var[@]]
 f =   339 H =   .958, p =  .001, H contr. =   .001
  global     210   .62      local          129   .38

(167)  assign[var[@]]
 f =  3885 H =   .752, p =  .013, H contr. =   .010
  local     3114   .80      global         752   .19      field          19   .00

(168)  assign[*,var[@]]
 f =   775 H =   .959, p =  .003, H contr. =   .003
   4 cases
  local      590   .76      entry           28   .04
  global     155   .20      field            2   .00

(169)  assignx[var[@]]
 f =   223 H =   .688, p =  .001, H contr. =   .001
  local      182   .82      global          41   .18

(170)  assignx[*,var[@]]
 f =    30 H =  1.159, p =  .000, H contr. =   .000
  local       20   .67      global           8   .27      entry           2   .07

(171)  call[var[@]]
 f =  6370 H =   .992, p =  .022, H contr. =   .021
  global    4016   .63      entry         2319   .36      local          35   .01

(172)  call[*,var[@]]
 f =  1739 H =   .806, p =  .006, H contr. =   .005
   4 cases
  local     1445   .83      field           33   .02
  global     238   .14      entry           23   .01

(173)  dot[var[@]]
 f =  2941 H =   .939, p =  .010, H contr. =   .009
  local     1929   .66      global        1009   .34      field           3   .00

(174)  dot[*,var[@]]
 f =  5697 H =   .855, p =  .019, H contr. =   .017
   4 cases
  field     4570   .80      entry          171   .03
  global     944   .17      local           12   .00

(175)  dollar[var[@]]
 f =   964 H =   .682, p =  .003, H contr. =   .002
```

C.6. *Final Pattern Set -- Sorted by Pattern Number*

```
 local         840   .87       field         63   .07       global         61   .06

(176)  dollar[*,var[@]]
 f =  2754 H =   .126, p =  .009, H contr. =   .001
 field        2709   .98       global        42   .02       local           3   .00

(177)  error[var[@]]
 f =   128 H =  0.000, p =  .000, H contr. =  0.000
 global        128  1.00

(178)  signal[var[@]]
 f =   164 H =  0.000, p =  .001, H contr. =  0.000
 global        164  1.00

(179)  {relE|relG|relN|relL|relGE|relLE}[var[@]]
 f =   955 H =   .549, p =  .003, H contr. =   .002
 local         853   .89       global        89   .09       field          13   .01

(180)  {relE|relG|relN|relL|relGE|relLE}[*,var[@]]
 f =   403 H =   .998, p =  .001, H contr. =   .001
 global        212   .53       local        191   .47

(181)  return[var[@]]
 f =   337 H =   .692, p =  .001, H contr. =   .001
 local         292   .87       global        29   .09       field          16   .05

(182)  uparrow[var[@]]
 f =   372 H =   .268, p =  .001, H contr. =   .000
 local         355   .95       global        17   .05

(183)  upthru[var[@]]
 f =   132 H =   .156, p =  .000, H contr. =   .000
 local         129   .98       global         3   .02

(184)  {index|dindex|seqindex}[var[@]]
 f =   757 H =  1.317, p =  .003, H contr. =   .003
 global        423   .56       local        264   .35       field          70   .09

(185)  {index|dindex|seqindex}[*,var[@]]
 f =   289 H =   .615, p =  .001, H contr. =   .001
 local         245   .85       global        44   .15

(186)  assign[*,num[@]]
 f =  1078 H =  3.056, p =  .004, H contr. =   .011
  79 cases (11 shown)
 0             431   .40       65535         51   .05       3              10   .01
 1             282   .26       2             26   .02       16              8   .01
 -1             81   .08       17            11   .01       4               5   .00
 16383          58   .05       32767         11   .01       <<others>>    104   .10

(187)  index[*,num[@]]
 f =    85 H =  1.606, p =  .000, H contr. =   .000
   7 cases
 0              54   .64       -65536         2   .02       1788            1   .01
 1              17   .20       -65535         2   .02
 2               8   .09       8              1   .01

(188)  intCC[num[@]]
 f =   168 H =  2.569, p =  .001, H contr. =   .001
  19 cases
 0              79   .47       18             3   .02       -326            1   .01
 1              42   .25       -507           2   .01       7               1   .01
 48              8   .05       -506           2   .01       8               1   .01
 2               6   .04       -325           2   .01       24              1   .01
 97              6   .04       3              2   .01       65408           1   .01
 -505            4   .02       34             2   .01
 65              4   .02       -327           1   .01

(189)  intCO[num[@]]
 f =    78 H =   .477, p =  .000, H contr. =   .000
```

C.6. *Final Pattern Set -- Sorted by Pattern Number*

