Working Paper Series
ISSN 1170-487X

Compression by induction of hierarchical grammars

by Craig G. Nevill-Manning, Ian H. Witten & David L. Maulsby

Working Paper 93/9

October, 1993

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

Compression by Induction of Hierarchical Grammars

Craig G. Nevill-Manning
Computer Science, University of Waikato, Hamilton, New Zealand
Telephone +64 7 838–4021; email cgn@waikato.ac.NZ

Ian H. Witten
Computer Science, University of Waikato, Hamilton, New Zealand
Telephone +64 7 838–4246; email ihw@waikato.ac.NZ

David L. Maulsby
Computer Science, University of Calgary, Calgary T2N 1N4, Canada
Email maulsby@cpsc.UCalgary.CA

1 Introduction

Adaptive compression methods build models of symbol sequences. In many areas of computer science, models of sequences are constructed for their explanatory value. In contrast, data compression schemes use models that are *opaque* in that they do not provide descriptions of the sequence that can be understood or applied in other domains. Statistical methods that compress text well invariably generate large models that are not so much a structural description of the sequence as a record of frequencies of short substrings. Macro models replace repeated text by references to earlier occurrences and generally work within a small moving window of symbols so that any implicit model is transient. In both cases the model is flat and does not build up abstractions by combining references into higher level phrases.

This paper describes a technique that develops models of symbol sequences in the form of small, human-readable, hierarchical grammars. The grammars are both semantically plausible and compact. The technique can induce structure from a variety of different kinds of sequence, and examples are given of models derived from English text, C source code and a file of numeric data.

The grammar for a particular text can be compressed using a simple arithmetic code and transmitted in its entirety, yielding a compression rate that rivals the best macro techniques. Alternatively, it can be transmitted adaptively by using it to select phrases for a macro scheme; this gives a compression rate that outperforms other macro schemes and rivals that achieved by the best context models. Neither of these techniques operate on-line: they both involve building a model for the entire sequence first and then coding it as a second stage.

This paper explains the grammatical induction technique, demonstrates its application to three very different sequences, evaluates its compression performance, and concludes by briefly discussing its use as a method of knowledge acquisition.

2 Induction of the grammar

The idea is simple. Given a string of symbols, a grammar is constructed by replacing any repeated sequence of two or more symbols by a non-terminal symbol. For example, given a string S ::= a b c d e b c d f, the repeated 'b c d' is condensed into a new non-terminal, say A:

S ::= a A e A f
A ::= b c d.

Sequences that are repeated may themselves include non-terminals. For example, suppose S was augmented by appending 'b c d e b c d f g'. First, the two occurrences of 'b c d' would be replaced by A, yielding S ::= a A e A f A e A f g, and then the repeated sequence 'A e A f' would be replaced by a new non-terminal, say B:

S ::= a B B g
A ::= b c d
B ::= A e A f.

Because every repeated sequence is replaced by a non-terminal, grammars produced by this technique satisfy the constraint that *every digram in the grammar is unique.*

In order to implement this method efficiently, processing proceeds from left to right, and greedy parsing is used to select the longest repetition at each stage. This will not necessarily produce the smallest grammar possible. To do this would require finding two things: the best set of productions, and the best order in which they should be applied. The latter is called 'optimal parsing', and can be implemented by a dynamic programming algorithm [1]. Parsing is dependant upon the set of productions, or dictionary, and this technique builds a dictionary and performs parsing simultaneously. Unfortunately, the selection of the best dictionary can be shown to be NP-complete, [4, 7]. It is not known how much better this technique would perform if an optimal dictionary could be constructed and optimal parsing performed using this dictionary, but the process would be at least NP-complete.

Although this technique for grammar induction is simple and natural in concept, it is—surprisingly—not obvious how to implement it efficiently. The problem is that at any one time there may be several partially matched rules at different levels in the hierarchy. Suppose that the string in the example above continues 'b c d e b c'. At this point rules A and B are both partially matched in anticipation that 'd' will occur next. However, if the next character is not 'd' both partial matches have to be rolled back and two new rules created, one for the sequence 'b c' and the other for 'A e'. The algorithm must keep track of a potentially unbounded number of partial matches, and when an unexpected symbol occurs it must roll them back and create new rules. To decide whether the next symbol continues the partial matches or terminates them, it is necessary to consider all expansions of the matching rules at all levels. Furthermore, at any particular point in time, the grammar may not satisfy the 'digram uniqueness' constraint, as there are repeated digrams pending the match of a complete rule.

