Compression via Guided Parsing

William S. Evans
Department of Computer Science
University of Arizona

An increasingly common situation in today's internet community is the automatic downloading and local execution of a program obtained from an external source. For example, many web-sites contain "Java Applets" that are programs transmitted to any user who visits the site. The programs must pass the user's security check and then they are executed on his or her machine. The user never looks at the program; its execution is automatic.

The fact that the program is intended for execution and not for the user's view provides interesting and novel opportunities for data compression. The compression may be lossy as long as the reconstructed program's execution is identical to the original. As a simple example, parts of the program that are never executed may be discarded. Any decrease in the program's description size translates directly into a reduction in transmission time. If the savings in transmission time compensates for the time spent in reconstructing a functionally equivalent executable then performance improves.

Another degree of flexibility is in the choice of the specification language for the program. In general, programs are written in some high-level language which may be compiled into a sequence of intermediate representations leading to a final executable or interpretable form. Any of the representations in this sequence may be compressed, some more effectively than others, even though they all describe the same program.

This paper examines the compression of *source code*, the original high-level language representation of a program. We discuss the results (both advantages and disadvantages) of this choice in more detail in later sections, but two intuitive reasons for supposing compressed source code results in the smallest program representation are worth mentioning at this point.

One reason is that later stages in the compilation process seem to add information. For example, a high-level language construct may require a complicated pattern of machine instructions. In a sense, the compiler adds this translation information to the representation. The compiler may also add information about where to store variables or how to allocate registers. Of course, since the algorithms that perform the compilation are fixed (and known to both sender and receiver), no information (in the Shannon sense) is actually added. However, it seems easier for a compression scheme to recognize and exploit program patterns from source code rather than after compilation has perhaps obscured these patterns.

A second reason is the highly structured syntax of the source code. The syntax of most high-level languages can be described by a language specific context-free grammar. Later stages of representation lack such a rigid overall syntax.

The most important contribution of this paper is the manner in which we exploit the rigid grammar of the high-level language. We call the general technique *guided parsing* since it is a compression scheme based on predicting the behavior of a parser when it parses the source code and guiding its behavior by encoding its next action based on this prediction. In this paper, we describe the implementation and results of two very different forms of guided parsing. One is based on bottom-up parsing while the other is a top-down approach.

Bottom-up parsing constructs a pushdown automata (PDA) that recognizes the same language as the context-free grammar. The source code is fed, as a sequence of *tokens*, into the PDA which transitions from state to state pushing and popping items from its stack. One way to represent the source is as the sequence of transitions performed by the PDA with the source as input. Bottom-up guided parsing predicts the next transition based on the current configuration (state and stack contents) of the PDA.

Top-down parsing starts with the start symbol of a grammar and applies grammar rules to expand nonterminals until the result is the input source code. At each step, the parser chooses a grammar rule to expand the leftmost nonterminal in the current derivation. In general, the parser's choice is based on the next token in the source code. Alternatively, we could guide the parser by specifying the actual grammar rule to use. This could result in substantial savings since, typically, there are many fewer rules to expand a given nonterminal than there are tokens. Top-down guided parsing predicts the next grammar rule to apply based on the parser's past performance.

Both methods may be used to compress strings from any language that can be represented with an appropriate context-free grammar. Our particular application is the compression of computer programs expressed in source code.

1 Source Code

We use Pascal and Java as test cases for our guided parsing. Pascal is a mature language with a well-defined syntax. Java is a new language that is popular for specifying programs to be transmitted between computers. Java source code is not the representation that is typically transmitted. A compiled version of the original source, a *class file*, is the transmitted form.[1] The class file contains a sequence of Java bytecodes that are interpreted (rather than executed) by the receiving computer.

Our motivation for compressing source files rather than class files comes from a desire to produce as small a representation of the program as possible. The hope is that over slow transmission lines, the savings in the transmission time of compressed source compensates for the additional cost of converting the source to bytecode by the receiver. The relative sizes of compressed source versus compressed bytecode for our test suite is shown in table 1 and supports our conjecture that source is more compressible.

One major objection to transmitting compressed source is that the receiver obtains a copy of the original high-level source code, thus revealing potentially sensitive algorithms or structures. Class files supposedly obscure this information. However, they are not designed for this purpose and decompilation of a class file back to a source file is possible [13].

