CABIOS INVITED REVIEW

Vol. 13 no. 4 1997
Pages 333–344

Linguistic approaches to biological sequences

David B.Searls

Abstract

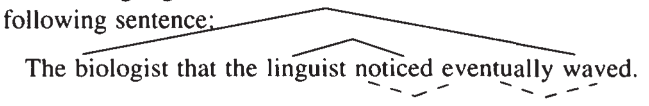
Biologists have long made use of linguistic metaphors in describing and naming cellular processes involving nucleic acid and protein sequences. Indeed, it is very natural to view the genetic 'text' and its sequential transliterations in these terms. However, a metaphor is not a tool, and it is necessary to ask whether the techniques used in analyzing other kinds of languages, such as human and computer languages, can in fact be of any use in tackling problems in molecular biology. This paper reviews the work of the author and others in applying the methods of computational linguistics to biological sequences.

Introduction

Only in recent years has the long-standing metaphor of DNA as language been rigorously examined, from both theoretical and practical perspectives. This metaphor arose early in molecular biology, when nucleic acids were recognized as strings of nucleotide bases comprising the famous four-letter alphabet. The complementarity of this alphabet—with the letter G always pairing with C across the double helix, and T with A—was seen to permit the faithful replication of DNA molecules. The metaphor was strengthened when the relationship of nucleic acids to proteins was elucidated; the so-called central dogma recognized the fundamental two-step process that first transcribes a subsequence of DNA into RNA, and then translates successive triplets of RNA bases to amino acids according to the mapping called the genetic code. Later, it was discovered that the RNAs that are translated, called messenger or mRNAs, are further processed in higher organisms so as to splice out intervening untranslated regions, called introns, leaving only the exons of the ultimate transcript. These biological transformations can be seen as analogous at a number of levels to mechanisms of processing other kinds of languages, such as natural languages and computer languages, particularly in the approaches pioneered by Noam Chomsky in his revolutionary work in the field of linguistics (Chomsky, 1957).

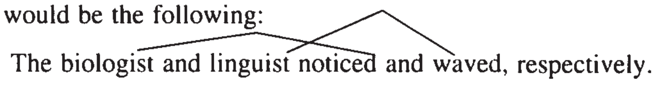
Bioinformatics Group, SmithKline Beecham Pharmaceuticals, 709 Swedeland Road, PO Box 1539, King of Prussia, PA 19406, USA

A comprehensive review of this branch of linguistics is well beyond the scope of this review, but a few examples may serve to highlight some of the relevant issues. Consider the following sentence:


The biologist that the linguist noticed eventually waved.

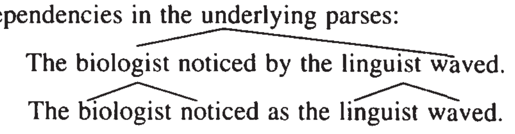
The lines above the sentence serve to connect each noun with its corresponding verb, i.e. it was the linguist who noticed and the biologist who waved; note that the relative clause thus separates remote features of the sentence that are meaningfully related to each other. This sort of 'action at a distance' is very characteristic of natural language and provides much of the motivation for attempts to account for such structure using rule-based systems called grammars. It will be seen that grammars are capable of neatly generating just such hierarchical descriptions of the components of sentences. They also capture another important feature of natural languages, syntactic ambiguity, by which alternative structural interpretations or parses are possible. In this example, the dashed lines below the sentence highlight an ambiguity surrounding the referent of the adverb 'eventually': was the biologist slow to wave to the linguist, or was the linguist slow to notice the biologist?

The dependencies between nouns and verbs in the sentence above are nested, insofar as any number of relative clauses could theoretically be inserted within each other to create tiers of such dependencies. Somewhat less common in English are crossing dependencies; an artifical example would be the following:


The biologist and linguist noticed and waved, respectively.

As will be seen in the next section, whether a language entails strictly nested or crossing dependencies has significant consequences for the types of grammars required, with greater complexity attaching to crossing dependencies.

