ANL-91/29

Automated Insertion of Sequences into a Ribosomal RNA Alignment: An Application of Computational Linguistics in Molecular Biology

by Ronald C. Taylor

Mathematics and Computer Science Division
Mathematics and Computer Science Division
Mathematics and Computer Science Division

Argonne National Laboratory, Argonne, Illinois 60439
operated by The University of Chicago
for the United States Department of Energy under Contract W-31-109-Eng-38

Reproduced from the best available copy.

Available to DOE and DOE contractors from the
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831
Prices available from (615) 576-8401, FTS 626-8401

Available to the public from the
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161

Distribution Category:
Mathematics and
Computer Science (UC-405)

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439-4801

ANL--91/29

DE92 004954

ANL-91/29

Automated Insertion of Sequences into a Ribosomal RNA Alignment: An Application of Computational Linguistics in Molecular Biology

by

*Ronald C. Taylor**

Mathematics and Computer Science Division

November 1991

This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, and in part by Case Western Reserve University. It was submitted in partial fulfillment of the requirements for the Degree of Master of Science, Department of Biology, Case Western Reserve University, August 1991 (thesis advisor: Dr. Ross Overbeek).

*Present address: National Institutes of Health, Div. of Computer Research and Technology, Bethesda, MD 20892

Contents

List of Figures

List of Tables

Automated Insertion of Sequences into a Ribosomal RNA Alignment: An Application of Computational Linguistics in Molecular Biology

by

Ronald C. Taylor

Abstract

This thesis involved the construction of (1) a grammar that incorporates knowledge on base invariancy and secondary structure in a molecule and (2) a parser engine that uses the grammar to position bases into the structural subunits of the molecule. These concepts were combined with a novel pinning technique to form a tool that semi-automates insertion of a new species into the alignment for the 16S rRNA molecule (a component of the ribosome) maintained by Dr. Carl Woese's group at the University of Illinois at Urbana. The tool was tested on species extracted from the alignment and on a group of entirely new species. The results were very encouraging, and the tool should be of substantial aid to the curators of the 16S alignment. The construction of the grammar was itself automated, allowing application of the tool to alignments for other molecules. The logic programming language Prolog was used to construct all programs involved. The computational linguistics approach used here was found to be a useful way to attack the problem of insertion into an alignment.

1 Introduction

Since January 1990, with the interruption of other projects, I have been working on the representation of molecular structure via computational linguistics, that is, via a grammar-based method.

The first serious application to which I have applied this approach is the devising of a means to automate or semi-automate insertion of new species into the alignment of the 16S rRNA ribosomal subunit created and maintained by Dr. Carl Woese and his colleagues at the University of Illinois at Urbana. This is the subject of my master's thesis in biology at Case Western Reserve University. My thesis supervisor at Argonne National Laboratory was Dr. Ross Overbeek, a senior scientist in Argonne's Mathematics and Computer Science Division.

One way to look at what I have been doing is to say that I have been working on a representation (encoding) of the structure of a molecule that is closer to how biologists think of such things. Previously the norm was to manipulate only primary sequences and bonded-pairs in the computer programs used for DNA/RNA sequence analysis. I use the concept of a *grammar* and of *parsing* according to a grammar (concepts more normally associated with human languages) here to allow groups of bases to be treated as structural units. This grammar concept is combined with a *pinning* technique to produce the complete insertion tool. Automated alignment insertion could be described as the first payoff of this new organizational approach.

A short summary of the cellular role fulfilled by the ribosome starts off this introductory section. The role of the 16S rRNA subunit within the ribosome is also discussed. Some background information on the computer language (Prolog) used to create the alignment insertion tool follows. I then pose and answer the following questions: (a) What is an *alignment*, what does such an alignment represent, and what is its use? (b) What does *insertion* into such an alignment mean? (c) Why is automating insertion into such an alignment important?

This section concludes with subsections on why the grammar/parser approach was chosen over others and on what a grammar for a biological molecule means. I also describe an algorithm that figures prominently in this work (the Smith-Waterman algorithm).

In the section following this introduction, the overall flow of control through the insertion tool is described. Next, the particular grammar used here, along with the development of an automated grammar generation process, is depicted in more detail. This is followed by separate sections specifically devoted to pinning and parsing components of the complete tool. The results are then summarized and closing comments made.

Almost all the source code for the tool is contained in one file called "parse_combined_ops.pl". A few relatively low-level code fragments are employed from a toolkit that Dr. Overbeek has developed. Also, the code to generate node vectors (done once for a given alignment, a procedure that is described in the section on the pinning component) is contained in a small separate file. We would be happy to distribute all the source code necessary to run the tool to anybody who requests it.

1.1 Biological Role of the Ribosome and Its 16S rRNA Subunit

The central dogma of molecular biology is "DNA makes RNA makes proteins". The first stage, the copying of the information coding for a protein from DNA into RNA, is called *transcription.* The second stage, the actual production of a protein using the instructions now encoded in a piece of RNA called *messenger RNA*, or *mRNA* for short, is called *translation.* This is where the ribosome plays its role; it is the structure in the cell where translation is done. Since the various proteins are crucial to every part of the cell's metabolism, we see that the presence of the ribosome itself is vital to the cell's continued existence. All cells in all organisms currently known contain complexes that can be identified as ribosomes. In other words, the ribosome is a universal structure. The ribosome is typically present in a cell in an extremely large number of copies. (Two examples: there are over 10 million ribosomes in an average mammalian cell; and, depending on the rate of cell growth and division, there can be anywhere from somewhat less than 10,000 to over 30,000 of these structures in a cell of the bacterium E. coli.)

To start the translation process, the ribosome locks onto a linear piece of mRNA (an information "tape", so to speak) which specifies the code for the amino acids in the target protein. It then captures amino acids one by one in the proper sequence (each amino acid being carried in by what is called a *transfer RNA*, or *tRNA*, molecule) and joins the amino acids in a chain to make the desired protein.

All variants of the ribosome are similar in structure and function, both in prokaryotes and eukaryotes. Differences do exist, of course, particularly over such a broad division as that

between kingdoms. (This is one reason why the genetic engineer has a difficult time getting eukaryotic genes properly translated when they are inserted into prokaryotic bacteria.) All ribosomes are composed of one large unit and one small unit. Together the two units form a large complex that contains several pieces of *ribosomal RNA*, *rRNA* for short, (over 50% of the ribosome by weight) and a substantial number of proteins (about 50 proteins in prokaryotes, 75-80 in eukaryotes). The large unit is used to link the amino acids together by catalyzing the peptide bond formation. The small unit provides the docking sites for the mRNA and the tRNAs.

The 16S alignment from Urbana of concern to us here contains solely prokaryotic species. In prokaryotes there are three distinct molecules of rRNA in the ribosome. Two of them (named the 5S rRNA and the 23S rRNA) are located in the ribosome's large unit. The particular rRNA fragment that we work with, the 16S rRNA, is located in the small unit.

While the general cycle of protein formation in the ribosome is understood, there are still substantial mysteries about how the ribosome works and what, in particular, is the function of the rRNA molecule (the 16S rRNA in prokaryotes, the slightly larger 18S rRNA in eukaryotes) contained in the small unit. One fact that we do know about the 16S rRNA is that it ensures that translation of the mRNA is started at the proper site. This is done through base-pairing between a short sequence at the 3'-end of the 16S and a sequence on the mRNA upstream of the translational start site. (This is known as the Shine-Dalgarno interaction.)

We also note that the discovery by Tom Cech and his co-workers in 1982 that RNA molecules could function as enzymes with catalytic activity provided the first solid evidence that rRNA could contribute to the ribosome's catalytic activity (other than by providing binding specificity) [1]. In fact, it is now widely believed that the contributions of the ribosomal proteins are almost completely limited to assembly, stabilization, and optimization and that the rRNA molecules are the parts that perform the basic ribosomal functions. However, much remains to be learned. The best understood version of the ribosome is in the species *Escherichia coli* (*E. coli*), where the most work has been done. A fuller introduction to the ribosome can be found in reference [2]. More information on the 16S rRNA in particular is in [3] (for comparative anatomy of its structure) and in [4] (for structure-function relationships).

1.2 Background Information on Prolog

With the exception of a few subroutines that were coded in the language C for increased speed, our entire alignment insertion tool was coded in a computer language called *Prolog*. This is a somewhat unusual language, which contains some concepts that may be confusing to the typical biologist whose programming experience, at most, consists of a few programs in BASIC or Pascal. Within the confines of this paper, I do not have the space to give an exhaustive description of the language. Nor would this be a wise use of the reader's time, since experts in the field have produced explanations of the ideas inherent in Prolog better than I ever will. I shall limit myself here to pointing out a few relevant features, giving a bit of the history of the Prolog language, and describing some work previously done in the field of molecular biology that has employed Prolog.

Prolog is one of the two leading languages used in artificial intelligence work (the other being a language called *LISP*. LISP was created in the late 1950s at Stanford. Prolog is much newer, having been developed in the early 1970s by Alian Colmerauer in Marseilles, France, and Robert Kowalski in Edinburgh, England.

Prolog's name comes from the term *PROgramming in LOGic*. Prolog is, in fact, an implementation of the processes involved in first-order predicate calculus. A Prolog program is simply a set of clauses. Each clause expresses either a fact or a rule. A typical fact might be

```
struc_unit(b38,71,71,gap,null).
```

This is one of a set of facts used to describe the 16S rRNA molecule. Its name, or *functor*, is "struc_unit". It has five arguments. The first argument (b38) identifies which structural unit we are talking about. The other arguments contain information pertaining to that particular unit. (These arguments are fully explained in the section in this paper devoted to the 16S grammar and its generation.)

A rule has a somewhat different format. A fact simply has a *head*. A rule has a *head* and a *body*, with the two structures separated with the symbol ":-". Here is a sample rule from Appendix 8:

```
cap_type(Bases,pentaloop,14) :-
                    length(Bases,5),
                    Bases = [Base1,Base2,Base3,Base4,_],
                    cap_type([Base1,Base2,Base3,Base4],tetraloop,_).
```

The meaning of this rule is as follows: If the string of bases passed to the cap_type clause in the variable "Bases" is of length five, and if the first four bases form a "tetraloop", then we identify that string of bases as a "pentaloop" and pass back a point count of 14 (what these points are used for is not important here) when the clause is exited. A rule holds true (succeeds) only if all the conditions in its body hold true.

More than one clause can share the same functor name. For example, several hundred *struc_unit* clauses are used in our tool, one for each different structural unit in the 16S molecule. Here are three more:

```
struc_unit(b39_46,72,80,lhs,null).
struc_unit(b47_51,81,87,gap,null).
struc_unit(b52_53,88,89,lhs,null).
```

The collection of all clauses sharing the same functor name (and the same number of arguments) is called a *predicate*. For example, the struc_unit/5 predicate is the collection of all struc_unit clauses used by the tool. (The */5* is added to the name to indicate that five arguments are used in the clauses of the predicate.)

A Prolog program draws conclusions and makes inferences from the knowledge or information contained in the clauses. We say that Prolog is a *declarative* language; that is, the

programmer states the facts and relationships involved with a problem rather than describing a sequence of steps or a specific algorithm. An inference engine built into Prolog performs searches and pattern matching automatically. When a query is made, the inference engine is invoked. The engine then uses a *depth-first* search strategy combined with backtracking to examine the facts and rules in the database (which is formed out of clauses of the program itself) for evidence to answer the query. For readers desiring further information, three good introductions to the Prolog language are given in [5,6,7].

There has been no previous work in Prolog that I am aware of relating to the problem at hand, that is, relating to insertion into an alignment for a molecule. (Nor am I aware of any other work in any language that uses a grammar/parser approach for this specific problem.) However, there has been previous work done in Prolog in the general areas of molecular biology, chemistry, and sequence analysis. Dr. David Searls has been using Prolog to apply a computational linguistics methodology to problems in DNA/RNA sequence analysis [8,9,10]. While only a very few basic constructs have been carried over into our code (the ellipsis operator and the predicates that process base bonding in the sides of a helix), his work was the direct inspiration for the grammar/parser method described in this paper. The basic idea of capturing secondary structural information (in addition to information on the base sequence itself) in a grammar comes from him. I give an detailed example of his work later in this section. Other work in Prolog has been done on describing chemical structures and predicting chemical reactions [11,12] and in describing protein structure and topology in particular [13,14].

There has also been some computational linguistics work in the area of biological sequence analysis that has not employed Prolog as the implementation vehicle. Some recent examples are given in references [15] through [27]. In fact, there has been expanding interest recently among many scientists in applying aspects of linguistic theory to DNA/RNA sequence analysis. Enough attention has been generated so that articles on this subject are starting to appear in the semi-popular press [28].

1.3 What Is an Alignment?

In molecular biology it is often useful to create an *alignment* to help study a particular macromolecule. By means of such an alignment, the correspondences that exist between versions of the molecule in different species can be visually portrayed. As an aid, this is something in the nature of "one picture is better than a thousand words". For example, Figure 1 below shows a small extract from the 16S rRNA alignment that we have worked with.

As you can see, the alignment is a two-dimensional matrix. Each species occupies one row, and each character for a given species fills one column in that row. To make the alignment, *indels* (dashes, representing insertions or deletions) are added to the primary sequence of each species, which has been obtained through sequencing. (The primary sequence for an RNA molecule, of course, is simply a string of characters from the alphabet [A,C,G,U]. The *E. coli* 16S rRNA molecule, for example, has a primary structure of 1,542 such characters.) The addition of the indels causes corresponding parts of the sequences to align visually. (Or at least that is the hope. Different biologists might have different opinions regarding some sections of the molecule, and the alignment that you produce also depends on what correspondences you

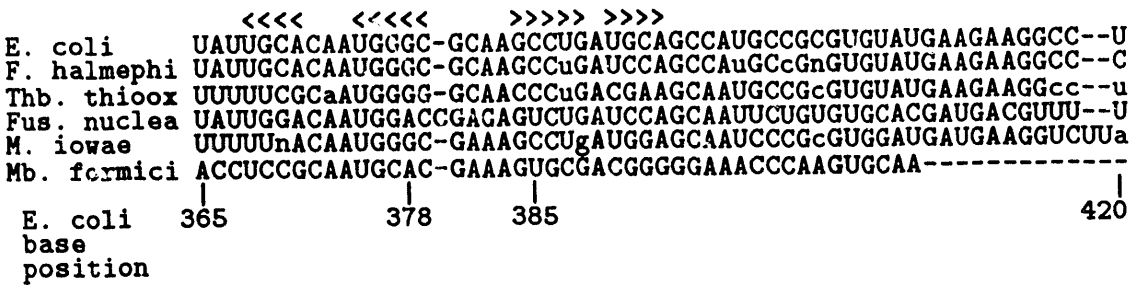


Figure 1: Extract from the 16S rRNA alignment for six species covering the region from E. coli position 365 through position 420

are looking for. More on this below.) The character n is used to indicate ambiguity in that we know some character (base) occupies the given column, but not which one. Lowercase letters (c instead of C, for example) are used to indicate uncertainty.

Accompanying the 16S alignment, an associated *phylogenetic tree* has been built by the Urbana team. This is a data structure called a binary tree whose leaves are the names of the species in the alignment. Each interior node of the tree can be said to represent a hypothetical common ancestral organism of all the species lying on leaves that fall beneath that node in the tree. If a set of species is known to fall into a family, then the interior node that lies at the root of the subtree containing only those species is labeled with the appropriate family name. The tree is used to represent evolutionary relationships between species. The closer two species have been placed in the tree, the closer they are presumed to be in terms of descent from a common ancestor. While algorithms exist to assign numeric distances to the edges (branches) that connect the nodes in the tree ([29]), in the particular tree used for the 16S we do not at this time assign any such numeric value to the edges between the various leaves and nodes. Hence the evolutionary distance between any two species in type of the phylogenetic tree we use in our insertion tool is represented solely by the number of nodes (or edges) that lie between them as one traverses the tree from the first species to the second.

There are many types of alignments, with no one type being accepted as the most "correct" alignment. Different underlying relationships can be represented by different alignments. To summarize here, we can divide alignment types into two broad categories: alignments representing evolutionary relationships and alignments representing structural correspondence. If we have an alignment of the first type, then when two characters (bases) fall in the same column in two different species, the assertion by the people who have created the alignment is that those characters correspond to the same character in the closest common ancestor. In an alignment of the second type, two bases falling in the same column indicate that the bases lie within the same structural unit. For example, the bases that make up the structural unit consisting of the left-hand side of a particular stem-loop construct would lie within the same set of alignment columns in every species in the alignment. Usually an alignment of the first type and an alignment of the second type for the same molecule drop the bases for each species into the same columns; that is, they are consistent with each other. However, there are cases where inconsistencies do arise. (Molecular systematics and the derivation of evolutionary relationships

in general are very complex subjects. For more information see [29,30]).

Some of the more valuable insights that can be gained from genetic sequence analysis are produced by comparing sequences of different organisms. Such insights have motivated the creation of alignments. The alignment of the 16S rRNA sequences that I refer to throughout this paper was produced by a group headed by Carl Woese at the University of Illinois. It now includes sequences for about five hundred organisms (all prokaryotes), and more sequence data are rapidly accumulating. This alignment is of the structural correspondence type. Or, at least to a first approximation, it is of that type. However, there is a complex feedback between the alignment and the accompanying phylogenetic tree that has been concurrently built up. The alignment has been used to improve the tree, and then the tree has been used to improve the alignment, in successive cycles of modification. (I have more to say on these cycles shortly.) At present it would be wisest to say that the 16S alignment reflects both structural unit information and evolutionary information. Or you could say that this alignment is dominated by structural unit information (using both primary and secondary structure, which I describe shortly) but also influenced by evolutionary information where structural unit information is insufficient or ambiguous. Dr. Woese is primarily interested in using this alignment to find evolutionary relationships, that is, in deriving the "tree of life", and has published extensively with results based on the 16S alignment [31,32,33,34]. Other scientists, such as Henry Noller at the University of California at Santa Cruz, are interested in determining in detail the structure of the 16S molecule and in how such structure relates to the function of the ribosome of which the 16S rRNA is a subunit [35]. Here the alignment can be used to find constraints on the possible structure.

I stress the significance of this particular 16S alignment. As the alignment upon which Dr. Woese has depended most heavily, it has contributed in a fundamental way to phylogenetic tree construction in prokaryotes. It has served as one of the best such tools — indeed, perhaps the best yet known.

There are several reasons that Dr. Woese chose an alignment of 16S rRNA molecule to investigate phylogeny. First of all, rRNA molecules (as mentioned above) are universally present. Second, they are constant in at least some of their functions, which ensures relatively good behavior as a molecular clock. Third, different sections of the larger rRNA molecules (such as the 16S and 23S) change at quite different rates. These changes allow a very wide range of phylogenetic relationships to be measured. (At the same time, however, the primary sequence and secondary structure in a rRNA molecule such as the 16S remain sufficiently conserved so that homologous positions can be identified between various species.) Fourth, an rRNA molecule such as the 16S is readily identifiable across a wide range of organisms. (One cannot work with a molecule if one cannot isolate it.) Fifth, the 16S molecule is large enough and contains enough loosely coupled functional units so that its functioning as a chronometer is relatively insensitive to nonrandom changes in one of the units, that is, the chronometer runs "smooth". Finally, there is a practical consideration: an rRNA molecule like the 16S can be sequenced directly (and hence rapidly) by using the enzyme reverse transcriptase. Sequencing technology is changing rapidly nowadays (through the use of the polymerase chain reaction, for one factor), but when Dr. Woese began his studies, this fact was of great significance.

In the 16S alignment the number of column positions exceeds the number of bases contained in the sequence for any individual species. Currently the alignment has 1,892 columns, while no species has more than 1,660 bases in its primary sequence. The great majority of species contain from 1,450 to 1,600 bases. This means that each species must have at least some indels inserted in order to properly fit into the alignment. (Indeed, some species lack any bases in some of the structural units. No bases in a structural unit indicates a complete absence of the structure represented by that unit. The absence is represented by only indels appearing in the set of alignment columns for that particular unit.)

As stated above, the 16S alignment was built not only on base invariancy (matching between species based on the primary sequence), but also on secondary structure. In fact, in order to divide up the 16S molecule into smaller structural units Woese's group relied primarily on secondary structure patterns detected through a technique called *covariance analysis*. Therefore let me say a few words on the prediction of secondary structure and the use of secondary structure in the 16S alignment construction.

The best estimate at present of the secondary structure of the 16S rRNA molecule for *E. coli* is given in Figure 2. (Some species might have a diagram that looks very similar, while other species might be lacking certain regions entirely.) Looking at this diagram (direct your attention to the area slightly below and to the left of center), one can see that positions 375-379 bond with 384-388. Such a structure, in which two sections bond in consecutive positions, is called a helix. The vast majority of bonds in helices are A-U, G-C, or G-U bonds (e.g., the 378-385 bond is an G-C bond).

The two sections are called the left-hand side (*lhs*) and the right-hand side (*rhs*) of the helix. We take the lhs of each helix as the side first encountered when the 16S molecule is traversed in Figure 2 in the direction from base 1 to base 1542. The rhs is always the second side encountered in such a traversal. (The position of the first base in any new sequence fed into our insertion tool corresponds to the start position held by base 1 in Figure 2. The parser component of the tool traverses a sequence in the direction described above. Hence the parser component always reaches and processes the lhs of a helix before the associated rhs.) Make a note of the *lhs* and *rhs* abbreviations, since they will be used throughout the rest of this paper.

If the bases that lie between the lhs and rhs do not form any bonds, then we have a simple gap, or *cap*, between the lhs and rhs. We say that the lhs, cap, and rhs together form a *loop* or a *stem-loop* structure. This is what we have in the above example. In my grammar such a structure is called a *loop_unit*. Looking at the entire region from *E. coli* position 368 to position 393, we see that the loop described above falls between the lhs (positions 368-371) and rhs (positions 390-393) of another helix. Thus a more complex structure is built out of positions 368-393 considered as a whole, namely what we would call a "a helix with an interior loop" (although such a structure might occasionally also be referred to as two distinct helices).

Going back to Figure 1, we see there the portion of the alignment corresponding to positions 365 through 420 in the *E. coli* sequence for six species, including *E.coli* itself. The columns that make up the lhs and rhs of the helix with an interior loop described above are indicated by the "<" and ">" symbols, respectively, at the top of the figure. These symbols represent the bonded positions. Now, if we take isolate the columns that include *E. coli* positions 378 and 385, we obtain Table 1.

