Representing Genetic Information with Formal Grammars

David B. Searls
Unisys Paoli Research Center
P.O. Box 517, Paoli, PA 19301

Abstract

Genetic information, as expressed in the four-letter code of the DNA of living organisms, represents a complex and richly expressive natural knowledge representation system, capturing procedural information that describes how to create and maintain life. The study of its semantics (i.e., the field of molecular biology) has yielded a wealth of information, but its syntax has been elaborated primarily at the lowest lexical levels, without benefit of formal computational approaches that might help to organize its description and analysis. This paper discusses such an approach, using generative grammars to express the information in DNA sequences in a declarative, hierarchical manner. A prototype implemented in a Prolog-based Definite Clause Grammar system is presented, which allows such declarative descriptions to be used directly for analysis of genetic information by *parsing* DNA. Examples are given of the utility of this method in the domain, and speed-ups and extensions are also proposed.

1. Introduction

Beginning with the understanding of the "genetic code" in the 1950's and '60's, the essential lexical elements of the language based in DNA have been understood, and a great deal about its higher-level features has also been discovered, but not formalized in the sense of computational linguistics. More recently, the advent of techniques for efficiently isolating genes and determining their DNA sequence has led to an explosive accumulation of data. DNA sequence databases now contain thousands of entries, each consisting of hundreds or even thousands of nucleotide bases (the elements of the genetic code, abbreviated g, a, t, and c), and the rate at which new data is accumulating is accelerating rapidly. In the face of this mass of data, computerized DNA sequence analysis is becoming an increasingly important tool for molecular biologists in such realms as the identification of evolutionarily related sequences using homology (similarity) algorithms, the detection of specific sequences in large DNA sequence databases by pattern-matching techniques, more sophisticated algorithmic investigations of the relation of primary sequence to higher-order structure and function, and in the planning of recombinant DNA experiments.

This work was supported by the National Institutes of Health (DRR) under grant RO1RR04026-01.

Despite the proliferation of such tools and the explosive growth of sequence databases, methods for the specification and analysis of such genetic information tend to approach the underlying DNA sequence as a linear data set, as opposed to a highly structured *language* specifying biological information. Yet, it has been observed that an abstracted, hierarchical view is desirable in dealing with applications such as the prediction of 3-dimensional structure of biomolecules based on their primary sequence [Lathrop87]. We propose here an approach to DNA and protein sequence description and analysis, founded in computational linguistics, that provides a unifying conceptual framework for all the diverse activities described above, and which may also itself lead to new insights into the organization of DNA. This paper will first introduce the notion of applying general grammars to some simple DNA features, then discuss the linguistic power required for DNA, and then further elaborate a biological example along with an actual implementation and sample run.

2. A Genetic Grammar

Consider the partial grammar given below as an example. It consists of a collection of *rules* or *productions* denoted by arrows, each with a *non-terminal* (NT) symbol on its left-hand side (LHS), and a string of NTs and/or *terminals* (Ts) on its right-hand side (RHS) separated by commas and ending with a period. Ts correspond to the alphabet of the language being described, in this case double quoted strings of lower-case nucleotide bases (e.g. "gatc"); a vertical bar (|) signifies disjunction on RHSs. These conventions are familiar from BNF descriptions of computer languages.

```
catBox --> pyrimidine, "caat".
tataBox --> "tata", base, "a".
capSite --> "ac".
base --> purine | pyrimidine.
purine --> "g" | "a".
pyrimidine --> "t" | "c".
```

What is captured here, in terms of the four DNA bases and some prelexical elements describing base classes (i.e. purine and pyrimidine), are highly simplified descriptions of a few fairly low-level features of biological interest (catBox, etc.) that occur near the beginnings of many genes and act as signals delineating those genes and regulating their expression. This grammar fragment could also be described by *regular expressions*, and thus falls into the class of *regular languages* (RLs). In fact, many current pattern-matching algorithms used for DNA sequences are based on regular expression search. However, the advantages of a grammar-based representation begin to emerge as higher-level features are expressed, as shown below.

```
gene --> upstream, xscript, downstream.

upstream -->
    catBox, 40...50, tataBox, 19...27.

xscript -->
    capSite, ..., xlate, ..., termination.
```

