Proc. 2nd Intl. Conf. on Bioinformatics, Supercomputing, and Complex Genome Analysis

A SYNTACTIC PATTERN RECOGNITION SYSTEM FOR DNA SEQUENCES

DAVID SEARLS and SHAN DONG
Department of Genetics, University of Pennsylvania School of Medicine
422 Curie Boulevard, Philadelphia, PA 19104-6145 USA

ABSTRACT

We review both theoretical and practical results of a linguistic approach to studying the structure of features of DNA sequences. Using generative grammars, complex assemblages can not only be described and analyzed abstractly, but also concretely, such that features can be searched for by a general-purpose parser. Our parser, called GenLang, uses an extended logic grammar formalism and has found features as complex as tRNA genes, group I introns, and protein-encoding genes, within input sequences on a genomic scale.

1. Introduction

Recently a number of authors have adopted a "linguistic" view of DNA sequences. Most of their work has involved examinations of the occurrences of "words" in DNA in what is essentially an information-theoretic approach[1,2], or the use of statistical analyses of vocabularies in the tradition of comparative linguistics[3,4]. These approaches, however, can be distinguished from another methodology in linguistics, pioneered by Noam Chomsky, which attempts to characterize higher-order phenomena in languages by way of generative *grammars*. Only a few authors have used this discipline in the realm of biological sequences[5,6,7]; we have found that such a linguistic approach proves useful not only in theoretical characterization of certain structural phenomena in sequences, but also in generalized pattern recognition in this domain, via *parsing*. In this paper we will review both these theoretical findings, and practical results using a logic-based parser for *syntactic pattern recognition.*

2. Background

We begin with the necessary background concerning the formalism of language theory, as well as some of its uses and implementations.

2.1. Formal Language Theory

In formal language theory, a language is defined in terms of an *alphabet*, which is a finite set of *symbols*; in the case of DNA sequences, such symbols would be the nucleotide bases. A DNA molecule can then be represented as a finite *string* of symbols from this alphabet; a *language*, formally, is any set of such strings.

The concern of formal linguistics is the finite representation of languages which may themselves be infinite; the goal is an economy of expression, in an abstract representation, as an alternative to exhaustively enumerating all the allowable strings in a language. Such cogency may also have the benefit of capturing some kind of essential, clarifying generalization about the structure or *syntax* of a linguistic system, preferably related to the meaning or *semantics* of the language elements. For this purpose, language generators called *grammars* have proven extremely useful. Grammars specify languages through sets of *rules*, which achieve the desired succinctness largely by referring to each other and to themselves *recursively*. Perhaps the most important class of grammars is the *context-free grammars* (CFGs, which specify the *context-free languages*, CFLs). A CFG has a set of *terminals* or alphabetic elements, and an additional set of symbols called *nonterminals*; these symbols are used in a finite set of rules whose members are written as $A \rightarrow u$ where A is a nonterminal and u is a string of terminals and nonterminals. A grammar generates the string elements of its language by taking a starting symbol S and rewriting it, by repeatedly finding a rule whose left-hand side matches some nonterminal in the current string, and substituting that rule's right-hand side, until the string contains all terminals. Such derivation steps are denoted by a double arrow, $\Longrightarrow$, so that for the simple grammar with alphabet $\{a, b, c\}$, nonterminals S and X, and rules $S \rightarrow aX$, $X \rightarrow bX$, and $X \rightarrow c$, one possible derivation is:

$$S \Longrightarrow aX \Longrightarrow abX \Longrightarrow abbX \Longrightarrow abbc$$

This grammar formalism would appear to be preferable to either trying to list the infinite set of strings $\{ac, abc, abbc, abbbc, \cdots\}$, or using the informal description "all strings consisting of an a followed by any number of b's followed by a c." For one thing, it makes feasible the computational task of *parsing* a string to determine whether it is in the language specified by the grammar. A useful byproduct of parsing is the production of a *parse tree* reflecting the grammar rules applied and giving a kind of structural description of grammatical features in the input string—exactly the kind of output that is desired in describing certain biological sequence data.

CFGs have proven to be very useful in the field of compiler construction, where, in the form of BNF (Backus-Naur Form) descriptions, they are used to formally specify programming languages. An even more interesting application of computational linguistics, however, is in understanding natural language—a complex problem that has stimulated a large body of research. Although straightforward CFGs can be written that cover many aspects of natural language syntax, natural languages in their full generality are now thought to require greater than context-free power.

Regular expressions, using the basic operations of concatenation, disjunction (logical OR), and "star" (denoting any number of occurrences of its argument), also specify languages. However, the set of *regular* languages is strictly a subset of the CFLs, for no regular expression can specify certain *self-embedding* structures such as palindromes; computationally, these require a stack to store information about *de-*

pendencies between distant elements of the string. In fact, even the CFLs are a strict subset of the *context-sensitive* languages (CSLs), described by grammars that have more than one symbol on the LHS of rules. While CFLs are restricted to describing *nested* dependencies, CSLs can specify *crossing* dependencies, such as those found in *copy languages,* which contain duplicated strings of arbitrary extent. These language classes all take their place on the *Chomsky hierarchy* of languages, which categorizes the linguistic complexity of any given language, and which serves as the basis for analysis of the decidability and/or tractability of recognizing strings of any language with general-purpose parsers. $O(n)$ parsers are easily designed for regular and *deterministic* CFLs—those that can be recognized without the need to backtrack on the input string—and $O(n^3)$ parsers exist for *any* CFL. Certain well-defined characteristics of CFGs, such as *ambiguity* (the ability to derive the same strings via multiple distinct parses), determine their potential for more efficient general-purpose parsing, and for any *particular,* narrowly-defined language special-purpose linear-time recognizers can often be designed. Languages beyond context-free are increasingly more difficult to recognize by general-purpose parsers, and much effort has gone into defining language classes "slightly greater" than context-free that are adequate to a particular domain (such as natural language) yet can be parsed efficiently.

2.2. Syntactic Pattern Recognition

This methodology is not limited to computer languages or human natural languages, but can be extended to all manner of signals, images, or other data which have underlying structure. The field of Syntactic Pattern Recognition (SPR)[8] makes use of the tools and techniques of computational linguistics, such as grammars and parsers, to specify and search for patterns in data. Because grammars intrinsically promote the hierarchical abstraction of features, these can be built up to a very high level while maintaining a clear, modular "knowledge base." Moreover, grammars by their nature detect individual features in this higher-level context, which creates a much greater degree of discrimination than isolated searches. SPR benefits from a strong formal foundation, but also incorporates features that extend the expressive power of grammars where necessary for the domain. For example, "noisy" signals can be dealt with by so-called stochastic grammars[8], which incorporate probabilities into grammars in a natural way. SPR, in fact, has been classified as a form of *pattern-directed inference*, and indeed we have found that it provides an excellent framework for the incorporation of heuristics at many levels. SPR has been successfully applied to such problems as general signal processing, handwritten character recognition, and karyotype analysis by us[9,10] and many others[8], and our initial results with SPR and the linguistic analysis of DNA (described below) suggest that it can address many of the complexities of this domain as well.