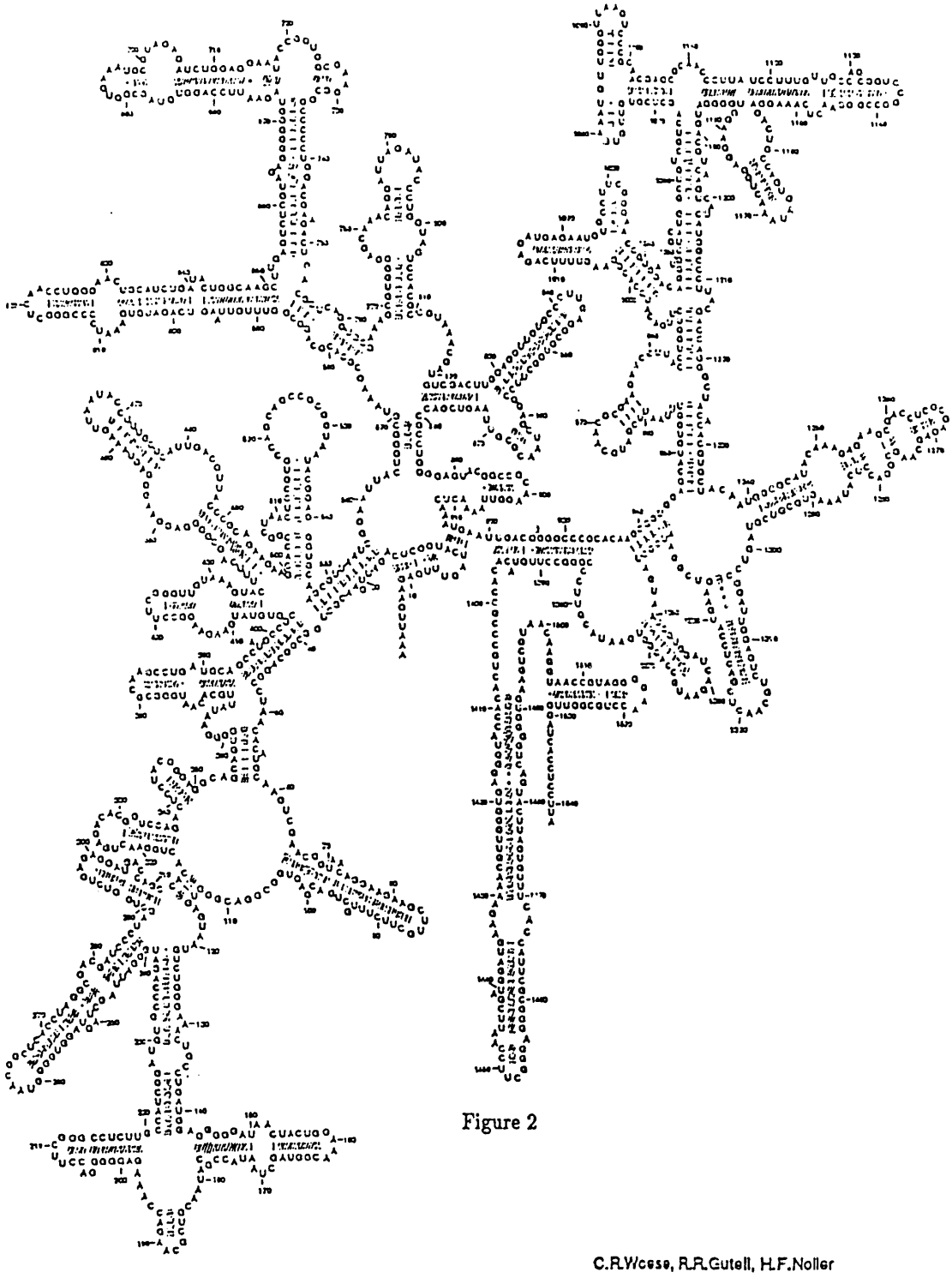


Figure 2: Diagram of the 16S rRNA secondary structure in the species Escherichia coli (E. coli)

Table 1: Contents of six species in the alignment columns corresponding to E. coli base positions 378 and 385

	378	385
E. coli	G	C
F. halmephi	G	C
Thb. thioox	G	C
Fus. nuclea	A	U
M. iowae	G	C
Mb. formici	A	U

Note that in each case the positions contain one of the common bonding pairs (G-C for four of the organisms and A-U for the other two). The columns appear to *covary*. It seems as if a mutation in one position produced a corresponding mutation in the other bond-pair position. Of course, this is not quite the case. A mutation in one of the bonding columns normally produces a nonbonding pair; and if a second mutation does not occur fairly quickly in the other bond-pair column, then the organism will typically suffer from reduced efficiency and thus disappear through natural selection. It is the selection against nonbonding pairs that produces the detectable phenomenon of covariance of concern to us here. (I am simplifying in order to clarify the basic idea. It is possible that a mutation-caused base change at a particluar location would have little or no effect on ribosomal activity, and hence natural selection could not act upon it. However, the concept as briefly described above lies at the heart of covariance analysis.)

The covariance among the six organisms displayed in Figure 1 could correspond to a single "event" or to two events, depending on the phylogenetic relationship of the six organisms. For example, if the organisms were related as shown in Figure 3a, then a single event would suffice. On the other hand, if they were related as shown in Figure 3b, then pair of events (one for *Fus. nuclea* and a separate one for *Mb. formici*) would be required.

This can be seen as follows. In both Figure 3a and Figure 3b all six organisms share a common ancestor (the root node at the top of the small subtrees depicted). Suppose that in this common ancestor we had a G in position 378 and a C in position 385. Then for four species (*E. coli*, *F. hapmephi*, *Thb. thioox*, and *M. iowae*) no event needs to occur, since the G-C bond pair remains as is. The problem then is to explain how the G-C pair became an A-U pair in *Fus. nuclea* and *Mb. formici*. In Figure 3a we see that these two species share a common ancestor not shared by any of the other six species. Hence the simplest explanation is that one event in that common ancestor changed the G-C pair to an A-U pair. (For simplicity in this discussion, I refer to the replacement of both bases as a single event.) The change was then passed on to *Fus. nuclea* and *Mb. formici*. In Figure 3b there is no common ancestor between *Fus. nuclea* and *Mb. formici* that is not also shared by the other four species. Hence the simplest explanation becomes that the G-C to A-U change occurred twice, once when the *Fus. nuclea* species came into being and once when *Mb. formici* separated from the common ancestor of all six species. Hence two separate events took place.

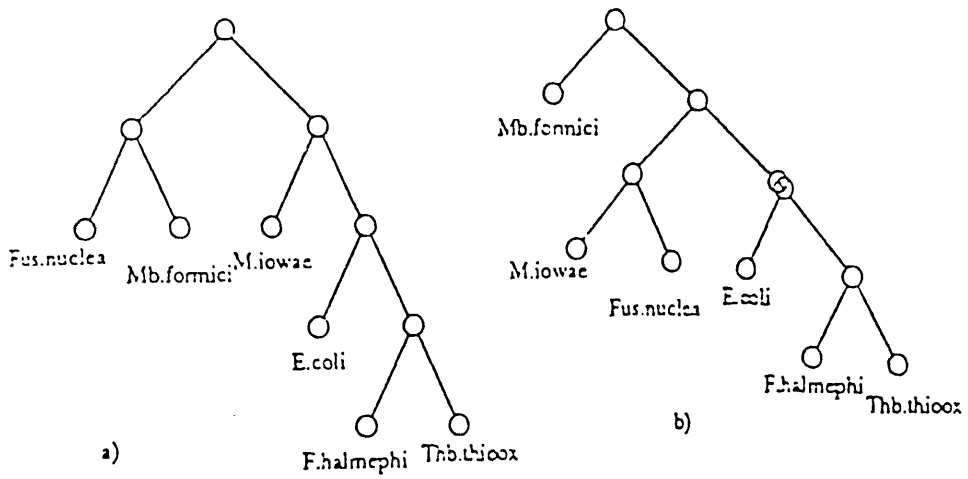


Figure 3: Two hypothetical phylogenetic trees for the six species used in Table 1

Current wisdom places the organisms in the tree given as Figure 3b, indicating two distinct events. By studying the small portion of the 16S alignment shown in Figure 1, along with the tree in Figure 3b, we see that clues exist to support the existence of the following bonds: 379-384, 378-385, 377-386, 371-390, 370-391, and 369-392.

Thus we see that covariance analysis allows the corresponding lhs and rhs of helices to be determined [36]. With these results, secondary structure (the helices) can be used to decompose the molecule into small structural units. Looking at Figure 2, note the large number of helices, which allow the decomposition of the molecule into a sizable number of smaller structural units based on the helix boundaries.

One might think that there would be a simpler means to calculate how an rRNA molecule folds in a particular species, a means that would not require also gathering data from many other species. After all, given the constraints on the characters that bond, one is tempted to believe that it would be simple to write a program that would take the sequence for a given species and produce the "most likely" folded form of the molecule for that particular sequence. Unfortunately, such programs can produce literally thousands of plausible patterns. More accuracy could be gained by looking for configurations that minimize free energy, but such computations are prohibitively expensive for large molecules. Also, there is no hard-and-fast rule that states that a macromolecule must occupy the globally minimized energy state. In the actual cellular environment it might be perfectly feasible for a molecule to occupy one of the deep local minima. Following this route, one can easily fall into a quagmire of calculations that produce no definitive answer, or anything close to a definitive answer [37]. Furthermore, since the 16S rRNA exists in a much larger complex (the ribosome), a fact that cannot currently be included in our energy minimization calculations, such calculations can at best reduce the number of plausible configurations. Hence the key to producing an accurate estimate of the secondary structure has been found to involve comparative analysis between sequences from several organisms by means of the covariance method illustrated in the above example.

From the last several paragraphs, one might conclude that covariance analysis alone built the estimate of the secondary structure shown in Figure 2. That is not accurate. While covariance analysis is indeed the central tool used in formulating such estimates, there are details that

were not made explicit in the above discussion; that is, the three problems of

1. aligning the genetic sequences,
2. producing the phylogenetic tree, and
3. detecting the secondary structure

are all deeply interrelated and, in fact, must be solved simultaneously. We begin with the construction of an initial alignment (via covariance and simple base matching of primary sequence). The initial alignment is then used to get an initial phylogenetic tree and secondary structure. Next, these estimates are used to go back and improve the alignment (because the alignment involves making decisions that reflect alignment of secondary structure). In turn, the new alignment is used to improve the computations of secondary structure and phylogeny, and so on.

These cycles of modification might appear circular, but truly they are not. For example, suppose the initial covariance data allow us to define where helix boundaries lie in parts of the 16S molecule, but in other parts of the molecule the data are too ambiguous for us to break things down into smaller structural units. (We are not sure whether there really are bonds forming between bases in such regions.) What we do is this: we construct an alignment and then a phylogenetic tree with the data we have. The tree can then be used to resolve some of the ambiguities in the regions of the molecule where we earlier refrained from deciding on base placement and unit boundaries. This stage improves our estimate of secondary structure and improves the alignment. We can then generate an improved phylogenetic tree that should resolve even more ambiguities, and so on. (Additional data from various laboratory tests or other sources can also enter into this loop, of course. For one example that checks on rRNA secondary structure predictions by using chemical probing experiments, see [38].)

How does the phylogenetic tree resolve ambiguities in the secondary structure? Here is a concrete example: suppose, for simplicity, that we have six species (rather than the 500 or so in the current alignment). Also suppose that an initial alignment has already been done and that we now wish to examine the base in alignment column M and the base in alignment column N in each of the six species to see whether they form a bond. Now suppose that we have these bases in columns M and N:

	column M	column N
species 1	G	C
species 2	A	U
species 3	C	G
species 4	U	A
species 5	G	C
species 6	A	U

This evidence is clear-cut. No matter how the tree has been set up, it would take several independent events to produce the base variation seen. The simplest (and thus, by Ockham's Razor, best) explanation for all these events (and, just as important, the absence of events that would produce nonbond mismatches such as A-A, A-C, etc.) is that they are necessary to retain

a base pair bond that is useful or vital to the functioning of the molecule. Hence we conclude that a bond does exist between the bases in these two alignment columns, and we do not need to look at the associated phylogenetic tree to do so. (Actually, such a clear-cut bond should have been discovered when we created the initial alignment. I use it here simply in contrast to the example that follows.)

However, suppose that instead of the above base set we instead had this:

	column M	column N
species 1	G	C
species 2	G	C
species 3	G	C
species 4	A	U
species 5	A	U
species 6	A	U

Admittedly, all the pairs are of the type that can form Watson-Crick bonds. But do they really? Could what we see have been produced by random chance instead, with no bonding involved? Note that we only have two types of pairs (G-C and A-U). The answer depends on the guidance we receive from the phylogenetic tree. If all six species share a common ancestor in the tree, with species 1 through 3 hanging off one branch and species 4 through 6 hanging off the other, then we would have very weak evidence for a bond. This conclusion follows because only one event would be needed to produce the above set of bases. For example, if the common ancestor contained a G-C pair, then no event would be needed in the branch containing species 1 through 3, while in the other branch one event changing the G-C pair to an A-U pair in a descendant of the common ancestor of all six species, said descendant serving as a common ancestor of just species 4 through 6, would be all that is needed. The occurrence of only base pairs that form bonds is of note, but it can remain only suggestive if the base pairs can be produced by a single event.

But suppose, say, that species 1 and species 4 lie very close together in one branch of the tree and that the last common ancestral node that they share contained a G-C pair in columns M and N. In such a case an event would be required in the branch containing species 4 in order to change the G-C to an A-U. Now suppose that the same situation occurs between species 2 and species 5 and between species 3 and species 6. We are now talking about a minimum of three separate events at widely separated locations in the tree. The more events we have, the more unlikely is the above outcome of only base pairs that can form Watson-Crick bonds (without mismatches like A-A mixed in) unless such bonds do exist and are useful in the molecule's function. Hence we are guided toward the conclusion that a bond exists between the bases in the columns.

1.4 What Is an Insertion into an Alignment?

The insertion, or addition, of the 16S rRNA sequence from a new species into the 16S Urbana alignment is currently achieved completely manually, either by Dr. Carl Woese himself, or by one of his colleagues. The new species sequence is brought into a customized text editor, along with a subset of the alignment (the whole alignment now being too large to fit on one screen). The human then lines up the bases by eye, typing in the indels (the dashes) one by one.

Although some progress has been made in automatically generating an alignment and in formulating an estimate of secondary structure (an up-to-date reference is given in [39]; no further discussion of this work will be given here, since virtually none carries over to the methodology used in the current version of our alignment insertion tool), most biologists agree that careful analysis by a human expert is still required to produce an alignment that is reliable enough to be useful. Hence most alignments are constructed as in the 16S example above, using an editor (either a standard text editor or a program that has been customized to support alignment of genetic sequences). Further, the actual estimate of secondary structure is often left totally implicit; occasionally it is included in "comment lines" (much like the top line in Figure 1, above the species data), and sometimes it is maintained as a separate file of "known bonds". As researchers begin to use alignments as the basis for formulating and testing numerous hypotheses, it will become increasingly important that the known structural units be represented explicitly, as is done here in the grammar.

1.5 Why Is Automation of Insertion Important?

In the computer age, it is fairly absurd to waste the time of such senior scientists as Woese in a nonresearch task that can be automated. The particular 16S alignment I worked with is already among the largest of its type (if not the largest) in the world. It seems likely that the alignment, which now holds sequences from 500 species, will include thousands of sequences within just a few more years. Hence maintenance issues are becoming increasingly important.

We also expect that a large and growing number of sequences are going to have to be aligned on a weekly basis due to the recent establishment of what is officially known as the Ribosomal Database Project, which will make the Urbana data publicly available. One of the project's goals is to allow a biologist to send in a 16S rRNA sequence via e-mail and get back an aligned version with the indels inserted in the proper positions. Hence the interest of Dr. Woese and of Dr. Overbeek, who is providing computational support to Woese's group, in automating the alignment insertion procedure.

There are many molecules in addition to the 16S rRNA where our best tool for discerning structural features would be the study of cross-species alignment. The total effort involved in aligning the 500 species in the 16S alignment at Urbana over the past decade has been enormous, and the training required for Woese and his team to learn their skill is very large. The initial goal of this project was to automate insertion of a sequence from a new species into the 16S rRNA alignment. However, another goal of the project was to duplicate or encode at least a portion of the skill of the Urbana team in an automated tool that could bring to bear the same sort of expertise on other molecules.

1.6 Benefits of the Grammar/Parser Approach

Those readers familiar with multiple sequence analysis, might raise this question: Why not use one of the various dynamic programming type sequence alignment algorithms (Smith-Waterman, Sellers, or Needleman-Wunsch) across the entire new sequence, matched against all the species already in the alignment?

Unfortunately, that approach does not work very well. Trying to produce an optimal alignment of more than two sequences simultaneously over a molecule the size of the 16S rRNA (almost 1,900 alignment positions) is simply computationally intractable at the moment. (The computational effort is of order L raised to the Nth power, where L is the mean length of the sequences involved and N is the number of sequences [41].) It certainly cannot be done for an alignment of over 500 species. Sophisticated non-optimal alignment methods using various heuristics have also been tried. The results contain enough errors so that such methods are not a viable way to approach the alignment insertion problem. (See Section 1.3.2 in reference [40] for a brief discussion on two of the best of these methods.)

We have chosen a method that combines the concepts of pinning and parsing. First, a pinning program is run that pins a subset of the bases in the new species to given alignment positions. There are some regions that are so constant that virtually all species in the alignment fill those alignment positions with the same bases. Such regions are easy to pin. However, the pinning technique we use goes beyond this to also pin bases that occupy alignment positions where the base composition stays relatively constant within only a small subset of species (perhaps the species belonging to one subfamily of the phylogenetic tree associated with the alignment). Once this is done, the problem remains of filling in the gaps between the pinned bases. We do this with a parser and a grammar extracted from the alignment. For future reference, note that "the parser" refers to the computer program that I have written (combined with Prolog's built-in inference mechanism). The actual physical act or process of parsing is called "the parse". The parse (act) is performed by the parser (program).

1.7 The Meaning of a Grammar for a Molecule

Now, what exactly do the words *grammar* and *parsing* mean in a biological context? A linguist might say that the objective of a parser is to determine whether an input string (in molecular biology, a string of bases) is derivable from the rules of a grammar. That is, the act of parsing would determine whether there is a pattern in the input string that can be recognized according to the rules of the associated grammar.

In standard linguistics a grammar would give a hierarchical breakdown of a sentence into noun clause and verb clauses, then break down the noun clause into adjective and noun, and so on. In the variant grammar that I build, the grammar breaks down a molecule in a similar fashion into structural units. There are the simplest structural units (corresponding to noun, verb, adjective — things that correspond to a single word) and then there is the superstructure of larger structural units (my noun and verb clauses) that can be superimposed to recognize desired patterns.

In the context of RNA molecules, our underlying alphabet is *acgu*, the simplest structural units correspond to indivisible words, the more complex structural units built up from the simpler units correspond to phrases or clauses, and the entire molecule can be considered to be a well-defined sentence. Our situation is this: we know that the sequence of bases handed to us for a new species is supposed to fill the entire 16S molecule. There is no ambiguity about this; we know that the bases are for the 16S and that no bases are going to be left over or left out. Hence the use of a grammar/parser approach here to recognize the pattern of the 16S rRNA

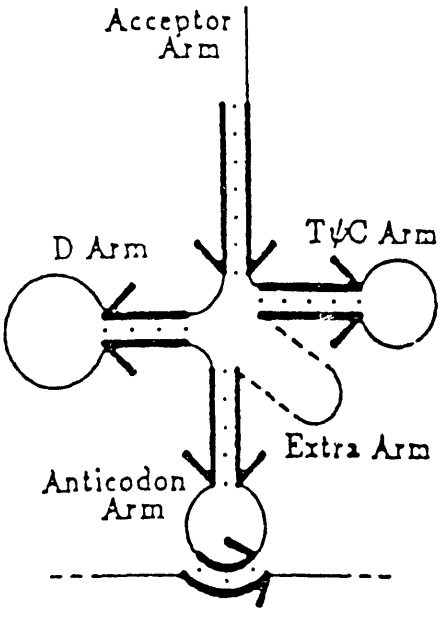


Figure 4: Diagram of typical tRNA secondary structure

in the input string as a whole is not the point. If that was all the grammar could do, then there would be no point in using such an approach here. But (and this is crucial) as the parser engine chugs along the input string to see whether the entire sequence can be fit into one of the allowed patterns for the 16S rRNA (according to the grammar rules), the engine automatically "slots" the bases into words or phrases, that is, into the structural units. This breakdown of of the input string into linguistic subunits by the parser is precisely what we need to perform an accurate alignment.

As mentioned earlier, Dr. David Searls has done some pioneering work in the area of applying grammars to DNA/RNA sequence analysis, and in a sense we are extending his work. One of his grammars is used to search for tRNA genes in genomic sequences. It is quite short, but also quite powerful because it incorporates knowledge of the secondary structure of the tRNA. A diagram of typical tRNA secondary structure is shown in Figure 4, while Figure 5 contains a listing of Searls' grammar itself.

In this tRNA grammar, secondary structure information (the length of the stems and how many mismatches are allowed between the lhs and rhs) has been easily folded into the restrictions on the primary sequence. The grammar has been tested successfully on raw genomic sequences. In the first try, Searls parsed all seven tRNA genes known to be contained in the sequence of DNA that he used. (For our input 16S sequences, where all bases should belong to the 16S molecule, a successful parse means that all bases are used up in the 16S pattern found and none are left over. However, here where long genomic sequences are used and we are trying to recognize a pattern hidden in a longer sequence, a prefix and/or postfix of the base string can remain, and a successful parse means that we have found a subsequence of bases in the input string that fit the pattern defined by the grammar.) The codon (the amino acid) carried by each tRNA was also correctly identified. This is quite remarkable for a dozen-odd lines of code. More information on Searls' work can be found in [8,9,10].

```
% This is Searls' grammar for tRNAs. It returns the expected amino acid
% for the anticodon found (or 'suppressor' if it is a stop codon).

tRNA(AA) -->
    Stem#7, ''t'', base, d_arm, base, anticodon_arm(AA),
    extra_arm, t_psi_c_arm, ~Stem$1, acceptor_arm, !.

d_arm -->
    Stem#4, ''a'', purine, 1...3, ''gg'', 1...3, ''a'', 0...1, ~Stem$1.

anticodon_arm(AA) -->
    Stem#5, pyrimidine, ''t'', anticodon(AA), purine, base, ~Stem$1.

extra_arm --> 3...20, pyrimidine.

t_psi_c_arm -->
    Stem#4, ''gttc'', purine, ''a'', base, pyrimidine, ''c'', ~Stem$1.

acceptor_arm --> base, ''cca''.

anticodon(AA) --> ~[X], ~[Y], ~[Z], {codon(AA) ==> [Z,Y,X]}.

anticodon(suppressor) --> ~[X], ~[Y], ~[Z],
                          {stop_codon(AA) ==> [Z,Y,X]}.
```

Figure 5: Prolog code listing for Searls' tRNA grammar

I am going to describe later in this paper how bases are parsed in a single structural unit. The procedure of parsing across a single unit might at first appear somewhat trivial. The point is that the grammar ties all of the units together and imposes a superstructure on them, allowing them to form "phrases" or "clauses" that can be recognized and treated as a separate objects in their own right. The superstructure, the hierarchy, can be made to reflect the structure of the molecule. Anyone can look at a diagram of the 16S molecule and say that this structural unit starts here and is made up of these bases, that one starts there and is made up of those bases, etc. What makes the grammar/parser technique extremely valuable is that relationships between the various structural units are automatically defined and maintained. The program knows automatically how many bases it takes to reach one structural unit from another structural unit as the molecule is traversed, along with how many structural units are passed through on the way and with what constraints that have to be obeyed. The program knows automatically what structural units are embedded in larger structural units, and precisely all the possibilities into which a larger structural unit can be broken down into smaller units.