The top-level rule for the NT gene in this grammar is an abstract declarative statement that a gene is composed of (1) an upstream region, containing the control regions defined above, followed by (2) a central region (xscript, so named because it undergoes *transcription* in the cell to an intermediate form, called messenger RNA, when the gene is expressed), followed by (3) a downstream region, not further described here. It also shows that the transcribed region xscript has within it a sequence xlate which we will see reflects a further *translation* of the messenger RNA into protein, the final product of gene expression. Note the introduction of a "gap" symbol ('...'), which simply specifies some otherwise undistinguished span of bases; a gap may be indefinite, as in the rule for xscript, or bounded by a minimum and a maximum span, as in upstream. In terms of parsing, a gap is just an indiscriminant consumer of bases.

These rules taken together show how the grammar can be "broken out" into its components in a perspicuous hierarchical fashion, with detail always presented at its appropriate level. Besides encouraging a higher-level description than the flat, left-to-right representation of regular expressions, grammars afford potentially greater expressive power than simple RLs; in the genetic domain, situations are encountered which seem to require the power of context-free (CF) languages, or greater. For example, the structure illustrated in Figure 1 represents a recurring theme in molecular biology. At the top are shown the two strands of DNA which bind to each other to form the double helix; the strands have directionality, indicated by the double arrows, and they bind in opposite orientations. Nucleotide bases positioned opposite each other are *complementary*, in that they fit each other, or *pair*, in a lock-and-key arrangement: the base "g" is complementary to "c", and "a" to "t". Because one strand thus determines the other's sequence, only one strand is described in any of these grammars. We denote substrings of bases by Greek letters, and their reversed complementary substrings by primes ($'$); base pairing is represented by dots.

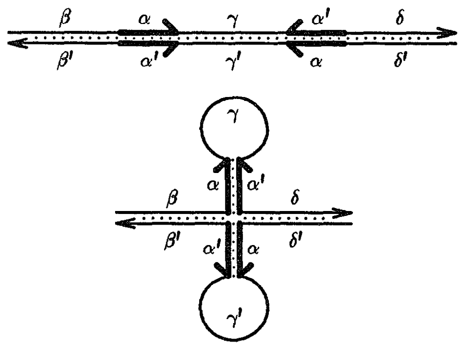

Figure 1. A Biological "Palindrome"

Alternative pairing of complementary substrings can give rise to biologically significant secondary structure in the DNA strands. In Figure 1, what is loosely referred to as a palindrome (i.e. α and α') in the top sequence produces a cruciform structure with unpaired loop-outs (γ and γ'), shown at the bottom. Sequences with potential to form such structures could formally be expressed as $\{ uvw \mid u,v,w \in \{g,a,t,c\}^*,\ |u|=|w|=j,$ and for $1 \leq i \leq j$ u_i is the complementary base to $w_{j-i+1}\}$, which can be shown to be a CF language (see Section 5) and not regular. In fact, tandem repeats, which are also common biological features, are formally *copy languages* (e.g. $\{ ww \mid w \in \{g,a,t,c\}^*\}$), which require even greater than CF power. There are many more complex examples of such features of secondary structure, particularly in RNA, and protein sequences offer an even richer variety.

Many *ad hoc* programs have been written, and variations on regular expression search implemented, that would address various shortcomings of current systems [Saurin87, Staden80], but these are neither formal nor general. Blattner has formally described a DNA-oriented language, which however does not treat the sequence itself as a formal language [Schroeder82]. Brendel has proposed a somewhat more formal linguistic approach for DNA sequences [Brendel84], but does not carry it much beyond a prelexical level; in any case, it has been argued persuasively that the formalism he offers (Augmented Transition Networks) is less clear, concise, efficient, and flexible than the formalism we will propose, that of Definite Clause Grammars (DCGs) [Pereira80].