2.3. Logic Grammars

Prolog is a programming language that gives a procedural interpretation to a subset of first-order predicate logic. It uses a clausal form that lets programs be

written as databases of atomic predicates called *facts*, e.g., `protein(hemoglobin)`, and *rules* of the form `protease(X) :- protein(Y), degrades(X,Y).` This can be read "X is a protease *if* there is a protein Y *and* X degrades Y." Prolog's rules and facts, together called *relations*, can be queried to perform inferences by backward-chaining proof, using a mechanism called *resolution*, and a powerful pattern-matching facility known as *unification*. The resulting system is able to perform computation as controlled deduction—in fact, a form of theorem proving.

Prolog's history is closely linked with the formalism of Definite Clause Grammars (DCGs), and the notion that grammars can be expressed as rules of a Prolog program[11]. The process of parsing a string then becomes that of proving a theorem given that string as input and the "axioms" of a grammar. In practice, the previous example grammar would appear as in the code below.

```
s --> [a], x.                    x --> [b], x | [c].
```

In Prolog, logical predicates begin with a lower-case letter, and in DCGs these correspond to nonterminals. (Prolog variables begin with upper-case letters.) Terminals are shown as Prolog list elements; lists generally appear within square brackets, with list elements separated by commas (e.g., `[a,b,c]`), but can also be double-quoted strings which correspond to lists of ASCII character codes, e.g., `"gattac"`. The vertical bar in the second rule is an "or" (disjunction), so that this rule for `x` actually represents the two rules given previously.

DCGs actually require a translation step to become Prolog clauses, because Prolog must have a mechanism for manipulating the input string, which it does by maintaining the string in "hidden" parameters of the nonterminals, called *difference lists*. In fact, the user may attach other parameters to nonterminals, as well as embed arbitrary code in rule bodies, all of which lends DCGs power far beyond CFGs. It is our belief that they offer a firm foundation for a language that can be a powerful descriptive tool for molecular genetic features, and also offers a flexible analytical and control mechanism through Prolog's built-in DCG parser.

3. Theoretical Results

We suggested several years ago[7] that nucleic acids were beyond regular and at least context-free, based on the phenomenon of secondary structure: the stem portion of a stem-and-loop structure entails *nested dependencies* between base-paired residues, which are easily specified by a self-embedding CFG, but which in the general case are beyond the capabilities of any formal regular expression. It was also suggested[7,12] that DNA may be beyond context-free as well, due to the phenomenon of direct repeats, which constitute a *copy language* with crossing dependencies that cannot be described with essentially stack-based context-free formalisms. Subsequent work[13] formalized these conjectures.

Formal discussion of the linguistic status of DNA, like that of natural language, may be based on empirical phenomenology, but in the case of nucleic acids may also rest on the actual physical structure of the molecules, and in particular the ability to form secondary structure. We have offered formal proofs that idealized representations of such structure are indeed non-regular[13]. Beyond this, however, the simple existence of direct repeats is somewhat unsatisfying as evidence for the purely *formal* status of DNA, since it is only when direct repeats with no particular bounds on their extent can be shown to be *necessary* in a language that it can be said to be greater than context-free on that account. So as not to depend upon *ad hoc* phenomenology, we sought examples which, like that of inverted repeats, could be grounded in actual physical structures and processes arising in the molecules themselves. A series of such arguments were presented[13] based upon (1) the potential for circularization of DNA with terminal direct repeats, (2) unequal crossing-over in multiple tandem repeats, and, most importantly, (3) the existence of pseudoknots in structural RNA, which entail crossing dependencies between stems within each other's loops. It is interesting that dealing with pseudoknots has required major reimplementations of some RNA structure prediction programs[14]; the relative difficulty of this can in part be explained by the transition to greater-than-context-free recognition which can thus no longer be strictly stack-based. This points again to the utility of a solid formal linguistic characterization in the design of recognition algorithms in any given domain.

A number of additional formal results were given[13], dealing with other linguistic attributes, again based on a somewhat idealized model of the structural characteristics of nucleic acids. Inverted repeats, for example, were shown to be *nondeterministic* languages, and the branching or *recursive* nature of secondary structures implies their language is *non-linear* (in this context, meaning that any grammar describing them must have a rule with more than one nonterminal on its right hand side). The *ambiguity* of general secondary structural grammars (that is, their ability to produce more than one essentially distinct parse for the same primary sequence) was also explored[13], and it was shown that this grammatical ambiguity reflects alternative secondary structures in a biologically relevant way. (Since that time, we have proven that, while the most general language of ideal orthodox secondary structure is actually deterministic and thus unambiguous, biologically plausible secondary structure sublanguages are *inherently ambiguous*, i.e. impossible to describe by any unambiguous grammar). These results rule out the use of certain $O(n^2)$ simplifications of general-purpose context-free parsers (which are otherwise $O(n^3)$).

Given that biological sequences are beyond context-free, it is of interest to circumscribe their exact boundaries. We suggested[7] and subsequently demonstrated[13] that the language encompassing all of the phenomena described above in nucleic acid structure lies not only in the CSLs, but within a restricted subset known as the *indexed languages.* A subset of the indexed languages with a very perspicuous grammar formalism particularly well-suited to nucleic acids, known as *string variable grammar,* was developed by us[12,13]; examples are given in the next section.

We have also explored *closure* properties of the Chomsky hierarchy under biological operations—that is, whether language classes of interest, after undergoing certain biological processes, can be expected to remain at the same level in the Chomsky hierarchy or not. We have found[13] that regular languages and CFLs are closed under double-stranded replication and under simple recombinational events such as scission and ligation. Deterministic languages, however, are not closed under these operations, so that, for example, certain features can be recognized more efficiently on one strand of DNA than on the other, or in other than a leftmost fashion, suggesting the use of so-called *island parsing* strategies. With regard to *evolutionary* operations such as duplication, inversion, and transposition, it was shown that CFLs are *not* closed, suggesting that genomic rearrangements on an evolutionary scale may be responsible for increasing the mathematical complexity of the genetic language.

4. Practical Results

GenLang, our logic-based syntactic pattern recognition system for DNA sequences, has been described previously[7,10,12,13,15]. Some of its newer developments will be reviewed here.

4.1. String Variables

One advantage of logic grammars lies in the rapid prototyping capabilities of Prolog, including the ability to easily add new syntactic constructs. For example, in GenLang queries are of the form <*pattern*>:<*parse variable*> `==>` <*input*>, where new infix operators separate grammar elements: the *pattern* generally contains the top-level nonterminal in the grammar, the *parse variable* is a logic variable to which a parse tree will be bound, and the *input* is as described below. In the parse tree, nonterminals are typically adorned with information about their cost (in number of mismatches), their location in the input, the actual primary sequence recognized, etc. Similarly, *gaps* of either unbounded (`...`) or bounded (e.g. `19...27`) extent were added to the language.

A more significant extension to DCGs, called *string variables*[12,13], also benefits from the Prolog milieu. A string variable is a logic variable appearing in the body of a grammar rule, which stands for a string of arbitrary extent, and which may optionally have applied to it operators such as the tilde which denotes reverse complementarity. Such a feature in this domain makes it easy to specify even complex arrangements of direct and inverted repeats, such as are characteristic of secondary structure:

```
tandem_repeat ---> X, X.                    stem_loop ---> X, ..., ~X.
pseudoknot ---> X, ..., Y, ~X, ..., ~Y.
attenuator ---> X, ..., ~X, ..., X.
```

These descriptions are at a much higher level than the corresponding context-free grammars[12], and in fact most of them are even beyond context-free, yet despite their widely varying linguistic complexity they are expressed with comparable ease in

this formalism. Even so, these are relatively abstract descriptions and for purposes of parsing require "real world" constraints on their length and degree of mismatch allowed in stems; we next describe the mechanism for controlling these.