To summarize: if insertion into an alignment based on primary sequence is not good enough (and it is not, at present), then one naturally turns to the idea of using the molecule's secondary structure to aid in the insertion. But how do we describe such structure in a form that a computer program can easily manipulate, and at the same time feels natural to the biologist? We believe that the use of a grammar/parser approach meets these needs. A grammar is inherently hierarchical, recursive, and a good overall framework to which other algorithms can be added or in which different types of information (such as primary and secondary structure) can be superimposed on each other. Also, many features of a molecule can be expressed clearly in a grammar. A grammar can stand as a concise, human-readable description of a molecule.

1.8 The Role of the Smith-Waterman Algorithm

The Smith-Waterman algorithm is a member of a class of dynamic programming algorithms used for comparison and alignment of molecular sequences. It has an important role in our alignment insertion tool. It is used in two places, for two different purposes:

1. First, it is used to score different helix configurations in what are called *loop_units* in the grammar.
2. Second, it is used to find the best match in the alignment for those bases of a new species lying in a particular structural unit.

I have more to say about each of these cases later on. Because of its prominent use in our tool, I shall point out a few features of the Smith-Waterman algorithm in particular, and of the class of dynamic programming sequence comparison algorithms in general.

Dynamic programming algorithms might be described as algorithms that take a "divide and conquer" approach. They break up the problem (usually an optimization problem of some sort, such as finding an optimal alignment between two strings) into subproblems, saving the solutions to the subproblems so that the subproblem calculations are done only once. An optimal solution to the subproblems yields an optimal solution to the original problem.

An alignment between two sequences is evaluated by means of a similarity score that is based on the deduction of points for each substitution, deletion, and insertion needed to get the sequences to match, and on the addition of points for each pair of matched elements in the sequences. The configuration of the two sequences with the best score is the one having the most points. Several other methods of sequence alignment do not provide such an explicit evaluation score, and hence lack a well-defined and easily understandable criterion for choosing between configurations. (One example is the system of Korn, Queen, and Wegman; see reference [42].) The dynamic programming approach is guaranteed to find the configuration with the best evaluation score within the preset parameters. Note that the score is available as a separate entity for other uses, such as determining whether two sequences show any similarity other than that arising by chance. (Such a test could be performed by comparing the score for the two original sequences against the scores for random permutations of the sequences.)

All alignment methods incorporate some sort of parameters. In the dynamic programming method the parameters are the weights assigned to the penalties for the various substitutions, deletions, and insertions. The idea here is that the less likely a change would be biologically, the larger the penalty we assign. For example, one substitution event is more likely than two separate insertion and deletion events, so we bias the score toward the alignment using the substitution by making the substitution cost less than a deletion and insertion combined. The type of parameters used by the typical dynamic programming algorithm impose *soft* limits. For example, if too many deletions occur, then the evaluation score is reduced, and the trial alignment using those deletions will not be picked as the optimal alignment. This approach is more realistic than a sharp limit that, say, permits four deletions without penalty but positively forbids five or more deletions. (Dynamic programming algorithms can also use sharp limits

where useful; it is simply the case that soft limits are better in scoring biological sequence alignments.)

The prime distinguishing characteristic of the Smith-Waterman algorithm is its use of a linear gap cost function. There are other dynamic programming algorithms that set the gap cost (the cost of K consecutive insertions or deletions) to K times g1 (the cost of one insertion or deletion). However, this means that two separate deletions, each of length one, would cost the same as two consecutive deletions. Since the consecutive deletions can be caused by one event while two separate deletions take two events, the consecutive deletions are more likely and hence should cost less in order to reflect biological reality. The Smith-Waterman algorithm does just that by setting the cost C of K consecutive indels to

$$C = U + (V * K) \quad .$$

The value of the U and V parameters used in our tool are -36 and -6, respectively. (These values were suggested by Dr. Overbeek, based on his previous work in the area of sequence alignment.) Hence the cost of two separate deletions is -84 $(-42 + -42)$, while the cost of two consecutive deletions is -48 $(-36 + -6 * 2)$. We add +18 points for a identical match between bases in the two sequences being compared, so in the scoring that the Smith-Waterman algorithm uses for a projected alignment one can see that the cost of one deletion (-42) slightly outweighs the bonus points we get for lining up two bases in the first sequence with two identical bases in the second sequence $(+36 = 18 + 18)$.

At the end of its calculations, the Smith-Waterman algorithm (like any dynamic programming algorithm) through a traceback mechanism automatically provides a correspondence for the optimal alignment found for the two sequences being compared. This correspondence tells us which bases match and how to fill in the gaps in the most biologically realistic manner (according to the set parameters). From this we generate the alignment of the two sequences. The alignment consists of two lines, one species per line, with dashes inserted among the bases in each species to line up the bases that are found to be in correspondence in both species. Another way of stating what the Smith-Waterman algorithm does is this: the algorithm gives us the score and correspondence for the alignment that results from the sequence of elementary operations (substitution, insertion, and deletion) that minimize the cost for transforming the first sequence into the second (or vice versa). This, after all, is what we want: to show which bases correspond in the two species and to show the remaining bases that do not correspond (i.e., those bases that must be deleted or inserted to get from one sequence to the other) in a pattern with interwoven dashes (indels) in a way that reflects the most plausible deletion and insertion events needed to transform the sequences into each other.

Note that I am using the term *alignment* here to represent something a bit different from the 16S alignment described in Section 1. If the 16S alignment were built solely on the primary sequence, then aligning species by minimizing the evolutionary distance between them (distance being the number of elementary operations or evolutionary events that must occur to transform the primary sequences into each other) would be fine, and the 16S alignment would represent the same sort of data as the alignment described in the preceding paragraph. However, the 16S alignment does indeed take into account secondary structure, while the Smith-Waterman algorithm and similar algorithms do not use secondary structure in the calculation of distance.

This is why using the Smith-Waterman algorithm alone does not produce an optimal 16S rRNA alignment. Humans utilize more information in making their judgments.

Let us consider two concrete examples before passing on to the next section. First is a sample alignment between two sequences showing how our program would work if it was using the Smith-Waterman algorithm in its search for a best match to a sequence. The first sequence is *aauuuccggg* and the second sequence is *aaggggguuucc*. The optimal alignment found has a score of six. Counting from zero, the correspondence produced (which is displayed as a list of Prolog *pair/2* facts) shows that the zeroth base in the first sequence corresponds to the zeroth base in the second sequence (*pair(0,0)*), the first base in the first sequence corresponds to the first base in the second sequence (*pair(1,1)*), the second base in the first sequence corresponds to the seventh base in the second sequence (*pair(2,7)*), and so on. The optimal alignment is automatically constructed from the correspondence.

```
Seq1: aauuuccggg
Seq2: aaggggguuucc
Score: 6
Correspondence: [pair(0,0),pair(1,1),pair(2,7),pair(3,8),
                 pair(4,9),pair(5,10),pair(6,11)]

the optimal alignment:
   AlignedSeq1: aa-----uuuccggg
   AlignedSeq2: aaggggguuucc---
```

The second example is a sample alignment showing how the Smith-Waterman algorithm is used to score helices. Since we compare the lhs and rhs of a helix, we wish complementary bases that form bonds to correspond. Note that our program reverses the order of bases in the lhs before making the comparison.

```
lhs of the helix: uagcc
rhs of the helix: gcua
Score = 30
Correspondence = [pair(0,0),pair(2,1),pair(3,2),pair(4,3)]

the optimal  alignment:
     reverse order      true order

lhs    c c g a u        u a g c c   ->
rhs    g - c u a        a u c - g   <-
```

The higher the score, the better the helix we have. As the score increases, we expect the helix to contain fewer substitutions, insertions, and deletions.

We use matrices to store the cost values for mismatches. The Smith-Waterman algorithm looks up in such a matrix the cost defined, for example, when we know that one base is *c* but we know only that the other base is a purine (but not which purine). Obviously we need two such matrices, one for alignment of similar sequences and one for helices. The current matrices employed are shown in Appendix 1.

As a final note to this section, I point out that bonds other than Watson-Crick (A-U, G-C) are allowed in helices in our scoring scheme. Woese's group have found other bonds that are

tolerated in RNA helices. Such bonds are of lesser strength than the standard Watson-Crick bonds, but they do exist and they are important. After consultation with a senior member of the Urbana team [43], we came up with this ranking for use by the Smith-Waterman algorithm:

```
A-U bond :  18                  (100% of max value)
C-G bond :  18                  (100% of max value)
G-U bond :  13                  ( 86% of max value)
G-A bond :   3                  ( 58% of max value)
C-A bond : -11                  ( 31% of max value)
U-U bond : -11                  ( 31% of max value)
G-G mismatch:: -18              (  0% of max value)
A-A mismatch:: -18              (  0% of max value)
C-C mismatch:: -18              (  0% of max value)
U-C mismatch:: -18              (  0% of max value)
```

More information on the Smith-Waterman algorithm can be found in [44,45,46,47].

2 Overall Flow of Control in the Insertion Tool

The overall flow of control through the entire tool is shown in Figure 6. A more detailed description runs as follows:

a) The pinning component of our tool, taking as input the new sequence, the alignment, and the phylogenetic tree, outputs a set of pins and a subtree of the phylogenetic tree. (For an detailed explanation of what a pin is, see the first paragraph of Section 4 below.)

b) Three small programs are run after the pinning component and before the parser component. The first uses the set of pins outputted in (a) to find those structural units that are pinned at both ends. Since both ends are pinned, the program here can automatically slot the bases that fall between the two pinned ends into that unit without waiting for any aid from the parser component. A list of such pinned units with their bases is then fed into a second program, which checks to see whether the bases slotted into a unit obey the rules in the grammar for that unit. If not, we have a strong indication that at least one of the two boundary pins was bad (incorrect), so we delete that unit from the list of pinned units. (This has been a very rare occurrence so far, but a check is useful.) The remaining units that pass are outputted with their bases in a list that is passed on to the parser. The third program takes the subtree from (a) as input and outputs the set of families in the phylogenetic tree that have at least one member species in that subtree.

c) The parser component takes as input the new sequence (a string of bases), the structural unit definitions, the grammar, the set of pins from the pinning component, the set of (verified) pinned structural units found in (b), and the set of families found in (b) from the pinning subtree. It parses the new sequence into the structural units and then inserts the indels to properly position the bases. Its output is the aligned sequence, ready for inspection and insertion into the alignment.

All these (sub)programs are hooked together into our one unified tool, so the user has to type in only one line to start the process. The run can then proceed unattended. At the end of the run, output will be stored automatically in a file in the format shown in Appendix 4.

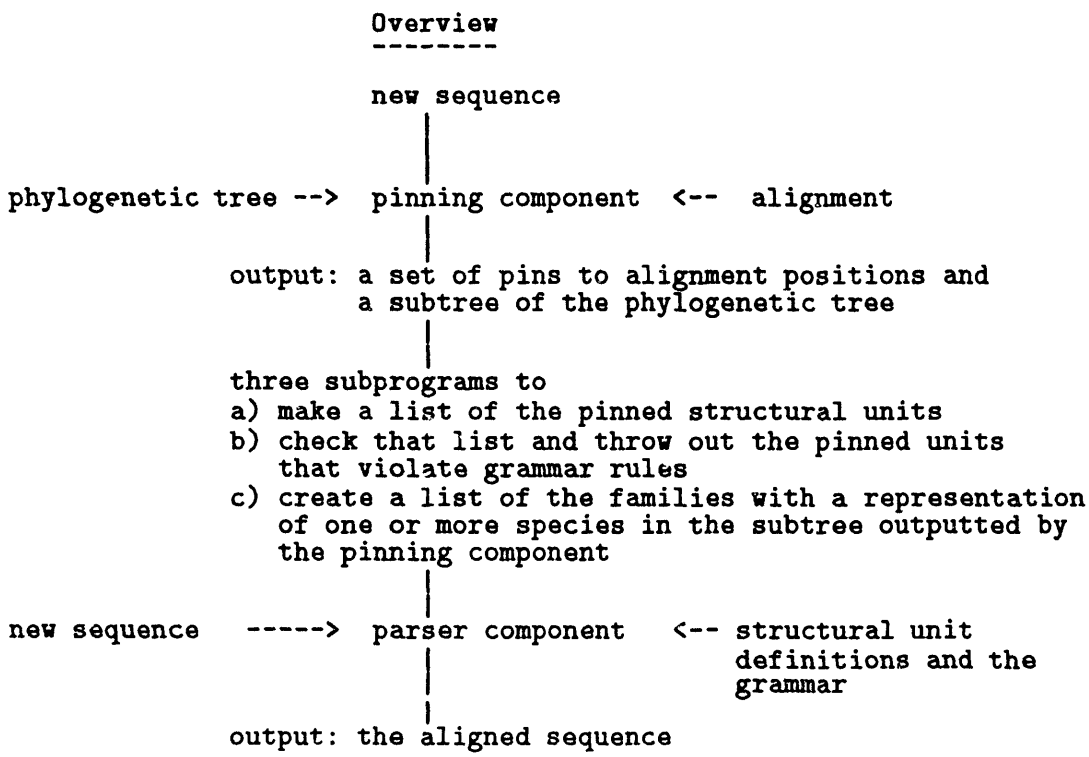


Figure 6: Diagram showing overall flow of control through the alignment insertion tool

3 The 16S rRNA Grammar

Our particular grammar works as follows: Dr. Gary Olsen annotated the Urbana alignment for us, using special characters to indicate the breakdown of the alignment columns into the various structural units. (A structural unit in this context can be a lhs of a helix, an rhs of a helix, or a gap, with the understanding that a gap can either form the cap of a helix or lie completely outside all stem-loop structures and simply form the linking bases between the last helix and the next.) This *annotation* is shown in Appendix 2.

I made one modification to this annotation by marking out the one pseudoknot (really a tertiary interaction; see [48]) in the 16S molecule with my own symbols. Using the modified annotation, I then generated automatically a list of the structural units in Prolog clause format. A brief extract from the list containing data on four structural units is shown below:

```
struc_unit(b64_65,100,101,gap,null).
struc_unit(b66_82,102,133,lhs,null).
struc_unit(b83_86,134,141,gap,null).
struc_unit(b87_103,142,174,rhs,lhs(b66_82,102,133)).
```

Each of these Prolog facts contains five arguments. The first argument simply names the structural unit. We create a unique identifier by using the base positions in the *E. coli* species that lie at the start and end of the unit. The unit name of *b66_82*, for example, means that bases 66 through 82 in *E. coli* lie in the alignment columns assigned to this structural unit. One point about this naming convention: in certain units the *E. coli* species lacks any bases, so we must use something like, for example, *b432_after_and_before_433*, as a name. The same problem would crop up in any species selected to name the units. All the names are generated automatically by the program, which converts the alignment annotation into the structural unit Prolog facts. The second and third arguments give the starting and ending alignment columns that mark the bounds of the unit. The *b83_86* unit, for instance, occupies alignment columns 134 through 141. The fourth argument gives the unit type (gap, lhs, rhs, loop_unit). The last argument (in rhs units only) is used to hook together the associated rhs and lhs of a helix. (In other units a placeholder *null* is used.) In the sample listing of structural units above we have a gap followed by a complete stem-loop structure whose lhs is *b66_82*, whose cap is *b83_86*, and whose rhs is *b87_103*.

I manually added entries to the list of structural units for a half-dozen more complex structural units called *loop_units* to aid in the parse. This is a simple process that takes ten to fifteen minutes (and will rarely need redoing). The parser can function without loop_units, but at least in the 16S rRNA it appears to function better with a few added. The purpose of loop_units is explained more fully below.

Taking the list of structural units and the alignment contents as input, another Prolog program then automatically generates a grammar for the alignment. This grammar is actually quite simple to understand. For each structural unit and each named family in the phylogenetic tree, there is a rule in the grammar that expresses the base invariancy constraints obeyed by the species belonging to that family over that particular structural unit. Also, for each helix formed by two structural units, that is, by a lhs and a rhs, the secondary structure constraints

obeyed by the helix for each named family in the tree are set down in the form of the maximum mismatches, insertions, and deletions allowed in the helix.

3.1 Features of the 16S rRNA Grammar

Grammar information is now stored in the six Prolog predicates described below. Five predicates consist of collections of facts that have to be processed (interpreted) by other Prolog predicates. The *constraint* predicate, however, is a collection of rules (in what is called *Definite Clause Grammer* format), and each such rule is processed in the parse by simply calling it in-line with the proper parameters passed to it.

For those readers unfamiliar with Prolog, there is no need to grasp the fine points of what follows. What is important is the type of information made available to the parser. The grammar predicates are as follows:

```
1) gap(Interval_name,Family,Msg,Min,Max).
```

In standard Prolog terminology this is called the *gap/5* predicate, since it has five arguments. A gap/5 clause is used to build a sub-goal for parsing the interval (structural unit) given by *Interval_Name* for family *Family*. The third argument (*Msg*) can be ignored in this predicate and in all the other predicates I discuss below; at present it is an unused placeholder for any comment (message) we wish to insert. The *Min* and *Max* arguments tell us that *Min* to *Max* number of bases are allowed in this unit.

```
2) lhs(Interval_name,Family,Msg,Min,Max).
```

An *lhs/5* clause is used to build a subgoal for parsing the left side of a helix having *Min* to *Max* number of bases.

```
3) rhs(Interval_name,Family,Msg,Mismatches_allowed, Insertions_allowed,
       Deletions_allowed,Lhs_interval_name).
```

An *rhs/7* clause is used to build a subgoal for parsing the right side of a helix with the permitted number of mismatches (substitutions), insertions, and deletions. Note that, unlike gap/5 and lhs/5 clauses, rhs/7 clauses do not incorporate length (*Min* and *Max*) information. Such data for an rhs is kept in its associated *constraint* clause, along with any base invariancy information for the rhs. The *Lhs_interval_name* argument identifies the lhs associated with this rhs in the complete helix.

```
4) sequence_parsed_by_family_set(IntervalName,
              Family,Msg,MinBases,MaxBases,
              [ (StrucUnitList,FamilyList) ]).
```

The list of structural units to use in the parse is found with a call to this one *sequence_parsed_by_family_s* clause in the grammar. *IntervalName* is set to *b1_1542*, since there are 1,542 bases in the *E. coli* 16S molecule. The *StrucUnitList* argument is a list that defines the entire set of 16S rRNA structural units in their proper order. The *Family*, *Msg*, *MinBases*, *MaxBases*, and *FamilyList* arguments are currently not used.

```
5) constraint(Interval_name,Family,Msg) -->
              <list of constraints, such as 'an,3..3,cn,gn'>.
```

Each gap/5, lhs/5, and rhs/7 clause has an associated *constraint/3* clause. The constraint/3 clause captures the base invariancy information. For example, suppose we had

```
constraint(b1358_1364,'Green Nonsulphur',no_msg) --> un,cn,an,gn,
                                                     0...1,cn,rn,un.
```

This clause states that in structural unit *b1358_1364* all the species in the alignment belonging to the *Green Nonsulphur* family must have this base composition: a *u*, followed by a *c*, followed by an *a*, followed by a *g*, followed by zero to one bases of any type, followed by a *c*, followed by an *a* or *g*, followed by a *u*. (Note that we allow unknown bases represented by *ns* to satisfy any of the base invariancy codes we use. That is why the code for *u* is given as *un*, and so on.) The standard sequence analysis coding scheme is used. Thus *rn* means an *a* or *g* (or an *n*), *yn* means *c* or *u*, and so on. The coding scheme is shown in Appendix 3.

The procedure used by the grammar generation program to create such a constraint clause is fairly simple in concept. For the example above, the program first checks the phylogenetic tree to find out what subset of species in the alignment lie within the *Green Nonsulphur* family, that is, which species lie in the subtree whose root node is labeled *Green Nonsulphur*. The program then checks the structural unit definitions for the Prolog clause corresponding to the *b1358_1364* unit. This clause will tell the program which alignment columns fall within the unit. In this particular case, columns 1690 through 1697 lie in unit *b1358_1364*. The program then goes through the unit's alignment columns one by one, to see what each column contains. In this example, the program found that all species in the *Green Nonsulphur* family contained a *u* (or an *n*) in the first column (column 1690) belonging to the unit, all species contained a *c* in the second column (1691), and so on. Note that if one or more species lacks a base in a particular column, that is, has an indel in that column, then no base invariancy constraint whatsoever is obeyed in that column. The only constraint we can set for that column in that particular family is that zero to one bases may exist in the column. That is, we can make some statement on the minimum and maximum number of bases allowed but not on the type of bases. In the example above, we see exactly this sort of result in alignment column 1694, where the only statement the grammar generation program can make is that *0...1* bases can lie in column 1694 for a species belonging to the *Green Nonsulphur* family. Note that such a situation can extend over more than one alignment column. If one or more indels had been found in column 1695 in the above example, then we would have had the two constraints *0...1*, *0...1* in succession for columns 1694 and 1695. In such cases we would wish to combine the two (or more) constraints on base number into one more concise constraint. However, before we could do so, we would have to check on the minimum and maximum number of bases falling into

the two columns combined. Otherwise we would not know whether we should use *0...1* or *1...2* or *1...1* or *0...2* as the combined constraint. The grammar generation program automatically does such rechecking before creating the single combined constraint on the number of bases in two or more adjacent indel-containing columns.

```
6) loop_unit(Interval_name,Family,Msg,
             Lhs_interval_name,Cap_interval_name,Rhs_interval_name,
             (Cap_interval_type,ListofCapSubIntervals),
             WiggleFactor,Variance,
             LocalLookAheadNumber).
```

The idea here is to treat a *(lhs, cap, rhs)* triplet as a single unit called a *loop_unit.* When this is not done, it sometimes is the case that the Prolog helix pairing rule used in the parser can find the helix, but the resulting helix will be off by one base (or perhaps two bases) from the helix found by using expert human judgment in the Urbana alignment. The rule will always find a helix (if there is one present) within the limit of the constraints we give it (SubsAllowed, InsertionsAllowed, DeletionsAllowed), but because some leeway must be given (that is, SubsAllowed, InsertionsAllowed, DeletionsAllowed must frequently be a bit greater than needed for a particular species) in order to succeed with all species in a family in the alignment, the helix pairing rule cannot guarantee returning the helix correctly in all of its bases. To address this problem, we do the following for a loop_unit clause (the explanation goes into the actual parsing process, so readers unfamilar with Prolog may want to skip the explanation below; then, after reading Section 5 on the parser component, later come back):

a We first parse the constituent parts (lhs,cap,rhs) of the loop_unit as we would ordinarily do if we were treating them separately.

b If (a) fails, we backtrack as usual. However, if (a) succeeds, then we perform additional steps before going on to the rest of the parse. To ensure that the bases have been optimally divided up between the loop's three subunits (lhs, cap, rhs), we "wiggle" the components and/or vary the number of bases passed on to the structural unit following the loop_unit, with the amount of "wiggle" being given by the WiggleFactor argument and the change in the number of bases passed on by the Variance argument. For example, if we have WiggleFactor = 1 and Variance = 0, then we vary the base configuration that succeeded in (a) in eight ways to try out nine configurations in all:

```
lhs-1, gap+2, rhs-1
lhs-1, gap+1, rhs
lhs-1, gap,   rhs+1
lhs,   gap+1, rhs-1
lhs,   gap,   rhs    <- (the original successful configuration)
lhs,   gap-1, rhs+1
lhs+1, gap,   rhs-1
lhs+1, gap-1, rhs
lhs+1, gap-2, rhs+1
```

If we have WiggleFactor = 1 and Variance = 1, then we examine 27 base configurations: First, we do cost calculations for the same nine configurations as is done above. Second, we do the

same nine cost calculations for the nine configurations that result after one base is added at the ending boundary of the rhs so that the rhs is lengthened by one base. Third, we again do the same nine cost calculations for the nine configurations that result after one base is deleted at the ending boundary of the rhs so that the rhs is shortened by one base.