2 Context-Free Grammars

Beyond the evidence provided by general purpose compression, source code appears to be a good program representation for compression because of its rigid structure. The syntax of the source code is captured by a context-free grammar that is known to both the sender and receiver.

A context-free grammar is a list of rules. Each rule, denoted $X \rightarrow \alpha$, consists of a left-hand symbol X and a right-hand string of symbols α. The set of symbols is partitioned into nonterminals

[1] Archived and compressed collections of class files and support files called JAR (Java ARchive) files are also used.

and terminals, and only nonterminals may appear as left-hand symbols. There is a special nonterminal S called the start symbol. The grammar specifies a language that consists of all strings of terminals that can be derived from the start symbol. A string ω is a *single-step derivation* of a string γ (written $\gamma \Rightarrow \omega$) if there is a nonterminal X in γ and a rule $X \rightarrow \alpha$ such that replacing the symbol X in γ with the string of symbols α results in the string ω. A string ω is a *derivation* of γ (written $\gamma \overset{*}{\Rightarrow} \omega$) if there is a finite sequence of single step derivations $\gamma = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \cdots \Rightarrow \gamma_k = \omega$. Aho, Sethi, and Ullman [1] provide a more detailed explanation of context-free grammars and parsing in general.

The grammar we use to represent Java syntax is from the Java Language Specification [8]. It contains 103 terminal symbols, 135 nonterminal symbols, and 350 grammar rules. The Pascal grammar[2] is based on the standard Pascal grammar. It contains 68 terminal symbols, 135 nonterminal symbols, and 257 grammar rules.

3 Lexical Analysis

Our source code encoder is based on the first two stages of compilation: *lexical analysis* and *parsing*. Lexical analysis takes as input a string of characters and outputs a sequence of *tokens* that are used by the parser. A token is a label for a set of character strings. For example in Java, the token for represents the single string `for` (a reserved word in Java), while the token intlit represents any integer literal such as `12` or `8`. The character strings that are labeled by a token are called *lexemes*.

Tokens are the terminal symbols of the grammar for the high-level language. So the language specified by the grammar contains sequences of tokens, rather than actual source code. To reconstruct the source code from a token sequence, each token must be replaced by the lexeme that inspired it. If a token represents only a single lexeme then this replacement requires no further information, but for those tokens representing multiple lexemes, enough information to reconstruct the lexeme must be provided.

The current system handles multiple-lexeme tokens by creating a file for each such token. When the lexical analyzer encounters a lexeme for such a token in the source file, it writes the lexeme into the token's file. These token files can be compressed individually using any general purpose compression technique. In section 6.1, we show results for two different compression schemes: Move-To-Front [3, 5] followed by Huffman coding [9], and Gzip (a variant of LZ77 [16] distributed by the Gnu Free Software Foundation). Move-To-Front performs somewhat better than Gzip on sequences of lexemes that exhibit locality of reference, such as identifier names and integer literals.

The lexical analysis phase discards comments and formatting information from the file. Thus, the compression that is performed is technically lossy. Decompression will only result in an equivalent sequence of lexemes; it will not reconstruct the original source. However, this sequence is sufficient for the purpose of compilation into an executable or interpretable form.

4 Parser

Lexical analysis provides the parser with a sequence of tokens. The parser takes these tokens and produces a *parse tree* according to the grammar. A parse tree shows how a token sequence is derived from the start symbol of a grammar. The root node represents the start symbol, and, in general, internal nodes of the parse tree represent nonterminals in the grammar. The children of an internal node representing nonterminal X represent the symbols on the right-hand side of a rule $X \rightarrow \alpha$. In

[2]http://wuarchive.wustl.edu/languages/c/unix-c/languages/pascal/iso-pascal.tar.Z

particular, if $\alpha = \alpha_1\alpha_2 \cdots \alpha_k$ (where α_i is a terminal or nonterminal symbol) then X has children representing α_1 through α_k (from left to right). Leaf nodes represent tokens (terminal symbols), and the tokens at the leaf nodes of a parse tree, reading from left to right, form the input token sequence.