4.2. Attributes

Objects in GenLang can have attached to them an *attribute list* using the syntax, <*object*>:[<*attribute1*>,<*attribute2*>,...,<*attributeN*>]. There are four primary types of attributes, which generally entail operations on keywords: (1) *control* attributes which control the parse and the position on the input string at a meta level; (2) *constraint* attributes which impose constraints on (typically numerical) quantities denoted by keywords, such as the cost (normally, the number of mismatches) of a subtree; (3) *specification* attributes whereby the values of keywords such as 'cost' can be redefined with arbitrary expressions; and (4) *assignment* attributes which assign the values of keywords to logic variables which may then be carried through the parse and/or reported at the top level. For example:

```
foo:[cost=S+2*C] ---> "atg", ...:[step=3,S=size], bar:[size<50,C=cost].
```

Here, the control attribute `step=3` specifies that the gap (...) is to increase in increments of 3; the constraint attribute `size<50` keeps the span of the nonterminal `bar` under 50; the specification attribute `cost=S+2*C` redefines the cost of the nonterminal `foo` to an arithmetic function of the size of the gap and the cost of the `bar`; and the assignment attributes `S=size` and `C=cost` serve to bind those variables. (The default cost of `foo` would have been the number of mismatches in the `"atg"` plus those within `bar`.)

Attributes are managed by way of additional "hidden parameters" in the implementation of the grammar. Just as the difference lists serve to unburden the grammar designer of the low-level programming involved in input list management, a total of ten hidden parameters now hide such details as the accumulated cost of parse trees and the cost thresholds applied by the grammar, additional constraints on the size ranges of individual elements in the grammar, and the parse tree itself.

4.3. Gaps

Gaps represent regions that are "skipped over", but in fact it is the gaps that do the skipping—they constitute the multiple, embedded search engines of a typical grammar, and the source of most of its non-determinism—and so careful attention to their implementation has been necessary. One important feature is *delayed* (or *lazy*) *evaluation:* gaps encountered in the course of a parse are "packaged" and passed down the parse tree, and are not actually evaluated until they in turn encounter some feature with which they may combine for more efficient evaluation. For example, the combination of a lazy gap with a string of bases might under the right circumstances allow the string simply to be looked up in a hash table, rather than searched for in the primary sequence. GenLang does, at the option of the user, hash its input into k-tuples of varying sizes, and permits many hundred-fold more efficient recognition of features such as direct repeats.

The implementation of lazy gaps also lends itself to finer control over the search strategy used by the parser. The logic-based parser is ordinarily breadth-first on the *input*, in the sense that all applicable rules will be tried at every position in a parse before moving on to the next position following a gap. However, a *lazy* gap will be passed to the first applicable rule, and that rule will be tried in every possible position permitted by the gap, before the gap is passed on to the next applicable rule—in other words, depth-first search on the input. The search style in GenLang can be controlled either at a global level, or locally through the use of the attributes `deep`, `wide` or `best` (the latter performing best-first search within a defined range).

4.4. Cost

A rule such as `foo ---> "gat" | "gaa" | "gta" | ...` would ordinarily succeed upon recognizing any of the disjuncts on the input string (or, any of the disjuncts with mismatches allowed up to the cost threshold established higher in the parse tree). However, with the addition of a `consensus` attribute, the disjuncts are treated as exemplars in the calculation, at compile time, of a weight matrix which now contributes the cost at this point of the parse. Moreover, the order in which the parser examines positions in the input string is reordered at compile time, so that the most "informative" positions—that is, those for which inappropriate input will most rapidly cause the nonterminal to exceed its cost threshold and thus fail—are examined first, for optimal efficiency. Base frequency data may also be entered into the grammar in tabular form, based on published data; however, compilation time even for large lists of exemplars is nearly as fast, and allows the user to enter new data freely, postulate classes by dividing exemplars among several nonterminals, etc. There are several methods available for calculating costs from base frequency data (e.g. the attribute `cost=neglog` uses a negative logarithm of base frequency), which are user-definable as well.

GenLang also offers cost attributes that support rules concerning the local enrichment of features. For example, the current rule used for the pyrimidine-rich region between the branch and the 3′ acceptor in a protein-encoding grammar is

```
pyrimidine_rich:[parse=[size,cost,list]] --->
     probably_pyrimidine, pyrimidine_rich | [].

probably_pyrimidine:[cost=(-5),once] ---> pyrimidine:[cost=0].
probably_pyrimidine:[cost=(3/5)] ---> base.
```

When a cost threshold is placed on this feature at a higher level in the grammar, the recursive rule finds the longest list it can of "`probably_pyrimidine`'s" which are defined as either an authentic pyrimidine (its cost constrained to be zero), in which case the overall cost is decremented by 5, or as any other base, in which case the cost given as a ratio indicates that the cost is to be calculated as 3/5th of the way from the current cost to the threshold. This has the effect of giving "credit" for every hit (though the cost is not permitted to go below zero), and on every miss causing a

logarithmic decay in the cost until it intersects the threshold in a region that is too pyrimidine-poor. This paradigm is not limited to simple rules for single bases, but can be applied to repetitions of very high-level patterns as well.

Note also the definition constraint for `parse` in the first rule above; which attributes are reported in the parse tree (in this case, the feature's `size`, overall `cost`, and a `list` displaying its primary sequence), are user-definable at every level.

4.5. Input

Prolog's linked lists proved to be an inappropriate data structure for the exceedingly long inputs found in the biological domain, and so GenLang is actually a hybrid software system in which the input and certain other data structures, as well as some low-level procedures, are implemented in 'C'. Besides the hashing of the input described above, another important speedup is the implementation of *chart parsing,* a dynamic programming technique that prevents repeated recomputation of sub-trees of the parse and ameliorates much of the cost of non-deterministic parsing. This technique has been enhanced with a domain-specific feature that allows for rapid traversal of regions already parsed, using a byte-encoding of flags marking these regions for any particular nonterminal.

The form of the input presented to GenLang can range from actual lists of bases, to sequence files, to directories of files, and even flat sequence database files, with individual entries handled sequentially, as if disjoined. This allows what amounts to grammar-based query of GenBank. The user may filter selections based on attributes of the database "object". Thus, a typical query is

```
(...,gene,...) ==> genbank:[division=rodent,locus='MUS*',
                            definition='*complete cds*',date>1/1/92].
```

Text strings are given in single quotes and may include an asterisk "wild card". This query would search for a `gene` in the rodent file of genbank, selecting entries with locus ID's beginning with "MUS" (for mouse), whose definition contains the string "complete cds", which were entered after the beginning of this year. The processing of the input attributes is accomplished through the use of an additional logic grammar within GenLang describing the structure of GenBank entries. Using this approach, new forms of data (e.g. ASN.1 or even relational databases) will be easily handled by simply adding new such attribute grammars, which can generally be developed in short order using DCGs.

