Detecting Sequential Structure

Craig G. Nevill-Manning and Ian H. Witten

University of Waikato, Hamilton, New Zealand. {cgn, ihw}@waikato.ac.nz

Abstract. *Programming by demonstration requires detection and analysis of sequential patterns in a user's input, and the synthesis of an appropriate structural model that can be used for prediction. This paper describes* SEQUITUR*, a scheme for inducing a structural description of a sequence from a single example.* SEQUITUR *integrates several different inference techniques: identification of lexical subsequences or vocabulary elements, hierarchical structuring of such subsequences, identification of elements that have equivalent usage patterns, inference of programming constructs such as looping and branching, generalisation by unifying grammar rules, and the detection of procedural substructure., Although* SEQUITUR *operates with abstract sequences, a number of concrete illustrations are provided.*

Key Words. PBD, sequence learning, grammatical inference

Imperative programming involves the specification of a sequential series of elementary activities. Consequently, one might imagine programming-by-demonstration (PBD) to proceed by demonstrating a sequence of actions to an agent and having it pick up the sequence, predict its continuation, and execute the corresponding sequence of actions automatically. It seems that a core part of PBD is concerned with the inference of structural descriptions of sequences from examples, that is, the inference of intentional sequence descriptions from extensional ones.

However, a survey of actual PBD systems—say the ones documented in Cypher *et al.* (1993)—reveals a remarkable void in the area of sequence inference. Some that do attempt to infer sequential structure, like TELS (Witten and Mo, 1993), embody rather trivial models of sequence structure that break down in realistic situations. The inference and prediction of sequences is a key concern of text compression, but current compression algorithms, whether macro-based or statistically-based, attempt to capture only simple repetitive lexical properties of sequences (Bell *et al.*, 1990). Work on grammatical inference is concerned with abstracting more complex structure from sequences, but tends to be preoccupied with the problem of finding a grammar that covers several independent examples from a single source (Angluin and Smith, 1983, Berwick and Pilato, 1987), and the methods used do not transfer to the detection of structure in a single, undifferentiated, behaviour sequence. Finally, the small amount of machine learning research on the sequence inference problem has produced systems (like SPARC/E, Dietterich and Michalski, 1986) that do not prove useful in practice, or ones (like TDAG, Laird and Saul, 1994) that are naive compared with contemporary compression models.

SEQUITUR is a scheme for inducing a structural description for a sequence from a single example. The sequences are abstract ones, and in the context of PBD are intended to represent series of concrete actions. Lacking a source of actual PBD sequences to serve as test material, a number of other kinds of example sequences have been used: a large sample of ordinary English, program text, strings created from L-system grammars, and sequences of actions generated by a sorting program.

The first step is lexical: combining neighbouring individual terminal symbols to produce higher-level symbols. Unlike traditional lexical analysis, however, this is done in a recursive manner, leading to a potentially highly-structured, compact, description of the sequence in terms of a hierarchical decomposition into "rules." This yields a sort of recursively-structured vocabulary for the sequence. The next step is to generalise the token sequence so generated. This can be done in several different ways. One is to identify symbols that are equivalent in their usage patterns—ones that tend to be preceded and/or followed by the same strings. Another is to locate programming constructs such as loops and branches in the token sequence. A third is to generalise the hierarchical decomposition rules by seeking patterns that allow different rules to be unified. And finally, if the sequence is sufficiently noise-free, it may be possible to detect procedural structure in it, and even locate recursive procedures.

1 Hierarchically structured vocabulary

The simplest kind of sequential structure is the repetition of a string of symbols. In natural language, these strings include roots, affixes, words and phrases. In programming by demonstration, they represent sub-tasks within the overall task. Sequitur's first step is to identify these phrases and restate the sequence in terms of this new vocabulary.

Finding phrases. SEQUITUR's operation is best explained using an example. The sequence *abcdbc* contains the repeated sequence *bc*. Figure 1a shows how *bc* is replaced by a non-terminal symbol *A*, and a new rule for *A* is added to the grammar. When repeated sequences contain smaller repeated sequences, the rules form a hierarchy. If the sequence *abcdbc* is repeated, say in the sequence *abcdbceabcdbc*, a new rule is added to the grammar, and is expressed in terms of the existing rule *A* (Figure 1b). This hierarchy describes the original sequence more compactly.