Note that the additions and deletions used in the set of nine cost calculations when wiggling are made only to the ending boundary of the lhs and the starting boundary of the rhs (i.e., at the interface between the cap and the left-hand and right-hand sides). Using a variance other than zero adds the capability of modifying the base placement at the ending boundary of the rhs.

The Smith-Waterman dynamic programming algorithm (modified for helix testing) is used to compare the helix formed by the lhs and rhs of the above configurations. The base subsequences in the lhs, gap, and rhs that make up the helix formed with highest net score (Smith-Waterman similarity score + bias for cap type) are the three subsequences returned. Presumably this would be the helix found by using the best human judgment. Note that we allow certain types of cap to be weighted. Dr. Woese, Dr. Robin Gutell, and Dr. Steve Winker have discovered some cap motifs in the 16S of which I have made use [49]. While most cap types are assigned a score of zero and hence do not affect the net score, a cap type such as a predominant tetra-loop motif can add some points to the total similarity score, thus biasing the parser to the configuration whose cap forms a tetra-loop. Further discussion on this subject can be found in Appendix 8.

c The third and last step is to parse the string passed to the cap and break it down into substructures, if necessary. If the cap is a simple gap (which it is for all loop_units currently in use), then the (Cap_interval_type,ListofCapSubIntervals) argument is filled in as (gap,[Cap_interval_name]). Other, more complicated possibilities for the cap were once envisioned and coded for, but they no longer appear necessary and are not discussed here.

LocalLookAheadNumber is an argument to tell the parser the number of structural units we wish to look ahead. The idea here is that, since performing all the wiggling and calculations in a loop_unit is quite costly relative to parsing other types of units (gaps and so on), we would like to have some assurance that the effort is not going to be wasted. Ideally we would like to ensure, before doing all these time-consuming calculations, that we would not need to backtrack over the loop_unit later on because the parse failed at a later point. While we cannot make such a guarantee, looking ahead by a certain small number of structural units (a number given in LocalLookAheadNumber) is a step in that direction and will avoid some wasted effort and thus speed up the parse. For N (= *LocalLookAheadNumber*) structural units after the loop_unit we check whether the bases that would be passed on will meet the base invariancy (constraint/3 clause) restrictions on those intervals. We do this before we start wiggling. Although this means that we, in effect, parse this block of structural units that fall after the loop_unit twice (at least in terms of their base invariancy constraints), this process is still much cheaper than simply wiggling without any guidance. If the bases to be passed on fail the lookahead test, then we skip wiggling for that variance (we do the test for each different variance, that is, for each different set of bases to be passed on). If the lookahead test fails over all variances, then we skip wiggling entirely and exit the loop_unit with failure at that point, backtracking to

```
gap(b59_60,'Lactobacillus/Bacillus',no_msg,2,2).
gap(b59_60,'Low G+C Gram Positive',no_msg,2,2).

lhs(b9_13,'Fusobacteria',no_msg,5,5).
lhs(b9_13,'High G+C Gram Positive',no_msg,5,5).

rhs(b21_25,'Fusobacteria',no_msg,0,0,0,b9_13).
rhs(b21_25,'High G+C Gram Positive',no_msg,1,0,0,b9_13).

loop_unit(b66_103,'Methanococcales',no_msg,b66_82,b83_86,b87_103,
   (gap,[b83_86]),2,0,10).
loop_unit(b66_103,'Methanobacteriales',no_msg,b66_82,b83_86,b87_103,
   (gap,[b83_86]),2,0,10).
```

Figure 7: Sample Prolog clauses from 16S rRNA grammar

the structural unit that came before the loop_unit. The use of this lookahead test may make the point of failure returned by the parser for parse failures appear to come earlier (at the loop_unit) than it really is. The trade-off in increased parser speed is, I believe, worth this increased diagnostic imprecision.

LocalLookAheadNumber is set to a default of ten. However, at present this default is ignored and the actual LocalLookAheadNumber used is calculated on-the-fly by the parser, which simply extends the number of units to look ahead to the unit immediately preceding the next pinned unit.

Two examples each of gap/5, lhs/5, rhs/7, constraint/3, and loop_unit/10 clauses from the actual grammar we are now using are shown in Figure 7.

From all families and structural units, over 20,000 of these Prolog grammar clauses are produced in total for the 16S rRNA molecule and saved into a file for use by the parser. (Excluding loop_units, there are currently 392 structural units defined, and clauses for each of the 26 families used by the parser are created for each grammar predicate that applies to a given unit.)

How many clauses pertain to a particular structural unit? Each elementary structural unit will have 26 gap, lhs, or rhs clauses (one for each family). There will also be 26 constraint clauses, so the total comes to 52 (26 + 26) clauses per unit. Structural units of the loop_unit type (of which there are six at present) do not use constraint clauses (they rely on the base invariancy constraint clauses of their three subcomponents), so each unit of this type simply has 26 loop_unit clauses in the grammar.

3.2 Development of Automated Grammar Generation

Automation of grammar generation developed gradually. When I first started working on this project in January 1990, Dr. Overbeek had already developed a few basic tools to generate base invariancy statistics from the alignment. However, there was nothing that automatically generated any of the Prolog facts or rules for a grammar. To make a constraint/3 clause for

a particular family and structural unit, I had to take the base invariancy statistics for the alignment positions making up that structural unit (using the species in the designated family) and manually type in the resulting constraint rule. Building the rhs/7 facts with the tightest allowable substitutions, deletions, and insertions was even harder. To begin, I typed in an rhs fact with a large default number for the maximum substitutions, deletions, and insertions allowed. I then tried parsing over and over with that fact, gradually tightening (lowering the maximum) the constraints on the helix until the parse would fail in one or more species. I did this in order to come up with an rhs/7 fact that would parse as precisely as possible. Obviously, this was tedious and very time-consuming. You can see that the task of constructing the grammar was quite laborious. Further, this same task would have to be done over again whenever one or more new species were added to the existing alignment and we wished a new grammar to catch the information in the new alignment contents.

However, it was soon clear that both the data (the base statistics and the definition of the structural units) and the means existed to automate this task. I therefore wrote a Prolog program to do so. (The generation of a rhs/7 clause was handled by getting the optimal helix configuration in each species via the Smith-Waterman algorithm and then counting up the substitutions, deletions, and insertions used in that configuration. The maximum values found over all species tried were the values placed in the rhs/7 clause as the allowed maxima.) On a Sun workstation, it now takes from three to six hours (depending on workstation type) to regenerate the complete grammar for the 16S rRNA. Hence changes can be made to the structural unit definitions, the new definitions can be fed into the grammar generation program, and a new grammar using those changed definitions can be produced in a single day.

Over the course of development, as I noticed where problems were occurring, I have made some minor modifications to the structural units defined by Dr. Olsen's annotation. These modifications involved the consolidation of certain structural units into larger units. The units still reflected the secondary structure, but the consolidation helped the parse speed and its accuracy. These consolidations reflected an expert human programmer's judgment (mine) and I have not attempted to automate them. The parser would work without them, but somewhat more slowly and a bit less accurately.

Let me give an example. Suppose a stem-loop structure lies in a extremely variable region. Also suppose that in the annotation the rhs of this helix is split in two, with a gap defined in the middle to show a bulge (and hence we also have the lhs split in two to match the two rhs units). If the helix sides were fairly short in the first place, then the secondary structure constraints might become so loose for each subdivision of the rhs that they are no longer of any aid to the parser. More specifically, suppose that in a typical new sequence the full-length rhs defined by my consolidation contains six bases and that the two rhs subdivisions contain three and two bases, respectively (with one base going in the bulge unit). Now suppose that our automated grammar generation program, when the structural units containing my consolidation are fed in, reports back that a maximum of two substitutions are used by any alignment species in the rhs, and thus we can set *2* as the maximum allowed in the grammar clause for that rhs unit. If there are a couple of species in the alignment that use two substitutions at the upper end of the rhs and another species or two that use two substitutions at the lower end, then when the rhs is split in two, as in the original annotation, the grammar generation program is going to have to allow two substitutions in both the shorter rhs units. Since the rhs units are now two

and three units long, allowing two substitutions means just about any base sequence can satisfy the secondary structure constraints for those two units. Hence the parser can actually make more use of the secondary structure information when it acts on the longer full-length rhs, even if it cannot place a base into quite as small a box. That is, the box is somewhat larger, but we have a better chance of placing the correct bases in it. That is the prime reason I would consolidate the two rhs and the bulge into one rhs unit and the two corresponding lhs units into one lhs unit. However, cutting down on the number of structural units also cuts down on the amount of backtracking we do when a parse failure occurs. (I will not get into the fairly complex reasons for this here; see the discussion later in Section 5.1.2 on backtracking.) Hence consolidation can also speed up the parse of a species.

3.3 Uses for the 16S rRNA Grammar

A final point about the grammar. In addition to acting as the information pool for the parser component of our insertion tool, the grammar can also serve us in other ways. First, it can perform as an exploration tool in itself. For example, the grammar might be changed to lengthen the lhs and rhs of a helix by one base each, without altering the alignment itself. This might be done to see the effect of a longer helix on, say, a covariance analysis we were running.

Second, the grammar labels the structural units of the alignment. For example, the grammar tells us which intervals in the alignment are left-hand sides of helices. This information could also be extracted from the alignment annotation entry shown in Appendix 2. We could even match up an lhs and a corresponding rhs as the grammar does by searching for matching sets of lhs-rhs symbols in the annotation. However, if more complex structural units are built out of the elementary structural units (gap, lhs, rhs), then a grammar can contain information that the alignment itself will not have, even with the annotation entry added. Such a *hierarchical* grammar, with its multiple layers of description, allows us to label larger structural units and manipulate them easily.

The grammar that is in fact now used in our alignment insertion tool is relatively "flat", with loop_units forming the only combinations ("phrases", so to speak) of the elementary units. I started out with a much more hierarchical version, with the 16S rRNA molecule broken down into a central core and four large arms, with subunits formed out of each arm, and so on, until the elementary unit types were reached. However, over the course of development I found that this was not necessary, and the concept of hierarchy was weakened. One might say that we are currently breaking the sentence (molecule) down directly into individual words, and skipping the groupings into noun and verb phrases. This was all that is needed in terms of hierarchy for the particular type of parsing done here. (Generation of a flat grammar was also much the simplest to automate.) However, in other forms of parsing, such as recognition of complex structures like genes (which are built out of clearly defined substructures which can have substructures of their own), highly hierarchical grammars are preferable. (For examples, I again direct your attention to the work of Dr. David Searls in this area [8,9,10].)

Finally, the grammar serves as a convenient, concise summary of what is known about the molecule in terms of the alignment. There is nothing in this summary that cannot be recreated from the alignment and the associated phylogenetic tree, but it does appear to be a more

convenient representation than the alignment for some tasks. For example, we can generate an explanation of the structure in terms of a series of natural-language-like English sentences in a set of paragraphs more easily using the grammar rather than the alignment as the starting point.

4 The Pinning Component

What is a pin? A *pin* is simply a statement that the base at position M in the sequence for the new species should be placed in alignment column N. Such a statement may be true, or it may be false. Placement of each base in the new sequence into a particular alignment column is exactly what our alignment insertion tool is trying to achieve, so the more (reliable) pins we have, the better. If the pinning component of our tool could pin all the bases by itself, then the second grammar/parser component would not be needed. However, that is not the case (though in some species closely related to the species already in the alignment, we do get a very large number of correct pins using the technique described below).

I will not go into as much detail about the pinning program as I will for grammar and parser, since the basis of the particular pinning technique used here is well laid out in an Argonne technical report by Dr. Overbeek and Dr. Ian Foster [40]. We employ a standard dynamic programming algorithm, with one important addition. As stated in their report, the core of this technique is the employment of "consistency checks to produce reliability estimates". We can compute confidence levels for pins and throw out those pins that fall below a minimum confidence level. Hence we can set the minimum level to produce extremely reliable pins, with an extremely small number of bad (incorrect) pins slipping through. Even with the minimum set quite high, pins will be generated for a majority of the bases. The parser then addresses the minority of bases that fall in the more variable regions and thus cannot be pinned.

Pins are generated as follows. A random set of species is selected from the alignment. A set of pins is then generated for the new species when compared to this subset of alignment species. (The process takes time, so we use a subset of species, not the entire alignment.) We then employ the phylogenetic tree to provide guidance. The phylogenetic tree is a binary tree, so the root node, like all other internal nodes, has two branches. The species that are most related to the new species are going to hang off only one of these branches, not both. What we would like to do is move down the tree via one of the two branches toward the subset of species related to the new species. If we can do that, then we can do another pin generation run using a subset from the new, smaller subtree. That is, when we move down to another node we generate a new set of pins using a representative set of species from the subtree of which the new node is the root. This should give better results (more pins) since we are now using a collection of species closer to the new species. We keep reiterating this cycle until we can move down no further in the tree. The pins generated from the last run before we stall then become the set of pins outputted. A diagram of the process is given in Figure 8.

The big question is: How do we choose which branch to move down? The answer is that we *vote*. For each node in the phylogenetic tree, a vector has been computed. (This computation has to be done only once for a given alignment; then the vectors are loaded in automatically each time the program is used.) Each vector has 1,892 components (one for each alignment

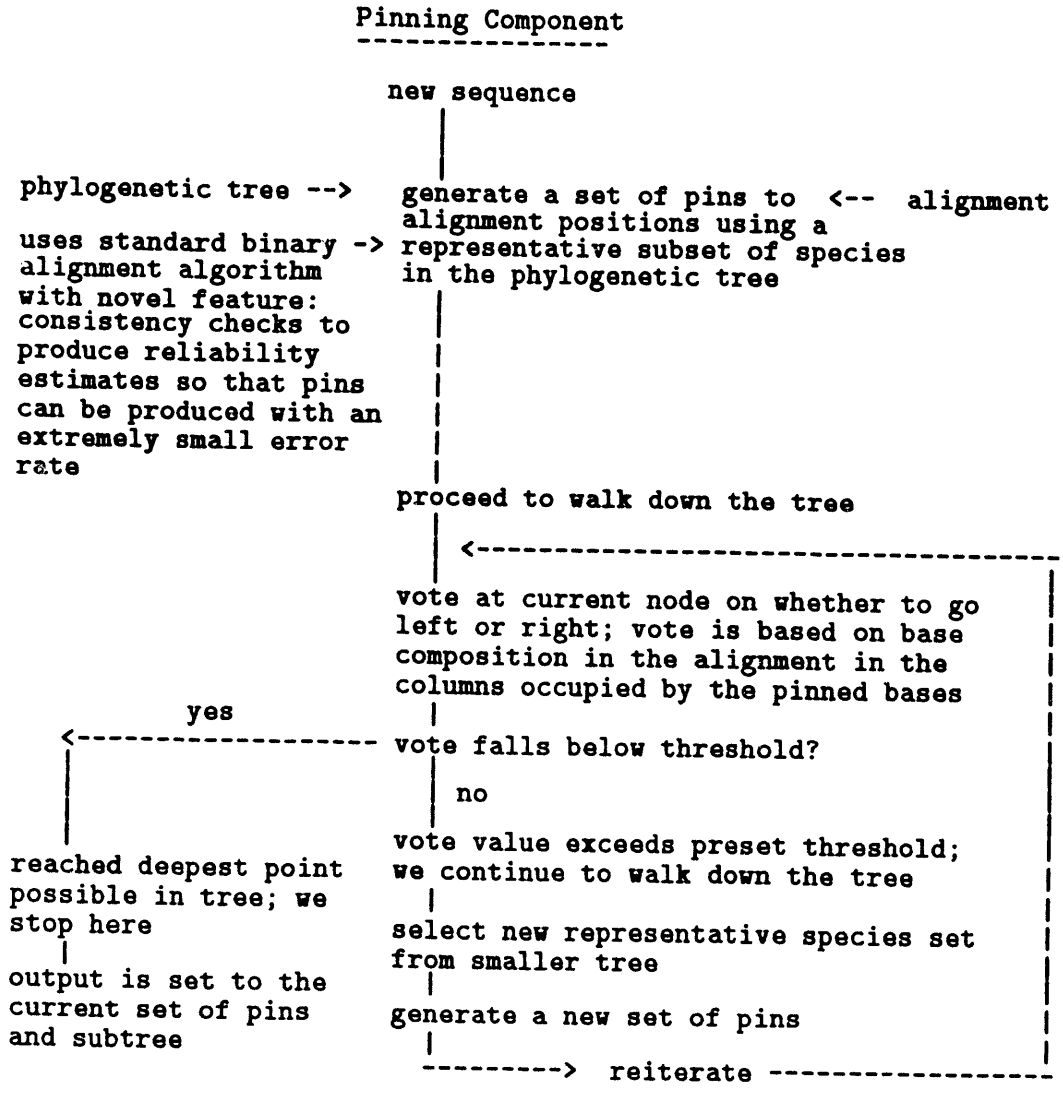


Figure 8: Diagram for the pinning component

column). Each component, in turn, stores eight pieces of information, four for each branch. For the right branch the component stores how many species in the subtree formed by right branch of the node contain an *a* in that alignment column, how many contain a *c*, how many contain a *g*, and how many contain a *u*. The same data are stored for the left branch. These vectors are then used to calculate a vote for each pinned base in the new species. For example, suppose that after the set of pins is generated at the current node we find that base *a* at species position 455 is pinned to alignment column 503. Then the component of the vector for the current node for alignment column 503 would be examined. If there are more *a* bases in the the species in the right branch of the node than in the left branch in column 503, then a vote is made to go right. The magnitude of this vote is the number of *a* bases in the right branch divided by the size of something called the *changelist*.

The *changelist* concept is simple to understand. One changelist is associated with each column in the alignment. If there are 1,892 alignment columns, then we have 1,892 changelists. A changelist for a particular column is a list of the internal nodes in the phylogenetic tree whose predominant base type in that alignment column changes between the left and right subtrees of the node. The larger the changelist, the more variation we have in that particular alignment column. The changelist values are calculated by a program written by Dr. Steve Winker at Argonne. For each alignment column, Dr. Winker's program examines each internal node in the tree, one by one. At each node it counts the occurences of each base type (A, C, G, or U) to find the predominant base. It then checks to see if this predominant base type changes between the two branches of the node. If so, then we have a *change*, and we add the node to the changelist. If not, that is, if the same base predominates in both subtrees, then the node is not added to the changelist. Note that the a node may be included in the changelist for alignment column M, but in another alignment column N (where its two branches share the same predominant base in that column) the same node may be left out of the changelist. The magnitude of the changelist associated with a column reflects the stability of that column throughout the alignment. The smaller the size of the changelist, the higher the stability [50]. The changelist concept is used here to implement the fact that we desire that a vote from a stable column should weigh more in the voting than one from a column in a variable region. (As with the node vectors, the set of changelists only has to be calculated once for a given alignment.)

We perform the voting process for all pins, obtaining a vote vector (magnitude and direction) for the alignment column associated with each pinned base. For example, if 800 of the 1,500 bases in a species were pinned, then we would have a total of 800 votes. We then add all the votes for the right branch together into RightSum and all the votes for the left branch together into LeftSum. Next, we calculate two ratios: $RightRatio = RightSum/LeftSum$ and $LeftRatio = LeftSum/RightSum$. If RightRatio is greater than LeftRatio, and if it is also greater than a preset threshold, then we move down to the next node on the right branch of the current node. We then restart the whole process by calculating a new set of pins, as described above. If LeftRatio is greater than RightRatio, and if it is also greater than the preset threshold, then we move down to the next node on the left branch of the current node. If neither ratio exceeds the threshold, then we cannot move down the tree any farther, and we end the pin run at the current node. (The threshold was determined empirically. Obviously, if we do not use a threshold of some kind, we are eventually going to run into noise that can cause us to take

the wrong branch. We tried out various thresholds on the species already in the alignment and phylogenetic tree to see what was the minimum we could use without ending up at the wrong place in the tree.)

In addition to the pins themselves, the pinning program also returns the subtree whose root is the last node where pins were calculated. The set in this subtree is presumably those species most closely related to the new species. The parser later makes use of this fact by confining its parse to the subset of grammar clauses belonging to only those families which have at least one member species in the pinning subtree. This speeds up the parse and improves its accuracy. If the end user is interested in evolutionary relationships, the subtree can be included with the aligned sequence in the tool's output. This arrangement will help the user track down the location where the new species should be placed in the phylogenetic tree associated with the alignment.

5 The Parser Component

The parser component consists of two sections. The first section performs the actual parsing of the input base string, which slots the bases into the structural units. The problem then remains of inserting the indels at the proper positions among the bases placed within each unit. This is the job of the second section. Once that is done, the aligned version of the sequence is put together as simply the concatenated string of all the aligned structural units. A diagram for the parser component is shown in Figure 9. A much more detailed set of diagrams for the first section of the parser mechanism that does the parse and assigns bases to structural units can be found in Appendix 9.