It is noteworthy that very subtle lexical changes (i.e. at the level of the words of a sentence) can have drastic effects on the parse. In the following pair of sentences, changing the preposition 'by' to the conjunction 'as' completely rearranges the dependencies in the underlying parses:


The biologist noticed by the linguist waved.

The biologist noticed as the linguist waved.

To appreciate the relevance of these linguistic issues to biological sequences, consider the various kinds of structure found in the latter. By analogy with sentences, it might be claimed that genes have a hierarchical syntactic structure, and indeed it is natural to draw tree-structured descriptions of gene structure (Searls, 1993). Just as sentences exhibit distant dependencies, e.g. between related nouns and verbs, so do genes exhibit such dependencies, e.g. between splice donors and acceptors. In this case, syntactic ambiguity would be a reflection of the phenomenon of alternative splicing; it is well known that subtle changes in 'lexical' signals in mRNA can result in patterns remarkably like the example sentences immediately above. Similar ambiguities can be found in the apparatus regulating gene expression.

Another analogy to linguistics arises from the fact that biological strings fold up in three-dimensional space, in such a way that distant parts of those strings interact with and thus create dependencies with each other. In RNA, the most obvious manifestiation of such dependencies is base-pairing interaction. As will be seen in the next section, such dependencies are very naturally expressed via grammars. Again, the notion of syntactic ambiguity has a biological manifestation, in this case the phenomenon of alternative secondary structure (Searls, 1993). Both nested and crossing dependencies are observed, for example in antiparallel and parallel strands in proteins (Mamitsuka and Abe, 1994).

The author has discussed a number of other correspondences between linguistic notions and biological phenomena (Searls, 1992, 1993), and has also built tools for pattern-matching search based on linguistic parsers (Searls and Noordewier, 1991; Searls and Dong, 1993). The latter have extended to the problem of detection of genes in genomic DNA sequences (Dong and Searls, 1994). These three topics will be discussed in the following sections of this review. Readers with interest and background in the field of formal language theory, the bare rudiments of which are introduced in the next section, may wish to refer to developments in that arena that have been inspired by the biological domain (Searls, 1989, 1995a,b, 1996); this work is also summarized in a recent review (Searls, 1997).

Formal language theory and biological sequences

The processes of transcription and translation from strings of one kind to strings of a different kind by processive cellular machinery suggested to some the behavior of certain kinds of finite state automata (FSAs). FSAs are simple models of computation, in fact lying at the foundation of computer science, that comprise directed graphs with labeled transitions among states. By traversing such a graph from state to state and emitting the lexical/alphabetic labels upon each transition, a variety of strings can be generated constituting a language. Brendel and co-workers exhibited FSAs that, on paper, modeled the processive processes implied by the central dogma (Brendel and Busse, 1984), and in fact such automata are implicit in numerous software packages that perform sequence analysis. For that matter, such packages often provide capabilities for pattern-matching search for short substrings of interest, through the use of regular expressions (such as the UNIX grep utility); in formal language theory, the most basic form of regular expressions corresponds to FSAs in terms of expressive power. Note that FSAs have a dual nature: they can be seen either as generators of languages, or as recognizers. A variation on this architecture, called a finite state transducer, can accomplish both at the same time; by labeling each transition with both an input and an output, a true transformation of one string to another can be accomplished, even more effectively modeling the process of gene expression.

Given the felicity of this correspondence between FSAs and biology, the question thus naturally arose as to exactly where DNA resides on the Chomsky hierarchy of formal languages. This hierarchy classifies the linguistic complexity of languages (viewed purely as sets of strings) and relates them to species of automata required to recognize or generate them. Similarly, each level of the hierarchy corresponds to a particular type of grammar, or rule-based system for formally specifying languages. It happens that regular languages, those that can be specified by FSAs and/or by regular expressions (a kind of grammar), occupy the lowest level of the Chomsky hierarchy. However, there are certain languages, such as the set of palindromes (strings that read the same forward and backward), that cannot be expressed by any FSA or pure regular expression. Fundamentally, this is because FSAs and regular expressions have no notion of memory that would permit them to describe arbitrary numbers of dependencies, other than strictly local ones.