	Sequence	Grammar

(a)	abcdbc	S ← aAdA
		A ← bc

(b)	abcdbceabcdbc	S ← BeB
		A ← bc
		B ← aAdA

Figure 1: Two simple sequences and the phrases extracted from them

Figure 2 shows the hierarchies from several sequences. Each bar represents a rule, and a character represents a terminal symbol. For example, Figure 2a represents the second sequence from Figure 1 (*abcdbceabcdbc*). The top line is rule *S*, consisting of a black bar for each instance of *B*. The next line shows how rule *B* consists of two instances of rule *A* (dark grey bars), and the bottom line shows the original sequence.

A similar hierarchy can be built for longer sequences. *Far from the Madding Crowd*[1] by Thomas Hardy consists of 768771 characters of English text. The grammar formed from this sequence has 26477 rules, the first one being 131416 symbols long. Figure 2b shows a short excerpt from the book, and illustrates how it is represented hierarchically (tildes represent spaces). The top row of black bars shows that the phrase *what may be called a small silver clock* is divided into words and phrases: *what, may be, called, a small, s, ilver, clock*. The word *called* is further divided into the root *call* and the suffix *ed*, and the phrase *may be* is divided into two words. Although the method frequently identifies words, it fails to keep *silver* as one word, and *a small* is divided into *a s* and *mall*.

Figure 2c shows the hierarchy for a portion of a 1487 line (39611 character) C program.[2] The for statement *for (code = 255; code >= 0; code– –)* is divided into *for (* and the initial assignment, the loop variable, comparison operator, 0 and the decrement operation, and finally the closing parenthesis. On the second row, the leftmost rule is split into *for (* followed by the loop variable *code*, then the assignment. On the third row, the reserved word *for* is separated from the opening parenthesis. The hierarchy fails to keep the loop variable as one rule, but instead associates the first character *c* with the preceding semicolon.

Greedy phrase formation. The grammars described so far have two important properties: no digram appears more than once, and every rule is used at least twice. The algorithm consists of rearranging the grammar to ensure that these invariants are maintained as each new symbol is added to rule *S*. If the digram uniqueness constraint is violated, a new rule is formed (as in Figure 1a). If a rule is only used once, it is superfluous and can be expanded and removed from the grammar. An implementation in C++ on a Sun SparcStation 10 processes sequences at a megabyte per minute.

Because constraints are re-satisfied after each symbol is seen, the algorithm never waits for more symbols

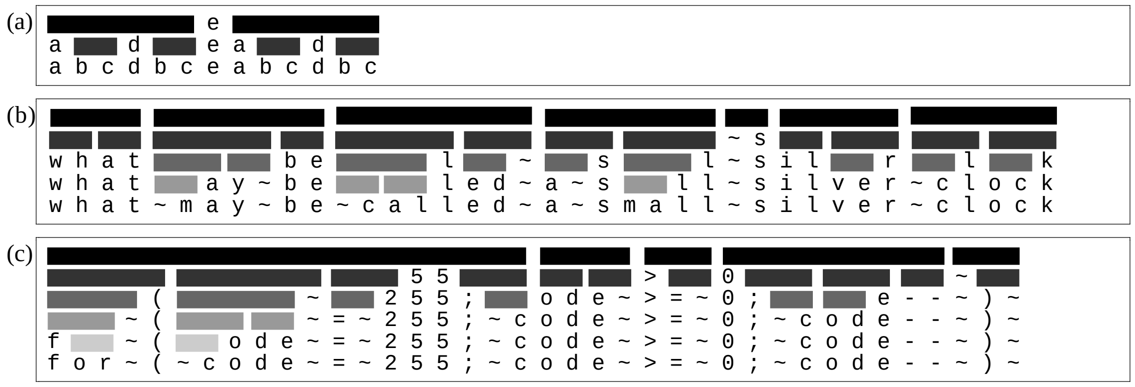

(a)
e
a d e a d
a b c d b c e a b c d b c
(b)
~ s
w h a t b e l ~ s l ~ s i l r l k
w h a t a y ~ b e l e d ~ a ~ s l l ~ s i l v e r ~ c l o c k
w h a t ~ m a y ~ b e ~ c a l l e d ~ a ~ s m a l l ~ s i l v e r ~ c l o c k
(c)
5 5 > 0 ~
(~ 2 5 5 ; o d e ~ > = ~ 0 ; e - - ~) ~
~ (~ = ~ 2 5 5 ; ~ c o d e ~ > = ~ 0 ; ~ c o d e - - ~) ~
f ~ (o d e ~ = ~ 2 5 5 ; ~ c o d e ~ > = ~ 0 ; ~ c o d e - - ~) ~
f o r ~ (~ c o d e ~ = ~ 2 5 5 ; ~ c o d e ~ > = ~ 0 ; ~ c o d e - - ~) ~