There are two basic methods of creating the parse tree for a given token sequence: bottom-up and top-down parsing. As their names imply, one constructs the parse tree in a bottom-up fashion while the other constructs it top-down. We consider both methods as a means of succinctly representing the token sequence.

5 Guided Parsing

The basic technique to compress a token sequence, using either a bottom-up or top-down parser, is to run the parser without input, but provide information to guide its execution so that its execution, in the absence of input, is identical to its execution with the token sequence as input. Thus the parser produces a parse tree (from which the token sequence can be extracted) without knowledge of the token sequence. The only information that is needed is that which guides the parser's execution.

5.1 Bottom-up

The type of bottom-up parsing we use is called *LR(1) parsing.* The "L" indicates that the input is scanned from left to right. The "R" indicates that the result is a parse tree representing a rightmost derivation of the token sequence. An LR-parser is a pushdown automata and can be automatically generated from any LR-grammar (a restricted class of context-free grammars). See Aho, Sethi, and Ullman for details [1].

Once the pushdown automata is generated, we need only guide its operation in order to produce the token sequence. In particular, we need only specify what transition the PDA should take given that it is in a particular configuration (state and stack contents).

In order to provide this information, we treat the PDA as a context-based model. The context is a configuration or a partial configuration – perhaps the state of the PDA and the top k symbols on its stack. The number of transitions from a state is finite, and which transition is taken is determined by the next input token. Rather than providing the PDA with this token, we specify the transition. As in other context-based methods, the transition may be specified explicitly or encoded. In our case, we encode using context-based adaptive arithmetic coding.

Initially, all transitions from a context (based on the state of the PDA specified in the context) are assumed equally likely and given frequency count 1. The frequency count distribution is used by an arithmetic encoder to encode the transition from the context. When the parser takes the transition, it increments that transition count, thus modifying the distribution for further encoding. See Bell, Cleary, and Witten [2] for a more detailed description of context based arithmetic encoding.

We have created a system that, given a grammar, automatically constructs the encoder and corresponding decoder. The system operates in a manner similar to the way in which Yacc [10] automatically creates a parser (pushdown automata) from a given grammar.

Our system, like Yacc only produces parsers from LALR-grammars (lookahead LR-grammars). In practice, this is not a restriction since most programming languages have LALR-grammars (perhaps because using Yacc is so much easier than constructing parsers by hand).

The system can produce either LALR-parsers or LR-parsers. LALR-parsers have traditionally been preferred to LR-parsers since the number of states in an LALR-parser is considerably fewer than the number of states in an LR-parser. For example, Java's grammar which has 350 rules produces a 624 state LALR-parser, while the same grammar produces a 2,953 state LR-parser. This means that the memory requirements of an LALR-parser are more modest than the corresponding LR-parser. On the other hand, the increased number of states in an LR-parser may provide much better context for prediction. Results for both LALR and LR-parsers are shown in tables 3 and 4.

5.2 Top-Down

Top-down parsing expands nonterminals starting with the start symbol in an attempt to construct a parse tree for the given input. The choice made by the parser at each step is which rule to apply to expand the leftmost nonterminal in the current derivation. Top-down parsing works efficiently (without backtracking) when the rule to apply can be determined by looking at very few tokens of the input (preferably one).

Actually, the manner in which the parse tree is constructed is immaterial to top-down compression. Given a parse tree, a top-down encoding first specifies the rule used to expand the root node (start symbol) in the parse tree. At every subsequent step, it specifies the rule to expand the leftmost nonterminal in the current derivation, that is the leftmost unexpanded nonterminal in the parse tree. The resulting sequence of rules, rather than the sequence of tokens, guides the performance of the top-down parser.

In order to specify the sequence of rules, we again use a context-based approach. The context, in this case, is the nonterminal to be expanded and, perhaps, the first k nonterminals on the path from that nonterminal to the root in the parse tree. Other contexts are possible and may provide good compression, but the nonterminal to be expanded provides very good context by itself. Since each nonterminal is the right-hand side of only a few rules, we need to provide very little information to specify the rule. As in the bottom-up case, we encode this sequence of conditional rule numbers using context-based adaptive arithmetic coding.

Our implementation encodes the parse tree produced by the bottom-up parser. As in the bottom-up case, the encoder and decoder are produced automatically from the grammar.

The results are shown in tables 3 and 4.