4.6. Parse Visualization

The current version of GenLang has a graphical parse visualization and management tool, implemented in 'X' using Motif widgets and the Quintus Prolog interface. This tool provides a dynamic, real-time picture of the developing parse, so that the user is aware of the current state of computation at all times. This is important in the development of efficient grammars, since it can be made readily apparent to an observer when a highly non-deterministic feature is causing "thrashing" of the parse

Figure 1. The parse visualization tool, showing a parse of a pair of tRNA genes.

at some particular point in the input. It is also very useful in debugging of grammars, since it can be seen at what point a parse fails on test input that is known to be valid; this is especially important given the relative difficulty of debugging complex logic programs such as this, in the absence of the external, uniform point of reference provided by the input string, and a means of visualizing the activities of the logic programming relative to it. Finally, an unanticipated benefit of the parse visualization tool has been the capacity it affords for manual control over the parse at a meta level; a series of such controls are available from the graphical interface to allow sub-trees of the parse to be altered selectively, in a manner not otherwise available in a "flat" interface. The overall display for the parse visualization tool is shown (Figure 1).

At the top of this window are two rows giving salient header information extracted from the GenBank entry. At the upper right are five buttons controlling the top level of the input processing: *Exit* and *Done* buttons, a *Redo* button which causes the parse to start over on the current entry, a *Next* button which causes it to skip to the next entry, and an *Info* button which opens a scrolling window giving the entire GenBank header. The latter feature is most useful when comparing the results of a parse to information in the GenBank Features table. All of these controls have also been made available in the flat TTY interface to GenLang, via modifications to the Prolog interrupt handler, so that any errant parse may be stopped by a control-C and redone, skipped over, compared with the Features table, etc. at any time.

The long narrow window near the top gives a picture of the overall input string, shown as a long line labelled with numerical positions. Immediately above this is a thin line extending over a portion of the entire sequence, with its bounds labelled numerically; this corresponds to the highest-level feature parsed.

The large main window portrays the primary sequence in a series of rows; the number of rows in the window and the number of bases placed in each row—either 100 or 200—are controlled by the user, as well as background colors and other graphical attributes. Above each row of primary sequence is drawn the parse tree as it develops. Lower-level nonterminals are first drawn immediately above the sequence as they are

```
tRNA("ctt"): span=90470/90583
   stem: list="ggttgtt"
   turn: list="tg"
   D arm: span=90479/90496
      stem: list="gcc"
      loop: list="gagcggtctaa" cost=0
      ~stem: list="ggc" cost=0
   base: list="g"
   anticodon arm: span=90497/90546
      stem: list="cctgat"
      loop: cost=0
         pre_codon: list="tc"
         ~codon: list="aag"
         purine: list="a"
         intron: span=90509/90539
            ...: size=5
            "ctt": span=90514/90517
            ...: size=22
         base: list="a"
      ~stem: list="ctcagg" cost=1
   extra arm: span=90546/90559
      ...: size=13
   TpsiC arm: span=90559/90576
      stem: list="aagag"
      loop: list="ttcgaat" cost=6
      ~stem: list="ctctt" cost=0
   ~stem: list="agcaacc" cost=0.5
```

```
tRNA("aac"): span=127320/127393
   stem: list="gactcca"
   turn: list="tg"
   D arm: span=127329/127346
      stem: list="gcc"
      loop: list="aagttggttaa" cost=0
      ~stem: list="ggc" cost=0
   base: list="g"
   anticodon arm: span=127347/127364
      stem: list="tgcga"
      loop: list="ctgttaa" cost=0
         pre_codon: list="ct"
         ~codon: list="gtt"
         purine: list="a"
         base: list="a"
      ~stem: list="tcgca" cost=0
   turn: list="agatc"
   TpsiC arm: span=127369/127386
       stem: list="gtgag"
       loop: list="ttcaacc" cost=73
      ~stem: list="ctcac" cost=0
   ~stem: list="tggggtc" cost=0.5
```

Figure 2. The first two parse trees returned by the tRNA grammar parsing chromosome III of *S. cerevisiae*. Note the intron and extra arm in the first, absent in the second.

recognized in the parse, as horizontal lines extending over the entire scope of the nonterminal and labelled by a tag associated with that nonterminal. Successively higher nonterminals are drawn in similar fashion above these as the parse progresses, in a bottom-up manner until a parse is completed. Which nonterminals are tagged and when they are drawn in the course of the parse are specified by the user with control attributes in the grammar; typically there is a tradeoff between the "informativeness" of the developing parse tree and the speed of the parse, and a useful development paradigm is to refine a grammar with a large amount of annotation and then to run the resulting specification with much less real-time annotation, or even batch-wise without benefit of the graphical interface.

The bottom left button causes the parser to backtrack and find another parse, and the adjacent buttons allow for variations on the usual backtracking scheme, as noted above. The user has the option of backtracking into not only the end, but also the beginning of any feature in the parse tree, so as to force it to move forward on the input string, and it can even be made to skip to any arbitrary position. The feature to be moved can be selected from a menu or by buttoning directly on the parse tree, and the desired position can be selected by buttoning on the sequence.

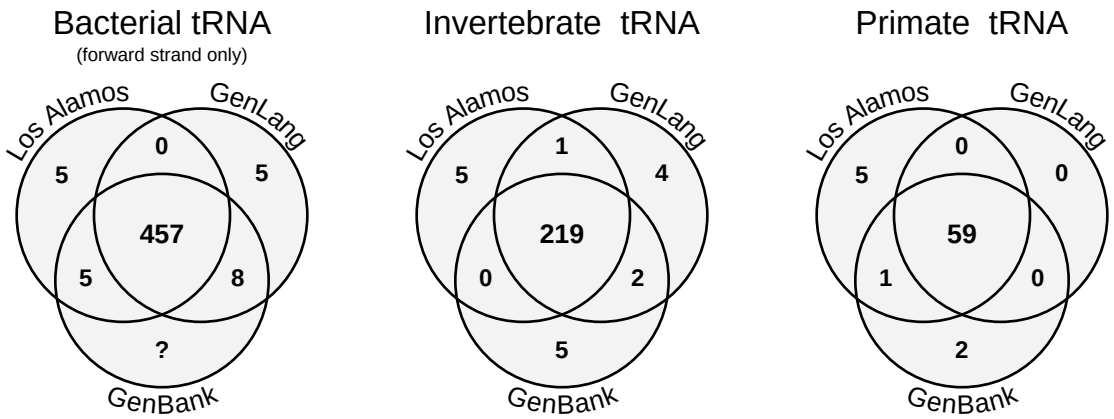

Figure 3. Results of batch runs of the tRNA grammar and the Los Alamos tRNA finder[18] against three GenBank (Release 71.0) divisions. The three-way Venn diagrams indicate the number of "hits" found by each program, and annotated in GenBank (except for the bacterial division, where only program hits were confirmed), with overlaps among the three sources indicated. Most "unannotated" hits were simply cases where two hits overlapped, and only one was correct, rather than outright false positives. At least one GenLang–only hit appears to be an authentic unannotated tRNA, found on the opposite strand of an entry with two other confirmed tRNAs.