Figure 2: Hierarchies for: (a) the sequence *abcdbceabcdb*
(b) a phrase from *Far from the Madding Crowd* by Thomas Hardy
(c) excerpt from a C program

[1] file book1 of the Calgary compression corpus, available from ftp.cpsc.ucalgary.ca in directory pub/projects/text.compression

[2] file progc from the Calgary compression corpus.

(a)
```
bcdbcdbcefacef
```
(b)
```
S ← AABCacC
A ← Bd
B ← bc
C ← ef
```
(c)
```
S ← BBbAaA
A ← cef
B ← bcd
```
Figure 3: (a) A sequence , (b) processed left to right, (c) processed right to left

to complete a longer match: it forms rules based on the symbols it has seen and extends them later on if necessary. This on-line processing makes it suitable for interactive applications such as PBD. However, it also means that phrases are formed greedily—a rule is always extended if doing so will reduce the overall size of the grammar.

These greedy decisions make the algorithm sensitive to the presentation order of the sequence, and local decisions are often suboptimal in the long run. Figure 3a shows the sequence *bcdbcdbcefacef*, which when processed left-to-right yields the grammar in Figure 3b. This has four rules and seven symbols in rule *S*. However, when the original string is processed from right to left, the resulting grammar has only three rules and six symbols in rule S (Figure 3c).

In Figure 2, greedy processing is responsible for grouping the *s* in *small* with the preceding *a*, rather than with the rest of the word, and for grouping the letter *c* in the variable *code* with the preceding semicolon.

Generous processing. Finding the set of phrases that minimises the size of a sequence is NP-hard (Storer, 1988). Furthermore, finding this set requires the complete sequence at one time. In PBD, predictions must be made as symbols are available; it is not possible to choose the presentation order of the sequence.

Nevertheless, it is possible to improve the performance of the basic algorithm. If all predecessors of a non-terminal symbol have the same suffix, the rule headed by the non-terminal can be extended. For example, in Figure 3b the symbols that precede *C* are *B* and *c*. The last symbol in *B* is *c*, so by expanding *B*, both instances of *C* are preceded by *c*. Now *c* can be prepended to the contents of rule *C*, yielding the rule $C \rightarrow cef$. *B* is now used once only, and is expanded to produce the rule $A \rightarrow bcd$. This grammar is equivalent to Figure 3c.

Rules can also be extended forward if their successors share a common prefix. Extending rules in this way retrospectively corrects some bad decisions made by greedy phrase formation, so we refer to it as "generous processing." Generous processing is most effective on well-structured sequences. Consider the grammar in Figure 4a. When the first rule is evaluated recursively to a depth of three, the sequence shown in Figure 4b is produced. The grammar formed from this sequence is shown in 4c. Although it is more compact than the original sequence, it fails to capture the regularity of the source grammar.

Figure 4d shows the grammar obtained with generous processing. Not only is it smaller than Figure 4c, but it more closely resembles the original grammar. The