```
 0             70   .90      1                8   .10

(190)  minus[*,num[@]]
 f =   318 H =  2.446, p =  .001, H contr. =   .003
  26 cases (17 shown)
 1            195   .61      4                6   .02      16             3   .01
 2             34   .11      48               6   .02      35             3   .01
 3             12   .04      107              5   .02      112            3   .01
 63            10   .03      5                4   .01      18             2   .01
 6              8   .03      32               4   .01      61             2   .01
 7              8   .03      65535            4   .01      <<others>>     9   .03

(191)  plus[*,num[@]]
 f =   316 H =  2.807, p =  .001, H contr. =   .003
  26 cases (19 shown)
 1            147   .47      6                5   .02      31             2   .01
 2             58   .18      16               4   .01      48             2   .01
 3             31   .10      32               4   .01      119            2   .01
 4             15   .05      20               3   .01      259            2   .01
 15            10   .03      64               3   .01      512            2   .01
 5              8   .03      256              3   .01      <<others>>     7   .02
 255            6   .02      19               2   .01

(192)  times[*,num[@]]
 f =   292 H =  2.387, p =  .001, H contr. =   .002
  17 cases
 2            133   .46      8                4   .01      11             2   .01
 3             71   .24      12               4   .01      6              1   .00
 16            42   .14      15               4   .01      9              1   .00
 256           10   .03      1                3   .01      20             1   .00
 127            5   .02      10               3   .01      64             1   .00
 4              4   .01      512              3   .01

(193)  register[num[@]]
 f =    87 H =  2.221, p =  .000, H contr. =   .001
   5 cases
 253           29   .33      2               17   .20      1             10   .11
 3             19   .22      254             12   .14

(194)  {relL|relLE|relE|relN|relGE|relG}[*,num[@]]
 f =  2610 H =  4.744, p =  .009, H contr. =   .042
 152 cases (25 shown)
 0            625   .24      4               39   .01      32767         19   .01
 1            254   .10      16              35   .01      2047          17   .01
 16383        245   .09      5               28   .01      11            15   .01
 3            197   .08      10              28   .01      32            15   .01
 -1           164   .06      12              27   .01      66            15   .01
 2            147   .06      9               25   .01      6             14   .01
 65535         93   .04      14              25   .01      63            14   .01
 7             49   .02      8               21   .01      <<others>>   433   .17
 13            45   .02      15              21   .01

(195)  return[num[@]]
 f =   148 H =  2.227, p =  .001, H contr. =   .001
   9 cases
 0             65   .44      16383            9   .06      2047           2   .01
 1             34   .23      -1               7   .05      49151          2   .01
 32767         25   .17      65535            3   .02      2              1   .01

(196)  {assign|assignx}[var[local,@]]
 f =  3296 H =  2.995, p =  .011, H contr. =   .033
  27 cases (12 shown)
 0            868   .26      5              170   .05      10            21   .01
 1            762   .23      6              107   .03      11            17   .01
 2            500   .15      7               79   .02      <<others>>    78   .02
 3            386   .12      8               48   .01
 4            226   .07      9               34   .01

(197)  {assign|assignx}[*,var[local,@]]
 f =   610 H =  2.949, p =  .002, H contr. =   .006
```

C.6. *Final Pattern Set -- Sorted by Pattern Number*

```
   22 cases (11 shown)
  0          186   .30     4            50   .08      8              7   .01
  1          122   .20     5            37   .06      9              5   .01
  2           88   .14     6            25   .04      10             4   .01
  3           59   .10     7            12   .02      <<others>>    15   .02

(198)  call[*,var[local,@]]
  f =  1445 H =  2.275, p =  .005, H contr. =   .011
   18 cases (11 shown)
  0          647   .45     4            53   .04      10             4   .00
  1          363   .25     5            38   .03      9              3   .00
  2          175   .12     7            21   .01      11             3   .00
  3          113   .08     6            15   .01      <<others>>    10   .01

(199)  dot[var[local,@]]
  f =  1929 H =  1.384, p =  .007, H contr. =   .009
   13 cases
  0         1365   .71     5             8   .00      21             2   .00
  1          340   .18     6             5   .00      7              1   .00
  2          115   .06     22            3   .00      9              1   .00
  3           63   .03     8             2   .00
  4           22   .01     20            2   .00

(200)  forseq[var[local,@]]
  f =    84 H =   .963, p =  .000, H contr. =   .000
    4 cases
  0           67   .80     2             5   .06
  1           11   .13     6             1   .01

(201)  list[*,...var[local,@]]
  f =  1909 H =  2.798, p =  .006, H contr. =   .018
   23 cases (11 shown)
  0          578   .30     4           133   .07      8             19   .01
  1          421   .22     5            85   .04      9             14   .01
  2          313   .16     6            47   .02      10            13   .01
  3          227   .12     7            30   .02      <<others>>    29   .02

