On Compression of Parse Trees

Jorma Tarhio
Helsinki University of Technology
Department of Computer Science and Engineering
P.O. Box 5400, FIN-02015 HUT, Finland
Jorma.Tarhio@hut.fi

Abstract

We consider methods for compressing parse trees, especially techniques based on statistical modeling. We regard a sequence of productions corresponding to a suffix of the path from the root of a tree to a node x as the context of a node x. The contexts are augmented with branching information of the nodes. By applying the text compression algorithm PPM on such contexts we achieve good compression results. We compare experimentally the PPM approach with other methods.

1 Introduction

General-purpose text compression works normally at the lexical level assuming that symbols to be encoded are independent or they depend on preceding symbols within a fixed distance. If the source text has some internal structure, the successful modeling of the structure will lead better compression. Traditionally such models has been focused on compression of source programs [7, 13, 17], but also other areas are feasible [8, 16].

In compression of programs, only syntactically correct source programs are acceptable as input, and comments and formatting features are often omitted. A pretty printer is assumed to be available to achieve a readable program from the decoded one. A program can be represented as a parse tree and lists of token classes: identifiers, numbers, strings etc. Each of these components can have a dedicated compression method. E.g. static semantics of the programming language can be utilized in compressing the occurrences of identifiers, because the scope rules limit the visibility of identifiers thus decreasing the coding space. In this paper, we will concentrate on compression of parse trees, which is an important and challenging part of syntactical coding. A parse tree is represented as a sequence of productions, which is the input of our compression scheme.

For a long time the best known method has been based on counts of production alternatives of nonterminals [7]. In this paper we will present a better statistical method, which utilizes contexts of productions.

Stochastic context-free grammars has been used for modeling e.g. natural language. In a stochastic grammar each production has an associated probability, which tells how often that production is applied. An interesting approach is the class of weakly restricted stochastic grammars [2], where probabilities are set for a group of two productions with adjacent nodes in a parse tree. This approach gave us inspiration for defining contexts in trees to be able to efficiently apply the text compression algorithm PPM [3] for syntactical coding. The idea of our compression method without details has been earlier explained in the poster [20]. The implementation of the method was finished in 2000. Recently, similar approaches have been presented by Cheney [8] and Stork et al. [19]. A related method was introduced by Lake [15].

The rest of the paper is organized as follows. In Section 2 we consider sequences of productions as representations of parse trees. In Section 3 we describe the PPM technique in text compression and explain our context model for coding of parse trees using PPM. Some aspects on grammar transformations are discussed in Section 4. Compression results are reported in Section 5 before final discussion in Section 6.

2 Representations of Parse Trees

We introduce basic concepts of parsing following mainly Aho and Ullman [1]. A *context-free grammar* is a four-tuple $G = (N, \Sigma, P, Z)$. The disjoint sets N of *nonterminals* and Σ of *terminals* form the *vocabulary* $V = N \cup \Sigma$. $P \subseteq N \times V^*$ is the set of *productions*, where V^* consists all strings over V. A production $p \in P$ is written $X \to \alpha$, where $X \in N$ is called the *left-hand side* of p and $a \in V^*$ is called the *right-hand side* of p. The symbol $Z \in N$ is the *start symbol* which does not appear on the right-hand side of any production.

(1) $Z \rightarrow i := E$
(2) $E \rightarrow E + T$
(3) $E \rightarrow T$
(4) $T \rightarrow T * F$
(5) $T \rightarrow F$
(6) $F \rightarrow (E)$
(7) $F \rightarrow i$

Figure 1. Productions of G_1.

As an example, we define a grammar G_1 for a small expression language. The productions of $G_1 = (\{Z, E, T, F\}, \{i, :=, +, *\}, P, Z)$ are given in Fig. 1. The terminal symbol i in G_1represents variables and numbers.