(a)
```
S ← S[+S]S[–S]S
S ← 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[+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)
```
S ← C E I K J K D f
A ← f [
B ← D A
C ← H F
D ← f ]
E ← B +
F ← B –
G ← C D
H ← A +
I ← G N
J ← E F
K ← E G
L ← F G
M ← H I D J K N
N ← J L J
```
(d)
```
S ← B F A G A
A ← B ] B
B ← D F C G C
C ← D ] D
D ← f F E G E
E ← f ] f
F ← [ +
G ← [ –
```
(e)
```
S ← B [ + B ] B [ – B ] B
B ← D [ + D ] D [ – D ] D
D ← f [ + f ] f [ – f ] f
```
Figure 4: (a) a recursive grammar,
(b) part of the sequence produced by (a),
(c) the grammar induced from (b),
(d) the grammar induced with generous parsing,
(e) 4d with rules *B*, *C*, *E*, *F* and *G* expanded

source grammar could be compressed by making a rule for the repeating *S]S* pattern, and this is what rules *A*, *C* and *E* in Figure 4d represent. If rules *A*, *C*, *E*, *F* and *G* are expanded and removed, the result is Figure 5e, which is a non-recursive version of the original grammar. This grammar is an example of an L-system, and is discussed further in the section on recursion.

Figure 5 shows the hierarchies for the sequences of Figures 2a and 2b when generous processing is applied to the entire sequence. In Figure 2a the word *silver* is now one rule, and *a small* is divided correctly into *a* and *small*. The word *called* is divided differently due to the appearance of *walled* later in the sequence, but the suffix *ed* remains. The entire grammar contains 20% more complete words than the grammar without generous processing. In Figure 2b, the complete for statement is identified as a repeating pattern, and the variable *code* appears as a single group. Both these hierarchies are more plausible descriptions of the sequence structure than the hierarchies without generous processing.

Background knowledge. The algorithm described so far does not take advantage of any domain-specific knowledge to enhance its performance. Certain kinds of knowledge can be used in a fairly simple way. For example, the brackets in Figure 4b are correctly nested, and this information can be used to construct the correct grammar without the extra overhead of generous processing.

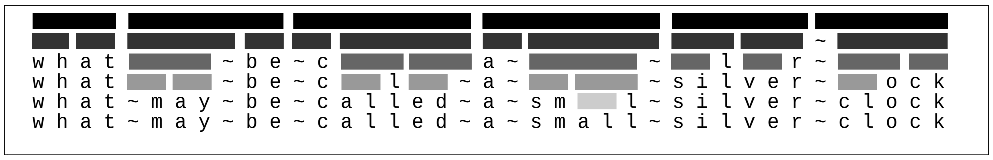
When a new digram is formed, the grammar is searched for an identical one. If the search is successful, a new rule is formed. Otherwise, no further processing occurs, because the digram uniqueness constraint is not violated. Background knowledge can be used to influence the success of the digram search, and therefore whether a particular rule is formed. In the bracket matching example, if the sequence represented by the digram is incorrectly nested the search returns false, and a rule is not formed for that digram. This forces the greedy algorithm to wait for further symbols before forming rules, and avoids retrospective modifications to the grammar, resulting in greater efficiency. Background knowledge from other domains can be added in a similar way, e.g. forbidding rules to cross white space in English text.

Compression. It is interesting to look at SEQUITUR's performance as a data compression technique. Occam's razor suggests that simple theories should be preferred over more complex ones. In machine learning, this is formalised by the minimum description length (MDL) principle, which prefers theories that minimise the sum of the size of the theory and the size of the observations given the theory. SEQUITUR's algorithm can be viewed as compressing the sequence by removing repetitions, deleting infrequent rules, and extending rules retrospectively.

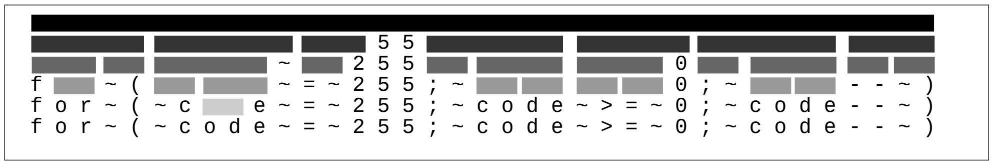
A key issue in the application of MDL is the language in which the theory and the observations are expressed. For two theories to be fairly compared, the language should be as efficient as possible. Treating SEQUITUR as a compression scheme opens up the possibility of using existing compression techniques as efficiency benchmarks.

Representing the grammar textually does not yield any compression for non-trivial sequences, because non-terminals need to be expressed as multi-digit numbers when the number of rules grow. Instead of text, the grammar can be encoded using a statistical model and an arithmetic coder, which encodes frequent symbols with short codes and uncommon symbols with long ones. *Far from the Madding Crowd* is 768771 symbols long, but encoding the grammar in this way results in a file size of 327458 bytes; 42.6% of its original size. UNIX COMPRESS compresses the book to 332056 bytes, or 43.2%, but more sophisticated compression techniques perform even better, for example GZIP (40.1%).