(202)  minus[var[local,@]]
  f =   199 H =  2.803, p =  .001, H contr. =   .002
   13 cases
  2           49   .25     6            11   .06      9              1   .01
  1           48   .24     5             5   .03      11             1   .01
  0           39   .20     7             4   .02      16             1   .01
  3           24   .12     8             3   .02
  4           11   .06     10            2   .01

(203)  minus[*,var[local,@]]
  f =    90 H =  2.595, p =  .000, H contr. =   .001
   10 cases
  1           33   .37     5             6   .07      7              1   .01
  0           15   .17     9             3   .03      8              1   .01
  2           14   .16     4             2   .02
  3           13   .14     6             2   .02

(204)  plus[var[local,@]]
  f =   321 H =  2.684, p =  .001, H contr. =   .003
   15 cases
  0          102   .32     4             9   .03      16             2   .01
  1           74   .23     6             9   .03      10             1   .00
  3           54   .17     7             5   .02      14             1   .00
  2           38   .12     8             2   .01      18             1   .00
  5           20   .06     9             2   .01      23             1   .00

(205)  plus[*,var[local,@]]
  f =  3027 H =  1.795, p =  .010, H contr. =   .018
   15 cases
  0         1852   .61     5            29   .01      17             2   .00
  1          565   .19     6            27   .01      10             1   .00
  2          278   .09     7            25   .01      12             1   .00
  3          150   .05     8             5   .00      14             1   .00
```

C.6. *Final Pattern Set -- Sorted by Pattern Number*

```
 4             86   .03     9             4   .00     23              1   .00

(206)  times[var[local,@]]
 f =    162 H =  2.598, p =  .001, H contr. =   .001
  11 cases
 0             66   .41     3            10   .06     10              1   .01
 1             30   .19     6             8   .05     15              1   .01
 4             17   .10     7             7   .04     17              1   .01
 2             16   .10     5             5   .03

(207)  {index|dindex|seqindex}[var[local,@]]
 f =    264 H =  3.453, p =  .001, H contr. =   .003
  21 cases (13 shown)
 2             51   .19     5            27   .10     16              7   .03
 7             37   .14     18           12   .05     6               3   .01
 0             36   .14     3            10   .04     28              2   .01
 1             29   .11     9             8   .03     <<others>>      8   .03
 4             27   .10     11            7   .03

(208)  {index|dindex|seqindex}[*,var[local,@]]
 f =    245 H =  2.364, p =  .001, H contr. =   .002
  12 cases
 0             95   .39     4            10   .04     11              2   .01
 1             69   .28     5             6   .02     16              2   .01
 2             35   .14     9             3   .01     7               1   .00
 3             19   .08     6             2   .01     8               1   .00

(209)  {relL|relLE|relE|relN|relGE|relG}[var[local,@]]
 f =    853 H =  2.607, p =  .003, H contr. =   .008
  16 cases
 0            313   .37     6            20   .02     12              1   .00
 1            189   .22     8            12   .01     13              1   .00
 2            134   .16     7            10   .01     16              1   .00
 3             71   .08     9             6   .01     20              1   .00
 4             57   .07     10            3   .00
 5             31   .04     15            3   .00

(210)  {relL|relLE|relE|relN|relGE|relG}[*,var[local,@]]
 f =    191 H =  3.006, p =  .001, H contr. =   .002
  13 cases
 1             46   .24     6            11   .06     11              1   .01
 2             33   .17     5             8   .04     16              1   .01
 0             30   .16     7             6   .03     68              1   .01
 3             29   .15     8             6   .03
 4             17   .09     10            2   .01

(211)  uparrow[var[local,@]]
 f =    355 H =  1.130, p =  .001, H contr. =   .001
   8 cases
 0            281   .79     2            11   .03     5               1   .00
 1             39   .11     4             3   .01     7               1   .00
 3             17   .05     6             2   .01

(212)  upthru[var[local,@]]
 f =    129 H =  2.023, p =  .000, H contr. =   .001
   9 cases
 0             67   .52     3             5   .04     5               1   .01
 1             30   .23     4             5   .04     9               1   .01
 2             15   .12     6             4   .03     11              1   .01

(213)  assign[var[global,@]]
 f =    752 H =  4.743, p =  .003, H contr. =   .012
  95 cases (31 shown)
 1            100   .13     30            6   .01     28              4   .01
 0             85   .11     45            6   .01     32              4   .01
 3             79   .11     63            6   .01     33              4   .01
 2             68   .09     23            5   .01     38              4   .01
 4             61   .08     31            5   .01     40              4   .01
 5             59   .08     35            5   .01     42              4   .01
 6             41   .05     37            5   .01     43              4   .01
```

```
  7          40  .05     41             5  .01     44             4  .01
  27          7  .01     64             5  .01     53             4  .01
  34          7  .01     16             4  .01     <<others>>   107  .14
  24          6  .01     17             4  .01