4.7. Selected Results

Using a rudimentary protein-coding gene grammar, GenLang was able to "discover" all five genes in the human beta globin cluster, in a region of 73,000 bp, with no false hits[15]; it also ignored the pseudogenes in this region, since they are syntactically invalid genes. Using relatively simple weight matrices for splice junctions and no compositional information about coding vs. non-coding regions, this grammar does not provide sufficient discrimination to reliably predict genes with more than a few introns, due to the combinatorial explosion of possibilities. Nevertheless, this nondeterminism of the parser proved to be useful in modeling thalassemia mutations in which alternative splicing results in a number of untranslateable messages also predicted by the parser[15]. Our current efforts in this area involve the incorporation of more sophisticated statistical and compositional measures into the syntactic framework of the grammar.

An *E. coli* tRNA grammar developed by us[10] was subsequently used by George Michaels at the NIH to parse all available *E. coli* sequence data (about 3,700,000 bp in both directions). After making one minor change to the grammar (loosening a constraint on a single base), the grammar found all 34 known tRNA genes in about three hours of total parse time, and was able to correct the standard *E. coli* map (EcoSeq[16] Version 5 in four cases (two sites wrong and two anticodons wrong) [G. Michaels, personal communication]. A generalized tRNA grammar, which was more loosely specified and which handled introns as well, was recently used by our group to parse all of yeast *(S. cerevisiae)* chromosome 3, over 315,000 bp, taking about eleven minutes in each direction (Figure 2). This parse found exactly the ten known

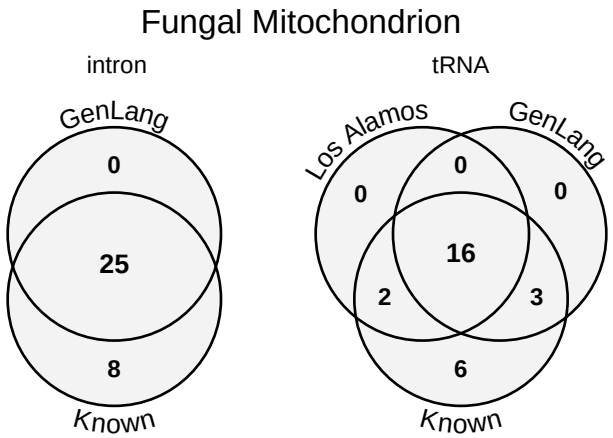

Figure 4. Results of parsing the mitochondrial genome of *P. anserina* with the Group I intron and tRNA grammars. While neither tRNA finder performed as well on the organelle division as on others, the GenLang grammar was easily modified to detect five additional tRNAs on this genome (a total of 24/27), with no increase in false positives.

tRNA genes on this chromosome[17]; surprisingly, it was actually faster than a highly successful special-purpose tRNA finder from Los Alamos[18] (which however is faster with GenBank flat files). When run on a variety of divisions from GenBank, this grammar demonstrated both sensitivity and specificity well in excess of 95% when measured against known, annotated tRNA genes, and in several cases (like the Los Alamos program) apparently found unannotated tRNA genes (Figure 3).

The grammar is able to easily implement the constraint that tRNA introns contain within them a triplet which base-pairs with the anticodon, before splicing occurs. Although not done in the current grammar, it would also be relatively straightforward to incorporate postulated pseudoknot-like covariances between the D-arm and TΨC-arm loops, or the fact that the introns and extra arms are required to have extensive internal base-pairing. Obviously, for purposes of search it is also easy to add contextual clues, such as the fact that these genes tend to terminate in T–rich regions, or in yeast are found in proximity to certain transposable elements[17].

Our most recent efforts in grammar design involve modelling of the self-splicing Group I introns, which compared to tRNA show a much more complex and variable secondary structure (including pseudoknots) with superimposed conserved regions[19]. This grammar was used to parse the 100,000 bp mitochondrial genome of the fungus *P. anserina,* with the results shown (Figure 4). We have parsed the same genome with a combination of the two grammars described here, plus a simple grammar for long open reading frames[15], with results (Figure 5) that suggest that GenLang may be a useful tool for assisting in the annotation of genomic sequence data.