6 Results

The Java programs used to test the compression methods discussed in this paper come from four packages. The first is the Java Development Kit 1.1.2[3]. We use 41 Java source files from the JDK written in Java version 1.0.2. (The version 1.1 source files do not conform to the published version 1.0 grammar.) The second is JavaCup[4], a collection of 30 Java files implementing an LALR parser generator. The third is JLex[5], a single Java file implementing a lexer. The fourth is Toba[6], a collection of 41 Java files that translate Java into C.

The Pascal programs in the test suite are: TeX[7] a typesetting program, Ecp[8] an error correcting parser, and Pcom[9] a portable pascal compiler.

[3]http://java.sun.com/

[4]http://www.cs.princeton.edu/ appel/modern/java/CUP/

[5]http://www.cs.princeton.edu/ appel/modern/java/JLex/

[6]http://www.cs.arizona.edu/sumatra/toba/

[7]http://tug2.cs.umb.edu/ctan/

[8]http://www.cs.wisc.edu/ fischer/ftp/tools/

[9]http://www.cwi.nl/ftp/pascal/pcom.p

	raw	Gzip
source	291860	59454 (20%)
lexemes	175153	37225 (21%)
class	169792	74502 (44%)

Table 1: General (Gzip) compression of 41 concatenated source files from JDK, the lexeme sequence, and class files. Numbers are in bytes.

	lexemes	raw	Gzip	Move-To-Front
ident	13730	87089	25371	24511
			29%	28%
strlit	577	6584	4248	4574
			65%	69%
intlit	1579	3769	2197	2180
			58%	58%

Table 2: Compression of lexemes. Numbers are totals over all 41 source files in JDK. The first column is number of lexemes. The last three columns are number of characters.

Table 1 justifies our interest in source code as a potentially compressible representation of a program. After removing comments and redundant whitespace, the source files are approximately the same size as the class files. However, the source files are much more compressible using the general purpose compression tool Gzip.

6.1 Compressing Lexemes

Table 2 compares the Move-To-Front strategy versus Gzip for compressing sequences of lexemes that have been grouped according to token type.

The table does not include Boolean literal (boollit), character literal (charlit), or floating point literal (floatlit) tokens. The average number of such tokens per file in the test suite was less than 8.

Identifier names (ident tokens) are by far the most numerous. Gzip manages to compress these files to, on average, 29% of their original size. Move-To-Front followed by Huffman coding improves this to about 28% presumably because it exploits some locality of reference of identifiers.

The number of string literal (strlit) and integer literal (intlit) tokens per source file is so small that compression of these lexemes is rather poor. Again, locality of reference of integer literals gives Move-To-Front a slight edge over Gzip.

The Move-To-Front (MTF) compression scheme keeps a list of lexemes it has seen so far. On reading the next lexeme from the file, MTF outputs its index in the list and moves it to the top of the list. If the lexeme is not in the list, MTF outputs 0 and the lexeme itself. Thus MTF outputs a sequence of indices and a sequence of lexemes. We Gzip the sequence of lexemes and Huffman encode the sequence of indices. These two files, along with a description of the Huffman code, constitutes the Move-To-Front representation.

