Inferring Sequential Structure

Craig G. Nevill-Manning

This thesis is submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Computer Science at the University of Waikato.

May 1996

Abstract

Structure exists in sequences ranging from human language and music to the genetic information encoded in our DNA. This thesis shows how that structure can be discovered automatically and made explicit. Rather than examining the meaning of the individual symbols in the sequence, structure is detected in the way that certain combinations of symbols recur. In speech and text, these repetitions form words and phrases. They can be concisely represented by a hierarchical context-free grammar, where each repetition gives rise to a rule. As well as exact repetitions, sequences often exhibit branching and looping structure. These can be inferred and visualised as an automaton.

We develop simple, robust techniques that reveal interesting structure in a wide range of real-world sequences including music, English text, descriptions of plants, graphical figures, DNA sequences, word classes in language, a genealogical database, programming languages, execution traces, and diverse sequences from a data compression corpus. These techniques are useful: they produce comprehensible visual explanations of structure, identify morphological units and significant phrases, compress data, optimise graphical rendering, infer recursive grammars by recognising self-similarity, and infer programs.

Acknowledgments

First and foremost, I would like to thank Ian Witten, my supervisor. It is impossible to enumerate the ways in which he has supported me throughout writing this thesis, but I shall attempt tonevertheless.

Ian has taught me, by word and by example, what it means to do research—academically, organisationally and socially. His enormous energy for performing research, and especially for instigating collaboration between graduate students and faculty around the globe, is astonishing. Many of the acknowledgments below testify to his constant enthusiasm for introducing people to each other. His grasp of the broader picture—of distinguishing the significant from the trivial, and expansive knowledge of many subjects—has been enormously beneficial in planning my research directions. His insistence on allowing me to find and define my own directions has been a source of both frustration ('just tell me what I should do!') and learning.

Ian was also a great help financially, offering research and teaching assistant work throughout my degree. However, he had an impeccable sense of timing, and it is not entirely coincidental that the submission of my thesis occurred a mere three months after the expiry of my assistantship. In 1994, Ian arranged for my wife, Kirsten, and me to follow him to Calgary for six months. This time was immensely productive and enjoyable. It has been good to spend out of work time with Ian, his wife Pam, Nikki and Anna: at dinner parties and barbeques, walking in the bush, playing jazz at the Witten home, and sailing on their yacht. Thanks for making us feel like part of the family. Thanks, Ian, for your friendship, and your infectious laughter.

Kirsten was my closest companion during my thesis. She is the one who agreed to move to Hamilton, and to support me emotionally, financially and practically. It is her psychology training, I am sure, that helped her to choose the right blend of positive and negative reinforcement, and she is the second most relieved person in the world that the thesis is finished. I look forward to spending many years together sharing the fruits of our labour. Thanks also to my wider family for their support and encouragement over many decades.

The Department of Computer Science and the School of Computing and Mathematical Sciences have gone beyond the call of duty in supporting my work. Thanks to Lloyd Smith, Geoff Holmes and Mark Apperley who were heads of department at various times during my degree, and provided me with assistantship positions, travel funds to attend international conferences, and a superb research environment. Ian Graham, dean of the school, funded an airfare to Calgary, as well as a scholarship in the last few months of writing, both of which were invaluable to the researching and writing process. Thanks to all of the staff of the department for numerous discussions over coffee, and constant needling with the 'forbidden question' near the end. Thanks to John Cleary for reading the first full draft, and raising some intriguing issues. Thanks to Sally-Jo Cunningham for her support throughout, and the best departmental parties I think I shall ever partake in!

At the University of Calgary, David Maulsby was a great collaborator and example. He provided the germ for this thesis when he was in the final throes of his. I hope I can provide the same support for other students in the same position. Thanks to Camille Sinanan for her hospitality, organisational skills, and skilled cross-country skiing instruction. I have enjoyed working with Przemek Prusinkiewicz, who provided a fascinating application area in artificial life. Moving south of Calgary, I am also thankful to Dan Olsen from Brigham Young University in Provo, Utah for working with me on genealogical databases. Thanks also to Tim Bell from the University of Canterbury, Christchurch, New Zealand for his excellent supervision of my undergraduate project, for his continued interest, and for introducing me to Ian Witten.

Despite the length of these expressions of gratitude, I have only scratched the surface of the support that I have had during my thesis. There is one last group, the largest, who I will thank as a nameless throng. They are the ones who, day or night, rain or shine, weekday or weekend, would expectantly demand: "Is it done yet?"

Well, yes it is.

Table of Contents

Abstract iii

Acknowledgments v

List of Tables xi

List of Figures xiii

1. Introduction 1

1.1 Motivation 1

1.2 Thesis statement 3

1.2.1 Existence of structure 3

1.2.2 Detection of structure 5

1.3 Some examples 7

1.4 Contributions 11

1.5 Thesis structure 12

2. Background 15

2.1 Sequence modelling 16

2.1.1 Techniques 17

2.1.2 Applications 23

2.2 Grammatical inference 26

2.2.1 Techniques 26

2.2.2 Applications 31

2.2.3 Single versus multiple sentences 32

2.3 Machine learning and mdl 33

2.3.1 Machine learning vs sequence modelling 34

2.3.2 The minimum description length principle 35

2.4 Data compression 38

2.4.1 The relationship between learning and compression 39

2.4.2 Dictionary techniques 41

2.5 Linguistic segmentation 43

2.6 Human sequence learning 45

2.7 Summary 47

3. Forming a hierarchical grammar 49

3.1 The algorithm 51

3.1.1 Digram uniqueness 53

3.1.2 Rule utility 55

3.2 Implementation issues 55

3.2.1 Storing and manipulating the grammar 56

3.2.2 Maintaining rule utility 58

3.2.3 Maintaining digram uniqueness 59

3.2.4 Transient rules 62

3.3 Computational complexity 64

3.4 Exploring the extremes 66

3.5 A unifying representation 70

3.5.1 Enumerating repetitions 71

3.5.2 Forming a hierarchy 73

3.6 Summary 76

4. Improving the grammar 77

4.1 Preview of results 78

4.2 Inferring L-systems from example strings 79

4.2.1 L-systems 80

4.2.2 Comparing L-systems with Sequitur's output 83

4.3 Domain knowledge 85

4.3.1 Restricting rules using domain knowledge 86

4.3.2 Results of the knowledge based restriction 88

4.4 Retrospective reparsing 89

4.4.1 The reparsing technique 89

4.4.2 Reparsing based on local optimality 90

4.4.3 Oblivious reparsing 93

4.4.4 Evaluation 95

4.5 Summary 97

5. Generalising the grammar 99
5.1. Inferring recursion from self-similarity 100
5.1.1 Unification as generalisation 101
5.1.2 Implementation of the generalisation 104
5.2 Reconstructing a program from an execution trace 106
5.2.1 Identifying procedures 109
5.2.2 Comments on the procedure identification algorithm 113
5.2.3 The effect of recursion 115
5.2.4 Summary of procedure identification 118
5.3 Constructing automata from other sequences 118
5.3.1 Inferring branches and loops 119
5.3.2 Problems with inferring non-deterministic structure 123
5.3.3 Justifying inferences by the mdl principle 125
5.4 Detecting structure in semi-structured text 127
5.4.1 The genealogical database 127
5.4.2 Learning the structure of the genealogical database 128
Manual generalisation 129
Interpreting the grammar 130
Automatic generalisation 131
5.5 Summary 134

6. Data Compression 135
6.1 Encoding sequences using Sequitur 136
6.1.1 Sending the grammar textually 137
6.1.2 Sending symbols based on frequency 137
6.1.3 Sending rules implicitly 138
6.1.4 Sending the sequence incrementally 141
6.2 Compression performance 142
6.3 Compression and grammar size 144
6.4 Compressing semi-structured text 146
6.5 Summary 150

7. Applications 151
7.1 Natural language 152
7.1.1 Structure identification in large text sequences 152
7.1.2 Domain knowledge and reparsing 155
7.1.3 Segmentation of language 157
7.1.4 Sentence structure 162
7.2 Improving rendering performance 166
7.3 Music 169
7.4 Phrase identification 172
7.4.1 Computer-assisted abstracting 173
7.4.2 Query formulation for full-text retrieval systems 174
7.5 Macromolecular sequences 177
7.6 Summary 179

8. Conclusions 181
8.1 The techniques 181
8.2 The applications 184
8.3 Future work 188

References 190

List of Tables

Table 2.1 Characterisation of related fields 15
Table 5.1 Predictions based on part of the GEDCOM database 133
Table 6.1 Compression of *book1* by several schemes 138
Table 6.2 Performance of various compression schemes 143
Table 6.3 Compression of context-sensitive L-system output 144
Table 6.4 Compression rates of various schemes on the genealogical data 147
Table 7.1 Statistics on the text of the Bible in three languages 152
Table 7.2 Statistics for grammars inferred from the Bible 153
Table 7.3 Application of domain knowledge and reparsing to text 156
Table 7.4 Compression of macromolecular sequences 178

List of Figures

Figure 1.1 Hierarchies for Genesis 1.1 7
Figure 1.2 Inference of hierarchies from L-system output 8
Figure 1.3 Inference of a program from a trace of its execution 9
Figure 1.4 Detecting the structure of a C *switch* statement 10
Figure 1.5 Illustration of matches within and between two chorales 11
Figure 2.1 Structure detection by enumeration of automata (after Gaines, 1976) 18
Figure 2.2 The finite context prediction mechanism for PUSS and FLM (after Cleary, 1980) 20
Figure 2.3 Inferring a program from an execution trace (after Gaines, 1976) 22
Figure 2.4 Inferring an automaton by insertion and deletion of cycles 27
Figure 2.5 Automata to illustrate *k*-reversibility 29
Figure 2.6 One-reversible automaton for the English auxiliary system (after Berwick and Pilato, 1987) 31
Figure 2.7 Inferring a customised data entry form from input (after Schlimmer and Hermens, 1993) 32
Figure 2.8 Inferring a decision tree from data 34
Figure 2.9 Dictionary compression schemes 41
Figure 3.1 Example sequences and grammars that reproduce them 51
Figure 3.2 Operation of the two grammar constraints 54
Figure 3.3 The entire SEQUITUR algorithm 55
Figure 3.4 The four operations required to build the grammar 57
Figure 3.5 Routines for creating and deleting symbols, incorporating maintenance of rule utility constraint 58
Figure 3.6 Updating the digram index as links are made and broken 60
Figure 3.7 Routines for updating the digram index when symbols are linked 61
Figure 3.8 Outline of the SEQUITUR algorithm 61
Figure 3.9 Two outcomes for the rule creation postponement approach, after seeing the string *abcdebcdabcdebc* 62
Figure 3.10 Reductions in grammar size for the three grammar operations 65
Figure 3.11 Some extreme cases for the algorithm 67
Figure 3.12 Growth rates on English text 69
Figure 3.13 Forming a matrix of repetitions 72
Figure 3.14 Illustration of possible rules 74
Figure 3.15 Effects of choosing between overlapping rules 75
Figure 4.1 Summary of results for the chapter 79

Figure 4.2 The LOGO language ...80
Figure 4.3 Derivation of an L-system ...82
Figure 4.4 Another plant image ...83
Figure 4.5 The grammar produced by SEQUITUR from the sequence in Figure 4.3b, step 3 ...84
Figure 4.6 The first part of the sequence from Figure 4.3b, step 3 ...85
Figure 4.7 Grammar induced from Figure 4.3b using background knowledge ...87
Figure 4.8 Transforming a good grammar to a better one ...89
Figure 4.9 Effects of reparsing ...90
Figure 4.10 An L-system for drawing a Koch curve ...93
Figure 4.11 Hierarchical parsing adjustment ...94
Figure 4.12 How retrospective reparsing corrects initial mistakes ...95
Figure 4.13 Identification of non-recursive L-systems ...96
Figure 5.1 Unifying a rule to produce a recursive grammar ...102
Figure 5.2 The results of a different initial unification ...102
Figure 5.3 Consequences of initial unifications on a more complex grammar ...103
Figure 5.4 Prolog code for transforming a non-recursive grammar to a recursive one ...104
Figure 5.5 Generating recursive grammars ...105
Figure 5.6 Inferring a program from an execution trace (after Gaines, 1976) ...108
Figure 5.7 Inferring procedures from an execution trace ...110
Figure 5.8 Algorithm for identifying a procedure in an automaton ...112
Figure 5.9 A case where a simpler algorithm identifies the wrong procedure ...114
Figure 5.10 Sensitivity of loop-finding algorithm to sequence order ...115
Figure 5.11 The effect of recursion ...117
Figure 5.12 Detecting the structure of a C *switch* statement ...120
Figure 5.13 Forming a recursive rule for a loop ...121
Figure 5.14 A *switch* statement with *break* statements ...122
Figure 5.15 A spurious branch ...125
Figure 5.16 An excerpt from the GEDCOM genealogical database ...128
Figure 5.17 A phrase from the genealogical database ...130
Figure 5.18 Examples of two ways of coding symbol *B* ...132
Figure 5.19 The text in Figure 5.16, with structure and content distinguished ...133
Figure 6.1 Alternative encodings for a sequence ...136
Figure 6.2 Encoded size versus number of symbols in grammar ...145
Figure 6.3 Bits per symbol in grammar as a function of grammar size ...146
Figure 7.1 Hierarchies for Genesis 1.1 ...154
Figure 7.2 Reconstruction of Wolff's (1975) segmentation experiment ...159

Figure 7.3 A sentence from the London-Oslo-Bergen corpus 164
Figure 7.4 Parses of the sentence from Figure 7.3 165
Figure 7.5 Identical parts of a tree 167
Figure 7.6 Bottom-up rendering of a scene 169
Figure 7.7 Illustration of matches within and between two chorales 171
Figure 7.8 Hierarchy for O *Welt, sie hier dein leben*, chorale 50 by J.S. Bach 172
Figure 7.9 Phrases discovered by SEQUITUR for automated abstracting 174
Figure 7.10 Hierarchy of phrases for searching a large text base 175
Figure 7.11 Examples of macromolecular sequences 178
Figure 8.1 Relationships between techniques (in grey boxes) and applications described in the thesis 182
Figure 8.2 Review of results 185
Figure 8.2 Review of results (continued) 186

1. Introduction

Order is heaven's first law

Alexander Pope, *An essay on Man*

We live in a causal universe. We believe this because the laws that we infer from nature successfully predict the observations that we make. Sometimes we cannot perceive the mechanisms that give rise to our perceptions, but we can infer their structure by the nature of their effects. In this thesis, we claim that for sources that produce discrete sequences, there are certain fundamental structures that can be discovered automatically from observing their output. Rather than examining the meaning of the individual symbols in the sequence, this structure is detected in the way that certain combinations of symbols recur. We develop techniques that reveal interesting structure in a wide range of real-world sequences including music, English text, descriptions of plants, graphical figures, DNA sequences, word classes in language, a genealogical database, programming languages, execution traces, and diverse sequences from a data compression corpus. These techniques are simple and robust. They are also useful: they produce comprehensible visual explanations of structure, identify morphological units and significant phrases, compress data, optimise graphical rendering, infer recursive grammars by recognising self-similarity, and infer programs.

1.1 Motivation

The great advantage that computers have over people is their ability to process large amounts of data quickly and accurately. People, on the other hand, excel at thinking creatively and bringing intuition to bear on problems. These complementary strengths encourage a division of labour where machines undertake large-scale, monotonous analyses and present concise results to humans for interpretation. This is particularly true in structure discovery. As the amount of raw information that is collected about the world increases (and it has been alleged that it doubles every twenty months[1]), it becomes less feasible for people to process this data unaided. For example, in an effort to understand our own biology and heredity,

[1] Preface of Piatetsky-Shapiro and Frawley (1991).

biologists are mapping the human genome. When the map is complete, it will comprise a sequence of three billion base pairs. If a scientist were to examine one pair every second, the analysis would take almost a century, without allowing time to reflect on the meaning and function of the sequence.

But it is not necessary to look to computational biology to find massive quantities of data—corporate databases overflow with data on companies' performance. Because good management relies partly on human intuition about products and markets, and because it is not humanly possible to analyse the relevant data, computers are employed to sift through it, detect significant patterns, and report the patterns to decision makers. Researchers have coined the term *data mining* to describe this process, because the patterns that are discovered are analogous to gold hidden within the mountains of data. Thus computers are entrusted with the task of emulating a limited intuition: judging, at a mechanical level, the significance of a pattern or relationship in a database. To extend the metaphor, the computer performs the dirty, exhausting work of the miner, while humans assay the nuggets.

Consistent with this division of labour, algorithms for structure detection should be simple, generic, robust, and efficient. They can be simple because they are not expected to interpret patterns—their role is confined to detection. They should be generic, so that they can be applied to a wide variety of sequences, and be free to find surprising structures unanticipated by the designer of a domain-specific algorithm. They should be robust, so that they can produce a result in any domain. Finally, they should be efficient, to enable analysis of very large sequences, because that is where computers excel.

The specific aspect of structure addressed by this thesis occurs in sequences of discrete symbols: sequences such as natural or artificial language, music, and DNA, where symbols are drawn from a finite alphabet, and each individual symbol does not itself contain significant meaning. The task of finding structure in such sequences is a familiar one: it is performed by infants trying to make sense of the sounds that are produced by people around them, linguists making sense of a new language, archaeologists attempting to decipher an ancient clay tablet, military intelligence decoding an encrypted message, and biologists unlocking the structure of DNA. Of course, in all these situations purely syntactic analysis is combined with semantic knowledge—the infant recognises objects that are associated with certain sound combinations, archaeologists make associations with other artifacts, and biologists understand the chemistry of the structures they find. This thesis, however, concentrates on syntax.

1.2 Thesis statement

The thesis makes two claims: the first is a hypothesis about the structured nature of the world, whereas the second asserts the feasibility of detecting that structure.

1. A variety of sources manifest their structure as repeated segments that are related hierarchically, and as larger-scale branching and looping patterns.
2. Such structure can be detected efficiently and incrementally without reference to the meaning of individual symbols. Once detected, it can be employed to explain and compress the original sequence.

The following two subsections clarify and discuss the implications of these claims.

1.2.1 Existence of structure

We restrict attention to sequences of discrete symbols, where two symbols can be either identical or utterly different—equality is the only meaningful relationship. This restriction is illustrated by a property that holds for the inherent structure: if the sequence is encrypted using a substitution cipher, so that each symbol in the alphabet is replaced by a new symbol, the structure of the result will be identical to the structure of the original. This precludes analysing sequences of measurements—such as a sample of a continuously varying waveform—based on the numerical relation between the samples. It also precludes analysis of natural language based on, say, the phonetic distinction between vowels and consonants. Furthermore, the sequence of symbols is ordered, but there is no temporal aspect to this order: symbols are assumed to arrive at equally spaced intervals.

Despite the absence of semantics, we propose that such sequences possess structural properties that can be inferred automatically, and moreover that there are particular simple kinds of structure that occur in a variety of domains. This is a theory about the world: that these regularities are fundamental to naturally occurring sequences. Its veracity will be demonstrated by the successful detection of such regularities in several case studies.

The first kind of regularity proposed is a hierarchy of repetitions. Any sequence that consists of symbols drawn from a finite alphabet will inevitably contain repeated subsequences as it grows longer. Some of these repetitions may be coincidental, but in many cases they represent significant morphological elements. For example, in a sequence of letters representing English text, words appear as repeating subsequences, as do word parts such as roots and affixes, and multi-word phrases. In music such an element might be a motif or theme that unifies a melody. Repeated groups of actions performed by a computer user may be common sub-tasks within the user's overall task.

As well as being repetitive, subsequences of English text and user action sequences also form a hierarchy. In English, letters combine to form word roots and affixes. Roots and affixes make up whole words, which in turn make up phrases. In the context of a task performed on a computer, short action sequences combine to form larger sub-tasks. In other contexts, hierarchies are employed to reduce the complexity of a system. The biological hierarchy of phylum, class, order, family, genus and species allows the relationships between many kinds of organisms to be concisely and comprehensibly specified. Similarly, structured top-down programming enables complex problems to be broken down into successively simpler parts, until the individual parts are trivial. In this case, the hierarchy provides layers of abstraction within which implementation details can be hidden. Analogously, detecting natural hierarchies in repeated subsequences allows a sequence to be described in a concise, perspicuous manner.

The second kind of regularity proposed by the thesis is larger-scale branching and looping. Rather than comprising exact repetitions, a sequence may contain subsequences that are similar, but vary slightly. In the vocabulary of programming, the variations are branches and the repetitions are loops. Consider a sequence produced by observing a computer user reformatting a bibliography. They might perform identical actions for the authors and page numbers of all publications, but vary the actions in between based on whether the publication is a journal paper or a book. The sequence of actions would be identical at the start and end of processing each publication, with a branch to the two strategies in between. Similarly, a textual database may have a fixed template for each record, using fixed keywords to provide the structure and free-form text between the keywords. These structures can be represented by a branching structure in an automaton.

It is equally important to consider what kind of regularities are *not* considered in this study. The thesis ignores statistical regularities. For example, a sequence formed by randomly choosing the symbols *a* and *b* with probability 2/3 and 1/3 respectively has statistical regularities that, if known, can be utilised in a cipher–text-only attack on a simple encryption scheme such as a substitution cipher. While this sequence will certainly contain repeated subsequences, and longer repetitions of *a*s than *b*s will appear, recognising such repetitions reveals little about the structure of the source. This thesis addresses properties of groups of symbols, such as repetition, rather than properties intrinsic to a particular symbol, such as frequency. Another kind of structure that may exist in sequences is periodicity, where a particular symbol appears at certain intervals, regardless of context. A typical example is the placement of line feed characters in a text file justified for an 80 column page, where line feeds occur every 80 characters. This kind of structure is also ignored in this thesis. Of course, if the periodicity is due to contiguous verbatim repetitions—that is, if the interval is the length of the repetition—the structure will be recognised.

1.2.2 Detection of structure

To show that hierarchical repetitions can be detected efficiently and incrementally, it suffices to create a technique that performs the detection. We describe a technique that forms a hierarchy of repetitions from a sequence in time proportional to the size of the sequence. Furthermore, each symbol is integrated into the existing structure as soon as it is observed, so that the hierarchy can be used at any point in the sequence. This property is referred to as 'incrementality.' Linear time complexity is important for lengthy sequences and for application to real-time systems. The technique can be stated very succinctly in terms of two constraints, but these constraints interact in interesting ways when presented with real-world sequences. The computer program that implements this algorithm has been dubbed SEQUITUR, Latin for 'it follows'—a *double entendre* that alludes to both its proficiency with sequences and its ability to infer structure.

The demand for incrementality stems from the application of these techniques to on-line learning situations, such as the prediction of a system's behaviour. On-line applications require linear-time algorithms to avoid slowing down as processing proceeds. Incremental processing and the linear time constraint inevitably lead to greedy parsing. Greedy parsing makes decisions based purely on local optimality, which enables efficient incremental processing but usually results in a sub-optimal parse overall. This is particularly noticeable in some artificial sequences. The thesis describes two techniques for overcoming this deficiency. The first is to reconsider the initial parse when evidence of a better one appears. A less expensive approach can be employed if knowledge about preferred forms of rules is available from the domain of the sequence. This is the only part of the thesis that takes advantage of domain knowledge, but the mechanism is clearly separated from the main algorithm, which operates without this knowledge.

We also develop a framework for understanding various approaches to choosing between different parses and forming a hierarchy. The hierarchy provides a concise representation of the sequence. None of the techniques developed in this thesis guarantees finding the smallest possible hierarchy—this has been shown to be NP-complete in the size of the input sequence. Previous techniques for finding sub-optimal hierarchies operate in quadratic time, whereas those described here are linear.

Branching and looping structure is more difficult to identify than exact repetition. It can be detected efficiently in certain situations, such as in traces of program execution, but in other cases requires an expensive search of many possibilities. Furthermore, distinguishing correct structure from spurious structure is problematic. Whereas the existence of an exact repetition is undeniable, the validity of a particular branching structure is open to question.

Once structure in a sequence has been detected, it can be exploited in several ways. The primary application is *explanation*: the hierarchy provides a comprehensible account of the significant segments of the structure. In the case of natural language such as English text, it can be used to partition the text into meaningful segments. The same analysis could be applied to a sequence of unknown structure, such as DNA, and provide some insight into its structure. In some cases, it is possible to identify the exact structure of the source. We show that simple L-systems, a class of recursive rewriting systems, can be inferred from their output. The inferred structures are relevant to computational biology, as well as to graphical rendering. In general, however, no guarantees can be made about identification of particular source structures. In this sense, the goal of the thesis is distinct from that of grammatical induction, where researchers assume that the source belongs to a particular class of grammars, and attempt to provide guarantees about identifying the source grammar from its output.

The hierarchy of phrases provides a concise representation of the sequence, and conciseness can be an end in itself. When the hierarchy is appropriately encoded, the technique provides compression. Data compression is concerned with making efficient use of limited bandwidth and storage by removing redundancy. Most compression schemes work by taking advantage of the repetitive nature of sequences, either by creating structures or by accumulating statistics. Building a hierarchy, however, allows not only the sequence, but also the repeated phrases, to be encoded efficiently. This success underscores the close relationship between learning and data compression.

1.3 Some examples

This section previews several results from the thesis. SEQUITUR produces a hierarchy of repetitions from a sequence. For example, Figure 1.1 shows parts of three hierarchies inferred from the text of the Bible in English, French, and German. The hierarchies are formed without any knowledge of the preferred structure of words and phrases, but nevertheless capture many meaningful regularities. In Figure 1.1a, the word *beginning* is split into *begin* and *ning*—a root word and a suffix. Many words and word groups appear as distinct parts in the hierarchy (spaces have been made explicit by replacing them with bullets). The same algorithm produces the French version in Figure 1.1b, where *commencement* is

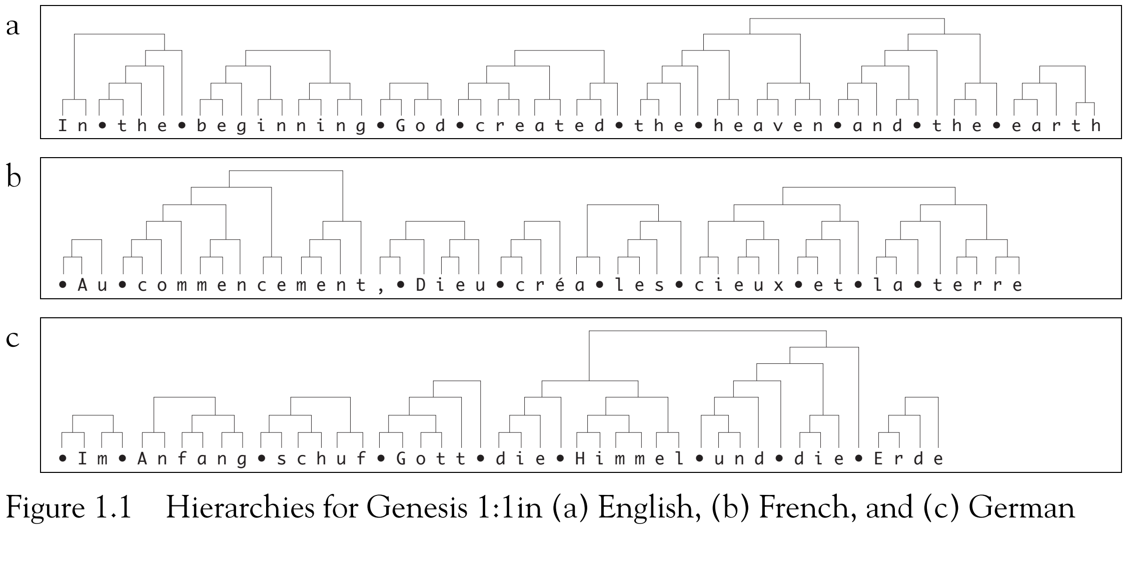

a
I n • t h e • b e g i n n i n g • G o d • c r e a t e d • t h e • h e a v e n • a n d • t h e • e a r t h
b
• A u • c o m m e n c e m e n t , • D i e u • c r é a • l e s • c i e u x • e t • l a • t e r r e
c
• I m • A n f a n g • s c h u f • G o t t • d i e • H i m m e l • u n d • d i e • E r d e

Figure 1.1 Hierarchies for Genesis 1:1in (a) English, (b) French, and (c) German

split in an analogous way to *beginning*—into the root *commence* and the suffix *ment*. Again, words such as *Au*, *Dieu* and *cieux* are distinct units in the hierarchy. The German version in Figure 1.1c correctly identifies all words, as well as the phrase *die Himmel und die*. In fact, the hierarchy for *the heaven and the* in Figure 1.1a bears some similarity to the German equivalent. These examples are discussed in more detail in Section 7.1.1.

Sequences produced by L-systems (a kind of grammar described in Section 4.2.1) have associated graphical representations that include fractals and biological forms. The L-system in Figure 1.2a produces the sequence partly reproduced in Figure 1.2b, which draws the prototypical plant in Figure 1.2c. Identifying a hierarchy of repetitions in the sequence is tantamount to finding a graphical hierarchy that describes the relationships between different parts of the plant. Figure 1.2d shows the non-recursive grammar inferred from Figure 1.2b. The original L-system can be recreated by pattern matching on Figure 1.2d. This has two implications. First, the original L-system can, under certain circumstances, be inferred from the sequence. Second, identification of identical forms and hierarchies within a Figure can make rendering the graphic much more efficient in terms of time and memory. For example, Figure 1.2c can be rendered by traversing the hierarchy represented by the grammar in Figure 1.2d, rather than executing all the instructions in Figure 1.2b.

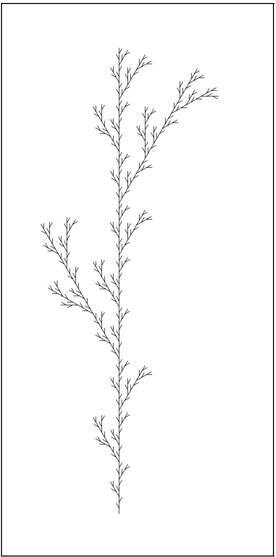
a

```
S → f
f → f[+f]f[-f]f
```

b

```
f[+f]f[-f]f[+f[+f]f[-f
]f]f[+f]f[-f]f[-f[+f]f
[-f]f]f[+f]f[-f]f[+ff[
+f]f[-f]f[+f[+f]f[-f]f
]f[+f]f[-f]f[-f[+f]f[-
f]f]f[+f]f[-f]f]f[+f]f
[-f]f[+f[+f]f[-f]f]f[+
f]f[-f]f[-f[+f]f[-f]f]
f[+f]f[-f]f[-f[+f]f[-f]
f[+f[+f]f[-f]f]f[+f]f[
-f]f[-f[+f]f[-f]f]f[+f
]f[-f]f]f[+f]f[-f]f[+f
[+f]f[-f]f]f[+f]f[-f]f
[-f[+f]f[-f]f]f[+f]f[-
f]f[+f]f[-f]f[+f[+f]f[
-f]f]f[+f]f[-f]f[-f[+f
]f[-f]f]f[+f]f[-f]f[+f
f[+f]f[-f]f[+f[+f]f[-f
]f]f[+f]f[-f]f[-f[+f]f
[-f]f]f[+f]f[-f]f]f[+f
]f[-f]f[+f[+f]f[-f]f]f
[+f]f[-f]f[-f[+f]f[-f]
```

c



d

```
S → B[+Z[-Z
Z → B]B
B → A[+Y[-Y
Y → A]A
A → f[+X[-X
X → f]f
```

Figure 1.2 Inference of hierarchies from L-system output
(a) an L-system
(b) output from (a)
(c) graphical representation of (b)
(d) hierarchy inferred from (b)

This is especially important for complex figures that are used to produce realistic simulations. Chapter 4, Section 5.1 in Chapter five and Section 7.2 in Chapter 7 examine these issues in more detail.

The sequence of instructions performed by a computer when executing a program provides a well understood example of branching and looping structure. Figure 1.3a shows a bubblesort program, which when executed performs the instructions partly shown in Figure 1.3b. This stream can be reconstructed to form the original

a

```
main()
{
  int i;
  for (i = N; i >= 0; i --)
    for (j = 1; j < i; j ++)
      if (a[j - 1] > a[j]) {
        t = a[j - 1];
        a[j - 1] = a[j];
        print_array(a);
        a[j] = t;
        print_array(a);
      }
}

print_array(array)
int array[];
{
  int i;

  for (i = 0; array[i] != -1; i ++)
    printf(“%d “, array[i]);
  putchar(‘\n’);
}
```

b

```
i = N
if (i >= 0)
j = 1
if (j < i)
if (a[j - 1] > a[j])
j ++
if (j < i)
if (a[j - 1] > a[j])
t = a[j - 1];
a[j - 1] = a[j];
a[j] = t;
i = 0
if (array[i] != -1)
printf(“%d “, array[i])
i ++
if (array[i] != -1)
printf(“%d “, array[i])
i ++
printf(“%d “, array[i])
i ++
printf(“%d “, array[i])
...
```

c

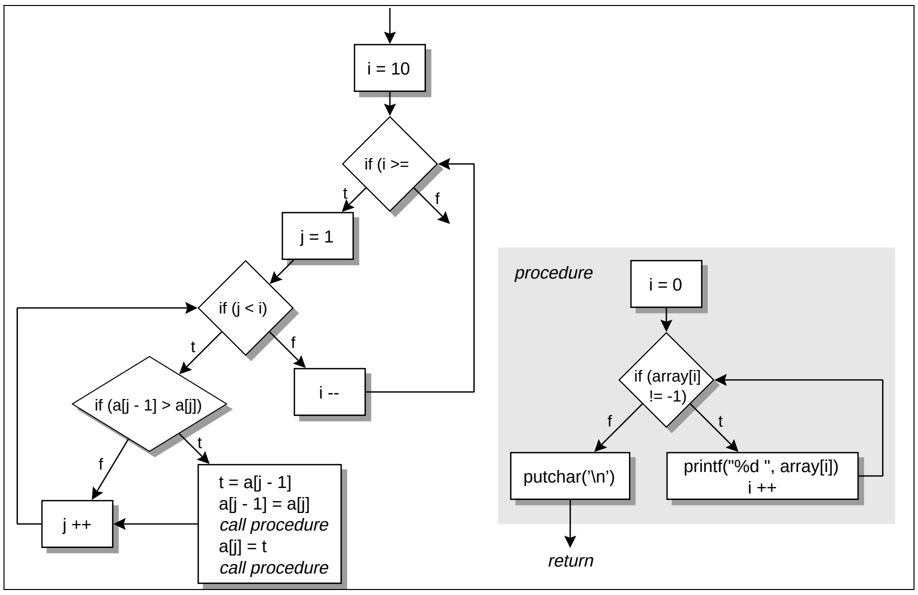


Figure 1.3 Inference of a program from a trace of its execution
(a) a bubblesort program
(b) the instructions executed by (a)
(c) an automaton reconstructed from (b)

program, even if it contains procedure calls and recursion, producing the automaton in Figure 1.3c. The reconstruction algorithm is discussed in Section 5.2.

The techniques for inferring branching and looping structure used for program inference perform poorly at recognising structure such as the textual regularities in Figure 1.4a. The automaton resulting from analysis of this sequence character-by-character is shown in Figure 1.4c, and reflects little of the structure in the original. The techniques for forming hierarchies of repetitions go some way to describing the regularity in the sequence by identifying significant elements. Figure 1.4b shows the grammar formed from the sequence. A hybrid technique drawing on the abilities of each approach is much more successful, as shown in Figure 1.4d, which provides a plausible explanation of the textual structure. This is discussed in Section 5.3.

To return to the humanities, Figure 1.5 shows the regularities detected in two of Bach's chorales. It transpires that both chorales are harmonisations of the same

a
```
switch (c) {
 case 1: value = 2
 case 2: value = 3
 case 3: value = 4
 case 4: value = 5
}
```

b
```
S → switch (c) {A1B2A2B3A3B4A4B5↲}
A → ↲case
B → : value =
```

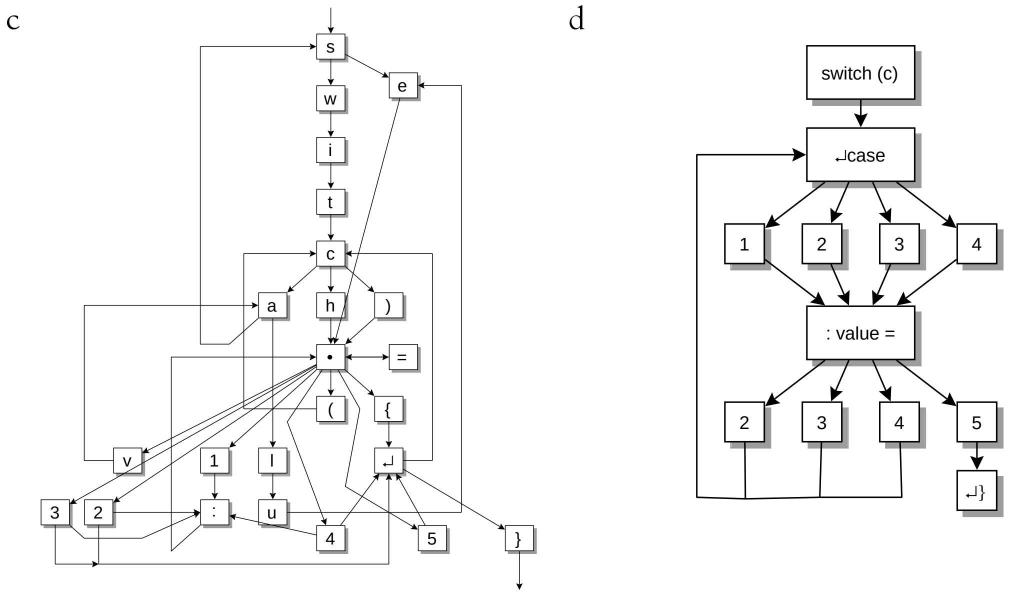


Figure 1.4 Detecting the structure of a C switch statement
(a) the switch statement
(b) SEQUITUR's grammar for (a)
(c) an automaton formed from the individual characters in (a)
(d) an automaton formed from the contents of rule *S*

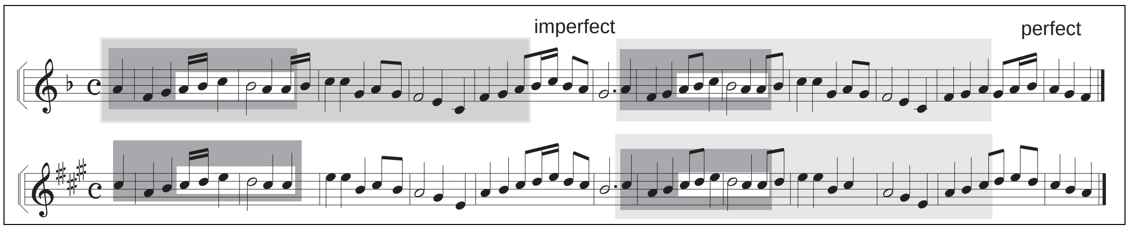


Figure 1.5 Illustration of matches within and between two chorales: for chorales O *Welt, sieh hier dein leben* and O *Welt, Ich muss Dich lassen* by J.S. Bach.

original melodies, as indicated by the matching parts between the chorales. The hierarchy identifies the common first and second half of the top melody, represented by the light gray box, which also occurs in the second half of the bottom melody. It also identifies the imperfect and perfect cadences labelled in the figure. A hierarchy of repetitions is shown in the darker gray box and the white box within it. This discussion is expanded in Section 7.3.

1.4 Contributions

The thesis makes contributions in the form of new algorithms for forming hierarchies, for generalising hierarchies and inferring automata, and in applying these algorithms to a range of sequences.

Grammar formation:

- It is possible to infer a hierarchical representation of a sequence in time linear in the length of the sequence.
- Two constraints on grammar—digram uniqueness and rule utility—are sufficient to form a hierarchical grammar from a sequence.
- A quadratic-time algorithm based on dynamic programming permits visualisation of alternative parses.
- Reparsing can produce a better grammar using retrospective modifications, while maintaining incremental qualities.
- Domain knowledge can be elegantly incorporated to improve the parsing of the algorithm.

Generalisation:

- It is possible to infer a recursive, non-deterministic L-system from a single example of its output.
- Push-down and recursive push-down automata can be inferred from a sequence of instructions.
- Phrase detection and non-deterministic structure recognition can be combined to form a powerful structural inference technique.

Applications:

- A data compression scheme can perform explanation as well as achieve conciseness.
- The minimum description length principle can be applied to detect structure in textual databases.
- Inferring a hierarchy of repetitions can optimise graphical rendering of biological forms.
- Structure of text in a variety of languages can be inferred using a generic technique.
- Musical form can be inferred from melodic sequences.

1.5 Thesis structure

Chapter 2 surveys research related to the thesis, and partitions it into six areas: sequence modelling, grammatical induction, machine learning, data compression, language acquisition, and human sequence learning. All the areas except machine learning are principally concerned with sequences, and all except data compression address learning. Despite their commonalities, there has been little cross-fertilisation between the disciplines. This thesis bridges some of the gaps by demonstrating a scheme for inferring structure from sequences which both learns, in the sense that it produces comprehensible models, and compresses.

Chapter 3 describes an algorithm for the recognition of repetitions and formation of a hierarchy. The first section introduces the two constraints that apply to SEQUITUR's output and drive its operation. Section 3.2 explains some crucial parts of the algorithm that ensure its efficiency when maintaining the constraints. A proof of the linear time processing bound is provided in Section 3.3, while Section 3.4 presents some extreme cases of input, and the performance bounds that they imply for SEQUITUR. Section 3.5 describes a quadratic time algorithm that forms a framework within which various parses can be compared.

Chapter 4 studies problems with the greedy parsing of the basic algorithm when applied to an artificially generated sequence. Section 4.1 previews results obtained in the chapter. The L-system domain is described in Section 4.2, which provides the motivation for the chapter. Where domain knowledge is available to provide an extra constraint on the form that rules should take, this can be used to form a better parse efficiently as described in Section 4.2. Section 4.4 describes how this problem can be corrected by adjusting the parse of an earlier part of a sequence when more data has been seen.

Chapter 5 discusses generalising the grammar. Section 5.1 shows how a recursive grammar can be inferred from the reparsed grammar in Section 4.4. Section 5.2 gives a simple algorithm for inferring an automaton from the trace of a program, and extends it to identify procedures calls and recursion. Section 5.3 demonstrates how this technique can generalise SEQUITUR's grammars to capture branching and looping structure. This section also presents problems with this approach, and discusses a solution based on searching for models that compress the sequence. Section 5.4 describes the application of these generalisation techniques to a large textual database in order to distinguish between fixed structure and variable content.

Chapter 6 explains how the grammars produced by SEQUITUR can be efficiently encoded. A brief review of the state of the art in data compression is given in Section 6.1. Section 6.2 describes an efficient way of encoding grammars that SEQUITUR produces, which transforms SEQUITUR into a data compression scheme. It is then compared against other techniques on a standard data compression corpus in Section 6.3. Section 6.4 returns to the textual database in Section 5.4 and shows how inference can significantly increase compression performance.

Chapter 7 describes the application of SEQUITUR to sequences from several domains. Section 7.1 discusses applications to natural language, including the discovery of words, word parts and multi-word phrases. Here, SEQUITUR is compared to earlier research in language acquisition. Section 7.2 shows how SEQUITUR, with the aid of domain knowledge, can optimise graphical rendering of complex natural scenes. Section 7.3 discusses the hierarchical structures that occur in music, while Section 7.4 investigates how word hierarchies from large corpora can be employed in full-text retrieval and abstracting. Section 7.5 describes experiments in the compression of DNA and amino acid sequences, where SEQUITUR significantly outperforms the best current techniques.

2. Background

Bernard of Chartres used to say that we are like dwarfs on the shoulders of giants, so that we can see more than they, and things at a greater distance, not by virtue of any sharpness of sight on our part, or any physical distinction, but because we are carried high and raised up by their giant size.

John of Salisbury, *Metalogicon*, 1159

If I have seen further it is by standing on the shoulders of giants.

Sir Isaac Newton, *letter to Robert Hooke*, 1675

This thesis is about how to make inferences from a sequence. Six areas of study have been identified that are concerned either with making inferences, processing sequences, or both: *sequence modelling*, *grammatical induction*, *machine learning*, *data compression*, *linguistic segmentation* and *human sequence learning*. Table 2.1 summarises the distinctive features of each area in terms of the kind of input the techniques expect, the form of the output that they produce, and the predictions that they make.

Sequence modelling techniques bear the most similarity to the methods developed in this thesis. They take a single sequence as input, and form a model of it. The model, which is usually represented as an automaton, can then be used both to explain and to extrapolate the sequence.

Whereas sequence modelling is motivated by the need to predict behaviour, *grammatical induction* is inspired by problems in language. Rather than dealing with the internal regularity of a single sequence, techniques for grammatical induction examine several sentences generated by the same grammar, and attempt to identify

field	input	output	predicts
sequence modelling	one sequence	automaton or context table	next symbol
grammatical induction	set of sentences	grammar or automaton	membership
machine learning	unordered tuples	decision tree, rules, etc.	class
data compression	one sequence	compressed sequence	next symbol
linguistic segmentation	stream of language	parsing into words	word boundaries
human sequence learning	artificial sequence	anticipation	next symbol
SEQUITUR	one sequence	grammar and automaton	next symbol

Table 2.1 Characterisation of related fields

the grammar itself. These techniques attempt to emulate people's ability to infer natural language grammars directly from the utterances that they hear.

Research in *machine learning* is less concerned with sequential structure, but concentrates on finding relationships between individual objects with internal structure. These relationships are encoded in structures such as decision trees and rules. Studying the problems of learning these relationships has led to the development of the minimum description length (MDL) principle, which is a formalisation of Occam's razor. A theory is chosen by MDL if it allows the original data to be encoded more concisely than any competing theory does. Chapter 6 employs this principle to justify inferences of branching and looping structure.

Drawing on work in sequence modelling and information theory, *data compression* involves the application of sequence modelling to enable efficient use of storage and transmission resources. Despite its *ad hoc* beginnings, data compression is now firmly based on information theoretic results, and the best current techniques grew out of behaviour prediction in sequence modelling. In addition to providing practical benefits, it represents a robust evaluation methodology for competing sequence models.

There have been several computational attempts to account for *linguistic segmentation* by children—how they learn to divide speech into words. Techniques that have been developed for segmentation have similarities to the methods described in this thesis.

People deal with sequences other than language, such as predictable sequences of events. In the 1950s, the prevailing hypothesis for the *human sequence learning* mechanism was a stimulus-response model. Since then, experiments have been performed that indicate that we form mental hierarchies to record repetitions and to replay them in learned skills. The last section reviews work in experimental psychology concerned with human sequence learning.

2.1 Sequence modelling

Sequence modelling encompasses techniques that take a single string and form a model based on its internal regularity. This section describes these techniques and discusses how they have been applied to real prediction and explanation problems.

The first subsection describes the modelling techniques, and the second outlines the way in which they have been applied to various problems.

We partition the techniques into three groups: techniques that enumerate automata and select the ones that perform well, ones that construct finite context models, and ones that form automata by construction. Enumeration is the simplest approach, and it guarantees to find the optimal model. However, it is usually infeasible. Constructive techniques have the advantage of tractability, but at the expense of optimality. Finite-context predictors are a constructive technique, but are discussed separately because they represent models as context tables rather than automata.

Automata are the representation of choice for sequential structure—even context prediction methods can be recast as automata. Automata can capture the branching and looping structure of a sequence and represent it analogously to a flowchart representation of program sequence. Although there is a theoretical equivalence between grammars and automata—every regular grammar has an equivalent finite state automaton, and every context free grammars has an equivalent push-down automaton—the visual impact of automata makes them useful for illustrative and explanatory purposes.

Current research has by no means exhausted the possibilities for modelling techniques—in fact, the relevant work is rather sparse. The existing techniques, however, are capable of providing useful inferences in practice, and the second subsection shows how simple techniques have been successfully applied.

2.1.1 Techniques

Enumerative

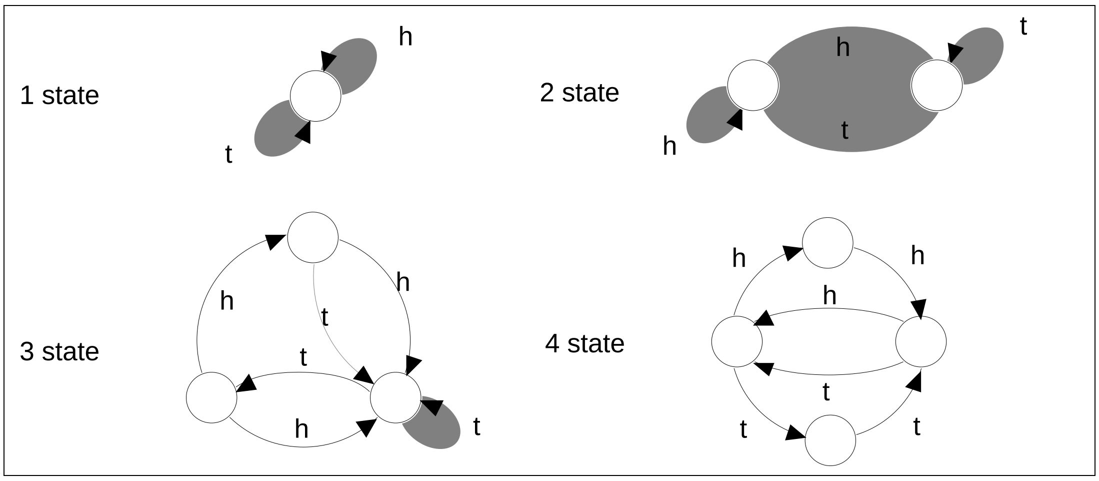
For a given space of models, the simplest inference technique is to enumerate all possible models and choose the best according to some preference criterion. This approach is usually infeasible, because of the large (or infinite) number of models in the space. However, enumeration provides a bound on how successful any heuristic search of the space can be, because it guarantees to maximise any given criterion. Other techniques use knowledge about the problem to eliminate large parts of the space, but risk missing the optimal solution. A key advantage of enumerative techniques is their robustness in the presence of noise, or when the source of the sequence does not belong to the class of automata that is being enumerated. In these cases, enumerating all models guarantees to choose the one that maximises the preference criterion. A heuristic search may be misled by one piece of spurious data, causing it to reject a part of the model space that contains the best solution.

The idea of modelling acausal phenomena, or sequences generated by a process belonging to a different class to the structure being inferred, is central to this thesis, which describes general-purpose techniques that use structures such as grammars and automata as a convenient framework rather than an exact replica of the source. Real world sequences are often acausal—or the causes are not recorded in the sequence—and there is no evidence that structure in sequences such as DNA can be fully captured by a grammar or automaton. It is nevertheless useful to capture some

a

```
h h t h h t h h h t t h h h h t t h t t t h h h h h t h h t t h h h t t t h h h
h h h t t h t t t t t h t t h t t t h h h t t t h h t h h h h h h h h h t t h h
h t t t h h h h h h h t t t t t t h
```

b



c

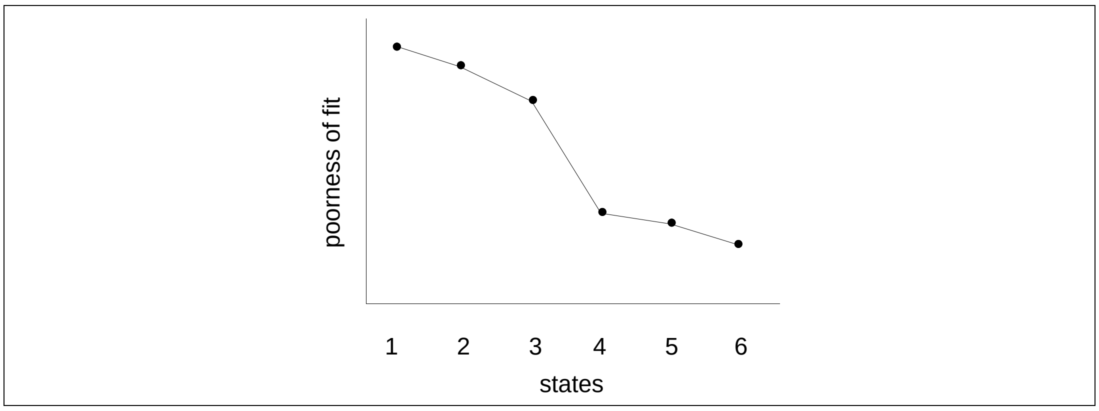


Figure 2.1 Structure detection by enumeration of automata (after Gaines, 1976)
(a) results of a rigged coin toss
(b) admissible 1, 2, 3 and 4 state models
(c) tradeoff between automaton size and fit

parts of the structure.