The solution to these problems is to create rules as early as possible, and to extend them when appropriate. As soon as a new symbol is appended to the first rule (S) it forms a new digram with the preceding symbol. If the new digram is unique, nothing further is necessary: all digrams are still unique. If the digram matches another in the grammar, there are three possibilities. It may

1. extend an existing rule,
2. be replaced by an existing two-symbol rule, or
3. result in the creation of a new two-symbol rule.

The first action is performed if the first symbol in the digram is a non-terminal that appears exactly twice in the body of the rules in the grammar. This is because the two places in which the non-terminal appears must be the new digram and the matching digram, and so must be followed by the same symbol. This means that the rule corresponding to the non-terminal can be extended by appending the second symbol in the digram and removing this symbol from the two digrams.

The second action is performed if the matching digram is the same as the entire right-hand side of an existing production. In this case, the non-terminal corresponding to the matching rule replaces the new digram. The third action is performed if the first two do not apply and serves to create new rules.

To illustrate the algorithm, the earlier example is reworked in Figure 1. Whenever a new digram matches an existing one, the action is explained on the right in terms of the three possibilities identified above. A slight complication may occur in the application of the first action. If it results in the matching digram being reduced to a rule of length one, the non-terminal that was just extended is removed from the grammar, and its contents substituted in the two places in which it appeared. This indicates that a rule longer than two symbols has been completely matched. The creation and deletion of rules in this seems inefficient, but it requires little processing, and ensures that the digram uniqueness constraint is obeyed at all times, even during partial matches.

a	S ::= a	
b	S ::= a b	
c	S ::= a b c	
d	S ::= a b c d	
e	S ::= a b c d e	
b	S ::= a b c d e b	
c	S ::= a b c d e b c ⇓ S ::= a A d e A A ::= b c	The repeated 'b c' creates a new rule (case 3)
d	S ::= a A d e A d A ::= b c ⇓ S ::= a A e A A ::= b c d	A has only been used twice, so instead of creating a new rule B ::= A d, A is extended (case 1)
f	S ::= a A e A f A ::= b c d	
b	S ::= a A e A b A ::= b c d	
c	S ::= a A e A f b c A ::= b c d ⇓ S ::= a A e A f B A ::= B d B ::= b c	The repeated 'b c' creates a new rule (case 3)
d	S ::= a A e A f B d A ::= B d B ::= b c ⇓ S ::= a A e A f B A ::= B B ::= b c d ⇓ S ::= a A e A f A A ::= b c d	B has only been used twice, so it is extended (case 1) As A has only one symbol on the right-hand side, it is expanded, and B is removed (see text)
e	S ::= a A e A f A e A ::= b c d ⇓ S ::= a B A f B A ::= b c d B ::= A e	The repeated 'A e' creates a new rule (case 3)
b	S ::= a B A f B b A ::= b c d B ::= A e	
c	S ::= a B A f B b c A ::= b c d B ::= A e ⇓ S ::= a B A f B C A ::= C d B ::= A e C ::= b c	The repeated 'b c' creates a new rule (case 3)
d	S ::= a B A f B C d A ::= C d B ::= A e C ::= b c ⇓ S ::= a B A f B C A ::= C B ::= A e C ::= b c d ⇓ S ::= a B A f B A A ::= b c d B ::= A e ⇓ S ::= a B f B A ::= b c d B ::= A e A	C has only been used twice, so it is extended (case 1) As the rule for A has only one element, it is expanded and C is removed (see text) Now that A has been matched, the digram 'B A' causes B to be extended (case 1)
f	S ::= a B B A ::= b c d B ::= A e A f	The repeated 'B f' extends B (case 1)
g	S ::= a B B g A ::= b c d B ::= A e A f	This is the final grammar

Figure 1: Execution of the induction algorithm

3 Examples of structure discovery

To illustrate the structures that can be discovered using this technique, Figure 2 shows a sample from three grammars that are inferred from different kinds of file. In each case the rules are expanded to demonstrate how symbols are combined to form higher level rules.