5.1 Parsing of the Bases into the Structural Units

The criterion for final parser success is simply this: no bases remain after the last structural unit has been parsed. That is, all bases from the initial input string (typically about 1,500 bases long for the 16S) have been slotted into some structural unit (consumed by some structural unit) without causing a violation of that unit's grammar clauses or the grammar clauses for any of the following structural units.

Localized parse failures in unpinned groups of structural units that lie between two pinned units (or between a pinned unit and one of the ends of the molecule) can occur. The overall parse will still succeed, however, because the parser can recover its position in the input string at the right-hand pinned unit. Nevertheless, that is not an ideal situation, and we try to avoid such localized failures (though cases will indeed occur despite our best efforts, since some new species simply will not obey, in one place or another, even the weakest of the grammar rules extracted from the alignment). A localized parse failure diminishes the accuracy of the parse, since the bases falling between the pinned units cannot be confined to a given structural unit but rather are spread throughout all the unpinned units between the two pinned units.

The way in which the parser is currently written and the way it currently employs pinned units virtually guarantee overall success (no bases left over at the end; program ends normally

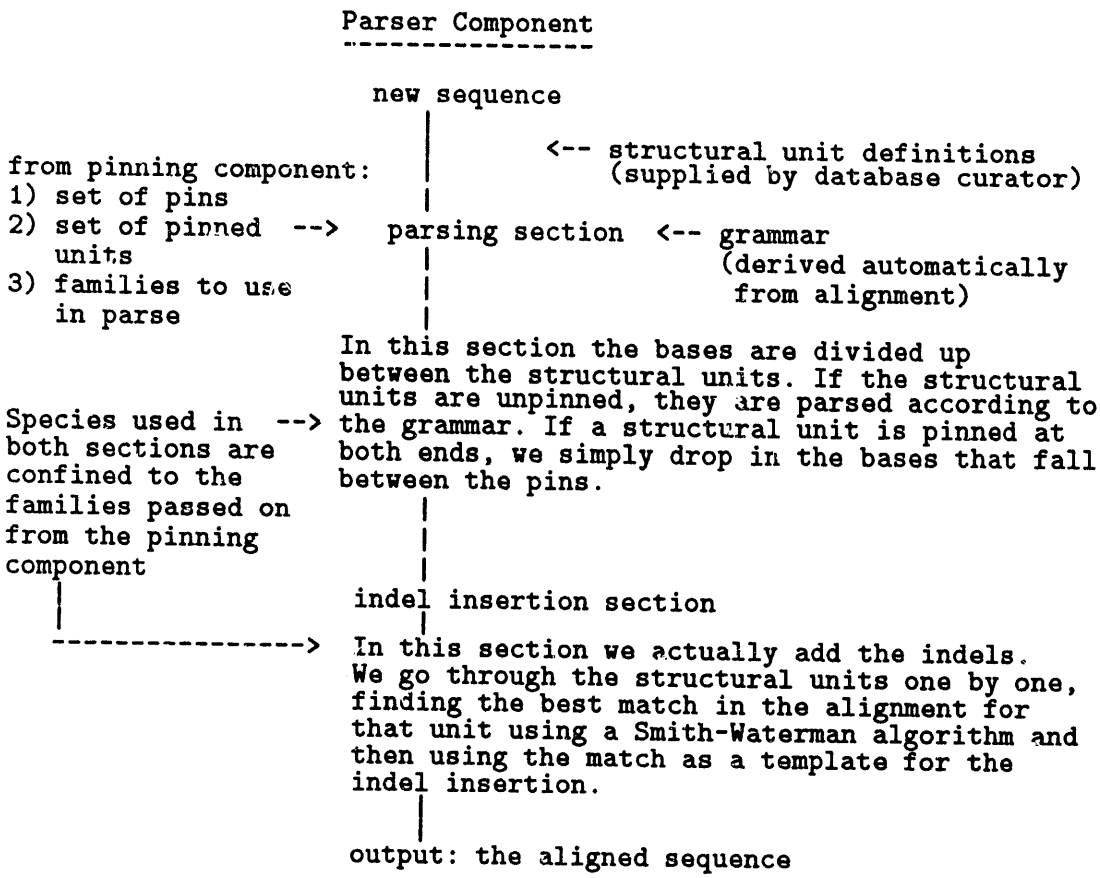


Figure 9: Diagram for the parser component

and reports results). However, if the number of localized parse failures is very large the overall success would mean very little. Fortunately, this does not look like a problem. Localized failures have been few and, in general, confined to small regions. (See the results section below.)

A localized failure, per se, is not currently flagged for human attention. If the region of failure is small, there is still a fair chance (with the aid of interior pins) for the indel insertion section of the parser to later insert the indels properly. However, in a future version of the parser we might add a line of output to flag such a region as one where errors are more likely to occur, and thus as an area that should be checked more closely.

The parser is written in Prolog, which means that we have a built-in backtracking inference engine. If the parse fails in some structural unit, the parser automatically backtracks to the last decision point to see whether it can come up with a different input string to the structural unit that failed to parse. This feature is what makes Prolog so appropriate for computational linguistics and certain artificial intelligence applications. Indeed, Prolog is probably the leading language today in the field of computational linguistics. Its use in that area is so common that a certain type of syntactic "sugar" has been added to easily express Definite Clause Grammar (DCG) notation, which I used in the constraint/3 clauses above. The DCG right arrow symbol is used in those clauses. This symbol causes Prolog to rewrite the clauses when it reads them in from a file, adding two arguments for an input string and an output string. (Thus the constraint/3 clauses are really constraint/5 clauses, and I will refer to them as such from now on.) An excellent short introduction to the use of Prolog in parsing and to the use of DCG notation can be found in [51]. Longer texts on the general use of Prolog in computational linguistics are in [52] and [53].

At its most basic level, the parser works as follows. The parser feeds an input string consisting of all the remaining bases into a structural unit. The structural unit (if the parse is successful within it) then "consumes" a substring of bases from the start of the string and passes back to the parser the shortened remaining list of bases. (I am simplifying here. The structural unit itself does not do anything. Rather, a piece of code processes the string using all the constraints associated with that structural unit.) The parser then repeats the process with the next structural unit. The parser moves along the structural units, handing the base string off and getting a shorter string back. After the last structural unit is exited the empty (null) string will be what is left, if we have a successful parse.

Why would the parser fail in a structural unit? Here's one example. Suppose the parser feeds a string whose head (start) consists of the bases *acaaggg ...* into unit *b351_353*. To save space, I will simply use *acaaggg* as the input list to the unit in my discussion below, but remember that this can be simply the head of a much longer string. Now suppose that this unit is a gap, so no secondary structure information is involved and the only constraints are the base invariancy constraints on the primary sequence. Let the base invariancy constraints for this unit be *an,an,an*. These constraints can be described as three sequential tests, with the output of one fed as input into the next. The input to the first *an* test is the string *acaaggg*. Remember from our discussion of constraint clauses in Section 3.1 that the *an* test means that only an *a* or an *n* will satisfy the test. There is an *a* at the start of the input string, so the first *an* test is satisfied. It consumes (recognizes) the *a* and passes on the shortened list *caaggg* as input to the second *an* test. However, when the second *an* test looks at the start of its input

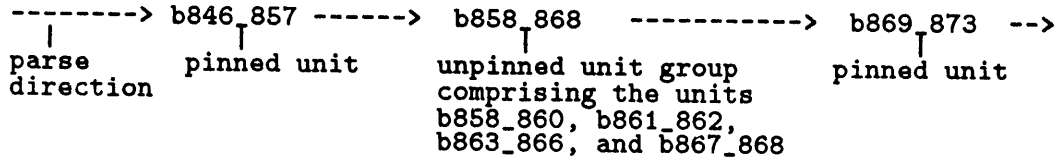


Figure 10: Diagram for detailed parse example

string, it sees a *c*. This will not pass, so the second *an* test fails, thus causing the parse of the entire structural unit to fail. Note that the unit (and each of the tests contained within it) is limited to taking bases from the head of the input string and in the order that they are given.

Now, suppose we took the same input but a different set of base invariancy constraints: *1...2,an,an*. The *1...2* test can consume one to two bases of any composition. The first time through it consumes one base from the input string of *acaaggg* and passes on the string *caaggg* to the *an* test. As before, the *an* test will fail because of the *c* at the start of its input. However, in this case there is a decision point within the unit. The *1...2* test has two possible choices, and it has used only one. Hence our Prolog engine automatically backtracks to the *1...2* test upon the failure of the first *an* test. The *1..2* test then tries a different choice by consuming two bases and passing on the string *aaggg*. The *aaggg* satisfies the first *an* test, which consumes an *a* and passes on *aggg* to the second *an* test. The second *an* test in turn is satisfied with the *a* at the start of its input string and outputs the string *ggg*. Since this is the last test for the unit, the parse now succeeds, and the unit as a whole passes along the string *ggg* back to the parser. The parser notes the difference between the input string fed into the unit and what it received back. This difference tells the parser what bases to store as slotted into that unit. The parser then passes on the output string of *ggg* to the following structural unit, where the same process will be repeated. (One way of looking at the parser controller is as a messenger that carries the ever-shortening input string from unit to unit.)

5.1.1 Detailed Parse Example

Let us go through a trace of a parse over several structural units combined, to give a better idea of the parser in action. Suppose that the parser has reached unit *b846_857* in its parse of a new sequence. Also suppose that the pinning component has provided the information that unit *b846_857* is a pinned unit; that units *b858_860*, *b861_862*, *b863_866*, and *b867_868* are unpinned units; and that unit *b869_873* is a pinned unit. The units *b858_860*, *b861_862*, *b863_866*, and *b867_868* form what I call an *unpinned structural unit group*, sandwiched between the two pinned units. Hence we obtain Figure 10.

The pinned units provide reference points for the parser, telling it what bases must go into the unpinned unit group and what bases must come out. The parser confines backtracking to within a single unpinned unit group. (If the parse fails for some reason, there is no point in backtracking past a pinned unit in order to try to find an alternative, since the pinned unit will always pass on the same string to the following unit.) The entire parse is divided up into

unpinned unit groups and the pinned units that lie between them. Note that a localized parse failure always extends through exactly one unpinned structural unit group.

The following happens when the parser goes through the section of the 16S molecule given in the above example:

1) First, the parser looks at the *b846_857* unit. Checking, the parser finds that it is a pinned unit. The bases that belong to that unit have already been slotted in and hence are known. All the parser has to do is put those bases in its own storage list as slotted into this unit, and then pass on the shortened string (with those bases removed from its head).

2) Second, the parser looks at the *b858_860* unit. Seeing that it is an unpinned unit, the parser immediately looks ahead to see how far this list of unpinned units goes. It finds the next pinned unit (or the end of the molecule, whichever comes first). From the following pinned unit, the parser then deduces what bases should fall within the unpinned unit group. These bases will form the input string to the group. (The group will have parsed successfully if none of these bases are left over after the last unit in the group is processed, that is, if the null string is left over.) The parser then constructs a Prolog goal "on the fly" for the four unpinned units combined, and then calls that goal. This goal consists of four subgoals, one for each unit. The goal succeeds if each subgoal succeeds, that is, if all tests (base invariancy constraints and, if appropriate, secondary structure constraints) succeed for the each of the four units and all the bases end up slotted into one of the four units. If the goal fails, then we construct the another goal for the unpinned group using the constraints for a different family and try again. We continue, if necessary, until we have built and tried a goal using the grammar clauses of every possible family. We stop only when success is achieved or we run out of families to try. (Currently there are 26 different clauses in each Prolog predicate that apply to a given unit, each expressing the constraints according to the alignment species in a different family. Hence, up to 26 goals could be tried. Ordinarily, somewhat fewer goals would be used, since not all 26 families would show up in the subtree returned by the pinning component, which is used as a guide here.)

The question might arise as to why we are trying to parse according to one of the 26 subfamilies (which together span the alignment) rather than just according to the constraints that apply to all the species in the alignment (that is, to the constraints that hold for the family *Prokaryotes*, to which all the species in the current 16S rRNA alignment belong). The answer is that a constraint that applies to all the species is liable to be very weak, and hence not of much use in correctly slotting the bases into the proper units. Tighter constraints that come from a subfamily of more closely related species are much better. For example, in the family *Prokaryotes* it might be the case that all we could say for a given unit in terms of its base invariancy constraints is that it contains one to five bases (constraint: *1...5*). On the other hand, in a subfamily such as *Flavobacteria* the grammar might be able to state that there should be an *a*, followed by two to three bases of any composition, followed by another *a* (constraint: *an,2...3,an*), a much tighter pattern for the input string to match.

We are using here a technique called *meta-programming* to impose an extra level of control by constructing and then launching Prolog goals. Using parts (various clauses) of the program itself as data, we build an entirely new piece of code (which did not exist until then) and execute it using Prolog's *call(Goal)* predicate. If any one line can be called the "heart" of the parser

component, the line of code that performs *call(Goal)* is it. Two interesting points: (1) the set of goals so constructed is going to differ from species to species (since which units are pinned and which are not will differ in each species) and (2) each such goal will be "invisible" in the sense that it will never actually appear in a listing of the program's source code. This is a strong reminder of how different Prolog is from a more conventional computer language such as C or Pascal. Also note that since there is a complete equivalence of programs and data in Prolog, meta-programming (using a program itself as data) is, as Sterling and Shapiro say, "nothing special in Prolog" [6].

The reason for using this extra control level is our desire to sequentially try the constraints of each family, as explained above. The extra level of control allows another goal to be constructed and tried if the goals tried earlier have failed. Hence we can try alternative constraints from the grammar over the same series of structural units.

The parser makes use of pin information in two ways. First, as described earlier in the processing of the *b846_857* pinned unit, a pair of pins that occur on a unit's boundaries can slot in bases for that unit automatically for the parser. Second, there can be pins that fall within the interior of a unit. These pins, while not as useful as the boundary pins, can still aid the parser. Such a pin states that the base in such and such a position in the new sequence must lie within such and such a unit. This can be used to supplement the grammar constraints. However, there is the possibility that such a pin can be incorrect. This might force a false failure of the parse. In such a case the parse might have succeeded in the unpinned unit group if it had not made use of the interior pins that were available. The first time we construct a goal for the unpinned unit group using the constraints for one of the 26 families, we build the goal so that the information from all the interior pins falling within the four units is used. If the first goal succeeds, well and good. If it fails, we sequentially build and call goals for each of the families that show up in the pinning component's subtree. All of these goals make use of interior pins. However, if the entire set of goals fail, then we do something in addition. We reconstruct and launch the same set of goals, in the same order as before, but with the single difference that interior pins are not used. If the cause of the failure the first time through a goal was a bad pin, then the second time through that same goal will now succeed, and thus the parse will succeed for this unpinned structural unit group.

Now let us look at what will happen when a goal for the unpinned unit group is called. The first of the four unpinned units is *b858_860*, which is a gap. (The three following units, *b861_862*, *b863_866*, and *b867_868*, form the lhs, cap or gap, and rhs of a stem-loop, respectively.) The first subgoal tried is therefore for this gap. The only grammar constraint done for a gap is a base invariancy test (using a constraint/5 clause). (If this is the first time this family has been tried, there will also be a check using the interior pins for unit *b858_860*). If this test succeeds, then the *b858_860* subgoal will succeed, and we move on to the lhs in unit *b861_863*.

While the case here is that the associated rhs for the lhs in *b861_863* falls within the same unpinned unit group (in unit *b867_868*), there is no guarantee of that always happening. Often the associated rhs of a lhs will lie outside the unpinned unit group containing the lhs. The rhs will exist either as a separate pinned unit or as a unpinned unit in a separate unpinned unit group.

I made the decision to limit backtracking to within a single unpinned unit group. The motivation was to dramatically decrease the chance of running into an explosion of possibilities upon backtracking. (See the discussion of the problems associated with automatic backtracking below.) However, this decision meant that if the rhs lies in a separate unpinned unit group further on in the molecule, then when the parse later reached that rhs and the secondary structure constraint test was tried, we could have a problem. Suppose the secondary structure test fails. If the failure is due to incorrect bases in the rhs, that is all right; we can backtrack within the unpinned unit group containing the rhs to come up with different set of bases. However, if the true cause of the failure is that we placed the wrong bases in the lhs, then we have no way of backtracking to the lhs to fix that, and hence the parse of the unpinned unit group containing the rhs will fail. (By the way, the bases for each lhs are carried from unit to unit in a storage structure maintained by the parser, and are made available as needed to an rhs unit when the secondary structure (base bonding) test needs to be done between an rhs and its associated lhs.) Hence we must do as much as we can to ensure that we get the lhs right while we are still in its unpinned structural unit group. We do this by looking ahead, if necessary, to the unpinned unit group containing the associated rhs.

Hence whenever the parser runs across an unpinned lhs unit, it does the following:

a) It first checks to see that the lhs passes its own base invariancy constraint test. If not, we backtrack. If yes, then we proceed to (b).

b) It checks whether the associated rhs falls within the same unpinned unit group. If so, there is no problem. The lhs unit simply has to be tested using base invariancy constraints (already done in (a)), and the secondary structure constraint test involving base-bonding between the lhs and rhs will be performed when we arrive at the rhs unit.

c) If the associated rhs lies outside the current unpinned unit group as a separate pinned unit, then we test whether the bases in the lhs will satisfy the secondary structure (helix or base-bonding) constraints with the bases in the rhs being those that were slotted into the pinned unit. If the test succeeds, fine. If not, then we fail here on the lhs and start backtracking.

d) If the associated rhs is not pinned and lies in a separate unpinned unit group, then we find the first pinned unit falling before the rhs and the first pinned unit falling after it. Using these pinned units as references, we find the units and bases that fall within the unpinned structural unit group to which the rhs belongs. We construct and execute goals on that unpinned unit group in the same manner as discussed above for the original four-unit unpinned group, with the following change:

We treat all the unpinned units as gaps except for the one rhs. That is, we apply the constraint clause and interior pins tests to all the unpinned units in sequence, using the bases contained within that unpinned unit group as the starting input string. If the parse of the group fails (because of an incorrect pin or whatever), then we stop and let the lhs in the preceding unpinned structural unit group succeed. If the parse of the group containing the rhs succeeds, then we reparse the group a second time, using one additional test: we apply the secondary structure constraint test to the rhs using the bases found for the corresponding lhs. Now that we have established that the unpinned group containing the rhs can be parsed (if secondary structure constraints are ignored), if the secondary structure test matching the rhs with the

lhs fails, then that strongly indicates that the wrong bases are in the lhs and that we should fail and backtrack at the lhs, allowing the correct set of bases to be placed in the lhs. If this additional test succeeds, then we say that we succeeded in the lhs and move on. If the test fails, then we fail at the lhs. This is not the best possible testing method, since we are not using some information — i.e., we are not applying the secondary structure test to the other rhs units (if they exist, of course) — that fall within the same unpinned group that contains the original rhs of interest. However, the current combination of tests has produced excellent results so far.

If the subgoal for lhs unit *b861_862* succeeds, then the subgoal for *b863_866* is tried. Since *b863_866* is a gap, the only grammar constraint done will be a base invariancy test (using a constraint/5 clause). If this is the first time this family has been tried, there will also be a check that all interior pins that should fall within *b863_866* actually do so. If the constraint/5 test succeeds (and, if interior pins are used, if that test succeeds), then the *b863_866* subgoal will succeed, and we move on to the rhs in *b867_868*, which forms the last unit in the unpinned unit group. Both base invariancy and secondary structure (base-bonding) tests will be used for this unit. If both these grammar tests succeed (and, if interior pins are used, if that test succeeds), then the *b867_868* subgoal succeeds. At this point all four subgoals of the original goal have been tried and exited with success. The parser now checks to ensure that the empty string is what is left of the original input string. If so, then we exit the goal with success and proceed to step (3) below. If not, the parser starts backtracking back into the subgoals to try to find an alternative configuration of bases that will leave zero bases left over. Note that the goal as a whole has been constructed so that the output string of each subgoal serves as the input string to the next.

As mentioned earlier, the rhs in our example lies in the same unpinned group as its associated lhs. However, if a subgoal for an rhs is being executed and if the associated lhs lies in a separate unpinned group that previously failed to parse (hence no bases are available as slotted into just the lhs), then the rhs subgoal is allowed to succeed if its base invariancy and interior pins tests are satisfied. The base-bonding test is not used in such an instance.

3) Third, if one of the goals tried for the unpinned unit group succeeds, then using the information returned by that goal, the parser divides up the bases between the four units in the group and stores that away for later output to the indel insertion section. The parser then passes on a shortened string, with all the bases that fall into the group removed, to the pinned unit in *b869_873*, where the parse continues by repeating the same process as in (1) above.

If all the alternative goals have been tried and each has failed, then the parser accepts the fact that it will not be able to break down the bases in the input string so that the bases that belong to each of the four units are properly slotted into an individual unit. Instead, it simply stores all the bases as being slotted into a larger region made up of the four units combined. It then continues the parse by passing on a reduced input string (minus these bases) to the following pinned unit in *b869_873*. (This localized parse failure makes the job of the indel insertion section harder, of course.) This concludes our trace of the parse through the *b846_873* example.

5.1.2 Potential Problems in Prolog's Automatic Backtracking Mechanism

Automatic backtracking upon failure is one of the most valuable features of the Prolog computer language and also, on occasion, one of the most frustrating. The question is how to use this facility wisely: how to extract the most benefit while limiting the potential problems. Let me give an example of the type of problem we can run into. Suppose that we treat a region of some molecule as a single unit, with the only constraint on that unit being that we know that 1 to 100 bases can fit into it. There is no constraint on base composition, only on the number of bases. Our sole constraint could then be written as *1...100* in the notation used in the grammar. Now suppose that the parse failed at some point further on in the molecule, so the parser wishes to backtrack over this unit to generate other possibilities. Question: How many possibilities are there for the parser to generate? The parser is confined to taking one base off of the start (head) of the input string and slotting it into this unit, or taking two bases off the head, or taking three bases off the head, and so on. Hence there are exactly 100 possible ways of succeeding (or 99 ways to backtrack and succeed again after the initial success). This is a fair number for a human to plow through, but trivial for a computer. Now suppose that there are substructures in this unit, so that the biologists who design the structural unit definitions wish to reflect these structures in the grammar. Suppose that the unit is thus divided into ten smaller units, but that we still know nothing about base composition (and secondary structure constraints are also not used). For simplicity, also say that the ten units divide up the region somewhat equally so that zero to ten bases can be slotted into each of the ten units. The constraints for the ten units when hooked together would then look like this:

```
0...10,0...10,0...10,0...10,0...10,0...10,0...10,0...10,0...10,0...10
```

With no check on base composition (such as, say, that the third unit must contain an *a* followed by a *c*), the parser is free to spin through all of the possibilities. And there are a lot of them. Instead of the hundred possibilities we had with the single unit, there are now ten to the tenth possibilities for the parser to work through. What we face is nothing less than a combinatorial explosion. For example, when there was one unit in this region, there was only one way to slot 20 bases into the region. Now we can take five bases from the head of the input string for the first of the ten units and insert them into that unit, another nine bases from the head of the input list for the second unit, and the final six from the head of the input list for, say, the seventh unit. Or we could slot seven bases into the fourth unit, six bases into the ninth unit, and seven more into the tenth unit. And so on. The problem here is not getting an incorrect answer out (though weak constraints will indeed produce a more inaccurate alignment of a new sequence). The problem is getting any answer back at all in a reasonable period of time. Tightening the constraints by parsing according to subfamilies could be a solution. For example, perhaps in some subfamily of all species in the alignment for this hypothetical molecule the constraints in the fourth unit would change from *1...10* to something like *an,an,gn,2...3,cn,cn*. This would dramatically cut down on the possibilities.