An enumerative technique was proposed by Gaines (1976) for the space of finite automata. To make the problem more feasible, the inference technique, called ATOM, enumerated automata in order of size starting from the smallest, one state, automaton. Because ATOM's preference criterion is biased towards small automata, it is likely to find a solution sooner if the space is searched in this order. Each model that ATOM generates is evaluated according to both its size and its fit to the data. Gaines examines several ways of characterising the goodness of fit of a non-deterministic automaton: the expected number of errors, the entropy of the original sequence with respect to the distribution implied by the automaton, and the sum of the squares of the differences between the probabilities implied by the automaton and the symbols that occur. An admissible model is defined as one that cannot simultaneously be made smaller and more accurate. There is therefore at most one admissible model with two states, one with three states, and so on.

For example, consider a rigged betting machine proposed by Andreae (1977) that produces coin tosses as in Figure 2.1a. Figure 2.1b shows the admissible one-, two-, three- and four-state models, and Figure 2.1c shows the first six models plotted as size against fit. To choose the best model, ATOM chooses the model that is significantly better than the model with one less state, but not significantly worse then the model with one more state. In this case, the best model is chosen to be the four-state one, because of the large improvement in fit from the three-state model but the small gain in using the five-state model. The improvement in fit from three to four states indicates that the four-state model captures an important structure not present in the three state model. The actual structure of the sequence is that when it is broken into groups of three symbols, the second symbol in every triplet is a repetition of the first symbol. Other modelling procedures discussed below cannot detect this periodicity.

Extending this enumerative technique to a larger space could be expected to slow the inference process. However, this is not necessarily the case. Witten (1981b) extends ATOM to enumerate recursive transition networks. Surprisingly, in one situation inference is more efficient for recursive automata because a three-state recursive model captures structure as well as a four-state non-recursive model, and there are fewer of the former to search. Section 6.2 discusses constructive techniques for finding an automaton, and provides extensions for inferring push and pop operations.

Constructive techniques for finite context predictors

Finite context modellers are based on the observation that a symbol in a sequence is often determined by the symbols that immediately precede it. For example, in English text, the letter *q* determines that the next symbol is most likely to be *u*. In a less extreme example, the letters *th* usually precede the letter *e*, but may also precede *a*, *i*, *o*, *u*, or *r*. In fact, the most successful predictors used for data compression schemes are based on this principle.

In his pioneering work on learning sequences, Andreae (1977) developed a system, PUSS, that predicts future events based on past events in the same context. Andreae likened this approach to the stimulus-response model of behaviour in animals. PUSS stands for Prediction Using Slide and Strings, the 'slide' referring to the window of context preceding the current prediction, and 'strings' to the table of previous contexts and subsequent symbols. Along with PURR (Purposeful Unprimed Rewardable Robot), PUSS formed a system that interacted with a user through a

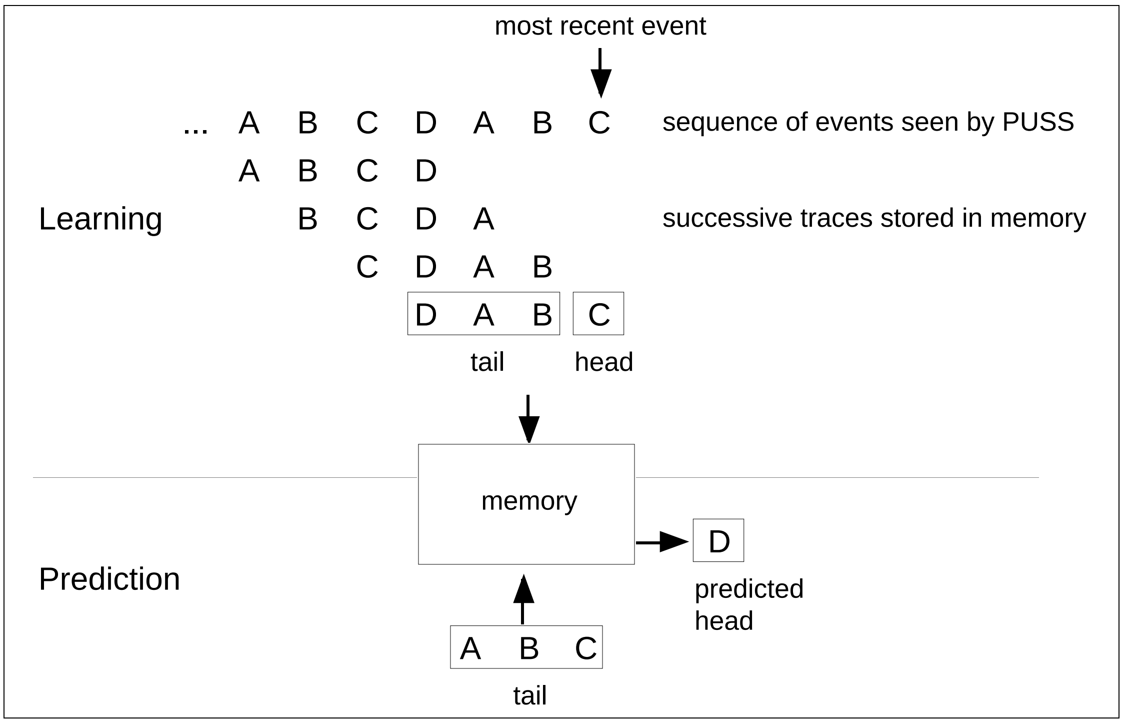


Figure 2.2 The finite context prediction mechanism for PUSS and FLM (after Cleary, 1980)

character terminal and learnt complex behaviour by conditioned reflex. In fact, PURR coordinated several PUSS predictors that acted on different kinds of sequences, and selected an action based on the predictions of the PUSSes. This system was capable of learning counting, simple picture recognition, and to navigate an imaginary world.

Figure 2.2 shows PUSS's basic learning and prediction mechanism. As each new symbol appears in the input, it becomes the head of a buffer whose tail is the three most recent symbols. The size of the tail is called the *order* of the predictor—in this case three. This buffer is stored in an associative memory in such a way that it can be efficiently retrieved based on its tail. When the mechanism is called on to make a prediction, the last three symbols are used as an index into the memory, and the most common head for that tail is returned. PUSS is programmed by example—incrementally providing actions or correcting predicted actions for a specific problem until the correct responses to the range of contexts has been learnt.

A modelling system based on context prediction can, in fact, be recast as a computing device capable of emulating a Turing machine. Cleary (1980) described such a system, the Finite context Learning Machine (FLM), and showed that despite the constantly changing context memory, it can be programmed to execute a fixed algorithm. The innovation that allows PUSS to be teachable and the FLM to act as a computer is the interleaving of *patterns* and *actions* in the sequence, analogous to *stimuli* and *responses*. The patterns are the input to the computer, and the actions represent the execution of the program. In this way, the memory contains previous predictions as well as inputs.

A finite-context predictor can be represented as an automaton by creating a state for each context, and transitions for each symbol seen in that context to the state representing the new context. The new context is formed by deleting the oldest symbol in the old context, and appending the new symbol on which the transition is made. Although efficient and effective at capturing some kinds of structure, finite-context predictors cannot capture counting events, and can only recognise periodicities that are no greater than the size of the context. Furthermore, the models lack comprehensibility—the learned relationships are not crystallised in a concise structure, but rather embodied in a potentially large and unstructured context memory.

Constructive techniques for automata

A particularly straightforward method of constructing an automaton from a sequence is to create a state for each unique symbol in the sequence, and a transition between two states whenever the symbols are adjacent in the sequence. This automaton is equivalent to a finite context predictor with a context of one: the state for a particular symbol represents the context of that one symbol. Witten (1979) calls this the *ingenuous non-deterministic model* of the sequence.

Despite its simplicity, this is a powerful way of producing an automaton. Consider the bubblesort program in Figure 2.3a, an example proposed by Gaines (1976). Tracing its execution on the given array results in the sequence of instructions in Figure 2.3b. Creating the ingenuous automaton from the sequence, where one

a
```
int a[] = {2,34,23,6,7,43,6,1,6,4};


main() {
  for (i = 10; i >= 0; i --)
    for (j = 1; j < i; j ++)
      if (a[j - 1] > a[j]) {
        t = a[j - 1];
        a[j - 1] = a[j];
        a[j] = t;
      }
}
```

b
```
i = 10
if (i >= 0)
j = 1
if (j < i)
if (a[j - 1] > a[j])
j ++
if (j < i)
if (a[j - 1] > a[j])
t = a[j - 1];
a[j - 1] = a[j];
a[j] = t;
...
```

c

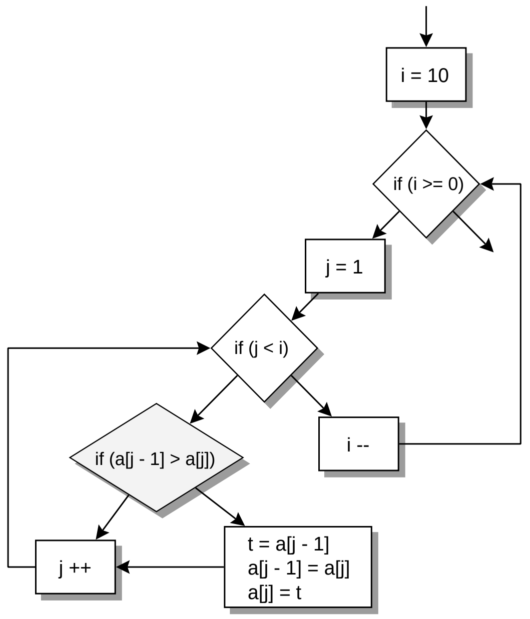


Figure 2.3 Inferring a program from an execution trace(after Gaines, 1976)
(a) bubblesort program
(b) part of the trace produced by (a)
(c) finite state automaton produced from the bubblesort trace

instruction becomes one node, results in the automaton in Figure 2.3c. This is, in fact, a correct program for bubblesort, except for the non-determinism at the conditional branches. Extending the idea to longer contexts, so that one state signifies the observation of *k* consecutive symbols, allows the technique to identify any deterministic automaton with *k* or less states. Section 5.3 describes how *k* can be allowed to vary from state to state in order to capture the changing nature of some structures.

2.1.2 Applications

The simple techniques described in the previous subsection can achieve considerable success in practical situations. Of course, enumerative techniques must generally be discarded as impractical, so these applications utilise finite-context and automaton construction techniques. The ability of these techniques to make predictions enables them to be used in a variety of ways, including data compression, programming by demonstration, optimisation of data entry, predictive data caching, and the optimisation of Prolog programs.

Using arithmetic coding, a technique described in Section 2.4, it is straightforward to transform a predictor into a data compression scheme. The finite-context computer (FLM) described above was adapted by Cleary and Witten (1984) to become a data compression scheme called *prediction by partial matching* (PPM). This scheme is described in more detail in Section 2.4, but the general idea is to use the finite contexts to provide a probability distribution for the next symbol in a stream of symbols. This distribution, which can be calculated by both the encoder and the decoder, can be used by the arithmetic coder to transmit the symbol efficiently. The key innovation in PPM is the blending of predictions from various context lengths, so that the effective order of the predictor grows as more of the sequence is seen, and it becomes more confident in its predictions.

A particularly fruitful application of these techniques is to programming by demonstration (PBD), where a computer is taught a repetitive procedure by demonstrating several examples of its execution. A key advantage of PBD over traditional programming is that it allows a computer user to specify a procedure in the environment where the task is performed—an environment with which they are familiar. The following paragraphs summarise the application of sequence modelling to PBD in four different domains: a hand-held calculator, a text-based computing environment, a text editor, and a graphical editor.

Whereas calculators alleviate the drudgery of manual calculation, they do little to allow automation of complex calculations involving several separate steps. Witten's (1981a) predictive calculator used a fixed-length context to predict the user's next action when making a repetitive calculation, such as plotting several points of a complicated function. In several example calculations, the calculator learnt the fixed parts of a computation, and after a couple of iterations only prompted the user for variable parts.

People's interaction with computers is often highly predictable. Darragh and Witten (1992) used a finite context predictor to expedite text entry. The system was especially targeted at disabled users, for whom typing is difficult or impossible. By predicting characters or groups of characters based on preceding characters, data entry speeds can be vastly increased over typing every character. The technique was applied to UNIX command line editing, where repeated commands and file names give rise to spectacular gains in efficiency, and many disabled users routinely take advantage of the system. The technique was also applied to a text editor, where predicted phrases can be selected with a pointing device.

The EMACS editor has a built-in programming language based on LISP. Despite its power, few casual users take advantage of it, so Witten and Mo (1993) set out to provide a PBD system that would construct programs automatically from user traces. This text editing learning system (TELS) forms an automaton from a trace of user actions in the same way that the bubblesort automaton in Figure 2.3c was formed. The system is able to relate two slightly different actions that are in fact the same action in different contexts. Once this generalisation has been performed, creation of the automaton is trivial.

Repetitive graphical editing is another domain in which sequence modelling can vastly improve efficiency. Maulsby *et al.* (1989) describe METAMOUSE, a system for programming by demonstration within a drawing package. A key part of METAMOUSE was its ability to understand graphical constraints indicated by the user, which is largely a user-interface issue, but an equally important function was its ability to infer loops and branches from the sequence of user actions. Witten and Maulsby (1991) describe a way of evaluating various automata inferred from a trace of user actions. They choose the automaton that allows the entire trace to be encoded in the smallest amount of information. This specifies a tradeoff between the size of the automaton and its fit to the sequence. A small automaton may not capture much of the sequence's structure, so the sequence must be transmitted explicitly. On the other hand, a very large automaton might record the sequence verbatim with one state per symbol in the sequence. In this case, the automaton itself takes as much space as the sequence in the first place. Minimising the sum of the size of the model and the size of the sequence encoded according to the model gives a particular tradeoff that works well in practice. This technique is described in more detail in Section 2.3.

Modelling and predicting sequences can be seen as a learning problem. While there has been little work on sequence learning in the machine learning literature (one notable exception is Dietterich and Michalski's (1986) SPARC/E, described in Section 2.3), Laird (1994) argues that it should be included with the other main paradigms of machine learning, which he summarises as concept learning, clustering, and associative learning. This issue is discussed further in Section 2.3, but it is appropriate to discuss Laird's contribution to sequence modelling here. He defines the *discrete sequence prediction* (DSP) task as finding 'statistical regularities in the input so that the ability to predict the next symbol progresses beyond random guessing,' and suggests five applications for DSP. Within *information theoretic applications*, he includes data compression and using prediction in on-line game-playing situations. *Data optimisation* refers to the use of prediction to optimise programs so that situations predicted by DSP take less time. *Dynamic buffering algorithms* predict which data an application is likely to require next, and use this prediction to prefetch data to a fast memory cache. *Adaptive human-machine interfaces* refers to systems such as the PBD systems described above, which use sequence modelling to adapt to the usage patterns of a computer user. Finally, *anomaly detection systems* flag unexpected events—those that are hard to predict—to identify unusual uses of a system.

The specific algorithm described by Laird (TDAG: transition directed acyclic graph) is a finite context predictor similar to PPM. It performs somewhat worse that PPM as a data compression algorithm because of the crude context blending and the use of Huffman rather than arithmetic coding. The paper describes two other applications of TDAG: the optimisation of Prolog programs and predictive caching. In the former, Laird notes that where a goal to be satisfied unifies with the heads of several clauses, selecting the correct clause at each step (avoiding clauses that will fail) eliminates backtracking and accelerates program execution. Of course, perfect prediction of the correct clause is unlikely, but TDAG, using the previous clauses satisfied as a context, predicts the correct clause often enough to provide a 10%–20% improvement in execution time on the Prolog programs tested. In predictive caching, TDAG was used to predict which block would be required next based on the pattern of blocks recently requested, and was shown to outperform standard caching techniques.

In summary, then, sequence modelling techniques provide simple but powerful ways of capturing regularities in a sequence, and have supported several useful applications.

2.2 Grammatical inference

Biermann and Feldman (1972) summarise the grammatical inference problem as follows: 'a finite set of symbol strings from some language *L* and possibly a finite set of strings from the complement of *L* are known, and a grammar for the language is to be discovered.' There are two important differences between sequence modelling and grammatical induction. First, sequence modelling deals with single strings and analyses their internal regularity, whereas grammatical induction deals with multiple strings generated by a single source, and analyses their similarity to each other. Second, grammatical induction assumes that the source grammar belongs to a particular class of grammars, whereas sequence modelling, especially when invoked on acausal sequences, seeks to approximate the source, which belongs to an unknown class. This means that it is possible for a grammatical inference procedure to guarantee identification of the source, whereas this is not possible for sequence modelling. This section describes specific work in the field of grammatical induction and the significance it has for this thesis.

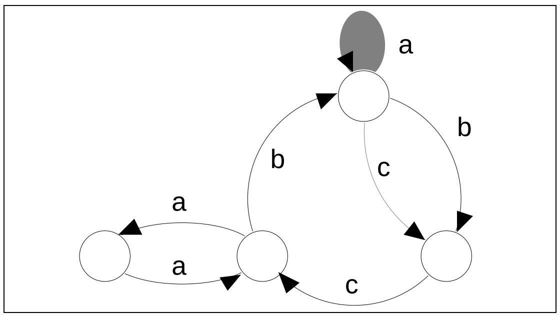
2.2.1 Techniques

Research in grammatical inference appeared after the development of formal grammars, most notably by Chomsky (1957b). Part of the motivation for formalising notions of grammar was to understand how people are able to learn language. The search for techniques to infer a grammar from its language followed naturally.

An early inference procedure was described by Chomsky and Miller (1957a), as reported in Solomonoff (1959). Chomsky proposed a method for detecting loops in finite state languages. The approach requires a set of valid sentences, and an oracle that determines whether a sentence is in the language.

The algorithm proceeds by deleting part of a valid sentence and asking the oracle whether the sentence is still valid. If it is, the deleted part is reinserted into the sequence and repeated, so that it appears twice. If the sentence is still in the language, a cycle has been detected. Figure 2.4 shows an example. Figure 2.4a is the

a



description	sequence	membership
b original sequence	abcca	yes
delete bcc	aa	yes
add bcc	abccbcca	yes
add bccbbc	abccbccbbca	yes
add a	abacca	yes

c delete one symbol	bcca	no
	acca	no
	abca	no
	abcc	no
delete two symbols	cca	no
	aca	no
	aba	no
	abc	no
delete three symbols	ca	no
	aa	yes
	ab	no
repeat bcc	abccbcca	yes
	abccbccbcca	yes

Figure 2.4 Inferring an automaton by insertion and deletion of cycles
(a) the automaton to be inferred
(b) some membership-preserving transformations
(c) a systematic search for cycles

target automaton, and the initial valid sentence, *abcca*, is given in Figure 2.4b. The sentence can be modified in several ways while maintaining its validity: the sequence *bcc* can be deleted, leaving the sentence *aa*. Alternatively, *bcc* can be repeated one or more times, or replaced with *bbc*. After the first *b*, the symbol *a* can be inserted an arbitrary number of times. To infer the automaton, parts of the initial sentence are systematically deleted, as shown in Figure 2.4c. First all the ways of deleting one symbol are tried, then the two-symbol deletions, and so on. Out of all the resulting sentences, only one, *aa*, is still valid. To establish *bcc* as a valid cycle, it is repeated and checked to see if it is still valid. Proceeding in this way, it is possible to identify the target automaton.

Cycles in a grammar may not always involve contiguous symbols. For example, in a phrase-structure grammar that defines nesting of brackets, cycles of open brackets must be matched by a corresponding cycle of closing brackets. Solomonoff (1959) described a method of grammatical inference for phrase structure grammars that detects this more complex cyclic structure. The approach for phrase structure grammars is similar to that for finite grammars—it relies on an oracle to determine whether a sentence with additional cycles belongs to the language of the source grammar.

People seem to be capable of learning a language without specific provision of negative examples. Their ungrammatical utterances are not always corrected, and they must rely mainly on inference from positive examples. However, it does not follow that an inference procedure should be able to infer any grammar from positive examples alone, as illustrated by Gold (1967). Gold proposed the concept of *identification in the limit*, which allowed inference algorithms to be analysed in terms of their ability to converge on a grammar, given enough examples. An important result is that given positive and negative examples of strings from a language, the grammar generating that language can be identified, but without negative examples no algorithm can guarantee identification.

Gold defines identification in the limit in the following way. Sentences from the language are presented to the learner. After each one, the learner makes a guess at the correct grammar. If after some finite time the learner's guesses are all the same, and the guessed grammar is the target grammar, then the language has been identified. If this guarantee can be given for all grammars in a class for a particular learner, the class of grammars is identifiable in the limit. 'In the limit' means it is not possible to bound the identification time. This process assumes some effective algorithm for inferring a grammar from the sentences, and the discovery of such an algorithm for various classes of grammar is the subject of ongoing research in grammatical induction. For Gold's purposes, however, an algorithm that enumerates all grammars in a class suffices to demonstrate some important results.

Gold's result that identification in the limit cannot be guaranteed where negative examples are unavailable is justified as follows. Imagine a class of languages that includes all finite languages and one infinite language L_∞. After some number of examples, a finite language L_1 can be inferred (say, by the enumerative algorithm just described). By presenting some examples not covered in L_1 but covered by L_2, the learner's guess will change to L_2. Since there are an infinite number of finite languages, the learners guess can be made to change indefinitely, preventing it from ever identifying the infinite language.

Whereas Gold's theorem holds for many classes of grammar, it is not true for some restricted classes. Angluin (1982) proposed a new class of *k-reversible* automata and an associated inference mechanism that, for each *k*, guarantees identification from positive examples only. Explaining *k*-reversibility is best achieved by generalising from the case where *k* is zero, i.e. zero-reversibility. If both an automaton and its reverse are deterministic, the automaton is zero-reversible. The reverse of an automaton is formed by changing the directions of all the transitions and swapping the initial and final states. This effect can be achieved by starting at the final state of the automaton and following transitions only in the reverse direction of the arrows. For example, Figure 2.5b is the reverse of Figure 2.5a.

K-reversibility is a generalisation of the requirement of determinism in the reversed

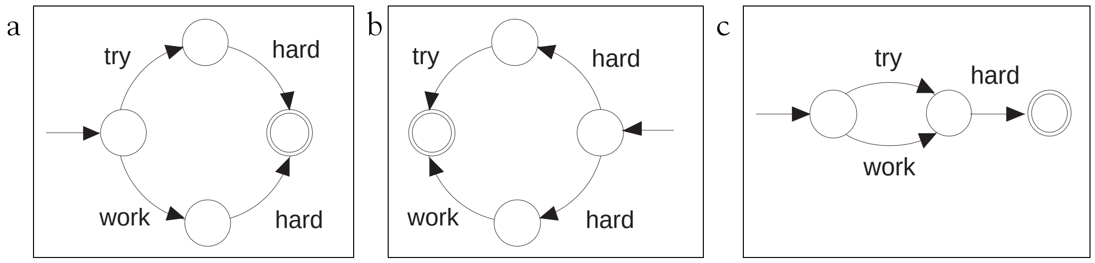


Figure 2.5 Automata to illustrate *k*-reversibility
(a) an automaton
(b) the reverse of (a)
(c) a zero-reversible version of (a)

automata. Instead of having to determine a unique next state given a single symbol, the process interpreting the automaton is permitted to look k symbols ahead in the sequence. If these symbols always determine a unique state, the automaton is *deterministic with lookahead k*. If an automaton is deterministic, and its reverse is deterministic with lookahead k, the automaton is *k-reversible*.

For example, Figure 2.5a shows an automaton that accounts for two English verb phrases, *try hard* and *work hard*. The automaton is deterministic because the start state is the only one with more than one transition leading from it, and the transitions have different labels. This means that considering the transition symbols as input symbols, it is always clear what the next state is whenever a new symbol is seen. Figure 2.5b shows the reverse of 2.5a. The start and accepting states have been interchanged, and the transition directions have been reversed. The start state here is non-deterministic: it has two transitions leading from it with the same label, so when accepting a sentence starting with *hard*, it is not clear which transition to take. This means that the original automaton is not zero-reversible. However, it is one-reversible, because by looking ahead one symbol to *try* or *work* in Figure 2.5b, the correct transition from the start state can be determined—the reverse automaton is deterministic with lookahead one.

Inferring a k-reversible automaton from a set of sentences proceeds as follows. To begin, the trivial automaton is formed from the sentences by making one path from start to accepting state for each sentence. Figure 2.5a was formed from the sentences *try hard* and *work hard*. To infer a zero-reversible automaton from this automaton, states are merged where they violate the determinism rules in either direction. The original automaton is already deterministic, but the reverse is not, so the two states that the start state transitions to on the symbol *hard* are merged. Performing this on the original automaton produces Figure 2.5c, which is deterministic, and its reverse is deterministic. This means that it is zero-reversible. In general this process is iterative, because merging states may cause further non-determinism. The algorithm for forming a k-reversible automaton is identical, except for replacing the determinism constraint with determinism with lookahead k. The time complexity of the zero-reversible inference algorithm is $O(n\alpha(n))$, where n is the sum of the lengths of the sentences. and $\alpha(n)$ grows very slowly (Tarjan, 1975). The time-complexity for the k-reversible inference algorithm is $O(kn^3)$, where n is defined in the same way.

2.2.2 Applications

K-reversible grammars form an important class because there exists a polynomial-time inference procedure for them that guarantees identification in the limit. The complexity of the inference algorithm means that inference can be performed on some real-world problems. Returning to one of the motivating problems behind grammatical induction, does *k*-reversibility help in the identification of the grammar for English? Berwick and Pilato (1987) used *k*-reversibility to infer automata for two sublanguages of English: the auxiliary system (Judy gives bread, Judy has given bread, Judy might have been given bread, etc.) and noun phrase specifiers (those several deer, no big deer, many deer, these two hundred deer, etc.). They found that the auxiliary system could be represented by a 1-reversible automaton, reproduced in Figure 2.6, while noun phrase specifiers required a 2-reversible automaton with 81 states and about 300 transitions. In both cases the correct automaton was inferred from a hand-formed corpus of examples using the *k*-reversible inference mechanism. The corpus for the auxiliary system is included in the paper, and represents a systematic exercising of various combinations of auxiliaries for Judy, giving, and bread. It is unclear how the procedure would perform on a representative sample of English.

Recall the PBD tasks discussed in the last section. Tedious computer work often consists of several repetitions of similar work, which can be treated as multiple sentences from the same language. Schlimmer and Hermens (1993) applied the inference algorithm for *k*-reversible automata to the design of an adaptive user interface. The domain is a form-filling exercise, where each sentence is the

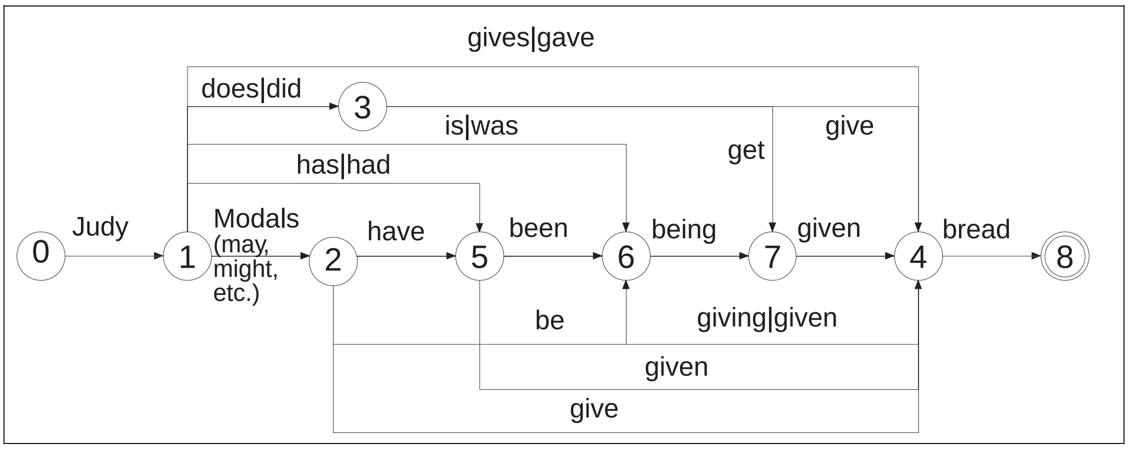


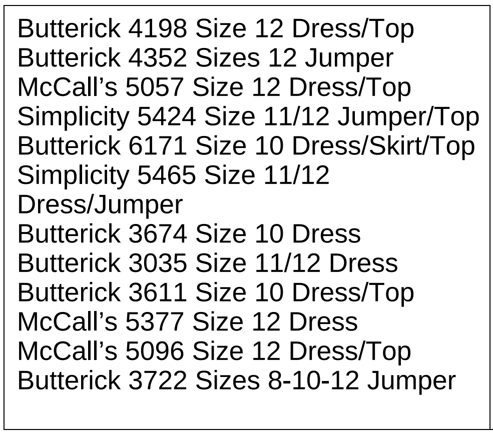
Figure 2.6 One-reversible automaton for the English auxiliary system (after Berwick and Pilato, 1987)

specification of a member of a class such as sewing patterns or computer models. The *k*-reversible automaton inferred from the sentences is not only useful for predicting values and saving the user work, but it can be automatically transformed into a custom-built form for quicker entry. For example, Figure 2.7a shows some sample sentences from a list of sewing patterns. The automaton inferred from these sentences is shown in Figure 2.7b, and the custom interface is shown in Figure 2.7c.

2.2.3 Single versus multiple sentences

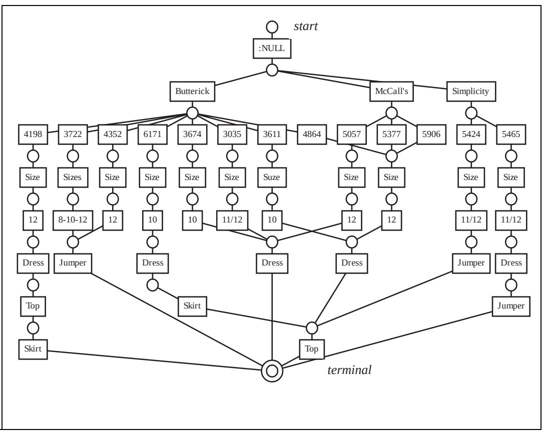
The techniques described in this thesis take only one sequence as input, so there can be no negative examples. It follows from Gold's (1967) theorem that there can be no guarantee of convergence to the source grammar. Furthermore, our sequences do not always emanate from a well defined grammatical source, whereas

a

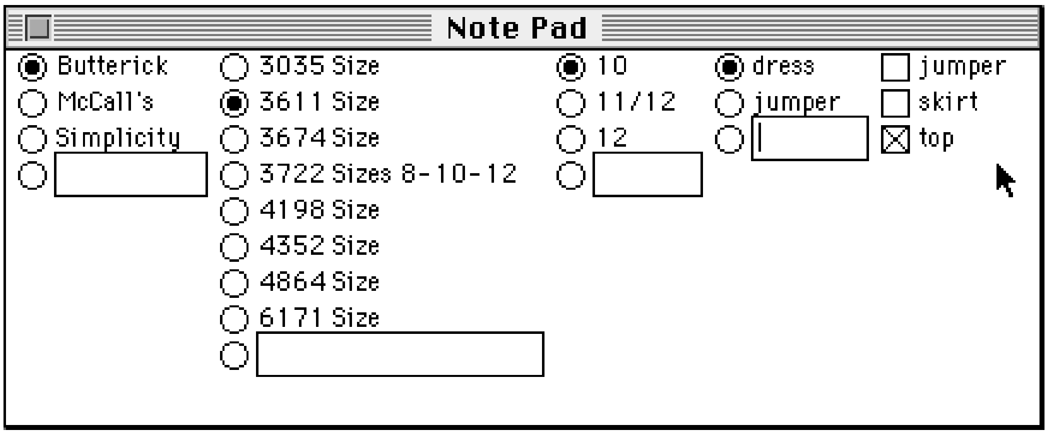


Butterick 4198 Size 12 Dress/Top
Butterick 4352 Sizes 12 Jumper
McCall's 5057 Size 12 Dress/Top
Simplicity 5424 Size 11/12 Jumper/Top
Butterick 6171 Size 10 Dress/Skirt/Top
Simplicity 5465 Size 11/12
Dress/Jumper
Butterick 3674 Size 10 Dress
Butterick 3035 Size 11/12 Dress
Butterick 3611 Size 10 Dress/Top
McCall's 5377 Size 12 Dress
McCall's 5096 Size 12 Dress/Top
Butterick 3722 Sizes 8-10-12 Jumper

b



c



● Butterick	○ 3035 Size	● 10	● dress	☐ jumper
○ McCall's	● 3611 Size	○ 11/12	○ jumper	☐ skirt
○ Simplicity	○ 3674 Size	○ 12	○	☒ top
○	○ 3722 Sizes 8-10-12	○		
	○ 4198 Size			
	○ 4352 Size			
	○ 4864 Size			
	○ 6171 Size			
	○			

Figure 2.7 Inferring a customised data entry form from input (after Schlimmer and Hermens, 1993)
(a) the input data
(b) a reversible automaton inferred from (a)
(c) a customised form based on (b)

grammatical inference assumes a certain class of grammars to aid inference. This thesis uses grammars as a useful representation for different kinds of structure, even if the structure is not in fact generated by a grammar, and certain parts of the sequence cannot be explained by a particular class of grammar. In this sense, the techniques are pragmatic rather than designed to facilitate convergence proofs. This difference in approach is emphasised by the artificial nature of the input sequences described in most research on grammatical induction, as opposed to the real-world sequences treated here.

The multiple sentence assumption of grammatical induction makes it suitable for many realistic problems. Human speech can usually be partitioned into utterances, each of which corresponds to a single sentence from a language. Schlimmer and Hermens (1993) examine the sentence-like structure of a repetitive data-entry task, where the task varies from iteration to iteration but nevertheless exhibits some constant grammatical structure. However, there are many cases in which sequences do not fall naturally or automatically into sentences. Consider the text of a high-level computer program. A program represents a single sentence generated by a grammar, and is therefore unsuitable for analysis by grammatical inference techniques. It does, however, possess much internal structure such as repeating keywords and template structures, which can be exploited to provide insight into the nature of the source grammar.

2.3 Machine learning and MDL

Whereas machine learning largely deals with non-sequential data, the issues surrounding learning and inference discussed in the machine learning community apply equally to sequence modelling. This section describes the distinctive features of research in machine learning, and then discusses how a fundamental problem—learning approximate relationships from positive examples alone—relates to sequential structure detection.

2.3.1 Machine learning vs sequence modelling

One way to define learning is *the intentional adaptation of a system to improve its performance*. Machine learning is the study of algorithmic means of performing this adaptation. Although in its widest sense this encompasses sequence modelling and grammatical induction, the most commonly studied problem in machine learning is classification, where the target is a concept rather than a grammar or automaton. The inference technique is given a series of instances, each consisting of several attributes describing a particular object or event, and a class to which that object or event belongs. The inferred concept is a function that computes the class from the attributes. A plethora of machine learning techniques exist, and they can be categorised according the form of the function that they induce. Langley (1996) suggests five paradigms of learning: neural networks, instance based learners, genetic algorithms, rule induction, and analytic learning.

a

outlook	temperature	humidity	windy	class
sunny	85°	85%	false	Don't Play
sunny	80°	90%	true	Don't Play
overcast	83°	78%	false	Play
rain	70°	96%	false	Play
rain	68°	80%	false	Play
rain	65°	70%	true	Don't Play
overcast	64°	65%	true	Play
sunny	72°	95%	false	Don't Play
sunny	69°	70%	false	Play
rain	75°	80%	false	Play
sunny	75°	70%	true	Play
overcast	72°	90%	true	Play
overcast	81°	75%	false	Play
rain	71°	80%	true	Don't Play

b

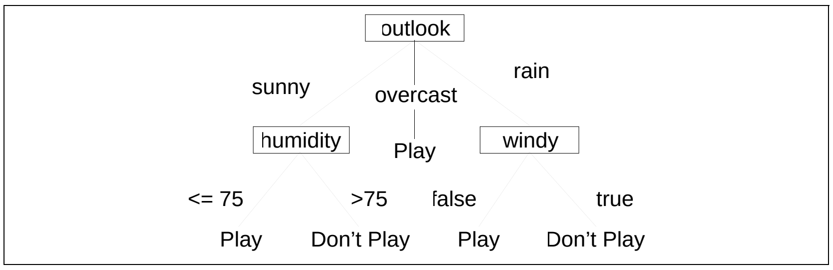


Figure 2.8 Inferring a decision tree from data
(a) the golf data set
(b) a decision tree inferred from (a)

Machine learning is distinct from both sequence modelling and grammatical induction because it is concerned with the relationship between single objects with internal structure, rather than between a sequence of objects with no internal structure. Figure 2.8a shows a toy machine learning data set that lists fourteen instances describing attributes of the weather on various days, and whether those days were suitable for playing golf (the class). Figure 2.8b shows a decision tree inferred from the data that can be used to classify other days according to their suitability for golf. To classify a new instance, the tree is traversed by starting at the root and following branches dictated by the value of the attribute tested at each node.

Not all machine learning systems ignore sequences. The card game *Eleusis* involves a dealer presenting players with a sequence of cards, where the sequence is determined by a rule invented by the dealer. The players must extrapolate the sequence by guessing the dealer's rule. Whenever the player places a card, the dealer informs them whether or not the card was a correct extrapolation. Dietterich and Michalski (1986) created a system, SPARC/E, which is capable of playing Eleusis by searching the space of possible rules. These rules involve use of the colour, suit and rank of the cards, which can be related in various ways. For example, one possible rule is *strings of cards such that each string contains cards all in the same suit and has an odd number of cards in it.* The problem therefore involves finding rules that relate particular attributes of the instances, as well as their sequential relationships. In contrast, sequence modelling and grammatical induction techniques would treat the cards as 52 distinct symbols with no greater similarity between cards of the same suit than cards in different suits.

2.3.2 The minimum description length principle

Despite the different context and approach to learning, sequence modelling and machine learning have broad issues in common. Because machine learning has repeatedly been applied to real-world problems, practitioners have acquired valuable experience in the necessary characteristics of a successful learning scheme. The issue that we concentrate on here is the evaluation of a candidate function in the absence of negative examples, or where few examples are available for testing.

The success of a machine learning system is usually measured by its prediction accuracy on test data—how good it is at anticipating new observations. This is applicable where instances from several classes are available, for example positive and negative examples. This kind of evaluation also assumes that some examples can be set aside from the set from which inferences are made, for the purposes of testing. Using a separate testing set minimises the possibility of overfitting the function to the idiosyncrasies of the particular training examples available. A test set is difficult to provide if very few instances are available, in which case it is best to use all the instances to maximise the justification for the inference. Where the data does not contain positive examples, or where there is only a small amount of data available, inference can be guided and evaluated by the minimum description length (MDL) principle.

MDL was independently formulated by Rissanen (1978) and Wallace and Freeman (1987) and draws its inspiration from Occam's razor. William of Occam is claimed to have stated the maxim *entia non sunt multiplicanda praeter necessitatem*, which essentially means 'things should not be more complicated than is necessary.' For example, when two competing theories describe the same observations, the simpler one should be chosen. In practice, theories rarely fit observations exactly, so fit must be traded off against complexity. Gaines (1976) provides one tradeoff in his choice of points on the poorness-of-fit vs automaton size curve, choosing the point where there is a large deterioration of fit by allowing one less state in the automaton, but only a small improvement in fit when one more state is allowed. This is an attractive intuitive notion that is unfortunately difficult to define rigorously—particularly the quantification of *large* and *small.*

MDL appeals to Bayes' theorem to provide a more concrete comparison. Let D be the data to be described, and H a hypothesis that explains the data. Bayes' theorem states that

$$p(H|D) = \frac{p(D|H) \cdot p(H)}{p(D)}$$

The 'maximum likelihood' principle chooses the most likely hypothesis for the data—the one where $p(H|D)$ is maximised. The MDL principle is the same as maximum likelihood except that it is usually expressed in terms of description sizes. Shannon (1948) showed that an event that occurs with probability p can be described in $-\log_2 p$ bits. Taking the negative logarithm of both sides of the equation above gives

$$-\log_2 p(H|D) = -\left[\log_2 p(D|H) + \log_2 p(H) - \log_2 p(D)\right]$$

When evaluating competing theories, $p(D)$ is constant, so the best theory minimises $-\left[\log_2 p(D \mid H) + \log_2 p(H)\right]$. This can be interpreted as choosing the theory that minimises the sum of the description length of the theory, $-\log_2 p(H)$, and the description length of the data given the theory, $-\log_2 p(D \mid H)$. These two quantities correspond to Gaines' automaton size and poorness of fit respectively. The advantage of MDL is that both quantities are expressed in the same terms, bits of information, and the evaluation function is a simple sum. Of course, this simplification comes at a price. MDL avoids having to define *small* and *large*, but replaces this decision with the selection of the *a priori* probabilities of the theories and of the data given a particular theory.

MDL's advantage over Bayes' theorem is that instead of talking in terms of vanishingly small probabilities, calculation is in terms of description sizes, and intuitions about efficiency and biases of various encoding schemes help in the choice of probability distributions. For example, some discussion has occurred in the literature about the merits of various coding schemes for decision trees, one of the most popular classes of classification functions in machine learning. Quinlan and Rivest (1989) proposed one coding scheme, which was subsequently analysed and modified by Wallace and Patrick (1993). These discussions centre on the structure of the data and how it could be transmitted efficiently. Because people may be more comfortable with the idea of expressing concepts concisely than assigning extremely small probabilities to them, it is easier to discuss coding schemes in terms of description lengths.

The MDL approach is used in two ways in this thesis. First, it justifies the general approach of the modelling algorithm, which enforces two constraints on the grammar that seek to reduce the overall size of the input sequence. While this is by no means a quantitative application of MDL, it is certainly in the spirit of Occam's razor. Second, it is used quantitatively to rank competing generalisations of a grammar according to their size (section 5.4). MDL is necessary here, because many generalisations are possible, of which the majority are misguided.

2.4 Data compression

Data compression is a pragmatic field that seeks techniques for reducing data storage and transmission. From *ad hoc* beginnings, a robust theoretical framework has been developed that relies heavily on information theory, and relates closely to learning. In the last section, the MDL principle was discussed. It assumes an efficient scheme for encoding a theory and the data given that theory. An inefficient coding scheme reduces the accuracy with which MDL can identify the best theory. One way to encourage efficiency is to provide a representative corpus of example theories and data, and allow researchers to compete and hone their encoding techniques. This is exactly what has taken place for sequences over the last two decades in data compression. Evaluation on a standard corpus measures how successfully a compression scheme captures regularities.

Data compression schemes can be distinguished according to whether they are 'lossy' or 'lossless.' Lossy schemes do not guarantee to reproduce the original data exactly from the compressed version, whereas lossless versions preserve the data exactly. Lossy schemes usually operate on data that has been sampled from an analog source, such as an image or a sound. In this case, people will tolerate some loss of information without noticing any degradation. Furthermore, some noise is introduced in sampling, and it would be wasteful to reproduce this exactly. However, in the sequences we deal with, loss of information might change the meaning of a sequence significantly. Indeed, as the symbols are individually meaningless, it is difficult to know which ones can safely be deleted. Consequently, this discussion centres on lossless compression.

This section discusses the two main approaches to lossless compression: statistical and dictionary techniques. Statistical techniques use finite context models, such as those described in Section 2.1, along with arithmetic coding, which turns predictions into a bit stream. This separation of modelling and coding clarifies the relationship between data compression and learning. Dictionary techniques, on the other hand, explicitly store and reference repeated subsequences, so modelling and coding are intertwined. They are discussed here because they resemble the techniques described in Chapters 3 and 4.

2.4.1 The relationship between learning and compression

As mentioned in the previous section, Shannon (1948) showed that the number of bits required to code a message that occurs with probability p is $-\log_2 p$. For example, assigning all 128 ASCII characters an equal probability of 1/128 leads to $-\log_2(1/128) = 7$ bits per symbol. Often, some symbols have a higher than average probability, e.g. the space symbol in text. In this case, it would be more efficient to assign a shorter code to a space. All probabilities must sum to one, so all of the other symbol probabilities must be reduced, implying longer codes.

It is easy to imagine a symbol having a probability that is not a negative power of two: for example, expecting a space symbol with 10% likelihood. In this case, the optimal code length is about 3.32 bits, which is impossible to encode in isolation. The solution to this problem lies in the fact that most messages contain more than one symbol. In this case, a sequence of three spaces could be coded in ten bits, wasting only 0.04 bits. A message can theoretically be coded in a length that asymptotically approaches that determined by its probability, with the wastage of less than one bit at the end becoming less significant as the message length increases.

In the late 1970s, a practical technique for performing such a coding was discovered, and dubbed *arithmetic coding* (Pasco, 1976; Rissanen, 1976; Rissanen and Langdon, 1979; Rubin, 1979; Guazzo, 1980; Witten *et al.*, 1987). Arithmetic coding operates by ordering all possible messages, and partitioning the interval from zero to one into subintervals for each message. The decoder forms the same partition, and to transmit a message, the encoder must unambiguously identify the subinterval covered by it. This can be done by transmitting a binary fraction that falls within that interval. This requires a number of bits determined by the size of the subinterval, which is the probability of the message. As described here, the method is impractical, because the number of possible messages is potentially very large. An important innovation of arithmetic coding is the ability to code a message incrementally, symbol by symbol. Each symbol narrows the subinterval, and every time the interval halves (which may take several symbols), one bit of the binary fraction can be transmitted. A fuller description of arithmetic coding is given by Bell *et al.* (1990).

Because arithmetic coding can code a message in a number of bits that is arbitrarily close to the negative logarithm of the message's probability, it is possible to separate the coding part of a data compression scheme, which can be performed by arithmetic coding, from the modelling part, which depends on the kind of structure expected in the input messages. This separation takes data compression from the domain of *ad hoc* heuristics that have only pragmatic applications, to the domain of systems that learn and predict. A data compression scheme can be viewed as a theory generator, where the success of the theories that it generates can be evaluated by measuring the size of its output.

The evaluation criterion for compressors has always been clear: the best one is the one that produces the smallest output on the kind of messages that occur in a particular domain. The most popular compression schemes are those that perform well on a wide range of sequences, and corpora have been created to test and compare performance on files including text, graphics and binary data. The most popular general corpus is the Calgary corpus created by Bell *et al.* (1990), which includes English text, program text, a transcript of a computer interaction, seismic data, a bilevel image, a bibliography and executable machine code. Machine learning also has a well-established corpus. The University of California at Irvine maintains a repository of over 80 datasets, most in the form of unordered tuples, and comparison between techniques is based on test set accuracy. Discussions of coding schemes in machine learning tend to be at an intuitive level, with little empirical investigation of the practical effects of coding decisions on a variety of problems. The system described in this thesis represents a bridging of these disciplines: a system that learns and produces comprehensible output in the tradition of decision trees, but that can also be evaluated alongside a huge number of data compression schemes. Of course, SEQUITUR is not alone in relating learning to compression. Cleary's (1980) FLM system was transformed, with the help of arithmetic coding, from a predictive learning system into a practical compression scheme (Cleary and Witten, 1984). It immediately set a prodigious new record for data compression, and today, variations on the algorithm remain at the leading edge of compression technology (Teahan and Cleary, 1996).

2.4.2 Dictionary techniques

Rather than forming a model and using an arithmetic coder, it is possible to take advantage of repetitions directly, by transmitting a repeated subsequence only once and thereafter referring to the single instance whenever it recurs. This idea was proposed by Ziv and Lempel (1977), and forms the basis of most popular data compression schemes. Because it is able to code several symbols at a time, it is faster than symbol-by-symbol statistical compressors. Techniques that use this approach, referred to as LZ77 techniques, operate by specifying an offset into a window of the last few thousand characters. Figure 2.9a shows a nursery rhyme, and Figure 2.9b shows how LZ77 would encode it. The repeated sequence *umpty* in *Dumpty* is replaced by a reference to the occurrence in *Humpty*, using the offset 1 into the sequence, and a length of five. Similarly, the whole sequence *Humpty Dumpty* in the second line can be replaced by a reference to the first fourteen characters: (0,14). Encoding the numbers efficiently results in a smaller representation of the sequence.

There are two problems with the LZ77 approach. First, the pointers are large because they are allowed to index any character in the window, even though most positions are never used. Second, the windowing technique for reducing search means that repetitions that are separated by more than a windowful of symbols cannot be used. To address these problems, Ziv and Lempel (1978) proposed a new scheme, LZ78, which operates by building up a dictionary of phrases and referencing the phrases when repetitions occur. Figure 2.9c shows an artificial sequence, *aaabbabaabaaabab*, and its coding by LZ78. The dictionary starts with one phrase, the null phrase. The entire sequence is encoded as pairs of phrase numbers and symbols, so the first *a* is coded as $(0,a)$—the null phrase with *a* appended. After a

a

```
Humpty Dumpty sat on a wall.
Humpty Dumpty had a great fall.
all the king's horses and all the king's men
couldn't put Humpty together again.
```

b

```
Humpty D(1,6)sat on a wall.
(0,14)had(20,3)gre(15,3)f(24,4)
(24,4) the king's horses and (61,16) men
couldn't put (0,7)toge(66,3)r again.
```

c

input	a	aa	b	ba	baa	baaa	bab
phrase	1	2	3	4	5	6	7
output	(0,a)	(1,a)	(0,b)	(3,a)	(4,a)	(5,a)	(4,b)

Figure 2.9 Dictionary compression schemes
(a) a nursery rhyme
(b) LZ77 coding of (a)
(c) an artificial sequence and its LZ78 coding

pair has been encoded, this new phrase is added to the dictionary. In this case, dictionary entry 1 is the sequence *a*. This can be used in coding the next two characters, *aa*, by using dictionary entry 1 and appending *a*. This results in a new entry, *aa*, being added to the dictionary. Continuing in this way, the dictionary grows by appending a single symbol to an existing entry. New entries are added speculatively in the hope that they will be used later on the sequence. In this case, entry 2 is never used, so its slot in the dictionary is wasted. The advantage of LZ78 over LZ77 is that there are less dictionary entries than possible pointers into the window, and the length of the matches do not need to be transmitted. LZ78 has been shown to be asymptotically optimal under some assumptions about the source, but the rate of convergence is very slow because dictionary builds slowly. For example, in the case of *Humpty Dumpty*, little compression is obtained for the first repetition of *umpty*, because none of the phrases in the dictionary are longer than one character.

The objective of compression is to encode the sequence as efficiently as possible. It has been shown, however, that the problem of finding the dictionary that produces the best compression is NP-complete. Storer (1982) proves this by showing that various schemes are equivalent to either the *node cover problem*, the *restricted node cover problem*, the *K-node cover problem* or the *superstring problem* (references given in the original paper). Here we review the equivalence to the node cover problem. The node cover problem involves choosing a set of nodes from a directed graph such that all edges originate or terminate in at least one of the nodes in the set. It is NP-complete to determine whether this can be done for less than or equal to *K* nodes.

Choosing a dictionary for a text that produces a compressed version of size $\leq K$ is NP-complete. To show this, an instance of the node cover problem is chosen, and the graph is transformed to an example sequence. The sequence is constructed by concatenating the node name for each node, and the names of the originating and terminating nodes for each edge. A special symbol is used to delimit the node names. This string can be optimally compressed by choosing a dictionary of node names that corresponds to the node cover for the graph. The string representation of an edge has at least one node name replaced by a pointer to the dictionary, because the node cover guarantees to contain at least one node that participates in each edge. Similarly, the string representation of every node in the cover can be replaced by a pointer. For the full proof, see Storer (1977). Because the techniques in this thesis can never guarantee to find the smallest grammar for a sequence, evaluation involves more than a single proof: the techniques must be justified empirically in a variety of domains.

A final observation about data compression schemes: Storer classifies macro techniques into those that build new entries out of existing entries (compressed pointer macros, CPM) and those that build entries out of the original text (original pointer macros, OPM). The techniques described in this thesis fall into the CPM category, because the hierarchy is formed by building longer repetitions out of shorter ones. LZ78 and its variants are also CPM schemes, while LZ77 is an OPM scheme. LZ78 builds its phrases from one dictionary entry and one terminal symbol, so its hierarchy is of a restricted form. A variant, LZMW (Miller and Wegman, 1984), forms phrases by concatenating the previous two phrases encoded, which allows phrases to grow more quickly. This produces a balanced hierarchy of phrases, but the opportunistic phrase construction means that many phrases are unused in coding.

2.5 Linguistic segmentation

The sequence that people are required to understand most often is written or spoken language. Linguists have studied these sequences for centuries, and in a sense every person is an amateur linguist, learning to make sense of language. An open question in linguistics is how people acquire language, and an important component of this is segmentation: how people break a string of phonemes into meaningful segments such as words. This section looks at computational models that have been constructed to account for segmentation.