3. A DCG-Based Implementation

DCGs are grammar systems that can be translated to rules in a Prolog program, which then constitutes a parser for the grammar. The translation involves the attachment of two parameters to each NT; one represents an input string of Ts passed into the NT, and the other a remainder list to be passed back out, should an initial substring of the first list parse as that NT. DCGs have three features that extend their power beyond CF: parameter passing, procedural attachments, and terminal replacement. This section introduces each of these features in turn, and in doing so continues a running biological example.

The grammar below picks up from that of Section 2, concentrating on *translation* products, i.e. the results of translating messenger RNA (previously transcribed from the DNA) to protein in the cell. The prelexical elements for the protein coding regions of the DNA are *codons*, or triplets of bases, each of which sequentially specifies corresponding amino acids in the resulting protein chain. That is, codon translation could be represented as a function $f{:}B^3 \rightarrow A \cup \{\tau\}$, where $B = \{g,a,t,c\}$, $A = \{met,phe,leu,ser, \ldots \}$ (a set of 20 amino acids), and τ represents a termination signal, which always ends a protein chain. The function is onto but not one-to-one, since there are 64 available triplets; this *degeneracy* in the genetic code lies primarily in variability in the third base of the codons. This is captured in the following rules.

```
stopCodon --> "tga" | "ta", purine.

codon(met) --> "atg".
codon(phe) --> "tt", pyrimidine.
codon(leu) --> "tt", purine.
codon(ser) --> "tc", base.          % etc...
```

This introduces the use of parameters on NTs, although in this case they are simply used to tag each codon rule with its amino acid identity. We now wish to write the rule for xlate that will concatenate these codons into translation products, using parameters to return the resulting protein sequence. First, though, we must introduce one last biological complication, which however is handled well by the grammar paradigm. This is the fact that in genes of higher organisms, sequences of bases that are ultimately translated, called *exons*, are interleaved with untranslated regions, called *introns*. Introns are excised after the messenger RNA is transcribed, but before it is translated into protein, in a process called splicing, illustrated in Figure 2. The grammar fragment below uses recursion to express the fact that an exon consists of a sequence of codons, while an actual translation region is a series of exons interrupted by introns. Also depicted is the fact that translated regions always begin with a methionine (met) codon.

```
xlate([met|RestAAs]) --> codon(met),
   xlate1(RestAAs), stopCodon.
xlate1(AAs) --> exon(AAs).
xlate1(AAs) --> exon(AAs1), intron,
   xlate1(AAs2), {append(AAs1,AAs2,AAs)}.

exon([]) --> [].
exon([AA|AAs]) --> codon(AA), exon(AAs).
```

The NT xlate and its auxiliary relation xlate1 combine sublists of amino acids from multiple exons, using a call to a Prolog predicate, append, enclosed in curly braces. This demonstrates a simple use of the procedural attachment feature of DCGs, whereby any arbitrary Prolog code (or even code from other languages) can be invoked. This can range from simple utilities, to more complex search heuristics, to entire expert systems.

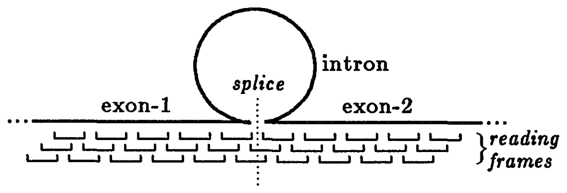

Figure 2. Intron Splicing

Note that message splicing may occur in such a way that a new codon is created when the "reading frame" (by which the DNA sequence is divided into triplets, partitioning it into three possible lists of codons) straddles the splice site. In order to account for this, we require an intron rule that deals with each possible reading frame, and furthermore can preserve the "spanning codon" that it may interrupt. This is elegantly handled by the terminal replacement feature of DCGs, a limited form of context-sensitivity by which Ts may be specified on the left hand side of a rule, trailing the NT. This has the effect of adding those Ts onto the front of the input string after such a rule parses. This feature is used in the rules for intron shown below; bases which contribute to the spanning codon are simply replaced after the intron is spliced out, in effect attaching them to the next exon, as far as the parser is concerned.[1]

```
intron --> splice.
intron, [B1] --> [B1], splice.
intron, [B1,B2] --> [B1,B2], splice.

splice --> donor, ..., acceptor.
```

The region to be excised is determined by the NT splice, and in turn by so-called splice donor and acceptor sites, to be described below in Section 6.