However, if the constraints have already been tightened as much as possible, is there anything else that can be done? The answer is yes. Note that no matter how we divy up the 20 bases inside this one region, that is, no matter which of the possibilities for slotting in the 20 bases we use, the same string (minus 20 bases) will be passed on to the rest of the parse that

takes place after the tenth unit. Hence we could pass on the same input string hundreds or thousands of times to the rest of the parse, and every time after the first will be wasted, since if the parse fails with a given input string the first time through, it is always going to fail the second and the following times that the same string is used as input. (Nothing has changed in terms of the primary sequence; and since we are doing a lookahead to the associated rhs when an lhs is reached, secondary structure constraint test results should also not change.) What we wish to do is keep track of which strings have been passed on as input to the rest of parse, at any given point in the parse. Then we can check after each unit to see whether the same string has been passed on before. If so, there is no point in going any farther (the parse must have failed at some point later on with that string), so we can force a failure right there instead of failing farther (perhaps much farther) on, and thus save a great deal of time. The storage of intermediate results for this purpose is a form of *chart* parsing [54], and I am indebted to Dr. Searls, who directed my attention to its possible use. I have implemented just such chart parsing in the parser component of our insertion tool. (In Prolog terminology, this is called *memoization* or *memo-function* use (see p. 88 in reference [5], pp. 181-182 in [6]).

I have given an extreme example above. Fortunately, such allowed number ranges are unusual. There is nothing even remotely like it in the 16S grammar. Also, we have plenty of base composition and secondary structure constraints we can use. Still, chart parsing saves time in the runs, and, more important, it serves as a effective guard against extreme time-wasting possibilities on those rare species that fail near the end of an unpinned structural unit group and thus do a vast amount of backtracking.

Hence I believe that I have the backtracking problem under control for the 16S molecule. I also have confidence that the problem is controllable in similar molecules. However, if you do happen to run across a couple of regions in some molecule like my hypothetical example above, and these regions were neighbors, and if you are doing your parse on a small desktop computer (and you were not using chart parsing), you should be prepared to get an answer back sometime after the turn of the century. The biggest surprise I had during this project was just how fast the possibility explosion could appear when backtracking over weak constraints.

5.2 Insertion of Indels in the Proper Positions

The second section of the parser works as follows: Suppose for some new species the parser groups the bases *aggcu* in structural unit *b9_13*. For simplicity, also suppose that there are no interior pins within the *b9_13* unit. (If there were, the program would make use of this information by breaking the unit down into subintervals. However, it is not necessary to go into that level of detail here.) Then the code to add indels in the proper positions in *b9_13* does the following:

1) The program first checks whether the number of bases slotted into *b9_13* exceeds the number of alignment columns for that unit. If such is the case, obviously there is no room to add any indels, so we stop here and simply add a pair of brackets ([ ]) around the bases in the visual output to notify the user that something unusual (more bases than alignment positions) occurred in this unit. If the parser correctly slotted the bases, then the alignment will need one or more columns inserted to handle the new species properly. If the parser made a mistake,

the brackets will flag the error for easier correction. This is a fairly rare occurrence, but it can happen. (If the parser failed in an unpinned structural unit group and the number of bases slotted into the alignment columns for the entire group exceeded the number of columns, then that slightly different type of anomaly would be signaled by a double pair of brackets ([[ ]]).) Let us assume that the number of bases in our example do not exceed the limit, and so proceed to step 2.

2) The code now searches through all alignment species belonging to the the pinning subtree (which may be as large as the entire alignment or as small as four or five species) for those species that have the same number of bases (in this case, five) in unit *b9_13*. If there are no such species having the same number of bases the program skips directly to 4b below. Otherwise we proceed to step 3.

3) At this point the program searches within subset of species found in step 2 for the species whose base group in the interval most closely resembles that of the new species we are inserting. That is, the bases in each corresponding position are compared and the alignment species that has (or have, if more than one such species) the most matches with the new species will be the result returned from this step.

4a) If the result found in step 3 differs by at most one mismatch from the sequence of the new species, then we duplicate for the new species the indels used in the alignment species found in step 2. For example, if there is eight alignment positions in this unit (room for three indels in addition to the five bases), and the alignment species most closely resembling the new species found in step 2 has indels at positions 2, 5, and 6, then we place indels (dashes) at positions 2, 5, and 6 in the new species.

4b) If the result found in step 3 is not accurate enough (i.e., differs by more than one mismatch) then we scan the alignment again. Using the Smith-Waterman algorithm, we score all species in the pinning subtree for this interval (with all indels removed) for a similarity comparison with the new incoming sequence. We then choose the alignment sequence with the highest score to line up the new incoming sequence. As in step 3a, we insert indels in the new sequence to make it match up with the chosen alignment sequence. The Smith-Waterman algorithm automatically produces a correspondence between the highest-scoring alignment species' sequence and the new species' sequence, so indel insertion will follow the correspondence produced. Indels are inserted in the new species in this fashion:

(1) If the correspondence produced shows one or more insertions in the new species relative to the closest matching species in the alignment, then we have a situation where inserting indels in the new species according to the correspondence produced by the Smith-Waterman algorithm will result in the new species having a number of bases + indels in the structural unit that is greater than the number of alignment positions in that structural unit. Hence all species already in the alignment would have to be modified by also inserting one or more columns of indels in order to align with the new species. That is, the size (length) of the alignment would have to be increased in this structural unit. Note that such a situation occurs when we have one or more noncorresponding bases (bases not included in the correspondence pairs) in the new species. We can handle this situation in various ways. Here are two:

If we do not wish to throw all the following structural units out of sync with the alignment (when all the aligned units are concatenated together at the end), then in such a situation we can simply center the new species in the middle of the structural unit by padding it with indels on both sides. We also output a message in the commentary saying that simple centering was used for this unit.

Alternatively, we can insert indels (dashes) in the new species according to the Smith-Waterman correspondence, regardless of the effect on the total alignment length. We also signal that such an occurrence has taken place by surrounding the entire base and indel group with a pair of parentheses. Currently this method is used. Code for the first method has been written but is not employed at present.

(2) If the correspondence produced does not show any such insertions, but rather shows only deletions or substitutions (i.e., the only noncorresponding bases — which represent deletions relative to the best match in the alignment — are found in the matching alignment species, not the new species), then we proceed as follows:

First, we insert an indel in the new species wherever a noncorresponding base in the matching alignment sequence occurs (a deletion in the new species relative to the matching species). After this is done we have the same number of characters (bases + indels) in the new species as there are bases in the matching species.

Next, we insert an indel in the new species wherever an indel occurs in the matching sequence, with the matching sequence being used now consisting of its bases *and* its indels, as it originally looked in the alignment. Note that this is a trivial task, since we simply count up the number the bases in the match which come before an indel and then insert an corresponding indel at the location falling after that number of characters in the new species.

Obviously we have a better chance of placing the indels correctly in the new species if we can follow this insertion method for step 4b rather than the first discussed.

We try steps 2, 3, and 4a before trying step 4b simply to avoid performing the Smith-Waterman algorithm if possible. (The algorithm is time-consuming, and there is no point in using it if we can avoid doing so without loss of accuracy.)

6 Results

The new alignment insertion tool was tested in two ways: using species already in the 16S rRNA alignment, and using species not present in the alignment (but for which separate aligned versions did exist, so the results could be scored). Results are given for these tests in Sections 6.1 and 6.2, respectively. In both cases our insertion tool program was run under Quintus Prolog version 2.5.1 on a Sun workstation working under the Unix operating system.

The output of the alignment insertion runs was graded on the number of bases correctly placed, where the definition of a base being "correctly placed" is that the correct base type (A,C,G,U,N) shows up in the correct alignment column. A detailed discussion of the scoring scheme is given in Appendix 7.

Table 2: Results for alignment insertion runs on species already in the 16S alignment

Number of Species Lying in Range	Percentage of Total Species	Range (percentage of bases correctly placed by the tool in the aligned sequence)
12	9.52%	100.00%
35	27.78%	99.5 - 99.99%
36	28.57%	99.0 - 99.49%
18	14.29%	98.5 - 98.99%
12	9.52%	98.0 - 98.49%
7	5.56%	97.0 - 97.99%
5	3.97%	96.0 - 96.99%
1	0.79%	95.0 - 95.99%
0	0.00%	< 95.0%

6.1 Insertion of Species Already in the Alignment

Ideally, if our tool is given the base sequence of a species already in the alignment (and hence whose influence is reflected in the grammar we use), the aligned version that the tool produces should agree 100% with the aligned version of the species present in the alignment.

The actual results were very good and fell only a bit below that standard. Using all species in the alignment that contained a maximum of 20 consecutive unknown bases (a total of 126 species), we obtained the results shown in Table 2.

Almost ten percent of the species had perfect insertions, while 65.9% of the species had insertions that placed correctly in the aligned output 99 bases (or more) out of every 100. Of the species, 95.2% achieved an insertion accuracy of 97% or higher. The lowest scoring species came in with 95.9% of its bases placed correctly. The 125 other species scored 96.2% or higher in correct base placement.

I should note here that the scores of 28 of these species have been artificially lowered. When examining the run results, I noticed that all of these species were placing bases incorrectly in the *b1_8* unit at the start of the molecule. All 28 had the correct bases slotted within the unit, but one or more of the bases were incorrectly positioned after the indels were added. This situation was unusual and aroused my suspicion. Checking with Dr. Olsen at Urbana, I discovered that the workers there do not start to "lay" in the bases until the next unit (*b9_13*). Unit *b1_8* is considered unimportant and typically contains a fair number of unknown bases or sequencing errors, so they do not try to align the bases that fall within it. Hence our program was trying to align the bases in *b1_8* against an alignment that did not exist for that unit. This subtracted a base or two from the total number of correctly aligned bases in some of the 28 species, and in others it subtracted substantially more from the count.

A complete listing of the output of the insertion run for a typical species (*Arb. globif*) is given in Appendix 4. That is what the user of the our tool currently receives as output. An examination of Appendix 4 shows that the output is broken into three sections.

The first section displays those structural units in which one or more bases were misplaced. This first section is displayed only for sequences that are either already present in the alignment or, as in the case of the Urbana private sequences I use in Section 6.2, where special coding arrangements have been made to access aligned versions lying outside the alignment. (Obviously we cannot state which bases were misplaced by our tool if we have do not have an already aligned version to check the output against. Hence this section is skipped when a completely new species is being inserted.)

The second section is a listing of complete aligned sequence for the new species, broken down into 60 characters per line. Under each line for the aligned output from our tool three other lines are also displayed, for a total of four associated lines of data. The second line of this group (immediately under the tool output) is what is called the *insertion marker* line. This line is empty (blank) except where our tool has added an extra column in its output relative to the alignment. We mark such a column with a dash on this line. (More precisely, what we do is mark a column falling immediately after the structural unit or subinterval of the unit in which the column insertion was made. This procedure is done because, as stated in Appendix 7, the information currently returned by the tool localizes the extra column(s) to a unit or subinterval of the unit, but does not pinpoint the location within the unit or subinterval.) In the sample run for *Arb. globif*, the aligned output is 1,893 characters long. Since the current alignment length is 1,892 columns, one column position must have been added. Looking at the second section in Appendix 4, one can see the dash indicating this new column in column position 1253. The third line contains the aligned version of the *Arb. globif* sequence that already existed in the alignment. (For a new species that does not already have an entry in the alignment, this third line is removed, so a total of three associated lines would be displayed, not four.) The fourth line is the aligned version of the *E. coli* sequence (our standard reference sequence) that is already present in the 16S alignment. It is displayed for easy comparison.

The third and final section contains one Prolog clause. The clause name is *alignment_row/3* and it has three arguments. The first argument contains the name of the molecule we are dealing with (in this case, *16S*). The second argument contains the name of the species (*Arb. globif*). The third argument contains the complete aligned string outputted by our tool. This third section is used when we wish to bring up the output into a text editor. The uninterrupted aligned string in the third argument of the Prolog clause is much easier to handle than the aligned string spread over 32 lines in the second section. Also, the output string in the third argument (unlike the output string in the second section) contains the parentheses and brackets that the parser inserted to mark off any regions where columns were added. (These parentheses and brackets flag such regions as anomalies for the biologists to look at. For more information on what symbols are used where, see Appendix 6, where the symbols are catalogued.) Looking at the third argument of the alignment_row/3 clause in Appendix 4, one can see one pair of parentheses in the string. These parentheses surround the characters placed in unit *b1025_1028*, where, as one can see by referring back to the first and second sections, one column has been added relative to the alignment.

Table 3: Results for alignment insertion runs on new species

Number of Species Lying in Range	Percentage of Total Species	Range (percentage of bases correctly placed by the tool in the aligned sequence)
2	11.11%	100.00%
4	22.22%	99.5 - 99.99%
2	11.11%	99.0 - 99.49%
3	16.67%	98.5 - 98.99%
4	22.22%	98.0 - 98.49%
0	0.00%	97.0 - 97.99%
2	11.11%	96.0 - 96.99%
1	5.56%	95.0 - 95.99%
0	0.00%	< 95.0%

6.2 Insertion of New Species

Our insertion tool was also tested on a private set of 16S rRNA sequences from 18 species held by Woese's Urbana group. These sequences were not in the alignment at the time of testing, and hence the grammar extracted from the alignment did not reflect the contents of these species. However, an aligned version of the 16S sequence for each species did exist, thus allowing us to check the tool's aligned output against the expert human judgment of the Urbana biologists. The set contained a mixture of species; some were related quite closely to species already in the alignment, while several others appeared to lie quite distant from anything in the current alignment. This was a pseudo-random test in that we did not ask for any species in particular; we simply were given what the Urbana group had on hand at that moment.

The results were excellent and are shown in Table 3. Over eleven percent of the species were inserted perfectly, while 44.4% came in at an accuracy of 99% or higher and 83.3% scored at 98% or better in correct base placement. All but one species (that is, 94.4% of all species) scored at 96.2% or higher. Within the random error range implicit in the use of a relatively small number of species in this run, the results are remarkably similar to the run in Section 6.1 where we performed our tests on species from the alignment (and thus where the species being inserted were represented in the grammar) and 95.2% of the species scored at 97.0% or higher. The results here were also quite robust in regard to large sequences of consecutive unknown bases confusing the parser. To make the test as tough as possible, I allowed sequences containing up to 75 consecutive unknowns. Fifteen of the sequences tried had strings of 50 unknowns or more.

The worst-scoring species came in at 95.5%. Given its low score and the relatively low number of pins generated for it, I guessed that this was a rather unusual species compared to the current contents of the 16S alignment. Dr. Olsen later confirmed that guess, calling the sequence "very strange". The parse of this particular species was also hurt somewhat by a string of over 60 consecutive unknown bases in a variable region (where such factors can be quite detrimental).

There were a total of eight localized failures spread over 3 of the 18 species tested. Three moderate-sized failures occurred in the worst-scoring species described in the preceding paragraph. One small failure occurred in a species coming in at 96.2% accuracy. The remaining four failures (two fairly large) all occurred in one species, which still managed to come in at a score of 98.0%.

Interestingly, if one compares Tables 2 and 3, one sees that the percentage of bases accurately positioned by the insertion tool for both the species already in the alignment and the species not in the alignment are quite similar. Thus it appears that our insertion tool is working as well for species outside the alignment as for those species already in the alignment and hence represented in the grammar. I conclude that the grammar (at least when combined with the pinning technique) is not "brittle", that is, is not so specific to the contents of the alignment (upon which it was derived) that it fails on other species. The localized parse failures in three of the private species do not appear large enough or frequent enough to affect this overall conclusion. The current alignment contents therefore appear to be a fairly representative sampling of the universe of possibilities (at least for the outside sequences tried so far).

The time for a complete insertion run (from program start to program end with output automatically stored in a file) was typically less than 10 minutes on a Sun workstation. Only 4 species of the 18 tried exceeded that limit, and only 3 did so substantially. As one might expect, those 3 species were the ones that contained localized parse failures, and hence did the most amount of backtracking in the parse. The worst species spent slightly over 100 minutes in the actual parse and another 18 minutes doing the indel insertion. The great majority of the species spent 2 minutes or less in the parse and another 2 minutes or less in indel insertion. The time spent in the pinning component was only a few minutes for all 18 species; it never exceeded 5 minutes.

Since these species are still private data, I cannot give an output listing that shows a full sequence. However, I have put in Appendix 5 partial outputs for three sample insertion runs. Each of these partial outputs contains the first of the three sections of the total run output, and thus shows which structural units and bases were in error for the species.

6.3 Types of Mistakes Seen and Their Causes

The aligned sequence output from our program can differ from the preexisting human-aligned version in two ways:

a) A base can be assigned to the wrong structural unit.

b) A base can be assigned to the correct structural unit, but misarranged inside it when the indels are added to position the bases.

The place where a mistake of type (a) most often occurs is at the boundary between the cap of a stem-loop and the associated lhs and rhs. A base or two from the sides are moved incorrectly into the cap, or vice versa. I have provided below an example from the output of one of the private species in Section 6.2. The entire stem-loop covers *b829_857*. The lhs is unit *b829_840*, the cap is *b841_845*, and the rhs is *b846_857*. (For help in visualizing, refer back to the secondary structure diagram in Figure 2.) Two *C*s have been incorrectly shifted from the

cap into the sides by our program.

```
struc_unit(b829_840): not ok
              new version:        GAUGUUUGGUGCC-
    aligned private seq version: GAUGUUUGGUGC--
    # of bases correctly placed: 12
struc_unit(b841_845): not ok
              new version:        --UAG----
    aligned private seq version: -CUAGC---
    # of bases correctly placed: 3
struc_unit(b846_857): not ok
              new version:        -CGUACUGAGUGUC
    aligned private seq version: --GUACUGAGUGUC
    # of bases correctly placed: 12
```

There does not appear to be any particular location where an error of type (b) is most likely. (One could say that such errors occur most often in the most variable regions of the molecule, such *b63_104*, *b179_219*, and *b1435_1466*, but that is only common sense. All errors occur most often in the variable regions where pins are hardest to find and the grammar rules are weakest.)

I believe that the reasons for disagreements between our output and the version already aligned can be broken down into five categories:

(a) The parser is not making full use of the information available in the alignment. For example, the *loop_unit* concept was not employed for the b829_857 stem-loop shown above.

(b) Humans are basing some of their judgments on information not contained in either the alignment or the associated phylogenetic tree. Perhaps they have performed some test in their lab, looked at some X-ray crystallographic data, or whatever, and that has influenced the base placement. (I believe that this situation played a very small role in the mistakes seen in our program's output for the 16S alignment, but it might be a larger factor when working with other alignments.)

(c) Essentially arbitrary (or, at least, hard-to-explain) decisions are being made by the humans in base placement. The people at Urbana have told us that a very small number of base placements in the 16S alignment do indeed fall into this category. There simply was not enough data to make an incontrovertible placement. Members of the Urbana team themselves disagree on where such bases should be placed. Obviously, any computerized insertion program is going to have trouble in such situations. I believe that some of our mistakes in the output for the 16S runs are due to this residual disagreement limit where even human experts disagree.

(d) Mistakes exist in the alignment. Mistakes do occur; maintaining an alignment is a big job, and no human is infallible. In fact, we soon shall be converting over to a new version of the Urbana alignment (and its associated phylogenetic tree) in which several such errors have been corrected by the Urbana team. This should have a beneficial result on our insertion outcomes. (Note that one of the auxiliary uses of an alignment insertion tool like ours would be to run it on all the species already in an alignment. The disagreements could then be checked to see whether the problems lie in the version of the sequence in the alignment rather than in the tool output. In such a way our tool could help maintain a "clean" alignment.)

(e) Localized parse failures negate the use of the information stored in the grammar. There may always be the possibility of one or more localized parse failures in a new species where the

species does not obey even the weakest grammar rule extracted from the alignment. The 16S alignment is already a healthy sampling of the possibilities, but it by no means covers everything yet. (And maybe it never will. There may or may not be some finite limit on the number of biologically realistic patterns that can be used in this molecule.) Indeed, localized parse failures occurred in 3 of the 18 Urbana private species, as pointed out in Section 6.2. When such failures happen, the only guidance we have inside the region of failure is whatever pins have been found in that region. That may not be good enough to come up with a completely correct alignment in that region.

7 Discussion

The accuracy currently obtained by our tool is certainly good enough to aid biologists in maintaining an alignment. When a new species aligned via the insertion tool is now displayed on-screen for viewing, its alignment will be close enough that the remaining errors will be far easier to detect and handle. And, of course, all the indel insertion, all the hackwork, necessary to get to this point no longer has to be done by humans. The tool can be used on any Sun workstation with a fair-sized RAM memory (say 20 megabytes) running under Unix. Therefore, at this point, even without any further development, I would call the the insertion tool project a success.

To summarize, the novel features of this alignment insertion tool are as follows:

(a) A pinning technique that yields pins of almost 100% certainty. The reason we can make such extensive use of pins in the parser program is that we can trust them to be accurate.

(b) A grammar to utilize secondary structure information. In those variable regions where pins (or any sort of correspondence between species in terms of primary sequence) are difficult to find, the use of structural (helix) information is vital. The grammar/parser approach is one excellent way of organizing such information. Note that the pinning program makes absolutely no use of secondary structure; that is entirely the province of the parser.

And so the two programs, pinning and parser, work together. The pins allow the parser to limit incorrect parses to small regions and start anew at the next pin after the area of failure. The parser complements the pins in the variable regions where pins are extremely difficult to find.

Although, as I stated in the last section, the results are quite good, there are still improvements I wish to make. The addition of a loop_unit for *b829_b57* might improve the insertion in several species; other minor changes like this are part of the natural "tuning up" process that we shall go through as the tool is put into use and we get feedback from more runs. We may also find more heuristic rules that we can incorporate in the grammar in the same manner as the tetra-loop cap composition motif heuristic described earlier. A heuristic such as that may or may not apply to other rRNA molecules. However, even if it cannot be transferred over it shows that there is always the possibility that the curator of a database for a particular molecule can uncover such rules and thus improve alignment insertion accuracy.