In the grammar representation, we encode the theory (all the rules except S) separately from the observations given the theory (rule S). It is possible to improve compression by encoding the theory (rules) and observations (sequence) simultaneously, and have the decoder build the rules adaptively. This method operates as follows: the first time a rule is

(a)


w h a t ~ b e ~ c a ~ ~ l r ~
w h a t ~ b e ~ c l ~ a ~ ~ s i l v e r ~ o c k
w h a t ~ m a y ~ b e ~ c a l l e d ~ a ~ s m l ~ s i l v e r ~ c l o c k
w h a t ~ m a y ~ b e ~ c a l l e d ~ a ~ s m a l l ~ s i l v e r ~ c l o c k

(b)


5 5
~ 2 5 5 0
f ~ (~ = ~ 2 5 5 ; ~ 0 ; ~ - - ~)
f o r ~ (~ c e ~ = ~ 2 5 5 ; ~ c o d e ~ > = ~ 0 ; ~ c o d e - - ~)
f o r ~ (~ c o d e ~ = ~ 2 5 5 ; ~ c o d e ~ > = ~ 0 ; ~ c o d e - - ~)

Figure 5: (a) The hierarchy which results from applying generous processing to the sequence in Figure 2b. (b) Figure 2c processed as in 5a.

used, its contents are encoded. The second time it is used, it is sufficient to encode a pointer to the first occurrence. At this juncture the decoder forms a new rule, and each time the rule occurs again, it can be referenced with the appropriate non-terminal symbol. This scheme compresses *Far from the Madding Crowd* to 35.3% of its original size, outperforming all other dictionary compression schemes—at the expense of longer computation time. This compression performance is consistent across the whole Calgary compression corpus, which includes text, binary data and images (Nevill-Manning *et al.*, 1994).

This example demonstrates that finding an efficient representation is not a straightforward task. The textual representation implies that the structure that the algorithm finds is much worse than no structure at all. The probability-based encoding indicates that it captures some structure, but the adaptive scheme shows it to be better than the theories found by any other dictionary compression scheme. The implication is two-fold: finding an efficient representation for the purpose of applying MDL is difficult, and adaptive transmission of the model along with the sequence is a useful approach in finding efficient representations.

2 Analysing branching structure

The hierarchical decomposition produced by the algorithm described so far has three shortcomings as a description of the structure of the sequence:

- It describes the vocabulary of the sequence, but fails to capture any non-linear structure, such as loops or branches.
- It is expressed as a grammar, but can only produce one sentence—the original sequence.
- It has limited predictive power. While it can predict the completion of a partially matched rule, it cannot predict the sequence of symbols in the first rule.

The next step involves generalising the grammar to make it more descriptive, more productive, and more predictive. For the discussion below, it is helpful to represent the sequence as a transition network by creating a state for each unique symbol in the first rule and inserting transitions between states whose symbols are adjacent in the sequence (Figure 6). This network can be traversed to reproduce the original sequence, but can also produce many other sequences. The transition network is too general, because all context information is forgotten when a transition is made, and considerable extra information is required to reproduce the original sequence. The true structure of the sequence is likely to be a compromise between the grammar and the transition network, a compromise that minimises the size of the structure and the extra information required to recreate the sequence given the structure.

The goal of the generalisation is not to capture every conceivable structure that may be present in an arbitrary sequence, but instead to recognise certain structures that are likely to be produced by the particular source process. For grammar-based sequences, we look for subsequences that occur in the same contexts and infer that they are equivalent. For the purposes of programming by demonstration, where the source is (presumably) a program, the likely structures are branches, loops, procedure calls and recursion. These five structures are discussed in turn.