4. Example: Alternative Splicing

Given the DCG of Section 3, the following Prolog query attempts to parse complete translation products anywhere within a test string of bases.

```
| ?- [...,xlate(AAs)]:Parse ==>
   "aaatgaatagtatcttgttctactcgtagccaggtgaa".
```

Our parsing operator, '==>', has a number of special syntactic features. The LHS of the operator may be specified as a serial list of grammar elements, rather than a single NT. A parse tree may be returned, bound to a variable following the LHS and separated from it by a colon; this parse is carried by an additional hidden parameter augmenting the DCG mechanism. The RHS of the operator may be a list of bases, or a tag that references a database entry or external

[1] Note that Ts are shown here in conventional Prolog list structure ([...]), rather than by use of the double-quote convention, as above. This is necessary because we must show the individual bases as variables – in Prolog, atoms beginning with upper-case letters or underscores – while the double quotes are actually a shorthand for lists of ASCII character codes.

file. The query given above first succeeds in producing the following result.

```
AAs = [met,asn,ser,ile,leu,phe,tyr,ser],
Parse = [
...,
xlate([met,asn,ser,ile,leu,phe,tyr,ser]):
   codon(met):
      3/"atg"
   xlate1([asn,ser,ile,leu,phe,tyr,ser]):
      exon([asn,ser,ile,leu,phe,tyr,ser]):
         ...
   stopCodon:
      27/"ta"
      purine:
         29/"g"] ;
```

This parse returns a list of amino acids bound to the variable **AAs**, with no splicing. The parse tree returned to the variable **Parse** is printed so that the indentation shows the depth of the call, ellipses signify recursive calls whose outputs are omitted (due to a feature, not shown, that allows the user to "cut off" the parse tree at rules below which it becomes tiresome or unnecessary to store the entire parse), and Ts (bases) are always preceded with positional references set off by fore-slashes. For example, the last four lines indicate that a stop codon ("**tag**") was found at positions 27-29. The positional references are also carried through the parse by an added hidden parameter. The semicolon at the end is input by the user, and causes the query to fail, in effect asking for another answer by initiating backtracking.

```
AAs = [met,asn,thr,arg],
Parse = [
...,
xlate([met,asn,thr,arg]):
   codon(met):
      3/"atg"
   xlate1([asn,thr,arg]):
      exon([asn]):
         ...
      intron:
         9/"a"
         splice:
            donor:
               10/"gtatct"
            ...
            acceptor:
               24/"tcgtag"
      xlate1([thr,arg]):
         exon([thr,arg]):
            ...
   stopCodon:
      35/"tga"]
```

Upon backtracking, an additional parse is discovered, by introducing a splice; no other parses exist. Prolog-style backtracking is ideally suited to investigating real biological situations (e.g. certain viruses) that involve alternative splicing such as this, and also alternative start sites. This is in addition to the potential use of grammars for searching sequence databases for possible genes, described declaratively at this abstract level.