Languages such as palindromes fall on the next level of the Chomsky hierarchy, that of context-free languages, which require a pushdown (stack) automaton and a more powerful form of grammar consisting of rewrite rules. In this system, the alphabet is augmented with a set of temporary, place-holding symbols or non-terminals, and a string is derived by starting from some such symbol and applying the rewrite rules to non-terminals in the developing string until they are all eventually replaced by a terminal string of alphabetic elements only. Thus, for example, the grammar consisting of the rules

$$S \rightarrow \mathrm{g}S\mathrm{c} \quad S \rightarrow \mathrm{c}S\mathrm{g} \quad S \rightarrow \mathrm{a}S\mathrm{t} \quad S \rightarrow \mathrm{t}S\mathrm{a} \quad S \rightarrow \epsilon$$

specifies a set of DNA molecules. The S is a non-terminal, while the lower-case letters are terminal bases; the ϵ stands for the empty string, and effectively serves to erase an S. This grammar specifies an infinite number of strings via derivations like the following, in which the S is repeatedly rewritten using the above rules:

$$S \Rightarrow \mathrm{g}S\mathrm{c} \Rightarrow \mathrm{gc}S\mathrm{gc} \Rightarrow \mathrm{gca}S\mathrm{tgc} \Rightarrow \mathrm{gcat}S\mathrm{atgc} \Rightarrow \mathrm{gcatatgc}$$

The reader may confirm, by drawing lines connecting those nucleotides that were derived by the same rule invocation, that this grammar creates strictly nested dependencies, and in fact nested dependencies are the only sort possible from a context-free grammar. As noted, the resulting language lies outside the regular languages, though any regular language can be expressed by a grammar like the one above. Nevertheless, there are yet other languages that elude even the context-free formalism, in essence because they entail arbitrarily many crossing dependencies. Many of these can be captured with a context-sensitive grammar: one that can have more than one symbol on the left-hand side of a rule, as long as the number of symbols does not exceed that on the right-hand side. Relax this latter restriction, and the resulting unrestricted grammars are able to specify any language that can be recognized by a Turing machine—the automaton corresponding to this class of languages. Thus, ascending the Chomsky hierarchy appears to require more and more computational power to accomplish general-purpose recognition or parsing of strings, i.e. to determine their membership in a language specified by some grammar.

Much of the author's work has been concerned with the formal characterization of the language of nucleic acids, in terms of its position in the Chomsky hierarchy and related mathematical questions. For example, the grammar given above specifies, in an idealized way, an important class of biological sequences, those exhibiting dyad symmetry. The strings of this language constitute, in fact, a variety of biological palindrome in which a string reads the same on one strand of DNA as it does on the opposite strand reading backward (the so-called reverse complement). The resulting inverted repeats may also allow a single strand to fold up and base pair with itself instead of with its opposite strand, in a structure called a hairpin or, when there are a number of unpaired bases at the turn, a stem-and-loop. Such secondary structure is crucially important in, for example, the function of structural RNAs in the cell. To the extent that the capacity for secondary structure may be said to be a necessary feature of the language of nucleic acids, we may infer that they lie above the regular languages on the Chomsky hierarchy, since at least a context-free grammar is required to specify such structure in the general case. In fact, the most general grammar of orthodox secondary structure can be shown to consist of the grammar above, augmented with one additional rule: $S \rightarrow SS$. This rule, which simply doubles the start symbol, is sufficient to permit arbitrarily branching secondary structures (Searls, 1989, 1993). It also has the effect of introducing the potential for ambiguity, in that there are sequences that can be parsed by more than one path [the reader may wish to verify this by trying parses with this grammar on sequences that are double inverted repeats, such as gatcgatc; these can theoretically form hairpins, dumbbells, or even cruciform structures, each corresponding to a different parse (Searls, 1992)]; happily, these correspond to alternative secondary structures, providing another useful analogy between linguistic theory and biological reality (Searls, 1992, 1993). Stochastic forms of such grammars, i.e. where probabilities are attached to each rule, have proven very successful in machine-learning approaches to characterizing recurring secondary structures, as in the work of Haussler's group (Grate *et al.*, 1994; Sakakibara *et al.*, 1994).