(214)  call[var[global,@]]
 f =  4016 H =  4.241, p =  .014, H contr. =   .058
  68 cases (25 shown)
  8         916  .23     17           101  .03     26            40  .01
  9         536  .13     18            93  .02     27            35  .01
  10        380  .09     19            82  .02     29            32  .01
  11        285  .07     20            75  .02     28            31  .01
  12        227  .06     21            68  .02     30            28  .01
  13        188  .05     22            58  .01     31            24  .01
  14        147  .04     23            50  .01     32            24  .01
  15        128  .03     24            45  .01     <<others>>   269  .07
  16        113  .03     25            41  .01

(215)  dot[var[global,@]]
 f =  1009 H =  2.251, p =  .003, H contr. =   .008
  16 cases
  0         434  .43     7              7  .01     65             2  .00
  1         214  .21     6              5  .00     26             1  .00
  2         193  .19     59             5  .00     32             1  .00
  3          95  .09     22             3  .00     62             1  .00
  4          27  .03     25             3  .00
  5          16  .02     29             2  .00

(216)  dot[*,var[global,@]]
 f =   944 H =  4.912, p =  .003, H contr. =   .016
  48 cases (40 shown)
  3          97  .10     27            18  .02     2              9  .01
  14         87  .09     29            18  .02     21             9  .01
  0          60  .06     18            17  .02     25             8  .01
  6          51  .05     17            16  .02     5              7  .01
  12         48  .05     28            16  .02     8              7  .01
  22         48  .05     38            16  .02     15             7  .01
  1          46  .05     26            15  .02     36             7  .01
  32         45  .05     16            14  .01     44             7  .01
  4          43  .05     23            14  .01     41             6  .01
  7          27  .03     19            12  .01     33             5  .01
  24         27  .03     20            12  .01     42             5  .01
  31         24  .03     13            11  .01     43             5  .01
  30         23  .02     10            10  .01     <<others>>    17  .02
  45         20  .02     11            10  .01

(217)  signal[var[global,@]]
 f =   164 H =  3.938, p =  .001, H contr. =   .002
  39 cases
  0          42  .26     15             1  .01     60             1  .01
  1          24  .15     17             1  .01     61             1  .01
  2          22  .13     20             1  .01     66             1  .01
  3          10  .06     21             1  .01     67             1  .01
  4           9  .05     26             1  .01     69             1  .01
  5           7  .04     29             1  .01     70             1  .01
  7           5  .03     31             1  .01     74             1  .01
  58          5  .03     34             1  .01     104            1  .01
  6           4  .02     40             1  .01     105            1  .01
  27          3  .02     46             1  .01     112            1  .01
  542         3  .02     47             1  .01     118            1  .01
  16          2  .01     53             1  .01     553            1  .01
  538         2  .01     54             1  .01     554            1  .01

(218)  arraydesc[list[@]]
 f =    80 H =  0.000, p =  .000, H contr. =  0.000
  2          80  1.00

(219)  call[*,list[@]]
 f =  1816 H =  1.085, p =  .006, H contr. =   .007
   4 cases
```

C.6. *Final Pattern Set -- Sorted by Pattern Number*

```
  2          1390   .77     4             121   .07
  3           263   .14     5              42   .02

(220)  casestmt[*,list[@]]
 f =   639 H =  1.208, p =  .002, H contr. =   .003
   12 cases
  1           478   .75     4               7   .01     16              1   .00
  2           116   .18     5               4   .01     17              1   .00
  3            18   .03     9               2   .00     21              1   .00
  6             9   .01     10              1   .00     34              1   .00

(221)  dostmt[*,*,list[@]]
 f =   253 H =  2.208, p =  .001, H contr. =   .002
   10 cases
  2           111   .44     6               7   .03     11              2   .01
  3            62   .25     7               4   .02     12              2   .01
  4            40   .16     8               3   .01
  5            20   .08     10              2   .01

(222)  fextract[list[@]]
 f =   196 H =  1.099, p =  .001, H contr. =   .001
  2           136   .69     1              49   .25     3              11   .06

(223)  ifstmt[*,list[@]]
 f =   598 H =  2.132, p =  .002, H contr. =   .004
   12 cases
  2           314   .53     5              26   .04     11              4   .01
  3           120   .20     7              19   .03     12              2   .00
  4            70   .12     8               8   .01     16              2   .00
  6            27   .05     9               5   .01     10              1   .00

(224)  ifstmt[*,*,list[@]]
 f =   173 H =  2.051, p =  .001, H contr. =   .001
    8 cases
  2            77   .45     5              15   .09     8               1   .01
  3            50   .29     6               7   .04     9               1   .01
  4            20   .12     10              2   .01