Equivalent symbols. If two symbols often occur in the same contexts, the grammar can be generalised and simplified by treating them as the same. That is, if the set of the predecessors of one symbol is similar to the set of predecessors of the other, then the symbols are deemed equivalent.

(a)
```
S ← ...DE...DF...HE...HF...
D ← dd
H ← hh
E ← ee
F ← ff
```

(b)
```
S ← ...D'...D'...H'...H'...
D'← ddJ
H'← hhJ
J ← ee
J ← ff
```

Figure 7: Merging equivalent symbols *E* and *F*

In Figure 6b, nodes *E* and *F* are both preceded by nodes *D* and *H*. Figure 7a shows part of the grammar which gives rise to rule S in Figure 6a. Figure 7b shows how *E* and *F* are generalised to form a new rule *G*, reducing the number of symbols in the grammar from 16 to 14. To reproduce the original sequence, it is necessary to supply extra information to select the correct right hand side for *G*. In this case, one bit has to be supplied each time *D'* or *H'* is used, so four bits

(a)
```
S ← ABDEACDFGCHEHFI
```

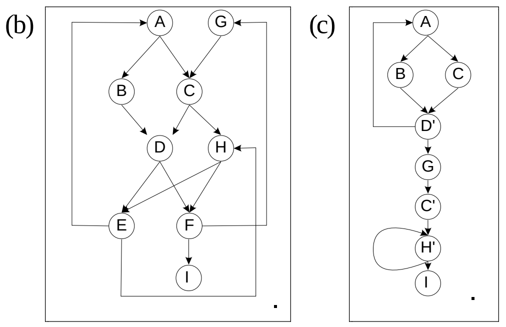


Figure 6: (a) the first rule of a grammar, (b) the transition network representing the sequence, (c) the transition network after recognition of a branch and equivalent symbols.

should be added to the size of the grammar.

We would expect to find such symbols when they are alternative right-hand sides for the same rule in a non-deterministic grammar, e.g. they are both verbs in a sequence of English text, or variables in a program.

Branches. Nodes *A*, *B*, *C* and *D* in Figure 6b typify a branch structure that could be generated by an *if...then...else* or *case* construct in a program. Because *B* and *C* share only one common predecessor, combining them into one node in the same way as *E* and *F* does not reduce the size of the grammar. In the absence of *G* and *H*, the structure would prove useful—*D* follows *A* after an intervening symbol, so *D* can be predicted, making it less costly to encode. However, the presence of *G* and *H* in this example makes some further processing necessary.

The transition network implies that *D* may follow *G* with the intervening node *C*. Similarly, *H* appears to follow *A*. However, recall that the network was formed from the sequence in Figure 6a, where, in fact, neither *GCD* nor *ACH* occur. *C* is used in two different structures: *ACD* and *GCH*, but appears in the network as one node. To correct this, *C* is cloned to produce *C'*, resulting in Figure 6c, which illustrates how the *ABCD* branch has been isolated from *H* and *G*.

After recognising the equivalent symbols *E* and *F* and the *ABCD* branch, the network has one fewer nodes and four fewer transitions. Furthermore, to reproduce the original sequence from Figure 6b takes 12 bits, while to reproduce it from Figure 6c takes only 9 bits.

Loops. There are two ways in which loops might appear in the transition network. First, if the sequence repeats exactly, a rule will be formed for all of the symbols in the loop, and there will be a transition from a node to itself.

Second, if the loop includes branches, then there will be no exact repetition, but there will be a transition from the last node in the loop to a previously visited node, which is the first node in the loop. All nodes reachable from the first node without going through the last node belong within the loop. Borrowing from structured programming constraints, we require that loops do not overlap, which means that there must be no transitions from states in the loop to states outside the loop or vice versa. However, the first node may have incoming transitions from other nodes, and the last node may have outgoing transitions to other nodes.

For example, in Figure 6c there are two candidate loops: from *D'* to *A*, and from *H'* to itself. Neither of these violate the constraint on internal nodes, so no further processing is required. These loops were not possible in Figure 6b—they only appear after the equivalent symbols and branch have been recognised.

3 Procedures and recursion