The *derivation relation* $\Rightarrow$ is defined as follows. For any $\alpha, \beta \in V^*$, $\alpha \Rightarrow \beta$ if $\alpha = \omega_1 A \omega_2, \beta = \omega_1 \omega_0 \omega_2$ and $A \rightarrow \omega_0$ is a production where $A \in N$ and $\omega_0, \omega_1, \omega_2 \in V^*$. If $\omega_1 \in \Sigma^*$ or $\omega_2 \in \Sigma^*$ we write $\alpha \Rightarrow_{lm} \beta$ or $\alpha \Rightarrow_{rm} b$, respectively. If $\alpha \Rightarrow^* \beta$, we say that the string α *derives* the string β ($\Rightarrow^*$ denotes the reflexive closure of $\Rightarrow$). In particular, if $\alpha \Rightarrow^*_{lm} \beta$ or $\alpha \Rightarrow^*_{rm} \beta$, we say that β is obtained by a *leftmost derivation*, or respectively by a *rightmost derivation* from α. A sequence $p_1, p_2, ..., p_k$ of productions is called a *left parse* of β in grammar G, if β is obtained by a leftmost derivation from Z by applying the productions in the order. A sequence $p_1, p_2, ..., p_k$ of productions is called a *right parse* of β in the grammar G, if β is obtained by a rightmost derivation from Z by applying the productions in the reversed order.

Let M be the total number of productions. We associate a *production number*, i.e. a unique integer in $[1, M]$ with each production. Then we can represent a parse as a sequence of production numbers.

A common representation for a parse is a *parse tree*. The parse tree for a terminal string $w \in L(G)$ is a finite ordered tree in which every vertex is labeled by $X \in V$ or by ϵ. The label of the root is Z. The label of a node u is denoted by $Lab(u)$. If a node u has sons $u_1, u_2, ..., u_m$ such that $Lab(u) = X$ and $Lab(u_i) = X_i, i = 1, ..., m$, then $X \rightarrow X_1 ... X_m$ must be a production in P. The labels of the leaves of the tree for w, concatenated from left to right, form w. The parse tree for $I =$ '$i := i * (i + i)$' with nonterminal labels is shown in Fig. 2.

There is an alternative labeling scheme for internal nodes: $Lab'(u)$ is the production number of the production applied at u. Then a left parse is simply the sequence of the labels of the internal nodes of a parse tree in preorder and and a right parse the sequence of the labels of the internal nodes in postorder.

The parse tree of Fig. 2 is shown with production numbers in Fig. 3. When we traverse the tree in preorder, we get sequence `134576235757` which is the left parse of I. We

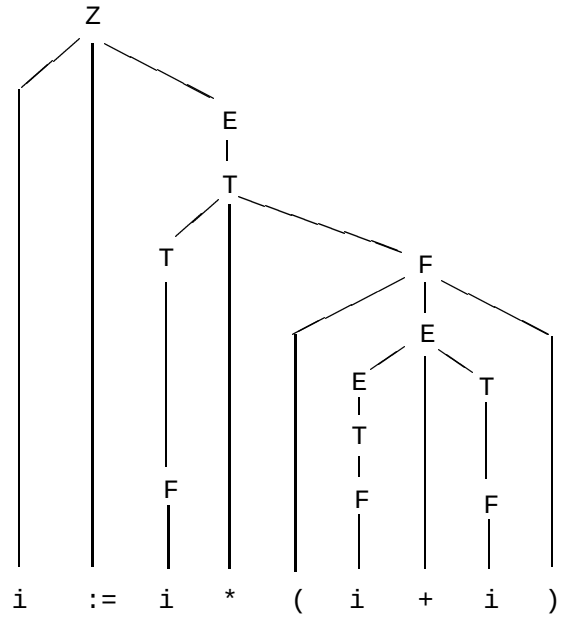

Figure 2. Parse tree of '$i := i * (i + i)$'.

call this representation of a parse tree a global production number sequence or shortly a *GPN sequence*.