Figure 5 (overleaf). Parse tree for the entire mitochondrial genome of *P. anserina* using a grammar composed of a disjunction of the Group I intron, tRNA, and ORF grammars. The parse took exactly two hours and produced most of the annotations found in published maps[20].

```
tRNA("atc"): span=600/672
   primary stem: list="gattcct"
   D arm: list="acttaactggttaaagt"
   ~codon arm: list="tatttttgataagaata"
   TpsiC arm: list="cagcgttcgagccgctg"
   ~primary stem: list="aggaatc"

tRNA("tca"): span=832/917
   primary stem: list="agagaat"
   D arm: list="accgagtggtttaaggt"
   ~codon arm: list="gctagcttgagattagt"
   TpsiC arm: list="atgggttcgaatcccat"
   ~primary stem: list="attctct"

tRNA("cca"): span=1204/1277
   primary stem: list="cgggtta"
   D arm: list="gtcagattggttacgac"
   ~codon arm: list="tctcatttgggatggga"
   TpsiC arm: list="ctgtgttccgagtcacag"
   ~primary stem: list="taacccg"

group I intron: span=3230/4661
   p3: span=3230/3235 list="ttggg"
   p4: span=3240/3246 list="ttcaaa"
   p5/~p5: span=3249/3283
   ~p4: span=3285/3291 list="tttgaa"
   p6/~p6: span=3290/3431
   p7: span=3435/3440 list="gacta"
   ~p3: span=3476/3481 list="cccaa"
   p8/~p8: span=3481/4618
   ~p7: span=4632/4637 list="tagtc"
   p9/~p9: span=4639/4661

group I intron: span=6296/8517
   p3: span=6296/6301 list="ttggg"
   p4: span=6306/6312 list="ttcaaa"
   p5/~p5: span=6315/6372
   ~p4: span=6374/6380 list="tttgaa"
   p6/~p6: span=6379/6438
   p7: span=6442/6447 list="gacta"
   ~p3: span=6484/6489 list="cccaa"
   p8/~p8: span=6489/8482
   ~p7: span=8489/8494 list="tagtc"
   p9/~p9: span=8497/8517

tRNA("gaa"): span=9196/9267
   primary stem: list="ggattag"
   D arm: list="gtcaagtggtacgac"
   ~codon arm: list="tagctctttcaatgcta"
   TpsiC arm: list="gcaggttcgattcctgc"
   ~primary stem: list="ctagtct"

tRNA("atg"): span=9437/9509
   primary stem: list="gaggcta"
   D arm: list="gctcaattggtagagc"
   ~codon arm: list="tactgctcataacagta"
   TpsiC arm: list="aagtgttcaagtcactt"
   ~primary stem: list="tagcctt"

tRNA("atg"): span=9600/9670
   primary stem: list="tgcgggt"
   D arm: list="gtgtaatagaaacat"
   ~codon arm: list="tttggctcatgcccaaa"
   TpsiC arm: list="aagggtgcaaatccttt"
   ~primary stem: list="acccgca"

tRNA("tta"): span=9914/9995
   primary stem: list="atccgag"
   D arm: list="ctggaatggtagacag"
   ~codon arm: list="gcaagcttaagttttgc"
   TpsiC arm: list="gaacgttcaagtcgttc"
   ~primary stem: list="ctcggat"

tRNA("gca"): span=10108/10180
   primary stem: list="ggggata"
   D arm: list="gtctaatttgggtaaaac"
   ~codon arm: list="ctgacttgcacgtcgg"
   TpsiC arm: list="tcaggttcaattcctga"
   ~primary stem: list="tttctcc"

tRNA("ttc"): span=10184/10256
   primary stem: list="gcctgag"
   D arm: list="gctcaattggtagagc"
   ~codon arm: list="gagcattgaagctgctt"
   TpsiC arm: list="gtaagttcaagtcttac"
   ~primary stem: list="cttaggc"

tRNA("cta"): span=10266/10349
   primary stem: list="atggatg"
   D arm: list="ctggaaatggtagacag"
   ~codon arm: list="agccgcttaggacggtt"
   TpsiC arm: list="acaagttcaaatcttgt"
   ~primary stem: list="catctat"

tRNA("cac"): span=10883/10955
   primary stem: list="gataata"
   D arm: list="gctcaataggtagagc"
   ~codon arm: list="actgtatgtggcacagt"
   TpsiC arm: list="ctctgttcaaatcagag"
   ~primary stem: list="tattgtc"

tRNA("atg"): span=11266/11336
   primary stem: list="gcccggg"
   D arm: list="gtttaaaggtaaaac"
   ~codon arm: list="ttaatttcatacgttaa"
   TpsiC arm: list="gagaattcgattttctc"
   ~primary stem: list="cccaggt"

long ORF: span=11606/13277 size=1671

long ORF: span=13277/14579 size=1302

group I intron: span=13386/13635
   p3: span=13386/13391 list="caatt"
   p4: span=13394/13400 list="tgggta"
   p5/~p5: span=13404/13476
   ~p4: span=13478/13484 list="tactca"
   p6/~p6: span=13486/13559
   p7: span=13569/13574 list="gagta"
   ~p3: span=13580/13585 list="agttg"
   p8/~p8: span=13585/13593
   ~p7: span=13609/13614 list="tactc"
   p9/~p9: span=13619/13635

tRNA("aga"): span=15217/15288
   primary stem: list="ttcccat"
   D arm: list="gctcaatggtagagc"
   ~codon arm: list="gaatacttctaatatta"
   TpsiC arm: list="ttagttcgagtctaaa"
   ~primary stem: list="atgggaa"

tRNA("tac"): span=17598/17685
   primary stem: list="aggagag"
   D arm: list="ccgtttgttggtatgacgg"
   ~codon arm: list="ttaagcctgtaaacttaa"
   TpsiC arm: list="gaaggttcaattccttt"
   ~primary stem: list="ctttcct"

long ORF: span=17916/18726 size=810

tRNA("aaa"): span=18766/18837
   primary stem: list="aggatct"
   D arm: list="gtataatggtaatac"
   ~codon arm: list="tgtgacttttaatcatt"
   TpsiC arm: list="atcggttcgaatccggt"
   ~primary stem: list="agatcct"

tRNA("gga"): span=18971/19042
   primary stem: list="gcgatat"
   D arm: list="gtctaattggtaaaac"
   ~codon arm: list="tccatcttccaaatgtt"
   TpsiC arm: list="gtgagttcgaatctcac"
   ~primary stem: list="atatcgt"

tRNA("gac"): span=19272/19344
   primary stem: list="gaattag"
   D arm: list="acttaattggtagagg"
   ~codon arm: list="tttccttgtcacggaag"
   TpsiC arm: list="gccggttcgatgccggt"
   ~primary stem: list="ctagttc"

tRNA("gca"): span=19354/19434
   primary stem: list="ggaaaag"
   D arm: list="gccataggtaaggt"
   ~codon arm: list="agatatttgctaaatat"
   TpsiC arm: list="aggtgttcgagtcacct"
   ~primary stem: list="cttttcc"

tRNA("tga"): span=19474/19544
   primary stem: list="aagagta"
   D arm: list="gtttaaaggtaaaa"
   ~codon arm: list="agaaagcttcaacctttaa"
   TpsiC arm: list="ctfagttcgagtctaag"
   ~primary stem: list="tgctctt"

long ORF: span=19737/20403 size=666

tRNA("gta"): span=20537/20608
   primary stem: list="aagaaat"
   D arm: list="gctcaacggtagagc"
   ~codon arm: list="accgttttacacacgga"
   TpsiC arm: list="atgagttcgaatctcct"
   ~primary stem: list="atttctt"

tRNA("aac"): span=20871/20941
   primary stem: list="gccccta"
   D arm: list="gctcaatggtagagc"
   ~codon arm: list="taatactgttaatatta"
   TpsiC arm: list="agatgttcgattcatct"
   ~primary stem: list="tgggggc"

long ORF: span=21456/22347 size=891

group I intron: span=23759/24006
   p3: span=23759/23764 list="aagag"
   p4 stem: list="tccaagg"
   p5/~p5: span=23780/23885
   ~p4 stem: list="tcgtgga"
   p6/~p6: span=23895/23933
   p7: span=23938/23943 list="gagta"
   ~p3: span=23948/23953 list="ttctt"
   p8/~p8: span=23953/23973
   ~p7: span=23980/23985 list="tactc"
   p9/~p9: span=23989/24006

long ORF: span=24530/25361 size=831

tRNA("atg"): span=26071/26141
   primary stem: list="gcccggg"
   D arm: list="gtttaaaggtaaaac"
   ~codon arm: list="ttaatttcatacgttaa"
   TpsiC arm: list="gagaattcgattttctc"
   ~primary stem: list="cccaggt"

tRNA("gta"): span=29624/29695
   primary stem: list="aagaaat"
   D arm: list="gctcaatggtagagc"
   ~codon arm: list="accgttttacacacgga"
   TpsiC arm: list="atgagttcgaatctcct"
   ~primary stem: list="atttctt"

group I intron: span=30104/30263
   p3: span=30104/30109 list="tcggg"
   p4: span=30115/30121 list="tgctga"
   p5/~p5: span=30124/30144
   ~p4: span=30149/30155 list="tcagca"
   p6/~p6: span=30154/30200
   p7: span=30206/30211 list="gacta"
   ~p3: span=30215/30220 list="cttga"
   p8/~p8: span=30220/30234
   ~p7: span=30240/30245 list="tagtc"
   p9/~p9: span=30247/30263

long ORF: span=30272/31574 size=1302

long ORF: span=31740/32412 size=672

long ORF: span=32581/33676 size=1095

group I intron: span=34532/34717
   p3: span=34532/34537 list="tttcg"
   p4: span=34542/34548 list="tgctgg"
   p5/~p5: span=34551/34578
   ~p4: span=34581/34587 list="tcagca"
   p6/~p6: span=34586/34636
   p7: span=34640/34645 list="ggcta"
   ~p3: span=34649/34654 list="cgaga"
   p8/~p8: span=34654/34689
   ~p7: span=34695/34700 list="tagtc"
   p9/~p9: span=34701/34717

long ORF: span=34766/35444 size=678

long ORF: span=38537/41036 size=2499

long ORF: span=41253/42237 size=984

group I intron: span=42287/42525
   p3: span=42287/42292 list="tataa"
   p4: span=42297/42303 list="tgcggg"
   p5/~p5: span=42306/42411
   ~p4: span=42414/42420 list="tccgca"
   p6/~p6: span=42419/42457
   p7: span=42461/42466 list="gacta"
   ~p3: span=42470/42475 list="ttata"
   p8/~p8: span=42474/42494
   ~p7: span=42501/42506 list="tagtc"
   p9/~p9: span=42507/42525

long ORF: span=42533/43469 size=936

group I intron: span=43536/43807
   p3: span=43536/43541 list="cttcg"
   p4: span=43546/43552 list="tgcatg"
   p5/~p5: span=43555/43600
   ~p4: span=43682/43688 list="tgtgca"
   p6/~p6: span=43687/43727