Can people perform segmentation without accompanying semantics? To answer this question, Hayes and Clark (1970) performed an experiment where adult subjects listened to an 'artificial speech analogue' which was constructed by assembling artificial 'phonemes' into a small set of short strings ('words'). They generated these strings in random sequence for as long as required. Tests showed that, after listening to this artificial speech, adult subjects had, in varying degrees, learned to identify the beginnings and ends of the word segments even though there were no cues from pause, intonation or correlation with entities outside the stream of sound.

In discussing possible mechanisms to explain their observations, Hayes and Clark describe a 'subject's eye view' of the experiment:

> The process seems to proceed roughly as follows. At first, the sound stream seems quite amorphous and featureless. After a minute of listening, an event—perhaps a phoneme or part of a phoneme—stands out of the stream. When the event has recurred several times, the listener may notice that it is typically preceded or followed by another event. The combination of events can in turn be related to events that happen in its neighbourhood. Recognition of a word, then, seems to proceed from perceptually distinctive foci outwards in both directions towards the word boundaries. Presumably, the process would tend to stop at word boundaries because the correlations across the boundaries are weak.

This result suggests that it may be feasible to mimic human segmentation ability by recognising repetitions in a stream. It inspired Wolff (1975) to construct 'an algorithm for the segmentation of an artificial language analogue,' called MK10. Rather than applying the algorithm to phoneme sequences, Wolff produced an artificial sequence of words using a simple finite state automaton. The words were not delimited by spaces, and the task of the program was to infer word boundaries by analysing repetitions. MK10 processed the text from left to right, and whenever a digram appeared more than ten times, it was added to a list of elements, the digram counts were set to zero, and the process started again, either from the beginning of the text, or continuing on from the current point. The system successfully formed elements for the twelve words in the sequence, as well as the 64 possible sentences generated by the automaton. It did not form elements that crossed word or sentence boundaries or contained partial words or sentences. This experiment will be further analysed in Section 7.1, where we will see that if the form of the automaton is changed very slightly the sentence formation method fails to work. A better test of its abilities is real text.

To determine whether MK10 is effective on natural language, Wolff (1977) applied it to 10,000 letters from a children's book, 20,000 letters from a novel, and 20,000 letters from speech transcripts. MK10 successfully delimits most words in the text. Wolff relates the word inference process to observations of human learning with respect to segmentation, and makes some interesting comparisons. It is also interesting to consider the structure of sequences of word classes, and Wolff (1980) applies MK10 to a sequence of word classes from the 20,000 letter novel. The novel was hand-tagged with word classes, and the hierarchy produced by MK10 was compared with surface structures assigned by a linguist and the author. The hierarchies were shown to be better than random in a statistically significant way, and correctly identified about half of the structure. However, much of this success is due to identifying pairs such as determiner-noun and adjective-noun, rather than larger structures.

In experiments with English text, the number of entries grows linearly with the size of the sequence. Because MK10 scans the sequence for each entry, run time is quadratic in the size of the sequence. The techniques described in this thesis run in linear time. Moreover, MK10's learning rate based on the size of the sequence is slow, because it requires digrams to appear ten times. This threshold is arbitrary, whereas SEQUITUR forms new rules as soon as a repetition appears.

Another attempt to simulate children's acquisition of segmentation was made by Oliver (1968), described in Brown (1973). The task, as with Wolff, was word discovery from text. His system started with a dictionary containing the 26 letters of the alphabet, as did MK10. On each pass through a 480 character sample, all pairs of elements in the dictionary are added to the dictionary. Brown writes that the program 'finds for each stretch a maximum likelihood parsing into words. For this task certain "shortest path" algorithms from operations research are employed.' It seems that Olivier used some form of optimal parsing, where weights in the parsing graphs come from frequencies for the dictionary entries. This dictionary contains many non-words, but entries are culled if they have only occurred once when the dictionary becomes full. Olivier found that after 550 sections, 45% of entries were words and 30% were groups of words. This approach builds the dictionary much faster than MK10, allowing many errors, but culling removes spurious entries. The repeated optimal parsing using updated probabilities may lead to better results than Wolff's longest-first parsing at the expense of processing time.

2.6 Human sequence learning

One aspect of human behaviour that psychologists have sought to explain is our ability to learn sequences and to perceive sequential structure. A key experiment in this area was conducted by Nissen and Bullemer (1987), and involved pressing buttons in response to asterisks appearing on a computer screen in certain positions. Whenever an asterisk appeared, the subject was required to press the button in the corresponding position. There were four buttons and asterisk positions. In one experiment, the asterisks appeared in a set sequence which repeated after ten steps. In another experiment, the asterisks appeared at random. In the first experiment, subjects became faster with practice, indicating that they had learnt the pattern and were able to anticipate the next position in which an asterisk would appear. In the random experiment, the subjects showed no improvement over time. The research also showed that sequence learning and awareness were not necessarily related. That is, in certain circumstances the subjects could not articulate the structure of the sequence even though they had unconsciously learnt it. This indicates that identifying sequential structure is an in-built cognitive process in humans. Cohen *et al.* (1990) show a more extreme effect: even when subjects are distracted from the sequence learning task by another competing task, they still improve their performance. That is, sequence learning occurs without the subjects' attention to the task. If neither attention nor awareness are essential to sequence learning, it must be a fundamental cognitive skill.

The mechanisms in the brain for learning sequences have been the subject of continuing research. One theory, articulated and investigated by Restle (1970), is that people form internal hierarchies to deal with sequences:

> Human speech can be divided into passages, which in turn divide into sentences, clauses and phrases, words, and speech sounds. Music is divided into movements, sections, themes, and measures. It seems overwhelmingly obvious that long and complex serial patterns are divided into natural subparts, and that mastery is facilitated if the incoming sequence is marked off into natural subparts.

The advantage of recognising such structures is that sequences are easier to remember in this way. Restle says that 'the first characteristic of such a theory is that it provides a concise description of very long sequences, yet gives a kind of structural analysis at the same time.' This is precisely the goal of the techniques in this thesis: to provide compression and explanation simultaneously.

The claim that hierarchies are used to learn sequences was initially made by Lashley (1951) in his article 'The Problem of Serial Order in Behaviour.' The prevailing idea at the time was that sequences were remembered by stimulus-response pairs where the stimulus was either previous parts of the sequence or positional cues. Lashley argued that such associative models were not capable of explaining people's ability to learn sequences, whereas a hierarchical model could. Restle and Brown (1970) tested this hypothesis by performing an experiment similar to Nissen and Bullemer, but using lights rather than asterisks. Because the sequence contained the same light more than once, and was followed by different lights in each instance, the associative theory would predict that the subjects could not learn the association. This is because two different responses would be required for the same stimulus. Restle and Brown called these patterns *branching sequences*. The positional cue hypothesis would predict no difference in ability to learn any of the elements of the sequence. The results indicated that both of these hypotheses were inadequate, and that a hierarchical internal model is a more plausible explanation for sequence learning.

The hierarchical nature of structures inferred from sequences is also indicated by the pauses and errors that occur in the replication of sequences. Rosenbaum *et al.* (1983) show this for a finger tapping experiment where subjects were required to tap certain patterns with the middle and index fingers of both hands. Both pauses and errors are consistent with a depth-first traversal of a hierarchy to execute the patterns.

While these experiments do not exactly parallel the patterns treated in this thesis, they nevertheless lend weight to the usefulness of hierarchies for acquiring and representing sequential structure.

2.7 Summary

This chapter has discussed various aspects of learning structure from sequences: learning from a single sequence; from multiple sequences; evaluating learning based on the size of the inference; compression of sequences; learning linguistic sequences; and human learning of discrete sequences. This sets the background for the rest of the thesis, which concentrates on learning structure from a single sequence, and evaluating the structure in terms of plausibility and conciseness. The techniques are inspired by the recognition of repetition by compression schemes, the building of automata by sequence modelling techniques, the minimum description length principle from machine learning, and the evaluation of linguistic models. But the techniques advance the state of the art by combining grammars with automata, inferring linguistic structure efficiently, devising compression schemes that explain, and inference schemes that compress.

3. Forming a hierarchical grammar

What I tell you three times is true.
The Bellman, from *The Hunting of the Snark* by Lewis Carroll

The thesis statement in Section 1.2 asserts that sequences exhibit hierarchical repetitive structure, and that it can be detected efficiently and incrementally. These claims will be proven in the reverse order: first by building a recognition mechanism, then by applying it to a range of sequences. This chapter describes the mechanism.

Repetition is a fundamental kind of structure in sequences of discrete symbols. Wolff (1975) has shown that detecting repetitions is sufficient to perceive words and sentences in undelimited text. Hayes and Clark (1970) showed that humans find repetition sufficient to identify morphemes in continuous speech. In Chapter 7 we will see that meaningful units in many other sequences can be elicited in a similar way. Moreover, the thesis asserts that repeated subsequences often relate to each other hierarchically—that long repetitions are composed of shorter ones. We have already mentioned the hierarchy in language of phrase, word, word part and letter. Similar hierarchies will be demonstrated in other sequences including music, textual databases and plant descriptions. It is important, however, to distinguish hierarchical repetitions from hierarchies in general. The biological taxonomy mentioned in the introduction of phylum, class, order, family, genus and species is a hierarchy based on *similarity* between members of the taxonomy rather than *repetitions* within a sequence. Similarly, whereas the sentence is normally part of a hierarchical decomposition of language, sentences are rarely repeated verbatim, so do not usually appear in the hierarchies described here.

The thesis requires that repetitions be detected efficiently and incrementally. Any algorithm that examines a set of data in its entirety has complexity at least linear in the size of the data. The technique described here matches this bound and is therefore as efficient as possible in a computational complexity sense. Higher complexity, such as quadratic time, would preclude analysis of very long sequences, as the time to incorporate successive symbols in the sequence would increase linearly. Some of the sequences analysed in this thesis comprise many millions of symbols, and are analysed in several minutes. A naive algorithm running in quadratic time would take years to process the same sequence.

The second constraint is incrementality—the algorithm should, after each symbol, be in the same state as it would if the sequence ended there. That is, there is no lag between a symbol appearing and its incorporation into the hierarchy. This ensures that any use made of the hierarchy at any point benefits maximally from the elicited sequence structure. Incrementality and efficiency are somewhat interdependent: a linear time algorithm means that the average time required to process one symbol does not increase throughout the sequence, so the incremental processing requirements are constant. Furthermore, the incremental inclusion of each symbol into the hierarchy means that all state information is embodied in the hierarchy, rendering an expensive search of pending matches unnecessary.

A context-free grammar provides a natural representation for the hierarchy. Each repetition corresponds to a rule, and rules are allowed to contain non-terminals. These non-terminals reference other rules in the grammar, forming the hierarchy. Furthermore, the sequence and the grammar are not separate entities, with the sequence being restated in terms of the grammar. Rather, the sequence is embodied in the grammar. The rule headed by *S*, the start symbol, (hereafter referred to as rule *S*) expands to reproduce the entire sequence. This simplifies transformations on the grammar, as rule *S* can be treated equivalently to other rules in the grammar. *S*, however, remains distinct as the rule to which new symbols are appended, and is the only rule allowed to occur just once.

The first section of this chapter gives a concise description of the algorithm in terms of the two grammar properties it maintains: digram uniqueness and rule utility. Section 3.2 describes its implementation in more detail, with particular emphasis on efficiency. Section 3.3 shows that the running time and storage requirements are linear in the number of input symbols, while Section 3.4 discusses the algorithm's behaviour on extreme input strings. The final section introduces a naive quadratic time algorithm that is useful for demonstrating the transformations that the efficient technique performs.

3.1 The algorithm

SEQUITUR forms a grammar from a sequence based on repeated phrases in the sequence. Each repetition gives rise to a rule in the grammar, and is replaced by a non-terminal symbol, producing a more concise representation of the sequence. It is this pursuit of brevity that drives the algorithm to form and maintain the grammar, and as a by-product, provide a structural explanation of the sequence.

For example, at the left of Figure 3.1a is a sequence that contains the repeating string *bc*. Note that the sequence is already a grammar—a trivial one with a single rule. To compress the sequence, a new rule $A \rightarrow bc$ is formed, and both occurrences of *bc* are replaced by A. The new grammar is shown at the right of Figure 3.1a.

The sequence in Figure 3.1b shows how rules can be reused in longer rules. It is formed by concatenating two copies of the sequence in Figure 3.1a. Since it represents an exact repetition, compression can be achieved by forming the rule $A \rightarrow abcdbc$ to replace both halves of the sequence. Further gains can be made by forming rule $B \rightarrow bc$ to compress rule A. This demonstrates the advantage of treating the sequence, rule *S*, as part of the grammar—rules may be formed in rule *A* in an analogous way to rules formed from rule *S*. These rules within rules constitute the grammar's hierarchical structure.

	Sequence	Grammar		Sequence	Grammar
a	S → abcdbc	S → aAdA A → bc	b	S → abcdbcabcdbc	S → AA A → aBdB B → bc
c	S → abcdbcabcdbc	S → AA A → abcdbc	d	S → aabaaab	S → AaA A → aab
		S → CC A → bc B → aA C → BdA			S → AbAab A → aa

Figure 3.1 Example sequences and grammars that reproduce them
(a) a sequence with one repetition
(b) a sequence with a nested repetition
(c) two grammars that violate the two constraints
(d) two different grammars for the same sequence that obey the constraints.

The grammars in Figures 3.1a and 3.1b share two properties:

p_1: no pair of adjacent symbols appears more than once in the grammar, and
p_2: every rule is used more than once.

p_1 can be restated as 'every digram in the grammar is unique,' and will be referred to as *digram uniqueness*. p_2 ensures that a rule is useful, so it will be called *rule utility*. These two constraints exactly characterise the grammars that SEQUITUR generates.

For example, the sequence in Figure 3.1a contains the repeated digram *bc*. To conform to property p_1, rule *A* is created, so that *bc* occurs only within rule A. The sequence in Figure 3.1b contains five repeated digrams: the creation of rule *A* reduces this to one, and rule *B* takes care of the remaining repetition. p_2 allows rules longer than two symbols to be formed, as described in Section 3.1.2. To show what happens when these properties are violated, Figure 3.1c gives two other grammars that represent the sequence in Figure 3.1b, but lack one of the properties. The first grammar contains two occurrences of *bc*, so p_1 does not hold for this grammar. In this case, there is redundancy because *bc* appears twice. In the second grammar, *B* is used only once, so p_2 does not hold. If *B* were removed, the grammar would shrink by one rule and one symbol, forming a more concise grammar.

The grammars in Figures 3.1a and 3.1b are the only ones for which both properties hold for each sequence. However, there is not always a unique grammar with these properties. For example, the sequence in Figure 3.1d can be represented by both of the grammars on its right, and they both obey p_1 and p_2. They are both valid grammars, but the topmost one has one fewer symbols, and by virtue of its conciseness is preferred over the other. Chapter 4 discusses the issue of generating and choosing between two such grammars.

SEQUITUR's operation consists of ensuring that both properties hold. When describing the algorithm, the properties will be referred to as *constraints*. The algorithm operates by enforcing the constraints on a grammar: when the digram uniqueness constraint is violated, a new rule is formed, and when the rule utility constraint is violated, the useless rule is deleted. The next two sections describe in detail how this is performed.

3.1.1 Digram uniqueness

When a new symbol is observed, it is appended to rule *S*. The last two symbols of rule *S*—the new symbol and its predecessor—form a new digram. If this digram occurs elsewhere in the grammar, the first constraint has been violated. To remedy this, a new rule is formed with the digram on the right-hand side, headed by a new non-terminal. The two original digrams are replaced by this non-terminal.

Figure 3.2 shows the grammar as new symbols are added in the sequence *abcdbcabcd*. The left-most column states the action that has been taken to modify the grammar—either observing a new symbol and appending it to rule *S*, or enforcing a constraint. The next column shows the sequence observed so far. The third column gives the grammar created from the sequence. The fourth column lists any duplicate digrams, while the final column lists any underused rules.

When the final *c* is added in Figure 3.2a, the digram *bc* appears twice. The new rule *A* is created, with *bc* as its right-hand side. The two occurrences of *bc* are replaced by *A*. This illustrates the basic procedure for dealing with duplicate digrams.

The appearance of a duplicate digram does not always result in a new rule. If the new digram exactly matches the right-hand side of a rule, then no new rule need be created: the digram is replaced by the non-terminal that heads the existing rule. Figure 3.2b demonstrates the changes that occur in the grammar when a third *bc* digram appears. The existing non-terminal *A* is substituted for the third occurrence of *bc*. This results in a new pair of repeating digrams, *Aa*, shown in the last line of Figure 3.2b. In Figure 3.2c, a new rule *B* is formed accordingly, with *aA* as its right-hand side, and the two occurrences of *aA* are replaced by *B*. The right-hand side of this new rule not only contains terminals, but also non-terminals referring to other rules.

The hierarchy is formed and maintained by an iterative process: the substitution of *A* for *bc* resulted in the new digram *aA*, which was itself replaced by *B*. For larger sequences, these changes ripple through the grammar, forming and matching longer rules higher in the hierarchy. The details of this process are given in the section below on efficiency.

	new symbol or action	the string so far	resulting grammar	duplicate digrams	underused rules
a	a	a	S → a		
	b	ab	S → ab		
	c	abc	S → abc		
	d	abcd	S → abcd		
	b	abcdb	S → abcdb		
	c	abcdbc	S → abcdbc	bc	
	enforce digram uniqueness		S → aAdA A → bc		
b	a	abcdbca	S → aAdAa A → bc		
	b	abcdbcab	S → aAdAab A → bc		
	c	abcdbcabc	S → aAdAabc A → bc	bc	
	enforce digram uniqueness		S → aAdAaA A → bc	aA	
c	enforce digram uniqueness	abcdbcabc	S → BdAB A → bc B → aA		
d	d	abcdbcabcd	S → BdABd A → bc B → aA	Bd	
	enforce digram uniqueness		S → CAC A → bc B → aA C → Bd		B
	enforce rule utility		S → CAC A → bc C → aAd		

Figure 3.2 Operation of the two grammar constraints
(a) Enforcing digram uniqueness by creating a new rule
(b) Re-using an existing rule
(c) Forming a hierarchical grammar
(d) Producing a longer rule by enforcing rule utility

3.1.2 Rule utility

Up until now, the right-hand sides of rules in the grammar have been only two symbols long. Longer rules are formed by the effect of the rule utility constraint, which ensures that every rule is used more than once. Figure 3.2d demonstrates the formation of a longer rule. When *d* is appended to rule *S*, the new digram *Bd* causes a new rule, C, to be formed. However, forming this rule leaves only one appearance of rule *B*, violating the second constraint. For this reason, *B* is removed from the grammar, and its right-hand side is substituted in the one place where it occurs. Removing *B* means that rule C now contains three symbols. This is the mechanism for forming long rules: form a short rule temporarily, and if subsequent symbols continue the match, allow a new rule to supersede the shorter rule, and delete the shorter rule. Section 3.2.4 discusses the merits of this apparent inefficiency.

Figure 3.3 summarises the algorithm. Line 1 deals with new observations in the sequence. Lines 2 through 6 enforce the digram utility constraint. Line 3 determines whether the new digram matches an existing rule, or whether a new rule is necessary. Lines 7 and 8 enforce rule utility. Lines 2 and 7 should be triggered whenever the constraints are violated. Section 3.2 demonstrates how this can be performed efficiently.

3.2 Implementation issues

The SEQUITUR algorithm operates by enforcing the digram uniqueness and rule utility constraints. It is essential that any violation of these constraints be efficiently detected. One way to detect duplicate digrams is to scan the entire grammar after each new symbol appears, and after each subsequent modification of the grammar. This process would produce a quadratic time algorithm, as the grammar usually

```
1   As each new input symbol is observed, append it to rule S.

2   Whenever a duplicate digram appears,
3       if the other occurrence is a complete rule,
4           replace the new digram with the non-terminal that heads the other digram,
5       otherwise
6           form a new rule and replace both digrams with the new non-terminal

7   Whenever a rule is used only once,
8       remove the rule, substituting its contents in place of the non-terminal
```

Figure 3.3 The entire SEQUITUR algorithm.

grows in proportion to the size of the input sequence. This section first investigates efficient data structures for storing the grammar to ensure that common operations can be performed quickly. Next it describes efficient techniques for detecting violations of the two constraints, by incorporating checks within low-level operations on the grammar. Section 3.2.3 discusses one apparent inefficiency and examines and rejects other approaches.

3.2.1 Storing and manipulating the grammar

The choice of an appropriate data structure depends on the kind of operations that need to be performed to modify the grammar. The fundamental operations, along with the corresponding line in Figure 3.3, are:

- appending a symbol to rule S (line 1),
- using an existing rule (line 4),
- creating a new rule (line 6), and
- deleting a rule (line 8).

Appending a symbol involves lengthening rule *S*. Using an existing rule involves substituting a non-terminal for two symbols, thereby shortening the rules containing the digrams. Creating a new rule involves creating a new non-terminal for the left-hand side, as well as inserting two new symbols as the right-hand side. After creating the rule, substitutions are made as for an existing rule, by replacing the two digrams with the new non-terminal. Deleting a rule involves moving the contents of a rule to replace a non-terminal, which lengthens the rule containing the non-terminal. The left-hand side of the rule must then be deleted.

Figure 3.4 illustrates each of these functions. To ensure that the lengthening and shortening of rules is performed efficiently, a doubly linked list structure was chosen. Adding or removing elements in a data structure such as an array involves copying all the elements that follow the modification, whereas deletion and insertion in doubly linked lists both require only two and four pointer assignments respectively.

The start and end of the list are connected to a single guard node. The guard node also serves as an attachment point for the left-hand side of the rule, because it remains constant even when the rule contents change. Figure 3.4c illustrates this arrangement. While the guard node is shown to the left of the rule contents, it could equally be shown at the right. Each non-terminal points to the rule it heads, which is not shown in the figure. With these pointers, no arrays are necessary for accessing rules or symbols, because operations only affect adjacent symbols, or rules headed by a non-terminal symbol. The rules, however, are linked together so that they can be traversed for printing.

Creating a new symbol involves allocating space in memory and initialising it appropriately. Changing a link in the diagram involves one pointer operation. Appending a symbol to a rule involves creating one symbol and making four pointer assignments, as shown in Figure 3.4a, where the symbol *b* is appended to a rule ending in *a*. The pointers that change are shown as grey arrows. Using an existing rule involves deleting two symbols (the digram to be replaced), creating one symbol for the new non-terminal, and making four pointer assignments. This takes the sequence *abcd*, and creates the sequence *aAd*. Figure 3.4b shows the substitution of *A* in place of *bc*, with the changed pointers highlighted. Creating a new rule involves creating a new left-hand side for the rule, creating two symbols and a guard

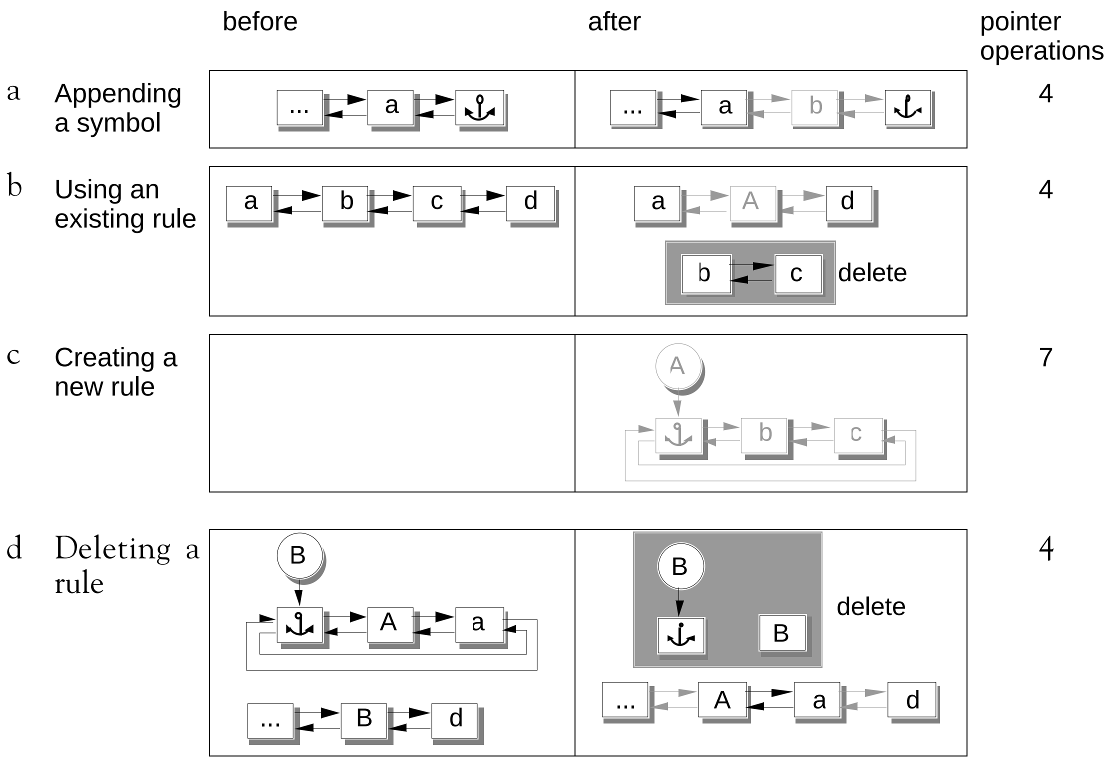


Figure 3.4 The four operations required to build the grammar
(a) appending a symbol,
(b) using an existing rule,
(c) creating a new rule, and
(d) deleting a rule

node, and making seven pointer assignments. This creates the rule $A \to bc$. Figure 3.4c shows the new elements in grey. Deleting a rule requires deleting the left-hand side of the rule, deleting the guard node and the terminal symbol, as well as making four pointer assignments to reposition the contents of the rule. Figure 3.4d shows the new symbols, along with the deleted rule head and old non-terminal. This transforms the sequence *Bd* and the rule $B \to Aa$ to the sequence *Aad*. Note that the number of operations required to move the contents does not depend on the length of the rule. As for garbage collection, because the grammar always grows as more of the sequence appears, no symbol ever need be deallocated. Instead, symbols are kept in a 'spare' rule, where they are inserted or extracted as necessary. This adds a small overhead of a symbol insertion to destroying symbols, and of a symbol removal to creating symbols, but these overheads are smaller than the overhead of memory allocation on the heap.

3.2.2 Maintaining rule utility

The rule utility constraint demands that a rule is deleted if it is referred to only once. One way to enforce this constraint is to scan the entire grammar at each step, counting non-terminal frequencies. However, it is more efficient to record rule frequencies and update them at the same time as symbols are created and deleted.

Every time a non-terminal symbol is created (either allocated on the heap or reused from the list of spare symbols), the frequency for that rule is incremented, and every time a non-terminal is deleted, the frequency is decremented. When the frequency

```
CreateSymbol(symbol)
    allocate space or reuse a previously deleted symbol
    initialise symbol
    if symbol is non-terminal then
        increment frequency of rule headed by symbol
```

```
DeleteSymbol(symbol)
    if symbol is non-terminal then
        decrement frequency of symbol
        if frequency of symbol is less than two then
            delete the rule headed by symbol
    move symbol to the list of spare symbols
```

Figure 3.5 Routines for creating and deleting symbols, incorporating maintenance of rule utility constraint.

of a rule falls to one, the rule is automatically deleted. The frequency of the rule is recorded within the rule data structure, to which the non-terminal symbol has a pointer, obviating the need for an array to index the appropriate rule. The routines for symbol creation and deletion are shown in Figure 3.5, incorporating maintenance of the rule utility constraint.

3.2.3 Maintaining digram uniqueness

The second constraint is more difficult to enforce. When a new digram appears, the grammar must be searched for any other occurrence of it. One simple solution would be to scan the entire grammar each time looking for a match, but this is inefficient, particularly as the grammar grows. A better solution requires an index that is efficient to search, as well as a suitable strategy for checking the index.

The data structure for storing the digram index must permit fast access and efficient addition and deletion of entries. A hash table provides constant-time access, and adding and deleting entries requires little extra work. Because no digram appears more than once, the hash table need only contain a pointer to the first symbol in the single matching digram. The hash table consists of a simple array of pointers. Collisions are handled by *open addressing*, to avoid the allocation of memory that chaining requires. In open addressing, described by Knuth (1968), new entries that collide with existing ones are stored in the next free position in the table. To avoid clustering of entries in one region of the array, a second hash function calculates an offset to jump to find a free position, rather than choosing the next one. This is referred to as *double hashing*. If each symbol object records the table index of the digram it starts, a digram can be deleted from the table by simply clearing that table entry directly. Next, it is necessary to consider the efficiency of updating the index when digrams are created and deleted.

Every time a new digram appears in the grammar, it should be added to the index. A new digram appears as a result of two pointer assignments linking two symbols together in the doubly-linked list (one forward pointer and one back pointer). Thus updating the index can be incorporated into the low-level pointer assignments. A digram also disappears from the grammar when a pointer assignment is made—the pointer value that is overwritten by the assignment represents a digram that no longer exists.

For example, Figure 3.6 shows the example at the end of Figure 3.1a, with the addition of the contents of the digram index. To demonstrate the mechanism for updating the hash table when a new rule is created, the creation will be performed in several steps.

When the second *c* is appended to rule *S*, the digram table shows that *bc* already exists in the grammar, so the rule *A* → *bc* is created. Creating the link between *b* and *c* in the right-hand side for rule *A* updates the entry in the index for *bc* to point to its new location—the hash table now contains a pointer to the symbol *b* at the start of rule A. Temporary pointers are kept to the two other occurrences of *bc* in order to replace them. Next, the first *bc* is removed. This breaks the link between the *b* in the digram and the preceding symbol *a*, so *ab* is removed from the index. It also breaks the link between *c* and the following *d*, so *cd* is removed from the index. Next, *A* is inserted in place of *bc*, creating links between *a* and A, as well as between *A* and *d*, adding these digrams to the index. This process continues, resulting in a correct index of digram pointers, but costing just one indexing operation per two pointer operations.

Having an efficiently maintained index of digrams is of little use if it is checked unnecessarily often. Rechecking the entire grammar whenever a symbol is added is infeasible, and inefficient if large portions of the grammar are unchanged since the last check. In fact, the only parts of the grammar that need checking are those where links have been made or broken. That is, when any of the actions that affect the maintenance of the digram table are performed, the newly created digrams should be checked in the index. Of course, every time a link is created, the digram is *entered* into the index, and this is the very time to check for a duplicate. So if while attempting to add a new digram to the index, an entry is found to be already present, then a duplicate digram has been detected, and the appropriate actions

Action	grammar	change in digrams	digram index
observe symbol ‘c’	S → abcdbc		{ab, bc, cd, db}
make new rule A	S → abcdbc A → bc	bc updated	{ab, bc, cd, db}
substitute A for bc	S → aAdbc A → bc	ab, cd removed, aA, Ad added	{bc, db, aA, Ad}
substitute A for bc	S → aAdA A → bc	db removed, dA added	{bc, dA, aA, Ad}

Figure 3.6 Updating the digram index as links are made and broken

```
1  LinkSymbols(left, right)
2      assign forward and back pointers in left and right to link the symbols
3      look up digram <left, right> in the digram hash table
4      if the hash table entry is blank,
5          assign it the location of this digram and record the table index within left
6      otherwise,
7          deal with the duplicate digram
8      if left contains a hash table index for the digram it previously began,
9          clear the hash table entry
```

Figure 3.7 Routines for updating the digram index when symbols are linked

should be performed. This is the only time when it is necessary to consult the digram index. For example, when a new symbol is appended to rule *S*, the digram thus created must be entered in the table. If an entry already exists for this digram, the digram uniqueness constraint has been violated, and the appropriate response is triggered.

Figure 3.7 describes the entire procedure for using and maintaining the digram index. Line 4 tests whether an entry already exists for the new digram. If it does not, line 5 simply assigns the entry a pointer to the first symbol in the digram. If the entry is not empty, line 7 calls a procedure with the locations of both digrams, so that either a new rule can be created or an existing one reused. In line 8, if the leftmost symbol has an index into the hash table for the previous digram in which it participated, the hash table entry is cleared in line 9.

```
                                                                                      action
1    As each new input symbol is observed, append it to rule S.                          1

2'   Each time a link is made between two symbols                                        2
2a       if the new digram is repeated elsewhere,
3            if the other occurrence is a complete rule,
4                replace the new digram with the non-terminal that heads the rule,       3
5            otherwise,
6                form a new rule and replace both digrams with the new non-terminal      4
6a       otherwise,
6b           insert the digram into the index

7'   Each time a digram is replaced by a non-terminal
7a       if either symbol is a non-terminal that only occurs once elsewhere,
8            remove the rule, substituting its contents in place of the other non-terminal   5
```

Figure 3.8 Outline of the SEQUITUR algorithm

Figure 3.8 shows a more precise description of the SEQUITUR algorithm of Figure 3.3. Line 2 in Figure 3.3, 'whenever a duplicate digram appears,' has been replaced by line 2' in Figure 3.8, containing the specific test based on the formation of a new link. Line 7, 'whenever a rule is used only once,' is replaced by line 7' that tests whenever when a digram is replaced by a non-terminal. Line 2a has been added to detect an existing entry in the digram table, while 6a and 6b record a new digram in the table. Finally, line 7a has been added to explicitly check the frequency of a rule headed by a non-terminal. The numbers at the right are used in the complexity analysis in Section 3.3.

3.2.4 Transient rules

In Figure 3.2d, the rule $B \rightarrow aA$ is formed, but when the next symbol, *d*, appears, rule *B* is subsumed into a longer rule, $C \rightarrow aAd$. It seems inefficient to create a new rule and then to destroy it when the next symbol is observed. One modification to SEQUITUR could postpone the creation of a rule until it is clear that it is necessary. Another possibility is to extend rules like *B*, rather than creating a new rule and deleting the existing one. This section examines these two options and demonstrates why they offer no improvement over the algorithm described so far.

First, let us examine the postponement of rule creation. The idea is to recognise repetition in the grammar, but to put off creating a rule until it is clear that the repetition cannot grow any longer. This occurs when a symbol is observed that does

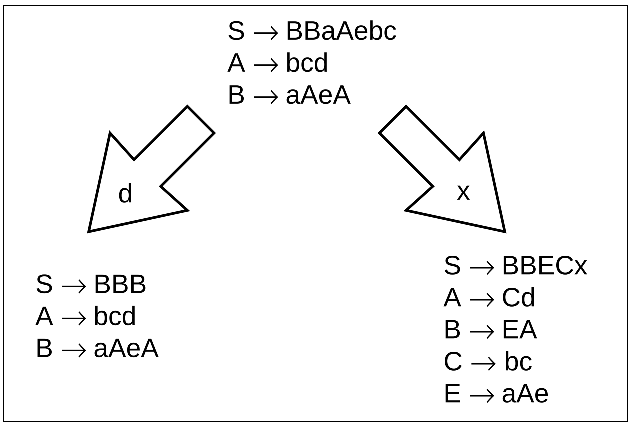


Figure 3.9 Two outcomes for the rule creation postponement approach, after seeing the string *abcdebcdabcdebc*
(a) the grammar with match *bc* awaiting a longer match
(b) the grammar if the awaited *d* occurs
(c) the grammar if *d* does not occur next

not continue the match. At this point, the rule can be created knowing that it will not be superseded by a longer rule. This means that every rule in the grammar is permanent—there are no transient rules.

However, tracking partial matches is difficult. Figure 3.8 shows a situation where two matches are in progress. There is a repeated subsequence *aAe*, which awaits another *A* to complete rule *B*. There is also a repeated *bc*, which expects a *d* to complete rule *A*. If the expected *d* arrives, *A* is exactly matched, which allows the match with *B* to complete. This is illustrated on the left-hand side of Figure 3.8. If some other symbol arrives, however, the match with *A* fails, and a rule *C* must be created to account for the successful match so far. Because the match with *A* failed, the match with *B* cannot succeed, so another rule *E* is created to account for the matched portion. In a large grammar the tracking and recovery from partial matches becomes even more complicated.

Transient rules can be seen as performing this tracking within the grammar. No partial match recovery is necessary because the rules represent the worst case, while allowing longer matches to succeed by disappearing if necessary. Furthermore, at any point the grammar obeys the two constraints, so if the sequence were to end, no further transformations would be necessary. This is useful when using the grammar to predict or explain the sequence: all that is known about the sequence is contained within the grammar, rather than in the partial match tracking data structures. This is the essence of the incrementality of the algorithm. The fact that the state information is completely contained in the grammar rather than in other data structures, and that the grammar is efficiently indexed, means that the algorithm is simple, and that it stores and indexes partial matches efficiently.

The second solution to transient rules is to extend rules instead of replacing them. This introduces a new rule to the algorithm: 'if a digram appears more than once, and its first symbol is a non-terminal, and that symbol appears only twice in the grammar, then append the second symbol in the digram to the rule headed by the first symbol in the digram.' For example, when the digram *Bd* appears, because *B* only appears twice, *d* is appended to rule *B*.

The reason for the stipulation that the rule must appear only twice is this: if the rule is extended, it affects all instances of that rule. For the transformation to preserve the sequence, all instances of the rule must be followed by the same symbol. Because all digrams are unique apart from the duplicate in question, then any other occurrence of the non-terminal apart from the two in the duplicate digrams must be followed by some other symbol. So if the non-terminal occurs more than twice, the rule it heads cannot be extended. The introduction of this new case in the algorithm does not result in the removal of the digram utility rule. Allowing rule extension may increase efficiency by saving one rule creation, but it complicates the algorithm rules unnecessarily—it is superfluous, because its effect is achieved by the combination of the other rules.

3.3 Computational complexity

This section shows that the algorithm is linear in space and time. This fulfils the requirement of the thesis that the detection technique be efficient. The complexity proof is an amortised one—it does not put a bound on the time required to process one symbol, but bounds the time taken for the whole sequence. The processing time for one symbol can in fact be as large as O($\sqrt{n}$) where n is the number of input symbols so far, as shown in Section 3.4. However, the pathological sequence that produces this worst case requires that the preceding O($\sqrt{n}$) symbols involve no formation or matching of rules.

The basic idea of the proof is this: the two constraints both have the effect of reducing the number of symbols in the grammar, so the amount of work done satisfying the constraints is bounded by the compression achieved on the sequence. The saving cannot exceed the original size of the input sequence, so the algorithm is linear in the number of input symbols.

In Figure 3.8, which summarises the SEQUITUR algorithm, the main parts are numbered at the right for reference, and the proof will demonstrate bounds on the number of times that each of them are executed. Action 1 appends symbols to rules S, and is performed exactly n times, once for every symbol in the input. Action 2 is performed when a link is created. Action 3 corresponds to using an existing rule, action 4 to forming a new rule, and action 5 to removing a rule.

Figure 3.10 shows examples of actions 3, 4 and 5, and the savings in grammar size associated with each one. The savings are calculated by counting the number of symbols in the grammar before and after the action. The non-terminals that head rules are not counted, because they can be recreated based on the order in which the rules occur. Actions 3 and 5 are the only actions performed on the grammar that reduce the number of symbols. There are no actions that increase the size of the grammar, so the difference between the size of the input and the size of the grammar must equal the number of times that both these actions have been taken.

More formally, let

n	be the size of the input string,
o	be the size of the final grammar,
r	be the number of rules in the final grammar,
a_1	be the number of times new symbol is seen (action 1),
a_2	be the number of times a new digram is seen (action 2),
a_3	be the number of times an existing rule is used (action 3),
a_4	be the number of times a new rule is formed (action 4), and
a_5	be the number of times a rule is removed (action 5).

According to the reasoning above, the reduction in the size of the grammar is the number of times actions 3 and 5 are executed. That is,

$$n - o = a_3 + a_5 \tag{1}$$

Next, the number of times a new rule is created (action 4) must be bounded. The two actions that affect the number of rules are 4, which creates rules, and 5, which deletes them. The number of rules in the final grammar must be the difference between the frequencies of these actions:

$$r = a_4 - a_5$$

In this equation, r is known, and $a5$ is bounded by equation (1), but a_4 is unknown. Noting that a_1, the number of times a new symbol is seen, is equal to n, the total work is

$$a_1 + a_2 + a_3 + a_4 + a_5 = n + a_2 + (n - o) + (r + a_5) \tag{2}$$

	action	before	after	saving
Matching existing rule	3	...ab... A → ab	...A... A → ab	1
Creating new rule	4	...ab...ab...	...A...A... A → ab	0
Deleting a rule	5	...A... A → ab	...ab...	1

Figure 3.10 Reductions in grammar size for the three grammar operations

To bound this expression, note that the number of rules must be less than the number of symbols in the final grammar, because each rule contains at least two symbols, so

$$r < o$$

Also, from (1):

$$a_5 = n - o - a_3 < n$$

Consequently,

$$a_1 + a_2 + a_3 + a_4 + a_5 = 2n + (r - o) + a_5 + a_2 < 3n + a_2$$

The final operation to bound is action 2, which checks for duplicate digrams. Searching the grammar is performed by hash table lookup. Assuming an occupancy less than, say, 80% gives an average lookup time bounded by a constant (Knuth, 1967). This occupancy can be assured if the size of the sequence is known in advance, or by enlarging the table and recreating the entries whenever occupancy exceeds this amount. The number of entries in the table is just the number of digrams in the grammar, which is the number of symbols in the grammar minus the number of rules in the grammar, because symbols at the end of a rule do not form the left hand side of any digram. So the size of the hash table is less than the size of the grammar, which is bounded by the size of the input. This means that the memory requirements of the algorithm are linear.

As for the number of times that action 2 is performed, a digram is only checked when a new link is created. Links are only created by actions 1, 3, 4 and 5, which have already been shown to bounded by $3n$, so the time required for action 2 is also $O(n)$.

The sum of all the actions in the algorithm is therefore $O(n)$.

3.4 Exploring the extremes

Having described SEQUITUR algorithmically, we now characterise its performance on a variety of data. This section explores how large or small a grammar can be for a given sequence length, as well as determining the minimum and maximum amount of work the algorithm can perform, and the amount of work required to process one symbol. Figure 3.11 summarises these extreme cases, giving part of an example sequence and the grammar that results. Bounds are given in terms of *n*, the number of symbols in the input.

The deepest hierarchy that can be formed has depth $O(\sqrt{n})$, and an example of a sequence that forms such a hierarchy is shown in Figure 3.11a. In the deepest hierarchy, each rule (except the one at the lowest level) must contain a non-terminal, so that the hierarchy deepens at each rule. Furthermore, it is unnecessary for any rule to be longer than two symbols. Therefore, to produce a deep hierarchy from a short string, each rule should be one terminal symbol longer than the one on which it builds. In order to create these rules, the string represented must appear in two different contexts, otherwise the rule will be incorporated into a longer rule. One context is the tallest hierarchy, which it must participate in. The other context should not be in any hierarchy, to reduce the size of the input string, so it should appear in rule *S*. Note that every rule in Figure 3.11a appears both in the hierarchy and in rule *S*. At each repetition of the sequence, one terminal symbol is appended,

		bound	example sequence	example grammar
a	deepest hierarchy	$O(\sqrt{n})$	ababcabcdabcdeabcdef	S → ABCDDf A → ab B → Ac C → Bd D → Ce
b	largest grammar; shallowest hierarchy	n	aabacadae...bbcbdbe...	S → aabacadae...
c	smallest grammar	O(log *n*)	aaaaaaaaaaaaaa...	S → DD A → aa B → AA C → BB D → CC
d	largest number of rules	*n*/4	aaaaababacacadad...	S → AABBCCDD A → aa B → ab C → ac D → ad
e	maximum processing for one symbol	$O(\sqrt{n})$	yzxyzwxyzvwxy	S → ABwBvwxy A → yz B → xA
f	greatest number of rule creations and deletions	*n* new rules *n* deleted rules	abcde abcde abcde...	S → AAA... A → abcde•

Figure 3.11 Some extreme cases for the algorithm

producing a new level in the hierarchy. There is no point in including a repetition of length one, so the m^{th} repetition has length $m + 1$. This repetition gives rise to the m^{th} rule (counting rule S). The total length of the sequence for a hierarchy of depth m is therefore

$$n = 2 + 3 + 4 + \ldots + (m + 1) = \frac{m(m+1)}{2} - 1 = O(m^2)$$

and the deepest hierarchy has depth $m = O(\sqrt{n})$.

At the other end of the spectrum, the grammar with the shallowest hierarchy is shown in Figure 3.11b. It has no rules apart from rule S. It is also the largest grammar for a sequence of a given length, precisely because no rules can be formed from it. The sequence that gives rise to it is one in which no digram ever recurs. Of course, in a sequence with an alphabet of size $|\Sigma|$, there are only $O(|\Sigma|^2)$ different digrams, which bounds the length of such a sequence. This kind of sequence produces the worst case compression: there are no repetitions, and therefore no structure is detected by SEQUITUR.

Turning from the largest grammar to the smallest grammar, Figure 3.11c depicts the grammar formed from the most ordered sequence possible—one consisting entirely of the same symbol. When four contiguous symbols appear, such as *aaaa*, a rule $B \to aa$ is formed. When another four *a*s appear, rule S contains *BBBB*, forming a new rule $C \to BB$. Every time the number of symbols doubles, a new rule is created. The hierarchy is thus $O(\log n)$ deep, and the grammar is $O(\log n)$ in size. This represents the greatest data compression. It is not necessary to have a sequence of only one symbol to achieve this logarithmic lower bound—any recursive structure will do. Chapter 4 discusses a sequence generated by a recursive grammar that produces similar compression. Furthermore, we will see how the grammar can be generalised to the original recursive grammar, compressing a sequence of arbitrary length to a grammar of constant size.

To produce the grammar with the largest number of rules, each rule should only include terminal symbols, because building a hierarchy will reduce the number of rules required to cover a sequence of a given size. Furthermore, no rule should be longer than two symbols or occur more than twice. Therefore each rule requires $2 \times 2 = 4$ symbols for its creation, so the maximum number of rules for a sequence of length n is $n/4$, as shown in Figure 3.11d.

Having discussed the size of grammars, we now move to the effort involved in maintaining them. The upper bound for processing a sequence has been discussed, and shown to be linear. However, it is still useful to characterise the amount of processing involved for each new symbol. Figure 3.11e shows a sequence where the repetition is built up as yz, then xyz, then $wxyz$, etc. Just before the second occurrence of $wxyz$ is completed, no matches have been possible for the w, x, and y. When z appears, yz matches rule A, then xA matches rule B. Finally, wB forms a new rule. This cascading effect can be arbitrarily large if the repetitions continue to be built up in this right-to-left fashion. The amount of processing required to deal with the last z is proportional to the depth of the deepest hierarchy, as the matching cascades up the hierarchy. The maximum time to process one symbol is therefore O($\sqrt{n}$). The fact that w, x, and y failed to match means that they required little

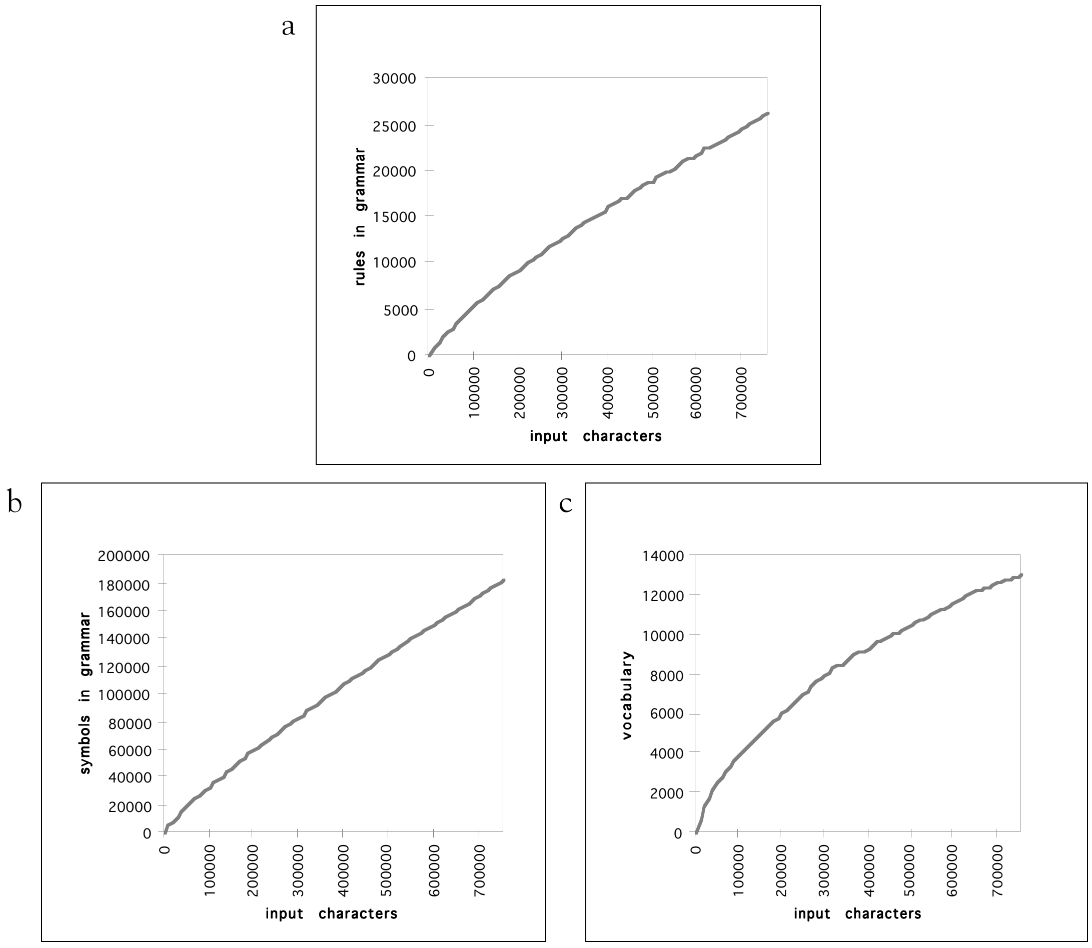


Figure 3.12 Growth rates on English text
(a) rules in the grammar
(b) symbols in the grammar
(c) vocabulary size in the input

time to process, ensuring that the linear time bound overall is preserved.

While the linear bound has been given in order notation, sequences certainly differ in the proportion of work to sequence length. This ratio is minimised by the sequence in Figure 3.11b, where no repetitions exist and no grammar is formed. It is maximised by the sequence in Figure 3.11f, which consists of multiple repetitions of a multi-symbol sequence. Each time the repetition appears, there are several rule deletions and creations as the match lengthens. In fact, every symbol except *a* incurs a rule creation, and a subsequent deletion, so there are $O(n)$ creations and deletions. If m is the length of the repetition, the proportion of symbols that do not incur this work is $1/m$, which tends toward zero as the repetition length approaches infinity.

To give an idea of how SEQUITUR performs on realistic sequences, we turn from extreme artificial cases to a sequence of English text. Figure 3.12a shows that the number of rules in the grammar increases approximately linearly with the number of input symbols, for a 760,000 character English novel. Figure 3.12b shows the linear growth of the total number of symbols in the grammar. The growth of the number of unique words in the text, shown in Figure 3.12c, is high at the start and slows toward the end. It has been observed in much larger samples (Zobel, *et al.*, 1995) that new words continue to appear at a fairly constant rate. In this example, the number of rules grows linearly because once words have been recognised, multi-word phrases are constructed, and the number of such phrases is unbounded.

3.5 A unifying representation

The algorithm described in this chapter detects repetitions and represents them in a hierarchy. There is often more than one way to perform this—Figure 3.1d shows a sequence that can be represented by more than one grammar. In longer sequences, the number of different grammars is potentially very large, and it is useful to be able to enumerate the possibilities. We first show how this enumeration can be performed, then sketch a technique that minimises the number of top level symbols. The enumeration takes time quadratic in the size of the sequence, so this discussion is of mainly theoretical interest: it is not proposed as a practical technique.

3.5.1 Enumerating repetitions

The first step is to identify all the repetitions in a sequence. This can be performed as follows. The algorithm starts by making two copies of the sequence and aligning them one above the other. The overlapping parts are scanned, and the matching parts are recorded in a diagonal line. One of the copies is shifted one symbol to the right, and again matches are recorded. This continues until the sequences fail to overlap.

Figure 3.13 demonstrates this process. With no shift, all the symbols in the sequence match, and the sequence under the horizontal line records the match. To the right is the beginning of the matrix that will record all of the matches, which contains a single diagonal reproducing the result of the match. With the bottom copy shifted right by one symbol, none of the corresponding symbols match, indicated by a row of periods, which is transferred to the matrix at the right. There are no matches after a shift of two symbols. At three symbols, however, three repetitions of *bc* line up, giving the collection of periods and symbols in the string below the line. Another match occurs after the bottom copy has shifted six symbols. This time, the whole overlapping portion matches, because the sequence is an exact repetition of *abcdbc*.