(225)  inlinecall[*,list[@]]
 f =   230 H =   .549, p =  .001, H contr. =   .000
  2           203   .88     3              26   .11     5               1   .00

(226)  item[list[@]]
 f =    97 H =  1.190, p =  .000, H contr. =   .000
    5 cases
  2            74   .76     6               6   .06     4               2   .02
  3            12   .12     5               3   .03

(227)  item[*,list[@]]
 f =   425 H =  2.096, p =  .001, H contr. =   .003
   12 cases
  2           215   .51     6              15   .04     10              2   .00
  3            95   .22     7              11   .03     12              1   .00
  4            52   .12     8               3   .01     14              1   .00
  5            27   .06     9               2   .00     42              1   .00

(228)  signal[*,list[@]]
 f =    44 H =  0.000, p =  .000, H contr. =  0.000
  2            44  1.00

(229)  assign[var[local,*,*,@]]
 f =  3114 H =  2.078, p =  .011, H contr. =   .022
   22 cases (11 shown)
  16         1719   .55     8              64   .02     3              28   .01
  14          726   .23     32             64   .02     9              19   .01
  1           284   .09     11             39   .01     7              14   .00
  15           73   .02     5              30   .01     <<others>>     54   .02

(230)  assign[*,var[local,*,*,@]]
 f =   590 H =  2.026, p =  .002, H contr. =   .004
```

C.6. *Final Pattern Set -- Sorted by Pattern Number*

```
   18 cases (12 shown)
  16          340   .58       11              10   .02       4               3   .01
  14          142   .24       3                9   .02       176             3   .01
  1            21   .04       32               9   .02       <<others>>     10   .02
  8            19   .03       5                4   .01
  15           16   .03       10               4   .01

(231)  assignx[var[local,*,*,@]]
 f =    182 H =  1.963, p =  .001, H contr. =   .001
   10 cases
  16           99   .54       8                9   .05       9               1   .01
  14           43   .24       11               3   .02       32              1   .01
  1            13   .07       3                1   .01
  15           11   .06       5                1   .01

(232)  call[*,var[local,*,*,@]]
 f =   1445 H =  1.463, p =  .005, H contr. =   .007
   11 cases
  16          773   .53       11              14   .01       4               2   .00
  14          564   .39       15               7   .00       5               2   .00
  8            47   .03       3                4   .00       13              1   .00
  32           27   .02       6                4   .00

(233)  dollar[var[local,*,*,@]]
 f =    840 H =  2.252, p =  .003, H contr. =   .006
   12 cases
  16          420   .50       304             41   .05       8               8   .01
  32          164   .20       208             19   .02       96              7   .01
  176         109   .13       160             11   .01       64              5   .01
  48           43   .05       128              9   .01       80              4   .00

(234)  dot[var[local,*,*,@]]
 f =   1929 H =  0.000, p =  .007, H contr. =  0.000
  16         1929  1.00

(235)  error[*,var[local,*,*,@]]
 f =     45 H =  0.000, p =  .000, H contr. =  0.000
  16           45  1.00

(236)  ifstmt[var[local,*,*,@]]
 f =    113 H =  0.000, p =  .000, H contr. =  0.000
  1           113  1.00

(237)  list[*,...var[local,*,*,@]]
 f =   1909 H =  2.104, p =  .006, H contr. =   .014
   17 cases (11 shown)
  16          986   .52       32              51   .03       5              31   .02
  14          542   .28       3               43   .02       8              17   .01
  15           77   .04       9               38   .02       7               6   .00
  1            70   .04       11              32   .02       <<others>>     16   .01

(238)  seqindex[var[local,*,*,@]]
 f =     58 H =  0.000, p =  .000, H contr. =  0.000
  16           58  1.00

(239)  seqindex[*,var[local,*,*,@]]
 f =     50 H =   .242, p =  .000, H contr. =   .000
  16           48   .96       3                2   .04

(240)  uparrow[var[local,*,*,@]]
 f =    355 H =  0.000, p =  .001, H contr. =  0.000
  16          355  1.00

(241)  upthru[var[local,*,*,@]]
 f =    129 H =  1.400, p =  .000, H contr. =   .001
    9 cases
  16           99   .77       4                4   .03       8               2   .02
  15           10   .08       9                4   .03       5               1   .01
  3             5   .04       6                3   .02       7               1   .01
```

C.6. *Final Pattern Set -- Sorted by Pattern Number*

```
(242)  dollar[*,var[field,@]]
 f =  2709 H =  2.433, p =  .009, H contr. =   .022
   32 cases (11 shown)
  0          1432   .53      8             73   .03      4             47   .02
  1           370   .14      6             60   .02      5             40   .01
  2           343   .13      9             54   .02      10            23   .01
  3           157   .06      7             48   .02      <<others>>    62   .02