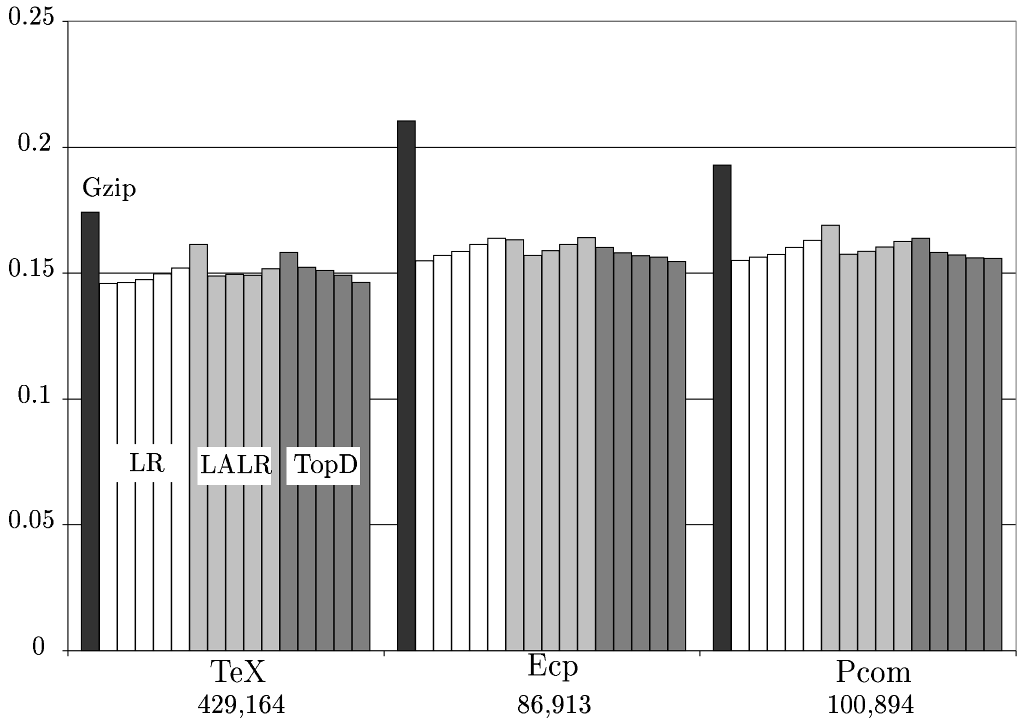

Table 3: Compression of Pascal using bottom-up (LR and LALR) and top-down guided parsing. The bars are grouped by parsing method. From left to right in each group, the context increases. Context is the number of stack symbols (bottom-up) or previous nonterminals (top-down) used to predict. The graph shows percentage of lexeme sequence size (the number below the program name).

6.2 Varying Context Size

Tables 3 and 4 illustrate the small improvement afforded by increased context size. The performance of both the top-down and LALR bottom-up methods improve slightly by increasing the context. The performance of the LR bottom-up method degrades. The problem is that increasing the context size, and thus the number of contexts, decreases the number of times a context arises during parsing. This results in poor prediction on the part of the arithmetic encoder, leading to a decrease in compression.

7 Related and Future Work

We chose to work on source code compression after experimenting with the compression of other program representations [6]. Source code seems to provide a better starting point for compression than the intermediate representations treated in our earlier work. The source code for those representations is in the language C rather than Java, so a direct comparison is still a matter of future work. In the earlier work, we compressed bytecode-like program representations to 20% of their original size.

The compression of an abstract syntax tree representation of a program has been studied in some detail by Michael Franz [7]. This representation is similar to a parse tree representation