3.1 ENGLISH TEXT

Figure 2(a) represents the decomposition of part of the grammar induced from Thomas Hardy's novel *Far from the Madding Crowd*. The darkest bar at the top represents one non-

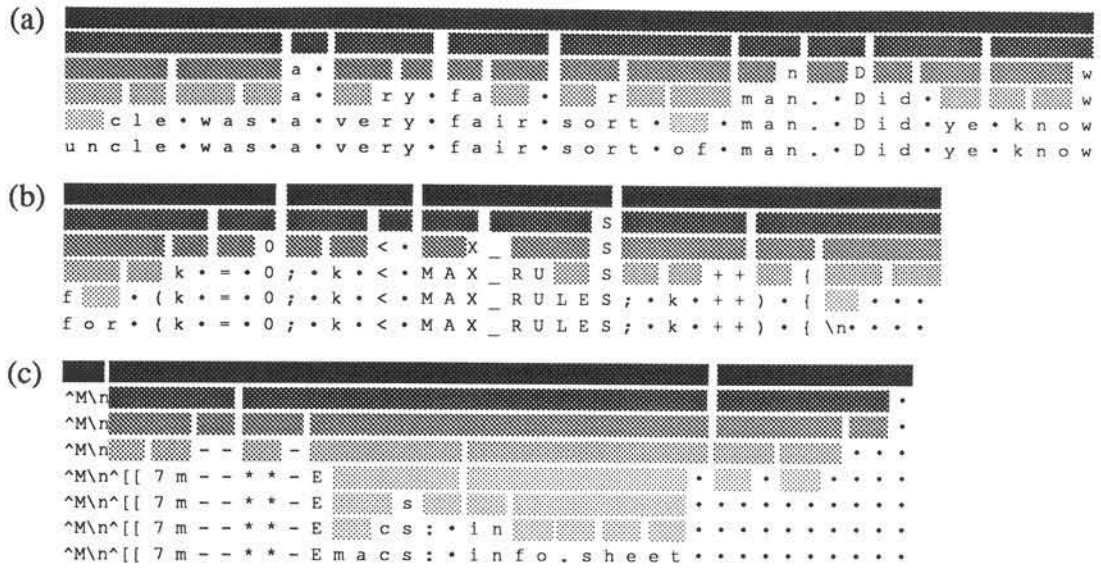

Figure 2: Structure derived from (a) English text,
(b) C source code, and (c) transcript of an Emacs editor session

terminal that covers the whole phrase: 'uncle•was•a•very•fair•sort •of•man.•Did•ye•know'. The existence of this rule indicates that the whole phrase occurs at least twice in the text. Bullets are used to make the spaces explicit.

The top-level rule comprises nine non-terminals, represented by the nine lighter bars on the second line. These correspond to the fragments 'uncle•was•', 'a•', 'very•', 'fair•', 'sort•of•', 'man', '.•D', 'id•ye' and '•know'. These fragments include two phrases ('uncle•was•' and 'sort•of•'), five words, and two fragments '.•D' and 'id•ye'. The rules indicate that some words and word groups in this sequence have been identified as significant phrases. It is interesting that the letter at the start of the sentence 'Did•ye•know' is grouped with the preceding period. Although this does not fit with what we know about word boundaries in English, without being aware of the equivalence of lower- and upper-case letters it seems reasonable that the relationship between the period and the capital letters is stronger than with the rest of the word. If the representation of the text had included capitalization as a separate "upper-case" marker that prefixed capital letters then it would be quite correct to associate it with a preceding period rather than with the letters that followed.

At the next level, the phrase 'uncle•was•' is split into its two constituent words, and 'id•ye' is also split on the space. The other phrase, 'sort•of•', is split after 'sor', which indicates that the word 'sort' has not previously been seen in any other context. The other words are split into less interesting digrams and trigrams. In other parts of the text prefixes and suffixes can be observed being split from the root, for example 'play' and 'ing', 're' and 'view'.