There is one last match between two *bc*s after shifting nine symbols. The resulting matrix records matches in diagonal lines sloping from top left to bottom right. The long match *abcdbc* after shifting six places contains the shorter match *bc*, which can be seen by scanning either vertically or horizontally towards the main diagonal.

This technique is related to—indeed inspired by—the dynamic programming approach to finding the least cost sequence of edits to transform one string into another. In the case where the strings are identical, as is true here, the main diagonal gives a zero-cost sequence. Ignoring this diagonal, however, forces the dynamic programming technique to transform the string to itself by deleting and inserting characters to align repeated subsequences within the string. Because the strings can be interchanged without changing the problem, the matrix is symmetric about the main diagonal, and only half need be shown.

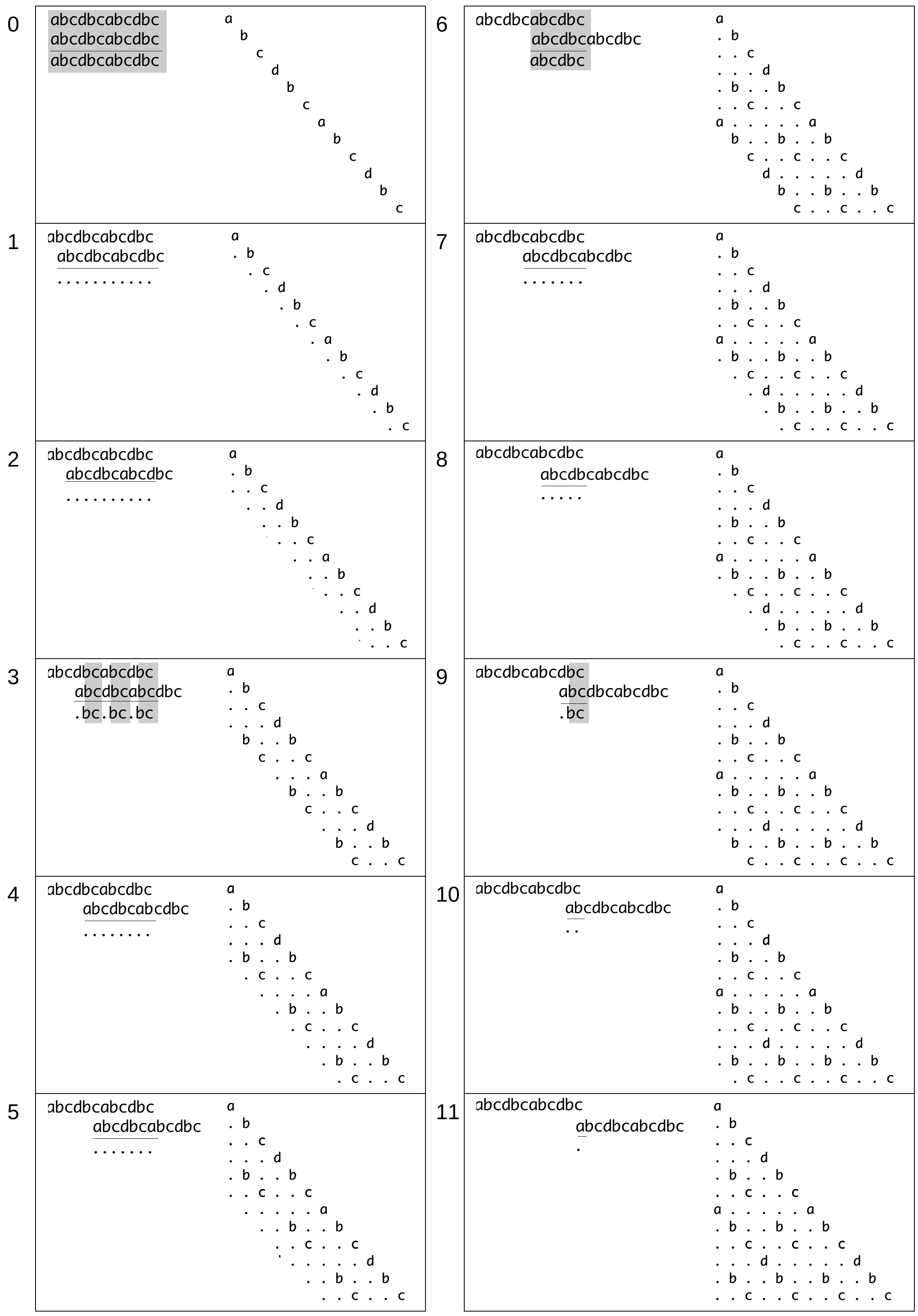


Figure 3.13 Forming a matrix of repetitions

In Figure 3.14, a match is related to the parts of the original sequence in the main diagonal that give rise to it. The vertical part of the grey region indicates where the leftmost instance of *bc* is located, while the horizontal part indicates the rightmost instance. Figure 3.14b shows the longer match *abcdbc*. The distance of the diagonal from the leading diagonal represents the distance from the start of one repetition to the start of the next in the sequence. It is the number of symbols that the sequence has been shifted to map the repetitions on top of each other. Figure 3.14c shows a match between two *bc*s separated by nine symbols. Figure 3.14d shows a hierarchy of matches created by the superposition of the smaller and longer matches. Note how the larger match encompasses the smaller ones.

It is more costly to create a new rule in a grammar than to reuse an existing one. No saving is made in the number of symbols when a rule is created, but when a rule is used, one symbol is saved. Figure 3.14e shows what reuse of a rule means: once the rule has been created, as it has in Figure 3.14a, the grey region is free to continue 'bouncing' between a match in the matrix and the main diagonal. In Figure 3.14e, it does this twice after forming the rule.

3.5.2 Forming a hierarchy

To form a hierarchy from the sequence based on the matrix, rules are formed from some of the diagonal runs. We ignore here the minimisation of the number of distinct rules, which would take reuse of rules into account—that is, we give no preference to covering part of a sequence with a rule that has already been used for another repetition. Instead, the objective is to cover the original sequence with a minimal number of rules, so that it can be re-expressed in the fewest symbols. In this situation, the trade-offs that need to be made are those that involve overlapping. If one repetition is properly contained within another, there is no conflict: this gives rise to a hierarchy. Decisions only have to be made where two repetitions cover the same symbols and cover other symbols as well—when the sets of symbols that they cover intersect, but neither is a subset of the other.

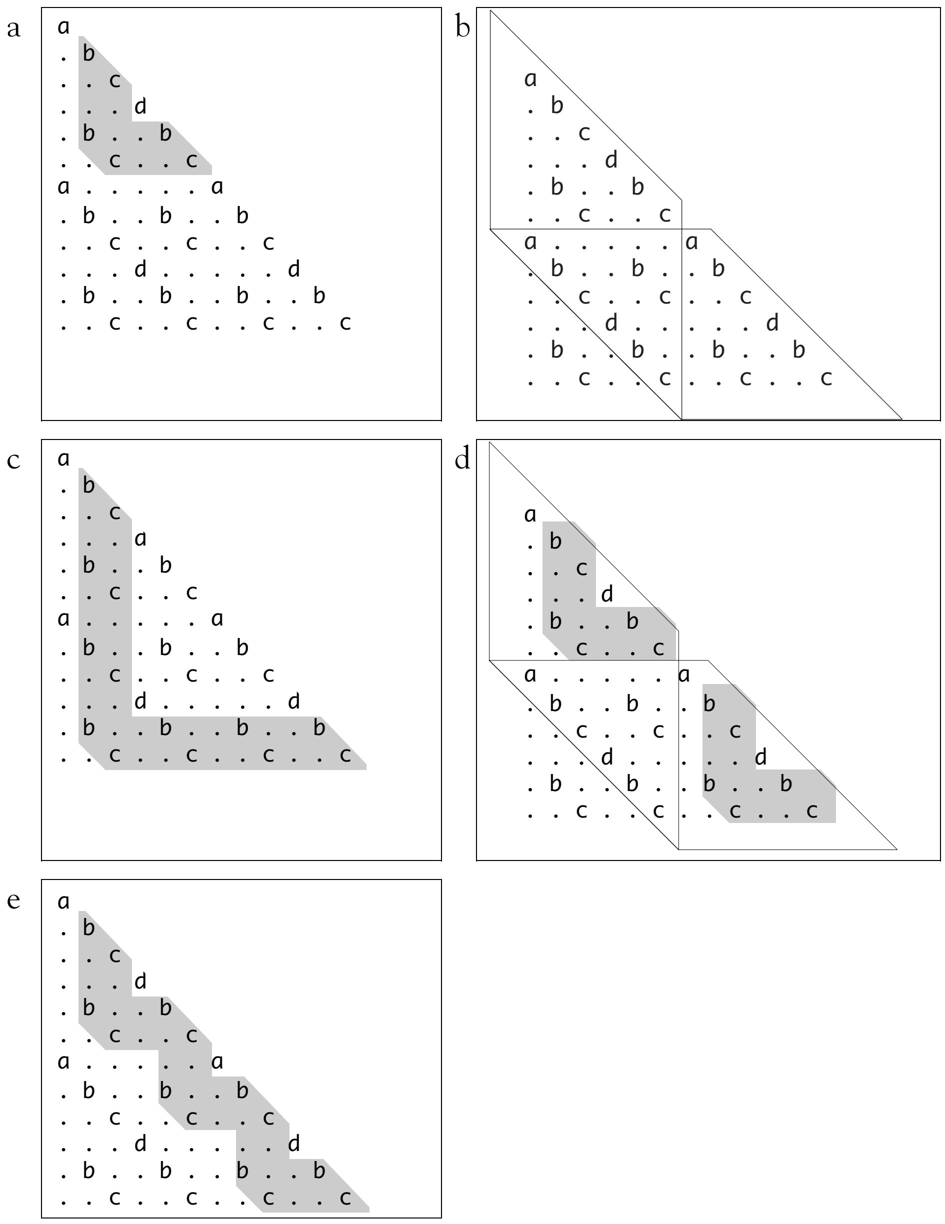


Figure 3.14 Illustration of possible rules
(a) a short repetition over a short distance
(b) a larger repetition
(c) a long-distance repetition
(d) a hierarchy of repetitions
(e) a rule used multiple times

There are two cases that need to be considered. First, if rules overlap so that they cover more than two symbols outside the overlapping symbols, then either rule can be used even though the other covers the intersection. In either case, all of the symbols are covered by a rule. For example, in Figure 3.15a, the sequence *abcde* is to be covered, where the sequences *abc* and *cde* appear elsewhere. Regardless of which repetition, *abc* or *cde*, is chosen to cover the overlapping portion, *c*, the sequence is covered by two rules, because each repetition persists even when shortened by one symbol (bottom part of Figure 3.15a).

The second situation is where a repetition only extends one symbol outside the overlap. In this case, if the other rule is used to cover the overlap, the first rule disappears, because a rule must cover more than one symbol. However, the symbols in question are still covered by two top-level symbols: one is the rule and the other is the single non-terminal that was covered by the now defunct rule. Figure 3.15b shows the sequence *abc* which can be covered by either *ab* or *bc*. If one rule is chosen, the other disappears, but the sequence is still covered by two symbols.

Locally, then, choices between overlapping repetitions make no difference to the number of top-level symbols that are necessary to describe a subsequence. However, the choices have repercussions in other parts of the grammar. Figure 3.15c shows a sequence *abababcbc* that contains the sequence *abc* from Figure 3.15b. As noted previously, *abc* can be covered by either *ab* or *bc*. In the overall picture, however, *ab* occurs three times, whereas *bc* occurs only twice. The impact of this is that if *ab* is used, which obviates the need for *bc*, the other occurrence of *bc* cannot be covered, because every rule must be used at least twice. This results in the entire sequence being covered by six units. If, alternatively, *bc* is chosen to cover *abc*, *bc* is able to cover the other occurrence of *bc*, and the *ab* rule still survives because it occurs twice elsewhere. As a result the sequence is covered by only five units. The key to the overlapping criterion is therefore frequency: if one repetition occurs only twice while a competing one occurs more often, choose the least frequent. Using this criterion to choose between overlapping rules results in an algorithm to form a rule set from the matrix. This algorithm has not been implemented, because the

a	abcde	b	abc	c	abababcbc
	abc de		ab c		ab ab ab c b c
	ab cde		a bc		ab ab a bc bc

Figure 3.15 Effects of choosing between overlapping rules
(a) both repetitions survive: no advantage either way
(b) one repetition disappears: no advantage either way
(c) global considerations give advantage to using infrequent repetitions

quadratic time of the matrix formation rules this technique out for efficiency reasons. It is likely to outperform SEQUITUR in terms of minimising the length of rule *S*, but we are usually interested in the size of the entire grammar, including the other rules.

The constraints on finding the best set of rules are more complicated when the number of rules is required to be minimised. In fact, as discussed in Section 2.5, the problem of minimising the total size of the grammar is NP-complete. The next Chapter discusses two efficient heuristics for reducing the grammar size.

3.6 Summary

This chapter has introduced the SEQUITUR algorithm, which infers a hierarchy of repetitions from a sequence. The algorithm can be specified in terms of two constraints that encourage a reduction in the size of the hierarchy. We have shown that the algorithm can be efficiently implemented, and enumerated some extreme cases. Finally, a quadratic-time algorithm has been described that exhaustively discovers repetitions in the sequence. The next chapter builds on the SEQUITUR algorithm to improve parsing in certain structured situations.

4. Improving the grammar

The thesis requires an efficient algorithm that can be applied to explain and compress a sequence. Chapter 3 has described such an algorithm, which performs well in many situations. In some cases, however, other hierarchical grammars explain and compress the sequence better than the one formed by SEQUITUR. One of these cases will be described in this chapter to motivate two modifications to the SEQUITUR algorithm that improve its explanatory and compressive facilities.

Figure 3.1d gives an example of alternative grammars for a sequence. SEQUITUR produces the grammar at the bottom of Figure 3.1d. The one above it is smaller, and more appealing as a description of the sequential structure. The reason that SEQUITUR does not find it is its greedy parsing. If, instead of creating the rule $A \rightarrow aa$ as soon as the first repetition of *aa* is seen, the rule creation were postponed until seeing the second repetition of *aa*, the grammar at the top of Figure 3.1d would have been formed. However, after seeing just the first repetition, there is no way of knowing that postponement would lead to a better grammar.

One problem with improving SEQUITUR's performance is determining when the resulting grammar has improved—that is, evaluating one grammar relative to another. This chapter uses three evaluation techniques in different situations to determine how good a grammar is: comparison with the source structure, the size of the grammar, and domain-specific evaluation. The case study that will be described is a sequence that is produced by a grammar-like rewriting system called an L-system. Because the original grammar is known, it is possible to evaluate SEQUITUR's output by comparing it directly with the source. However, the source grammar is not always known, and other evaluation metrics are necessary in these situations. The most compelling metric from the point of view of Occam's razor is to choose the smallest grammar that describes a sequence. In this chapter, the size of a grammar will be determined by counting the rules and symbols in it. For now, this symbol counting approach will be left unjustified, but Chapter 6 shows that the number of symbols is proportional to the size of the grammar when encoded in a principled way. The third evaluation method is to use some domain knowledge to evaluate a grammar—for example, counting the correctly partitioned English words in the rules of a grammar for a sequence of text. Whereas size is an objective measure, and is often useful, small grammars do not always correspond to an intuitive notion of how the sequence should be parsed.

Equipped with an appropriate evaluation metric, it is possible to consider modifications to the SEQUITUR algorithm to produce better grammars. The main weakness of the algorithm is that it performs greedy parsing—that is, it forms rules based on the sequence seen so far, rather than waiting until the end of the sequence to see if a different parse would produce a better grammar. This approach is consistent with incrementality—if the algorithm were to postpone making a decision about rule formation, its predictions and the explanatory value of the grammar would not improve during the postponement. This chapter presents two solutions to the problem. The first employs domain-specific knowledge to substitute for clairvoyance about the unseen sequence. If there are general rules about how the sequence should be parsed, they can be employed to substitute for knowledge about the overall sequence. The modified algorithm will be distinguished from the SEQUITUR algorithm described in Chapter 3 by referring to it as SEQUITUR-K, for knowledge-based. The second situation, when domain knowledge is unavailable, acknowledges that there is insufficient information about how to parse the sequence at the time of parsing. However, later in the sequence it may become clear that an earlier parse was incorrect. Thus the second technique performs retrospective reparsing when evidence appears for a different partitioning of the earlier sequence. This modified algorithm will be called SEQUITUR-R, for reparsing.

This chapter is structured in the following way. Section 4.1 previews results from the chapter. Section 4.2 gives an overview of L-systems, the example that will motivate the developments in this chapter. Section 4.3 discusses the use of domain knowledge to improve parsing, while the retrospective reparsing is discussed in Section 4.4.

4.1 Preview of results

This chapter will be motivated with a sequence produced by a simple grammar. Figure 4.1a shows an example of an L-system, a class of rewriting systems. A sequence is produced from this L-system, and a grammar is formed from it using the SEQUITUR algorithm, as shown in Figure 4.1b. The reconstructed grammar bears little resemblance to the original. By employing knowledge specific to the L-system domain, it is possible to influence SEQUITUR's rule-building mechanism to produce the grammar in Figure 4.1c from the same sequence. It is smaller than the grammar in Figure 4.1b, and exhibits some similarities to the original grammar. Rules *S*, *D*, *H* and *L* correspond to the rule *f* → *f*[+*f*]*f*[–*f*]*f* in the original grammar at different levels of recursion. The size of the grammar grows logarithmically with the size of the sequence. Even better results can be achieved by adjusting the initial parse of the sequence as more of the sequence becomes available. In this example, the grammar in Figure 4.1d is produced, which is as close to the original grammar as any grammar from its class can be. Rules *S*, *A*, *B*, and C represent the original rule at different levels of recursion, while rules *D*, *E*, *F*, and G compress the repeated *f*]*f* pattern in the original rule. Inferring the original recursive, non-deterministic grammar from the non-recursive version will be discussed in Section 5.1.

4.2 Inferring L-systems from example strings

In order to demonstrate first the problems and then some solutions to the greedy, incremental aspects of the SEQUITUR algorithm, this section introduces a class of rewriting systems called L-systems. L-systems are used in computational biology and computer graphics to simulate natural objects. Because the source of the sequences is well understood, it is straightforward to compare the grammars that SEQUITUR produces to the real model. Furthermore, because L-systems have practical applications, this chapter provides some demonstrations of SEQUITUR's utility. We

a
```
S → f
f → f[+f]f[-f]f
```

b
```
S → CEJLIJOMNIGPOPDf
A → f[
B → DA
C → IF
D → f]
E → B+
F → B-
G → HK
H → CD
I → A+
J → GM
K → EF
L → EH
M → FG
N → FH
O → DKLK
P → NK
```

c
```
S → LIK]LJK]L
A → [+f
B → [-f
C → A]fB]f
D → fC
E → AC
F → BC
G → E]DF]D
H → DG
I → EG
J → FG
K → I]HJ]H
L → HK
```

d
```
S → AHDID
A → BHEIE
B → CHFIF
C → fHGIG
D → A]A
E → B]B
F → G]G
G → f]f
H → [+
I → [-
```

Figure 4.1 Summary of results for the chapter
(a) an L-system,
(b) the grammar formed from the output of (a) by SEQUITUR,
(c) the grammar formed with domain knowledge,
(c) the grammar formed by retrospective reparsing.

first describe the characteristics of L-systems, and then discuss the issues involved in comparing L-systems with the grammars that SEQUITUR generates.

4.2.1 L-systems

L-systems are a special class of grammars that can model the growth of living organisms. They were devised by a biologist, Aristid Lindenmayer, after whom the grammars were named (Lindenmayer, 1968). They produce sentences that can be interpreted graphically to produce images of fractals or organisms. Prusinkiewicz and Hanan (1989) describe how L-systems are interpreted, and distinguish them from Chomsky grammars:

> The essential difference between Chomsky grammars and L-systems lies in the method of applying productions. In Chomsky grammars productions are applied sequentially, one at a time, whereas in L-systems they are applied in parallel and simultaneously replace all letters in a given word. This difference has an essential impact on the properties of L-systems. For example, there are languages which can be generated by context-free L-systems (called 0L-systems) but can not be generated by context-free Chomsky grammars.

One interesting feature of L-systems is that there is no distinction between terminal and non-terminal symbols. All symbols that appear in the grammar are valid in the final string, and any symbol in the alphabet can head a rule. The evaluation of Chomsky grammars stops when only terminal symbols appear in the string. Because this stopping criterion cannot be used with L-systems, a string is generated by

a

f	Move forward one unit, drawing a line
–	Turn left
+	Turn right
[	Remember position and orientation
]	Recall former position and orientation

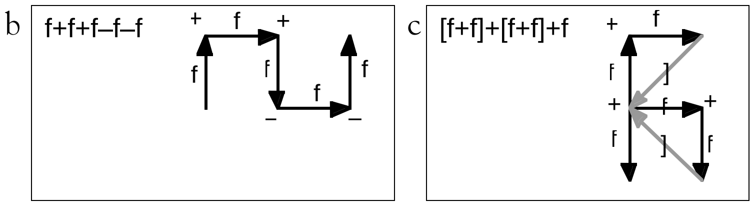


Figure 4.2 The LOGO language
(a) Five turtle commands and their interpretation
(b) interpretation of a turtle command sequence
(c) the effect of state save and restores shown in gray

specifying a number of derivation steps, essentially the depth of the derivation tree.

L-systems may be context-free or context-sensitive, and either deterministic, non-deterministic, or stochastic. They have been used in various ways to describe plant development topologically (Hogeweg and Hesper, 1974; Smith, 1978; Smith, 1984; Szilard and Quinton, 1979; Frijters and Lindenmayer, 1974). They have also been used to describe plants graphically, where sentences produced by L-systems are interpreted as instructions to a LOGO turtle, as described by Prusinkiewicz (1986).

The LOGO language, devised by Seymour Papert (Abelson and deSessa, 1982), defines a set of instructions that direct a software turtle to draw graphical figures. Figure 4.2a summarises them: *f* draws a line in the current direction, + and – turn left and right respectively, and [and] save and restore the turtle's state.[2] For example, Figure 4.2b shows the graphical interpretation of *f*+*f*+*f*–*f*–*f*. This pattern consists of five line segments with left and right turns between them. Figure 4.2c shows the effect of the save and restore instructions in the string [*f*+*f*]+[*f*+*f*]+*f*. The initial orientation and position of the turtle are saved by the first [instruction. After the first two line segments are drawn, the turtle returns to this state when it executes the] instruction. The next two line segments start from this point, as does the final one.

A simple L-system can produce complex figures. Figure 4.3a shows an L-system and Figure 4.3b illustrates the sequence produced at each derivation step. Figure 4.3c shows the corresponding graphical form—a tree-like picture (Figures 4.3d and 4.3e will be referred to in Section 4.2.1). In this case, the angle through which the turtle turns when executing – or + is set to 30°. At step zero on the left, the string contains just one move forward command.

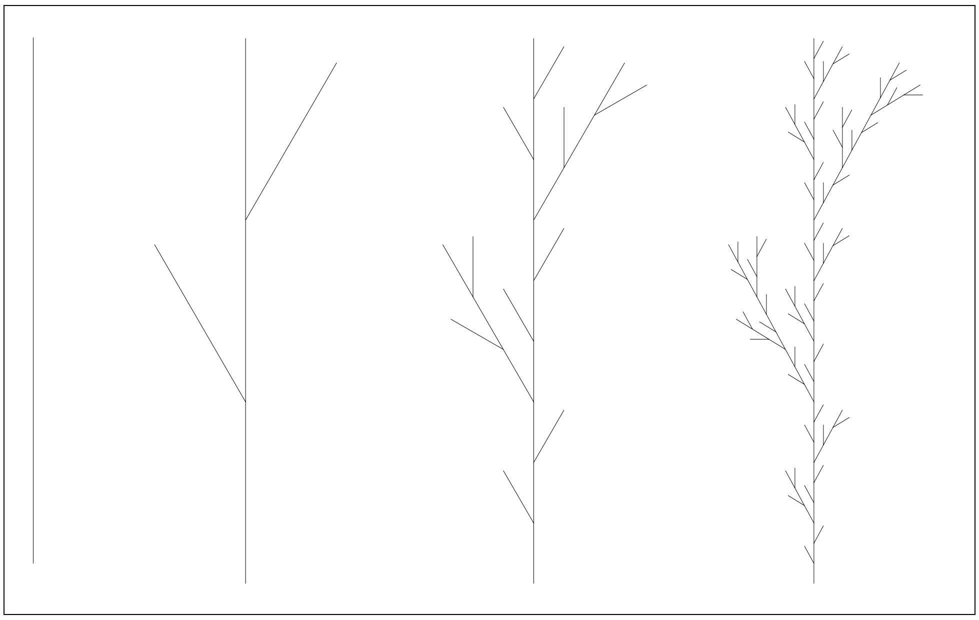
After the first application of the rewriting rule for *f*, the sequence is *f*[+*f*]*f*[–*f*]*f*, which is graphically interpreted as the prototypical plant form in the second column. The scale has been reduced for the purposes of presentation: one line segment in the second diagram is only one third as long as the line segment in the first diagram. Subsequent applications of the rewriting rule produce more complex and realistic images. A more complex L-system is shown in Figure 4.4.

2 Normally an uppercase *F* denotes moving forward and drawing a line, while *f* denotes moving without drawing. The latter symbol is useful for drawing disconnected figures. To avoid confusion of *F* with non-terminals in grammars, *f* is used to mean the drawing version, and *f̲* is used for the non-drawing version.

a
```
S → f
f → f[+f]f[-f]f
```

b
```
0       1                2                        3
f       f[+f]f[-f]f      f[+f]f[-f]f[+f[+f]f[-f]f f[+f]f[-f]f[+f[+f]f[-f]f
                         ]f[+f]f[-f]f[-f[+f]f[-f  ]f[+f]f[-f]f[-f[+f]f[-f
                         ]f]f[+f]f[-f]f           ]f]f[+f]f[-f]f[+ff[+f]f
                                                  [-f]f[+f[+f]f[-f]f]f[+f
                                                  ]f[-f]f[-f[+f]f[-f]f]f[
                                                  +f]f[-f]f]f[+f]f[-f]f[+
                                                  f[+f]f[-f]f]f[+f]f[-f]f[
                                                  -f[+f]f[-f]f]f[+f]f[-f]f
                                                  [-f[+f]f[-f]f[+f[+f]f[-
                                                  f]f]f[+f]f[-f]f[-f[+f]f[
                                                  -f]f]f[+f]f[-f]f]f[+f]f
                                                  [-f]f[+f[+f]f[-f]f]f[+f
                                                  ]f[-f]f[-f[+f]f[-f]f]f[
                                                  +f]f[-f]f
```

c



d
```
S → f   S → f[+f]f[-f]f      S → A[+A]A[-A]A          S → A[+A]A[-A]A
                             A → f[+f]f[-f]f          B → f[+f]f[-f]f
                                                      A → B[+B]B[-B]B
```

e
```
S → f   S → f[+X-X           S → BDAEA                S → BFAGA
        X → f]f              A → B]B                  A → B]B
                             B → DDCEC                B → DFCGC
                             C → f]f                  C → D]D
                             D → [+                   D → fFEGE
                             E → [-                   E → f]f
                                                      F → [+
                                                      G → [-
```

Figure 4.3 Derivation of an L-system
(a) the L-system
(b) the sequence produced at each derivation step
(c) the graphical representation
(d) the equivalent non-recursive grammar
(e) SEQUITUR's constraints enforced on (d)

```
S → f
f → ff+[+f-f-f]-[-f+f+f]
```

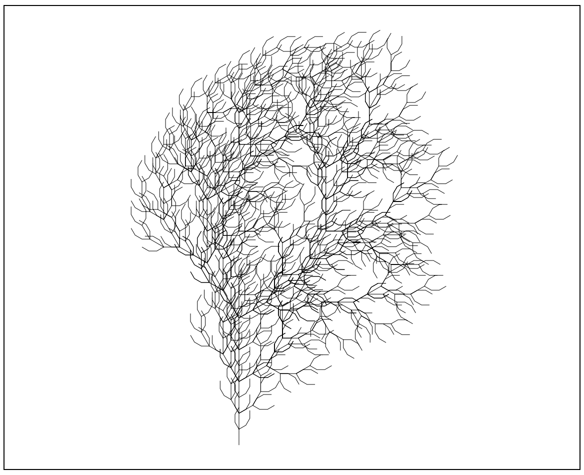


Figure 4.4 Another plant image

4.2.2 Comparing L-systems with SEQUITUR's output

L-systems produce a sequence of symbols, while SEQUITUR produces a grammar from a sequence, so at a broad level SEQUITUR performs the inverse of L-system evaluation. An interesting empirical question is whether SEQUITUR can exactly reconstruct an L-system from its output. There are four fundamental difficulties that stem from the mismatch of grammar classes between L-systems and the output of SEQUITUR. First, SEQUITUR's grammars are non-recursive, whereas most useful L-systems are recursive. This means that SEQUITUR can only ever produce a non-recursive approximation to the original grammar. Second, L-systems may violate the two constraints on the grammars that SEQUITUR produces, which makes them difficult to compare directly with SEQUITUR's output. Third, L-systems may be stochastic and context-sensitive, whereas SEQUITUR's grammars are deterministic and context-free. Finally, in L-systems, terminals may head rules.

The L-system in Figure 4.3a exhibits three of these problems. First, it is recursive: the rule headed by *f* contains the symbol *f*. Second, it does not obey the constraints: the strings *f*[*f* and *f*[both appear twice. Third, the terminal symbol (in Chomsky terminology) *f* heads a rule.

The first step in remedying these problems is to restate the L-system in a non-recursive way. Because it is recursive, and there is no distinction between terminal and non-terminal symbols, it is necessary to specify the number of times the rewriting rule is applied in order to generate a sequence. Choosing three steps results in the right-most sequence in Figure 4.3b. A recursive L-system along with a derivation length has an equivalent non-recursive version, which is shown for each derivation step in Figure 4.3d. Each non-recursive grammar reproduces the same sequence as the recursive version evaluated to that number of steps, and also maintains a resemblance to the original grammar. In Section 6.1, this resemblance will be exploited to form a recursive version from the non-recursive one, but for now the aim will be for SEQUITUR just to reproduce the non-recursive version.

Producing a non-recursive version for a particular number of derivation steps solves two of the three difficulties in the original grammar: as a by-product of producing a non-recursive grammar, the grammar can now be interpreted as a Chomsky grammar, treating *f* as a terminal symbol, as it no longer heads a rule. The third difficulty is that the L-system does not obey the digram uniqueness constraint. This can be remedied in two ways, by replacing either *A*[or *A*]*A* (or the equivalent sequence in the other rules) with a rule. The result of the second, more compressive replacement is shown in Figure 4.3e for each level of evaluation.

The grammars are now in a form compatible with SEQUITUR's output, and the question of SEQUITUR's ability to reproduce such modified L-systems can now be addressed. Figure 4.5 shows the grammar produced by SEQUITUR given the sequence in Figure 4.3b at the third derivation step. The second column in Figure 4.5 gives

```
S → CEJLIJOMNIGPOPDf

A → f[       f[
B → DA       f]f[
C → IF       f[+f]f[-
D → f]       f]
E → B+       f]f[+
F → B-       f]f[-
G → HK       f[+f]f[-f]f]f[+f]f[-
H → CD       f[+f]f[-f]
I → A+       f[+
J → GM       f[+f]f[-f]f]f[+f]f[-f]f[-f[+f]f[f]f]f[+f]f[-
K → EF       f]f[+f]f[-
L → EH       f]f[+f[+f]f[-f]
M → FG       f]f[-f[+f]f[-f]f]f[+f]f[-
N → FH       f]f[-f[+f]f[-f]
O → DKLK     f]f]f[+f]f[-f]f[+f[+f]f[-f]f]f[+f]f[-
P → NK       f]f[-f[+f]f[-f]f]f[+f]f[-
```

Figure 4.5 The grammar produced by SEQUITUR from the sequence in Figure 4.3b, step 3

the substrings that the rules represent. This grammar bears little resemblance to the grammar in 4.3e: it is much larger, and lacks a rule representing the fundamental *f*[+*f*]*f*[–*f*]*f* building block. It appears that SEQUITUR is incapable of recognising the structure in the sequence.

The cause of this poor performance is the greedy left-to-right processing. Once SEQUITUR forms a rule, the two symbols in that rule are bound together for the rest of the processing. The rule may be incorporated into a longer rule, but this longer rule will still group the two symbols together. For example, the initial sequence *f*[+*f*]*f*[–*f*]*f* should be in a single rule on its own. However, the [following this sequence means that SEQUITUR takes advantage of the *f*]*f*[repetition and forms a rule for it. Figure 4.6a shows the preferred parse and Figure 4.6b shows the actual parse after SEQUITUR has seen the first 24 symbols. In the preferred parse, one rule covers both instances of *f*[+*f*]*f*[–*f*]*f* (only the rules that appear in rule *S* are shown). In Figure 4.6b the darker grey area denotes the incorrect rule covering *f*]*f*[, which spans the end of one rule and the [from one level above.

The next two sections describe two distinct solutions to this problem: using domain knowledge to aid SEQUITUR's selection of phrases, and performing retrospective reparsing in order to improve SEQUITUR's performance.

4.3 Domain knowledge

One solution to the incorrect partitioning of the sequence into rules is to use knowledge about the preferred form of rules to help SEQUITUR choose the correct rules in the first place. In the simple example illustrated in Figure 3.1d, ignoring the first repetition of *aa* would result in a better overall grammar. If an oracle were

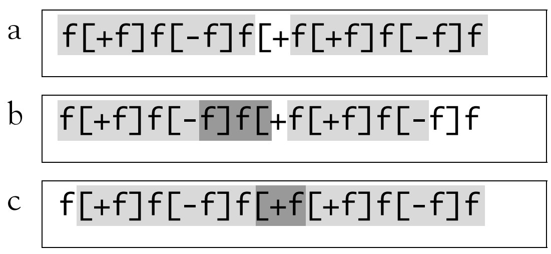


Figure 4.6 The first part of the sequence from Figure 4.3b, step three
(a) parsed correctly
(b) parsed incorrectly
(c) an improved parsing using bracket nesting constraint

available to advise SEQUITUR whether or not to form a rule, better grammars could be produced.

4.3.1 Restricting rules using domain knowledge

In L-systems that contain square brackets (save and restore state instructions), each rule has correctly nested brackets. That is, for every restore instruction, there is a corresponding save instruction earlier in the rule, ensuring that there is always a state to restore. It is easier for SEQUITUR to recognise the structure of the source if this constraint is taken into account. To ensure that every rule in SEQUITUR's grammar has correctly nested brackets, a further constraint is imposed on the grammar: in the expansion of every rule,

1. either all brackets must match, or
2. all close brackets must have a matching open bracket, and
   the rule must start with an open bracket.

The first part corresponds to the domain knowledge about the original L-system. In the second part, the nesting constraint is relaxed to allow rules that have more open that close brackets. This clause exists for the following reason. Because rules are grown from left to right, rather than created in one action, they always start by consisting of two symbols. The only rule containing brackets that would be able to be created under this more stringent requirement would be []. Any other rules, such as [+, would be disallowed, so in fact no rules at all would be formed for this sequence. At the same time as the matching is relaxed, an additional constraint is added: an unbalanced rule must begin with an open bracket. This is to provide a starting point for a rule. In other words, whereas rule 1 and the first clause of rule 2 stop a rule from growing (when the next symbol is a close bracket with no matching open bracket.), the second clause of rule 2 controls when one may be formed.

Enforcing this constraint on SEQUITUR's rules results in an improved grammar which includes the decomposition of Figure 4.6c. For example, the sequence *f*[+*f*]*f*[-*f*]*f* in Figure 4.6a is well-formed, and can be evaluated on its own to produce a graphical figure. The prefix *f*[+*f*]*f*[–, the first rule in Figure 4.6b, leaves a superfluous state on the stack, but can nevertheless be drawn without raising an error. However, the sequence *f*]*f*[, the second rule in Figure 4.6b, generates an error when it is evaluated, because the first restore state command lacks a corresponding save state. Rejecting this rule—specifically the initial *f*]—allows the first light gray rule to be extended later to include *f*], resulting in the parse of Figure 4.6c. When this constraint is applied at each stage of grammar formation, the grammar in Figure 4.7 is produced.

SEQUITUR can be modified to ensure that grammar rules always obey the save/restore nesting constraint by checking at the very core of the algorithm, within the digram-indexing procedure. Recall that every time a new digram appears, the digram index is consulted for an identical digram elsewhere in the grammar. If one is found, a new rule is created, or an existing one re-used. If none is found, no action is taken. The modification to the algorithm is this: if the digram would produce a rule which violates the nesting constraint, then the indexing procedure will return false, even if a duplicate digram exists. This prevents the formation of rules that violate the nesting constraints. This restriction relieves the digram uniqueness constraint, because it allows duplicate digrams to remain in the grammar. Ignoring a duplicate digram is dangerous—if the second symbol in the digram does not eventually participate in another rule, then the final grammar will not obey the constraints. In structured sequences, however, postponing rule creation will allow another digram to match instead.

To illustrate the restriction, if the first and second symbols in the digram are [and] respectively, the digram-finding routine will operate as normal, returning a matching digram if one exists. However, if the digram consists of] followed by [, the routine will return false, even if a match exists, so that the badly-nested rule will not be created. In fact whenever the first symbol is], the rule cannot succeed. The symbols involved could be non-terminals. If the first symbol is a non-terminal that expands to [+*f*], and the second symbol is], then the rule will not succeed, because the first symbol is exactly balanced, and the second symbol has no corresponding

```
S → LIK]LJK]L

A → [+f
B → [-f
C → A]fB]f
D → fC
E → AC
F → BC
G → E]DF]D
H → DG
I → EG
J → FG
K → I]HJ]H
L → HK
```

Figure 4.7 Grammar induced from Figure 4.3b using background knowledge

save instruction. However, if the first symbol expands to [+*f*, then the rule succeeds, because the second symbol balances the [in the expansion of the first symbol.

4.3.2 Results of the knowledge based restriction

Rules C, G, and *K* in Figure 4.7 all begin with [and are perfectly balanced. Each participates in another rule (*D*, *H*, and *L* respectively) which prefixes them with a rule that does not start with [. It is these latter three rules that expand to the three derivation steps in the L-system. Rules *A*, *B*, *E*, *F*, *I* and *J* are merely compressive: they take care of the [+ and [– sequence at each level. The grammar in Figure 4.7 is not identical to the grammar in Figure 4.3e. However, it captures the structure of the L-system: the three derivation steps are clear from the three sets of rules. This use of background knowledge has a dramatic effect on the size of the grammars that SEQUITUR induces from L-system sequences. The grammar produced by SEQUITUR in Figure 4.5 has 17 rules and 67 symbols. The grammar in Figure 4.7, which explains the same sequence, has only 13 rules and 47 symbols. The length of rule *S* reduces from 16 in Figure 4.5 to 9 in Figure 4.7. This indicates that the second set of rules capture the structure of the original L-system much better than the first. For more derivation steps, the difference is more marked: at four derivation steps, the number of symbols drops from 88 to 59 and the number of rules from 31 to 16.

The additional computational requirements to enforce the bracket nesting constraint are minimal if the number of surplus save instructions is stored along with each rule. Each time a new rule is formed, the surplus can be computed from the surpluses of each symbol in the digram, making it independent of the length of the rule contents. In Section 7.3, this approach will be used to accelerate the rendering of figures based on L-systems.

It is interesting to consider applications of this technique in other domains. For example, for English text, SEQUITUR may benefit from constraining the rules to respect the white-space boundaries that delimit words. The modification can be made in exactly the same way, by adding a condition to the digram retrieval routine. This possibility is examined in Section 7.1.2.

4.4 Retrospective reparsing

Where domain-specific knowledge is unavailable, for example when L-systems contain no brackets, it is difficult to know when the formation of rules should be postponed. The solution described here involves making greedy decisions about parsing, but reviewing an initial parsing once it becomes clear that a better decision could have been made. We first describe, in Section 4.4.1, a simple way of adjusting a parse, but observe that it does not always improve the grammar. Section 4.4.2 discusses a criterion for deciding when to apply reparsing, which is intuitively compelling but problematic. Section 4.4.3 describes a new criterion that is somewhat surprising, but effective. Section 4.4.4 evaluates this solution on a range of L-systems.

4.4.1 The reparsing technique

Returning to the problematic sequence introduced in Figure 3.1d, the grammar at the bottom is the one produced by SEQUITUR. The grammar at the top represents the same sequence, and obeys the constraints, but contains one less symbol. According to Occam's razor, the topmost grammar should be preferred, as it offers a more compact explanation of the sequence. In the bottom grammar, the fact that both *b*s are preceded by an *aa* is obscured by rule A. Removing rule A and extending a match to the left of both *b*s would arrive at the grammar at the top. The crucial part of this operation is recognising when such a transformation is possible.

The key is noticing that every symbol that precedes *b*, in this case both *a* and A, have the suffix *a*. This means that a rule for *ab* can be formed by borrowing an *a*

output from SEQUITUR	S → AbAab
	A → aa
expand rule A	S → aabAab
	A → aa
form rule for ab	S → aBAB
	A → aa
	B → ab
remove useless rule A	S → aBaaB
	B → ab
extend rule B to the left	S → BaB
	B → aab

Figure 4.8 Transforming a good grammar to a better one

from the *A* that precedes *b*. Figure 4.8 demonstrates this process. First, the initial occurrence of rule *A* is expanded to reveal its contents. Next, the rule *B* → *ab* can be formed. Now rule *A* occurs only once, and can be removed. This exposes another *a* to the left of both *B*s, which is incorporated into rule *B*. The resulting grammar is the same as the grammar at the top of Figure 3.1d (with relabelling).

This process can also operate in the opposite direction. Consider the sequence in Figure 4.9a. A better grammar can be obtained by extending rule *B* to the right, taking advantage of the common prefix *a* of all its successors. This results in the disappearance of rule *A* and the shortening of rule *S*.

This technique of extending a rule to the left or the right by borrowing from neighbouring symbols successfully produces a smaller grammar in these two examples. However, it can also make the grammar worse. Figure 4.9b shows a sequence for which SEQUITUR produces the grammar in the middle. Extending *b* to the right to form *B* → *ba* increases the length of rule *S*, because it disturbs the hierarchy formed by the eight consecutive *a*s. It is therefore important to apply this technique judiciously. The next two subsections discuss ways of deciding whether to perform reparsing.

4.4.2 Reparsing based on local optimality

The first approach appeals to Occam's razor and performs the reparsing only if doing so would reduce the size of the grammar. That is, an action is only performed if it results in a grammar with fewer symbols or rules. The expectation is that over a

	sequence	initial grammar	reparsed grammar
a	baaaabaaa	S → BABa A → aa B → bA	S → BaB B → baaa
b	baaaaaaaaba	S → bBBba A → aa B → AA	S → CaAAAC A → aa C → ba
c	abcabbc	S → AcAbc A → ab	S → aBabB B → bc

Figure 4.9 Effects of reparsing
(a) A better grammar by extending *B* to the right
(b) Extending *b* to the right makes the grammar worse
(c) *a* and *c* fight over a neighbouring *b*

large sequence, local improvements in the grammar will result in an overall improvement. The difficulty with this technique is knowing in advance whether the grammar will shrink, as the operation may have wide-ranging effects due to the cascading updating that occurs when the constraints are imposed. In fact, the only reliable way to tell what size a grammar will be after an action is to actually perform the action. If the grammar shrinks, no further action is necessary. If the grammar grows, the action must be undone. If the grammar remains the same size, it seems sensible to leave the grammar in the new state, rather than perform additional work to undo the action.

There are five problems with this approach:

- the difficulty of undoing,
- the problem of evaluation,
- efficient detection of opportunities for rule extension,
- infinite loops, and
- grammar quality.

First, undoing an action is difficult because the grammar may change significantly as a result. In principle, undoing actions incurs a constant overhead if each change to the grammar is recorded, and undoing is performed by reversing each action. However, it is important to record the changes at a low level, and efficient implementation is challenging. A brute-force solution is to copy the entire grammar, and revert to the copy in the event of an undo. However, the time required to perform this grows with the size of the grammar, and results in a quadratic time algorithm overall.

Second, it is not clear how the size of grammars should be measured. So far we have measured grammars by counting symbols and rules. This approach is valid on final grammars that differ markedly, but when comparing two grammars that differ only in one or two symbols, the exact metric is more crucial. If a grammar has fewer rules and fewer symbols, it is most likely to smaller according to any measure. However, if a grammar has one fewer rule but one more symbol, the decision is more difficult. Additionally, since the object is to find structure in rule *S*, any reduction in the length of rule *S* should carry greater weight than a reduction in other rules. It turns out that the quality of the final grammar is sensitive to the exact choice of grammar metric, and there seems to be no good way of justifying the choice theoretically.

The third difficulty is efficiently detecting when it is possible to extend a rule. The naive way is, after each new symbol appears and has been processed, to examine the predecessors and successors of each unique symbol in the grammar. If they all have a suffix or prefix in common, then perform the transformation. This naive algorithm takes quadratic time, as all symbols in the grammar are examined after each new symbol.

The fourth problem related to global scanning of the grammar is that the extension of two different symbols may conflict. It is possible for two neighbouring non-terminals to 'fight' over a symbol at their border. For example, in Figure 4.9c, SEQUITUR produces the grammar in the centre. A new rule for *bc* can be formed by borrowing the preceding *b* from rule *A*. This new rule is formed at the expense of rule *A*, which disappears. The size of the grammar has not changed, so the grammar is left in this state. In the new grammar, a new rule can be formed by extending *a* to the right. Now the grammar has returned to its initial state, and the process continues, looping forever. Checks can be implemented to prevent this happening, but these are often difficult, because more than two operations may be necessary to perform a complete loop in a larger grammar. Furthermore, the check seems arbitrary and inelegant. This problem can also be avoided if operations that leave the grammar the same size are undone, but this involves considerable extra work since this situation occurs frequently.

The final problem is the quality of the grammars that are produced. Figure 4.10a presents an L-system that draws the Koch curve in Figure 4.10e. Biological forms generally require brackets because several parts may sprout from a single point which must be saved on the state stack. The fractal here, however, is more artificial, and brackets are unnecessary. Without brackets, SEQUITUR-K is equivalent to SEQUITUR, and both produce the grammar in Figure 4.10d. Figure 4.10b shows the non-recursive version of the L-system with a derivation length of three. This is the grammar that we are attempting to reproduce.

Using a size metric that works well in practice, and using the brute force method for undoing, along with a complex system for detecting looping in rule extension, the grammar in Figure 4.10c is formed. Although the grammar exhibits a similar structure to the ideal non-recursive version, rules *B* and *F* are slightly different from rule *J*. This prevents the procedure described in Section 5.1 from inferring the recursive version of the L-system.

4.4.3 *Oblivious reparsing*

The solution to these problems is simple, but somewhat non-intuitive. An explanation of the solution will be given first, followed by a discussion of the reason for its efficacy. In the new scheme, extensions are only made if they are near the end of rule *S*, and transformations are performed regardless of their effect on the grammar. After each new symbol is observed and the constraints have been enforced, the last symbol in rule *S* has an opportunity to extend to the left, and the second last symbol has an opportunity to extend to the right. The same process is

a
```
S → f-f-f-f
f → f+f-f-ff+f+f-f
```

b
```
S → AAAB
A → B-
B → DCCDEC
C → -F
D → FE
E → +F
F → HGGHIG
G → -J
H → JI
I → +J
J → LKKLMK
K → -f
L → fM
M → +f
```

c
```
S → AAAB
A → B-
B → CDEC
C → EFD
D → F+
E → -F
F → GHIG
G → IJH
H → -J
I → J+
J → LKKLMK
K → -f
L → fM
M → +f
```

d
```
S → JIDAAAJC
A → JBB
B → CE
C → FD
D → MGE
E → HFNG
F → +T
G → HL
H → If
I → QP
J → K+K
K → RONL
L → RM
M → SN
N → Of
O → WP
P → +WQV
Q → -f
R → TS
S → VU
T → XU
U → WV
V → ZX
W → Yf
X → ZY
Y → f-
Z → f+
```

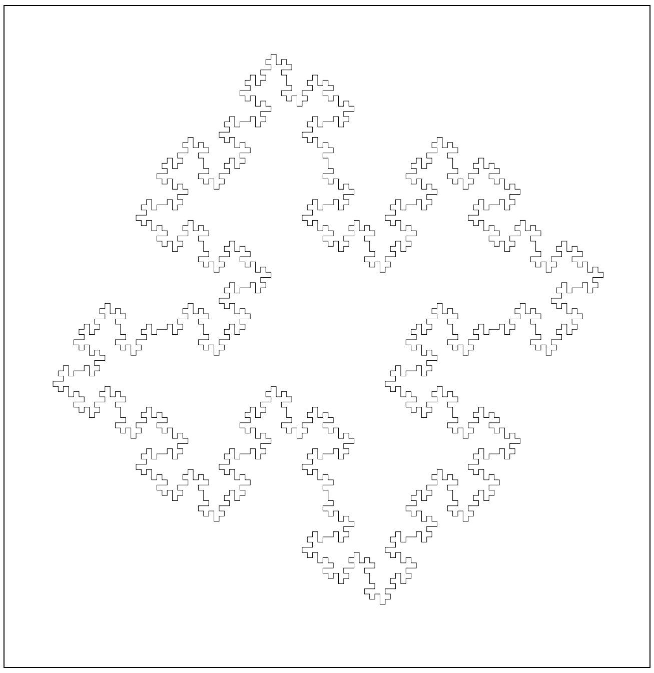
e


Figure 4.10 An L-system for drawing a Koch curve
(a) the original L-system
(b) the equivalent non-recursive L-system for three derivation steps
(c) the grammar induced by an Occam's razor version of reparsing
(d) the grammar produced by SEQUITUR
(e) the graphical interpretation of the sequence

```
S → ...AB
A → ...
B → ...CD
C → ...
D → ab
```

Figure 4.11 Hierarchical parsing adjustment

then applied to the symbols at the end of the rule headed by the last symbol in rule *S*, and continues in this way through the hierarchy. For example, for the grammar in Figure 4.11, SEQUITUR-R examines the predecessors of *B* for a common suffix, and if one exists, extends rule *B* to the left. If this is unsuccessful, it examines the prefixes of *A*'s successors, and attempts to extend rule *A* to the right. Next, it examines the contents of rule *B*, and attempts to extend *D* to the left and C to the right. Finally, the contents of rule *D* are given the same opportunity: *b* to the left and *a* to the right.

These operations are performed regardless of their effect on the grammar's quality, so this technique is called *oblivious reparsing*. It nevertheless works well on a wide range of sequences. So how does it succeed in improving SEQUITUR's operation? The process is effective by virtue of two properties: the cancelling effect of extension in one direction with respect to the other, and limited opportunity for rules to be extended.

Consider the first property. Extending a rule in one direction serves to undo extending the neighbouring rule in the other direction. Not only does one undo the other locally, but by virtue of the constraints, it undoes the effect on the entire grammar. The effect is demonstrated in Figure 4.9c, where two consecutive extensions result in a return to the original grammar. In larger grammars, an infelicitous extension will affect all of the neighbours of the extended symbol. To completely undo the effects of the extension, each of those neighbouring symbols must appear again later in the sequence, so that they have an opportunity to extend in the other direction and recapture the symbol that was taken from them.

Now consider the second property. Each non-terminal gets one chance to extend leftward when it is at the end of rule *S*, and once chance to extend rightward when it is the second-last symbol. The problem of rules fighting over a symbol is solved by the 'one chance per symbol' rule, as opposed to the previous scheme where a symbol anywhere in the grammar had an opportunity for extension after each new symbol.

f[+f]f[–•f]f[+	f[+f]f[–•f]•f[•+	f[+f]f[–f]•f[+	f[+f]f[–f]•f[+
S → CE...	S → CDA+...	S → HI...	S → HI...
B → DA	B → DA	B → DA	B → DA
C → IF	C → IF	C → IF	
E → B+	E → B+	E → B+	E → B+
H → CD	H → CD	H → CD	H → IFD
I → A+	I → A+	I → A+	I → A+
...	...	...	...

Figure 4.12 How retrospective reparsing corrects initial mistakes

It also makes reparsing more efficient: reparsing is performed at most once for each symbol in the grammar, whereas before it could be invoked many times.

4.4.4 Evaluation

When this oblivious reparsing is applied to the sequence produced by the L-system in Figure 4.10, the exact non-recursive version in Figure 4.10b is reproduced. Section 4.4 showed that it is possible to use domain knowledge in a principled way to improve induction of grammars from a known source. Now we demonstrate that retrospective reparsing can achieve even better results than SEQUITUR-K, even when brackets are present. Whereas the approach based on domain knowledge attempts to form correct rules in the first place, reparsing initially forms bad rules, but corrects them when evidence appears that the rules should have been different.