   p7: span=43731/43736 list="gacta"
   ~p3: span=43740/43745 list="cgagg"
   p8/~p8: span=43745/43772
   ~p7: span=43780/43785 list="tagtc"
   p9/~p9: span=43786/43807

long ORF: span=43922/46241 size=2319

group I intron: span=46558/46758
   p3: span=46558/46563 list="gtaat"
   p4: span=46573/46579 list="tggtga"
   p5/~p5: span=46582/46621
   ~p4: span=46626/46632 list="tcagca"
   p6/~p6: span=46632/46651
   p7: span=46660/46665 list="gagta"
   ~p3: span=46672/46677 list="gttgc"
   p8/~p8: span=46677/46721
   ~p7: span=46730/46735 list="tactc"
   p9/~p9: span=46737/46758

long ORF: span=46893/47778 size=885

long ORF: span=47956/49246 size=1290

group I intron: span=49282/49511
   p3: span=49282/49287 list="tctaa"
   p4: span=49292/49298 list="tgctgg"
   p5/~p5: span=49301/49352
   ~p4: span=49355/49361 list="tcagca"
   p6/~p6: span=49360/49434
   p7: span=49438/49443 list="gacta"
   ~p3: span=49447/49452 list="ttaga"
   p8/~p8: span=49452/49474
   ~p7: span=49480/49485 list="tagtc"
   p9/~p9: span=49486/49511

long ORF: span=49526/50732 size=1206

long ORF: span=51894/52923 size=1029

long ORF: span=53727/54339 size=612

long ORF: span=54342/55665 size=1323

group I intron: span=54506/55738
   p3: span=54506/54511 list="aatgc"
   p4: span=54522/54528 list="tgcagg"
   p5/~p5: span=54531/54634
   ~p4: span=54638/54644 list="cccgca"
   p6/~p6: span=54643/54721
   p7: span=54725/54730 list="ggcca"
   ~p3: span=54780/54785 list="gcatt"
   p8/~p8: span=54785/55696
   ~p7: span=55712/55717 list="tggtc"
   p9/~p9: span=55718/55738

group I intron: span=55839/56582
   p3: span=55839/55844 list="aatta"
   p4: span=55855/55861 list="tgctgg"
   p5/~p5: span=55864/55915
   ~p4: span=55918/55924 list="tcagca"
   p6/~p6: span=55923/56387
   p7: span=56396/56401 list="aacta"
   ~p3: span=56425/56430 list="tagtt"
   p8/~p8: span=56430/56543
   ~p7: span=56558/56563 list="tagta"
   p9/~p9: span=56564/56582

long ORF: span=57088/57835 size=747

long ORF: span=58331/59315 size=984

group I intron: span=58379/59359
   p3: span=58379/58384 list="atctt"
   p4: span=58390/58396 list="tgctgg"
   p5/~p5: span=58399/58463
   ~p4: span=58466/58472 list="tcagca"
   p6/~p6: span=58473/58725
   p7: span=58736/58741 list="gacaa"
   ~p3: span=58750/58755 list="aagat"
   p8/~p8: span=58755/59321
   ~p7: span=59337/59342 list="ctgtc"
   p9/~p9: span=59343/59359

long ORF: span=59474/60800 size=1326

group I intron: span=59548/59742
   p3: span=59548/59553 list="gacca"
   p4: span=59558/59564 list="tgctgg"
   p5/~p5: span=59567/59627
   ~p4: span=59630/59636 list="tcagca"
   p6/~p6: span=59635/59676
   p7: span=59680/59685 list="gacta"
   ~p3: span=59689/59694 list="tggtc"
   p8/~p8: span=59694/59713
   ~p7: span=59719/59724 list="tagtc"
   p9/~p9: span=59725/59742

long ORF: span=60971/61748 size=777

group I intron: span=61785/62069
   p3: span=61785/61790 list="tacgt"
   p4: span=61793/61799 list="tcggta"
   p5/~p5: span=61802/61928
   ~p4: span=61931/61938 list="taccgt"
   p6/~p6: span=61937/61986
   p7: span=61990/61995 list="gacta"
   ~p3: span=61996/62001 list="acgtg"
   p8/~p8: span=62002/62032
   ~p7: span=62041/62046 list="tagtc"
   p9/~p9: span=62049/62069

group I intron: span=62177/62521
   p3: span=62177/62182 list="tttga"
   p4: span=62187/62193 list="tgctgg"
   p5/~p5: span=62196/62254
   ~p4: span=62257/62263 list="tcagca"
   p6/~p6: span=62262/62289
   p7: span=62293/62298 list="gacta"
   ~p3: span=62302/62307 list="tcaaa"
   p8/~p8: span=62307/62484
   ~p7: span=62495/62500 list="tagtc"
   p9/~p9: span=62501/62521

tRNA("cgt"): span=64487/64557
   primary stem: list="ggtgtgt"
   D arm: list="gtttaatggcaaaa"
   ~codon arm: list="atggcgctacggacgttgt"
   TpsiC arm: list="ataagttcgagtcttat"
   ~primary stem: list="acacatc"

long ORF: span=64906/66193 size=1287

group I intron: span=66026/66544
   p3: span=66026/66031 list="tatat"
   p4: span=66041/66047 list="tttttg"
   p5/~p5: span=66050/66115
   ~p4: span=66120/66126 list="taaaga"
   p6/~p6: span=66125/66442
   p7: span=66446/66451 list="ggcta"
   ~p3: span=66452/66457 list="atgtg"
   p8/~p8: span=66457/66512
   ~p7: span=66523/66528 list="tagtc"
   p9/~p9: span=66529/66544

long ORF: span=66813/68103 size=1290

group I intron: span=68291/68902
   p3: span=68291/68296 list="aacct"
   p4: span=68302/68308 list="tgcggg"
   p5/~p5: span=68311/68440
   ~p4: span=68442/68448 list="tctgca"
   p6/~p6: span=68447/68687
   p7: span=68691/68696 list="gacta"
   ~p3: span=68701/68706 list="aggtt"
   p8/~p8: span=68706/68873
   ~p7: span=68880/68885 list="tagtc"
   p9/~p9: span=68886/68902

long ORF: span=69221/71057 size=1836

long ORF: span=71771/73154 size=1383

group I intron: span=71880/72135
   p3: span=71880/71885 list="aactg"
   p4: span=71891/71897 list="tccggg"
   p5/~p5: span=71900/72004
   ~p4 stem: list="tcgcgga"
   p6/~p6: span=72014/72052
   p7: span=72057/72062 list="gagta"
   ~p3: span=72067/72072 list="cagtt"
   p8/~p8: span=72072/72099
   ~p7: span=72106/72111 list="tactc"
   p9/~p9: span=72114/72135

long ORF: span=73337/74627 size=1290

group I intron: span=74524/75137
   p3: span=74524/74529 list="aagta"
   p4 stem: list="ttagaga"
   p5/~p5: span=74547/74570
   ~p4: span=74572/74578 list="tactaa"
   p6/~p6: span=74578/75022
   p7: span=75025/75030 list="gacta"
   ~p3: span=75064/75069 list="tactt"
   p8/~p8: span=75069/75101
   ~p7: span=75107/75112 list="tagtc"
   p9/~p9: span=75113/75137

long ORF: span=75233/77585 size=2352

long ORF: span=77898/79155 size=1257

long ORF: span=80511/81660 size=1149

group I intron: span=81043/82004
   p3: span=81043/81048 list="ttaat"
   p4: span=81054/81060 list="tatttg"
   p5/~p5: span=81063/81080
   ~p4: span=81082/81088 list="taagta"
   p6/~p6: span=81092/81251
   p7: span=81254/81259 list="aacta"
   ~p3: span=81267/81272 list="attag"
   p8/~p8: span=81272/81964
   ~p7: span=81974/81979 list="tagtc"
   p9/~p9: span=81980/82004

group I intron: span=82143/82375
   p3: span=82143/82148 list="ttaaa"
   p4: span=82158/82164 list="ttcaag"
   p5/~p5: span=82167/82186
   ~p4: span=82188/82194 list="cttgaa"
   p6/~p6: span=82193/82250
   p7: span=82254/82259 list="gacta"
   ~p3: span=82296/82301 list="tttag"
   p8/~p8: span=82301/82325
   ~p7: span=82340/82345 list="tagtt"
   p9/~p9: span=82352/82375

long ORF: span=82582/83857 size=1275

long ORF: span=85118/86468 size=1350

group I intron: span=85741/86001
   p3: span=85741/85746 list="aagag"
   p4: span=85752/85758 list="accggg"
   p5/~p5: span=85761/85858
   ~p4 stem: list="tcgcggt"
   p6/~p6: span=85866/85919
   p7: span=85923/85928 list="gagta"
   ~p3: span=85933/85938 list="ttctt"
   p8/~p8: span=85938/85962
   ~p7: span=85969/85974 list="tactc"
   p9/~p9: span=85977/86001

long ORF: span=87162/88077 size=915

long ORF: span=89833/91396 size=1563

group I intron: span=91107/91676
   p3: span=91107/91112 list="agcta"
   p4: span=91116/91122 list="taaagg"
   p5/~p5: span=91125/91171
   ~p4 stem: list="tctttta"
   p6/~p6: span=91179/91595
   p7: span=91607/91612 list="gacta"
   ~p3: span=91617/91622 list="tggtt"
   p8/~p8: span=91622/91642
   ~p7: span=91652/91657 list="tagtc"
   p9/~p9: span=91659/91676

group I intron: span=91954/92237
   p3: span=91954/91959 list="ttata"
   p4: span=91968/91974 list="tacggg"
   p5/~p5: span=91977/92100
   ~p4 stem: list="tcgcgta"
   p6/~p6: span=92110/92156
   p7: span=92161/92166 list="gagca"
   ~p3: span=92172/92177 list="tatag"
   p8/~p8: span=92177/92202
   ~p7: span=92210/92215 list="tgctc"
   p9/~p9: span=92217/92237

group I intron: span=94718/96891
   p3: span=94718/94723 list="tccga"
   p4: span=94729/94735 list="ttcaag"
   p5/~p5: span=94738/94753
   ~p4: span=94755/94761 list="cttgaa"
   p6/~p6: span=94760/94810
   p7: span=94814/94819 list="gacta"
   ~p3: span=94858/94863 list="tcggg"
   p8/~p8: span=94863/96855
   ~p7: span=96862/96867 list="tagtc"
   p9/~p9: span=96869/96891

long ORF: span=97027/98350 size=1323

group I intron: span=98675/98848
   p3: span=98675/98680 list="aattg"
   p4: span=98685/98691 list="tgctgg"
   p5/~p5: span=98695/98716
   ~p4: span=98719/98725 list="tcagca"
   p6/~p6: span=98724/98774
   p7: span=98778/98783 list="gacta"
   ~p3: span=98790/98795 list="taatt"
   p8/~p8: span=98795/98810
   ~p7: span=98824/98829 list="tagtc"
   p9/~p9: span=98831/98848
```