3.2 C Source Code

Figure 2(b) shows the structure of a model formed from a C source program—in fact, in the true spirit of recursive presentation, the program is part of the source code for the actual modeling program. The top level shows four rules which expand to

for•(k•=•0 ;•k•<• MAX_RULES ;•k•++)•{\n••••.

The first and last fragments correspond to the beginning and end of a C 'for' statement with k as the counter variable. The middle two fragments identify a less-than comparison involving k, and a constant MAX_RULES.

At the next level, 'for•(k•=•0' is split into 'for•(k•' and '=•0', indicating that something other than an assignment has followed 'for•(k•' in the past. '=•0' is probably a very common phrase in C. The phrase ';•k•<•' is split into the variable and the operator ';•k•' and '<•', indicating that other comparisons have been made on k. 'MAX_RULES' is broken into 'MAX', '_RULES' and 'S'. The last non-terminal, ';•k•++)•{\n••••', is broken into ';•k•++' and ')•{\n••••'; that is, the incrementing of the counter variable and the end of the 'for' statement combined with the start of the next block.

The third line separates 'for•(', the standard beginning of a C 'for' statement, from 'k•', the specification of the counter variable. The assignment operator '=•' is separated from the value '0'; the operator '++' is separated from ';•k•' and ')•{' is separated from the following white space '\n•••'. The other divisions are less significant. Finally, on the fourth line the keyword 'for' is split from '•(', ';•k•' is divided into the separator ';•' and the variable 'k•, ')•' is separated from '{', and the white space at the end is split up—along with the less interesting division of 'RULE' into 'R', 'U' and 'LE'.

Overall, the divisions make structural sense in terms of the syntax of C, combining terminal symbols into groups on boundaries that coincide with blocks of meaning within the language.

3.3 Transcript of an Emacs Editing Session

Figure 2(c) is a part of a shell transcript, where some time was spend editing a file in the Emacs editor. The top line shows three rules which expand to

```
^M\n      ^[[7m--**-Emacs:•info.sheet•      ••••••••
```

The first rule is the sequence to send terminal cursor to the left-most column of the next line. The second rule draws part of the Emacs status line, and the last rule is a run spaces. This last rule shows how runs of symbols are combined into one rule, and it is clear that some way of expressing runs even more concisely would help the modeling process in some cases.

The next line splits the escape sequence and hyphens '^[[7m--' from the rest of the status line. The split is made just before the '**' which indicate that buffer has not been saved. For the split to me made here, there must have been some saved buffers, where the stars are not present, earlier in the sequence.

On the next line, the VT100 escape code for reverse video text is separated from the hyphens, indicating that reverse video is used elsewhere. Also the '**-' status indicator is split from 'Emacs:•info.sheet•'. Next, '**' is split from the hyphen and 'Emacs:•' is split from the file name 'info.sheet•'.

The technique has identified the new-line-carriage return sequence, a run of spaces, the saved buffer indicator, the reverse-video escape sequence, the word 'Emacs:•' and the file name. Each of these constitute major structural components of the original sequence.

4 Transmission of the grammar

There are two ways in which the induced grammar can be used to transmit a compressed version of the original sequence. The grammar can be transmitted itself, or it can be used to parse the sequence so that the latter can be transmitted adaptively.

4.1 Transmitting the Grammar

The simplest way of transmitting the sequence compactly is to encode the grammar directly. Most compression schemes are adaptive in that the encoder and decoder build a model simultaneously, and the sequence is transmitted relative to the current model. It has been shown that adaptive modeling is at least as good as sending a static model and then sending the sequence relative to that model [1]. The present technique represents an interesting variation: rather than describing some characteristics of the sequence, the model describes the sequence *exactly*. No more information is needed once the model has been sent; the decoder simply expands the first rule in the grammar, and continues recursively.

First, the complete grammar is formed by processing the entire sequence. The grammar can be viewed as a sequence of terminals, non-terminals and end-of-rule markers. This sequence can be coded efficiently using a zero-order model together with an arithmetic coder [10]. Recall that every digram in the grammar is unique. This implies that each symbol is unique in the context of the preceding one, and so models of higher order cannot contribute any useful predictions. Similarly, macro schemes will not achieve any compression, as no phrase appears more than once.