It is not necessary to use global production numbers in the left parse, for it is sufficient to express only which *production alternative of the nonterminal* is applied at each node [7]. The reason is that if v is an internal node of a parse tree and if v is not the root, the production applied at the father of v determines the nonterminal associated with v. In addition we apply the following optimization: if a nonterminal has only one production alternative, occurrences of such production are omitted from the sequence. (This could have been done in GPN sequences as well.) We call such productions *insignificant* and other productions *significant*. Each insignificant production could be eliminated from a grammar by a straightforward transformation. If the production alternatives of each nonterminal are numbered starting from one, we get sequence `21221122222` for I. We call this representation a local production number sequence or shortly an *LPN sequence*. It is clear that the use of an LPN sequence leads to better compression than the use of a GPN sequence if no context information is utilized.

The fact that GPN and LPN sequences represent a parse tree implies that coding is actually based on an abstract syntax tree instead of the concrete one.

For low-level encoding, *arithmetic coding* [10, 21] gives the best compression. The idea of arithmetic coding is as follows. Relative intervals of the base interval $[0, 1)$ are assigned adaptively to the symbols of the high-level model according to their frequencies. The encoding of a symbol

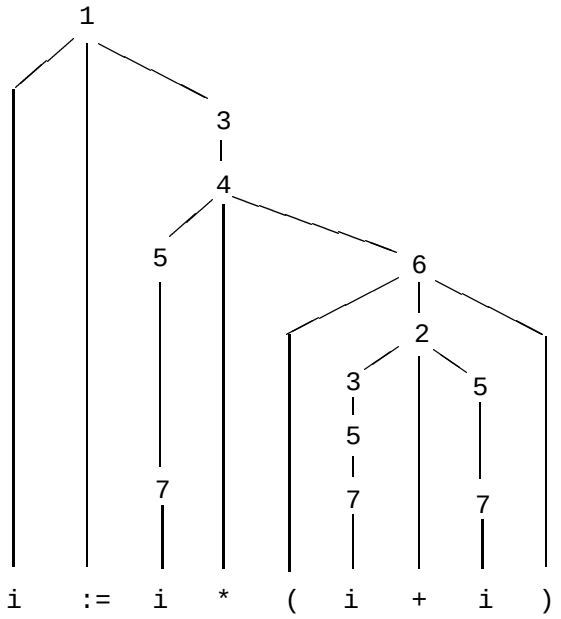

Figure 3. Parse tree with production numbers.

sequence is an interval. The process starts with the base interval $[0, 1)$, and each symbol decreases the interval according to its own interval. After the last symbol, a special end symbol must be encoded to make unambiguous decoding possible.

A natural way to code LPN sequences is to maintain counts of the production alternatives separately for each nonterminal and to use arithmetic coding encode frequencies. This method was first discovered by Cameron [7]. We call it *LPN coding*. An interesting feature of LPN coding is that no end symbol is necessary in arithmetic coding, because the end of a sequence is recognized according to the characteristics of a parse tree. LPN coding will be our reference method with which we compare other coding techniques.

There are also alternative ways to encode LPN sequences. In the simplest model, the sequence is encoded by keeping only one global distribution (the sequence is handled as a text without caring of its semantics). Another possibility is to renumber production alternatives adaptively in order to achieve a skewer distribution: the number of an alternative is maintained to correspond to its position in sorted counts of the productions of the same nonterminal. These models are suitable to on-line encoding. The following two-pass scheme may sometimes improve the compression result. In the first pass, frequencies of productions are compiled. In the second pass, the LPN sequence is encoded as the productions missing from sequence were missing from the grammar. In this scheme we need an additional flag for every production such that its left-hand side nonterminal has at least two production alternatives. If only one of the production alternatives appears in the sequence, occurrences of such a production need not to be encoded at all.

3 Application of the PPM Approach

The online modeling algorithm *Prediction by Partial Matching* (PPM) [3] produces very good compression on text files. PPM maintains statistics concerning which symbols have been seen in which contexts of preceding symbols. Practical forms of PPM are based on the escape mechanism. The k preceding symbols of the symbol to be encoded form the context of order k. There is a separate encoding model for each context. In the case of ordinary text compression, single characters are considered as symbols. Let n be the maximum context length applied. If the next symbol to be processed is present in the model of order n, the symbol is encoded according to that model. Otherwise an escape code is emitted and the model of order $n-1$ is consulted. Because it is demanded that all the possible symbols are present in the model of the lowest order, the search always ends. Normally, the model of order 0 contains counts of symbols without any context and the lowest order -1 is used to encode a new symbol. So the actual code of a symbol is typically a sequence of escape codes followed by a symbol code. In each context all the symbols appeared in longer contexts are excluded from the applied distribution according to the exclusion principle of PPM. Let us assume that the symbol was found in the model of order j. According to the update exclusion principle of PPM, the occurrence of symbol is recorded only at levels $n, n-1, \ldots, j$ but not at lower levels.