For example, Figure 4.12 reproduces part of the grammar in Figure 4.5, which is the result of running the greedy version of SEQUITUR over the turtle command sequence in Figure 4.3a. It was observed that the initial sequence *f*[+*f*]*f*[–*f*]*f*, which corresponds to one rule in the L-system, is split over two rules: C and *E*. Retrospective reparsing examines the successors to C, which are *E* in rule *S* and *D* in rule *H*. Rule *E* begins with *B*, which in turn begins with *D*; so both occurrences of rule C are followed by symbols that share a common prefix, *D*. Expanding rules *E* and *B* results in the grammar in the second column. Next, the duplicate digram *CD* matches rule *H* and the duplicate *A*+ is replaced by *I*, as shown in the third column. Rule C is no longer used more than once, and is deleted in the fourth column. The new rule *S* begins with rule *H*, which expands to *f*[+*f*]*f*[–*f*]. This is closer than the original rule C was to the sequence *f*[+*f*]*f*[–*f*]*f* that would be expected as the first rule. Of course, reparsing is performed incrementally, rather than after the entire grammar has been formed, but this description captures the essence of the process of revising rule boundaries as more of the sequence is observed.

Using reparsing on the sequence in Figure 4.3a produces the non-recursive version in Figure 4.3e. This grammar is even smaller than the one produced using domain knowledge: 30 symbols and 6 rules versus 47 symbols and 13 rules.

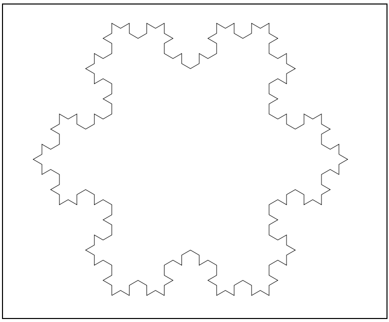
Figure 4.13 shows two more L-systems. In Figure 4.13a, the grammar inferred by SEQUITUR lacks a regular structure, whereas SEQUITUR-R captures the *f–f* structure

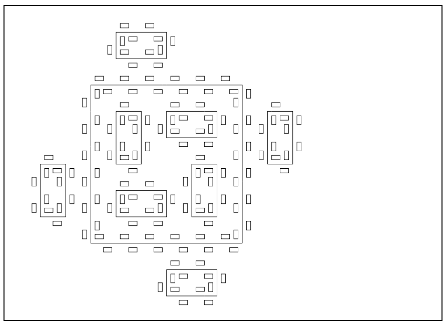
a

```
S → f++f++f
f → f-f++f-f
```

b

S → f-f-f-f
f → f-$\underline{f}$+ff-f-ff-f$\underline{f}$-ff$\underline{f}$-ff+f+ff+f$\underline{f}$+fff
$\underline{f}$ → $\underline{ffffff}$





```
S → HEAABC
A → BB
B → CD
C → DFFD
D → GE
E → GF
F → IIG
G → ++H
H → f1
I → -f
```

S → PWMJKLJLXAAADI
A → DCCE-BB
B → CI
C → I-
D → FFHGIE
E → MXI
F → N-G
...
Y → f+
Z → $\underline{f}$-

```
S → AABC
A → BB
B → CH
C → D-D
D → EHE
E → F-F
F → GHG
G → f-f
H → ++
```

S → AAAB
A → B-
B → I-DGFGCCIEHDI
C → PF+
D → PE
E → HI
F → fI
G → -I
H → +I
I → f-KNMNJJfLOKf
J → $\underline{f}$M+
K → $\underline{f}$L
L → Of
M → Nf
N → -f
O → +f
P → QQQ
Q → $\underline{ff}$

Figure 4.13 Identification of non-recursive L-systems
From top to bottom: original L-system, graphical interpretation, grammars inferred by SEQUITUR and SEQUITUR-R
(a) snowflake (von Koch, 1905),
(b) combination of islands and lakes (Mandelbrot, 1982),

in rules C, *E* and G and the larger structure joined by ++ in rules *F*, *D*, and the combination of rules *S*, *A*, and *B*. The L-system in Figure 4.13b has a much more complex rule, and also makes use of non-drawing *f* characters to form 'lakes' in the fractal. SEQUITUR forms a grammar that has 26 rules (not all are shown in the Figure), and little structure. The grammar produced by SEQUITUR-R displays two distinct rules, *B* and *I*, which correspond to the two derivation steps used to produce the figure. Rules C through *H* and *J* through O compress the redundancy in the original rule headed by *f*, while rules *P* and *Q* compress the rule headed by *f* in the original.

4.5 Summary

This chapter has highlighted a deficiency of the basic SEQUITUR algorithm described in Chapter 3: it fails to detect the source grammar of certain highly-structured sequences. As argued in Section 2.5, no guarantees can be made about source identification, but we have nevertheless presented two techniques that improve the hierarchy inferred from L-system output. The first relies on knowledge about the domain, and places additional constraints on SEQUITUR's grammars. The second uses hindsight to correct initial parsing, which works extremely well on a range of L-systems. Both these techniques will be built upon in subsequent chapters: the domain-knowledge constraints are applied to graphical rendering in Section 7.2, and a constraint for natural language is used in Section 7.1. Section 5.1 describes how the grammars produced by reparsing can be transformed into the original recursive, non-deterministic L-system.

5. Generalising the grammar

The grammars produced by SEQUITUR can only generate one sequence. The real power of a grammar, however, is its ability to produce a whole language of sentences. This chapter examines ways of generalising the grammars produced by Sequitur to make them more expressive.

The thesis proposes two kinds of structure. The first, exact repetition, has been discussed in Chapters 3 and 4. Here we address the second kind of structure: branching and looping. Inferring this kind of structure allows models to be produced that can generate a variety of sequences, based on the contents of a single sequence. This transformation from a single sequence to many sequences is a process of generalisation. This chapter discusses four kinds of generalisation:

- Generalising a grammar produced by SEQUITUR to form a recursive L-system,
- Generalising a trace of a program executing on one input to a program that can process other inputs,
- Generalising SEQUITUR's rule *S* by recognising branches and loops, producing non-deterministic rules for branching and recursive rules for looping,
- Generalising the sequence of tokens in a textual database to separate the fixed structure of the database from the variable content.

Four case studies are presented to demonstrate these generalisations. The first concerns inferring the original recursive L-system from the non-recursive version found by reparsing in Chapter 4. This generalisation enables the grammar to generate sequences with different numbers of derivation steps, based on the crucial observation that the same rule, at different derivation levels, has the same form. In fact, different instances of a rule *unify*. A Prolog program will be used to infer the original L-system, with unification providing the generalisation technique.

The second study is the inference of a computer program from a trace of its execution, which involves recognition of loops, branches and procedure calls. The technique presented is an extension of the *ingenuous model* described by Witten (1979), and performs remarkably well on an input sequence expressed in an appropriate way. The effects of procedure calls and recursion are studied, and a technique for inferring them is presented. An essential element of this analysis is a method of extracting an acyclic version of the automaton.

The third case study involves the sequential structure produced by grammars for programming languages. This section combines SEQUITUR's ability to recognise repetitions with the automaton approach to recognising branching and looping structure. The rationale for combining these approaches is based on the observation that when SEQUITUR is invoked on a sequence such as the text of a computer program, rule *S* in the final grammar is often very long, and accounts for about two thirds of the symbols of the entire grammar. One way to interpret this is to say that the rules apart from rule *S* capture the repetitive structure of the sequence, and *S* contains the unstructured residue—all the digrams that only appear once in the input sequence. To reduce the size of the grammar further, that is, to recognise more structure in the sequence, the obvious point of attack is the size of rule *S*. The kind of structure in rule *S*, if any, is of a more subtle nature than exact repetition. This non-deterministic structure is exactly where the automaton modelling technique excels.

It will be shown, however, that detection of non-deterministic structure is more difficult than detection of exact repetition, because the space of possibilities is much larger, and the potential for recognition of spurious patterns is greater. Even the application of the minimum description length principle is unhelpful if the examples are small. The fourth case study involves a much larger sequence: a 9 Mb textual database. The size of the database is helpful in justifying certain generalisations. Once these generalisations are made, SEQUITUR can compress the database much more successfully than any other general-purpose or specialised scheme.

5.1. Inferring recursion from self-similarity

Section 4.4 described how reparsing a sequence allows SEQUITUR to infer a non-recursive version of a context-free L-system from its output. This section shows how the non-recursive grammar can be generalised to the original recursive L-system. The key to this process is that successive applications of an L-system rule appear as separate rules in the non-recursive version, but have the same form. That is, self-similarity in the fractal figure can be detected in the inferred rules The first subsection describes how pattern matching can be used to identify rules in the non-recursive grammar that stem from the same recursive rule in the L-system. The second subsection presents a Prolog program that performs unification to recreate the recursive grammar.

5.1.1 Unification as generalisation

Recall from Chapter 4 that an L-system, evaluated to a particular number of derivation steps, can be re-expressed as a non-recursive grammar. Figure 4.3d shows the non-recursive L-system equivalent to Figure 4.3a. The non-terminals *A* in Figure 4.3d represent the *f* symbols produced by the first application of the rewriting rule to the start string of *f*. The non-terminals *B* represent the *f*s in the second application of the rewriting rule to the string. To recreate the original L-system, it is necessary to recognise the equivalence of rules *S*, *A*, and *B*. This recognition is based on pattern matching. Each of the rules has the same form: if a terminal symbol (such as *f*, [,], + or –) appears in one of the rules, the same symbol, or a non-terminal, appears in the same position in all the other rules. If a non-terminal appears in two places in one rule, the symbols at the corresponding positions in a different rule will be the same. If non-terminals are read as variables, and terminals are read as literals, this pattern matching corresponds to unification, so a language such as Prolog can be used to identify corresponding rules.

The process of identifying the recursive L-system from Figure 4.3d after three derivation steps proceeds as follows. The first two rules, *S* and *A*, unify: all of the terminals occur in the same places, and *A* occurs in the same places within *S* as *B* does in *A*. Unifying these two rules first binds the two heads together, unifying *S* with *A*, then unifies *A* and *B*, because they occur in the same position in both rules. The bindings so far are $S \equiv A \equiv B$. The equivalence of *A* and *B* has further ramifications. Because each of *A* and *B* head rules, the contents of those rules are unified, producing $B \equiv f$. There are no further effects of these bindings, so the final bindings are $S \equiv A \equiv B \equiv f$. That is, the grammar can be re-expressed in terms of *f* alone, by replacing all occurrences of *A*, *B*, and *S* by *f*. Now all rules are identical: $f \rightarrow f[+f]f[-f]f$. Deleting all but one of them and adding the fact that the start symbol has become *f* gives the original grammar in Figure 4.3a.

Figure 5.1 shows the same process on the grammar that SEQUITUR-R produces from the sequence in 4.3b after three derivation steps. The non-recursive grammar (the same as that in Figure 4.3e) is shown in the first column. Unifying rules *S* and *B* produces three bindings: $S \equiv B$, $B \equiv D$ and $A \equiv C$, as shown in the second column. There

original	unify rules S and B	match bodies of two rule S	final	expand A,F and G
	S≡B B≡D A≡C	S≡f A≡E		
S → BFAGA A → B]B B → DFCGC C → D]D D → fFEGE E → f]f F → [+ G → [–	S → SFAGA A → S]S S → SFAGA A → S]S S → fFEGE E → f]f F → [+ G → [–	f → fFAGA A → f]f f → fFAGA A → f]+ F → [+ G → [–	f → fFAGA A → f]f F → [+ G → [–	f → f[+f]f[–f]f

Figure 5.1 Unifying a rule to produce a recursive grammar

are now two pairs of identical rules, so one of each is deleted. Next, there are two rules headed by *S*. Unifying their bodies gives new bindings *S≡f* and *A≡E*, shown in the third column. Deleting duplicate rules produces a grammar where all heads and bodies of rules are unique, and the process stops. The final grammar is shown in the fourth column. Expanding *A*, *F* and G produces the original L-system *f* → *f*[+*f*]*f*[–*f*]*f*.

Note that bindings are made in two ways. The first binding, of rules *S* and *B*, is determined by unification of rule contents, and subsequent bindings are motivated by equality of the head of two rules, such as the two rules headed by *S*. In more complex grammars, the bodies of two rules may be identical, in which case the heads should be unified. Once the first binding is made, the rest follow as a consequence. Not all initial bindings are equally effective. Figure 5.2 shows the result of binding rules *S* and *D*. This binding appears no less promising than *S* and *B* at the outset, but the consequences are much less wide-ranging. The result is a recursive grammar, but one with two more rules than the correct version in Figure 5.1. Furthermore, this grammar is not capable of producing the sequence produced by the original grammar after one derivation step. The solution, as

original	unify rules S and D	final
	S≡D B≡f A≡E	
S → BFAGA A → B]B B → DFCGC C → D]D D → fFEGE E → f]f F → [– G → [+	S → fFAGA A → f]f f → SFCGC C → S]S S → fFAGA A → f]f F → [– G → [+	S → fFAGA A → f]f f → SFCGC C → S]S F → [– G → [+

Figure 5.2 The results of a different initial unification

a original	b unify G and K	c unify B and J	d unify D and K
S → AAAB	S → AAAf	S → AAAB	S → AAAB
A → B-	A → f-	A → B-	A → B-
B → DCCDEC		B → LCCLfC	B → DCCDfC
C → -F			C → --
D → FE			D → -f
E → +F			f → +-
F → HGGHIG		f → HGGHIG	+ → HGGHIG
G → -J		G → -B	G → -J
H → JI		H → BI	H → JI
I → +J		I → +B	I → +J
J → LKKLMK	f → LKKLMK		J → LDDLMD
K → -f	K → -f	K → -f	
L → fM	L → fM	L → fM	L → fD
M → +f	M → +f	M → +f	M → +f

Figure 5.3 Consequences of initial unifications on a more complex grammar
(a) the non-recursive Koch grammar
(b) an effective unification
(c) a less effective but correct unification
(d) an incorrect unification

explained below, is to try all possible unifications, and pick the smallest grammar that results.

To demonstrate two further problems, Figure 5.3 shows three unifications of the inferred non-recursive form of the Koch L-system from Figure 4.10. Figure 5.3a gives the original L-system, and Figure 5.3b shows the result of a successful unification. Figure 5.3c provides an example of a sub-optimal but correct grammar. Initial bindings can produce consequences that conflict with each other. In Figure 5.3a, rules *D* → *FE* and G → –*J* unify, even though their form does not suggest that they are necessarily related. This unification implies *E* ≡ *J*. However, the body of rule *E* is two symbols long, whereas the body of rule *J* is six symbols long. Their bodies therefore prevent them from unifying, contradicting the initial unification. This is an important constraint in cases where many there are many possible bindings between rules in the grammar.

The worst possible case is where an initial binding does not produce incompatible consequences, but gives rise to an grammar that is incapable of reproducing the original sequence. Figure 5.3d shows the result of unifying *D* and *K*. None of the consequences are incompatible, but the unification produces rules such as C → – – and *f* → +–, although neither – – nor +– appear in the original sequence.

5.1.2 Implementation of the generalisation

To find the correct recursive grammar, a system should try all possible initial bindings, detect incompatible consequences, and choose the smallest final grammar to avoid the larger incorrect ones. This subsections presents such a system, and it infers the correct L-system for all of the L-systems discussed in Chapter 4, as shown in Figures 4.3, 4.4 and 4.13. The program was written in Prolog because of its in-built unification and backtracking capabilities. The code is shown in Figure 5.4. The input is the non-recursive grammar, represented as in Figure 5.5. A grammar is a list of rules, a rule is a two-element list of the head and the body, and the body is also a list. Non-terminals are represented as variables, and terminals as literals. The predicate *match-rules* is called with G bound to the non-recursive grammar, and *Unify* set to *yes*. This means that at the top level, two rules will be unified, and at every recursive call after that, the consequences of the unification will simply be propagated.

The two calls to *member* retrieve successive rules from the grammar. *N1* and *N2* contain the indexes of the rules in the grammar. So that a rule is never compared to itself, *N1* is required not to be equal to *N2*, and to ensure that rules are only compared to each other once, *N1* must be less than *N2*. Next, if *Unify* is *yes*, the

```
match_rules(G, G2, Unify) :-
   member([H1|T1], G, N1),
   member([H2|T2], G, N2),              pick two rules
   N1 < N2,                             if they're different
   (
      Unify=yes,                        if this is a top-level call
          [H1|T1]=[H2|T2];              unify the rules
      T1==T2, !, H1=H2;                 otherwise, if the heads are identical unify the bodies
      H1==H2, !, T1=T2                  otherwise, if the bodies are identical unify the heads
   ),
   delete([H1|T1], G, G1),              remove one of the identical rules
   match_rules(G1, G2, no).             and keep going

match_rules(G, G, _).                   stop when there are no identical heads or bodies

member(X, [X|_], 1).                    pick a rule from the grammar
member(X, [_|T], N) :-
   member(X, T, N1), N is N1 + 1.

delete(X, [Y|Tail], Tail) :-
   X == Y,                              careful not to unify X and Y
   !.                                   we don't care which matching rule is deleted

delete(X, [Y|Tail], [Y|Tail1]) :- delete(X, Tail, Tail1).
```

Figure 5.4 Prolog code for transforming a non-recursive grammar to a recursive one

two rules are unified. If this fails, the Prolog interpreter backtracks to find a new pair of rules. If the unification succeeds, one of the rules is deleted, because they are now identical, and *match-rules* is called recursively with *Unify* set to *no*.

In each recursive call, two rules are chosen as before, and their bodies are examined for equality. The == operator is used to test for exact equality rather than unification. If they match, the heads are unified, one of the rules is deleted, and *match-rules* is called again. If the bodies match but the heads do not unify, this indicates that some consequences are incompatible. In this case, the program should not backtrack to find another matching pair: any incompatibility should nullify the initial unification. For this reason, a cut is inserted just after this test, so that as soon as the incompatibility is found, the goal fails. If the bodies are not equal, the heads are tested for equality. If they are equal, the bodies are unified. Again, backtracking is prevented if == succeeds but the unification does not.

One final point should be made about handling the grammar, which contains unbound variables. The *delete* predicate is intended to delete a rule from the grammar that exactly matches the one passed in. The straightforward stopping condition for delete is *delete*(*X*, [*X* | *Tail*], *Tail*). Unfortunately, this attempts to *unify* the rule passed in with the rule at the head of the list, the two *X* variables. If this succeeds, it deletes a rule that unifies with the rule passed in, rather than one that exactly matches it. To correct this, different variables, *X* and *Y*, are used for the *X*s,

```
?- match-rules([[S, [B,G,A,I,A]],
               [A, [B,’]’,B]],
               [B, [D,G,C,I,C]],
               [C, [D,’]’,D]],
               [D, [F,G,E,I,E]],
               [E, [F,’]’,F]],
               [F, [f,G,H,I,H]],
               [G, [‘[‘,-]],
               [H, [f,’]’,f]],
               [I, [‘[‘,+]]],
               G).

G = [[f,[f,G,H,I,H]],[G,[‘[‘,-]],[H,[f,’]’,f]],[I,[‘[‘,+]]]
G = [[f,[F,G,E,I,E]],[E,[F,’]’,F]],[F,[f,G,H,I,H]],[G,[‘[‘,-]],[H,[f,’]’,f]],[I,[‘[‘,+]]]
G = ...
```

Figure 5.5 Generating recursive grammars

and literal equality is used instead of unification. Furthermore, there are always at least two different rules that can be deleted from the grammar, since the deletion is to get rid of duplicates. It is not important which one is deleted, so to prevent identical alternate grammars produced by backtracking in the *delete* predicate, a cut is inserted.

Match-rules returns successive solutions; picking the one with fewest rules gives the correct recursive grammar. Figure 5.5 shows the first two solutions produced by the program for the bracketed L-system discussed earlier, the first of which is the correct grammar.

This process is inefficient, because many combinations of rules are checked for matching heads or bodies, even if they have not changed since the last check. Indeed, because the space of matching rules is searched in this way, the process is exponential in the number of rules. At the top level, there are $O(r^2)$ pairs of rules, where r is the number of rules. Unifying a pair reduces the number of rules by one, so there are $O((r-1)^2)$ pairs to compare. Proceeding in this way gives a total time complexity of $O(r!^2)$. Assuming that the sequence is derived from a recursive L-system, the size of the non-recursive grammar produced by SEQUITUR is proportional to the logarithm of the size of the sequence, i.e. $r \propto \log n$, where n is the size of the sequence. The complexity of the inference procedure is therefore $O((\log n)!^2)$, which, applying Stirling's approximation for factorials, approximates $O(n)$.

The Prolog implementation is useful for illustrative purposes. However, a more efficient procedure would efficiently build and maintain a table of matching heads and tails, so that after the quadratic process of finding unifying pairs at the top level, following the consequences of the unification would be linear in the number of rules. This would lead to an overall complexity of $O(r^2)$, or $O(\log^2 n)$.

5.2 Reconstructing a program from an execution trace

Procedural programming languages such as C provide flow of control constructs including branches, loops, and procedure calls. This section considers how a program might be reconstructed from a trace of its execution—that is, how flow of control statements can be inferred from a sequence that they create. These branching and looping structures occur in a wide variety of sequences, some of which will be seen in Section 5.3, and this situation represents a well-understood, controllable environment in which to construct inference techniques. In this section, SEQUITUR's ability to form grammars will be temporarily ignored in order to concentrate on this non-deterministic structure. In Section 5.3, however, the grammatical techniques and the automaton techniques described here will be combined.

Gaines (1976) proposed a straightforward algorithm for inferring a program from a trace and demonstrated it on a trace of bubblesort. The idea is to construct a finite-state automaton to represent the program, where each state corresponds to a unique instruction in the trace, and transitions indicate adjacent instructions. That is, for each instruction in the trace, a transition is created between the current node and the node representing the newly observed instruction. If the instruction is novel, a new node is created for it.

For example, the bubblesort program in Figure 5.6a executes the sequence of instructions partially shown in Figure 5.6b. This sequence consists of one C expression per line. Figure 5.6c is the finite-state automaton derived from the trace. The resulting automaton is non-deterministic, because it is not clear which branch to take based on the conditional statements. To transform this automaton into an executable program, the branches following conditions such as *if* (i >= 0), the shaded node in Figure 5.6c, must be labelled true or false. This can be performed by extending Gaines' technique to run the original data through the automaton and follow the trace. For example, the first time that $i >= 0$ is evaluated, N is equal to 10, and the comparison is 10 >= 0. This is true, so the next statement, $j = 1$, must represent the true branch of the condition. Similarly the appropriate branches for true and false results of the other conditions can be reconstructed. The resulting finite state automaton, shown in Figure 5.6d, represents a correct bubblesort program, and can be translated back to the executable program in Figure 5.6a.

The success of this technique is due to two simplifications. First, the instructions do not include the specific data that they operate on—rather, they are stated in terms of the variables in the code. Second, the program does not make any calls to procedures. The first simplification is difficult to remove. If the sequence consisted of lines such as *if* (10 > 0) instead of *if* (i > 0), it would be necessary to find the correct substitution of variables, and expressions including variables, before

a
```
int a[] = {2,34,23,6,7,43,6,1,6,4};


main() {
  for (i = 10; i >= 0; i --)
    for (j = 1; j < i; j ++)
      if (a[j - 1] > a[j]) {
        t = a[j - 1];
        a[j - 1] = a[j];
        a[j] = t;
      }
}
```

b
```
i = N
if (i >= 0)
j = 1
if (j < i)
if (a[j - 1] > a[j])
j ++
if (j < i)
if (a[j - 1] > a[j])
t = a[j - 1];
a[j - 1] = a[j];
a[j] = t;
...
```

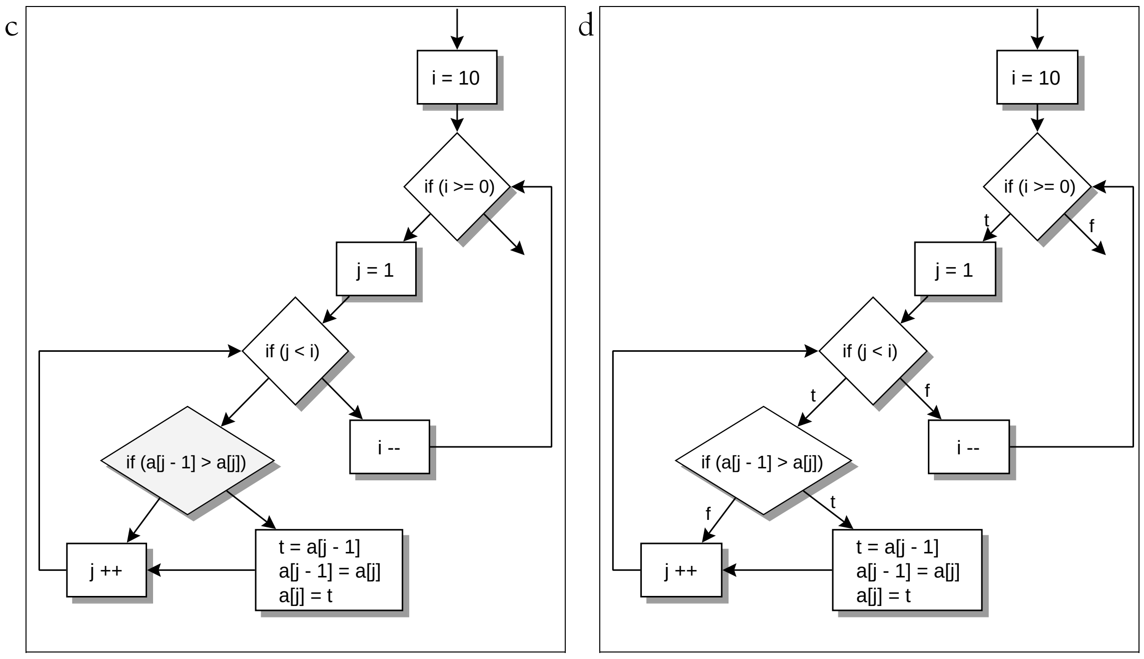


Figure 5.6 Inferring a program from an execution trace (after Gaines, 1976)
(a) bubblesort program
(b) part of the trace produced by (a)
(c) finite state automaton produced from the bubblesort trace
(d) labels added to the conditional branches in (c)

attempting to infer an automaton. This is beyond the scope of this thesis, which is concerned with individually meaningless symbols.

The second problem, which occurs when programs include procedure calls, is more relevant. Figure 5.7a shows a bubblesort program where the array is printed before and after one of the assignments. Executing it and forming an automaton from the trace results in the automaton in Figure 5.7b. In the automaton, the starred state, containing the instruction *putchar*(‘*n*’), has branches to two different instructions with no accompanying condition. With no basis for deciding which branch to take, the program cannot be executed: the automaton is non-deterministic. The next subsection describes how this situation can be remedied to produce a deterministic automaton.

5.2.1 Identifying procedures

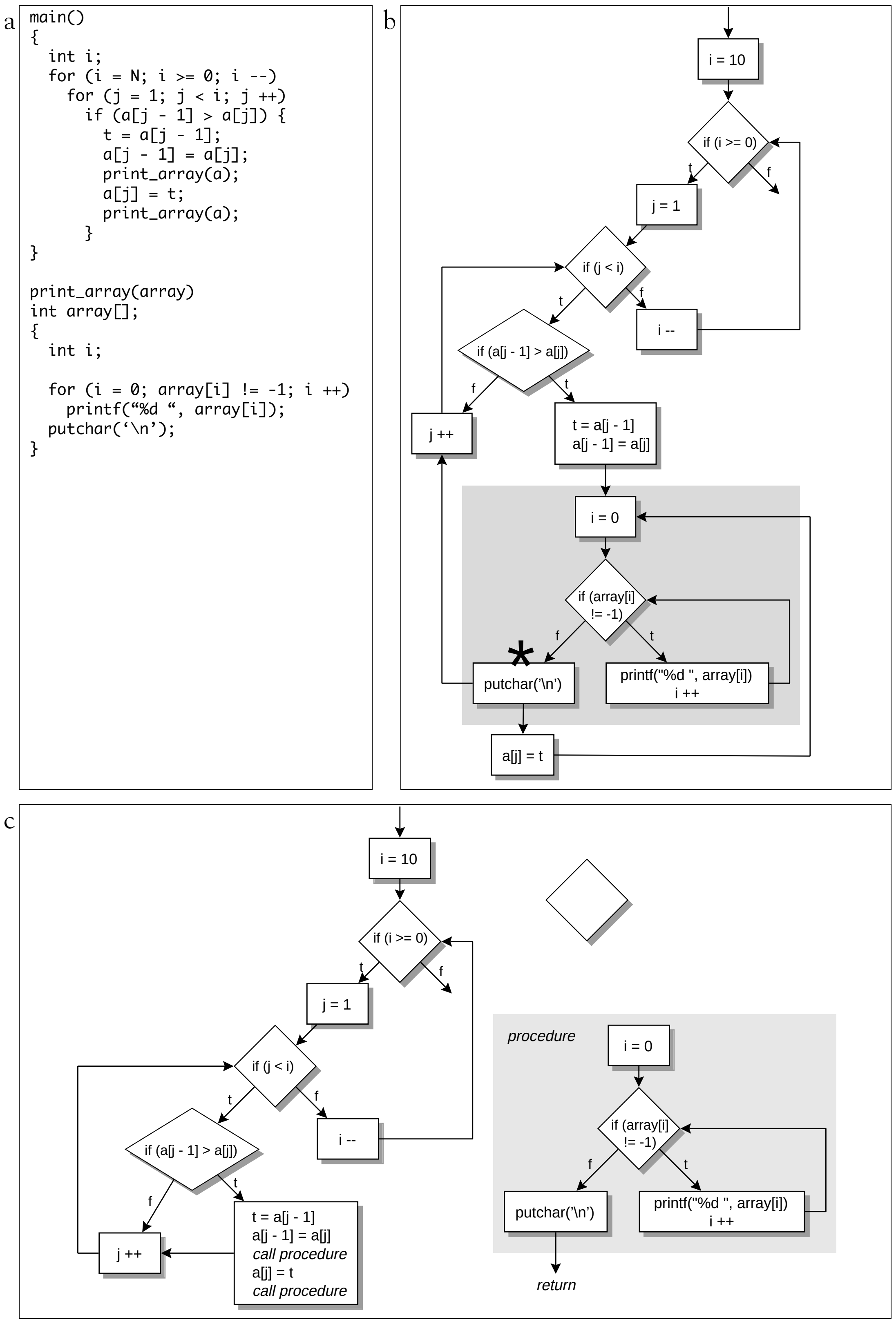
The two transitions from the starred state in Figure 5.7b correspond to the return from the *print_array* procedure. The next state is not determined by information contained in the trace, but rather by the return address pushed on the program stack at the start of the procedure. If the start of the procedure can be identified, then each transition from the return state can be associated with the corresponding transition into the start state. That is, the procedure call associated with the procedure return can be identified. Using this information, and separating the states in the procedure from the rest of the graph, the deterministic automaton can be reconstructed. The new automaton will be a push-down automaton, because a stack is necessary to remember the return states.

The automaton can be made deterministic in the following way:

For each state with non-deterministic transitions leading from it:

1. Replace the transitions with a procedure return,
2. Find the start of the procedure,
3. Remove the subgraph beginning at the start of the procedure and ending at the procedure return and label it a procedure,
4. Pair the states before the start of the procedure with the states following the return by examining the original trace,
5. Connect each pair by inserting a procedure call between them.

When all procedures have been identified, no non-deterministic states will remain, and the set of graphs will form an executable program. Figure 5.7c shows the final form of the automaton derived from the trace of the program in Figure 5.7a. The procedure is shaded in Figure 5.7b, and has been separated in Figure 5.7c. The non-deterministic branches have been replaced by the word *return*, and the two branches into the procedure have been replaced by *call procedure*. This subsection examines the problem of identifying the procedure (step 2) in more detail.



```
main()
{
  int i;
  for (i = N; i >= 0; i --)
    for (j = 1; j < i; j ++)
      if (a[j - 1] > a[j]) {
        t = a[j - 1];
        a[j - 1] = a[j];
        print_array(a);
        a[j] = t;
        print_array(a);
      }
}

print_array(array)
int array[];
{
  int i;

  for (i = 0; array[i] != -1; i ++)
    printf("%d ", array[i]);
  putchar('\n');
}
```

Figure 5.7 Inferring procedures from an execution trace
(a) bubblesort program with procedure calls,
(b) automaton derived from trace of (a),
(c) the effect of extracting a procedure from (b)

In the automaton, a procedure appears as a group of states which can all be reached from one 'source' state (the first expression in the procedure), and from which a unique 'sink' state (followed by a return as determined in step one above) is reachable. Furthermore, the only states in the procedure that may branch to or from states outside the group are the source and sink sates. That is, a procedure is defined as a set of nodes which all have a common ancestor, and a common descendant which is followed by a return. The source node may only be connected to the rest of the graph by incoming edges, and the sink only by outgoing edges. For example, in Figure 5.7b, the only transitions entering the grey area go to the start of the procedure, $i = 0$, and the only transitions leaving the area emanate from the end of the procedure, the starred node. Given this definition, detecting the start of the procedure is tantamount to identifying the whole procedure. Once the start is known, all the paths leading from the start state to the end state are followed. All of the nodes traversed belong to the procedure. In Figure 5.7b, the procedure could not include the node directly above the grey area, because the transition from $a[j] = t$ to $i = 0$ would then lead from outside the procedure to a state that is not the first state in the procedure. The state directly below the procedure, labelled $a[j] = t$, could not be incorporated, because the transition from it is deterministic, and only states with non-deterministic transitions may be the last state in the procedure.

The terms *ancestor* and *descendant* were used above to denote nodes that precede or follow other nodes in the graph. These terms only make sense in an acyclic graph, because a cycle allows a node to be its own ancestor and descendant. For example, the presence of a single transition from the final state in the automaton to the first state makes every state an ancestor of every other state. Because we are interested in programs that contain iteration, the automaton is usually cyclic. To allow ancestor and descendant to be defined, an acyclic structure must be imposed on the graph for the purposes of detecting a procedure.

The acyclic structure is computed in the following way. The graph is traversed depth-first, as if it were already acyclic. If a transition leads from the current state to a state on the path from the root to the current state, it is marked as a loop. Loops are not followed by the traversal procedure. When the traversal is completed, ignoring the loops allows the graph to be laid out as a tree, where the branching structure defines the layout and the loops are incidental.

```
1   while there are non-deterministic branches,
2       find-procedure(root)

3   global start, success, end

4   find-procedure(node)
5       let start = node
6       let success = false
7       find-procedure-end(node)
8       if success,
9           remove the subgraph beginning at start and ending at end
10      otherwise
11          for each non-loop transition from node
12            find-procedure(descendant of node)

13  find-procedure-end(node)
14      if node has non-deterministic branches,
15          if another end has already been found,
16            let success = false
17          otherwise,
18            let end = node, let success = true
19      otherwise, if node has a loop to an ancestor of start
20            let success = false
21      otherwise,
22          for each non-loop transition from node
23            find-procedure-end(descendant of node)
```

Figure 5.8 Algorithm for identifying a procedure in an automaton

The definitions above can now be transformed into an algorithm, which is shown in Figure 5.8. *Find-procedure* is called with a candidate procedure start, initially the root node (line 2). When it returns to the top level, a procedure has been identified and removed from the main graph. *Find-procedure* is called repeatedly until all procedures have been identified—that is, until there are no non-deterministic branches remaining (line 1).

For each candidate *start* (line 5), *find-procedure-end* (line 13) is called to identify the end of the procedure (line 7). If this is successful—that is, a unique *end* node is found, and the intervening nodes have no transitions going out of the procedure—then the subgraph starting at *start* and ending at *end* is removed and made into a new procedure (line 9). If it is unsuccessful, *find-procedure* calls itself recursively on all of the current *node*'s descendants, looking for other possible procedure starts (line 11-12).

Find-procedure-end (line 13) is called with a candidate start node, and calls itself recursively to descend down the graph. If it encounters a node that is a procedure return—a node with non-deterministic branches—it checks to see whether a return has already been identified. If one has, the procedure sets *success* to false and returns. Thus a procedure must have a single return point. If it is the first return to be identified, it sets *success* to true, and records the state in *end*. If the node does not have non-deterministic branches, all of the loops out of that node are examined to see if they lead to ancestors of the *start* node. If they do, the constraint forbidding transitions out of the procedure has been violated, and *success* is set to false. If the node is not a return node, and loops lead back into the procedure, *find-procedure-end* is called recursively on the non-loop transitions to continue its search for the end of the procedure.

Once the start has been identified, all of the nodes visited are removed from the graph, made into a separate subgraph, and labelled as a procedure. All of the parent nodes of the start node are joined to a node that calls the procedure, and this new node is joined to the appropriate child node of the end node of the procedure. This matching can be performed based on the order in which transitions are added—transitions coming into the start node are added in the same order as the corresponding transitions out of the end node.

5.2.2 Comments on the procedure identification algorithm

This procedure potentially examines all nodes in the graph as candidate procedure starts. For each of these, it traverses a potentially large space of the graph looking for the end. For this reason, the process is quadratic in the number of nodes in the graph, i.e. in the number of unique instructions in the trace. By traversing the graph from the root looking for a procedure start, the largest procedure possible will be identified. This avoids a problem with a simpler and more efficient scheme that would operate as follows. Since the last node in a procedure is known, it should be possible to traverse the graph in the reverse direction until all paths came together. This would avoid the expensive search through the whole tree, evaluating each node as a potential procedure beginning.

Consider, however, the graph in Figure 5.9. The correct procedure is the larger box, containing six nodes. This would be identified by the algorithm in Figure 5.8, but the more efficient algorithm would identify the smaller procedure containing three

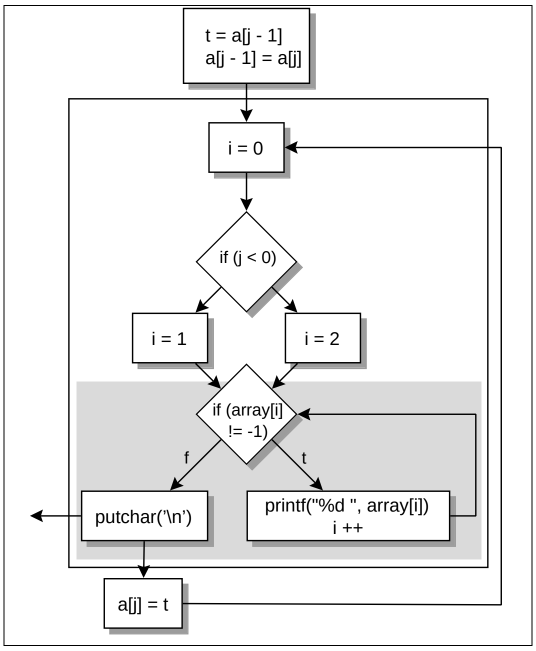


Figure 5.9 A case where a simpler algorithm identifies the wrong procedure

nodes. This is because starting at the return node and working back identifies the node where the two branches rejoin as the start node: it has two transitions into it that appear to match the two branches out of the procedure.

In some cases, the algorithm for identifying loops depends on the graph traversal order. The sequences in Figure 5.10a and Figure 5.10b produce the same automaton. Consider a traversal algorithm that iterates over branches in the order in which they were added. When the automaton is built using the sequence in Figure 5.10a, the sequence *ed* appears before *de*, so the traversal algorithm infers that the *de* transition is a loop, as marked in the automaton with the symbol O. In the sequence in Figure 5.10b, however, *de* appears before *ed*, so the transition from *e* to *d* is labelled a loop. Because the procedure-finding algorithm utilises the acyclic version of the graph, and because the transitions included in the acyclic graph depend on the traversal order of the graph, the algorithm for identifying loops is sensitive to the presentation order of the sequence, whereas the rest of the automaton inference algorithm is not.

The problem does not in fact lie with the identification algorithm but with the automaton. The situation for which the procedure identification method is designed is one where the sequence is the output of a procedural program, and branches are the result of conditionals. Nodes *b* and *e* are the result of a condition at *a*. Node *d* is

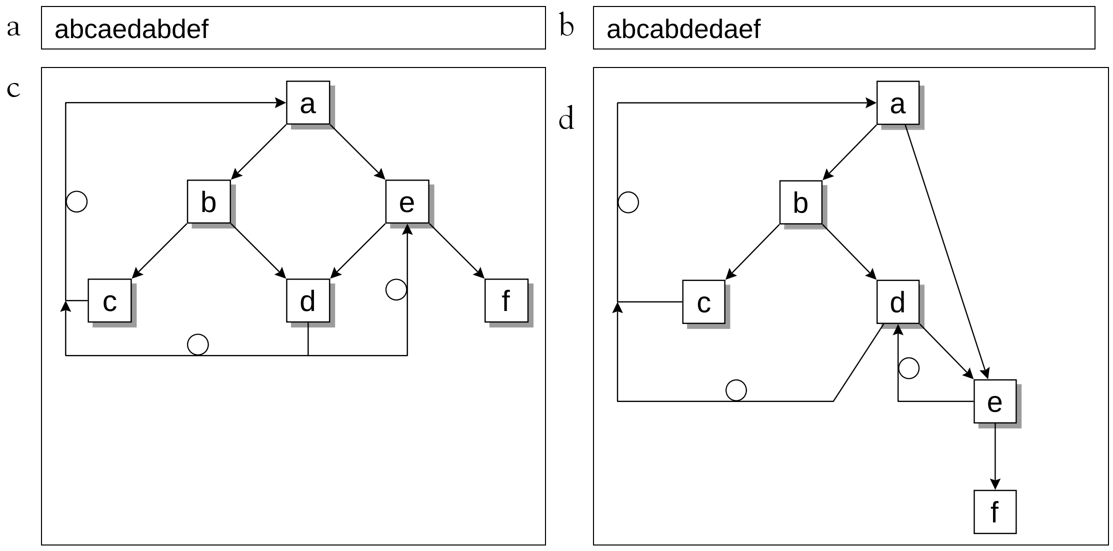


Figure 5.10 Sensitivity of loop-finding algorithm to sequence order
(a) & (b): two sequences that produce the same automaton
(c) identifying loops based on the automaton produced from (a)
(d) identifying loops based on the automaton produced from (b)

a result of conditions at *b* and *e*. It is impossible to construct a conditional in a high-level language that would give rise to a construction where a branch of one condition is also a branch of another condition, and where the both branches can lead back to one of the conditions. In traces of actual programs, loops only go to ancestors that are determined by both paths up from a node. That is, a loop could go to *a* because it is an ancestor via *b* as well as via *e*. Thus, the procedure finding algorithm is only claimed to operate correctly on sequences produced by a program trace, not on arbitrary sequences.

5.2.3 The effect of recursion

To study the effect of recursion, consider the prototypical doubly recursive program in Figure 5.11a. This program, while it does not perform a useful function, illustrates how the recursive program appears when a non-recursive automaton is inferred from a trace. The beginning of the procedure is marked by an instruction *enter recursion*, and the condition that terminates recursion is marked *no more recursion*. The program makes two recursive calls, before which the instruction *before recursion* has been inserted, along with *between recursion* and *after recursion* in appropriate places. Figure 5.11b shows the automaton produced from a trace of Figure 5.11a. There are two states with non-deterministic branches, which are shaded grey. Replacing the non-deterministic transitions out of these states with returns leaves two states with no incoming transitions: *between recursion* and *after recursion*, as shown in Figure 5.11c. Because the transitions to these states were non-deterministic, corresponding to procedure returns, the states must represent instructions that follow procedure calls. This information will be used when the procedure calls are inserted.

The beginning of the procedure can then be identified described above. This finds the state that begins with *enter recursion*. The procedure is isolated in Figure 5.11d, and marked by a grey rectangle. The two transitions to the start state must be procedure calls, so they are removed, and the instruction *call procedure* is inserted in their place. Finally, by re-examining the trace, the order of the calls and the states following calls can be determined, and the automaton can be reconstructed in Figure 5.11e. This involves matching the two states mentioned earlier with the appropriate procedure calls.

Section 5.1 also discussed recursion. Recursion in L-systems is, however, somewhat different to recursion in programs: the examples in Chapter 4 all produce a very regular pattern, giving rise to similar rules at different levels of recursion. In programs, branches are dictated by data outside the context of the sequence, so the rules that would be formed are much less regular. That is, whereas a rule in an L-system will give rise to a fixed number of symbols that can head rules and thus continue the recursion, the execution of a recursive program such as *quicksort* will recurse to a depth dictated by the number of items that it is required to sort. This leads to the different approach for inferring each kind of recursion.

a
```
int recurse(int n) {
  enter recursion;

  if (n == 0) {
    no more recursion;
    return;
  }
  else {
    before recursion;
    recurse(n - 1);
    between recursion;
    recurse(n - 1);
    after recursion;
    return;
  }
}
```

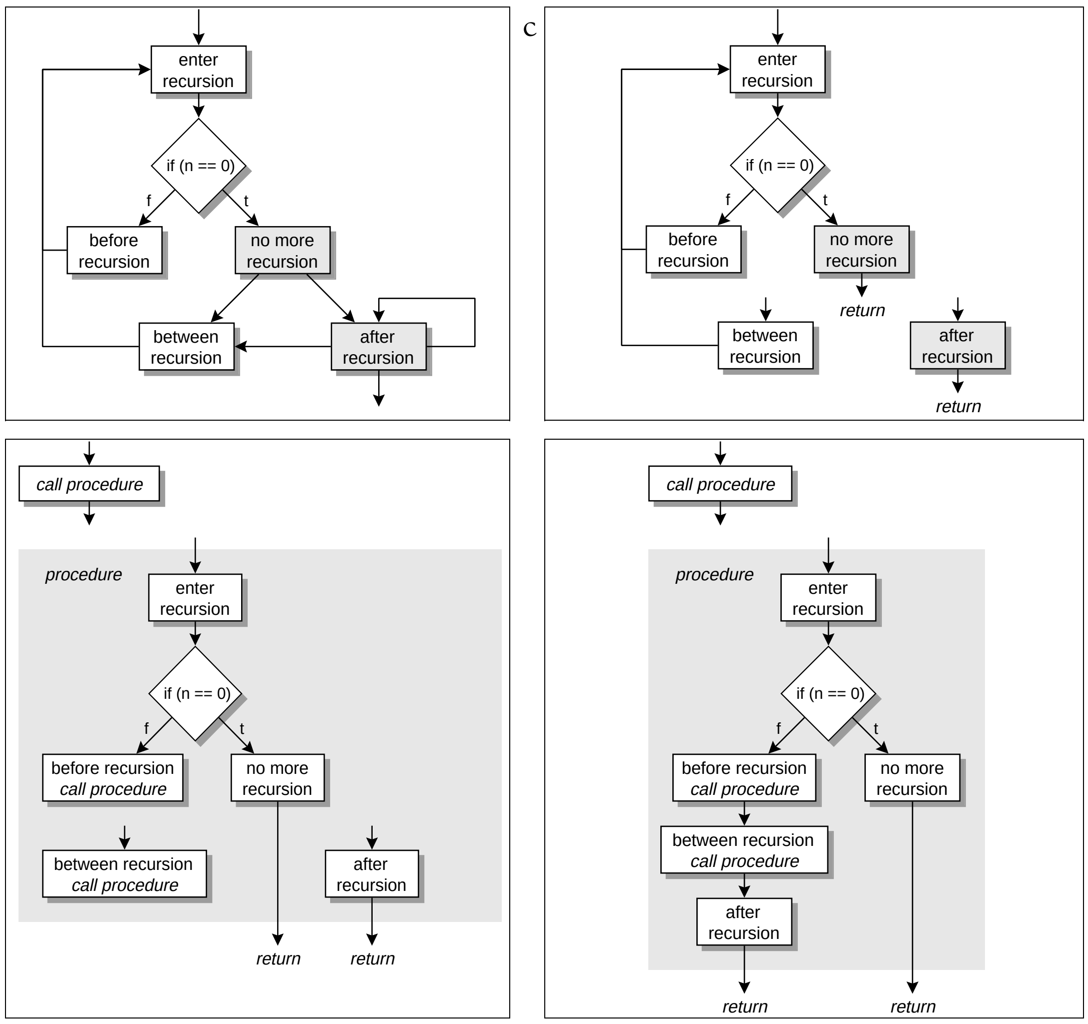
b c


Figure 5.11 The effect of recursion
(a) prototypical doubly-recursive program
(b) automaton produced from trace of (a),
(c) the effect of treating non-determinism as function returns in (b),
(d) part (c) with the procedure identified, and procedure calls inserted
(e) part (d) with states before and after calls matched up.

5.2.4 Summary of procedure identification

It has been shown that programs that contain only iteration can be easily reconstructed from a trace of their execution. Furthermore, it is also possible to infer programs that contain procedure calls and recursion from traces by constructing a push-down automaton. This second operation takes quadratic time in the number of unique instructions. Applications of these techniques to actual situations where program traces are available has not been investigated. Rather, the principles will be applied, in the next section, to sequences that originate from sources other than programs. The branching and looping structures that exist in programs also present themselves in sequences such as those generated by context-free grammars, and the same structure detection techniques turn out to be useful. The recognition of procedure calls and recursion, however, will not be built on.

5.3 Constructing automata from other sequences

So far in this chapter, sequences have been generalised in two ways. Section 5.1 showed how to transform a non-recursive grammar into its recursive original. Section 5.2 showed how branching and looping structure can be identified using a simple automaton induction procedure. SEQUITUR, as described in Sections 3 and 4, excels at recognising exact repetition, but has no way of capturing non-determinism: any non-determinism ends up as a residue in rule *S*. On the other hand, the automaton inference technique easily captures non-deterministic structure but, as will be shown in this section, is ineffective when the sequence is not presented at the appropriate level of granularity.

This section shows how SEQUITUR's ability to identify significant segments in a sequence can be combined with the automaton identification technique's prowess at recognising branching and looping structures. The case studies come from sequences that are generated by non-deterministic grammars. A ready source of these is found in program source code. Although this section deals with program code, it does so in a way that is completely unrelated to that in Section 5.2—instead of analysing the order in which the lines are executed, it is concerned with the text of the programs at a character level.

The section starts by providing a motivating example, where the automaton technique discussed in Section 5.2 is ineffective in detecting structure but, in cooperation with SEQUITUR, captures structure perfectly. Section 5.3.2, however, demonstrates that slight variations on a sequence make detecting branching structure much more difficult. Section 5.3.3 shows that even the minimum description length principle is of little use in justifying intuitive notions of structure, at least in small sequences. Despite the gloomy prognosis for non-deterministic structure detection, Section 5.4 will present a success story, where a larger sequence allows MDL to be usefully applied.

5.3.1 Inferring branches and loops

Figure 5.12a shows part of a C program—a switch statement.[3] Visually, the structure of the statement is clear: the keyword *switch* and a pair of curly braces enclosing a list of *case* labels with colons and line breaks in well-defined places. However, some of this clarity is due to the layout of the sequence—the special symbols that represent white space and vertical delimiters—as well as, potentially, the reader's familiarity with the grammar of the C programming language. The aim of this subsection is to detect this structure automatically, without specialist knowledge.

Given that the sequence consists of a fixed skeleton interspersed with variable parts such as the case labels, it seems reasonable to apply the automaton modelling technique in order to capture the branching structure. Doing this at a character level results in the automaton in Figure 5.12c. The messy graph reveals little about the structure of the sequence. The problem is that the branching and looping structure does not apply to the individual symbols, but to groups of symbols. For example, there is one node for the symbol *s*, regardless of whether it is used in the word *switch* or the word *case*.

At the other extreme, applying SEQUITUR to the sequence results in the grammar in Figure 5.12b. Rule *S* is long, and only exact repetitions in the sequence have been detected. For example, the words *case* and *value* are each represented by a single non-terminal in rule *S*.

[3] The absence of semicolons and a break statement at the ends of the lines will be explained later.