In the future we might also include in the loop_unit construct calculations based on Zuker's secondary structure prediction algorithm, if that turns out to be useful. (Zuker's algorithm is one of a class of dynamic programming algorithms that use a global minimum energy search method to predict secondary structure in RNA sequences. Briefly, the algorithm derives the best structure for a given part of the molecule from the best structures of the smaller sequences embedded within it, the "best" here referring to that with the minimum free energy. More information on this subject can be found in references [55, 56, 57].)

The grammar/parser concept is a neat organizational principle that can easily incorporate new tests and constraints for a given type of structural unit, a linear group of structural units that follow one another, or any defined collection of units from any place in the molecule.

As I mentioned earlier, the subtree of species returned by the pinning component of our tool provides the end user some guidance on where the new species falls in the phylogenetic tree. It should be possible to supplement this information by modifying the indel insertion section of the parser. It is extremely unlikely that one alignment species would be found as the best match to the new species across all the structural units. However, if we store the names of the species that serve as the best matches for all the units, it is quite possible that a few species will serve as the best matches in several structural units and thus show up quite frequently. The program could be changed so that a ranking of all species that were used (say, five or more times as a best match) would be included in the tool's output. Presumably the top-ranking species would be the ones lying closest to where we should insert the new species in the phylogenetic tree, and hence such a ranking would be useful to biologists.

Steve Smith, formerly at Urbana and now working in Walter Gilbert's lab at Harvard, has created a software environment for sequence analysis that has built-in "hooks" to which outside tools can be attached. I would like to attach our alignment insertion code as such a tool, with a form of output that can be fed directly into Smith's customized text editor. Doing this, we could bring up on-screen the proposed aligned version of the new species in a color-coded fashion. The bases that were pinned could be one color, the ordinary unpinned bases another color, and the suspected problem regions yet another.

We also hope to extend our work to other molecules, the 23S rRNA for a start. With automated grammar generation, it might be possible to set up operation for a new molecule in a matter of a few days. Thus the additional goal mentioned in Section 1.5 of transferring or duplicating at least a portion of the skill of the 16S rRNA human experts to an alignment for another molecule looks to be within reach.

In conclusion, the grammar/parser approach promises to be useful for any molecule where (a) secondary structure in alignment construction is important and (b) at least a minimal alignment and phylogenetic tree already exist, to provide the necessary starting point.

Acknowledgments

First and foremost, I thank Dr. Ross Overbeek. It would be hard to imagine a better thesis supervisor. He was always there when I needed him, yet knew when stand back and let me

make my own mistakes.

Dr. Carl Woese and Dr. Gary Olsen could always be reached when necessary; I thank them for their time and advice.

I had several profitable discussions with Dr. Steve Winker on RNA motifs.

I also acknowledge the aid of the support crew in the Argonne Mathematics and Computer Science Department: John Winans, Mark Henderson, Gene Rackow, and Rick Stevens. These people kept the department's Sun workstation network up on a 24-hour basis and were a major factor in making Argonne such a wonderful place to work.

Finally, I express my appreciation to the members of my thesis committee, Drs. James Zull, Randy Beer, and Chris Cullis, for their support. A thesis in the subarea of computational molecular biology (particularly with a linguistics methodology) would be a *rara avis* in any biology department.

References

1. Cech, T. (1986) "RNA As an Enzyme" Scientific American, vol. 255, no. 5, 64-75.

2. Lake, J. A. (1981) "The Ribosome," Scientific American. August 1981, 84-97.

3. Gutell, R. R., Weiser, B., Woese, C. R., and Noller, H. F. (1985) "Comparative Anatomy of 16S-like Ribosomal RNA," Progress in Nucleic Acid Research and Molecular Biology, vol. 32, 155-216.

4. Thompson, J. F., and Hearst, J. E. (1983) "Structure-Function Relations in E. coli 16S RNA," Cell, vol. 33, 19-24.

5. Covington, M. A., Nute, D., and Villino, A. (1988) Prolog Programming in Depth. Scott, Foresman and Company, Glenview, Ill.

6. Sterling, L., and Shapiro, E. (1986) The Art of Prolog: Advanced Programming Techniques. The MIT Press, Cambridge, Mass.

7. Clocksin, W. F., and Mellish, C. S. (1987) Programming in Prolog, 3rd edition. Springer-Verlag, New York.

8. Searls, D. B. (1988) "Representing Genetic Information with Formal Grammars" in Proceedings of the Seventh National Conference on Artificial Intelligence, pp. 386-391.

9. Searls, D. B. (1989) "Investigating the Linguistics of DNA with Definite Clause Grammars" in Proceedings of the North American Conference on Logic Programming, vol. 1, pp. 189-208.

10. Searls, D. B. (1990) "The Computational Linguistics of Biological Sequences," Unisys Paoli Research Center report CAIT-KSA-9010.

11. Vellino, A. N. (1987) "Searching Chemical Substructures using Prolog," Research Report 01-0018, Advanced Computational Methods Center, University of Georgia.

12. Karickhoff, S. W., et al. (1987) "Predicting Chemical Parameters with Prolog," Research Report 01-0020, Advanced Computational Methods Center, University of Georgia.

13. Rawlings, C. J., et al. (1985) "Reasoning about Protein Topology Using the Logic Programming Language Prolog," J. Mol. Graphics, vol. 3, 151-157.

14. Rawlings, C. J., et al. (1986) "Using Prolog to Represent and Reason about Protein Structure" in Proceedings of the Third International Conference on Logic Programming, pp. 536-543.

15. Park, K., and Huntsberger, T. L. (1990) "Inference of Context Free Grammars for Syntactic Analysis of DNA Sequences," Working Notes of the AAAI Spring 1990 Symposium on Artificial Intelligence and Molecular Biology, 110-114.

16. Brendel, V., Beckmann, J. S., and Trifinov, E. N. (1986) "Linguistics of Nucleotide Sequences: Morphology and Comparison of Vocabularies," Journal of Biomolecular Structure and Dynamics, vol. 4, 11-21.

17. Pietrokovski, S., Hirshon, J., and Trifonov, E. N. (1990) "Linguistics Measure of Taxonomic and Functional Relatedness of Nucleotide Sequences" Journal of Biomolecular Structure and Dynamics, vol. 7, 1251-1268.

18. Pevzner, P. A., Borodovsky, M. Y., and Mironov, A. A. (1989) "Linguistics of Nucleotide Sequences I: The Significance of Deviations from Mean Statistical Characteristics and Prediction of the Frequencies of Occurrence of Words," Journal of Biomolecular Structure and Dynamics, vol. 6, 1013-1026.

19. Pevzner, P. A., Borodovsky, M. Y., and Mironov, A. A. (1989) "Linguistics of Nucleotide Sequences II: Stationary Words in Genetic Texts and the Zonal Structure of DNA," Journal of Biomolecular Structure and Dynamics, vol. 6, 1027-1038.

20. Borodovsky, M. Y., and Gusein-Zade, S. M. (1989) "A General Rule for Ranged Series of Codon Frequencies in Different Genomes," Journal of Biomolecular Structure and Dynamics, vol. 6, 1001-1012.

21. Brendel, V., and Busse, H. G. (1984) "Genome Structure Described by Formal Languages," Nucleic Acids Research, vol. 12, 2561- 2568.

22. Collado-Vides, J. (1991) "The Search for a Grammatical Theory of Gene Regulation Is Formally Justified by Showing the Inadequacy of Context-free Grammars," CABIOS, vol. 7, 321-326.

23. Collado-Vides, J. (1989) "A Transformational-grammar Approach to the Study of the Regulation of Gene Expression," J. Theor. Biol., vol. 136, 403-425.

24. Collado-Vides, J. (1989) "Towards a Grammatical Paradigm for the Study of the Regulation of Gene Expression," in Goodwin, B. and Saunders, P. (eds.) Theoretical Biology: Epigenetic and Evolutionary Order. Edinburgh University Press, Edinburgh, pp. 211-224.

25. Collado-Vides, J. (1991) "A Syntactic Representation of Units of Genetic Information," J. Theor. Biol., vol. 148, 401- 429.

26. Head, T. (1987) "Formal Language Theory and DNA: an Analysis of the Generative Capacity of Specific Recombinant Behaviors," Bull. Math. Biol., vol. 49, 737-759.

27. Jimenez-Montano (1984) "On the Syntactic Structure of Protein Sequences and the Concept of Grammar Complexity," Bull. Math. Biol., vol. 46, 641-659.

28. Angier, Natalie (1991) "Biologists Seek the Words in DNA's Unbroken Text: Linguistic Methods Help Experts Pinpoint Key Genetic Instructions Amid the Biochemical Babble," New York Times, July 7, 1991.

29. Hillis, D., and Moritz, C. eds. (1990) Molecular Systematics. Sinauer Associates.

30. Olsen, G. J. (1987) "Earliest Phylogenetic Branchings: Comparing rRNA-based Evolutionary Trees Inferred with Various Techniques" in Cold Spring Harbor Symposia on Quantitative Biology, vol. LII, pp. 825-837.

31. Bown, W. (1990) "A New Tree of Life Takes Root," Science, 11 August 1990, 30.

32. Woese, C. R. (1987) "Bacterial Evolution," Microbiological Reviews, vol. 51, 221-271.

33. Margulis, L., and Guerrero, R. (1991) "Kingdoms in Turmoil," New Scientist, 23 March 1991, 46-50.

34. Woese, C. R., Kandler, O., and Wheelis, M. L. (1990) "Towards a Natural System of Organisms: Proposal for the Domains Archaea, Bacteria, and Eucarya," Proceedings of the National Academy of Sciences (USA), vol. 87, 4576-4579.

35. Noller, H.F. (1984) "Structure of Ribosomal RNA," Annual Review of Biochemistry, vol. 53, 119-162.

36. Brown, J. W. (1991) "Phylogenetic Comparative Analysis of RNA Structure on Macintosh Computers," CABIOS, vol. 7, 391-393.

37. Thanaraj, T. A., Kolaskar, A. S., and Pandit, M. W. (1989) "An Extension of the Graph Theoretical Approach to Predict the Secondary Structure of Large RNAs: Samples of 16S and 23S rRNAs from E. coli As a Case Study," CABIOS, vol. 5, 211-218.

38. Maozed, D., Stern, S., and Noller, H. F. (1985) J. Mol. Biol., vol. 187, 399-416.

39. Abrahams, J. P., et al. (1990) "Prediction of RNA Secondary Structure, Including Pseudoknotting, by Computer Simulation," Nucleic Acids Research, vol. 18, 3025-3044.

40. Overbeek, R., and Foster, I. (1991) "Aligning Multiple RNA Sequences," Argonne National Laboratory Mathematics and Computer Science Division preprint MCS-P207-0191. (to appear in Boyer, R. S. ed. (1991) Festschrift for W. W. Bledsoe. Kluwer Academic Publishers).

41. Carillo, H., and Lipman, D. (1988) "The Multiple Sequence Alignment Problem in Biology," SIAM J. Appl. Math., vol. 48, 1073-1082.

42. Korn, L. J., Queen, C. L., and Wegman, M. N. (1977) "Computer Analysis of Nucleic Acid Regulatory Sequences," Proceedings of the National Academy of Sciences (USA), vol. 77, 6309-6313.

43. Olsen, G. (1990) personal communication.

44. Smith, T.F., and Waterman, M.S. (1981) "Identification of Common Molecular Subsequences," J. Mol. Biol., vol. 147, 195-197.

45. Gotoh, O. (1982) "An Improved Algorithm for Matching Biological Sequences," J. Mol. Biol., vol. 162, 705-708.

46. Waterman, M. S., and Eggert, M. (1987) "A New Algorithm for Best Subsequence Alignments with Application to tRNA-rRNA Comparisons," J. Mol. Biol., vol. 197, 723-728.

47. Tyler, E. C., Horton, M. R., and Krause, P. R. (1991) "A Review of Algorithms for Molecular Sequence Comparison," Computers and Biomedical Research, vol. 24, 72-96.

48. Wyatt, J. R., Puglisi, J. D., and Tinoco, I. (1989) "RNA Folding: Pseudoknots, Loops and Bulges," BioEssays, vol. 11, no. 4, 100-106.

49. Woese, C. R., Winker, S., and Gutell, R. "The Architecture of Ribosomal RNA: Constraints on the Sequence of Tetra-loops," Proceedings of the National Academy of Sciences (USA), vol. 87, 8467-8471.

50. Winker, S., Overbeek, R., Woese, C. R., Olsen, G. J., and Pfluger, N. (1989) "An Automated Procedure for Covariation-Based Detection of RNA Structure," Argonne National Laboratory report ANL-89/42, pp. 6-8.

51. Boisen, S. (1987) "Language Processing Using Definite Clause Grammars," AI Expert, June 1987, 46-56.

52. Gazdar, G., and Mellish, C. (1989) Natural Language Processing in Prolog: An Introduction to Computational Linguistics. Addison-Wesley, Reading, Mass.

53. Pereira, F. C. N., and Shieber, S. M. (1987) Prolog and Natural-Language Analysis. Center for the Study of Language and Information, Menlo Park, CA.

54. Grishman, R., (1986) Computational Linguistics. Cambridge University Press, Cambridge, pp. 32-33.

55. Zuker, M., and Stiegler, P. (1981) "Optimal Computer Folding of Large RNA Sequences Using Thermodynamics and Auxiliary Information," Nucleic Acids Research, vol. 9, 133-148.

56. Zuker, M., and Sankoff, D. (1984) "RNA Secondary Structures and Their Prediction," Bull. Math. Biol., vol. 46, 591-621.

57. Waterman, M. S., and Smith, T. F. (1986) "Rapid Dynamic Programming Algorithms for RNA Secondary Structure," Advances in Applied Mathematics, vol. 7, 455-464.

Appendix 1: Smith-Waterman Matrices

Smith-Waterman matrices:

```
    /*G   A   T/U C   R   Y   M   K   S   W   H   B   V   D   N */
similarity_matrix(me(
  me( 18,-18,-18,-18,  0,-18,-18,  0,  0,-18,-18, -6, -6, -6, -9), % G
  me(-18, 18,-18,-18,  0,-18,  0,-18,-18,  0, -6,-18, -6, -6, -9), % A
  me(-18,-18, 18,-18,-18,  0,-18,  0,-18,  0, -6, -6,-18, -6, -9), % T/U
  me(-18,-18,-18, 18,-18,  0,  0,-18,  0,-18, -6, -6, -6,-18, -9), % C
  me(  0,  0,-18,-18, 18,-18, -9, -9, -9, -9,-12,-12, -6, -6, -9), % R
  me(-18,-18,  0,  0,-18, 18, -9, -9, -9, -9, -6, -6,-12,-12, -9), % Y
  me(-18,  0,-18,  0, -9, -9, 18,-18, -9, -9, -6,-12, -6,-12, -9), % M
  me(  0,-18,  0,-18, -9, -9,-18, 18, -9, -9,-12, -6,-12, -6, -9), % K
  me(  0,-18,-18,  0, -9, -9, -9, -9, 18,-18,-12, -6, -6,-12, -9), % S
  me(-18,  0,  0,-18, -9, -9, -9, -9,-18, 18, -6,-12,-12, -6, -9), % W
  me(-18, -6, -6, -6,-12, -6, -6,-12,-12, -6, -6,-10,-10,-10, -9), % H
  me( -6,-18, -6, -6,-12, -6,-12, -6, -6,-12,-10, -6,-10,-10, -9), % B
  me( -6, -6,-18, -6, -6,-12, -6,-12, -6,-12,-10,-10, -6,-10, -9), % V
  me( -6, -6, -6,-18, -6,-12,-12, -6,-12, -6,-10,-10,-10, -6, -9), % D
  me( -9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9)  % N
)).


    % G   A   T/U C   R   Y   M   K   S   W   H   B   V   D   N
helix_match_matrix(me(
  me(-18,  3, 13, 18, -8, 15, 10, -3,  0,  8, 11,  4,  1, -1,  4), % G
  me(  3,-18, 18,-11, -8,  3,-15, 10, -4,  0, -4,  3, -9,  1, -2),  % A
  me( 13, 18,-11,-18, 15,-15,  0,  1, -3,  3, -4, -6,  4,  6,  0),  % T/U
  me( 18,-11,-18,-18,  3,-18,-15,  0,  0,-15,-16, -6, -4, -4, -8),  % C
  me( -8, -8, 15,  3, -8,  9, -2,  4, -2,  4,  3,  3, -4,  0,  1),  % R
  me( 15,  3,-15,-18,  9,-17, -8,  0, -2, -6,-10, -6,  0,  1, -4),  % Y
  me( 10,-15,  0,-15, -2, -8,-15,  5, -2, -8,-10, -2, -6, -2, -5),  % M
  me( -3, 10,  1,  0,  4,  0,  5, -1, -2,  5,  3, -1,  2,  3,  2),  % K
  me(  0, -4, -3,  0, -2, -2, -2, -2,  0, -4, -3, -1, -2, -3, -2),  % S
  me(  8,  0,  3,-15,  4, -6, -8,  5, -4,  1, -4, -1, -3,  3, -1),  % W
  me( 11, -4, -4,-16,  3,-10,-10,  3, -3, -4, -8, -3, -3,  1, -3),  % H
  me(  4,  3, -6, -6,  3, -6, -2, -1, -1, -1, -3, -3,  0,  0, -1),  % B
  me(  1, -9,  4, -4, -4,  0, -6,  2, -2, -3, -3,  0, -4, -1, -2),  % V
  me( -1,  1,  6, -4,  0,  1, -2,  3, -3,  3,  1,  0, -1,  2,  0),  % D
  me(  4, -2,  0, -8,  1, -4, -5,  2, -2, -1, -3, -1, -2,  0, -2)   % N
)).
```

Appendix 2: Urbana Alignment Annotation

The annotation to the Urbana alignment provided by Dr. Gary Olsen defining the elementary structural units follows.

```
LOCUS       R.O.pair     1892 BP
DEFINITION  Pairing for Overbeek
ORIGIN
   1 __________ __________ __________ __________ _<<<<<___[ [[_>>>>>_<
  61 <<<_[[[[[[ _<<<<<<<<< _______<<< _[[[__<<<_ _{{{{{{{{{{ {{{{{{{{{{
 121 {{{{{{{{{{ {{{======= =}}}}}}}}} }}}}}}}}} }}}}}}}}} }}}}>>>___
 181 _____<<<__ _________< <<<<<<____ [[____(((( (((_<<<<<< <__[[[[[[=
 241 ===]]]]]]_ __>>>>>>>_ _____<<<<< <<<<<<<<<< ====>>>> >>>>>>>>>>
 301 ______<<<< <<__[[[[[[ [========= ==]]]]]]]> >>>>>___)) )))))___]]
 361 >>>>>>>_<< <__{__[^[[ ^^^^^((((( (^((====== =)))))))] ]]_____}>>
 421 >__[[[[((( (====)))^ ^^^]]]]>>> _<<<<<<<== ======>>>> >>>_<<<^<=
 481 =====>>>>_ __]]]_>^>> _______{_[ [[___(_((( (======))) ))_]]]}_>>
 541 >>>>>>>___ <<^<<_____ _[[[[[[==== =]]]]]____ _((((===== ))))>>>>_<
 601 <__[[[[[__ _{_______( (((((((((== =====))))) )))))_____ _}____]]]]
 661 ]]]___>>__ _<<<<<<___ ___[[[[[[= ========== =======]]] ]]]>>>>>>_
 721 ]]]]]]>>>> __________ <<<_______ [[[[___((( _<<<<<<<<_ [[(((((((((
 781 _______{{{ {{=======} }}}}_____ )))))))__] ]>>>>>>>>_ __<<<<<<<<
 841 ___[[[[^[[ [(((((((((( ^^^^^(((__ _{{{====== }}^}_____ ))))))))))
 901 ))____(___ _{{====}}) ]]]]]]]__> >>>>>>>__ _)))___]]] ]____<<<<<
 961 <<_______ [[[[====== ===]]]]___ _>>>>>>>__ ________(( (((((__[[[
1021 [[[[[[[[[[ [========= ]]]]]]]]]] ]]]]___[[= ===]]_____ ))))))_>>>
1081 __________ ((((=====) )))_______ _____]]]__ <<_[[[^((( (((^^(____
1141 ___[[[[^[[ __<<<<<<<<< [[_______( (======== )_________ __(((((((_
1201 _____<<_[[ [[[[[____( (((((((=== ==)))))))) )__(((((((( (========)
1261 )))))))]]] ]]]__>>___ <<<<__[___ ___(((_{{{ =======}}} _)))_____]
1321 __>>>>___< <^[[[[__{{ _(({{{{____ <<<<<<[[[= ===]]]__[[ [[======]]
1381 ]]__>>>>>> _____<<<<_ _[[[[[[[__ ____((__{{ {<<<<<==== =====>>>>>
1441 ^}}}))____ _]]]]]]]__ ___[[[[[[== =====]]]]^ ^]________ __>>>>____
1501 __}}}___)) ____}}]]^] ]>>____)^) )^))))___] ]^>>>>>>>> _____<<<<<
1561 <<_______[ [[___((___ _{{{=====} }}________ ______))__ __]]]_____
1621 _>>>>>>>__ ____<____[ [[[[[[==== ====]]]]]] ]____>____ _]]]]]]___
1681 _<<<<<<<<= =======>>> >>>>>_____ ______)))) )))]]]^>>_ __________
1741 _((((((((( ((((((((((( (((____<<< <<<<<^[^[[ [___{{{=== =======}}}
1801 _]]]]>>>>> >>___)))))) )))))))))) ))))))___ __________ __<<<<<<<<
1861 <====>>>>> >>>>_____ __________ --
```

The equal sign ("="), the triangular "hat", and the underscore ("_") symbols indicate that an alignment column falls into what is called a gap unit in the grammar. The "<>", "( )", "{ }", and "[ ]" symbol pairs indicate a pair of alignment columns between which base bonds form (and thus which fall within a lhs and a rhs of a helix).