Let us consider an example. Let 2 be the maximum context length. Let `c` be the next character to be coded and `ab` the two preceding characters. In this situation the empty context, `ab`, and `b` are possible contexts. If `c` has earlier followed `ab`, `c` is encoded according this context. Otherwise if `ab` has not earlier appeared or `c` has not yet followed `ab`, an escape code is emitted and the algorithm is resumed with the next shorter context `b`. If `c` has earlier followed `b`, `c` is encoded according this context. Otherwise an escape code is emitted and the algorithm is resumed with the empty context. If `c` has appeared earlier, `c` is encoded according this context. Otherwise an escape code is emitted and `c` is encoded according to the context of order -1 which contains all the characters.

The low level coding is normally implemented with arithmetic coding. The scheme works in the same way in decoding.

In the following, we consider three variations of PPM: PPMA, PPMC, and PPMD [3, 10]. The variations differ

Table 1. Symbol and escape probabilities.

Type	p_i	e
PPMA	$c_i/(t+1)$	$1/(t+1)$
PPMC	$c_i/(t+q)$	$q/(t+q)$
PPMD	$(2c_i-1)/(2t)$	$q/(2t)$
PPMA′	$c_i/(t+h)$	$h/(t+h)$

slightly how the symbol and escape probabilities are computed. Let t be the total number of symbols that has been seen in a context. That means that the context has been seen t times. Let q be the number of different symbols seen in the context and let c_i be the count of symbol i. Table 1 shows how the escape probability e and the symbol probability p_i for symbol i are computed. A new variation PPMA′ (see Table 1) is a modification of PPMA with an additional parameter $h > 0$. PPMA′ for $h = 1$ is the same as PPMA. Variation PPMD is defined in Table 1 in an alternative way which is equivalent to the original definition [10].

Next we will consider how the PPM technique can applied to compression of parse trees. We regard the nodes on the path from the root of a tree to node x as the context of node x. More formally, context $C(x)$ of node x consists a sequence of pairs (production, branch) denoted $C(x) = ((p_1, b_1), \ldots, (p_n, b_n))$. Productions (global production numbers) $p_1, \ldots, p_n$ correspond to the n lowest nodes of the path above x. Branches $b_1, \ldots, b_n$ tell the ordinal number of the subtree the path takes at each node. The sequence for the context of order n contains n pairs. Let m be the length of the path. If $m < n$, then a prefix consisting of $n - m$ pairs of the form $(0, 0)$ is inserted to the sequence. In Fig. 3, there are three nodes labeled with 2 or 3 corresponding to productions of nonterminal E. Their contexts of order 2 are $((0, 0), (1, 3))$, $((4, 3), (6, 2))$, and $((6, 2), (2, 1))$.

The lowest order is zero. The symbol code of order 0 resembles LPN coding, but all the productions appeared in longer contexts are excluded according to the exclusion principle. Counts of the production alternatives are maintained separately for each nonterminal. If the maximum context length is zero, then the encoding is the same as LPN coding.

Let n be the length of the maximum context and let the occurrence of production p: $A \to \alpha$ be found at level k. In a normal case this is encoded as a sequence `<esc n>`$\cdots$`<esc k+1><symb k>`, where `<esc m>` is an escape code and `<symb m>` a symbol code at level m according to Table 1. Let us assume that all other production alternatives of A than p are present at levels $n, n-1, \ldots, j$, where $j \geq k$. If $j > k$ holds, it is sufficient to emit only `<esc n>`$\cdots$`<esc j>`, because we know for sure that the next production will be p. If $j = k$ holds, then `<symb k>` is encoded simply with the probability c_i/t without no reserve for an escape in all variations of PPM. According to the update exclusion principle, the occurrence is recorded at levels $n, n-1, \ldots, j$ but not at lower levels.