5. Efficiency

While the performance of this prototype system has been quite satisfactory in small test systems, for much larger search applications we must be concerned with efficiency, especially since we are giving up extremely fast linear-time string-matching algorithms.[2]

Because of the large amount of backtracking that may be expected in this application over huge input strings, chart parsing (i.e., saving intermediate results in a table that may be consulted dynamically) is an appropriate strategy to avoid the penalties of re-parsing complex features. A form of chart parser is easily implemented in a DCG by simply adding at the end of a rule an assertion of its LHS NT (with its associated position) as a ground clause into the database *ahead* of the rule itself (using the Prolog **asserta**). Thus, whenever the rule succeeds in parsing that NT, the clause entered will intercept any later attempts to parse the NT at that position in the input string, and succeed before the rule is invoked. Using a small test grammar, we find that the efficiency of the chart parser would begin to exceed that of the standard DCG parse after an average of only 1.9 backtracks over a distance of 40 bases — small on the scale of DNA sequences. Furthermore, this implementation of chart parsing can be greatly improved, because it is inefficient to index the NT chart entries by their list parameters, and because in fact the chart is much more effective if it also stores the points where parses fail (a far more frequent occurrence than success); such a volume of information in the chart would use excessive memory if stored as Prolog data structures. One answer is to use DCGs indexed by numerical position rather than by lists. This has the immediate benefit of allowing the input string to be stored as an external array (written in 'C'), rather than Prolog linked lists, with their time and space overhead. Also, numerical indexing allows a far more effective data structure for the chart, again using an external array. We have now implemented a chart which stores both success and failure information, the latter made feasible by storage in external bit arrays.

Related to efficiency concerns are potential problems with pure declarative logic-based expressions of grammar rules. Certain phenomena in genetic grammars might result in parses that are inefficient because of excessive backtracking, or even fail to terminate in the general case, or are under-specified in other regards. For example, a straightforward rule for the inverted

[2] However, note that regularizable sub-trees of genetic grammars could be detected and automatically converted to these fast string-matching algorithms. In fact, there are highly-optimized dynamic programming algorithms used for DNA similarity searches and other purposes which will probably be better left as external subroutines called from the DCGs as procedural attachments. The DCGs will then constitute organizing frameworks containing pure declarative grammars and, where appropriate, specialized algorithms, heuristics, etc. to maintain the tractability of the system.

complementary repeat or "palindrome" described in Section 2 is given by the following DCG rule:

```
invertedRepeat --> ... .        % loop-out
invertedRepeat --> [X],
   invertedRepeat, {X:::Y}, [Y].
```

The second clause collects complementary bases (with complementarity denoted by an infix operator ':::'), and calls itself recursively between them, while the first clause is a "gap" that corresponds to the loop-out. This rule is mathematically correct, and in fact is a direct encoding of the formula given in Section 2, but in practice it would not terminate, since the DCG implementation of the unbounded gap operator simply consumes any number of bases from the input string, and changing the clause order does not help. While a number of meta-level features and program transformation techniques have been proposed to handle such problems in logic programming, our current approach is to provide a library of built-ins which are individually optimized, or constrained, or which make use of external subroutines. For instance, a workable rule for inverted repeats is:

```
invertedRepeat(0,Loop) --> 0...Loop.
invertedRepeat(Length,Loop) -->
   {Next is Length - 1}, [X],
   invertedRepeat(Next,Loop), {X:::Y}, [Y].
```

By parameterizing the `Length` of match required and the maximum `Loop`, the rule can be properly constrained and forced to terminate. In practice, though, even more general rules should permit wider latitude in specification and deal with imperfect matching (see Section 6), while addressing efficiency concerns.

6. Imperfect Matching

Biologically speaking, the rules for `catBox` and other signal sequences given above were naive in portraying exact sequences, when in fact there is a great deal of variability observed. Usually such a signal would be expressed as a *consensus sequence*, a canonical representation of the most common bases in each position. This need to allow imperfect matching is recognized in the rules for splice donor and acceptor sites:[3]

```
donor --> "gt", [B3,B4,B5,B6],
   {2 of [purine==>[B3], [B4]="a",
          [B5]="g", [B6]="t"   ]}.
acceptor --> [B6,B5,_,B3], "ag",
   {2 of [pyrimidine==>[B3],
          pyrimidine==>[B5],
          pyrimidine==>[B6]    ]}.
```