5. Bibliography

1. W. Ebeling and M. A. Jimenez-Montano, *Math. Biosci.* **52** (1980) 53.
2. M. A. Jimenez-Montano, *Bull. Math. Biol.* **46(4)** (1980) 641.
3. V. Brendel, J. S. Beckmann, and E. N. Trifinov, *J. Biomol. Struct. Dynamics* **4** (1986) 11.
4. P. A. Pevzner, M. Y. Borodovsky, and A. A. Mironov, *J. Biomol. Struct. Dynamics* **6** (1989) 1013.
5. V. Brendel and H. G. Busse, *Nucleic Acids Res.* **12** (1984) 2561.
6. J. Collado-Vides, *Comput. Applic. in the Biosciences* **7(3)** (1991) 321.
7. D. B. Searls, in *Proceedings of the National Conference on Artificial Intelligence* (American Association for Artificial Intelligence, 1988), p. 386.
8. K. S. Fu, *Syntactic Pattern Recognition and Applications* (Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982).
9. D. B. Searls, *Intelligent Systems Rev.* **1(4)** (1989) 67.
10. D. B. Searls and S. Liebowitz, in *Proceedings of the Workshop on Syntactic and Structural Pattern Recognition* (IAPR, 1990), p. 402.
11. F. C. N. Pereira and S. M. Shieber, *Prolog and Natural-Language Analysis* (Center for the Study of Language and Information, Stanford CA, 1987).
12. D. B. Searls, in *Logic Programming: Proceedings of the North American Conference*, eds. E. Lusk and R. Overbeek (MIT Press, 1989) p. 189.
13. D. B. Searls, in *Artificial Intelligence and Molecular Biology*, ed. L. Hunter (AAAI Press, 1992), p. 47.
14. J. P. Abrahams, M. van den Berg, E. van Batenburg, and C. Pleij, *Nucleic Acids Res.* **18** (1990) 3035.
15. D. B. Searls and M. O. Noordewier, in *Proceedings of the Conference on Artificial Intelligence Applications* (IEEE, 1991), p. 3.
16. K. E. Rudd, W. Miller, J. Ostell, and D. A. Benson, *Nucleic Acids Res.* **18** (1990) 313.
17. S. G. Oliver et al., *Nature* **357** (1992) 38.
18. G. A. Fichant and C. Burks, *J. Mol. Biol.* **220** (1991) 659.
19. T. R. Cech, *Gene* **73** (1988) 259.
20. D. J. Cummings, K. L. McNally, J. M. DoMenico, and E. T. Matsuura, *Curr. Genet.* **17** (1990) 375.