Procedures are an essential part of the structure of real programs, and SEQUITUR is capable of recognising procedures in a sequence, including recursive ones. Unlike the recognition algorithms described so far, the techniques in this section require a very regular, noise-free sequence, such as an execution trace of an actual program.

Procedure calls. If a procedure is repeated verbatim, or the repetitions are generalised to the same sequence, then a rule will be formed for it. In this case, the rule is exactly equivalent to the procedure: using the non-terminal symbol that heads the rule is just the same as calling the procedure.

However, if the body of the procedure is not identical in every invocation, it must be recognised in some other way. As with a loop, there must be no direct transitions from states in the procedure to states outside the loop, or *vice versa*. It is possible to recognise procedures by searching the transition network for such a group of nodes.

The search traverses the network, treating each node as a candidate for the first node in a procedure, and searches the network for a descendant *d*, such that the set of nodes reachable from the first node without going through *d* are only connected to other nodes in the set. For a network with *n* nodes, this search takes $O(n^2)$ time.

The problem with this algorithm is that as the transition network is cyclic, a node's descendant may also be its ancestor. The solution is to identify loops when the automaton is created, making use of the order in which states are created. If an edge is added which connects a state to one of its ancestors, this edge is marked as a loop. Ignoring the loops makes the graph acyclic, and the procedure definition above becomes useful. This algorithm is described more fully in Nevill-Manning (1993).

Recursion. Recursion can appear in two different ways: as sub-graphs in the transition network, or as similar rules in the grammar.

If the procedure does not repeat verbatim, and the algorithm described in the last section is applied, recursion results in a transition from a node within the procedure to the node that begins it. Nevill-Manning (1993) describes how this kind of recursion can be recognised.

If the procedure does repeat exactly, recursion appears as similar rules in the grammar. As noted previously, Figure 5a is a recursive grammar. Evaluating it and performing generous phrase recognition on the resulting sequences yields the non-recursive grammar in Figure 5d, which is reproduced in a more readable form in Figure 5e. All of the rules in the non-recursive grammar are similar; in fact, they

(a)

f = draw a line forward
– = turn left
+ = turn right
[= save state
] = restore state

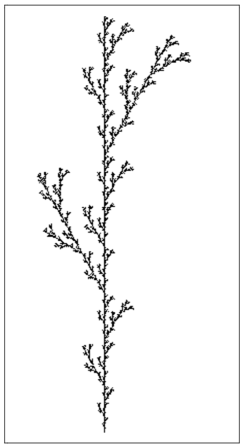
(b)


Figure 8:
(a) interpretation of turtle commands in Figure 5b,
(b) plant represented by the sequence in Figure 5b.

unify using standard pattern matching procedure. Unifying the bodies of the rules yields *B≡D≡f*, and unifying the heads of the rules yields *S≡B≡D*. Replacing *B* and *D* with *S* produces the recursive rule *S[+S]S[–S]S*, and unifying *S* with *f* results in the new rule $S \rightarrow f$. The unification can be performed by a short Prolog program and yields the original recursive grammar in Figure 5a.

The original grammar is an example of a Lindenmayer system, or L-system (Prusinkiewicz, 1990). Interpreting the sequence in Figure 5b as turtle commands (Figure 8a), produces the plant form in Figure 8. The procedure just described takes the primitive graphics operation used to draw the plant, and produces the correct description of the rules governing its growth.

4 Putting it all together

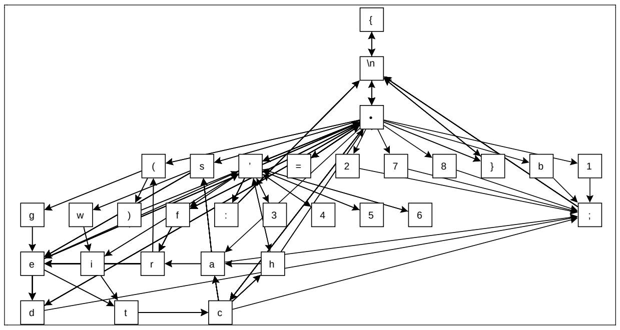
We have described techniques for efficiently forming a vocabulary from a sequence, and recognising equivalent symbols, branches, loops, procedure calls and recursion in a sequence. Now let us put these techniques together to make inferences.

SEQUITUR implements each of the generalisation techniques described so far. Each technique is applied incrementally, rather than as a post-processing step, so that maximum benefit can be gained in terms of on-line prediction and explanation. As each new symbol is observed, the vocabulary is updated, generous processing is performed, and then the grammar is examined to see whether it is possible to apply any of the transformations. Because only parts related to the last symbol in the sequence will have

(a)