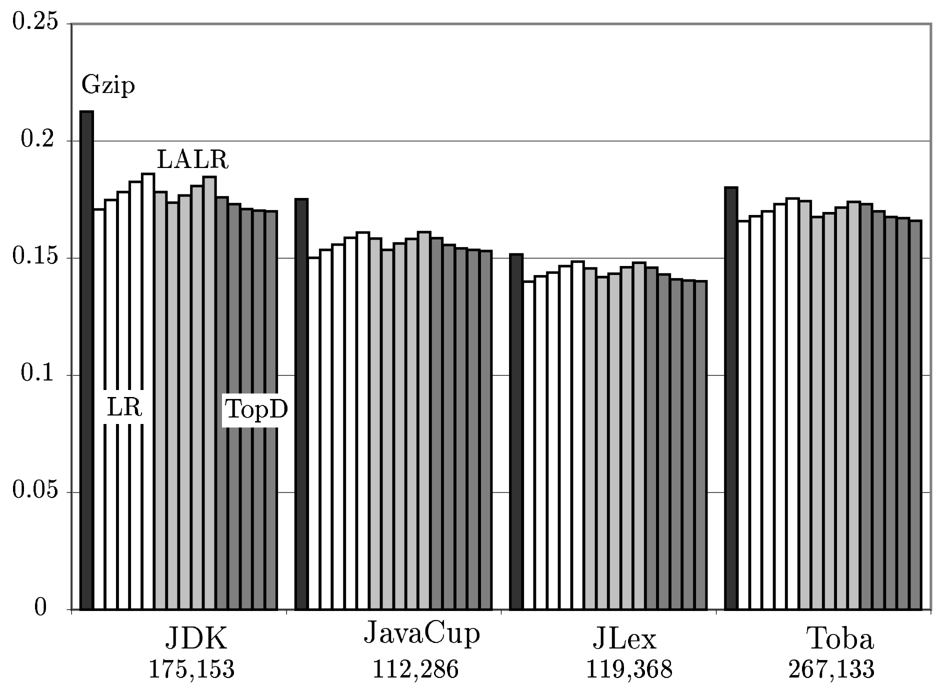

Table 4: Compression of Java using bottom-up (LR and LALR) and top-down guided parsing. The bars are grouped by parsing method. From left to right in each group, the context increases. Context is the number of stack symbols (bottom-up) or previous nonterminals (top-down) used to predict. The graph shows percentage of lexeme sequence size (the number below the program name).

but excludes some syntactic superficialities. Franz compresses these trees using a dictionary based scheme that is similar to LZW [15]. He achieves a compressed size that is 41% of Java bytecode. Our methods yield approximately 18% of Java bytecode (15-17% of the lexeme sequence).

The top-down compression scheme, for single nonterminal context, is similar to a technique used by Robert Cameron to compress Pascal source code [4]. His method relies on the user to annotate the grammar rules with probabilities, rather than having the compression method "learn" these probabilities.

Cameron also touches on the possibility of changing the grammar to make it more effective as a model for compressing source code. He mentions splitting rules to gain more detailed contextual information. These modifications would appear to be subsumed by the use of larger contexts from the parse tree. The general idea, however, seems promising: If one views the purpose of the grammar as a model for compression, what is the best grammar for a given language?

Andreas Stolcke addresses a somewhat related question in his PhD thesis [14]: What is the best grammar for a set of sample strings where best is according to a minimum description length criteria? The description length includes both the description of the grammar and the description of the sample strings given the grammar.

In our case, we know a valid grammar. However, the general idea still applies. We may, given a set of sample strings, search for modifications to the grammar so that the description of the modifications and the description of the sample strings given the modified grammar is minimal. Notice that the modified grammar need not describe the same language as the original. This is an area for future research.

The case when one is given a single string and asked to produce a grammar (with certain properties) for that string with no prior knowledge of a grammar, is discussed by Craig Nevill-Manning [11, 12]. The description length, in this case, is simply the size of the grammar, since the grammar produces only one string.

8 Conclusions

The use of a grammar to exploit syntactic regularity in source code leads to better compression than can be obtained by general purpose compression schemes. The method is made practical by the use of an automatic system to construct the encoder/decoder for a given grammar. This makes the process of designing syntax sensitive compression schemes practical.

9 Acknowledgements

I thank Todd Proebsting for several fruitful discussions and for helping me understand the intricacies of LR and LALR parsing.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools.* Addison-Wesley, 1985.

[2] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression.* Prentice-Hall, 1990.

[3] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):320–330, April 1986.

[4] R. D. Cameron. Source encoding using syntactic information models. *IEEE Transactions on Information Theory*, 34(4):843–850, 1988.

[5] P. Elias. Interval and recency rank source coding: two on-line adaptive variable-length schemes. *IEEE Transactions on Information Theory*, 33(1):3–10, January 1987.

[6] J. Ernst, W. Evans, C. Fraser, S. Lucco, and T. Proebsting. Code compression. In *SIGPLAN '97 Conference on Programming Language Design and Implementation*, 1997.

[7] M. Franz. Adaptive compression of syntax trees and iterative dynamic code optimization: Two basic technologies for mobile-object systems. Technical Report 97-04, Department of Information and Computer Science, University of California, Irvine, February 1997.

[8] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996. http://java.sun.com/docs/books/jls/index.html.

[9] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[10] S. C. Johnson. Yacc – yet another compiler compiler. Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[11] C. G. Nevill-Manning. *Inferring Sequential Structure*. PhD thesis, Department of Computer Science, University of Waikato, 1996.

[12] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *Computer Journal*, 1997. to appear.

[13] T. A. Proebsting and S. A. Watterson. Krakatoa: Decompilation in Java. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies (COOTS)*, 1997.

[14] A. Stolcke. *Bayesian Learning of Probabilistic Language Models*. PhD thesis, Dept. of Electrical Engineering and Computer Science, University of California at Berkeley, 1994.

[15] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[16] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.