In the following, our method is called *PPM coding*.

4 Grammar Transformations

Cameron [7] describes a transformation which has an aim similar to our PPM approach. If a nonterminal is present on right-hand sides of several productions and the typical derivations involved with these occurrences are different, one can split this nonterminal to several nonterminals with similar productions. This static approach will slightly improve the coding efficiency without utilizing the dynamic context information.

Stone [18] splits productions of a list structure in order to improve compression. He modifies e.g. productions

$$P \to \epsilon$$
$$P \to aP$$
$$P \to bP$$

to the form

$$P \to \epsilon$$
$$P \to a$$
$$P \to b$$
$$P \to aaP$$
$$P \to abP$$
$$P \to baP$$
$$P \to bbP$$

When choosing advantageous codes for productions in Huffman coding, Stone achieves better compression with the transformed grammar. However, it is easy to see that the transformed grammar yields worse compression than the original when arithmetic coding is used [17, 19]. So the effect reported by Stone is due to the inaccuracy of Huffman coding.

Katajainen et al. [13] suggest to leave out precedence and associative rules of expressions from grammars, because the compressed representation needs not to be semantically correct. This idea can be developed further allowing ambiguous grammars for coding.

The advantage of ambiguous grammars is that we can reduce the number of internal nodes in a parse tree. This is partly due to the fact that the distributions of production alternatives of most nonterminals are very skew in practice. The drawback is that number of production alternatives per nonterminal increases.

Let us consider the grammar G_2 in Fig. 4. Let the grammar G_3 be the same as G_2 augmented with the production

$$
\begin{aligned}
&Z \to L \\
&L \to L\ ;\ A \mid A \\
&A \to i := E \\
&E \to E + T \mid T \\
&T \to T * F \mid F \\
&F \to (E) \mid i
\end{aligned}
$$

Figure 4. Productions of G_2.

$A \to i := i$. Now G_3 is clearly ambiguous. While encoding the input '`i:=n; i:=n; i:=n; i:=n; i:=n`' with LPN coding, G_3 clearly leads to better compression than G_2.

All these transformations are static and so they cannot adapt to changes in input. As an obvious drawback, the size of the grammar may grow, although some of the ambiguous structures are even smaller. Peltola and Tarhio [17] consider a framework of adaptive grammars which are transformed online during encoding a source string.

General parsing based on an ambiguous grammar is slow, though some ambiguous structures may be handled by an LR parser [11]. Therefore it is advisable to perform parsing using an unambiguous grammar and map the result to the ambiguous one. Decoding can be done directly using the ambiguous grammar.

5 Compression Results

We implemented the PPM method described in Section 3. The syntax analysis part was constructed using the language processor generator HLP84 [14]. In order to be able to handle left-recursive grammars, the system applies LALR(1) parsing which produces the right parse of an input. To be able to process the nodes of a parse tree in preorder, we reversed the right parse. The reversal of a right parse corresponds to a left parse in the reversed grammar, for which the right-hand sides of productions have been reversed. In addition to reversals of right parses, we made experiments on left parses and and the results were almost identical in the both ways. Left parses were transformed directly from right parses. The use of a language processor generator makes it is rather easy to modify our compression system for other languages. Instead of HLP84, one could use other parser generators, e.g. Yacc [11].

We ran experiments on parse trees of four Pascal programs described in Table 2. The programs represent different programming styles. Program P1 is a string matching package, P2 is the standard parser used by HLP84, P3 is progp of [3], and P4 is MLTeX 3.1. Our Pascal grammar contains 277 productions of which 226 are significant.

Table 3 summarizes the results of our experiments. We report the compression ratio of each measurement as bits per a significant production. It would have been unfair to include insignificant productions, because they do not actually contain information. If insignificant productions were included, one could reduce the compression ratio just by adding new insignificant productions.

Table 2. Test programs.