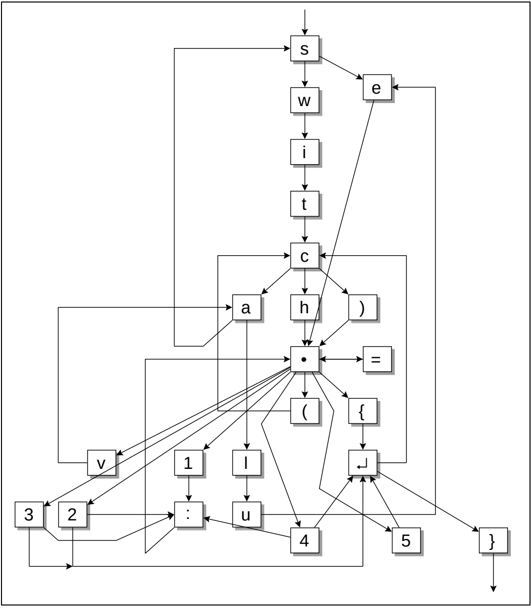
a

```
switch (c) {
 case 1: value = 2
 case 2: value = 3
 case 3: value = 4
 case 4: value = 5
}
```

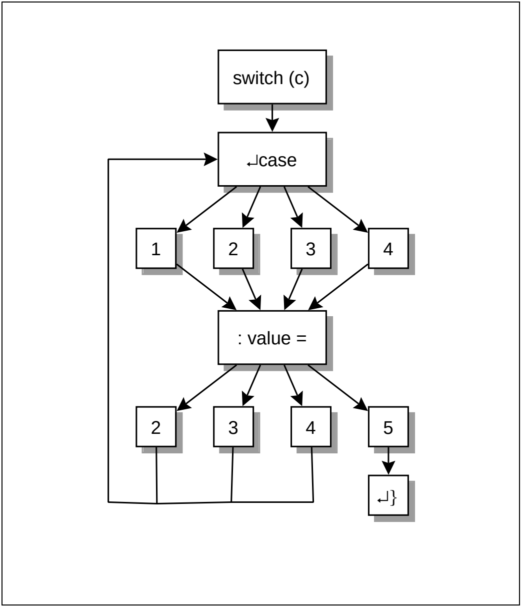
b

S → switch (c) {A1B2A2B3A3B4A4B5↵}
A → ↵case
B → : value =

c



d



e

S → switch (c) {AAAA↵}
A → ↵case B: value = C
B → 1|2|3|4
C → 2|3|4|5

f

S → switch (c) {D↵}
A → ↵case B: value = C
B → 1|2|3|4
C → 2|3|4|5
D → DA|A

Figure 5.12 Detecting the structure of a C switch statement
(a) the switch statement
(b) an automaton formed from the individual characters in (a)
(c) SEQUITUR's grammar for (a)
(d) an automaton formed from the contents of rule *S*
(d) inference of two non-deterministic rules from the branch in (d)
(e) inference of a recursive rule to account for looping in (e)

Neither technique on its own is capable of capturing the whole structure of the sequence. However, a combination of the techniques—SEQUITUR for identifying significant segments like *case* and *value*, and the automaton technique for capturing branching and looping, is more successful. Presenting rule *S* as a sequence to the automaton technique results in the automaton in Figure 5.12d, which describes the structure well. The various possibilities, 1, 2, 3, and 4, between the *case* node and the '*: value =*' node suggest a kind of branch structure. This can be implemented in

a	S → ...AAAA...	AAA represents two overlapping AAs
b	S → ...AAAA... D → AA	A rule is formed for AA
c	S → ...DAA... D → AA	The second AA has become DA...
d	S → ...DAA... D → AA D → DA	so a disjunctive rule is formed for D...
e	S → ...DA... D → AA D → DA	and D replaces DA.
f	S → ...D... D → AA D → DA	D replaces DA for the fourth A
g	S → ...D... D → DA|A	The rules are rearranged for conciseness

Figure 5.13 Forming a recursive rule for a loop

the grammar by forming a new disjunctive rule *B* → 1 | 2 | 3 | 4 that can take any of the values in the branch as its right hand side. The grammar with this modification, along with a similar one for the branch between the ‘*: value =*’ and ‘;↵’ nodes, is shown in Figure 5.12e. The generalised grammar requires extra information to recreate the original sequence. Each time the rule *B* occurs, two bits are required to distinguish between the four possibilities, and similarly for rule *C*. This totals 4×2 + 4×2 = 16 bits extra.

Having inferred a branch, the next structure in the automaton is a loop, to cover the sequence *AAAA* in rule *S* of Figure 5.12e. In a grammar, loops are produced by recursion, as in Figure 5.12f. *AAAA* has been replaced by rule *D*, which expands either to *DA* or simply to *A*. Applying the recursive rule *D* → *DA* many times results in the production of a *D* followed by a string of *A*s. Applying the second rule *D* → *A* terminates the recursion.

This rule can be inferred by a simple modification to the original SEQUITUR algorithm. *AAA* consists of a pair of overlapping *A A* digrams, shown in Figure 5.13a. In the original algorithm that produces deterministic grammars, overlapping digrams are prevented, because a rule covering *AA* can only be used once. However, if this restriction is lifted, the rules are formed in the following way. First, the rule *D* → *AA* is formed as usual, as shown in Figure 5.13b. Next, the first *AA* is replaced by *D*, so the *AAA* becomes *DA*, as in Figure 5.13c. In the non-overlapping case, there would still be *AA* to be replaced by *D*, but here the digram is *DA*, so a new version of rule *D* is created: *D* → *DA*, as in Figure 5.13d, and *DA* is replaced by *D* in 5.12e. This forms the recursive structure necessary for producing loops. The fourth *A* is part of a digram *D A*, so can be replaced by *D*, as in Figure 5.13f. In this case, the original can be reproduced by expanding *D* to *DA*, *DA* to *DAA*, and then *DAA* to *AAAA*. By removing one of the *A*s from the right-hand side of *D* → *AA*, the rules produce the same set of sequences, along with a new sequence *AA*. This is a more concise statement of the loop. Applying this process to Figure 5.12e produces Figure 5.12f, a recursive, non-deterministic grammar.

These techniques are further discussed in Nevill-Manning *et al.* (1994a), Nevill-Manning (1995) and Nevill-Manning and Witten (1995).

a
```
switch (c) {
 case 1: value = 2; break;
 case 2: value = 3; break;
 case 3: value = 4; break;
 case 4: value = 5; break;
}
```

b
```
S → switch (c) {A1B2C2B3C3B4C4B5D↵}
A→ ↵ case
B → : value =
C → DA
D →; break;
```

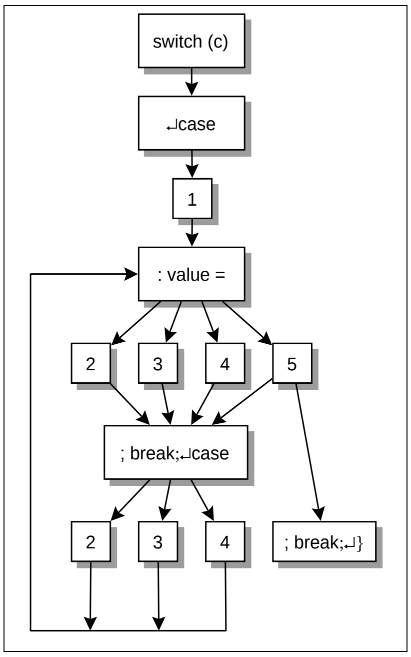
c


Figure 5.14 A switch statement with break statements
(a) the text of the switch statement
(b) SEQUITUR's grammar for (a)
(c) the automaton for rule *S* of (b)

5.3.2 Problems with inferring non-deterministic structure

Unfortunately, this example is idealised. Figure 5.14a is identical to Figure 5.12a apart from the addition of '; *break*;' at the end of each *case* line. This is correct for a C switch statement, but it presents a problem. The rule $A \rightarrow$ ↵ *case* is formed as before, as shown in Figure 5.14b, but after the third line, a longer rule, $C \rightarrow DA$ is created, which prepends the *break* statement of the previous line to the last three *case* lines. In rule *S*, then, the first line starts with *A*, whereas the others start with C. This disrupts the *A*…*B* structure in Figure 5.12d, and produces the automaton in Figure 5.14c. The problem is that in the last three *case* clauses, rule *A* has 'submerged' beneath the visible level of rule *S*, and is hidden within rule C.

In this particular example, a process similar to reparsing provides a solution. Retrospective reparsing, as described in Section 4.4, looks at the suffixes of the symbols that precede a particular symbol. If the process were to look *two* symbols back instead of just one, it would notice that both C and *A* end with *A*, expand C, and form the better branch. However, this solution only works if both C and *A* are both two symbols back in the sequence from *B*. If the case labels were of different lengths, as they often are, the submerged *A*s would be missed. Generalising the procedure to cope with this possibility results in a traversal of the tree moving back along transitions from *B*, examining the contents of every node and finding a set of nodes with common suffixes. Removing the fixed number of transitions means that many solutions may be found, some of them spurious. Furthermore, the search is now linear in the number of nodes in the graph, and this process must be performed starting at each node in the tree in order to recognise all similar correspondences, so the search is $O(n^2)$. This process is nevertheless feasible for some applications, and section 6.3 develops it further. The rest of this section describes a different approach—detecting branches incrementally.

Can branches be recognised during the processing of the sequence? The example starts with a *case* label that is not preceded by a *break*, and when the second *case* appears, rule *S* contains the sequence '*switch* (*c*) {*A*1*B*2*break;A*2*B*'. An incremental recognition procedure would infer a branch from the *A*…*B*…*A*…*B* structure. Having formed this branch, the third and fourth *case* statements will match the branch rather than forming a longer rule with the preceding *break*. Performing branch recognition incrementally has two advantages over a traversal of the graph at the end. First, in an interactive context, predictions and inferences are available immediately to the system for feedback to the user. Second, if analysis is performed on the fly, branches like the *case…value* branch can be captured before rule C is formed.

There are, however, two significant problems with this approach. First, it will not always be successful, because it relies on the order in which examples appear. If the *case* statement without the *break* appeared last, the incremental technique would form the three-way branch with rule C, and the final *case* statement would be left on its own. It is difficult to justify not forming a rule before the negative example has been seen. Of course, this is just the situation that reparsing was intended to fix, by revising decisions later on in the sequence. Unfortunately, as mentioned above, the change from inspecting the symbol immediately previous to inspecting one an arbitrary number of symbols previous makes the search much more expensive.

The second and more fundamental problem is the issue of justification for making such an inference. There are potentially many apparent relationships that are the result of random chance, and do not result from actual structure in the source. In particular, there are many more branching structures that it is possible to infer than there are duplicate digrams. Consider a string of length n over a binary alphabet $\{a, b\}$. The number of digrams in the string is $n - 1$. There are four unique digrams: *aa*, *ab*, *ba* and *bb*, so the expected number of each digram in the sequence is $(n - 1)/4$. Each pair of duplicate digrams could give rise to a rule. There are

$$\frac{1}{2}\frac{n-1}{4}\left(\frac{n-1}{4}-1\right)$$

such pairs, for each unique digram, so there are four times as many,

$$\frac{n-1}{2}\left(\frac{n-1}{4}-1\right)$$

possible rules in total. This expression is quadratic in n.

As for branching structures, these can be of two forms:

…a…b…a…b… or
…b…a…b…a…

An ellipsis represents any intervening symbols, i.e. {a, b}*. The second and fourth set of ellipses are the two different paths of the branch. There are sixteen forms in total, from *aaaa* (omitting the ellipses) through *aabb* to *bbbb*. In a sequence of length n, there are ${}^{n}C_4$ ways of selecting one of the sixteen forms, so the expected number of branches as above is

$$\frac{1}{8}{}^{n}C_4.$$

This is $O(n^4)$. The absolute numbers will decrease for larger alphabets with the same n, but the effect remains. The comparatively large number of branches versus repetitions—$O(n^4)$ vs $O(n^2)$—is one reason for the increased difficulty of inferring correct branches over inferring correct repetitions.

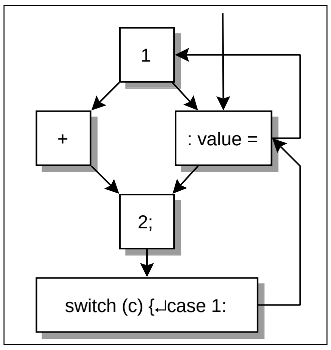
Inferring a branch on this basis could be disastrous. For example, Figure 5.15 shows the start of the case statement preceded by the assignment *value* = 1 + 2. A branch can be inferred from the resulting grammar, because of the pattern 1…*B*…1…*B*. In this case, however, the parts between 1 and *B* have little in common: they are + and '*: value* =.' These are made into a new rule, which in this case bears no relation to intuition about the sequence's structure. Furthermore, it is now impossible for the first *case* line to take part in a branch.

5.3.3 Justifying inferences by the MDL principle

It is therefore necessary to devise a test for choosing the *case*…*value* branch over the 1…2; branch. One way of doing this is to examine whether inferring a branch

a
```
value = 1 + 2;
switch (c) {
 case 1: value = 2;
```

b
```
S → A1 + Bswitch (c) { case 1 : AB
A → value =
B → 2;
```

c


d
```
S → ABswitch (c) { case B
A → value =
B → 1C2;
C → + | : A
```

Figure 5.15 A spurious branch
(a) the text of the program fragment
(b) SEQUITUR's grammar for (a)
(c) the automaton for rule *S* of (b)

allows the sequence to be encoded more concisely: to justify the inference using the minimum description length principle. The number of symbols in each grammar will serve as a rough estimate of the size of a grammar. The result is surprising and disconcerting: the original grammar in Figure 5.12b has 43 symbols. The grammar with the branch inferred in Figure 5.12e has 50 symbols: seven *more* symbols than the explicit grammar. Even worse, sixteen bits are required to reproduce the original sequence from the second grammar, whereas no extra information is required for the first grammar. There is little point comparing the size of the 1…2 branch if even the preferred grammar cannot be justified using MDL.

This situation is counter-intuitive, and in some ways misleading (it was certainly not clear at the outset of this research). The case statement *looks* structured, but the branching structure is not so intrinsic in the sequence as to be unquestionable. The gap between intuition and mathematics is partly caused by the rule-forming process. Looking at the correspondence between *case* and *value* suggests a useful prediction: the whole element *value* can be predicted by conditioning on the previous *case*. However, *value* is reduced to a single non-terminal symbol by SEQUITUR. This implies that much of the token *value* is predicted better by previous occurrences of the word *value*: once *v* appears, the rest of the sequence suggests the continuation *alue*. The apparent usefulness of predictive branches is greatly reduced when simple repetitive structure is recognised.

There is an important difference between forming the hierarchical phrases discussed in Section 3 and the branching structures described here. In the first case, retracting a parse can be easily performed by expanding a rule. Also, examining the sequence in the absence of a rule can be performed by examining the contents of a rule rather than looking at the non-terminal on the surface. When a branch has been inferred, the non-terminal that is created to stand for the intervening symbols may group together sequences that have no relationship to each other, whereas the basic algorithm at least knows that symbols are adjacent to each other.

The fundamental problem is that inferences can never be completely justified, so there is always potential for generalising the grammar incorrectly by acting on one. The next section uses an MDL justification, but acts at the end, and orders inferences based on the amount of information they save. In this way, the most justified inferences are made first.

5.4 Detecting structure in semi-structured text

This section approaches the problem of generalising a grammar in a different way from the previous section. Here the generalisation is performed on a grammar that is many orders of magnitude larger than the grammars discussed in the last section. Whereas the switch statement occupies 91 bytes, the sequence treated here is 9 Mb in size—100,000 times larger. The inference process, then, is a lot less ambiguous, because the effect of a useful generalisation in the model is marked. The choice of generalisation is made based on a calculation of the ability of one symbol to predict another. The effect of these generalisations on compression performance will be discussed in Section 6.4. The sequence discussed in this section is a textual database containing genealogical information about 38,000 individuals.

5.4.1 The genealogical database

The Church of Jesus Christ of Latter Day Saints, for various reasons, maintains the most comprehensive collection of on-line genealogical information in the world. The two largest of these databases are the International Genealogical Index, which contains birth, death and marriage records, and the Ancestral File, which contains linked pedigrees for families all over the world. The former contains the records of over 265 million people while the latter contains records for over 21 million; they are growing at a rate of 10% to 20% per year.

The database is expressed in a semi-structured textual form that is specifically designed to represent genealogical information. Figure 5.16 shows an example of an individual record, a family record, and the beginning of another individual record. The first gives name, gender, birth date, birthplace, and a pointer to the individual's family. The family record, which follows directly, gives pointers to four individuals: husband, wife, and two children—one of which is the individual himself. This example, however, gives an impression of regularity which is slightly misleading. For most of the information-bearing fields such as NAME, DATE, and PLACE, there are records that contain free text rather than structured information. For example, the last line of Figure 5.16 shows a name given with an alternative. The DATE field might be 'Abt 1767' or 'Will dated 14 Sep 1803.' There is a NOTE field (with a continuation line code) that frequently contains a brief essay on family history.

```
...
0 @26DS-KX@ INDI
1 AFN 26DS-KX
1 NAME Dan Reed /OLSEN/
1 SEX M
1 BIRT
2 DATE 22 JUN 1953
2 PLAC Idaho Falls,Bonneville,Idaho
1 FAMC @00206642@
0 @00206642@ FAM
1 HUSB @NO48-3F@
1 WIFE @93GB-DD@
1 CHIL @26DS-KX@
1 CHIL @21B7-WR@
0 @26DN-7N@ INDI
1 NAME Mary Ann /BERNARD (OR BARNETT)/
...
```

Figure 5.16 An excerpt from the GEDCOM genealogical database

There is great variability in the kinds of genealogical information that must be stored. Genealogical evidence comes from a wide variety of source records from a variety of cultural norms. This creates a strong requirement for flexibility. Despite this there is a high degree of structure. Places, dates and significant events all occur with great regularity and where similar information occurs, a similar representation is used. The GEDCOM standard for exchanging genealogical information stores information as textually encoded trees. Each node has a tag which identifies its type and some textual content. The content of a node is highly stylised depending on the node's type. Any node can have child nodes that provide additional information. This form is very flexible, easily transmitted and yet amenable to automatic processing because of its tree structure. Its variability, however, makes it more suited to full-text retrieval than to traditional database queries.

The records are variable-length, and may have any combination of fields. Each record is a line of text with a level number, tag and textual contents. The level numbers provide a hierarchy within a record in a scheme reminiscent of COBOL data declarations. Lines at the zero level can have labels that are used elsewhere in the file to refer to the entire record. These records are generally individuals (*INDI*) or families (*FAM*).

5.4.2 Learning the structure of the genealogical database

The data was presented to SEQUITUR as a sequence of words. Virtually all words were separated by single spaces, and so it was not necessary to have a separate sequence of non-words. Instead, a single space was defined as the word delimiter. On the rare occasions where extra spaces occurred they were prepended to the next word—this generally happened only with single-digit dates. Other punctuation was appended to the preceding word; again, this decision did not materially increase the dictionary size. The dictionary was encoded separately from the word sequence, which was represented as a sequence of numeric dictionary indexes.

The input comprised 1.8 million words, and the dictionary contained 148,000 unique entries. The grammar that SEQUITUR formed had 71,000 rules and 648,000 symbols, 443,000 of which were in the top-level rule. The average length of a rule (excluding the top-level one) was nearly 3 words.

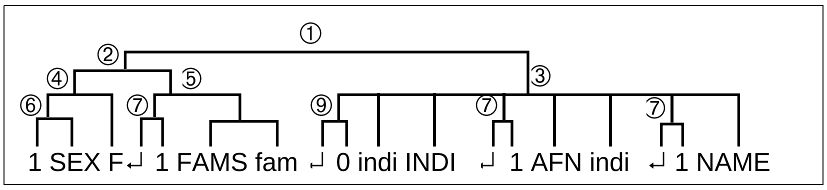
Examination of SEQUITUR's output reveals that significant improvements could be made quite easily by making small changes to the organisation of the input file. We first describe how this was done manually, by using human insight to detect regularities in SEQUITUR's output; next, we show how the grammar can be interpreted and finally, we show how the process of identifying such situations can be automated.

Manual generalisation

Of the dictionary entries, 94% were codes used to relate records together for various familial relationships. Two types of codes are used in the database: individual identifiers such as @26DS-KX@, and family identifiers such as @00206642@. These codes obscure template structures in the database—the uniqueness of each code means that no phrases can be formed that involve them. For example, the line 0 @26DS-KX@ *INDI* in Figure 5.16 occurs only once, as do all other *INDI* lines in the file, and so the fact that 0 and *INDI* always occur together is obscured: SEQUITUR cannot take advantage of it. In fact, the token *INDI* occurs 33,000 times in the rules of the grammar, and in every case it could have been predicted with 100% accuracy by noting that 0 occurs two symbols previously, and that the code is in the individual identifier format.

This prediction can be implemented by replacing each code with the generic token *family* or *individual*, and specifying the actual codes that occur in a separate stream. Replacing the code in the example above with the token *individual* yields the sequence *0 individual INDI*, which recurs many thousands of times in the file and therefore causes a grammar rule to be created. In this grammar, *INDI* occurs in a rule

a
```
① → ②③
② → ④⑤
③ → ⑨individual INDI⑦AFN individual⑦NAME
④ → ⑥ F
⑤ → ⑦⑧
⑥ → 1 SEX
⑦ → ↵1
⑧ → FAMS family
⑨ → ↵0
```

b


c
```
... ① Sybil Louise /MCGHIE/ (① Eliza) Jane /LOSEE/ (① Margaret) /SIMMONS/
(① Marie) Elizabeth /BERREY/ ① Athena Augusta /ROGERS/ (① William) Henry /WINN/ ...
```

Figure 5.17 A phrase from the genealogical database
(a) hierarchical decomposition
(b) graphical representation
(c) examples of use

that covers the phrase *↵0 individual INDI↵1 AFN individual↵1 NAME*. This is now the only place that *INDI* occurs in the grammar.

Overall, the number of rules in the grammar halves, as does the length of the top-level rule, and the total number of symbols.

Interpreting the grammar

Figure 5.17a shows nine of the 71,000 rules in SEQUITUR's original grammar, with ungeneralised codes, renumbered for clarity. Rule ① is the second most widely used rule in the grammar: it appears in 261 other rules.[4] The other eight rules are all those that are referred to, directly or indirectly, by rule ①: Figure 5.17b shows the hierarchy graphically. The topmost line in Figure 5.17b represents rule ①. The two branches are rules ② and ③, the contents of rule ①. The hierarchy continues in this way until all of the rules have been expanded.

Rule ① represents the end of one record and the beginning of the next. Rule ⑨ is effectively a record separator (recall that each new record starts with a line at level

4 The most widely used rule is 2 PLAC, which occurs 358 times, indicating that the text surrounding the place tag is highly variable. However, the structure of the rule itself is uninteresting.

0), and this occurs in the middle of rule ①. Although grouping parts of two records together achieves compression, it violates the structure of the database, in which records are integral. However, the two parts are split apart at the second level of the rule hierarchy, with one rule, ②, for the end of one record, and another, ③, for the start of the next. The short rules ⑦ and ⑨ capture the fact that every line begins with a nesting level number. There is also a rule for the entire *SEX* field indicating the person is female, which decomposes into the fixed part: *1 SEX*, and the value *F* on the end, so that the first part can also combine with *M* to form the other version of the *SEX* field. There is a similar hierarchy for the end of a male record, which occurs 259 times.

As for the usage of this rule, Figure 5.17c shows part of rule *S*. Here, rules have been expanded for clarity: parentheses are used to indicate a string which is generated by a rule. This part of the sequence consists mainly of rule ① in combination with different names. Separate rules have been formed for rule ① in combination with common first names.

Automatic generalisation

In order to automate the process of identifying situations where generalisation is beneficial, it is first necessary to define the precise conditions that give rise to possible savings. In the case described above, the rule *INDI ↵1 AFN* occurred many times in the grammar, and accounted for a significant portion of the compressed file. Conditioning this phrase on a prior occurrence of *↵0* greatly increases its predictability. The problem is that other symbols may be interposed between the two. One heuristic for identifying potential savings is to scan the grammar for pairs of phrases where the cost of specifying the distances of the second relative to the first (predictive coding) is less than the cost of coding the second phrase by itself (explicit coding).

Figure 5.18 gives two illustrations of the tradeoff. In the top half of Figure 5.18, using normal coding, the cost of coding the *B*s is three times the cost of coding an individual *B*: $\log_2$(*frequency of B / total symbols in grammar*) bits. For predictive coding, the statement ‘*A* predicts *B*’ must be encoded once at the beginning of the sequence. Reducing this statement to a pair of symbols, *AB*, the cost is just the sum of encoding *A* and *B* independently. Each time that *A* occurs, it is necessary to specify the number of intervening symbols before *B* occurs. In the example, *A*<3>

signifies that the next *B* occurs after three intervening symbols. These distances are encoded using an adaptive order-0 model with escapes to introduce new distances.

The bottom half of Figure 5.18 shows a more complex example, where two *A*s appear with no intervening *B*, and a *B* occurs with no preceding *A*. The first situation is flagged by a distance of ∞, and the second is handled by encoding *B* using explicit coding.

The algorithm for identifying useful generalisations as follows. For each terminal and non-terminal symbol, a list is made of all the positions where that symbol occurs in the original sequence. If the symbol appears within a rule other than rule *S*, it inherits all the positions of that rule. Next, for each pair of unique symbols, one symbol is chosen as the predictor, and one as the predicted symbol. The gaps between the predicting symbol and the predicted symbol are calculated as in Figure 5.18, and the total number of bits required to encode the predicted symbol using both explicit and predictive coding is calculated. For each pair, the predictive/predicted roles are reversed, and the savings are recorded in the corresponding cell in the matrix. At the end of this process, which is quadratic in the number of unique terminals and non-terminals, the predictions are ranked, and the most effective ones turned into generalisations.

Table 5.1 shows the top ranking pairs of phrases, the number of bits required for the predicted symbol with the explicit coding, the number of bits required using predictive coding, and the total savings. The ellipsis between the two phrases in the first column represents the variable content between the keywords. At the top of the list is the prediction *↵0…INDI ↵1 AFN*, which is the relationship that we exploited by hand—the intervening symbol between ↵0 and *INDI ↵1 AFN* is an individual code. Predictions 2, 3 and 6 indicate that the codes should be generalised after the *AFN*, *FAMS* and *FAMC* tags respectively. Predictions 5, 7 and 9 indicate that

		explicit	predictive
to encode the Bs in	encode	...B...B...B...	‘A predicts B’, A<3>...A<4>...A<4>
AabcB...AdefgB...AhijkB	cost	3 × cost(B)	cost(A) + cost(B) + cost(3, 4, 4)

		explicit	predictive
to encode the Bs in	encode	...B...B...B	‘A predicts B’, A<3>...A<∞>...A<4>...B
AabcB...A...AhijkB...B	cost	3 × cost(B)	cost(A) + cost(B) + cost(3, ∞, 4) + cost(B)

Figure 5.18 Examples of two ways of coding symbol *B*

	Prediction	Normal	Predicted	Saving
		(bits/symbol)	(bits/symbol)	(total bits)
1	↲ 0...INDI ↲ 1 AFN	3.12	0.01	2298
2	INDI ↲ 1 AFN...↲ 1 NAME	3.12	0.01	2298
3	FAMS...↲ 0	2.25	0.81	638
4	↲ 1 SEX...↲ 2	0.96	0.06	656
5	BAPL...↲ 1 ENDL	1.57	0.76	509
6	FAMC...↲ 1 FAMS	2.57	1.47	427
7	↲ 1 BIRT↲ 2 DATE...↲ 2 PLAC	1.88	1.11	382
8	↲ 1 NAME...↲ 1 SEX	1.25	0.82	315
9	ENDL...↲ 1 SLGC	2.15	1.58	266

Table 5.1 Predictions based on part of the GEDCOM database

dates should be generalised. Prediction 4 indicates that the *SEX* field can be generalised by replacing the two possible tags, *F* and *M*. Finally, prediction 8 indicates that names should be generalised. The generic tokens are equivalent to non-terminals that can have all possible codes, etc. as their bodies.

Figure 5.19 shows Figure 5.16 with the content in grey and the identified structure in black. This shows that the structure and content have been successfully distinguished. By identifying significant phrases and predictive relationships between them, the database template structure has been discovered. This is possible because of the reliable statistics that a large sample provides. In Section 6.4, we will see that these inferences significantly increase compression performance.

```
0 @26DS-KX@ INDI
1 AFN 26DS-KX
1 NAME Dan Reed /OLSEN/
1 SEX M
1 BIRT
2 DATE 22 JUN 1953
2 PLAC Idaho Falls,Bonneville,Idaho
1 FAMC @00206642@
0 @00206642@ FAM
1 HUSB @NO48-3F@
1 WIFE @93GB-DD@
1 CHIL @26DS-KX@
1 CHIL @21B7-WR@
0 @26DN-7N@ INDI
1 NAME Mary Ann /BERNARD
```

Figure 5.19 The text in Figure 5.16, with structure and content distinguished

5.5 *Summary*

This chapter has taken the techniques for finding exact repetitions described in Chapters 3 and 4, and explored ways of generalising their output to model non-determinism. In other words, the grammars produced by SEQUITUR that are only capable of producing a single sequence have been transformed into more general descriptions that describe a family of sequences. We have shown that recursive L-systems can be inferred from their non-recursive equivalent, that programs can be recreated from traces of their execution, and that the structure of text such as programming code and textual databases can be inferred. We have also explored the difficulty of justifying inferences with small samples, and shown how a larger sample can reduce ambiguity.

6. Data Compression

By removing redundancy, data can be stored and transmitted more efficiently, providing a highly practical application of techniques for structure detection. Not only do the techniques described in this thesis detect structure, and hence redundancy, but the very core of the algorithm is motivated by compression. The two constraints—digram uniqueness and rule utility—ensure that redundancy due to repetition is eliminated from a sequence. Digram uniqueness eliminates a repetition of two symbols by forming a rule that both occurrences can reference. Rule utility eliminates superfluous rules when repetitions continue for longer than two symbols.

Detection of repetition is not, however, sufficient for effective data compression. Other regularities, especially the unequal frequencies of the symbols and repetitions in the sequence, must also be accounted for in order to rival other compression techniques. Furthermore, it is inefficient to simply transmit SEQUITUR's grammar in the same way as it is printed. Instead, it is possible to transmit the sequence incrementally and allow the decoder to build the grammar itself. This not only maintains the incremental behaviour of the basic algorithm, but also increases compression. Apart from its practical application, the encoding scheme described here allows SEQUITUR to be compared, in terms of compression effectiveness, to the context-based prediction schemes and the dictionary compression schemes described in Chapter 1, thereby offering a concrete assessment of the predictive abilities of each approach.

Section 6.1 shows how SEQUITUR's grammars can be used to encode a sequence very concisely, and provides rationales for the design decisions that were made to arrive at this encoding scheme. Section 6.2 compares SEQUITUR's performance against other data compression schemes and identifies its advantages and disadvantages relative to them. Section 6.3 discusses the claim made in Chapter 4 that the number of symbols in a grammar is a good measure of its information content. Section 6.4 returns to the genealogical database discussed in Chapter 5, and shows how generalisation improves compression performance.

6.1 Encoding sequences using SEQUITUR

The encoding scheme for SEQUITUR will be explained by successive refinement, in order to demonstrate the advantages of each design decision. The stages are:

- sending the grammar in textual form;
- sending symbols using variable-length codes;
- sending rules implicitly;
- sending the sequence incrementally.

Figure 6.1 illustrates the effect of each encoding scheme in two ways. The first column describes the encoding scheme, while the second illustrates the encoding in detail on the sequence *abcdbcabcdbc*. The third and fourth columns show the effect of the coding scheme on a much longer sequence, *Far from the Madding Crowd*. This novel by Thomas Hardy is 768,771 characters in length. It is part of the Calgary corpus for evaluation of data compression, and so provides a useful comparison for other techniques. The third column of Figure 6.1 shows the size of the novel when encoded using each scheme, while the fourth column gives the size as a percentage of the original sequence length.

	encoding	sequence	size of *book1*	size relative to original
a	original	abcdbcabcdbc	768 771	100%
b	textual grammar	S → BB A → bc B → aAdA	–	–
c	numbered rules	S → #2#2↵ 1 → bc↵ 2 → a#1d#1↵	1 201 189	156%
d	without rule heads	#2#2↵ bc↵ a#1d#1↵	965 955	126%
e	frequency-based codes	②②↵ bc↵ a①d①↵	335 420	44%
f	implicit rules	abcd(1,2)(0,4)	271 303	35%

Figure 6.1 Alternative encodings for a sequence

6.1.1 Sending the grammar textually

The simplest way of transmitting the sequence to take advantage of SEQUITUR's compressive abilities is to send the textual form of the grammar, as shown in Figure 6.1b. However, when the sequence includes uppercase letters, or when there are more than 26 rules, capital letters cannot be used for non-terminals. For example, in the grammar for *Far from the Madding Crowd* there are 27365 rules, and the sequence contains many uppercase letters. To solve this problem, non-terminals are numbered and preceded by a special character, as shown in Figure 6.1c. Also, an end-of-rule marker is necessary, and this is denoted by the symbol ↲. This representation requires eight bits per character, and multiple digits to encode one non-terminal. The *book1* sequence can be unambiguously decoded when stored in this way, resulting in a size of 1,201,189 symbols (rule *S* alone contains 673,977 symbols). This represents an *expansion* of 56%—it would be better to transmit the original sequence than use SEQUITUR's grammar in this form. In some cases, such as the highly structured L-systems described in Chapter 4, this representation does provide compression, but generally it does not.

If the rules are sent in order, rule *S* first, then rules 1, 2, 3 and so on, there is no need to send the symbols that head rules, because their ordering can be reproduced by the receiver. Furthermore, no → symbol is necessary—it can be implicit at the start of each line. Figure 6.1d shows the short sequence encoded in this way. This reduces the encoded size of *book*1 to 965,955: better than the 1,201,189 characters of the full textual representation, but still not compressive.

6.1.2 Sending symbols based on frequency

There are two problems with the textual representation of the grammar. First, non-terminals are represented by several digits. Second, there is no allowance for different length codes for symbols with different probabilities. Both these problems are solved by encoding each symbol based on its probability. That is, instead of transmitting, say, the letter *a* or the five characters #, *1*, *3*, *3* and *7* to represent the non-terminal #1337, a code is used whose length is related to the probability of that symbol occurring in the grammar. The decoder needs to know the symbol codes, so the probability of a symbol is approximated by its frequency in the sequence as a proportion of all symbols sent so far. Because the decoder can calculate the same frequency, it can recreate the same set of codes as the encoder is using to send the message. The symbols can be encoded using an arithmetic coder, as described in Section 2.4. Moffat *et al.* (1995) provide an exemplary implementation of arithmetic coding, along with routines for coping with a large alphabet, maintaining frequencies, and allowing for the introduction of new symbols. This solves the problem of encoding non-terminals—instead of expressing them in terms of the 128-symbol ASCII character set, arithmetic coding treats them as another symbol in a much larger alphabet. Encoding the grammar in this way yields a file size of 335,420 bytes, 44% of the original size.

While this is a distinct improvement over the textual version, it nevertheless falls short of most data compression schemes. Table 6.1 summarises the performance of some notable techniques. UNIX *compress* (Thomas, *et al.*, 1985), which is based on LZ78 (Ziv and Lempel, 1978), popularised the use of Ziv-Lempel coding. It has remained unchanged for over a decade, and performs poorly relative to more recent schemes. It slightly outperforms the simple grammar encoding, SEQUITUR-0, reducing *book1* to 332,056 bytes, 43% of the original. A more recent implementation of LZ77, *gzip*, outperforms *compress*, resulting in a compressed size of 312275 bytes, 41% of the original. The best general purpose compression scheme, PPM (Cleary and Witten, 1984), described in Section 2.4, compressed the novel to 242558 bytes, 32% of the original.

6.1.3 Sending rules implicitly

SEQUITUR should be able to rival the best dictionary schemes, because forming rules is similar to forming a dictionary. Furthermore, SEQUITUR stores its dictionary as a hierarchy, so it should be capable of outperforming other dictionary techniques. This can be achieved if the grammar is sent implicitly. Rather than sending a list of rules, this algorithm sends the sequence, and whenever a rule is used, it transmits

scheme	compressed size	fraction of original size
SEQUITUR-0	335 420	44%
compress	332 056	43%
gzip	312 275	41%
SEQUITUR-I	271 303	35%
ppm	242 558	32%

Table 6.1 Compression of *book1* by several schemes

sufficient information to the decoder so that it can construct the rule. Because rule *S* represents the entire sequence, this is tantamount to sending rule *S* and transmitting other rules as they appear.

When a non-terminal is encountered in rule *S*, it is treated in three different ways depending on how many times it has been seen. The first time it occurs, its contents are sent. At this point, the decoder is unaware that the symbols will eventually become a rule. The second time that a non-terminal occurs, a pointer is sent that identifies the contents of the rule that was sent earlier. The pointer consists of an offset from the beginning of rule *S* and the length of the match, similar to the pointers used in LZ77. At the decoding end, this pointer is interpreted as an instruction to form a new rule, with the target of the pointer as the contents of the rule. The decoder numbers rules in the order in which they are received, and the encoder keeps track of this numbering. When the third and subsequent occurrences of the non-terminal appears, this number is used instead of the number of the non-terminal.

The main advantage of this approach is that the first two times a rule is used, the non-terminal that heads the rule need not be sent. For example, under the previous scheme, a rule that is used twice is transmitted by sending two non-terminals, the rule contents, and an end-of-rule marker. Under the new scheme, only the contents and a pointer are necessary. Furthermore, rules are sent only when they are needed. Sending the grammar rule by rule, starting with rule *S*, usually sends rules before they are referenced, which reserves a code unnecessarily, or references them before they are sent, which delays the decoder's ability to reconstruct the sequence. Sending the grammar for *book1* implicitly produces a compressed output of 271,303 bytes, for compression to 35%, shown as SEQUITUR-I in Table 6.1. This is better than the other dictionary techniques, and only 12% worse than PPM's 32%.

Figure 6.1f shows how the short sequence is transmitted. Because both rules only appear twice, no non-terminals appear in the encoding. The sequence is transmitted by transmitting rule *S*, which consists of two instances of rule 2. The first time rule 2 appears, its contents are transmitted. This consists of *a*①*d*①. The first symbol, *a*, is encoded normally. The first time rule 1 appears, its contents, *bc*, are sent. The next symbol, *d*, is sent as normal. Now rule 1 appears for the second time, and the pointer (1,2) is sent. The first element of the pointer is the distance from the start of the sequence to the start of the first occurrence of *bc*, in this case 1. The second element of the pointer is the length of the repetition: 2. Now the decoder forms a rule $1 \rightarrow bc$, and replaces both instances of *bc* in the sequence with ①. Having transmitted the first instance of rule 2 in its entirety, the encoder returns to rule *S* to transmit the second occurrence of rule 2. The repetition starts at the beginning of the sequence, at distance 0, and continues for 4 symbols. The length refers to the 4-symbol compressed sequence, rather than the uncompressed repetition, which is 6 symbols long.

The phrases that are discovered improve on the dictionaries of the other schemes in four ways. First, the dictionary is stored hierarchically, using shorter dictionary entries as part of longer ones. Second, there is no window to reduce searching time and memory usage at the expense of forgetting useful repetitions. Third, the length of dictionary entries is not limited. Fourth, there are no unnecessary phrases no the dictionary. Each of these advantages is expanded below.

Using a hierarchical representation for the dictionary means that rules can be transmitted more efficiently. The saving comes from the smaller pointer needed to specify both the start of the repetition and its length. Because rules properly contain other symbols—i.e. they do not overlap other non-terminals, the number of places a rule can start is reduced to the number of symbols currently in the grammar. Furthermore, the length of the repetition is expressed in terms of the number of terminal and non-terminal symbols that it spans, rather than the number of original terminals. This means that the length will usually be shorter than the corresponding length specified relative to the original sequence. This corresponds to Storer's (1982) *compressed pointer macro* classification.

The lack of a finite window for pointer targets has several ramifications. First, it undoes some of the improvement achieved by using a hierarchical dictionary, because it allows a greater number of targets for pointers. Second, the lack of windowing usually means that memory usage and search time grows. Memory usage is unavoidable given the basic design of SEQUITUR—efficient memory use was not one of the required qualities. However, SEQUITUR's compressed in-memory representation and efficient indexing procedures mean that the average search time is bounded as shown in Section 3.3. The advantage of the lack of a window is that all repetitions can be detected, no matter how far they are separated.

LZ78 techniques add items to the dictionary speculatively—a particular entry is not guaranteed to be used. This saves the cost of specifying which phrases should be included in the dictionary, but means that the codes assigned to unused entries are wasted. SEQUITUR only forms a rule when repetitions occur, combining the LZ77 policy of specifying a repetition only when needed with the LZ78 technique of maintaining a dictionary. Furthermore, LZ78 techniques grow the dictionary slowly, whereas SEQUITUR can send a rule of any length in one step. This does not always result in superior compression: Table 6.2 shows that although SEQUITUR beats *gzip* on average, it is worse on 8 out of the 14 individual files. The encoding presented up to this point is also described in Nevill-Manning (1994b).

6.1.4 Sending the sequence incrementally

One final problem remains: how to send the sequence incrementally. Incrementality is paramount in on-line situations such as the compression of communications traffic. The grammar is formed incrementally, so the material for transmission is available. The problem with incremental transmission is one that has been discussed briefly in Chapter 3: at many points in the sequence, it is possible that a longer match will result when subsequent symbols are seen. For example, Figure 3.11e shows a sequence that builds up repetitions from right to left, so that rules are only formed when the final symbol in the repetition appears. It would be wasteful to transmit the *wxy* of the last repetition, in case the last symbol was *z*, and the whole subsequence could be replaced by a pointer to *wB*. Unfortunately, it is possible to construct a situation, such as the one in Figure 3.11e, in which transmission could be postponed for an arbitrarily long period. Of course, such sequences are unlikely to occur in practice, so in most cases points will occur regularly when the sequence can be transmitted without danger of wasting symbols. Indeed, from the discussion of Figure 3.11e in Section 3.4, the longest postponement of transmission is $O(\sqrt{n})$ symbols, where n is the total number of symbols seen so far. The problem is in detecting situations in which a match can be ruled out.

Let α and β stand for the second to last and last symbols in rule S respectively. If β only occurs once in the grammar, it can be safely transmitted, because it cannot match any existing sequence in the grammar. Similarly, if the only other places that β occurs are at the end of rules, so that it does not appear as the left symbol in any digrams, no digram consisting of β and the next symbol to appear can match. Apart from these two situations (which are exceptional), it is impossible to determine when β can be safely transmitted: whatever follows β elsewhere in the grammar might occur next.

Therefore, a decision can only be made about transmitting a symbol when it is the second-last symbol in rule *S*. If β is terminal, $\alpha\beta$ evidently does not occur elsewhere in the grammar, or it would have been replaced by a non-terminal. However, it is necessary to check that β is not the start of a non-terminal that follows α elsewhere in the grammar. This can be checked by examining the prefixes of all non-terminals that follow the other occurrences of α. If none of the non-terminals start with β, it is safe to transmit α. For example, in Figure 3.11e, when the final *xy* has been seen, *x* is followed by *B* elsewhere in the grammar, and rule *B* begins with *y*. This indicates that *y* could be the beginning of rule *B*, and so sending *x* may eliminate the possibility of simply sending a pointer later on.

There is one further constraint on when a symbol can start eventually become part of a longer rule. Consider the sequence *AAa*, where $A \rightarrow ab$. The rule just described indicates that the second *A* cannot be transmitted, because elsewhere in the grammar *A* is followed by a non-terminal that starts with *a*. However, extending *A* in this way would mean that rules overlap, which is prohibited. Because the second *A* cannot be extended, it can safely be sent.

The criterion for deciding when another chunk of rule *S* can be transmitted, where *S* ends with $\alpha\beta$ is therefore: β is terminal, and no instance of α is followed by a non-terminal whose contents start with β, unless that non-terminal is another α. This is guaranteed to happen within an interval of length $O(\sqrt{n})$, where *n* is the length of the sequence. In *book1*, encoding can be performed on average every 14 symbols: the minimum interval is one, which is guaranteed to be the case after the first symbol has been received, and the maximum interval is 161.

6.2 Compression performance

Section 2.4 discussed the established methodology for evaluating compression schemes. The most popular compression schemes are those that perform well on a range of data. For this reason, the Calgary corpus includes a wide range of sequence types, listed in Table 6.2. The files include fiction and non-fiction text, an image, a transcript of computer interaction, binary geophysical data, computer object files

name	description	size	compress	gzip	SEQUITUR	ppm
bib	bibliography	111261	3.35	2.51	2.48	2.12
book1	fiction book	768771	3.46	3.25	2.82	2.52
book2	non-fiction book	610856	3.28	2.70	2.46	2.28
geo	geophysical data	102400	6.08	5.34	4.74	5.01
news	USENET	377109	3.86	3.06	2.85	2.77
obj1	object code	21504	5.23	3.84	3.88	3.68
obj2	object code	246814	4.17	2.63	2.68	2.59
paper1	technical paper	53161	3.77	2.79	2.89	2.48
paper2	technical paper	82199	3.52	2.89	2.87	2.46
pic	bilevel image	513216	0.97	0.82	0.90	0.98
progc	C program	39611	3.87	2.68	2.83	2.49
progl	Lisp program	71646	3.03	1.80	1.95	1.87
progp	Pascal program	49379	3.11	1.81	1.87	1.82
trans	shell transcript	93695	3.27	1.61	1.69	1.75
average			3.64	2.69	2.64	2.49
Bible	King James version	4047392	2.77	2.32	1.84	1.92

Table 6.2 Performance of various compression schemes (bits per character)

and source files. To avoid biasing evaluation toward performance on the larger files, comparison is based on the average compression rate rather than the total compressed size of the corpus. Rates are quoted in bits per symbol, where a symbol is an eight-bit byte in the original file. Typical rates differ between different types of file: figures for text are similar to each other, while the rate for the picture is lower and the rate for the geophysical data higher.

The results for four compression methods, *compress*, *gzip*, PPM and SEQUITUR, are shown in Table 6.2. The best figures for each row are highlighted. Overall, SEQUITUR outperforms all the schemes other than PPM, which is 6% better. It beats PPM on the geophysical data, the picture, and the transcript. The reason for this is that although PPM is good at capturing subtle probabilistic relationships between symbols that typically appear in highly variable sequences such as text, SEQUITUR excels at capturing exact, long repetitions that occur in highly structured files. While SEQUITUR does not perform as well as PPM on text such as *book1* and *book2*, it outperforms PPM on longer text such as the King James version of the Bible, shown in the last row of Table 6.2.

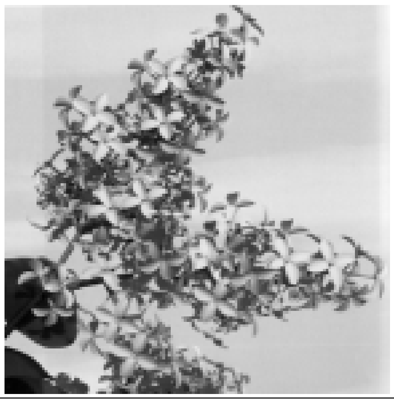
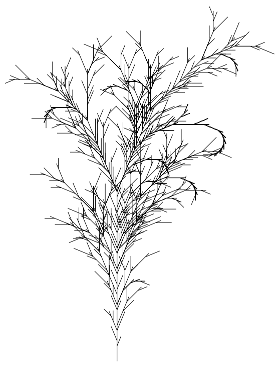
graphical interpretation				
sequence length	908 670		140 842	
PPM	37 037	(4.0%)	7436	(5.2%)
gzip	8 395	(0.9%)	2697	(1.9%)
SEQUITUR textual	3 179	(0.3%)	8572	(6.0%)
SEQUITUR	658	(0.07%)	1732	(1.2%)

Table 6.3 Compression of context-sensitive L-system output (after Prusinkiewicz and Lindenmayer, 1990)

For highly structured sequences such as the output from L-systems, SEQUITUR performs very well indeed. Table 6.3 shows compression performance of various schemes on the output of context-sensitive, stochastic L-systems. PPM performs the worst, because it fails to capitalise on the very long repetitions that exist. G*zip* performs two to four times better than PPM. Surprisingly, the textual version of SEQUITUR's grammar is 13 times smaller than PPM's output and 3 times smaller than *gzip*'s output on one sequence, and only slightly worse than PPM on the other. The encoded version of SEQUITUR's grammar is 1300 times smaller than the original in one case, and 80 times smaller in the other.

6.3 Compression and grammar size

In the introduction to Chapter 4, it was claimed that the number of symbols in the grammar is a good measure of the grammar's information content. It is now possible to substantiate this claim. The information content of a sequence can be measured by its Kolmogorov complexity (Li and Vitanyi, 1993), which is the size of the smallest universal Turing machine capable of reproducing the sequence. Since this quantity is not computable, it is necessary to devise other metrics that approximate it. One way of performing this approximation is to allow competition between

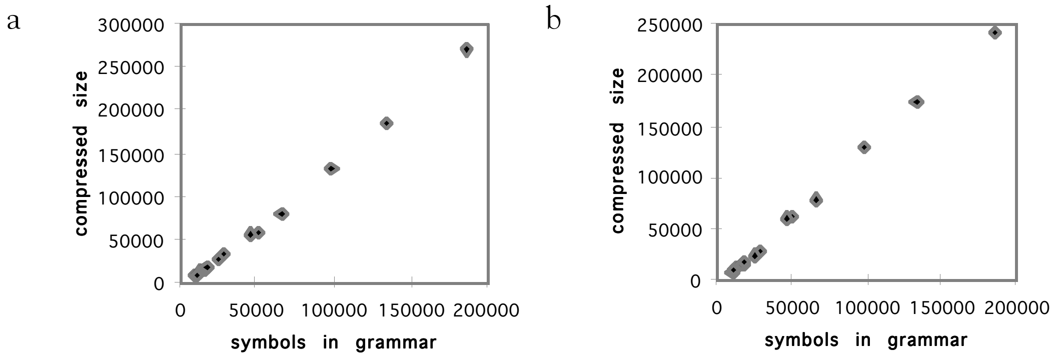


Figure 6.2 Encoded size versus number of symbols in grammar
(a) Size of compressed sequence from SEQUITUR (correlation 0.9989)
(b) Size of compressed sequence from PPM (correlation 0.9993)

different encoding schemes on a standard corpus and pick the best performing technique as the most likely approximation to the Kolmogorov complexity. This is exactly what occurs in the data compression community. The best approximation currently available, therefore, is the size of PPM's output given a sequence. As SEQUITUR performs only 6% worse than PPM on average, it is also a reasonable indicator of information content.

The goal of this section is to show that comparing grammars based on the number of symbols they contain is equivalent to comparing them based on the amount of information that they contain. This can be achieved by showing that information, which we will approximate by compressed size, is an approximately linear function of the number of symbols in a grammar. Figure 6.2 shows graphs relating the number of symbols in a grammar to the size of their compressed representation. The horizontal axis is the number of symbols in the grammar—the independent variable. The vertical axis is the compressed file size—the dependent variable. Figure 6.2a indicates the size when compressed using SEQUITUR. Each point represents one of the fourteen files in the Calgary corpus. The points appear to lie along a line. Linear regression bears this out: the correlation between the two quantities is 0.9989. The least-squares line has with a slope of 1.47 bytes (about 12 bits) per symbol, which indicates the encoded size of an average symbol in a grammar. Figure 6.2b shows an analogous graph for the file sizes when compressed using PPM. The horizontal axis remains the number of symbols in SEQUITUR's grammar—PPM is invoked because of its superior compression performance. The relationship is similarly linear, with an

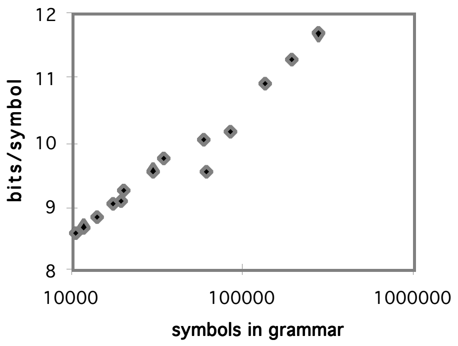


Figure 6.3 Bits per symbol in grammar as a function of grammar size

even higher correlation of 0.9993. The slope is lower—1.34 bytes (11 bits) per symbol—which follows from PPM's superior compression.

A closer analysis of the data show a slight non-linear trend: the slope of the line increases for larger grammars. That is, plotting the average number of bits per symbol for each file against the number of symbols in the file yields graph that increases. Making the scale for the horizontal axis logarithmic produces the graph in Figure 6.3. The points lie roughly on a line—the two quantities have a correlation coefficient of 0.974. This indicates that the number of bits required to code a symbol is proportional to the logarithm of the number of symbols, which makes sense: symbol size is related to alphabet size, and because the alphabet includes non-terminals, the alphabet is larger for larger grammars. This is borne out by the higher correlation between symbol size and the logarithm of the number of rules in a grammar: it is higher, at 0.985. The least-squares regression line for Figure 6.2a gives an average error rate of 9% when used to predict the size of SEQUITUR's output, due mainly to errors on the very small files. When a line is fitted to Figure 6.3 the average error falls to 1%. The size of a compressed file is therefore $n\log(0.68n - 0.48)$, where n is the size of the grammar. Because comparisons are usually made between grammars whose sizes are well within an order of magnitude, the logarithmic effect is minimal, and linearity can be assumed.