Still other biological phenomena indicate that the language of nucleic acids may be beyond context free as well. Pseudoknots (see below) are forms of secondary structure that require context-sensitive expression in the general case, as do repeated sequences that are common in DNA and are arguably necessary features of the language. Closure properties and decidability results suggest that evolution may be a powerful force toward increasing linguistic complexity, and that DNA may be inherently ambiguous, non-linear and non-deterministic (all formally defined properties from language theory). These results are described in Searls (1992), and presented in greater mathematical detail in Searls (1993, 1997); in the remainder of this review, we will address the more pragmatic problem of recognition or parsing of such linguistic features in DNA.

In speaking of higher-order pattern recognition in biological sequences, this linguistic point of view offers one way to categorize the problem space and suggests established tools for exploring it. For example, the existence of features in the domain that are at least context free implies that simple regular expression search will not suffice. Even in advance of any formal linguistic analysis, however, this problem was recognized, and a number of software packages have offered enhanced regular expressions with 'escapes' to specify, for example, inverted and direct repeats (Staden, 1990). Some early programs for recognizing patterns of motifs in proteins were based on extended regular expressions (Lathrop *et al.*, 1987). Whether or not these offer sufficient expressive power for a greater range of biological phenomena, the use of more sophisticated grammars and parsers can be seen to have other advantages. Perhaps most notable among these is the ability to create modular, hierarchical rule sets, with detail always presented at the appropriate level (and, of course, a well-studied formal foundation). Parsers also typically return tree-structured descriptions of the history of a derivation, called parse trees, which are not only clear depictions of the presumed structure under study, but are also appropriate data structures for further computational analysis. [One worker, in fact, has studied genetic structures from the point of view of transformational grammar, which entails operations on a presumed canonical parse tree in order to create variations in surface structure typical of natural language (Collado-Vides, 1989, 1992, 1993; Rosenblueth *et al.*, 1996); this work, and that of others (Bentolila, 1996), has primarily dealt with regulatory regions.] The author has shown that the parse trees for grammars depicting secondary structure in fact physically resemble that structure—a trait considered desirable in grammars purporting to model the (less literal) structures of natural language (Searls, 1992).

The linguistic metaphor has suggested other approaches to this domain besides one based in generative grammars and the tradition of Chomsky (e.g. Pesole *et al.*, 1996). A number of workers, notably Trifonov and his colleagues (Trifonov, 1988, 1989, 1993), have borrowed in part from speech processing and even cryptanalytic methodologies in their analyses of biological sequences, in order, for example, to identify and characterize 'words' or short recurring substrings of DNA that are likely to have a functional significance. In fact, such significant words, often recognition and/or binding sites for other molecules which act upon the DNA, are an important concern in systems attempting to recognize the higher-order structures that contain them. The difficulty arises when such words are imperfectly specified, i.e. when an exact instance of a word is not necessary, but rather a whole array of 'synonyms' will serve, more or less, the same purpose.

Although it may be possible to describe a canonical consensus sequence for such a feature, for purposes of recognition it is necessary to allow for imperfect matching, and in general for some notion of assessing the 'cost' of a match. The most obvious approach is simply to allow for, and count, mismatches at the level of individual bases (so that the cost is, in effect, the Hamming distance). Biologically, this would correspond to a base substitution. However, mutations involving insertions and deletions are also common; the so-called edit distance between words (or, more often, entire strings) accounts for the total number of such operations needed to transform one string into another. Algorithms for efficiently finding such a minimum edit distance alignment between two strings (allowing gaps) have historically played an important role in computational biology (Waterman, 1988). Again at the level of words, a more sophisticated variation on counting mismatches has been to compile frequency tables for the number of times each base occurs at each position in a number of exemplars; this frequency table is converted by a variety of techniques to a weight matrix which is used to assess the cost of a match over the whole word (Staden, 1984). Yet more sophisticated methods have been applied involving hidden Markov models, connectionist techniques, and the like, demonstrating that 'higher-order' structural analysis is important even at the level of word recognition in biology (reviewed by Gelfand, 1995).