The compression performance of the scheme was measured on the Calgary corpus. This corpus contains fourteen files ranging in content from English text to object programs [1].

The sizes of the files after being compressed by this technique are given in the column labeled 'Grammar' in Table 1. Results are also shown for Unix *compress* [1], a standard macro scheme, LZFG [3], an excellent macro scheme, and a high-performance context scheme, PPMC [6]. The second grammar induction column, labeled 'Adaptive', is discussed below. Compression rates in bits per character are given for each scheme on each file. The average rate is shown at the bottom of the column for each scheme. This gives an indication of compression performance over a broad range of file types.

Compress performs poorly relative to the other methods, but is a practical and widely used macro scheme. Macro methods achieve compression by replacing repeated sequences with references to earlier occurrences, either by explicit reference to a segment of the sequence already transmitted in the case of LZ77 [11], or by referring to a phrase in a list of phrases extracted from the sequence in the case of LZ78 [12]. LZFG is a macro scheme based on the same principle as *compress*, but with improved selection of phrases and correspondingly improved compression performance. It represents the best general macro method. PPMC, on the other hand, is based on statistical context modeling, where the next symbol is predicted based on the preceding symbols. This prediction indicates the amount of information conveyed by the next symbol, and arithmetic coding is used to transmit exactly this number of bits. PPMC achieves the best overall compression rate of any known general-purpose compression method.

The mean compression achieved by transmitting the grammar as above is 19% better than *compress* and only 3.7% worse than LZFG. It is still 23% worse than PPMC, which might be expected from its similarity in approach to the macro schemes. The performance relative to the macro schemes is gratifying given that this is a static model, and one that explains the sequence structure in a useful way.

4.2 ADAPTIVE TRANSMISSION

The grammar inference method can be used in a different way to achieve even greater compression. First, a grammar is constructed for the whole sequence as before. The first rule, S, consists of a sequence of symbols which can be expanded to reproduce the original text. This rule is transmitted from left to right. When a non-terminal appears, there are three possibilities:

- if it has not appeared before, the right-hand side of its rule is transmitted;
- if it has been seen exactly once before, a pointer to the first occurrence is transmitted;
- if it has been seen two or more times before, it is transmitted as a symbol.

		Grammar induction		**Other methods, for comparison**		
File	**Description**	**Grammar**	**Adaptive**	**PPMC**	**LZFG**	**Compress**
bib	bibliography	3.03	2.56	2.12	2.90	3.35
book1	fiction book	3.21	2.86	2.52	3.62	3.46
book2	non-fiction book	2.93	2.51	2.28	3.05	3.28
geo	geophysical data	4.87	4.78	5.01	5.70	6.08
news	electronic news	3.41	2.92	2.77	3.44	3.86
obj1	object code	4.43	3.89	3.68	4.03	5.23
obj2	object code	3.28	2.72	2.59	2.96	4.17
paper1	technical paper	3.52	2.93	2.48	3.03	3.77
paper2	technical paper	3.41	2.91	2.46	3.16	3.52
pic	bilevel image	1.08	0.94	0.98	0.87	0.97
progc	C program	3.51	2.90	2.49	2.89	3.87
progl	Lisp program	2.50	2.02	1.87	1.97	3.03
progp	Pascal program	2.45	1.97	1.82	1.90	3.11
trans	shell transcript	2.21	1.78	1.75	1.76	3.27
	Average	**3.13**	**2.70**	**2.49**	**2.95**	**3.64**

Table 1: Compression performance of several techniques (bits per character)

a	S ::= a	
♦	S ::= a ♦	marker 1
♦	S ::= a ♦ ♦	marker 2
b	S ::= a ♦ ♦ b	
c	S ::= a ♦ ♦ b c	
d	S ::= a ♦ ♦ b c d	
e	S ::= a ♦ ♦ b c d e	
↑(2, 3)	S ::= a ♦ A e A A ::= b c d	creates a rule of length 3 starting at marker 2
f	S ::= a ♦ A e A f A ::= b c d	
↑(1, 4)	S ::= a B B A ::= b c d B ::= A e A f	creates a rule of length 4 starting at marker 1
g	S ::= a B B g A ::= b c d B ::= A e A f	

Figure 3: Incremental transmission of the example sequence

If a non-terminal has not appeared before, then the rule it heads must be transmitted explicitly. At the start of the sequence a marker is transmitted which indicates a position in the sequence that will be referred to when the rule appears for the second time. The markers are numbered implicitly by both encoder and decoder so that they can be referred to later. The cases above for rule S apply equally to the transmission of the contents of the new rule; if the new rule contains any novel non-terminals, then contents of *these* rules will be transmitted, and so on recursively. An order-zero model with arithmetic coding is used to transmit both symbols and markers.