6.4 Compressing semi-structured text

Section 5.4 discussed the automatic detection of structure in the GEDCOM genealogical database. Structure detection has twin benefits: one is the discovery of patterns, a process of learning, and the other is efficiency: the ability to express the

	scheme	dictionary	code indexes	word indexes	total size (Mb)	compression
original	–				9.18	100.0%
byte-oriented	*compress*				2.55	27.8%
	gzip				1.77	19.3%
	PPM				1.42	15.5%
word-oriented	WORD-0	–	–	–	3.20	34.8%
	WORD-1	–	–	–	2.21	24.1%
	MG	0.14	–	2.87	3.01	32.8%
SEQUITUR		0.11	–	1.07	1.18	12.9%
SEQUITUR with generalisation of	codes	0.11	0.40	0.60	1.11	12.1%
	dates	0.11	0.64	0.31	1.06	11.5%
	gender	0.11	0.64	0.30	1.05	11.4%
	names	0.11	0.76	0.17	1.04	11.3%

Table 6.4 Compression rates of various schemes on the genealogical data

data more concisely. This section applies the encoding scheme developed in this chapter to compressing the database. SEQUITUR is compared to other state of the art compression schemes on the 9 MB sample used in Section 5.4.

The LDS genealogical databases are currently stored in a system called AIM which performs special-purpose compression designed specifically for the GEDCOM format. This begins by assembling a dictionary of the most frequently occurring one-, two- and three-symbol fragments. Any fragments that do not occur at least three times are dropped. This dictionary then is encoded with the 63 most frequent entries in one byte and the remaining entries in two bytes. In addition to this there are special encodings for dates and other types. This first-level approach provides about 40% compression, which is considerably worse than any of the other schemes we tested. However, the files so produced are then compressed using the standard STACKER compression product. This provides a further 40% compression, for a total compression in AIM of 16%.

Table 6.4 shows the results of several compression programs on the 9 MB sample. The standard compression programs (except MG) do not support random access to records of the database, so they are not suitable for use in practice because random access is always a *sine qua non* for information collections of this size.

The first block of Table 6.4 summarises the performance of the byte-oriented schemes discussed in Section 6.2. UNIX *compress* provides a benchmark lower bound on the possible compression, while *gzip* achieves substantially better compression. PPM performs extremely well on the data, giving a compression rate of over six to one. For all these schemes, compression rates are about twice as great as they are on *book1* from the Calgary corpus, which indicates the high regularity of this database relative to normal English text.

The next block of Table 6.4 summarises the performance of some word-oriented compression schemes. These schemes split the input into an alternating sequence of words and non-words—the latter comprising white space and punctuation. WORD uses a Markov model that predicts words based on the previous word and non-words based on the previous non-word, resorting to character-level coding whenever a new word or non-word is encountered (Moffat, 1987). We used both a zero-order context (WORD-0) and a first-order one (WORD-1). MG is a designed for full-text retrieval and uses a semi-static zero-order word-based model, along with a separate dictionary (Witten *et al.*, 1994). In this scheme, as in WORD-0, the code for a word is determined solely by its frequency, and does not depend on any preceding words. This proves rather ineffective on the genealogical database, indicating the importance of inter-word relationships. WORD-1 achieves a compression rate that falls between that of *compress* and *gzip*. The relatively poor performance of this scheme is rather surprising, indicating the importance of sequences of two or more words as well perhaps as the need to condition inter-word gaps on the preceding word and *vice versa*. None of these standard compression programs perform as well as the *ad hoc* scheme used in AIM, except, marginally, PPM.

As described in Section 5.4, the words were presented to SEQUITUR as if they were symbols drawn from a large alphabet. The encoding scheme was that described in Section 6.1. The dictionary was compressed in two stages: front coding followed by compression by PPM. Front coding (Gottlieb *et al.*, 1975) involves sorting the dictionary, and whenever an entry shares a prefix with the preceding entry, replacing the prefix by its length. For example, the word *baptized* would be encoded as *7d* if it were preceded by *baptize*, since the two have a prefix of length 7 in common. A more principled dictionary encoding was also implemented, but failed to outperform this simple approach.

The grammar, when encoded using the method described above, was 1.07 Mb in size. The dictionary compressed to 0.11 Mb, giving a total size for the whole text of 1.18 Mb, as recorded at the top of the bottom block of Table 6.4. This represents almost eight to one compression, some 20% improvement over the nearest rival, PPM.

Generalising the codes, as described in Section 5.4, halves the number of rules in the grammar, the length of the top-level rule, and the total number of symbols. The compressed size of the grammar falls from 1.07 Mb to 0.60 Mb. However, the codes need to be encoded separately. To do this, a dictionary of codes is constructed and compressed in the same way as the dictionary for the main text. Each code can be transmitted in a number of bits given by the logarithm of the number of entries in the code dictionary. The total size of the two files specifying the individual and family codes in this way is 0.40 Mb, bringing the total for the word indexes to 1.0 Mb, a 7% reduction over the version with codes contained in the text. Including the dictionaries gives a total of 1.11 Mb to recreate the original file.

Separating the dictionaries represents the use of some domain knowledge to aid compression, so comparisons with general-purpose compression schemes is unfair. For this reason, PPM was applied to the same parts as SEQUITUR, to determine what real advantage SEQUITUR provides. PPM was first applied in a byte-oriented manner to the sequence of word indexes. It compressed these to 1.07 Mb, far worse than SEQUITUR's 0.60 Mb. In an attempt to improve the result, PPM was run on the original file with generic tokens for codes, yielding a file size of 0.85 Mb—still much worse than SEQUITUR's result. Note that this approach does outperform running PPM on the unmodified file. Finally, the WORD-1 scheme was applied to the sequence of generalised codes, but the result was worse still.

Table 5.1 ranks possible generalisations in order of their usefulness. Predictions 1, 2, 3 and 6 encourage generalisation of codes, which has been performed. Predictions 5, 7 and 9 indicate that dates should be generalised. Extracting dates in a way analogous to the extraction of codes from the main text reduces the grammar from 0.60 Mb to 0.31 Mb, and adds 0.24 Mb to specify the dates separately. This represents a net gain of 0.05 Mb, or 5%. Prediction 4 indicates that the *SEX* field can be generalised by replacing the two possible tags, F and M. Acting on this reduces the size by a further 1%. Finally, prediction 8 indicates that names should be generalised, resulting in a final compressed size of 1.04 Mb, or a ratio of almost nine to one. The final block of Table 6.4 summarises these improvements. Another description of this application can be found in Nevill-Manning and Witten (1996).

6.5 Summary

The encoding scheme that this chapter has described transforms the SEQUITUR algorithm into a compression scheme. Its average performance over the Calgary corpus is greater than any other dictionary compression scheme, and on sequences such as long tracts of text, DNA sequences, and L-system output, it betters the state of the art technique, PPM. Compression is the most practical of applications—it allows efficient use to be made of storage and transmission resources. The next chapter evaluates the learning abilities of SEQUITUR.

7. Applications

The thesis claims in Chapter 1 that a variety of sources manifest their structure as regularities in their output. This chapter evaluates that hypothesis empirically by applying the techniques described so far to a range of sequences. Some demonstrations of structural inference have already been provided as motivation for techniques in previous chapters: they include L-system inference, data compression, inference of programs from an execution trace, and identification of structure in semi-structured text and computer programs. Here we present five more domains where the techniques work well.

Section 7.1 investigates the structure of a familiar sequence: human language. First, we show that SEQUITUR is capable of inferring structure from text in three different languages, and that accuracy improves if knowledge about textual structure is available. Next, its performance is compared to other inference techniques designed specifically for structure recognition in text. Finally, SEQUITUR is applied to a sequence of word classes to allow grammatical constructions to be inferred. Section 7.2 returns to the L-system domain discussed in previous chapters, and shows how SEQUITUR-K can be applied to speed graphical rendering by several orders of magnitude. The basic idea is to identify identical parts of a scene that can be rendered once and reused. The identical parts correspond to rules that SEQUITUR forms from the sequence of graphical commands. Section 7.3 examines structure in music, focussing on J.S. Bach's chorales, and identifies similar chorales, hierarchies of musical sequences, and common constructions such as imperfect and perfect cadences. Section 7.4 looks at the identification of interesting phrases in large textual corpora. The syntactic hierarchies that SEQUITUR builds correspond to conceptual hierarchies, which allow efficient access to the contents of a corpus by successive extension of a phrase to specialise a concept. Finally, Section 7.5 presents promising results in the identification of structure in macromolecular sequences such as DNA sequences and amino acid sequences. SEQUITUR compresses these sequences much better than other compression schemes, which indicates that it is recognising structure that other schemes cannot.

7.1 Natural language

Language is a familiar sequence—this very sentence is an example of a sequence that we must interpret. This section describes the application of SEQUITUR and its variants to linguistic sequences. We first consider, in Section 7.1.1, the performance of SEQUITUR on long texts in three different languages. Its success in recognising structure emphasises the even-handedness of the two constraints that drive the algorithm—there is no bias toward any particular language. In Section 7.1.2 we compare SEQUITUR to the MK10 system introduced in Section 2.5. For this purpose we use a reconstruction of MK10 and the data that Wolff (1975) used to test its operation. The two systems are also evaluated on natural English. Finally, Section 7.1.3 examines SEQUITUR's ability to infer grammatical constructs from sequences of word classes, rather than from the words themselves.

7.1.1 Structure identification in large text sequences

For the full capabilities of the structural detection techniques to emerge, a text must be large and consistent. SEQUITUR's performance improves as more of a sequence is observed, because more repetitions occur in a longer sequence. Furthermore, if the sequence is relatively homogeneous—written in a similar way with a consistent vocabulary throughout—more similarities will be recognised. Many examples of such text exist in electronic form, for example collections of articles from the Wall Street Journal, or encyclopaedic works such as *Le trésor de la langue française*. For these experiments the Bible was chosen, because in addition to being lengthy (over 750,000 words) and consistent (most versions are the work of a single team of translators who aim for consistency of style and vocabulary) it also exists in several languages. This allows SEQUITUR's language independence to be illustrated.

We were able to find electronic versions of the Bible in many languages, including Swedish, Italian, Spanish and Danish, but decided to use versions in English, French

language	characters	words	vocabulary
English	4 047 392	766 111	13 745
French	4 214 991	759 391	24 824
German	4 598 677	781 718	27 544

Table 7.1 Statistics on the text of the Bible in three languages

language	rules	symbols	rule S	most popular rules
English	84 000	576 000	399 000	and, the, of, to, not, that, of the
French	98 000	652 000	441 000	les, et, de, des, le, la, en, à, vous
German	106 000	764 000	542 000	und, die, der, sie, den, das

Table 7.2 Statistics for grammars inferred from the Bible

and German. Table 7.1 presents several characteristics of each sequence. All three sequences contain between four and five million characters. They contain a similar number of words: between 766,000 and 781,000, but their vocabulary sizes vary considerably more: from 14,000 for English to nearly double that for German.

Forming a hierarchy from the Bible takes several minutes, and the resulting grammar for the English version has 576,000 symbols and 84,000 rules. Table 7.2 summarises the statistics for all three languages. More rules are formed for the French and German versions, and the grammars are larger overall. The length of rule *S*, which provides an indication of the amount of the structure detected in a sequence, is smallest in the English version, indicating a greater amount of structure in the sequence. This is partly a consequence of the smaller vocabulary: a random sequence will contain more repetitions if its alphabet is smaller. The most popular rules, shown in the rightmost column of Table 7.2, correspond to common function words in the respective languages.

Figure 7.1 shows the hierarchies for Genesis 1:1. Each branch in the tree structure corresponds to a rule in the grammar. The topmost level of each tree represents the non-terminal that occurs in rule *S*. Spaces are made explicit as bullets. Figure 7.1a shows the hierarchy for the English version. At the top level, rule *S*, the verse is parsed into six subsequences: *In the*, *beginning*, *God*, *created*, *the heaven and the*, and *earth*. Four of the subsequences are words, and the other two are groups of words. *In the* is broken up, at the next level down, into *In* and *the*. The other phrase, *the heaven and the* is broken into *the heaven* and *and the*. At the next level, *the heaven* is incorrectly split into *the he* and *aven*. The words *beginning* and *created* consist of root words and affixes: *beginning* is split into *begin* and *ning*, while *created* is split into *creat* and *ed*. The root of *beginning* is *begin*, but the normal form of the suffix is *ing*, rather than *ning*. SEQUITUR has no way of learning the consonant doubling rule that English follows to create suffixes, so other occurrences of words ending in *ning* cause this rule to be formed. Similarly, whereas *create* is the present tense of the verb, *ed* is

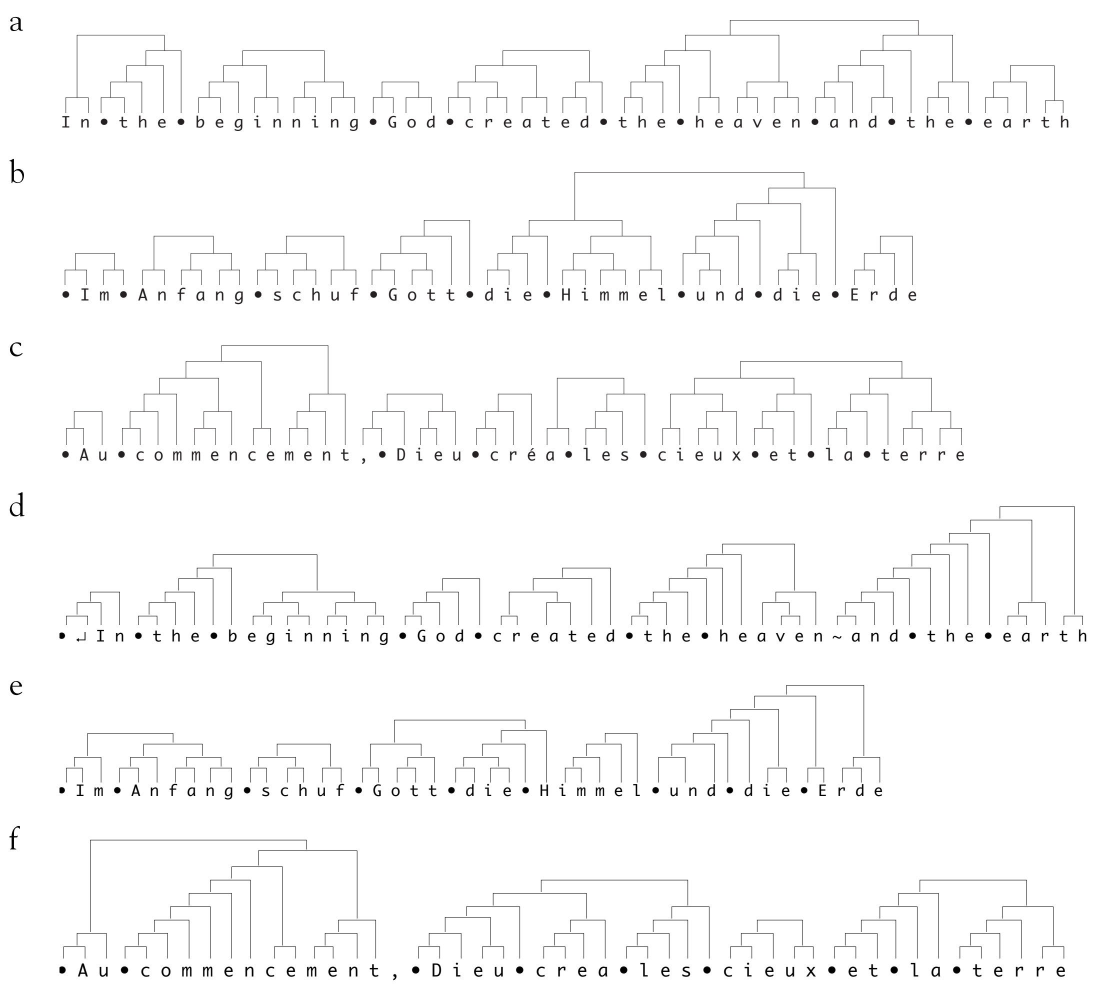

a
In•the•beginning•God•created•the•heaven•and•the•earth
b
•Im•Anfang•schuf•Gott•die•Himmel•und•die•Erde
c
•Au•commencement,•Dieu•créa•les•cieux•et•la•terre
d
•↵In•the•beginning•God•created•the•heaven~and•the•earth
e
•Im•Anfang•schuf•Gott•die•Himmel•und•die•Erde
f
•Au•commencement,•Dieu•crea•les•cieux•et•la•terre

Figure 7.1 Hierarchies for Genesis 1:1
produced by SEQUITUR: (a) English, (b) German and (c) French
produced by SEQUITUR-K: (c) English, (d) German and (e) French

usually the affix for past tense, so the division makes sense. For the most part, then, SEQUITUR segments English correctly.

SEQUITUR is language independent—the two constraints on the grammar are not designed to favour English. Figure 7.1b shows the German version of the same verse—incidentally somewhat more compact than the English sentence. The verse is split into *Im*, *Anfang*, *shuf*, *Gott*, *die Himmel und die*, and *Erde*. The word group is split into *die Himmel* and *und die*. *die Himmel* is split correctly into words, as is *und die*. The word *schuf* is split into *sch* and *uf*, because of the common *sch* consonant group in German.

The French version is illustrated in Figure 7.1c. The verse is split into *Au*, *commencement*, *Dieu*, *cré*, *a les*, *cieux et la terre*.[5] The phrase *cieux et la terre* is split into *cieux et* and *la terre*. The incorrect splitting of *créa* is due to greedy parsing, and along with the split of *la terre* into *la t* and *erre* will be corrected by the techniques described in Chapter 4, as discussed in the next subsection.

7.1.2 Domain knowledge and reparsing

Now consider the improvements that can be made by using the two variants discussed in Chapter 4, SEQUITUR-K and SEQUITUR-R, which make use of domain knowledge and perform reparsing respectively. To employ SEQUITUR-K, constraints must be formulated to guide the formation of rules, analogous to the bracketing constraints developed for text. The most prominent aspect of written language is the use of spaces, and adding a constraint that forces rules to respect word boundaries should improve SEQUITUR's grammar.

This means that a rule should not be formed that begins in the middle of one word and ends in the middle of another. A rule can only cross a word boundary if it contains an integral word, i.e. if it begins with a space. This is asymmetric: it requires a multi-word rule to contain a space on the left of a word but does not insist on one at the right. This is because words are usually separated by a single space, and requiring a space at both ends of a multi-word rule would create a conflict between adjacent rules. Furthermore, rules are created from left to right, and requiring a space at both ends of a rule would often mean that no rule could be created. The constraint that was added was: a rule starting with a space cannot be the second symbol in a digram unless the first symbol also starts with a space.

SEQUITUR-K with this constraint produces the hierarchies in Figures 7.1d, 7.1e and 7.1f. These rules are less appealing: much of the hierarchy consists of rules building up from left to right, rather than isolating parts of words. For example, in Figure 7.1d, the word *created* is split into *creat*, but the *e* and *d* are added on in two further rules, rather than being a rule on their own. The phrase *und die* lacks a component rule that just covers *die* in Figure 7.1e. The French version in Figure 7.1f fares somewhat better: the word *créa* is split correctly from the following *les*.

5 Accented letters such as à, á, â and ä are represented as a single symbol in the French and German sequences.

In order to evaluate the success of SEQUITUR-K more objectively, it is necessary to develop a metric to compare the new grammar against the grammar formed by the unmodified SEQUITUR algorithm. The metric we propose is to check the spelling of the rules in the grammar, and to prefer the grammar with larger number of correctly spelled words in the expansions of the rules (apart from rule *S*). Rules contain both whole words and fragments of words. For example, Figure 7.1d contains the rules *the b* and *and th*. The fragments *b* and *th* are not valid words, so these rules fail to capture the word structure of the text. Some of this is inevitable: fragments are necessary to build up a hierarchy within words. However, in other cases, the fragments are avoidable. Some fragments, such as *eat* in *created* and *he* in *heaven* will be incorrectly identified as valid words, but for the comparative study performed here, this effect is small.

To perform a spelling check, an appropriate dictionary is required. Dictionaries for generic spell checking will penalise the grammars for any words unique to the source document, such as proper nouns. The best dictionary is the vocabulary of the source document itself. The methodology is therefore as follows: extract all of the unique words in the source document, then for every word fragment in each rule of a grammar, check that fragment against the vocabulary and output the proportion of correctly spelled words. The results of this experiment are summarised in Table 7.3.

Using the space delimiting constraint, there is an improvement in the proportion of correct words in the grammar rules: in English from 93% to 95%, in French from 90% to 94% and in German from 90% to 93%. These figures are notable for two reasons. First, the baseline accuracy for SEQUITUR is quite high—at least 90% in all

language	SEQUITUR modification	percent correct	total words	total rules	symbols in rule S	total symbols
English	none	93%	247 000	84 000	399 000	576 000
	knowledge	95%	300 000	84 000	392 000	570 000
	reparsing	90%	280 000	87 000	449 000	640 000
French	none	90%	272 000	98 000	441 000	652 000
	knowledge	94%	320 000	93 000	416 000	620 000
	reparsing	86%	299 000	102 000	506 000	737 000
German	none	90%	241 000	106 000	542 000	764 000
	knowledge	93%	289 000	103 000	530 000	745 000
	reparsing	90%	268 000	110 000	608 000	849 000

Table 7.3 Application of domain knowledge and reparsing to text

cases. This indicates that the algorithm works well for detecting word boundaries. Second, the figures are relatively consistent over all the languages, demonstrating SEQUITUR's lack of bias towards particular sequences, and the similarity of the linguistic structure of the languages.

When domain knowledge is employed, not only does the proportion of correct words increase, but the size of the grammars decreases—by 1.2% for English, 4.9% for French and 2.4% for German. This is consistent with Occam's razor: a better explanation is also a smaller one. Counting the number of symbols in rule S alone indicates the amount of structure that has been extracted from the sequence, regardless of the size of the rule set. Here the English grammar improves by 1.8%, the French by 5.7% and the German by 2.2%. In English and French, the reduction in the length of rule S is even greater than the overall reduction in grammar size.

The number of rules decreases for French and German, and stays the same for English. The average rule length rises from 2.11 to 2.12 for the English grammar, rises from 2.15 to 2.19 for the French grammar, and stays constant at 2.09 for the German grammar. The number of words covered by the rules rises substantially for all three languages: 21% for English, 18% for French, and 20% for German. This indicates that the rule hierarchy covers the text with more efficiency: fewer rules, which are only slightly longer, cover many more words.

The reparsing mechanism of SEQUITUR-R results in poorer grammars in all respects. The reparsing rows of Table 7.3 show that the spelling check measure decreases for English and French, and remains the same for German. Furthermore, all the grammars expand when reparsing is used. This poor performance underscores the weakness of reparsing in sequences that are not highly structured. For example, reparsing should correct the mis-parsing of *the heaven* into *the he* and *aven*. However, for this to happen, the rule covering *aven* must be preceded in every occasion by *he*. In text, the variability of words means that this will very rarely be the case.

7.1.3 Segmentation of language

The process, described in the previous subsection, of identifying meaningful units in a linguistic sequence is not merely a peculiar side-effect of a sequence modelling technique—it is an active area of investigation in linguistics. People's ability to partition a stream of sound into segments is a phenomenon that has been recognised and studied by many linguists. In this subsection, we compare the generic techniques described in this thesis with a system built expressly to account for the cognitive processes involved in human segmentation.

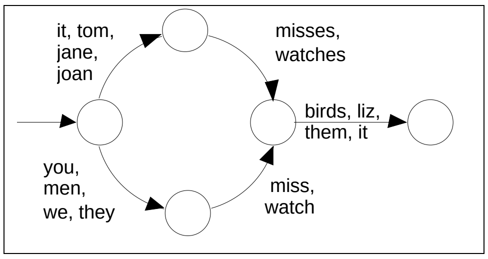
The universal phenomenon of segmentation into words was observed and recorded in the early part of this century. In arguing for the psychological plausibility of words, Edward Sapir (1921), based on his study of native American languages, said that:

> 'Linguistic experience, both as expressed in standardised, written form and as tested in daily usage, indicates overwhelmingly that there is not, as a rule, the slightest difficulty in bringing the word to consciousness as a psychological reality. No more convincing test could be desired than this, that the naive Indian [*sic*], quite unaccustomed to the concept of the written word, has nevertheless no serious difficulty in dictating a text to a linguistic student word by word; he tends, of course, to run his words together as in actual speech, but if he is called to a halt and is made to understand what is desired, he can readily isolate the words as such, repeating them as units. He regularly refuses, on the other hand, to isolate the radical or grammatical element, on the ground that it "makes no sense."'

In an attempt to determine the basis for this division into words, Hayes and Clark (1970) performed an experiment involving an artificially produced speech analogue containing repeated 'words' without any intervening pauses. Their theory was that words are distinctive because the strength of association between phonemes within words is greater than the association between word boundaries. Human subjects were able to distinguish the words without cues such as intonation and pauses, validating their theory. The techniques described in this thesis are able to make the same distinction, to delimit words from a stream of symbols in which no explicit delimiters are present.

Wolff (1975) proposed an algorithm, MK10, for performing this segmentation on text, and suggested that its success made it a candidate for the psychological process that gives rise to the ability in humans. His technique either required many passes over the same text, or threw away much information as it moved through a long stream of text. Although this is plausible in the context of explaining mechanisms for human acquisition, it is wasteful when perfect memory is available (as it is in a computer system). SEQUITUR makes much more efficient use of text without making several passes, and conserves statistics after inferences are made. This section explores what SEQUITUR learns from various texts.

a



b

```
youwatchthemyoumissbi
rdsitmissesliztomwatche
sthemwemissityoumissli
zjoanwatchesliztheymiss
itwemissthemtommisses
ittomwatchesbirdsjoanw
atchesliztheymissbirdsjo
anmi...
```

Figure 7.2 Reconstruction of Wolff's (1975) segmentation experiment
(a) automaton for generating test data
(b) an example of the output of (a)

The first experiment that Wolff performed using MK10 was to detect word and sentence boundaries in a sequence produced by the automaton in Figure 7.2a. The comma-separated lists are the various symbols on which the transition can occur. Wolff produced a 48,000 character sample by randomly traversing the automaton, which results in a sequence like that in Figure 7.2b. The object of his experiment was to show that word and sentence boundaries could be inferred computationally by virtue of the greater co-occurrence of symbols within words than symbols at the boundaries of words.

MK10 forms new elements (rules, in SEQUITUR's terminology) when it has seen a pair of elements ten times. Once this occurs, it replaces all pairs with the new element and starts again from the beginning of the text. It forms the first element, *mi*, after seeing 126 characters, and has formed all the words after seeing 1579 characters. After 3488 characters, it forms only complete sentences. The advantage of Wolff's threshold of ten repetitions is that it avoids any coincidental pairs—it only forms words or parts of words in the first phase, then only parts of sentences, then only whole sentences. There are no elements that cover parts of two words, or parts of two sentences. The disadvantages of this approach are the arbitrariness of the threshold of ten repetitions, and the slow rate of learning. SEQUITUR is somewhat analogous to MK10 with a threshold of two: repetitions are made into rules as soon as they occur.

An experiment was constructed by replicating Wolff's automaton and producing another corpus of 48,000 characters. Because the sentences are produced randomly, they are not identical to Wolff's corpus. For the purposes of comparison and experimentation, a version of MK10 was written based on the description in the paper. It reproduced Wolff's published results closely enough to indicate that the implementations were similar.[6] Wolff's version created its first full sentence at entry 77 after 3488 characters, whereas the reconstruction reached this point at entry 76 after 3128 characters.

After 1579 characters, when MK10 has formed all of the words, SEQUITUR-R has also formed all of the words, along with many other rules. When the rules are ranked by their frequency of use in the grammar, thirteen of the top sixteen rules are integral words. As well as the words, SEQUITUR has also formed nine complete sentences, whereas it will take MK10 over twice as many symbols to form the first sentence, and more than three times as many symbols to form nine sentences. Of course, this comes at a cost: in SEQUITUR's grammar, there are 26 rules that cross sentence boundaries out of the 80 rules in total. At this point, there is not yet sufficient evidence for SEQUITUR to shift the boundaries of these rules to conform to sentence boundaries.

When MK10 forms its first sentence at 3488 characters, SEQUITUR has formed 23 sentences. It has also formed 76 rules that cross sentence boundaries. Disregarding the rules that only occur twice, which is somewhat analogous to increasing Wolff's repetition threshold, SEQUITUR has formed fourteen sentences and nineteen spurious rules. It seems that for this example SEQUITUR outperforms MK10 in the speed of acquiring phrases, but is much poorer at distinguishing good phrases from bad.

MK10's threshold of ten merits investigation. Raising the threshold to twenty results in the same accuracy as a threshold of ten in terms of distinguishing the correct entries from spurious ones. However, the technique requires about twice as many characters, 2785 rather than 1664, to identify all the words. It takes 24025 rather than 12890 characters to identify all of the sentences. Reducing the threshold to five increases the learning rate for the words: all the words are identified after 825 rather than 1664 characters. However, the accuracy goes down. Although all sentences are identified, 64 non-sentences are also recognised, and it takes until character 25220 to identify all the sentences.

6 Written two decades ago, the original program used a complicated trie-like structure and merited a page of explanation, while the 1995 reconstruction took two hours to write and consists of forty lines of PERL code.

Reducing the threshold still further, to two, makes MK10 similar to SEQUITUR in this respect—the digram uniqueness constraint means that rules are formed when a digram occurs twice. MK10 now identifies all the words after 381 characters, but includes 138 non-sentences in the dictionary. SEQUITUR fares worse in terms of accuracy: it forms 905 rules, of which 653 are non-sentences. SEQUITUR-R performs better, reducing the number of non-sentences to 453. The reason for this difference in accuracy is the way in which MK10 parses the input sequence. Whereas SEQUITUR forms a rule as soon as a duplicate digram appears, MK10 first tries to parse the sequence relative to the current entries in the dictionary. For example, consider the sequence *it misses liz tom watches them … joan watches liz they miss it* produced by the automaton in Figure 7.2a (spaces have been added for clarity). Assuming that the entry *the* is already in the dictionary, MK10 will parse the word *they* in the last sentence into *the* and *y*. SEQUITUR, on the other hand, will form a new rule using the non-terminal for *liz* and appending the *t* from *they*. It will do this before the word *they* appears, which would have matched the earlier *the* in *them*. This is an issue of greedy parsing. SEQUITUR could emulate MK10's look-ahead parsing at the expense of incrementality: waiting for a possible match to an existing rule means that the sequence is partially unparsed, and if a matching rule does not appear, the unparsed portion must be reprocessed to form new rules. This is a feasible approach, and will be advantageous with structured sources where rules formed initially are guaranteed to be useful throughout. In this case, there are only sixteen different words, so making entries for them at the start and parsing the whole sequence using them produces a good parse. On real text, however, this advantage is reduced, as discussed below.

Because MK10 performs a slightly less greedy parsing than SEQUITUR, it is interesting to investigate its performance on the L-system sequences. On the example in Figure 4.3, after 4 derivation steps, MK10 has 44 elements in its dictionary, which describe the original sequence in 16 symbols. Without reparsing, SEQUITUR produces 32 rules and 21 symbols in rule *S*. With reparsing, SEQUITUR produces 11 rules and 6 symbols in rule *S*. It seems that MK10 is slightly better than SEQUITUR without reparsing, but worse than SEQUITUR-R.

There is, however, something troubling about the automaton. MK10 is based on the theory that there is more variability across sentence boundaries than within sentences. The automaton is perfectly designed to produce a stream that has these properties. The key is that for each path through the automaton there are only two verbs. This means that there are only 16 possible subject-verb combinations, and 16 verb-object combinations, whereas there are 32 object-subject cross-boundary combinations. It is hardly surprising that MK10 performs well under these conditions. Adding two more verbs, say *like* and *visit*, means that each pair—subject-verb, verb-object, and object-subject—have the same number of possibilities. This produces 128 valid sentences.

Now MK10, with the default threshold of ten, identifies 65 complete sentences amongst 209 spurious entries. A threshold of twenty produces 15 valid sentences out of 90 entries; a threshold of 5 produces 28 valid sentences out of 232 entries; and a threshold of two identifies all 128 sentences out of 585 entries. Evidently, the new automaton removes the cues by which MK10 identifies sentences. SEQUITUR identifies 36 sentences out of 653 rules, and with reparsing this rises to 60 out of 774 rules.

Moving from the artificial sequence to a real text, running both SEQUITUR and MK10 on the first 100,000 characters of *book1* results in a set of phrases inferred by each technique. With the original threshold of ten, MK10 identifies only 1021 entries, of which 52% are correct words. SEQUITUR forms 7000 rules, while MK10 with a threshold of two produces 7443 entries. Checking the phrases against the vocabulary of the text shows that 64% of the fragments are words for both techniques. This indicates that the techniques have comparable performance on actual text. Real, variable text favours a lower threshold for MK10.

7.1.4 Sentence structure

A hierarchical grammar inferred from natural language text describes the lexical structure of the text: how words and phrases are composed. Linguists, however, are often interested in the non-deterministic structure of language: the various combinations in which words can be used. These structures are described in terms of word classes—collections of words that fulfil a similar grammatical role, such as nouns, verbs, adjectives, adverbs and prepositions. If a text is expressed not as a sequence of the words themselves but as a sequence of word classes to which the words belong, exact repetitions can be interpreted as grammatical structures. That is, by generalising the text before the application of a technique such as SEQUITUR, the resulting hierarchy describes the interaction of word classes rather than words, and therefore represents a description of the text at the level of a natural language grammar.

Wolff (1980) describes an experiment in which he took a text, in this case a small novel, manually assigned tags to each word, and presented the sequence to MK10. The result was a hierarchy of phrases for each sentence, which he compared to a manual parse performed by a linguist. He found that the automatically-generated hierarchy was better than random, and that about half the time it parsed the sequence correctly. However, many of the 'correct' parses were simply *determiner-noun*, and *adjective-noun* combinations.

A similar experiment was performed using SEQUITUR on a large corpus, the London-Oslo/Bergen (LOB) corpus (Johansson *et al.*, 1978). The LOB corpus is a British English counterpart of the Brown Corpus (Francis and Kucera, 1979) and contains 500 text samples selected from texts printed in Great Britain in 1961. Each word in the corpus has been tagged with its part of speech. The tagging was performed using a combination of automatic techniques and manual post-processing. The corpus consists of 1.2 million words, and there are 140 different tags. Applying SEQUITUR to the sequence of tags—effectively an alphabet of 140 symbols—produces a grammar with 32,000 rules, 261,000 symbols in rule *S* and 325,000 symbols overall. It is difficult to evaluate a set of rules that represent grammatical structures in the same way as the rules representing words and phrases. The white space boundaries of a word are computationally simple to recognise, while recognising a correct grouping of word classes requires a parser for English. For this reason, one parse will be evaluated in detail.

Figure 7.3 shows a sentence from the LOB corpus: *Most Labour sentiment would still favour the abolition of the House of Lords*. Figure 7.4a shows a manual parse that breaks the sentence into two parts: a noun-phrase subject—*Most Labour sentiment*—and a verb phrase—*would still favour the abolition of the House of Lords*. The subject is subdivided into the determiner *Most* and the adjective-noun group *Labour sentiment*. The verb phrase is divided into a verb phrase *would still favour* and a noun phrase *the abolition of the House of Lords*. The smaller verb phrase is split into three words, as the adverb *still* is between the auxiliary *would* and the participle *favour*. The object is split into the article and noun, *the abolition*, and the adjectival phrase *of the House of Lords*. This is further split into the preposition *of*, the noun *the House* and the phrase *of Lords*.

Most	post-determiner	determiner
Labour	noun	noun
sentiment	noun	noun
would	modal auxiliary	auxiliary
still	adverb	adverb
favour	verb	verb
the	plural/singular article	article
abolition	noun	noun
of	preposition	preposition
the	plural/singular article	article
House	locative noun	noun
of	preposition	preposition
Lords	plural titular noun	noun

Figure 7.3 A sentence from the London-Oslo/Bergen corpus with assigned word classes and generalised word classes

The centre column of Figure 7.3 shows the tags assigned to each of the words in the sentence. When given the sequence of 1.2 million such tags, SEQUITUR forms rules that cross sentence boundaries, and such hierarchies have no chance of being correct parses for a sentence. For this reason, background knowledge similar to the word boundary constraint used for character-based analysis of text is used, in this case using the end of sentence markers in an analogous way to spaces.

Figure 7.4b shows the hierarchy that SEQUITUR generates for this segment of the sequence, showing the actual words rather than the tags. The determiner and the adjective *Most Labour* are grouped together, as are the noun and auxiliary. Next, the phrase *still favour the abolition of* is grouped together, and at the next level splits into the verb part and the object part. The final segment is *the House of Lords*, which is then split into *the*, and *House of Lords*. This parse is different from the manual parse, but still associates most of the verb part together, and groups this with the main part of the object, while leaving the noun phrase *the House of Lords* separate.

The hierarchy is shallow compared to other sequences studied in this thesis. This seems to be partly due to the large number of very specific tags that make up the sequence. There are 140 unique tags in the sequence, whereas text contains between 60 and 70 unique symbols, counting numbers and punctuation. In order to produce deeper hierarchies, the alphabet was reduced by combining all articles and determiners into one class, all verb types into another class, all nouns to another

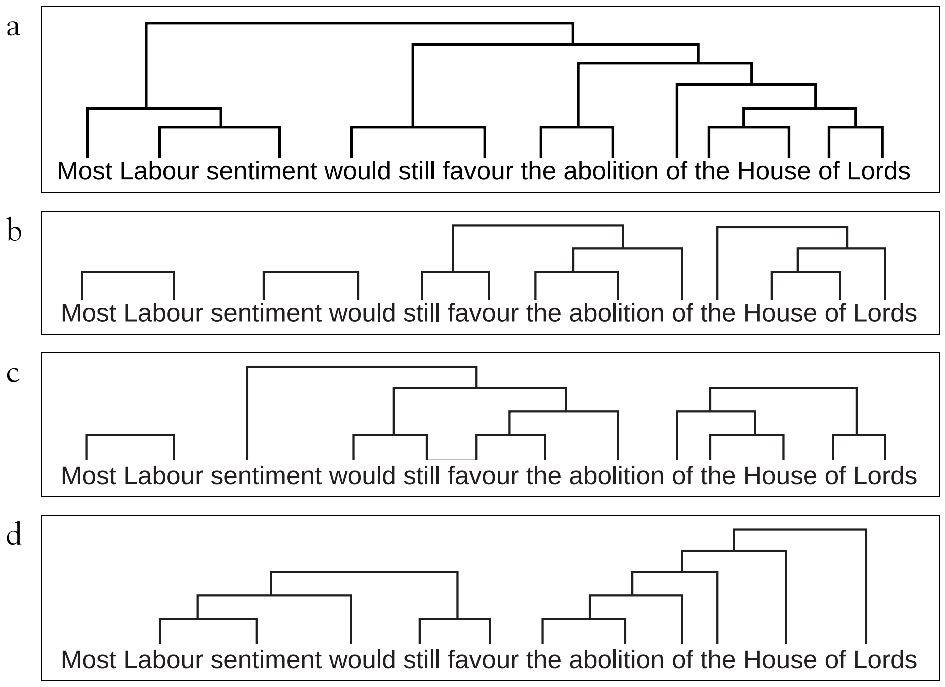

a
Most Labour sentiment would still favour the abolition of the House of Lords
b
Most Labour sentiment would still favour the abolition of the House of Lords
c
Most Labour sentiment would still favour the abolition of the House of Lords
d
Most Labour sentiment would still favour the abolition of the House of Lords

Figure 7.4 Parses of the sentence from Figure 7.3
(a) Manual parsing
(b) SEQUITUR's hierarchy based on original tags
(c) SEQUITUR's hierarchy based on generalised tags
(d) SEQUITUR's hierarchy based on generalised tags with reparsing

class, and so on. This reduces the number of tags to 18. The new tags for the original sentence are shown in the third column of Figure 7.3.

The hierarchy for the new sequence is shown in Figure 7.4c. At the top level, the adjectival phrase *of the House of Lords* is separated from the rest of the sentence, and is then broken into *of the House* and *of Lords*. The first part of the sentence is split into the adjectival part *Most Labour* and the subject-verb-object pattern *sentiment would still favour the abolition*. This indicates that this pattern is an important one, and the adjectival parts at the beginning and end are construed as appendages. Next the subject is separated from verb-object, and at the next level the auxiliary and adverb are separated from the rest. With reparsing, shown in Figure 7.4d, the subject-verb part is separated from the object, then the subject is grouped with the auxiliary and the adverb with the verb. The object is built up from left to right.

Overall, the parse is different from the manual parse, but maintains some plausibility, especially the core grouping of the subject-verb-object, with adjectives at the beginning and end. It is interesting to consider how justified, in the light of this kind of analysis, the manual parse is. However, this is beyond the scope of this thesis.

It is difficult to determine, without a manual parse for the entire 1.2 million word corpus, whether it would be more compressive than SEQUITUR's output. If, as some believe, human learning entails aspects of data compression, this would provide a metric with which to compare the manual and automatic parses. Wolff (1980) proposes:

> We should, perhaps, cease to look on language acquisition as a process of discovering a target grammar and should, instead, view it as a process of constructing an optimally efficient cognitive system. This revised view fits more naturally into a biological perspective and allows us to side-step certain logical problems which arise from the first view (Gold 1967…). [see Section 2.2] … If there is no target grammar but merely an evolutionary process of improving the efficiency of a cognitive system then this proof no longer presents a problem.

This is certainly an interesting proposal, and further reinforces the thrust of the thesis, that compression and understanding are strongly related.

7.2 Improving rendering performance

Section 4.3 describes the use of domain knowledge to form rules that do not violate semantics of the source—in particular, to constrain rules to nest brackets correctly when inferring an L-system. This constituted a useful constraint for ensuring concise grammars, but it has another important implication: it ensures that every rule has a computable graphical equivalent. This enables SEQUITUR to be used to reduce the complexity of rendering graphics based on L-systems.

Rendering realistic computer graphics of natural scenes can be very demanding in terms of computer time and storage. Each element (e.g. polygon) must be stored in memory, and must individually take part in calculations such as ray-intersection queries. One approach to reducing the time and space complexity of rendering is to find identical parts of a scene, render one of them, and reuse the result in each of the other cases. This process of processing one instance of a graphical form on behalf of several other identical forms is called *instantiation.* (Hart, 1992) For example, the tree in Figure 4.3c contains multiple instances of the same branch shape, as illustrated in Figure 7.5. Once one instance has been drawn, it can be translated and rotated to form another part of the tree.

A significant problem with instantiation is the identification of these identical forms: to efficiently find identical parts of a given scene. Where these natural scenes are based on the output from an L-system, SEQUITUR performs this task very effectively. The rules that SEQUITUR produces correspond directly to repeated graphical forms.

One objection to the use of SEQUITUR for finding phrases in L-system output is that if the original L-system is available, it should be possible to identify useful phrases by inspection of the L-system. This is certainly true for context-free L-systems—in fact we have shown in this case that SEQUITUR's phrases can be used to reconstruct the original L-system, demonstrating the equivalence between the L-system and the derived rules. However, most L-systems used to produce lifelike images are context-sensitive and stochastic, so the only way to determine what phrases will appear in the output of an L-system is to evaluate it. The phrases that SEQUITUR identifies

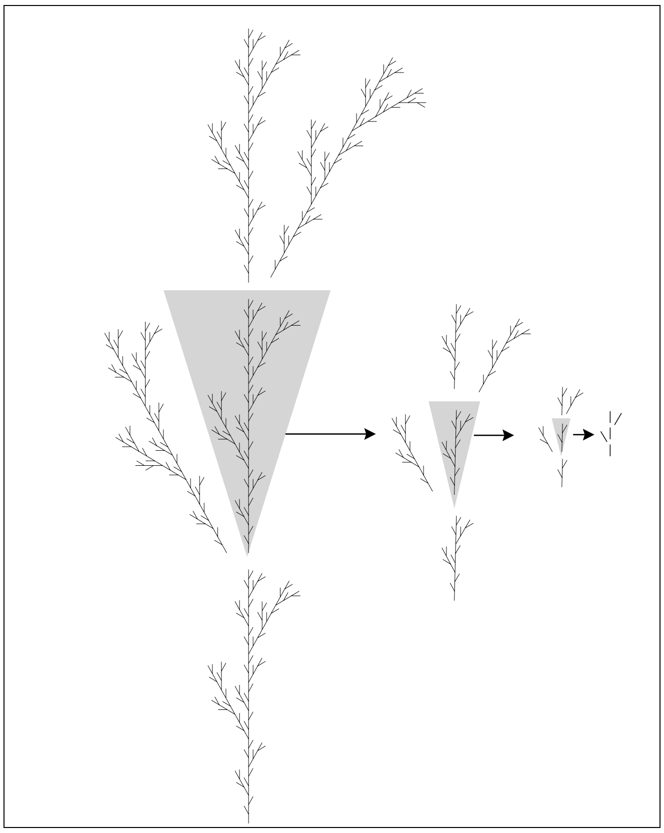


Figure 7.5 Identical parts of a tree

will include some of the simple rules from the L-system, but will also include many phrases whose occurrence is due to the randomness of the stochastic rules, and to the context-sensitivity of the rules. SEQUITUR's role, then, is to take a context-sensitive, stochastic L-system, and by analysis of its output, to produce an equivalent, context-free L-system that provides a hierarchy of instantiations. The context-free version will usually be much larger than the original L-system, but much smaller than the sequence of instructions.

To render a scene, SEQUITUR first produces a grammar from the sequence of graphical instructions. Next, each of the rules apart from rule *S* are rendered. Finally *S* is rendered, making use of the pre-rendered rules, to produce the entire scene. This two-stage approach is much more efficient than rendering the sequence itself, but does not take full advantage of the hierarchical nature of the grammar. Where several rules make use of a shorter rule, it is useful to render the shorter rule once, and re-use it in each of the longer rules. The idea is to render small components of the scene, and then use these pre-calculated components to build up larger structures. Spatially, the grammar corresponds to a hierarchy of forms, with small, simple objects combining to form larger, more complex ones. So this instantiation takes advantage not only of identical forms, but also of hierarchies of forms, to increase efficiency.

Processing starts with the rules that only contain terminal symbols. After these have been processed, the non-terminal symbols heading these rules have corresponding rendered forms, and never need to be reprocessed. Now all rules containing only terminal symbols and rendered non-terminals can be processed. The process continues in this way from the bottom of the hierarchy to the top, where rule *S* itself can be rendered in terms of all the other rules, producing the final image.

This can be implemented by depth-first recursion through the grammar, using a table to identify non-terminals which have already been rendered. Figure 7.6 gives pseudo-code for the recursive function. It is passed a symbol to render, and returns the rendered form of the symbol. It is invoked with the start symbol *S*, and examines each of the symbols in *S* in turn. If the symbol is terminal, or if it has already been rendered, it adds the rendered form of the symbol to the rendered form of the rule. If the symbol has not been rendered, it calls itself recursively. Initially it will recurse to the bottom of the hierarchy, then work its way up to the top, rendering rules as necessary.

```
scene = RenderSymbol(S)

Function RenderSymbol(symbol a)
    Form = {}

    For each symbol b in the rule headed by a
        if b is not terminal and hasn't been rendered then
            rendered[b] = RenderSymbol(b)
        Form := Form ∪ rendered[b]

    return Form
```

Figure 7.6 Bottom-up rendering of a scene

This function will be called once for every rule in the grammar, and will visit every symbol in the grammar in the loop. Therefore its time complexity is O(*grammar size*), rather than O(*sequence size*). The storage used by this procedure is the size of the grammar, plus the space required to store the rendered form for each rule. So the space complexity is O(*grammar size* + *number of rules*). It is therefore possible to compute the improvement in rendering time based on the amount of compression achieved on the sequence. For example, returning to the two context-sensitive, stochastic L-systems in Table 6.3, the 908,670 symbol sequence on the left is reduced to a 640 symbol grammar: a 1419-fold reduction in work. The 140,872-symbol sequence on the right is reduced to an 1879 symbol grammar: a 74-fold improvement.

7.3 Music

Repetition is a fundamental part of musical structure. It occurs at a high level in the verse-chorus-verse structure of popular music, where repetition is often curtailed only by 'fading' the music out. It also occurs at a micro level in music such as J.S. Bach's fugues, where themes or motifs consisting of a few notes are cleverly interwoven, often slightly modified by transposition or rhythmic changes. This section investigates whether these repetitions, and hierarchies of repetitions, can be detected in music. A prerequisite for such an investigation is a suitably large computer-readable corpus of music. Fortunately, several such corpora exist, and one consisting of transcriptions of Bach's chorales (Mainous and Ottman, 1966) was selected for the experiment. Bach's prolific output of 371 chorales, of which the corpus contains 97, makes it an excellent example of a homogeneous single-genre collection. Bach did not compose the melodies, but harmonised contemporary congregational hymns for performance by choirs in productions such as his cantatas. The chorales are short works, averaging 47 notes each.

The chorales are in a variety of keys, and it was decided to remove this source of variability for the purposes of comparing the form of each chorale. Furthermore, to admit the possibility of similar themes being repeated with transposition, the aspect of absolute pitch was removed entirely. This was performed by considering the sequence of intervals between notes rather than the absolute value of the notes themselves. For example, the sequence of notes *C*-*E*-*G* represents a major triad in the key of *C*, while *F*-*A*-*C* is a triad in *F*. Re-expressing this sequence as a series of intervals measured in semitones gives the sequences +4, +3 for both triads. That is, the distance from *C* to *E* and from *F* to *A* is four semitones, while the distance from *E* to *G* and from *A* to *C* is three semitones. Finally, to detect repetition over different rhythms, rhythmic information was removed. Performing this transformation and concatenating all 97 chorales resulted in a sequence of 4541 intervals.

From the interval sequence, SEQUITUR produced a grammar with 362 rules, 1253 symbols in rule *S*, and 2017 symbols in total. The most popular rules in the sequence were +2, +2 and –2, –2, which both occurred 140 times. These two represent ascending and descending tone steps respectively. The next most popular rules were –2, –1 and –1, –2, which representing descending tone/semitone pairs. Within the grammar, the rule that was most used as part of other rules was 0, 0 which translates to three identical notes in succession, while the next most popular was 0, –2, which represents two identical notes followed by a descending tone.

The longest rule contains 50 intervals, and occurs in chorales 66 (*Christ unser Herr zum Jordan kam*) and 119 (*Was alle weisheit in der Welt*). This curiously long sequence—the chorales are only 65 and 83 intervals long respectively—is explained by the fact that the chorales are different harmonisations of the same melody, which was composed by Johann Walther in 1524. The repetition does not cover the whole chorale, because one score is notated with a repeat sign for one section, whereas the other simply contains two copies of the section. The existence of multiple versions of the same chorale, which was not obvious from a cursory examination of the chorales, is typical of the serendipitous discoveries that techniques such as SEQUITUR make possible. Examining the other rules in descending length identifies

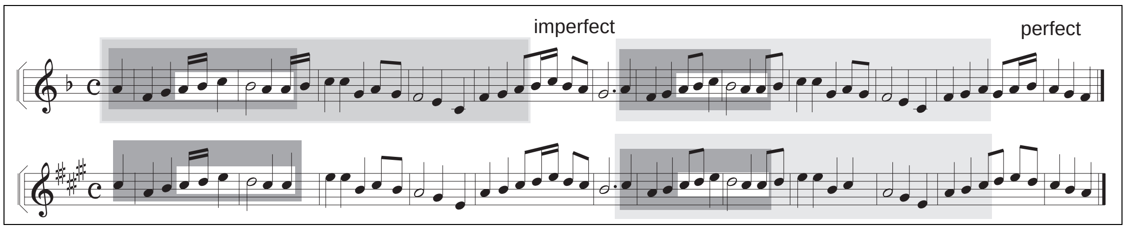


Figure 7.7 Illustration of matches within and between two chorales: for chorales 50 and 63 by J.S. Bach.

seven of the eight such pairs in the collection. The eighth pair was undetected because both chorales consist partly of a long repetition, which differs slightly between the chorales. The internal repetition is thus longer than the matching portion between the chorales, and the inter-chorale relationship is hidden.

Detecting simple repetitions, while conceptually less interesting than detecting a hierarchy, is nevertheless of practical use. In this case, the provider of the corpus, who was using it for investigation into musical structure, was unaware of the pairs of harmonisations, even though the structure was of a very simple nature. Detecting such repetitions is therefore a useful function, and indeed a fundamental one. The approach described in Section 3.5, which operates in quadratic time in the size of the input, guarantees to find the longest internal repetition in a sequence by exhaustive search. No linear time algorithm for this problem exists. SEQUITUR represents a compromise. There are no guarantees that it will identify the longest repetition, and where there are overlapping repetitions it reports only one. However, it operates in linear time, and will usually, except in pathological cases, form a rule for most of the longest repetition. This means that the longest repetition can be identified by extending a long rule in the grammar to incorporate matching symbols that are covered by a neighbouring rule.