Program	*Productions*	*Significant productions*
P1 (string matching)	7008	5412
P2 (parser)	17150	13337
P3 (progp)	26016	19312
P4 (TeX)	269802	198949

Table 3. Compression results (bits/prod).

		P1	P2	P3	P4
(a)	LPN coding	0.992	1.104	1.122	1.000
(b)	GPN/gzip	2.613	2.377	2.060	1.754
(c)	gzip	1.412	1.323	1.140	1.001
(d)	bzip2	1.373	1.339	1.183	0.865
(e)	huff+gzip	1.200	1.140	0.995	0.884
(f)	huff+bzip2	1.184	1.180	1.039	0.818
(g)	PPM	0.855	0.871	0.868	0.756

In Table 3, the method (a) is LPN coding described in Section 2. In the method (b) a GPN sequence is compressed with Gzip. In the methods (c) and (d) a LPN sequence is compressed with Gzip and Bzip2, respectively. Bzip2 is an implementation of the Burrows-Wheeler compression algorithm [6]. In the methods (e) and (f) a LPN sequence is first encoded into a bit string with Huffman so that the production alternatives of each nonterminal has a static model of its own. The result is then compressed with Gzip and Bzip2. The last method (g) is PPM coding.

The results of Table 3 show that PPM is clearly the best method for compressing parse trees. For large programs, static Huffman with Bzip2 is also good—only 8% worse than PPM.

In PPM coding we used the PPMA$'$ model of order 5 for $h = 0.35$. For P4 order 6 gave a slightly better result. The best value of h depends slightly on the input, but good results can be achieved using a fixed value. Experiments with other grammars suggest that the best value of h depends on the grammar.

We also tried variations PPMA, PPMC, PPMD, but PPMA$'$ was clearly the best for $h = 0.35$. Table 4 shows the relative compression results in order 5, where the results of PPMA$'$ are 1 and the smaller values are better. The gain of PPMA$'$ decreases, when the size of the input grows.

Table 4. Relative compression results (PPMA′ = 1.00).

	P1	P2	P3	P4
PPMA′	1.00	1.00	1.00	1.00
PPMA	1.08	1.05	1.04	1.01
PPMC	1.11	1.08	1.07	1.03
PPMD	1.10	1.06	1.04	1.01

In our model, a context is a sequence of items (production, branch). We tested also three other variations for an item: (1) production, (2) (nonterminal, branch), and (3) nonterminal, but all these three approaches gave clearly worse results. The alternative (2) was the second best.

6 Concluding Remarks

It is interesting that PPMA′, our variation of PPMA, showed to be better than PPMC and PPMD, though they are better than PPMA for ordinary texts. The obvious reason is that the number of different symbols in each context is small and distributions are skew when coding a parse tree. When the input becomes longer, the relative gain of PPMA′ gets smaller.

For ordinary texts, PPMA′ is clearly worse than PPMC or PPMD. However, a hybrid scheme applying a local selection of the encoding based on the length and the distribution of a context seems to lead to a minor improvement.

We did not try PPM$*$ [9], which applies unbounded contexts, although many researchers regard it as efficient. However, Bunton [5] reports that PPMC of order 5 is better than PPM$*$ in text compression.

Bloom [4] and Bunton [5] present several techniques to improve the compression result of PPM. Their approaches deal with fine-tuning the escape probabilities. As far we know, Bloom's PPMZ is the most efficient variation of PPM. We have not yet tried PPMZ for parse trees, but we predict a small improvement.

Though our context model is rather natural, it deals only with ancestors of a nodes. A more refined model could record more nodes. One possible addition to the context is the left uncle of a node. The left uncle of node x is the last predecessor of x in postorder which is not a descendant of x. This refinement could work e.g. with the following structure

```
IO_STMT → IO_PROC ( EXPR_LIST )
IO_PROC → read | write
```

because expressions associated with read and write statements are different. Such a structure is an example of transfering a part of syntax analysis to semantic analysis.

One alternative for coding parse trees would to apply general tree compression methods [12]. However, they could hardly lead to better compression, because dedicated methods for parse trees can better utilize the special information of parse trees.