(243)  dot[*,var[field,@]]
 f =  4570 H =  3.441, p =  .015, H contr. =   .053
   30 cases (17 shown)
  0          1032   .23      5            283   .06      26            36   .01
  3           701   .15      7            146   .03      14            28   .01
  4           597   .13      6            127   .03      17            27   .01
  1           588   .13      12            70   .02      19            24   .01
  2           390   .09      11            44   .01      9             23   .01
  8           290   .06      10            42   .01      <<others>>   122   .03

(244)  arraydesc[list[*,...@]]
 f =   160 H =  2.536, p =  .001, H contr. =   .001
   10 cases
  num          56   .35      plus          12   .08      dollar         1   .01
  addr         39   .24      dot            7   .04      base           1   .01
  var          21   .13      div            4   .02
  call         17   .11      register       2   .01

(245)  body[list[*,...@]]
 f =  6617 H =  2.639, p =  .022, H contr. =   .059
   27 cases (11 shown)
  assign     2312   .35      casestmt     313   .05      rowcons       47   .01
  call       1364   .21      dostmt       304   .05      openstmt      31   .00
  return     1207   .18      construct     89   .01      bump          31   .00
  ifstmt      744   .11      fextract      79   .01      <<others>>    96   .01

(246)  call[*,list[*,...@]]
 f =  4263 H =  2.425, p =  .014, H contr. =   .035
   29 cases (11 shown)
  var        1813   .43      dollar       144   .03      minus         32   .01
  num        1216   .29      addr         112   .03      ifexp         24   .01
  dot         447   .10      plus          84   .02      dindex        22   .01
  call        226   .05      str           63   .01      <<others>>    80   .02

(247)  caseexp[*,list[*,...@]]
 f =   120 H =   .495, p =  .000, H contr. =   .000
  item        107   .89      caseswitch    13   .11

(248)  casestmt[*,list[*,...@]]
 f =   982 H =   .363, p =  .003, H contr. =   .001
  item        914   .93      caseswitch    68   .07

(249)  casestmt[*,*,list[*,...@]]
 f =   159 H =  2.228, p =  .001, H contr. =   .001
   12 cases
  call         71   .45      signal         5   .03      bump           2   .01
  assign       49   .31      fextract       3   .02      casestmt       1   .01
  ifstmt       12   .08      construct      3   .02      exit           1   .01
  return        9   .06      dostmt         2   .01      error          1   .01

(250)  caseswitch[*,*,list[*,...@]]
 f =   572 H =  0.000, p =  .002, H contr. =  0.000
  casetest    572  1.00

(251)  casetest[list[*,...@]]
 f =   254 H =  0.000, p =  .001, H contr. =  0.000
  num         254  1.00

(252)  casetest[*,list[*,...@]]
 f =   600 H =  1.863, p =  .002, H contr. =   .004
   16 cases
  call        331   .55      fextract       5   .01      goto           1   .00
  assign        169   .28      dostmt         4   .01      start          1   .00
  ifstmt         46   .08      return         4   .01      stop           1   .00
  vconstruct     12   .02      exit           2   .00      lst            1   .00
  casestmt       12   .02      inlinecall     1   .00
  bump            9   .02      construct      1   .00

(253)  catchphrase[list[*,...@]]
 f =    42 H =  0.000, p =  .000, H contr. =  0.000
  item           42  1.00

(254)  {construct|constructx}[*,list[*,...@]]
 f =   670 H =  2.948, p =  .002, H contr. =   .007
   20 cases (13 shown)
  var           180   .27      dollar        28   .04      index          6   .01
  num           162   .24      call          15   .02      ifexp          5   .01
  unionx        124   .19      addr          14   .02      times          5   .01
  dot            55   .08      constructx    12   .02      <<others>>    13   .02
  <empty>        42   .06      dindex         9   .01

(255)  dostmt[*,*,list[*,...@]]
 f =   828 H =  2.427, p =  .003, H contr. =   .007
   14 cases
  assign        340   .41      dostmt        22   .03      signal         4   .00
  ifstmt        189   .23      fextract      15   .02      inlinecall     3   .00
  call          135   .16      construct     10   .01      openstmt       2   .00
  casestmt       55   .07      catchmark      6   .01      stop           1   .00
  bump           42   .05      label          4   .00

(256)  fextract[list[*,...@]]
 f =   354 H =   .663, p =  .001, H contr. =   .001
  assign        293   .83      <empty>       61   .17

(257)  inlinecall[*,list[*,...@]]
 f =   489 H =  3.016, p =  .002, H contr. =   .005
   17 cases
  num           160   .33      dollar        16   .03      call           5   .01
  var           114   .23      addr          16   .03      dindex         3   .01
  dot            44   .09      minus         12   .02      uminus         2   .00
  uparrow        31   .06      seqindex      12   .02      base           2   .00
  inlinecall     30   .06      index          8   .02      ifexp          1   .00
  plus           27   .06      times          6   .01