```
{
  switch (getchar()) {
  case 'f':value = 1;
  case 'h':mark  = 2;
  case 'e':mark  = 8;
  case 'i':value = 7;
  }

 switch (n) {
  case 4 : mark  = 45;
  case 30: value = 6;
  case 3 : value = 38;
  case 5 : mark  = 3;
  }
}
```

(c)


(b)

```
S ← { I H g e G C ( ) J ' f E
1 B h D 2 B e D 8 B i E 7 T F H
n J 4 R 4 5 P 0 N O 6 P Q O 3 8
M 5 R 3 T U
A ← ' :            ':
B ← M '            ;\n  case '
C ← a r            ar
D ← A L            ':mark  =
E ← A O            ':value =
F ← \n             \n
G ← t c h          tch
H ← s w i G   (    switch (
I ← F              \n
J ← )   { K        ) {\n  case
K ← I c a s e      \n  case
L ← m C k   S      mark  =
M ← ; K            ;\n  case
N ← :              :
O ← v a l u e S    value =
P ← M 3            ;\n  case 3
Q ←   N             :
R ← Q L             : mark  =
S ←   =             =
T ← ; I U          ;\n  }\n
U ← } \n           }\n
```

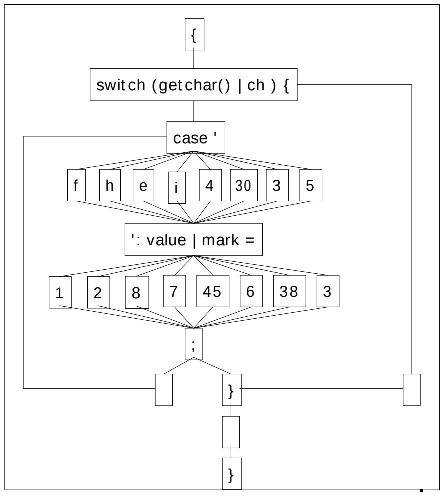
(d)


Figure 9: (a) fragment of C code, (b) the vocabulary, (c) structure recognition without vocabulary (d) vocabulary and structure recognition

changed, this check can be performed efficiently.

For generality, the heuristics used to recognise structure in the sequence do not involve arbitrary constants: a conscious design decision was made to use only the numbers 0 and 1 in any computations. This provides a simple system which is not tailored toward any particular domain, but to which domain-dependent heuristics can be added.

For example, Figure 9a shows a portion of a C program. The vocabulary derived from this sequence is shown in Figure 9b. It captures the reserved words, variable names and delimiters, but by itself does not recognise the overall structure because of the different variable names, switch values, case labels and values. Figure 9c shows the structure recognition techniques without the vocabulary formation part. As it must find structure between the individual characters, it is not able to give particularly useful insights. However, when the two parts are combined, they produce the structure in Figure 9d, which captures much of the desired structure.

The lexical analysis part of SEQUITUR is robust, and has been applied to a wide variety of sequences. The structural inference part is still experimental, and while it is successful on several test examples, is still being refined to deal with more complex and noisy sequences. By incorporating a number of techniques for recognising structure, SEQUITUR is capable of modelling a range of sequences, from noise-free, highly regular L-system sequences, to noisy action sequences containing branches, loops and procedure calls. It is also capable of inferring the structure of natural language, program code, and semi-structured textual databases like bibliographies. The crucial test, of course, will be to apply it to PBD sequences. To this end, it is planned to integrate SEQUITUR into a PBD system to complement Dave Maulsby's CIMA data description learner.

Acknowledgements

Thanks to Dave Maulsby, who proposed the idea of forming a hierarchical vocabulary, and to Przemyslaw Prusinkiewicz, who suggested the L-system application. This work is supported by the New Zealand Foundation for Research in Science and Technology.

5 References

Angluin, D. and Smith, C.H., "Inductive inference: theory and methods" (1983) Computing Surveys 15(3), 237-269.

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

Berwick, R.C., Pilato, S. (1987), Learning syntax by automata induction, *Machine Learning* 2(1), 9-38.

Cypher, A. (Editor) (1993) *Watch what I do: programming by demonstration*, MIT Press, Cambridge, Massachusetts.

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

Laird, P. and Saul, R (1994), "Discrete Sequence Prediction and its Applications", *Machine Learning* 15(1), 43-68.

Nevill-Manning, C.G. (1993), "Programming by Demonstration", *New Zealand Journal of Computing* 4(2), 15-24.

Nevill-Manning, C.G., Witten, I.H., & Maulsby, D.L., (1994) "Compression by Induction of Hierarchical Grammars" *Proceedings of the Data Compression Conference 1994*, IEEE Computer Society Press.

Prusinkiewicz, P. & Lindenmayer, A. (1990) *The algorithmic beauty of plants*, Springer-Verlag.

Storer, J.A.. (1988) *Data Compression—methods and theory*, Computer Science Press.

Witten, I.H., Mo, D. (1993), "TELS: Learning text editing tasks from examples", In Cypher (1993), 182-203.