If the non-terminal has appeared exactly once before, it is communicated simply by specifying the number of the marker that was transmitted on its first occurrence, together with the length of the rule's contents. This is similar to LZ77, which represents repeated sequences by referring to earlier occurrences. Markers are deleted once they are used, and the implicit marker numbers are adjusted accordingly. Given a certain number of markers which have been transmitted but not used, say k, the number of bits in which a marker number will be transmitted is $\log_2(k)$. The length of the new rule is transmitted using an adaptive arithmetic code, using fewer bits to transmit shorter lengths.

Once a non-terminal has been seen twice, the rule it heads has been completely specified with a marker and a length and so can be fully reconstructed by the decoder. It is now sufficient to transmit the non-terminal as an ordinary symbol. This is similar to LZ78, where the list of phrases is the set of rules identified so far. Again an order-zero arithmetic code is used.

Figure 3 shows how the earlier example would be represented for transmission. On the left is what is actually coded, and on the right is the grammar built up by the decoder. The symbol ♦ represents a marker, and markers are unparameterized, although both encoder and decoder refer to them by number. The symbol ↑ represents a pointer and has two parameters, marker number and length. It causes the creation of a rule, and rules are labeled (by convention) A, B, C, ... so that they can be referred to later on. Although it does not occur in the example, subsequent references to the non-terminals A and B are transmitted as simply 'A' and 'B'.

This technique has two advantages over other macro schemes. First, because the sequence up to the current point contains non-terminals in place of longer terminal sequences, it is shorter than the equivalent sequence that LZ77-based schemes refer to. As there are fewer symbols sent, the proportion of markers is higher, and the number of bits needed to specify the marker is correspondingly smaller. Also, given that the sequence is compressed, the lengths of the rules can be transmitted in fewer bits. Second, because the rules are chosen to be as long as possible, and all rules are used, there are fewer rules than there are phrases in most LZ78 schemes, so the rule number can be transmitted in fewer bits. In many LZ78 techniques, phrases are extended by only one symbol at a time, so rules grow slowly and prefixes of long phrases may only be used once.

The compression performance of this technique is given in the column labeled 'Adaptive' of Table 1. It achieves a mean compression rate of 2.7, which is 9.3% better than that of LZFG. It represents an improvement of 13% over sending the grammar using an order-zero model, and is only 8.4% worse than PPMC. For file *geo*, the grammar induction methods both outperform LZFG and PPMC. For file *pic*, the grammar induction method outperforms PPMC when adaptive transmission is used. However, PPMC is best on all the other files. Grammar induction using adaptive transmission wins out over LZFG for most files, the exceptions being the image, the programs, and the shell transcript.

The superior compression performance of the new method (using adaptive transmission) over LZFG, may be partly due to the use of arithmetic coding to encode the numbers and symbols in the exact fractional number of bits dictated by the frequency models. The use of arithmetic coding extracts a penalty in execution time, and to rectify this an integral-bit coding technique could be considered instead. However, experiments with such methods have not been conducted.

5 Application to knowledge acquisition

One of the key advantages of this modeling technique is that it is capable of identifying interesting structure in diverse sequences, and presenting the structure in a form that is easily understood and readily applied to problems in other research areas. Specific applications for this technique are still being investigated, and four of the more promising possibilities are briefly presented here.

5.1 Programming by Demonstration

The grammar modeling technique was originally conceived as a way of modeling a sequence of user actions, in order to induce a program that would perform the same actions in a different context automatically. This process is called *programming by demonstration*, and is a burgeoning area of research in human-computer interaction [9]. The ability of the new method of grammar inference to identify repeated sequences of actions, and to abstract these into higher level 'tasks,' simplifies the identification of control structures such as loops or branches. For example, a loop appears as a repeated non-terminal at some level in the grammar. Moreover, if user actions are only available at a very low level of abstraction, this method can identify more interesting, meaningful actions as non-terminals higher in the hierarchy of rules.