(258)  item[list[*,...@]]
 f =   243 H =   .422, p =  .001, H contr. =   .000
    5 cases
  relE          228   .94      relL           2   .01      relG           1   .00
  in             10   .04      lbl            2   .01

(259)  item[*,list[*,...@]]
 f =  1355 H =  2.625, p =  .005, H contr. =   .012
   21 cases (14 shown)
  assign        542   .40      dostmt        28   .02      resume        15   .01
  call          354   .26      exit          25   .02      continue      15   .01
  ifstmt        187   .14      goto          17   .01      construct     13   .01
  casestmt       47   .03      bump          16   .01      vconstruct    13   .01
  return         45   .03      fextract      15   .01      <<others>>    23   .02

(260)  label[list[*,...@]]
 f =   187 H =  2.311, p =  .001, H contr. =   .001
   12 cases
  call           62   .33      fextract       5   .03      label          1   .01
  assign         61   .33      bump           4   .02      exit           1   .01
  ifstmt         38   .20      construct      3   .02      openstmt       1   .01
  dostmt          7   .04      casestmt       3   .02      catchmark      1   .01

(261)  mwconst[list[*,...@]]
 f =   133 H =  0.000, p =  .000, H contr. =  0.000
  num           133  1.00

(262)  openstmt[*,list[*,...@]]
```

C.6. *Final Pattern Set -- Sorted by Pattern Number*

```
 f =   391 H =  2.113, p =  .001, H contr. =   .003
   15 cases
  assign       206   .53      fextract       6   .02      goto          2   .01
  call          88   .23      return         4   .01      start         2   .01
  ifstmt        47   .12      signal         4   .01      dst           1   .00
  casestmt      15   .04      construct      3   .01      lst           1   .00
  dostmt         8   .02      bump           3   .01      catchmark     1   .00

(263)  return[list[*,...@]]
 f =   160 H =  1.636, p =  .001, H contr. =   .001
    9 cases
  var           98   .61      dot            6   .04      div           1   .01
  num           41   .26      call           3   .02      mod           1   .01
  dollar         7   .04      index          2   .01      addr          1   .01

(264)  row[list[*,...@]]
 f =   686 H =   .882, p =  .002, H contr. =   .002
  num          480   .70      str          206   .30

(265)  signal[*,list[*,...@]]
 f =    88 H =  1.680, p =  .000, H contr. =   .001
    5 cases
  num           39   .44      addr          21   .24      call          1   .01
  var           26   .30      dollar         1   .01

(266)  vconstruct[*,list[*,...@]]
 f =    32 H =  0.000, p =  .000, H contr. =  0.000
  unionx        32  1.00

(267)  dot[*,var[field,*,*,@]]
 f =  4570 H =  2.548, p =  .015, H contr. =   .039
   27 cases (19 shown)
  16          2647   .58      10            84   .02      128          44   .01
  14           412   .09      2             75   .02      48           43   .01
  64           302   .07      8             66   .01      9            38   .01
  1            271   .06      11            64   .01      80           26   .01
  32           109   .02      3             51   .01      6            23   .01
  7            108   .02      4             44   .01      <<others>>   32   .01
  15            87   .02      5             44   .01

(268)  dollar[*,var[field,*,*,@]]
 f =  2709 H =  3.287, p =  .009, H contr. =   .030
   33 cases (18 shown)
  16           792   .29      3             73   .03      80           24   .01
  14           562   .21      15            70   .03      13           21   .01
  1            264   .10      12            52   .02      5            20   .01
  2            259   .10      11            41   .02      9            17   .01
  4            209   .08      0             32   .01      <<others>>   35   .01
  32            97   .04      48            32   .01
  8             80   .03      128           29   .01

(269)  bump[@]
 f =   163 H =  1.027, p =  .001, H contr. =   .001
    4 cases
  var          126   .77      dollar        10   .06
  dot           25   .15      uparrow        2   .01