Thus, what we have termed the linguistic view of biological sequences involves challenges for both word recognition and syntactic analysis. We will describe practical approaches to both these problems in succeeding sections.

A pattern-matching parser

Logic grammars are a well-studied set of grammar formalisms that are closely related to the logic programming language Prolog. Most Prolog language implementations, in fact, offer the capability to compile definite clause grammars (DCGs) directly into executable code that constitutes a recursive-descent parser for that grammar (Pereira and Warren, 1980). The built-in theorem prover of Prolog in effect becomes the parser, and significant advantage can be taken of the list-manipulation and unification features that are a great strength of logic programming.

In a DCG, a rule such as $S \rightarrow gSc$ would be written `s→[g],s,[c]`. Non-terminals are represented as Prolog atoms and terminals are given inside square-bracketed lists. This would be compiled to the Prolog rule:

```
s(S0,S):-S0=[g|S1],s(S1,S2),S2=[c|S].
```

Notice that a pair of variable parameters have been added to the non-terminals; these difference lists represent the input string and the remainder of the input string, respectively, after that non-terminal is successfully parsed—in effect, the span of the non-terminal on the input. In other words, the difference lists manage the input string, passing it from element to element, and hiding this 'implementation detail' from the user at the level of the DCG itself. Terminal lists in the DCG actually consume elements from the input list, in this case by unifying it (using the = operator) with a list having the terminal(s) as its head and the remainder list as its tail. It can be seen that the difference lists are arranged so that the span of the left-hand side non-terminal is that of the entire right-hand side of the rule. Thus, the rule $S \rightarrow \epsilon$, written in DCG form as `s→ []`, could be translated as simply `s(S,S)`. For the overall grammar, actual top-level calls to `s` would succeed in forms such as `s([g,g,g,c,c,c,],[])`, signifying that the non-terminal `s` indeed can span the entire input list.

DCGs also allow the embedding of arbitrary Prolog code in rules, set off by curly braces, and the attachment of additional parameters to non-terminals. These features raise the formal power of DCGs well beyond context free, in fact to the top of the Chomsky hierarchy. Moreover, with the recursive-descent parser inherent in Prolog, it is the responsibility of the programmer to write efficient, terminating rule sets. This is offset by the advantages Prolog offers in rapid prototyping and the ability easily to define new syntaxes and metalanguages especially tailored to a particular domain. The author's syntactic pattern-recognition system for biological sequence data, called GenLang, is a case in point (Searls and Dong, 1993).

GenLang was designed to process efficiently the huge input strings of DNA sequence data currently available and being produced at an exponentially increasing rate. Instead of

Prolog's linked lists, the input is represented in a 'C' array for efficiency which, however, entails extra bookkeeping behind the scenes in the parser; in fact, up to a dozen 'hidden' parameters are attached to non-terminals in order to manage information about the input string, the parse tree and the cost of imperfect matching.