As for hierarchies, Figure 7.7 shows chorales 50 and 63, which are both harmonisations of the same melody, with slight melodic variations. Figure 7.8 shows the hierarchy for chorale 50, where the fundamental symbols are intervals measured in semitones. Slashes denote whether an interval is up or down. The part of rule S that covers chorale 50 contains four non-terminals, the first and third of which are identical. This indicates that the chorale has two similar parts with different endings. These endings correspond to imperfect and perfect cadences respectively, which signal the continuation or end of a musical form. The two identical rules are represented by the lines at the top of Figure 7.8, and in Figure 7.7 the repetition is

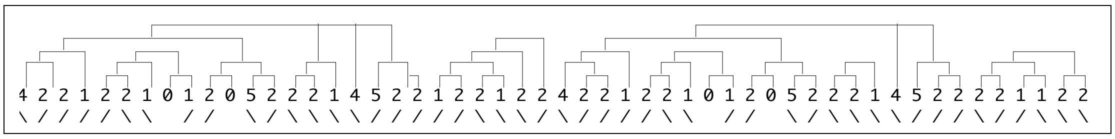


Figure 7.8 Hierarchy for O *Welt, sie hier dein leben*, chorale 50 by J.S. Bach.

marked with the large light grey boxes. This rule also occurs in the second half of chorale 60. Within this rule, there is a shorter rule which represents the match between the two chorales in the first half. This is indicated in Figure 7.8 by the first bar on the second row, and in Figure 7.7 by the darker grey box. An even smaller rule, the second bar in the third row of Figure 7.8, occurs seven times in five chorales, and is shown as a white box in Figure 7.7. The two cadences, represented by the second and fourth hierarchies in Figure 7.8, and labelled in Figure 7.7, occur ten times each in the 97 chorales, indicating their usefulness as musical phrases.

In total, SEQUITUR has proven capable of detecting simple but significant repetitions in music, as well as demonstrating hierarchical structure of a group of chorales. It has identified chorales that share the same original melody, and repetitions within a chorale where the melody has two parts which are identical apart from the cadences. It also identifies the cadences by comparison between chorales.

7.4 Phrase identification

It has been observed that in many large full-text retrieval systems, more information is stored than is ever retrieved (Witten *et al.*, 1994). This is a necessary consequence of the information explosion, where, as mentioned in Chapter 1, the amount of information in the world is claimed to double every twenty months. This asymmetry of storage and retrieval has implications for the kind of technology used for each process. For example, it makes sense to put greater emphasis on retrieval technology than storage technology. Rather than formally, manually categorising and cross-referencing information—much of which will never be accessed—at the time of storage, effort should be expended on improving retrieval strategies.

This section explores the use of SEQUITUR to form hierarchies automatically from large corpora and documents, coupled with techniques for making intelligent use of the hierarchies at retrieval time. The first technique relates to abstracting: rather than writing abstracts—a time-consuming, knowledge intensive task—prior to storage, significant terms and conceptual hierarchies can be reconstructed when a document is retrieved. The second application is to full-text retrieval *per se*, where the hierarchy can be used to guide formation of search terms, or even perform the whole retrieval process, making use of a logarithmic-time tree search through the hierarchy to specific portions of the text.

7.4.1 Computer-assisted abstracting

Creating abstracts of documents is a time- and knowledge-intensive operation which has traditionally been performed manually by human abstractors. Numerous attempts have been made to automate the abstracting process by writing computer programs that attempt a superficial comprehension of the text in order to produce a condensed version. While these systems may ultimately produce excellent results, they are currently unable to match human expertise. Craven (1993) suggests that a useful intermediate position is to provide automated tools to assist a human abstractor, in the same way that spelling checkers and thesauri are provided to assist writers. The TEXNET system developed by Craven analyses a text and makes frequent words and phrases available to the user for inclusion in the abstract. Although we do not necessarily subscribe to the view that this is a good way to produce abstracts, the phrases that SEQUITUR identifies, and the hierarchy in which they are arranged, at least provide a conceptual taxonomy that is of some use in identifying the subject of the text.

The idea is to provide the user with a list of common phrases from the text, that is, rules from SEQUITUR's hierarchy, and allow the user to traverse the hierarchy starting from an interesting phrase. When ranked by frequency, phrases such as *and the* and *to the* invariably appear at the top, but provide little insight into the subject matter of the text. For this reason, the vocabulary is divided into common, frequent words, referred to as *stop words*, and less frequent *content words*. A rule must contain at least 2 content words to appear in the list.

ranked list of rules	rules containing 'speech output'	rules containing 'speech output systems'
speech output	~ systems	...most ~ can adopt...
speech synthesis	~ from computers	...convenient basis for ~ which use high quality...
as well as	~ system	
db octave	~ peripherals	
tone group	~ technology	
reflection coefficients	~ one	
vocal tract	~ is	
linear prediction		
prosodic features		
frequency spectrum		
...		

Figure 7.9 Phrases discovered by SEQUITUR for automated abstracting

Book2 of the Calgary corpus, which is *Principles of computer speech* by Witten (1982) was presented word by word to SEQUITUR. The top 10 rules using a stop-word list of 72 words gives the phrase list in the first column of Figure 7.9. Most of the entries are descriptive of the content of the book, and are useful phrases to include in an abstract. If the most frequent phrase, *speech output*, is of interest, the seven rules that contain this phrase can be retrieved. Choosing one of these, *speech output systems*, results in a list of occurrences of this rule. In this case, there are two occurrences, both in rule *S*. As well as finding an efficient way of identifying a useful phrase, this hierarchy also gives the abstractor a guide to the hierarchy of concepts in the text, which is useful for understanding.

7.4.2 Query formulation for full-text retrieval systems

Full-text retrieval systems make it possible, in principle, to retrieve relevant information very efficiently from a huge indexed collection. Whereas the efficiency of indexing and storage are straightforward to evaluate, and practical techniques have been developed (Witten *et al.*, 1994), the task of formulating appropriate queries and retrieving only relevant documents remains problematic. Retrieval of matching documents involves both *precision* (not returning irrelevant documents) and *recall* (not overlooking any relevant documents) (Salton, 1989). There has been much research on how to maximise precision and recall given a particular query (Harman, 1992), but here we suggest a technique for aiding a user to formulate a query that is easier for the retrieval mechanism to fulfil. The technique allows a user to become familiar with the content of the database, and to include phrases that make the query much more specific. Furthermore, the hierarchy formed by

rules containing 'grammar'	rules containing 'context-free grammars'	rules containing 'probabilistic context-free grammars'	rules containing 'probabilistic context-free grammars (PCFGs),
context-free ~s	probabilistic ~	~ (PCFGs)	...researchers have become interested in ~ for language modelling the fact that a PCFG...
context-free ~	...the set of ~...	...recent interest in ~ for language modelling...	...brief aside let us come back to the ~ that we referred to...
~ procedure	...described by ~...	...language model used in particular ~ will have to deal with this problem...	
tree ~s			
functional ~			
systemic ~			
systemic ~s			
logic ~s			
attribute ~s			
with a corpus based ~			
a context-free ~			

Figure 7.10 Hierarchy of phrases for searching a large text base

SEQUITUR can replace the retrieval system entirely by allowing the user to pinpoint a useful section in logarithmic time.

SEQUITUR was invoked on a large body of computer science technical reports, part of the 700 Mb corpus contained in the New Zealand Digital Library (Witten *et al.*, 1996). The reports were presented as a sequence of words, and all words were case folded before processing. A 22 Mb sample was chosen, which included 350 technical reports from six sites. The sample represented a sequence of 3.5 million words with a vocabulary size of 30,000. The resulting grammar had 210,000 rules and 1.7 million symbols.

The grammar's role in constructing a query might proceed as follows. Imagine the user needs some information on grammars. The first column of Figure 7.10 lists the eleven rules that contain the word *grammar*. This might allow the user to narrow the search. Suppose they are not interested in functional grammars, systemic grammars, logic grammars, attribute grammars, tree grammars or grammar procedures; they therefore choose *context-free grammars*. Next, the system displays the places where this rule occurs: within a longer rule, *probabilistic context-free grammars*, as well as two places in rule *S*, signified by the ellipses surrounding the excerpts in the second column of Figure 7.10 After choosing the longer rule, because probabilistic grammars are of interest, the user is presented with places where that rule appears, shown in the third column of Figure 7.10. This includes a longer rule with the abbreviation *PCFGs* in parentheses, and two occurrences in rule *S*. Choosing the longer rule leads to the two excerpts from rule *S* in the fourth column of Figure 7.10.

The final rule, which associates the acronym *PCFGs* with the full phrase, warrants further discussion. The information that this acronym is used in place of the phrase is potentially extremely useful: the user may decide to perform another search on the abbreviation, because it is likely that once it has been defined, the acronym alone will be used. It is interesting to consider how this information might have been obscured in a different method of presentation. This hierarchy contrasts with methods such as keyword-in-context (KWIC) displays, where all occurrences of the search term are displayed along with the surrounding context. The fact that the acronym co-occurs with the phrase several times is not significant in a KWIC display, and would not be given particular prominence. Its embodiment in a rule, however, ensures that the SEQUITUR-based method puts it at the top of the list of occurrences.

In general, the user can traverse the grammar, extending and hence specialising the query term. The grammar offers a tree structure to go from a single word to a particular occurrence of that word in the text. In the tree, any occurrence in rule *S* is a leaf node, and any occurrence in another rule is an internal node, because the rule will appear elsewhere in the grammar. It is possible to stop at any internal node and use that phrase as a term in a query, or continue following the tree to a leaf and retrieve a document without posing a query to an indexing system.

An advantage of this approach is that it gives the user a good idea of the subject matter of the text, and presents it in manageable chunks determined by the branching factor of each rule. This is especially evident at the very top level, where the list of rules involving the word 'grammar' provide a plausible taxonomy of concepts involving grammars. While this application has only been investigated at a conceptual level and no doubt requires modification for practical use, it shows significant promise.

The technique is very efficient. Once the word *grammar* has been found in the lexicon, which can be performed quickly using a trie data structure, SEQUITUR's internal pointers between different instances of the same symbol can be used to list the rules in time linear in the number of phrases. When a rule is picked, the places in which the non-terminal appears can be accessed in the same way. Once the hierarchy has been formed, all the pointers necessary to traverse the hierarchy are in place, and there is no need for any search apart from the initial lexicon indexing.

7.5 Macromolecular sequences

This chapter began by asserting that linguistic sequences are those with which we have the most experience and which we encounter most frequently. There is, however, a sequence that is much more fundamental to our being, but which is much less well understood. The sequence of molecules that make up our genes determine both our individual characteristics and our collective species identity. For this reason its structure is of great interest to biologists, and to the human population at large. Its basic structure turns out to be surprisingly similar to the sequences that computer scientists are accustomed to dealing with in digital computers. Instead of being based on a binary alphabet, DNA sequences consist of four unique molecules. These molecules: adenine, cytosine, guanine and thymine, are known as *nucleotides*, or *bases*, and can be represented as the letters A, C, G, and T to form computer-readable sequences. The kinds of structures that exist in these sequences are only partly known, but they include repetition, palindromes, corruption with noise, segments of random 'junk' DNA, and substitution of one base for another.[7]

SEQUITUR can only capture one of the structures that are known to exist in DNA, namely repetition, but this nevertheless presents an interesting challenge. The National Centre for Biotechnology Information (NCBI), part of the National Institute of Health, maintains a database of nucleotide sequences (GENBANK), and publishes the sequences both on CD-ROM and over the Internet. A simple experiment was devised to measure the compression that SEQUITUR could achieve on a set of sequences relative to other compression techniques. This provides an indication of whether any structure is being identified. Previous research has shown that the sequences have an entropy close to the default value of two bits per base, so that compression is impossible. Williams and Zobel (1996), after much experimentation with various compression schemes, found that the best encoding was to pack four bases into an eight-bit byte.

[7] An introduction to the subject, along with a discussion of the Human Genome Project, can be found in Kevles and Hood (1992).

a
```
gtgagacccaagggagacctggccgggcccagtcctgggagtgagttgac
ctgtcgtttgcacatgcagggctctgtgcacaatgcgtgacaatggcttttag
```

b
```
rghwikdcpkrprdqkkpapvltlgedsetlgedseqgcqgsgappeprltls
vgghpttflvdtgaqhsvltkangplssrtswvqgatgrkmhkwtnrrtv
```

Figure 7.11 Examples of macromolecular sequences
(a) nucleotide sequence
(b) amino acid sequence

The sequences for *homo sapiens* were extracted from the primate file provided by the NCBI, and put in a file, one per line. This resulted in a file 2,869,960 bytes in size, consisting of sequences similar to the one shown in Figure 7.11a. Table 7.4a shows the results of the application of several compression schemes to the data, along with the compression achieved relative to the straightforward coding scheme of two bits per base. The *gzip* compression scheme expands the sequence relative to the simple coding, resulting in a file 2% larger. PPM compresses the file by 1%, while SEQUITUR achieves compression to 94%. It appears that SEQUITUR detects structure that the other techniques do not. A similar experiment was attempted with sequences of amino acids, which are built from subsequences of nucleotides. There are twenty different amino acids, and the sequences are similar to the example shown in Figure 7.11b. The distribution of the acids is somewhat skewed, so encoding them all in a fixed number of bits does not make sense. For this reason, the compressed sizes are compared against the original number of acids, and are summarised in Table 7.4b. *gzip* achieves compression to 51% of the original, while PPM achieves 44%. SEQUITUR is again the best, with 41% of the original size.

The grammars for both sequences contain some rules that are very long. On

a

encoding	size	size vs 2 bpb
original	2 869 960	400%
2 bits per base	717 490	100%
gzip	734 045	102%
PPM	709 179	99%
SEQUITUR	670 760	94%

b

encoding	size	% of original
original	1 586 289	100%
gzip	694 871	51%
PPM	799 179	44%
SEQUITUR	638 977	41%

Table 7.4 Compression of macromolecular sequences
(a) nucleotide sequences for *homo sapiens*
(b) primate amino acid sequence

inspection of the original sequences, it transpires that some of these sequences are the result of more than one researcher sequencing the same portion of DNA, or the same sequence being referenced twice. The sequences, however, are not usually completely identical. It is difficult to produce a set of sequences free of these spurious repetitions, so it is unclear whether interesting structure is being detected. SEQUITUR probably excels partly because of its ability to take advantage of repetitions that are widely separated. Its superior performance nevertheless provides motivation for further research.

7.6 Summary

This chapter has applied the techniques discussed in previous chapters to a variety of data that occurs naturally or in other human and scientific contexts: text in three languages, word-class sequences, music, L-systems for describing complex plants, word-based analysis of large corpora, and DNA sequences. The success of SEQUITUR in each domain supports the claim made in Chapter 1 that repetitive structure exists and can be efficiently detected.

8. Conclusions

This thesis makes two claims: that certain kinds of structure appear in sequences from a variety of sources, and that it is possible to detect such structure automatically and efficiently. The claims have been addressed in reverse order: first by constructing an efficient inference technique, and then applying it to a range of sequences.

We have described a technique that balances elegance and practicality. Elegant techniques are often general ones—a scheme that is peppered with arbitrary constants and procedures might succeed in one domain, but will lack applicability in a wider context. The simplicity of the SEQUITUR algorithm permits a concise complexity proof, a principled way of employing domain knowledge, and a novel reparsing mechanism for improving performance. That the scheme is practical has been demonstrated in a variety of ways, and we will recapitulate them in Section 8.2. First, however, we will summarise the fundamental techniques and how they relate to the applications discussed, which is summarised in Figure 8.1.

8.1 The techniques

The hierarchical modelling technique developed in this thesis has its roots in work by David Maulsby. While visiting from the University of Calgary in 1992, he described a hierarchical representation for action sequences of computer users. His belief was that such tasks are hierarchically organised—that small sub-tasks combine to produce larger-scale behaviours. The hierarchy, if it could be inferred, would provide a basis for automating the task. The idea, of course, is much more general than the task structure domain, and formed the germ of this thesis. The key problem was to produce an efficient algorithm to infer a hierarchy from a sequence. Even though the SEQUITUR algorithm is simple to describe, it was certainly not immediately obvious. One principle seemed paramount: the concept was simple, so the algorithm should be similarly uncomplicated.

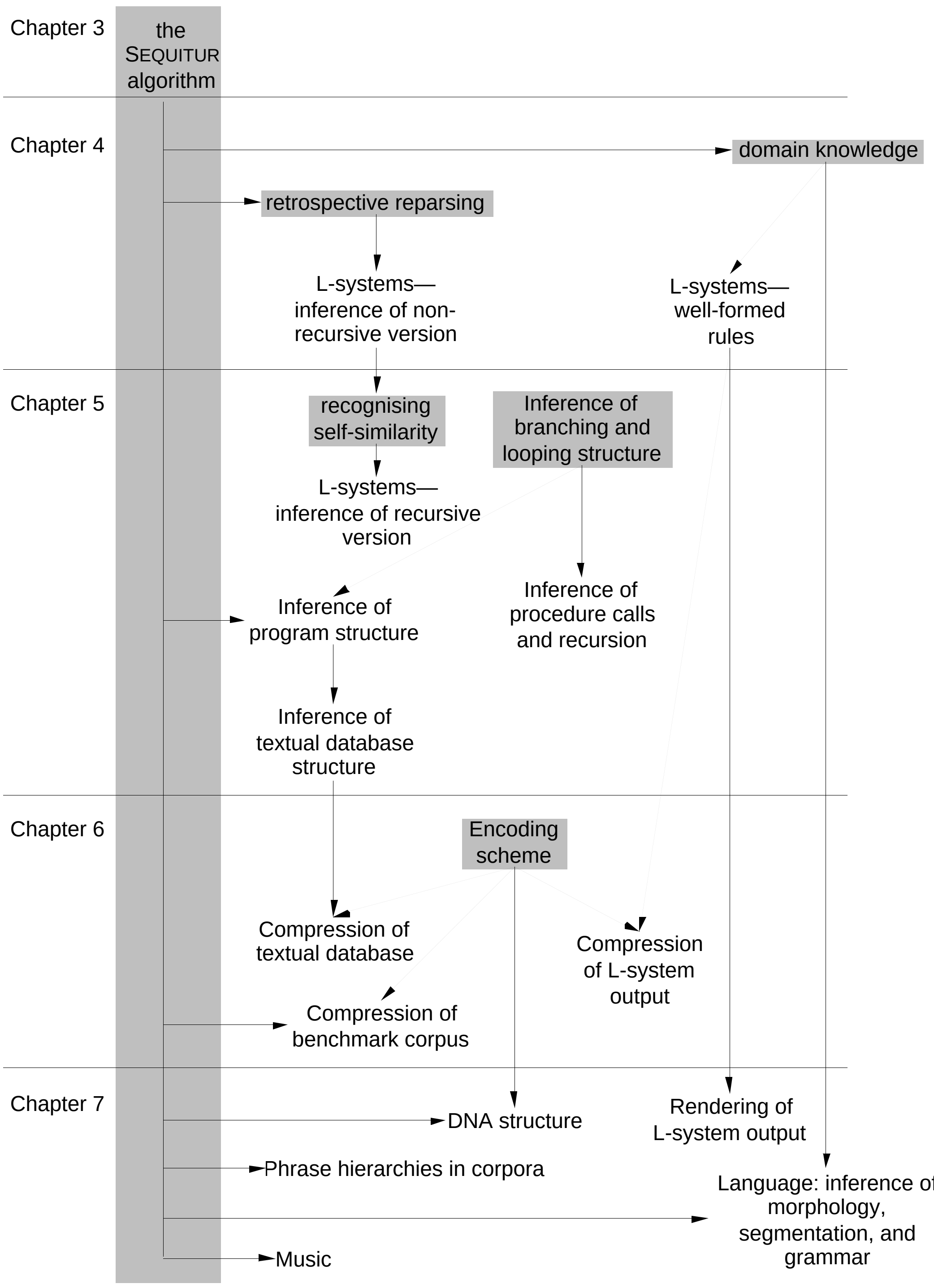

Chapter 3
the SEQUITUR algorithm
Chapter 4
domain knowledge
retrospective reparsing
L-systems—inference of non-recursive version
L-systems—well-formed rules
Chapter 5
recognising self-similarity
Inference of branching and looping structure
L-systems—inference of recursive version
Inference of program structure
Inference of procedure calls and recursion
Inference of textual database structure
Chapter 6
Encoding scheme
Compression of textual database
Compression of L-system output
Compression of benchmark corpus
Chapter 7
DNA structure
Rendering of L-system output
Phrase hierarchies in corpora
Language: inference of morphology, segmentation, and grammar
Music

Figure 8.1 Relationships between techniques (in grey boxes) and applications described in the thesis

The simplicity of the algorithm, as expressed in terms of the maintenance of two constraints described in Chapter 3, permits a concise proof of its computational complexity. We have shown the time and space complexity to be linear in the sequence size by relating the amount of processing to the number of symbols saved by transforming the sequence into a hierarchical grammar. Figure 8.1 shows the basic algorithm as a grey column, and it forms the basis for all the techniques and applications in the thesis, except for the inference of branching and looping structure in Chapter 5.

When information about the structure of a sequence is available, it should be possible to improve structural inference. However, it is important that in modifying the algorithm, the generality and simplicity of the original is not lost. We have shown that because of the simplicity of the original algorithm, it is sufficient to introduce a simple test in a fundamental part of the algorithm to bias the inference mechanism towards well-formed rules. We have shown examples of domain knowledge for L-systems, where brackets must be correctly nested, natural language, where word boundaries are significant, and word-class sequences, where sentence boundaries must be observed. This technique appears at the top right of Figure 8.1, labelled 'domain knowledge,' and contributes to four applications.

Where domain knowledge is not available—for example in an L-system that does not contain brackets—it is still possible to improve the grammars produced by the basic algorithm. The way in which this is performed is counter-intuitive. Throughout processing, rules at the end of rule *S* are extended to the left or right, even if the transformation increases the size of the grammar. Over an entire sequence, this reparsing can markedly improve the grammar size and quality. The way in which the individual transformations interact over a long sequence is not well understood at present, and we leave a rigorous explanation for further work. Intuitively, in structured sequences, rules are given several opportunities to regain symbols that they lose to neighbouring rules, and in the long term, the rules that exhibit the greatest consistency retain ownership of neighbouring symbols. This is retrospective reparsing, shown in the grey box at the top left of Figure 8.1.

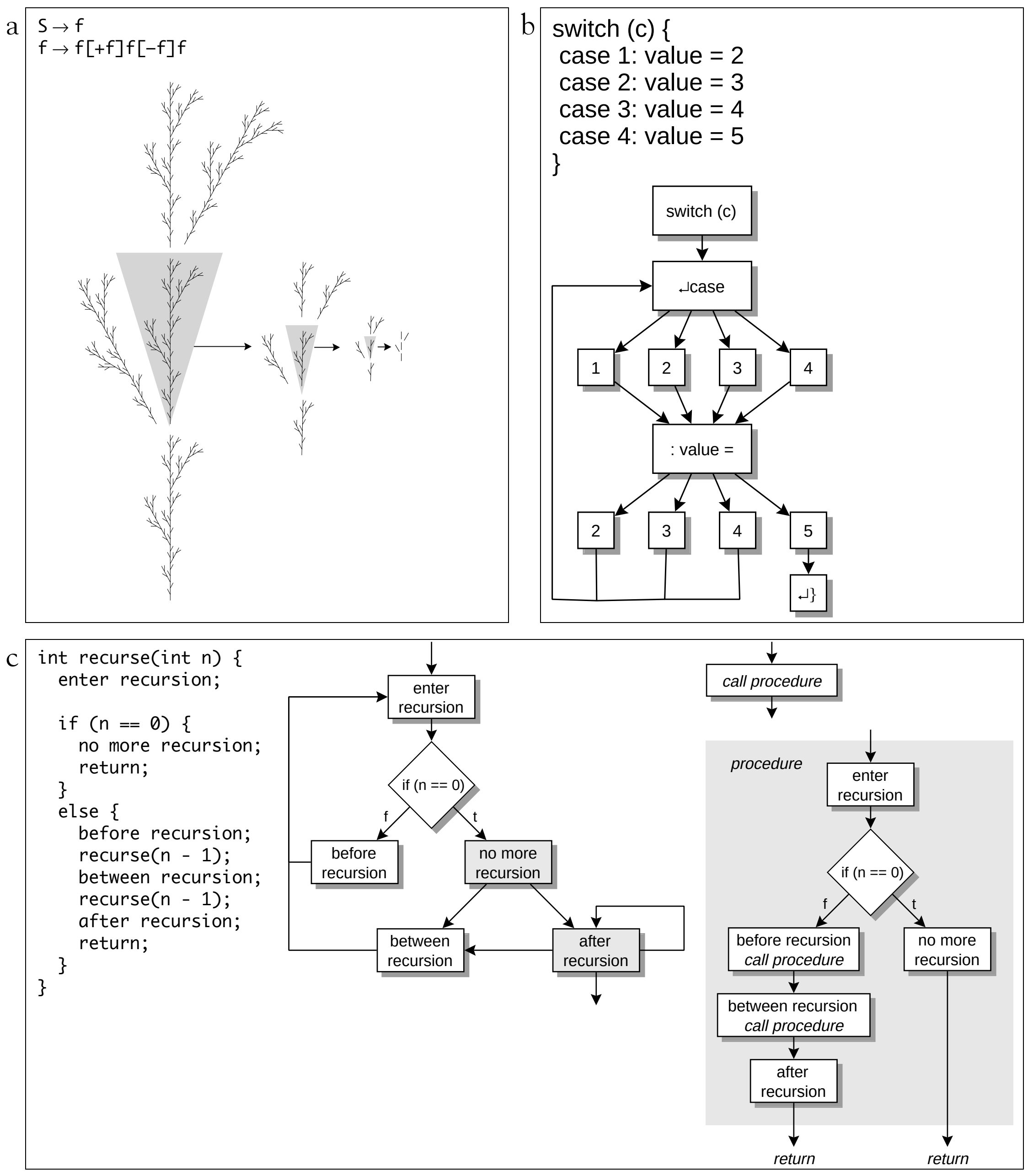
In addition to the algorithm for forming hierarchical grammars, we have applied and extended a simple technique by Gaines (1976) for inferring automata ('inference of branching and looping structure' in Figure 8.1). It is extremely effective in situations where the sequence is expressed in terms of symbols at the correct level of granularity. However, if each symbol has little significance—individual characters in program source, for example—the inferred automaton fails to capture the sequential structure. The solution to this problem is to apply the SEQUITUR algorithm to the sequence first, and then produce an automaton from the higher-level non-terminals. This is a compelling symbiosis: SEQUITUR identifies the significant segments of the sequences, and the automaton captures the branching and looping relationships between them. The combination is analogous to the lexical analyser and parsing passes of a compiler, where the first pass groups individual characters into meaningful tokens for the second pass. The main problem with this approach is the ease with which SEQUITUR can be misled to recognise segments that include more than a single token. Investigating solutions to this problem is also a direction for future work.

Another technique for generalising grammars involves recognising self-similarity in the hierarchical structure of a sequence ('recognising self-similarity' in Figure 8.1). Recursive grammars such as the fractal L-system grammars for producing snowflake and tree-like figures give rise to similar patterns at different levels of detail. With the improved inference achieved by retrospective reparsing, this self-similarity is clearly exhibited in SEQUITUR's hierarchies. Automatically recognising this recursive structure consists of finding a unification of two rules that produces a concise recursive grammar capable of reproducing the original sequence. Again, the algorithm is simple: it consists of only six Prolog clauses.

The final major technique introduced is a way of encoding a grammar to achieve data compression ('encoding scheme' in Figure 8.1). This transforms the SEQUITUR algorithm into a compression scheme that can be evaluated alongside a large cohort of similar schemes. It outperforms all other techniques in its class, and in several instances achieves results better than any current compressor. Simply recoding the final grammar for the sequence does not result in the best performance. Instead, SEQUITUR operates at both the encoding and decoding ends, and enough information is transmitted to trigger the two constraints operating at the decoder.

8.2 The applications

The problem of inferring concise and correct grammars from sequences precludes optimal solutions. It is known from data compression that the problem of finding the smallest hierarchy for a sequence is NP-complete (Storer, 1982). It is also known from grammatical inference that no algorithm can guarantee to infer a grammar from the sequences that it produces (Gold, 1967). Furthermore, in many domains, the sequence does not necessarily emanate from a member of a known class of grammars. These results mean that evaluation of the techniques that we have developed must be empirical, and consequently we have described many applications in a variety of domains.



```
S → f
f → f[+f]f[-f]f
```

```
switch (c) {
 case 1: value = 2
 case 2: value = 3
 case 3: value = 4
 case 4: value = 5
}
```

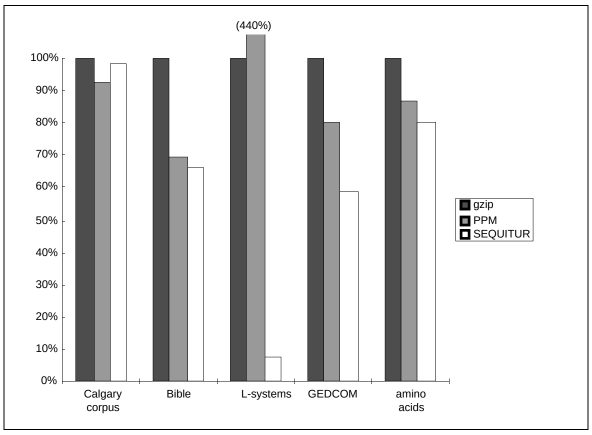
```
int recurse(int n) {
  enter recursion;

  if (n == 0) {
    no more recursion;
    return;
  }
  else {
    before recursion;
    recurse(n - 1);
    between recursion;
    recurse(n - 1);
    after recursion;
    return;
  }
}
```

Figure 8.2 Review of results
(a) L-system hierarchy inference
(b) textual structure inference
(c) program inference from an execution trace

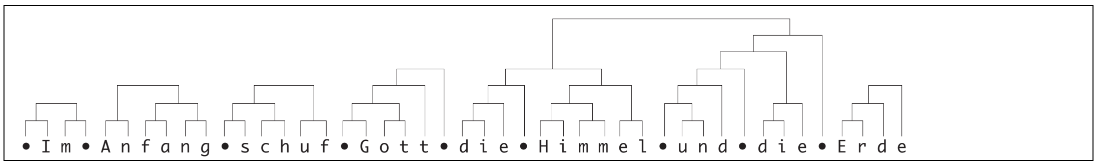
d

```
0 @26DS-KX@ INDI
1 AFN 26DS-KX
1 NAME Dan Reed /OLSEN/
1 SEX M
1 BIRT
2 DATE 22 JUN 1953
2 PLAC Idaho Falls,Bonneville,Idaho
1 FAMC @00206642@
0 @00206642@ FAM
1 HUSB @NO48-3F@
1 WIFE @93GB-DD@
1 CHIL @26DS-KX@
1 CHIL @21B7-WR@
0 @26DN-7N@ INDI
1 NAME Mary Ann /BERNARD
```

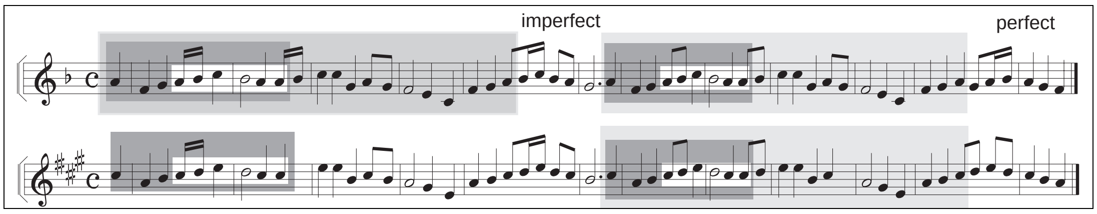
e



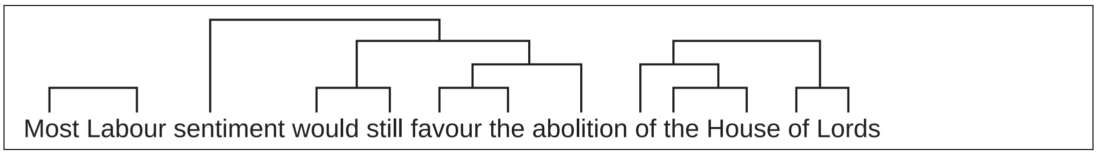
f


• I m • A n f a n g • s c h u f • G o t t • d i e • H i m m e l • u n d • d i e • E r d e

g



h


Most Labour sentiment would still favour the abolition of the House of Lords

i

rules containing ‘grammar’	rules containing ‘context-free grammars’	rules containing ‘probabilistic context-free grammars’	rules containing ‘probabilistic context-free grammars (PCFGs)’
context-free ~s context-free ~ ~ procedure tree ~s functional ~ systemic ~ systemic ~s logic ~s attribute ~s	probabilistic ~ ...the set of ~... ...described by ~...	~ (PCFGs) ...recent interest in ~ for language modelling... ...language model used in particular ~ will have to deal...	...researchers have become interested in ~ for language modelling the fact that a PCFG... ...brief aside let us come back to the ~ that we referred to...

Figure 8.2 Review of results (continued)
(d) inference of textual database structure
(e) compression performance
(f) inference of language structure
(g) inference of musical structure
(h) inference of grammatical structure
(i) word hierarchies from a large corpus

Working from top to bottom in Figure 8.1, we begin with the inference of grammars from L-system output. The L-system domain has been a fruitful one: it straddles the gap between artificial and natural sequences, because L-systems are formal grammars that provide realistic representations of natural phenomena. Thus they are of interest in formal languages, computational biology, and computer graphics.

With reparsing, a non-recursive equivalent of the original L-system can be inferred. Adding the unification process for recognising self-similarity allows the original to be reconstructed exactly. This represents a novel approach to grammatical inference that is interesting from a theoretical perspective in computer science, and from a practical perspective in computational biology. The process represents another step towards the biological grail of recognising plant structure directly from a graphical description.

Bracketed L-systems enable the use of domain knowledge to guide rule formation. The resulting grammar represents a hierarchy of graphical objects that can be used to expedite rendering. SEQUITUR can reduce both time and space complexity by several orders of magnitudes, making it possible to render much more complex scenes. Figure 8.2 is a gallery of results from the thesis. Figure 8.2a shows an L-system that draws the tree shape shown below.

Forming an automaton from a sequence allows a fundamentally different kind of structure to be recognised. For example, a program can be inferred from a trace of its execution, suitably expressed. We have extended this ability in two directions. First, we have described an algorithm for recognising the effects of procedure calls and recursion in the trace. The left-hand side of Figure 8.2c shows a recursive program and, in the centre of the figure, the automaton inferred from its execution trace. The inferred automaton is non-deterministic, but the deterministic, recursive version can be inferred from it automatically, and is shown on the right-hand side. Second, the automaton technique can be combined with SEQUITUR to form a powerful system that infers the structure at the bottom of Figure 8.3b from the text at the top. Each state in the automaton corresponds to a non-terminal in the top-level rule of the inferred grammar.

As Figure 8.1 shows, no further use is made of the inference of procedure calls and recursion, but the inference of program structure is adapted to deal with a textual database. Extending the idea of using SEQUITUR's grammar as the basis for recognising branching structure, Figure 8.2d shows an excerpt from a large textual database containing genealogical information. Analysis of phrases that predict each other allows the fixed template structure of the database from the variable content: the structure is shown in black, and the content in gray.

All of these techniques compress the sequence as a by-product of inference, and Figure 8.2e shows a histogram of compression results for various sequences. Figures are expressed relative to *gzip*'s performance: smaller is better. With the addition of an encoding scheme, shown in the grey box near the bottom of Figure 8.1, SEQUITUR outperforms other dictionary compression schemes such as *gzip* on a large benchmark corpus, and does nearly as well as the current best scheme, PPM. On other sequences, such as the Bible, amino acid sequences, the genealogical database and L-systems, SEQUITUR outperforms any other system. In the case of one L-system, SEQUITUR performs an order of magnitude better than its nearest rival.

SEQUITUR performs just as well on French and German sequences as it does on English. Figure 8.2f shows the hierarchy formed from a German version of the Bible, which corresponds closely with morphological intuition about the structure of German words and phrases. Figure 8.2g continues the germanic theme, this time with the music of Bach. Analysing a corpus of Bach's chorales results in the identification of a hierarchy of musical phrases, including long repetitions, and imperfect and perfect cadences. Figure 8.2h shows a kind of parse tree inferred from a sequence of word class tags. The structure is similar to the ideal manual parse. Finally, Figure 8.2i shows a hierarchy of phrases inferred from a large corpus of computer science technical reports. The hierarchy allows rapid traversal of the phrases from general to specific, and, if necessary, to a particular document in the collection.

8.3 Future work

The thesis is certainly not the last word on the subject of detection of sequential structure. In fact, it feels more like a Genesis than a Revelation. The main avenue for future exploration is the inference of branching and looping structure, discussed in Chapter 5. This is where the MDL principle, background knowledge, and sophisticated searching and backtracking techniques must be harnessed to corral the vast problem space that such inference entails. The thesis has helped to define the problem and elucidate the difficulties. It has also contributed simple, effective techniques that can be used as building blocks in forming a solution.

The success of the simple, blind algorithm for retrospective reparsing that we finally settled on in Section 4.4 remains inexplicable. While I have expressed some intuitions about its operation, it still requires a formal explanation. Such an explanation would need to account for the global effect of the algorithm, which improves the grammar, as opposed to the local effect, which may temporarily degrade it.

What originally motivated the hierarchical representation was programming by demonstration, which involves the automation of a computer user's task by inferring the structure of their actions (Nevill-Manning, 1993). Unfortunately it is hard to apply the new techniques to this task because it is hard to gather traces: interaction is paramount, and has a significant effect on the ability of a system to learn. Building a system in which to embed inference techniques is an ambitious task involving skills ranging from interface design to discourse management.

Many of the sequences discussed in the thesis have a structure that is currently understood. One exception is DNA, and the real test of these techniques is their ability to help people discover such structure. The tantalising results in macromolecular sequences must be analysed and explained in terms of the underlying biochemistry, a task that requires interdisciplinary collaboration. Other examples might include previously unencountered languages, or communication between other animals such as enigmatic dolphin clicks and whale songs. One day we may even receive an extraterrestrial message whose understanding requires this kind of lexical and grammatical inference.

In any realistic situation, sequential inference techniques will form part of a larger system, a system that will likely include human judgement and interpretation. For this reason, the mechanism developed in this thesis is simple and generic, performing a single task well, without bias towards a particular source. It is explanatory, complementing the natural abilities that people possess. It is dynamite with which to quarry knowledge from mountains of data.

References

Abelson, H., and deSessa, A.A. (1982) *Turtle geometry*. Cambridge, Mass.: MIT Press.

Andreae, J.H. (1977) *Thinking with the teachable machine*. London: Academic Press.

Angluin, D. (1982) "Inference of reversible languages," *Journal of the Association for Computing Machinery, 29*, 741-765.

Bell, T.C., Cleary, J.G., and Witten, I.H. (1990) *Text compression*. Englewood Cliffs, NJ: Prentice-Hall.

Berwick, R.C., and Pilato, S. (1987) "Learning syntax by automata induction," *Machine Learning, 2*, 9-38.

Biermann, A.W., and Feldman, J.A. (1972) "A survey of results in grammatical inference," in *Frontiers of Pattern Recognition*, edited by S. Watanabe, New York: Academic Press.

Brown, R.W. (1973) *A first language: the early stages*. Cambridge, Massachusetts: Harvard University Press.

Chomsky, N. (1957b) *Syntactic structures*. Gravenhage: Mouton.

Chomsky, N., and Miller, G.A. (1957a) "Pattern conception," AFCRC-TN-57-57.

Cleary, J.G. (1980) "An associative and impressible computer," Ph.D. thesis, Department of Electrical and Electronic Engineering, University of Canterbury, Christchurch.

Cleary, J.G., and Witten, I.H. (1984) "Data compression using adaptive coding and partial string matching," *IEEE Transactions on Communications, COM-32*(4), 396-402.

Cohen, A., Ivry, R.I., and Keele, S.W. (1990) "Attention and structure in sequence learning," *Journal of Experimental Psychology, 16*(1), 17-30.

Craven, T.C. (1993) "A computer-aided abstracting tool kit," *Canadian Journal of Information and Library Science, 18*(2), 19-31.

Darragh, J.J., and Witten, I.H. (1992) *The reactive keyboard*. Cambridge, England: Cambridge University Press.

Dietterich, T.G., and Michalski, R.S. (1986) "Learning to predict sequences," in *Machine learning: an artificial intelligence approach II*, edited by R.S. Michalski, J.G. Carbonell and T.M. Mitchell, Los Altos, CA: Morgan Kaufmann, 63-106.

Francis, W.N., and Kucera, H. (1979) "Manual of Information to Accompany a Standard Corpus of Present-Day Edited American English, for Use with Digital Computers," Providence, Rhode Island: Department of Linguistics, Brown University.

Frijters, D., and Lindenmayer, A. (1974) “A model for the growth and flowering of Ater novae-angliae on the basis of table (1, 0) L-systems,” in *L-systems*, edited by G. Rozenberg and A. Salomaa, Berlin: Springer-Verlag, 24-52.

Gaines, B.R. (1976) “Behaviour/structure transformations under uncertainty,” *International Journal of Man-Machine Studies*, 8, 337-365.

GEDCOM Standard: Draft release 5.4, Salt Lake City, Utah: Family History Department, The Church of Jesus Christ of Latter-day Saints.

Gold, M. (1967) “Language identification in the limit,” *Information and Control, 10*, 447-474.

Gottlieb, D., Hagerth, S.A., Lehot, P.G.H., and Rabinowitz, H.S. (1975) “A classification of compression methods and their usefulness for a large data processing center,” *Proc. National Computer Conference*, 453-458.

Guazzo, M. (1980) “A general minimum-redundancy source-coding algorithm,” *IEEE Trans. Information Theory*, *IT-26*(1), 15-25.

Harman, D.K.E. (1992) "Proc. TREC Text Retrieval Conference," Gaithersburg, MD: National Institute of Standards Special Publication, 500-207.

Hayes, J.R., and Clark, H.H. (1970) “Experiments on the segmentation of an artificial speech analogue,” in *Cognition and the Development of Language*, edited by J.R. Hayes, New York: John Wiley & Sons, Inc., 221-234.

Hogeweg, P., and Hesper, B. (1974) “A model study on biomorphological description,” *pattern Recognition*, 6, 165-179.

Johansson, S., Leech, G., and Goodluck, H. (1978) “Manual of Information to Accompany the Lancaster-Oslo/Bergen Corpus of British English, for Use with Digital Computers,” Oslo: Department of English, University of Oslo.

Kevles, D.J., and Hood, L. (Eds.) (1992) *The code of codes: Scientific and social issues in the human genome project*. Cambridge, Massachusetts: Harvard University Press.

Knuth, D.E. (1968) *The art of computer programming 1: fundamental algorithms*. Addison-Wesley.

Laird, P., and Saul, R. (1994) “Discrete sequence prediction and its applications,” *Machine Learning, 15*, 43-68.

Langley, P. (1996) *Elements of Machine Learning*. San Francisco: Morgan Kaufmann.

Lashley, K.S. (1951) “The problem of serial order in behavior,” in *Cerebral mechanisms in behavior*, edited by L.A. Jeffress, New York: Wiley.

Li, M., and Vitanyi, P. (1993) *An introduction to Kolmogorov complexity and its applications*. New York: Springer-Verlag.

Lindenmayer, A. (1968) “Mathematical models for cellular interaction in development, Parts I and II,” *Journal of Theoretical Biology, 18*, 280-315.

Mainous, F.D., and Ottman, R.W. (1966) *The 371 chorales of Johann Sebastian Bach.* New York: Holt, Rinehart and Winston, Inc.

Mandelbrot, B.B. (1982) *The fractal geometry of nature*. San Francisco: W.H. Freeman.

Maulsby, D.L., Witten, I.H., and Kittlitz, K.A. (1989) "Metamouse: specifying graphical procedures by example," *Computer Graphics*, *23*(3), 127-136.

Miller, V.S., and Wegman, M.N. (1984) "Variations on a theme by Ziv and Lempel," in *Combinatorial algorithms on words*, edited by A. Apostolico and Z. Galil, Berlin: Springer-Verlag, 131–140.

Moffat, A. (1987) "Word based text compression," Parkville, Victoria, Australia: Department of Computer Science, University of Melbourne.

Moffat, A., Neal, R., and Witten, I.H. (1995) "Arithmetic coding revisited," *Proc. Data Compression Conference*, Snowbird, Utah, 202-211.

Nevill-Manning, C.G. (1993) "Programming by demonstration," *New Zealand Journal of Computing*, *4*(2), 15-24.

Nevill-Manning, C.G., Witten, I.H., and Maulsby, D.L. (1994a) "Modelling sequences using grammars and automata," *Proc. Canadian Machine Learning Workshop*, Banff, Canada, xv-15–18.

Nevill-Manning, C.G., Witten, I.H., and Maulsby, D.L. (1994b) "Compression by induction of hierarchical grammars," *Proc. Data Compression Conference*, Snowbird, Utah, 244-253.

Nevill-Manning, C.G. (1995) "Learning from experience," *Proc. New Zealand Computer Science Research Students' Conference*, Hamilton, New Zealand.

Nevill-Manning, C.G., and Witten, I.H. (1995) "Detecting sequential structure," *Proc. Workshop on Programming by Demonstration, ML'95*, Tahoe City, CA.

Nevill-Manning, C.G., Witten, I.H., and Olsen, D.R. (1996) "Compressing semi-structured text using hierarchical phrase identification," *Proc. Data Compression Conference*, Snowbird, Utah, 53–72.

Nissen, M.F., and Bullemer, P. (1987) "Attentional requirements of learning: evidence from performance measures," *Cognitive Psychology*, *19*, 1-32.

Olivier, D.C. (1968) "Stochastic grammars and language acquisition devices," Ph.D. thesis, Harvard University,

Pasco, R. (1976) "Source coding algorithms for fast data compression," Ph.D. thesis, Department of Electrical Engineering, Stanford University, Palo Alto, CA.

Piatetsky-Shapiro, G., and Frawley, W.J.E. (1991) *Knowledge discovery in databases.* Cambridge, Mass.: MIT Press.

Prusinkiewicz, P. (1986) "Graphical Applications of L-systems to computer imagery," *Proc. Graphics Interface '86—Vision Interface '86*, 247-253.

Prusinkiewicz, P., and Hanan, J. (1989) *Lindenmayer systems, fractals, and plants.* New York: Springer-Verlag.

Prusinkiewicz, P., and Lindenmayer, A. (1989) *The algorithmic beauty of plants*. New York: Springer-Verlag.

Quinlan, J.R. (1986) "Induction of decision trees," *Machine Learning*, *1*, 81-106.

Quinlan, J.R., and Rivest, R.L. (1989) "Inferring decision trees using the minimum description length principle," *Information and Computation*, *80*, 227-248.

Restle, F. (1970) "Theory of serial pattern learning: structural trees," *Psychological Review*, *77*(6), 481-495.

Restle, F., and Brown, E.R. (1970) "Serial pattern learning," *Journal of Experimental Psychology*, *83*(1), 120-125.

Rissanen, J. (1978) "Modeling by shortest data description," *Automatica*, *14*, 465-471.

Rissanen, J.J. (1976) "Generalized Kraft inequality and arithmetic coding," *IBM Journal of Research and Development*, *20*, 198-203.

Rissanen, J.J., and Langdon, G.G. (1979) "Arithmetic coding," *IBM Journal of Research and Development*, *23*(2), 149-162.

Rosenbaum, D.A., Kenny, S.B., and Derr, M.A. (1983) "Hierarchical control of rapid movement sequences," *Journal of experimental psychology: Human perception and performance*, 9(1), 86-102.

Rubin, F. (1979) "Arithmetic stream coding using fixed precision registers," *IEEE Transactions on Information Theory*, *IT-25*(6), 672-675.

Salton, G. (1989) *Automatic Text Processing: the transformation, analysis and retrieval of information by computer*. Reading, Mass.: Addison Wesley.

Sapir, E. (1921) *Language*. New York: Harcourt, Brace & World.

Schlimmer, J.C., and Hermens, L.A. (1993) "Software agents: completing patterns and constructing user interfaces," *Journal of Artificial Intelligence Research*, *1*, 61-89.

Shannon, C.E. (1948) "A mathematical theory of communication," *Bell System Technical Journal*, *27*, 398-403.

Smith, A.R. (1978) "About the cover: reconfigurable machines," *Computer*, *11*(7), 3-4.

Smith, A.R. (1984) "Plants, fractals, and formal languages," *Computer Graphics*, *18*(3), 1-10.

Solomonoff, R. (1959) "A new method for discovering the grammars of phrase structure languages," *Information Processing*, 258-290.

Storer, J.A. (1977) "NP-completeness results concerning data compression," 234.

Storer, J.A. (1982) "Data compression via textual substitution," *Journal of the Association for Computing Machinery*, *29*(4), 928-951.

Szilard, A.L., and Quinton, R.E. (1979) "An interpretation for D0L systems by computer graphics," *The Science Terrapin*, *4*, 8-13.

Tarjan, R.E. (1975) "Efficiency of a good but not linear set union algorithm," *Journal of the Association for Computing Machinery*, *22*(2), 215-225.

Teahan, W.J., and Cleary, J.G. (1996) "The entropy of English using PPM-based models," *Proc. Data Compression Conference*, Snowbird, Utah, 53–62.

Thomas, S.W., McKie, J., Davies, S., Turkowski, K., Woods, J.A., and Orost, J.W. (1985) "Compress (version 4.0) program and documentation," *available from joe@petsd.uucp.*

von Koch, H. (1905) "Une méthode géométrique élémentaire pour l'étude de certaines questions de la théorie des courbes planes," *Acta Mathematica, 30*, 145-174.

Wallace, C.S., and Freeman, P.R. (1987) "Estimation and inference by compact coding," *Journal of the Royal Statistical Society (B)*, *49*, 240-265.

Wallace, C.S., and Patrick, J.D. (1993) "Coding decision trees," *Machine Learning, 11*, 7-22.

Williams, H., and Zobel, J. (1996) "Practical compression of nucleotide databases," *Proc. Proceedings of the Australian Computer Science Conference*, Melbourne, Australia.

Witten, I.H. (1979) "Approximate, non-deterministic modelling of behaviour sequences," *International Journal of General Systems*, *5*, 1-12.

Witten, I.H. (1981a) "Programming by example for the casual user: a case study," *Proc. Canadian Man-Computer Communication Conference*, Waterloo, Ontario, 105-113.

Witten, I.H. (1981b) "Some recent results on non-deterministic modelling of behaviour sequences," *Proc. Proc. Society for General Systems Research*, Toronto, Ontario, 265-274.

Witten, I.H. (1982) *Principles of computer speech*. London, England: Academic Press.

Witten, I.H., and Maulsby, D.L. (1991) "Evaluating programs formed by example: an informational heuristic," in *New results and new trends in computer science*, edited by H. Maurer, Berlin: Springer-Verlag, 388–402.

Witten, I.H., and Mo, D. (1993) "TELS: Learning text editing tasks from examples," in *Watch what I do: programming by demonstration*, edited by A. Cypher, Cambridge, Massachusetts: MIT Press, 182-203.

Witten, I.H., Moffat, A., and Bell, T.C. (1994) *Managing Gigabytes: compressing and indexing documents and images*. New York: Van Nostrand Reinhold.

Witten, I.H., Neal, R., and Cleary, J.G. (1987) "Arithmetic coding for data compression," *Communications of the Association for Computing Machinery*, *30*(6), 520-540.

Witten, I.H., Nevill-Manning, C.G., and Cunningham, S.J. (1996) "Building a digital library for computer science research: technical issues," *Proc. Australasian Computer Science Conference*, Melbourne, Australia, 534-542.

Wolff, J.G. (1975) "An algorithm for the segmentation of an artificial language analogue," *British Journal of Psychology*, 66(1), 79-90.

Wolff, J.G. (1977) "The discovery of segments in natural language," *British Journal of Psychology*, 68, 97-106.

Wolff, J.G. (1980) "Language acquisition and the discovery of phrase structure," *Language and Speech*, *23*(3), 255-269.

Wolff, J.G. (1982) "Language acquisition, data compression and generalization," *Language and Communication*, *2*(1), 57-89.

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

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

Zobel, J., Moffat, A., Wilkinson, R., and Sacks-Davis, R. (1995) "Efficient retrieval of partial documents," *Information Processing and Management*, *31*(3), 361-377.