(270)  bumpx[@]
 f =    25 H =   .482, p =  .000, H contr. =   .000
  var           23   .92      dot            1   .04      dollar        1   .04
```

Bibliography

[Abramson63] Norman Abramson, *Information Theory and Coding.* New York: McGraw-Hill, 1963.

[Alexander75] W. Gregg Alexander and David B. Wortman, "Static and Dynamic Characteristics of XPL Programs," *Computer*, vol 8, pp. 41-46, November 1975.

[Ash65] Robert Ash, *Information Theory.* New York: Interscience, 1965.

[Basharin59] G. P. Basharin, "On a Statistical Estimate for the Entropy of a Sequence of Independent Random Variables," *Theory Probability Appl.*, vol. 4, pp. 333-336, 1959.

[Bobrow72] Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, and Raymond S. Tomlinson, "TENEX, a Paged Time Sharing System for the PDP-10," *Communication of the A.C.M.,* vol 15, pp. 135-143, March 1972.

[Clark76] Douglas W. Clark, "List Structure: Measurements, Algorithms, and Encodings", Ph.D. disseration, Carnegie-Mellon University, 1976.

[Clark77] Douglas Clark and C. Cordell Green, "An Empirical Study of List Structure in LISP," *Communication of the A.C.M.,* vol. 20, pp. 78-87, February 1977.

[Cover76] Thomas M. Cover, private communication 1976.

[Deutsch73] L. Peter Deutsch, "A LISP machine with very Compact Programs," *Third International Joint Conference on Artificial Intelligence,* Stanford University, 1973.

[Feller68] W. Feller, *An Introduction to Probability Theory and Its Applications,* 3rd ed., vol 1. New Work: Wiley, 1968.

[Foster71] C. C. Foster and R. H. Gonter, "Conditional Interpretation of Operation Codes," *IEEE Transactions on Computers,* vol. C-20, pp. 108-111, Jan. 1971.

[Gallager68] Robert C. Gallager, *Information Theory and Reliable Communication.* New York: John Wiley and Sons, 1968.

[Geschke77] Charles M. Geschke, James H. Morris, and Edwin H. Satterthwaite, "Early Experiences with MESA", *Communications of the ACM,* to appear.

[Grenander67] Ulf Grenander, "Syntax-Controlled Probabilities," Division of Applied Mathematics, Brown University, Providence, R.I., Internal Report, December 1967.

[Grenander76] Ulf Grenander, *Pattern Synthesis,* New York: Springer-Verlag, 1976.

[Hehner74] Eric C. R. Hehner, "Matching Program and Data Prepresentations to a Computing Environment", Ph.D. disseration, University of Toronto, 1974.

[Hehner77] Eric C. R. Hehner, "Information Content of Programs and Operation Encoding", *Communications of the ACM,* to appear.

[Huffman52] D. A. Huffman, "A Method for the Construction of Minimum Redundancy Codes," *Proceding of the IRE,* vol 40, pp. 1098-1101, September 1952.

[Karp72] Richard M. Karp, Raymond E. Miller, Arnold L. Rosenberg, "Rapid Identification of Repeated Patterns in Strings, Trees, and Arrays," *ACM Symposium on the Theory of Computing* 4, pp. 125-136, May 1972.

[Knuth71a] Donald E. Knuth, "An Empirical Study of FORTRAN Programs," *Software -- Practice and Experience,* vol. 1, pp. 105-133, 1971.

[Knuth71b] Donald E. Knuth, "Optimum Binary Search Trees," *Acta Informatica,* vol 1, pp. 14-25, 1971.

[McKeeman68] W. M. McKeeman and J. J. Horning, "Information Content of Programs," unpublished draft, c1968.

[Nemetz72] T. Nemetz, "On the Experimental Determination of the Entropy," *Kybernetik,* vol. 10, pp. 137-139, 1972.

[Ott67] Gene Ott, "Compact Encoding of Stationary Markov Sources," *IEEE Transactions on Information Theory,* vol. IT-13, pp. 82-86, January 1967.

[Pasco76] Richard C. Pasco, "Source Coding Algorithms for Fast Data Compression," Ph.D. Dissertation, Stanford University, 1976.

[Pfaffelhuber71] E. Pfaffelhuber, "Error Estimation for the Determination of Entropy and Information Rate from Relative Frequencies," *Kybernetik,* vol. 8, pp. 50-51, 1971.

[Shannon48] C. E. Shannon, "A Mathematical Theory of Communication," *Bell System Technical Journal,* vol. 27, pp. 379-423, 623-656, July, October, 1948.

[Shannon51] C. E. Shannon, "Prediction and Entropy of Printed English," *Bell System Technical Journal,* vol. 30, pp. 50-64, 1951.

[Soule74] Stephen Soule, "Entropies of Probabilistic Grammars," *Information and Control,* vol. 25, pp. 57-74, 1974

[Thompson71] Richard A. Thompson, "Compact Encoding of Probabilistic Languages," Ph.D. Disseration, Univ. Connecticut, Storrs, 1971.

[Thompson71a] Richard A. Thompson and Taylor L. Booth, "Encoding of Probabilistic Context-Free Languages," in *Theory of Machines and Computations.* New York: Academic, 1971.

[Wade75] James F. Wade and Paul D. Stigall, "Instruction Design to Minimize Program Size", *Proceedings of the Second Annual Symposium on Computer Architecture,* pp. 41-44, January 1975.

[Wirth71] N. Wirth, "The Programming Language PASCAL," *Acta Informatica,* vol 1, pp. 35-63, 1971.

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304