5.2 Analysing Natural Language

The identification of words, phrases, suffixes, prefixes and roots of natural language text could form a basis for the analysis of language without a priori assumptions of the nature and morphology of words. Furthermore, the inclusion of a small amount of domain knowledge, such as the role of white space as a word separator, may improve the accuracy with which this technique identifies significant structural features of text. Leaving written languages aside, it could also help in the identification of words and phrases in new languages where only a phoneme-level transcript is available, grouping phonemes together into meaningful utterances.

5.3 Characterising Music

Repetition of note phrases, and recombination of phrases at various levels, is fundamental to musical composition—even at the simplest level of the ubiquitous verse, chorus, verse

structure. It has been shown that there is considerable redundancy in note sequences, and this has been successfully captured by a statistical modeler [8]. The statistical approach is, however, unable to identify the recursive and iterative structure that humans recognise in music [5]. This technique may provide a way of identifying these regularities.

5.4 DNA Sequences

Molecular analysis of DNA strands produces DNA sequences in symbolic form. Each strand consists of a sequence of nucleotides, which can be one of four types: adenine, cytosine, guanine and thymine. This technique could offer a way of extracting some of the structure from DNA sequences to offer a higher-level view of the patterns of nucleotides [2].

6 Conclusion

Data compression takes advantage of regularities in a symbol sequence to reduce its size. Researchers in machine learning and artificial intelligence are also interested in identifying the structure of sequences. The technique for inferring hierarchical grammars serves both purposes. The grammars induced from a variety of different sequences correspond with the structure that we would expect, and sometimes suggest novel relationships that offer insight into the nature of the sequence and the effect of notation. The compression performance of methods based on the induced grammar indicate that the models, while readable and semantically plausible, also approach the best compression techniques in their predictive ability.

7 Acknowledgments

We gratefully acknowledge Tim Bell for helping us to develop our ideas. This work is supported by the New Zealand Foundation for Research, Science and Technology.

8 References

[1] Bell, T.C., Cleary, J.G. and Witten, I.H. (1990) *Text Compression*. Prentice Hall, Englewood Cliffs, NJ.

[2] Chan, P.K. "Machine Learning in Molecular Biology Sequence Analysis," *CUCS-011-91*, Department of Computer Science, Columbia University.

[3] Fiala, E.R., Greene, D.H. (1989) "Data compression with finite windows," *Communications of the ACM, 32*(4) 490–505

[4] Fraenkel, A.S. Mor, M. and Perl, Y. (1983) "Is text compression by prefixes and suffixes practical?," *Acta Informatica*, (20) 371–389.

[5] Manzara, L.C., Witten, I.H., James, M., 1992 "On the entropy of music: an experiment with Bach Chorale melodies," *Leonardo Music Journal 2*(1) 81–88

[6] Moffat, A. (1990), "Implementing the PPM data compression scheme," *IEEE Transactions on Communications COM-38*(11), 1917–1921

[7] Storer, J.A. and Szymanski, T.G. (1978) "The macro model for data compression," *10th Annual Symposium of the Theory of Computing*, 30–39.

[8] Conklin, D., Witten, I.H. (1990), "Prediction and Entropy of Music," *Technical report 91/457/41* (Calgary: Department of Computer Science, Univ. of Calgary).

[9] Witten, I.H., Mo, D. (1993) "TELS: Learning text editing tasks from examples," *Watch what I do: programming by demonstration*, A. Cypher (Ed.), MIT Press, Cambridge, Massachusetts, 182–203

[10] Witten, I.H., Neal, R., Cleary, J.G. (1987) "Arithmetic coding for data compression," *Communications of the ACM 30*(6), 520–540

[11] Ziv, J., Lempel, A. (1977) "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory IT-23*(3), 337–343

[12] Ziv, J., Lempel, A.(1978) "Compression of individual sequences via variable-rate coding," *IEEE Transactions on Information Theory IT-24*(5), 530–536