Traditionally syntactical modeling has been used for storing source programs in a compressed form. Mobile computing [19] has created new needs to transfer compressed programs. Hierarchical PPM coding suits well compressing XML documents [8]. Nevill-Manning et al. [16] show how syntactical modeling can be applied to compression of texts in natural language. This opens new possibilities to apply our context model.

Acknowledgments. This work was supported by the Academy of Finland under grant 44449, the National Technology Agency under grant 70003, and the Finnish Cultural Foundation. Veli Mäkinen (University of Helsinki) completed the implementation of PPM coding. Hannu Peltola helped in doing the experiments. A part of the work was done in University of California at Berkeley, University of Helsinki, and University of Joensuu.

References

[1] A. Aho and J. Ullman. *The Theory of Parsing, Translation and Compiling, Vol. I: Parsing.* Prentice-Hall, Englewood Cliffs, N.J., 1972.

[2] R. op den Akker and H. ter Doest. Weakly restricted stochastic grammars. In *Proceedings of COLING '94, 15th International Conference on Computational Linguistics* (ed. M. Nagao), Kyoto, Japan, pages 929–934, 1994.

[3] T. Bell, J. Cleary, and I. Witten. *Text Compression.* Prentice Hall, Englewood Cliffs, N.J., 1990.

[4] C. Bloom. Solving the problems of context modeling. http://www.cco.caltech.edu/∼bloom/papers/ppmz.zip.

[5] S. Bunton. Semantically motivated improvements for PPM variants. *Computer Journal*, 40(2/3):76–93, 1997.

[6] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

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

[8] J. Cheney. Compressing XML with multiplexed hierarchical PPM models. In *Proceedings of DCC '01,*

Data Compression Conference, IEEE Computer Society Press, pages 163–172, 2001.

[9] J. Cleary and W. Teahan. Unbounded length contexts for PPM. *Computer Journal*, 40(2/3):67–75, 1997

[10] P. Howard and J. Vitter. Practical implementations of arithmetic coding. In *Image and text compression* (ed. J. Storer), Kluwer, pages 85–112, 1992.

[11] S. Johnson. Yacc – yet another compiler compiler. Computing Science Technical Report 32, AT&T Bell Laboratories, 1975.

[12] J. Katajainen and E. Mäkinen. Tree compression and optimization with applications. of program files. *International Journal of Foundations of Computer Science*, 4(1):425–447, 1990.

[13] J. Katajainen, M. Penttonen, and J. Teuhola. Syntax-directed compression of program files. *Software–Practice & Experience*, 16(3):269–276, 1986.

[14] K. Koskimies, O. Nurmi, J. Paakki, and S. Sippu. The design of a language processor generator. *Software–Practice & Experience*, 18(2):107–135, 1988.

[15] J. M. Lake. Prediction by grammatical match. In *Proceedings of DCC '00, Data Compression Conference*, IEEE Computer Society Press, pages 153–162, 2000.

[16] C. Nevill-Manning, I. Witten, and D. Maulsby. Compression by induction on hierarchical grammars. In *Proceedings of DCC '94, Data Compression Conference* (ed. J. Storer and M. Cohn), IEEE Computer Society Press, pages 244–253, 1994.

[17] H. Peltola and J. Tarhio. On syntactical data compression. In *Proceedings of the Second Symposium on Programming Languages and Software Tools* (ed. K. Koskimies and K.-J. Räihä), Report A-1991-5, Department of Computer Science, University of Tampere, pages 205–214, 1991.

[18] R. Stone. On the choice of grammar and parser for the compact analytical encoding of programs. *Computer Journal*, 29(4):307–314, 1986.

[19] H. Stork, V. Haldar, and M. Franz. Generic adaptive syntax-directed compression for mobile code. Technical Report 00-42 (Revised version), Department of Information and Computer Science, University of California, Irvine, 2001.

[20] J. Tarhio. Context coding of parse trees. In *Proceedings of DCC '95, Data Compression Conference* (ed. J. Storer and M. Cohn), IEEE Computer Society Press, page 442, 1995.

[21] I. Witten, R. Neal, and J. Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30(6):520–540, 1987.