Here, the infix operator 'of' takes as arguments an integer and a list of goals, and succeeds when at least that number of goals in the list is satisfiable. This provides a simple probabilistic element. While the "gt" and "ag" signals are invariant, matches on only a portion of the surrounding consensus bases (B3-B6) are required, sometimes only to the level of base classes. Some positions may even be unknown, as indicated by the Prolog anonymous variable (_) in `acceptor`. A more sophisticated mechanism, however, is required.

base/%	*1*	*2*	*3*	*4*	*5*
g	100	50	25	40	50
a	0	25	25	30	50
t	0	25	25	20	0
c	0	0	25	10	0
min→	35	35	35	10	0
max→	265	165	115	90	50
entropy	0.00	1.50	2.00	1.85	1.00

Figure 3. A Hypothetical Consensus Sequence

Consider the consensus sequence depicted in Figure 3, showing the relative base frequency over a hypothetical five-position sequence. These involve fractional match operators that require some score, measured in terms of a distance metric, to exceed a threshold parameter. For example, a trivial scheme might simply examine the additive percentages across the sequence, which would be a maximum of 265 for a "perfect" fit (e.g. "ggcga"). One might then define a 60% match as $0.6 \times 265 = 159$, for a threshold of 159 defining success. Such metrics have been well-studied [Kruskal83], but an implementation in logic permits a natural incorporation of mechanisms for "eager" success and failure, so that only as many comparisons are done, reading left to right, as are necessary to prove that the predicate cannot fail, given the minimum remaining score (min→), or cannot succeed, given the maximum remaining score (max→). For example, at a 60% threshold an initial string of "ca" would be destined to fail, whereas "gt" could not fail, regardless of the remaining sequence.

Furthermore, clause reordering can enhance the efficiency of logical proof based on such probabilistic elements. For example, in the example above, it would be better to test position 5 before position 4, because 5 would be more likely to fail on inappropriate input than 4. In other words, the distribution in position 5 has more information content, and in fact we can reorder our examination of base positions according to the inverse rank order of their informational *entropy*, $-\sum_{x \in \{g,a,t,c\}} p_x \log_2 p_x$ where p_x is the probability that base x appears in that position [Shannon64]. Thus, the order in which these positions would be examined is 1, 5, 2, 4, and 3, again using eager success and failure. The use of a numerically indexed grammar structure allows compile-time reordering of the examination of positions within a term, and we hope to generalize this to clause reordering for eager decision through entire rule bodies, though this will be significantly complicated by variable gaps within rules.

[3] Note also the recursive application of the parsing operator; while its use here is trivial, and could be replaced by base class predicates of arity one (e.g. `purine(B3)`), it is nevertheless a potentially powerful technique for such purposes as managing "multi-layered" parsing [Woods80].

7. Extensions

The utility of grammars may extend beyond structural descriptions of DNA, to encompass functional aspects of genetic and biochemical systems. For instance, there is a fundamental similarity between biochemical reactions and the notion of terminal replacement in DCGs. In the latter case, input is consumed and replaced with another terminal string, just as in a reaction substrate is "consumed" and replaced by product. For example, the rule

```
    oxidativeDeamination, "u" --> "c"
```

would replace a "c" residue with a "u" (a process used experimentally to create "directed" mutations). In other words, there is a sense in which the biochemical notation

$$\textit{substrate} \xrightarrow{\textit{reaction}} \textit{product}$$

corresponds to the grammatical notation

```
    reaction, product --> substrate.
```

Thus, DCGs are able to conceptually capture the idea of performing reactions on the DNA sequence, and thereby altering it. This should make it possible to extend the language to deal with phenomena like gene rearrangement, and experimental manipulation of DNA fragments. We are examining methods by which this formalism (though perhaps with improved syntactic representation) may be applied to *populations* of molecules, rather than single input strings, so that, for example, grammars could be used as scripts in complex experiment-planning systems.