Queries take the form <pattern>:<parse variable>⇒ <input>, where the pattern generally contains the top-level non-terminal in the grammar, the parse variable is a logic variable to which a parse tree will be bound, and the input is a DNA string (or file containing such a string). One other novel infix operator is the gap, denoted by an ellipsis, which simply consumes some length of input that may be either unbounded (...) or bounded by a minimum and maximum extent (e.g. `3...75`). Otherwise, GenLang grammars appear very much like ordinary DCGs, except that most objects, such as terminals, non-terminals, and gaps, may have attached to them lists of attributes, of the form:

$$\text{<object>:[<attribute}_1\text{>}, \ldots, \text{<attribute}_n\text{>]}$$

where the attributes are generally operations on keywords, and are of four types: (i) control operators, which modify the course of the parse and the position on the input string (departing from 'pure' logic programming, usually for the sake of efficiency); (ii) constraint attributes, which impose limits on quantities denoted by keywords, such as the cost (normally, the number of mismatches) of a subtree; (iii) specification attributes, which redefine the values of keyword quantities with arbitrary expressions; and (iv) assignment attributes, which bind the values of keyword quantities to logic variables that may be carried through the parse and used to report information at the top level. Besides the control attributes which can modify the backtracking search, another set of features, using the prefix operator `@`, can control the position of the parse on the input string, allowing for arbitrary translocations and additional kinds of constraints. This syntax is demonstrated in the following (relatively complicated) GenLang rule:

```
orf:[once,cost=C-S/10,
          parse=[span,cost]]→
     'atg',...:[step=3,S=(size>30)],
     @End, stop:[C=cost],@End.
```

Here, the essential framework of the rule states that the feature `orf` consists of the terminal string 'atg', followed by a gap, followed by the feature `stop`. The control attribute `step=3` specifies that the gap is to increase in increments of three, and the constraint attribute `size>30` indicates that the range 30 or less can be ignored. The latter is combined with an assignment that binds the value of `size` to the logic variable `S`, just as the `C=cost` assigns to `C` the cost of the stop. The cost of the rule, by default, is the sum of the costs of its components: the number of mismatches in the terminal strings plus (recursively) the costs of any non-terminals. Here, however, the specification attribute `cost=C-S/10` redefines the cost of this rule to be an algebraic function of the size of the gap and the cost of the `stop` feature. Another specification attribute controls what information will appear in the parse tree; in this case, just the numerical span of the orf and its computed cost, without the detailed subtree that would ordinarily be displayed.

The control attribute keyword `once` indicates that this rule can only succeed once in any starting position, and in this case prevents backtracking into the interior gap. (The effect of this keyword is similar to that of the Prolog 'cut', but it is just one of a family of such controls on the expansion of the parse.) This insures that orfs always end at a stop codon, and never include one in frame. The `@End`s in the rule body refer to position in the input string; the first one binds the position just before the `stop` to the variable `End`, and the second, since the variable is now bound, resets the input to that position. Thus, the span reported for the orf would extend up to, but not include, the stop codon. Again, there are a large number of variations on the `@` control syntax that allow for arbitrary movement on the input.

Not only are gaps first-class objects in GenLang, but they are in some ways the most important objects in terms of the implementation. This is because gaps, which 'skip over' input, are the search engines for individual features of interest that they precede. Since gaps produce the majority of the non-determinism, or backtracking behavior, in a grammar, they not only must be made very efficient, but they are treated specially by the grammar compiler. Gaps are typically not executed immediately in the course of a parse, but rather are 'packaged' and passed down the parse tree to succeeding non-terminals, until they encounter some feature with which they may combine for more efficient evaluation. For example, the combination of such a 'lazy' gap with a terminal string can be more efficiently evaluated than by brute force matching *à la* ordinary DCGs. GenLang will, at the option of the user, create a hash table of the position of every substring of a given length in the input string, so that a gap/string combination can simply be looked up for immediate evaluation, instead of scanning.

Lazy gaps also permit greater variety in the search strategy, which in the logic-based parser is ordinarily breadth-first on the input, i.e. all applicable rules are tried at every position before moving on to the next position. A lazy gap, however, is passed to the first applicable rule, which can be tried in every possible position allowed by the gap, before passing the gap to the next alternative clause or rule. This describes depth-first search on the input. The search strategy in GenLang can be varied locally through the use of the rule attributes `deep`, `wide`, or `best`.

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | **sales@docketalarm.com** | **1-866-77-FASTCASE**

fastcase® Smarter legal research.