By a further extension, grammars can be used to combine high-level descriptions of sequence with simulations acting on that sequence. The following grammar deals with regulatory sequences called *promoters*, which in simple systems may be feedback-inhibited by the protein product of the gene they regulate.

```
transcribedGene(Time) -->
    activePromoter(Time), xscript,
    {assert(product(Time))}.

activePromoter(Time) -->
    {concentration(Time,C), threshold(T),
    C<T}, promoter.

lifetime(10).        % average 10 seconds
threshold(7.5).      % concentration (mM)
```

This grammar deals not only with the sequence data but with the environment. The rule `transcribed-Gene`, having parsed an active promoter and a transcribed region, asserts into the database a discrete amount of time-stamped gene product. The rule `activePromoter` postulates a product inhibition, succeeding in recognizing its sequence (lexically encoded elsewhere in the NT `promoter`) only if the current concentration `C` of product is below some regulatory threshold `T` (set by the predicate `threshold`). The predicate `concentration` (not shown) maintains the Prolog database of `product` according to the expected `lifetime` of that product. Repeatedly parsing for `transcribedGene`, at a rate of one parse per second, produces the behavior shown in Figure 4.

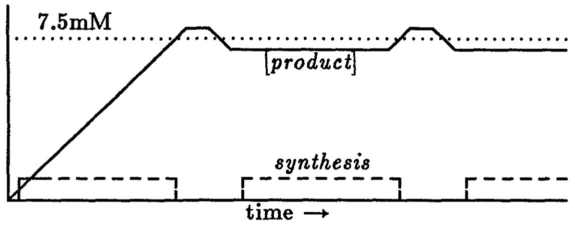

Figure 4. Simulation by Repeated Parsing

This shows grammars used not just for lexical analysis, but acting in and on a general context, and perhaps even modeling biological molecules (e.g. RNA polymerase, the enzyme which moves along the DNA copying the RNA transcript). A similar use of logic (though not involving grammars) has proven useful in more complex simulations and qualitative reasoning about biological systems [Koton85], and we believe that the virtues of linguistic descriptions can also be brought to bear in describing and experimenting with more involved sequence-dependent control systems, such as bacterial regulatory systems called *operons*, and *attenuators*, which depend on alternative palindromic secondary structures.

References

[Brendel84] V Brendel and HG Busse, Genome Structure Described by Formal Languages. *Nucleic Acids Research* **12**, 1984, pp. 2561-2568.

[Koton85] P Koton, Towards a Problem Solving System for Molecular Genetics. *MIT Laboratory for Computer Science Technical Report* **338**, 1985.

[Kruskal83] JB Kruskal, An Overview of Sequence Comparison. In *Time Warps, String Edits, and Macromolecules*, D. Sankoff and J.B. Kruskal (ed.), Addison-Wesley, 1983, pp. 1-44.

[Lathrop87] RH Lathrop, TA Webster, and TF Smith, Ariadne: Pattern-Directed Inference and Hierarchical Abstraction in Protein Structure Recognition. *Communications of the ACM* **30**, 1987, pp. 909-921.

[Pereira80] FCN Pereira and DHD Warren, Definite Clause Grammars for Language Analysis – A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence* **13**, 1980, pp. 231-278.

[Saurin87] W Saurin and P Marliere, Matching Relational Patterns in Nucleic Acid Sequences. *Computer Applications in the Biosciences* **3**, 1987, pp. 115-120.

[Schroeder82] JL Schroeder and FR Blattner, Formal Description of a DNA Oriented Computer Language. *Nucleic Acids Research* **10**, 1982, p. 69.

[Shannon64] CE Shannon and W Weaver, *The Mathematical Theory of Communication.* University of Illinois Press, Urbana, IL, 1964.

[Staden80] R Staden, A Computer Program to Search for tRNA Genes. *Nucleic Acids Research* **8**, 1980, pp. 817-825.

[Woods80] WA Woods, Cascaded ATN Grammars. *American Journal of Computational Linguistics* **6**, 1980, pp. 1-12.