Appendix 3: Base Invariancy Coding Symbols

The base invariancy coding symbols used in the grammar are as follows (note that both upper-case and lower-case symbols are permitted for the bases):

```
gn --> ''g''.
gn --> ''n''.
gn --> ''G''.
gn --> ''N''.

an --> ''a''.
an --> ''n''.
an --> ''A''.
an --> ''N''.

cn --> ''c''.
cn --> ''n''.
cn --> ''C''.
cn --> ''N''.

un --> ''u''.
un --> ''n''.
un --> ''U''.
un --> ''N''.

rn --> ''g''.
rn --> ''a''.
rn --> ''n''.
rn --> ''G''.
rn --> ''A''.
rn --> ''N''.
rn --> ''r''.
rn --> ''R''.

yn --> ''c''.
yn --> ''u''.
yn --> ''n''.
yn --> ''C''.
yn --> ''U''.
yn --> ''N''.
yn --> ''y''.
yn --> ''Y''.

mn --> ''a''.
```

```
mn --> ''c''.
mn --> ''n''.
mn --> ''A''.
mn --> ''C''.
mn --> ''N''.
mn --> ''m''.
mn --> ''M''.

kn --> ''g''.
kn --> ''u''.
kn --> ''n''.
kn --> ''G''.
kn --> ''U''.
kn --> ''N''.
kn --> ''k''.
kn --> ''K''.

sn --> ''g''.
sn --> ''c''.
sn --> ''n''.
sn --> ''G''.
sn --> ''C''.
sn --> ''N''.
sn --> ''s''.
sn --> ''S''.

wn --> ''a''.
wn --> ''u''.
wn --> ''n''.
wn --> ''A''.
wn --> ''U''.
wn --> ''N''.
wn --> ''w''.
wn --> ''W''.

% g is ASCII 103
% a is ASCII  97
% u is ASCII 117
% c is ASCII  99

hn --> [OneBase], { integer(OneBase), OneBase =\= 103}. % not g
bn --> [OneBase], { integer(OneBase), OneBase =\= 97}.  % not a
vn --> [OneBase], { integer(OneBase), OneBase =\= 117}. % not u
dn --> [OneBase], { integer(OneBase), OneBase =\= 99}.  % not c
```

Appendix 4: Insertion Run Results for *Arb. globif*

The complete output listing for the insertion run for species Arb.globif follows below:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Species Arb.globif already exists in the alignment.
Commentary on the correctness of the insertion, broken down by
structural unit, re already present version follows.
(Only structural units where the parse differs from the alignment
version are displayed.)

struc_unit(b184_186): not ok
    new version:       ACUCC--UC---AUC
    alignment version: ACUCC--U-C--AUC
    # of bases correctly placed: 9
struc_unit(b447_454): not ok
    new version:       GAAG--AAG--
    alignment version: GAAG--AAGC-
    # of bases correctly placed: 7
struc_unit(b455_462): not ok
    new version:       CG-------
    alignment version: ---------
    # of bases correctly placed: 0
struc_unit(b463_469): not ok
    new version:       ---AAA-
    alignment version: --GAAA-
    # of bases correctly placed: 3
struc_unit(b470_477): not ok
    new version:       ---------G
    alignment version: ----------
    # of bases correctly placed: 0
struc_unit(b478_487): not ok
    new version:       ---UGACGGUA
    alignment version: --GUGACGGUA
    # of bases correctly placed: 8
struc_unit(b829_840): not ok
    new version:       GGUGUGGGGGACAU
    alignment version: GGUGUGGGGGAC--
    # of bases correctly placed: 12
struc_unit(b841_845): not ok
    new version:       --U-CC---
    alignment version: AUUCCAC--
    # of bases correctly placed: 2
struc_unit(b846_857): not ok
    new version:       ACGUUUUCCGCGCC
    alignment version: --GUUUUCCGCGCC
    # of bases correctly placed: 12
struc_unit(b1024): not ok
    new version:       --
    alignment version: U-
    # of bases correctly placed: 0
struc_unit(b1025_1028): not ok
    new version:       (U-----CUC)
    new ver stripped : U-----CUC
    alignment version: CU----CC
    new ver and ali with asterisks added to align with each other:
      new ver:U-----CUC
          ali:CU----C*C
    # of bases correctly placed: 2
struc_unit(b1029_1032): not ok
```

```
    new version:       --CUUUU-
    alignment version: --UUUU--
    # of bases correctly placed: 3
struc_unit(b1136_1138): not ok
    new version:       --GUAA---
    alignment version: --GUAAUG-
    # of bases correctly placed: 4
struc_unit(b1139_1144): not ok
    new version:       ----UGGCGGG
    alignment version: ------GCGGG
    # of bases correctly placed: 5

Number of correctly placed bases: 1516

Total number of bases in species Arb.globif: 1531

Percentage of bases correctly placed: 99.02


Number of structural units completely correct: 378

Total number of structural units: 392

Percentage of structural units correctly placed: 96.43


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

new species = Arb.globif (species is already in alignment)
New species line length: 1893

Four lines follow in this order: new species, insertion marker,
new species as already in alignment, E. coli

(1)       ------------------------------UUUCAACGGAGAGUUUGAUCCUGGCUCAG
(1)
(1)       ------------------------------UUUCAACGGAGAGUUUGAUCCUGGCUCAG
(1)       ----------------------------------AAAUUGAAGAGUUUGAUCAUGGCUCAG


(61)      GAUGAACGCUGGCGGCG-UG-C-UUAACACAUGCAAGUCGAACG-AUG------------
(61)
(61)      GAUGAACGCUGGCGGCG-UG-C-UUAACACAUGCAAGUCGAACG-AUG------------
(61)      AUUGAACGCUGGCGGCA-GG-C-CUAACACAUGCAAGUCGAACG-GUAA-CAG-------


(121)     -----AUCCGGUG--CUUG--CACCGGGG-------------------AUUAGUGGCGA-
(121)
(121)     -----AUCCGGUG--CUUG--CACCGGGG-------------------AUUAGUGGCGA-
(121)     ------GAAGAAG--CUUG--CUUCUUU-------------G-CUG--ACGAGUGGCGG-


(181)     -ACGGGUGAGU---AAC-ACGUGAGUAA--CCUGC-CCUUGACUCUGGGAUAAGCCUGGG
(181)
(181)     -ACGGGUGAGU---AAC-ACGUGAGUAA--CCUGC-CCUUGACUCUGGGAUAAGCCUGGG
(181)     -ACGGGUGAGU---AAU-GUCUGGGA-A--ACUGC-CUGAUGGAGGGGGAUAACUACUGG


(241)     AAACUGGGUCUAAUACCGGAU-AUGACUCC--UC---AUC-GCAU-GGU--G-G-GGGGU
(241)
(241)     AAACUGGGUCUAAUACCGGAU-AUGACUCC--U-C--AUC-GCAU-GGU--G-G-GGGGU
(241)     AAACGGUAGCUAAUACCGCAU-AAC------------GUC-GCAA-GAC-----------


(301)     GGAAA-GC---------------UUUUU--------------GUG--GUUUUGG-AUGG
(301)
```

```
(301)      GGAAA-GC---------------UUUUU---------------GUG--GUUUUGG-AUGG
(301)      -CAAA-GAGGGGGACC--------UUCG---------GGCCUCUUG--CCAUCGG-AUGU

(361)      ACUCGCGGCCUAUCA-G-CUUG---UUGGUG-AGGUAAUGGCUCACCAAGGCGACGACGG
(361)
(361)      ACUCGCGGCCUAUCA-G-CUUG---UUGGUG-AGGUAAUGGCUCACCAAGGCGACGACGG
(361)      GCCCAGAUGGGAUUA-G-CUAG---UAGGUG-GGGUAACGGCUCACCUAGGCGACGAUCC

(421)      GUAGCCGGCCUGAGAGGGU-GACCGGCCACACUGGGACUGAGACACGGCCCAGACUC-CU
(421)
(421)      GUAGCCGGCCUGAGAGGGU-GACCGGCCACACUGGGACUGAGACACGGCCCAGACUC-CU
(421)      CUAGCUGGUCUGAGAGGAU-GACCAGCCACACUGGAACUGAGACACGGUCCAGACUC-CU

(481)      ACG--GGAGGCAGCA-G-UGGGGAAUAUUGCACAAU-GGGC-GAAA-GCCUGAUGCAGCG
(481)
(481)      ACG--GGAGGCAGCA-G-UGGGGAAUAUUGCACAAU-GGGC-GAAA-GCCUGAUGCAGCG
(481)      ACG--GGAGGCAGCA-G-UGGGGAAUAUUGCACAAU-GGGC-GCAA-GCCUGAUGCAGCC

(541)      ACGCCGCG-UGA-GGGAUGACGGCC--UUCG-GGUUGUAAA------------CCUCUU
(541)
(541)      ACGCCGCG-UGA-GGGAUGACGGCC--UUCG-GGUUGUAAA------------CCUCUU
(541)      AUGCCGCG-UGU-AUGAAGAAGGCC--UUCG-GGUUGUAAA------------GUACUU

(601)      UCAGUAGGGAAG--AAG--CG----------AAA----------G---UGACGGUACCU-
(601)
(601)      UCAGUAGGGAAG--AAGC------------GAAA------------GUGACGGUACCU-
(601)      UCAGCGGGGAGG--AA-GGGAGUAAAG-UUAAUAC-CUUUGC-UCA-UUGACGUUACCC-

(661)      G-CAG-AAGAAGCGCC-GGCUAACUACGUGCCAGCAGCCGCGGUAAUACGUAG-GGCGCA
(661)
(661)      G-CAG-AAGAAGCGCC-GGCUAACUACGUGCCAGCAGCCGCGGUAAUACGUAG-GGCGCA
(661)      G-CAG-AAGAAGCACC-GGCUAACUCCGUGCCAGCAGCCGCGGUAAUACGGAG-GGUGCA

(721)      AGCGUUAUCCGGAAUUAUUGGGCGUAAAGAGCUCGUAGGCGGUUU-GUCGCGUCUG-CCG
(721)
(721)      AGCGUUAUCCGGAAUUAUUGGGCGUAAAGAGCUCGUAGGCGGUUU-GUCGCGUCUG-CCG
(721)      AGCGUUAAUCGGAAUUACUGGGCGUAAAGCGCACGCAGGCGGUUU-GUUAAGUCAG-AUG

(781)      UGAAAGUCCGGGGCUCAACUCCGGAUC-UCCGGUGGGUACGGGCA-GACUAGAGUGAUGU
(781)
(781)      UGAAAGUCCGGGGCUCAACUCCGGAUC-UGCGGUGGGUACGGGCA-GACUAGAGUGAUGU
(781)      UGAAAUCCCCGGGGCUCAACCUGGGAAC-UGCAUCUGAUACUGGCA-AGCUUGAGUCUCGU

(841)      AGGGGAG-ACUGGAAUUCCU-----GGUGUAGCGGUGAAAUG-CGCAGAUAUCAGGAGGA
(841)
(841)      AGGGGAG-ACUGGAAUUCCU-----GGUGUAGCGGUGAAAUG-CGCAGAUAUCAGGAGGA
(841)      AGAGGGG-GGUAGAAUUCCA-----GGUGUAGCGGUGAAAUG-CGUAGAGAUCUGGAGGA

(901)      ACACCGA--UGGCGAAGGCAGGUCUCUGGGCAUUAACUGACGCUGAGGAGCGAAAGCAUG
(901)
(901)      ACACCGA--UGGCGAAGGCAGGUCUCUGGGCAUUAACUGACGCUGAGGAGCGAAAGCAUG
(901)      AUACCGG--UGGCGAAGGCGGCCCCCUGGACGAAGACUGACGCUCAGGUGCGAAAGCGUG

(961)      GG-GAGCGAACAGGAUUAGAUACCCUGGUAGUCCAUGCCGUAAACGUUGGGCA-CUAGGU
(961)
(961)      GG-GAGCGAACAGGAUUAGAUACCCUGGUAGUCCAUGCCGUAAACGUUGGGCA-CUAGGU
(961)      GG-GAGCAAACAGGAUUAGAUACCCUGGUAGUCCACGCCGUAAACGAUGUCGA-CUUGGA
```

```
(1021)    GUGGGGGACAU--U-CC---ACGUUUUCCGCGCCGUAGCUAACGCAUUAAGUGCCCCGCC
(1021)
(1021)    GUGGGGGAC--AUUCCAC----GUUUUCCGCGCCGUAGCUAACGCAUUAAGUGCCCCGCC
(1021)    GGUUGUGCC---CUUGA-----GGCGUGGCUUCCGGAGCUAACGCGUUAAGUCGACCGCC

(1081)    UGGGGAGUACGGCC-GCAAGGCUAAAACUC-AAA-GGAAUUG-ACGGGGGCCC--GC-A-
(1081)
(1081)    UGGGGAGUACGGCC-GCAAGGCUAAAACUC-AAA-GGAAUUG-ACGGGGGCCC--GC-A-
(1081)    UGGGGAGUACGGCC-GCAAGGUUAAAACUC-AAA-UGAAUUG-ACGGGGGCCC--GC-A-

(1141)    -CAAGCG-GCGGAGCAUGCGGA-UUAAUUCGAUGCAACGCGAAGAACCUUA-CCAAGGCU
(1141)
(1141)    -CAAGCG-GCGGAGCAUGCGGA-UUAAUUCGAUGCAACGCGAAGAACCUUA-CCAAGGCU
(1141)    -CAAGCG-GUGGAGCAUGUGGU-UUAAUUCGAUGCAACGCGAAGAACCUUA-CCUGGUCU

(1201)    UGACAUGGA-CCGG---ACCGC-CGCA-GAAAUGU-G-GUU--U-----CUC--CUUUU-
(1201)                                                      -
(1201)    UGACAUGGA-CCGG---ACCGC-CGCA-GAAAUGU-G-GUUU-CU----CC --UUUU--
(1201)    UGACAUC-C-ACGG---AAGUU-UUCA-GAGAUGA-G-AAUG-UG----CC --UUCG--

(1261)    GG----GG-CCGGU-UCA-------------------------------------------
(1261)
(1261)    GG----GG-CCGGU-UCA-------------------------------------------
(1261)    GG----AA-CCGUG-AGA-------------------------------------------

(1321)    --------CAGGUGGUGCAUGGUUGUCGUCAGCUCGUGUCGUGAGAUGUUGGGUUAAGUC
(1321)
(1321)    --------CAGGUGGUGCAUGGUUGUCGUCAGCUCGUGUCGUGAGAUGUUGGGUUAAGUC
(1321)    --------CAGGUGCUGCAUGGCUGUCGUCAGCUCGUGUUGUGAAAUGUUGGGUUAAGUC

(1381)    CCGCAACGAGCGCAACCCUCG-UUCCAUG---UUGCCAGCGC-------GUAA-------
(1381)
(1381)    CCGCAACGAGCGCAACCCUCG-UUCCAUG---UUGCCAGCGC-------GUAAUG-----
(1381)    CCGCAACGAGCGCAACCCUUA-UCCUUUG---UUGCCAGCGGU-------CCG-------

(1441)    UGGCGGG-GACUCAUGGGAGACUGCCGGGGU-CAA-CUCG--GAGG-A-AGGUGGGG-AC
(1441)
(1441)    --GCGGG-GACUCAUGGGAGACUGCCGGGGU-CAA-CUCG--GAGG-A-AGGUGGGG-AC
(1441)    G-CCGGG-AACUCAAAGGAGACUGCCAGUGA-UAA-ACUG--GAGG-A-AGGUGGGG-AU

(1501)    GACGUC--AAAUCAUCAUG-CCCCUUAUG-UC-UUGGGCUUCACGCAUGCUACAAUGGCC
(1501)
(1501)    GACGUC--AAAUCAUCAUG-CCCCUUAUG-UC-UUGGGCUUCACGCAUGCUACAAUGGCC
(1501)    GACGUC--AAGUCAUCAUG-GCCCUUACG-AC-CAGGGCUACACACGUGCUACAAUGGCG

(1561)    GGUA-C-AAAGGGU-UGC-GAUACUG-UGAGGUG-----------GAGCUAAUCCCA-AA
(1561)
(1561)    GGUA-C-AAAGGGU-UGC-GAUACUG-UGAGGUG-----------GAGCUAAUCCCA-AA
(1561)    CAUA-C-AAAGAGA-AGC-GACCUCG-CGAGAGC-----------AAGCGGACCUCA-UA

(1621)    AAGCCGGUCUCAGUUCGGAUUGGGGUCUGCAACUCGACCCCAUGAAGUCGGAGUCGCUAG
(1621)
(1621)    AAGCCGGUCUCAGUUCGGAUUGGGGUCUGCAACUCGACCCCAUGAAGUCGGAGUCGCUAG
(1621)    AAGUGCGUCGUAGUCCGGAUUGGAGUCUGCAACUCGACUCCAUGAAGUCGGAAUCGCUAG
```

```
(1681)    UAAUCGCAGAUCAGCAACGCUGCGGUGAAUACGUUCCCGGGCCUUGUACACACCGCCCGU
(1681)
(1681)    UAAUCGCAGAUCAGCAACGCUGCGGUGAAUACGUUCCCGGGCCUUGUACACACCGCCCGU
(1681)    UAAUCGUGGAUCAG-AAUGCCACGGUGAAUACGUUCCCGGGCCUUGUACACACCGCCCGU

(1741)    CAAGUCACGAAAGUUGGUAACACCCGAAG-CCGG-UGG-CCUAA-CCC--CUUGU---GG
(1741)
(1741)    CAAGUCACGAAAGUUGGUAACACCCGAAG-CCGG-UGG-CCUAA-CCC--CUUGU---GG
(1741)    CACACCAUGGGAGUGGGUUGCAAAAGAAG-UAGG-UAG-CUUAA-CC----UUCG----G

(1801)    GAGGGAGCCGU-CGAAGGUGGGACUGGCGAUUGGGACUAAGUCGUAACAAGGUAGCCGUA
(1801)
(1801)    GAGGGAGCCGU-CGAAGGUGGGACUGGCGAUUGGGACUAAGUCGUAACAAGGUAGCCGUA
(1801)    GAGGGCGCUUA-CCACUUUGUGAUUCAUGACUGGGGUGAAGUCGUAACAAGGUAACCGUA

(1861)    CCGGAAGGUGCGGCUGGAUCACCUCCUUUCU--
(1861)
(1861)    CCGGAAGGUGCGGCUGGAUCACCUCCUUUCU--
(1861)    GGGGAACCUGCGGUUGGAUCACCUCCUUA----

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

alignment_row('16S','Arb.globif',''----------UUUC
AACGGAGAGUUUGAUCCUGGCUCAGGAUGAACGCUGGCGGCG-UG-C-UUAACACAUGCAAGUCGAACG-
AUG----------AUCCGGUG--CUUG--CACCGGGG----------AUUAGUG
GCGA--ACGGGUGAGU---AAC-ACGUGAGUAA--CCUGC-CCUUGACUCUGGGAUAAGCCUGGGAAACU
GGGUCUAAUACCGGAU-AUGACUCC--UC---AUC-GCAU-GGU--G-G-GGGGUGGAAA-GC-------
--------UUUUU----------GUG--GUUUUGG-AUGGACUCGCGGCCUAUCA-G-CUUG---
UUGGUG-AGGUAAUGGCUCACCAAGGCGACGACGGGUAGCCGGCCUGAGAGGGU-GACCGGCCACACUGG
GACUGAGACACGGCCCAGACUC-CUACG--GGAGGCAGCA-G-UGGGGAAUAUUGCACAAU-GGGC-GAA
A-GCCUGAUGCAGCGACGCCGCG-UGA-GGGAUGACGGCC--UUCG-GGUUGUAAA----------C
CUCUUUCAGUAGGGAAG--AAG--CG----------AAA----------G---UGACGGUACCU-G-CAG
-AAGAAGCGCC-GGCUAACUACGUGCCAGCAGCCGCGGUAAUACGUAG-GGCGCAAGCGUUAUCCGGAAU
UAUUGGGCGUAAAGAGCUCGUAGGCGGUUU-GUCGCGUCUG-CCGUGAAAGUCCGGGGCUCAACUCCGGA
UC-UGCGGUGGGUACGGGCA-GACUAGAGUGAUGUAGGGGAG-ACUGGAAUUCCU-----GGUGUAGCGG
UGAAAUG-CGCAGAUAUCAGGAGGAACACCGA--UGGCGAAGGCAGGUCUCUGGGCAUUAACUGACGCUG
AGGAGCGAAAGCAUGGG-GAGCGAACAGGAUUAGAUACCCUGGUAGUCCAUGCCGUAAACGUUGGGCA-C
UAGGUGUGGGGGACAU--U-CC---ACGUUUUCCGCGCCGUAGCUAACGCAUUAAGUGCCCCGCCUGGGG
AGUACGGCC-GCAAGGCUAAAACUC-AAA-GGAAUUG-ACGGGGGCCC--GC-A--CAAGCG-GCGGAGC
AUGCGGA-UUAAUUCGAUGCAACGCGAAGAACCUUA-CCAAGGCUUGACAUGGA-CCGG---ACCGC-CG
CA-GAAAUGU-G-GUU--(U-----CUC)--CUUUU-GG----GG-CCGGU-UCA----------
----------CAGGUGGUGCAUGGUUGUCGUCAGCUCGUGUCGUG
AGAUGUUGGGUUAAGUCCCGCAACGAGCGCAACCCUCG-UUCCAUG---UUGCCAGCGC-------GUAA
-------UGGCGGG-GACUCAUGGGAGACUGCCGGGGU-CAA-CUCG--GAGG-A-AGGUGGGG-ACGAC
GUC--AAAUCAUCAUG-CCCCUUAUG-UC-UUGGGCUUCACGCAUGCUACAAUGGCCGGUA-C-AAAGGG
U-UGC-GAUACUG-UGAGGUG----------GAGCUAAUCCCA-AAAAGCCGGUCUCAGUUCGGAUUGG
GGUCUGCAACUCGACCCCAUGAAGUCGGAGUCGCUAGUAAUCGCAGAUCAGCAACGCUGCGGUGAAUACG
UUCCCGGGCCUUGUACACACCGCCCGUCAAGUCACGAAAGUUGGUAACACCCGAAG-CCGG-UGG-CCUA
A-CCC--CUUGU---GGGAGGGAGCCGU-CGAAGGUGGGACUGGCGAUUGGGACUAAGUCGUAACAAGGU
AGCCGUACCGGAAGGUGCGGCUGGAUCACCUCCUUUCU--'').
```

Appendix 5: Partial Insertion Run Results for Three New Species

Three partial outputs from the alignment insertion runs on the Urbana private sequences follow:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Species <private seq> is an aligned urbana seq not in our current
alignment.
Commentary on the correctness of the insertion, broken down by
structural unit, re already present version follows.
(Only structural units where the parse differs from the alignment
version are displayed.)

struc_unit(b66_82): not ok
              new version:       ACG-GCAG-CAC----------------GGAC
    aligned private seq version: ACGG-CAG-CAC----------------GGAC
    # of bases correctly placed: 13
struc_unit(b83_86): not ok
              new version:       --UUC---
    aligned private seq version: --UUCG--
    # of bases correctly placed: 3
struc_unit(b87_103): not ok
              new version:       GUCU----------------G-GUG--GCGAGU
    aligned private seq version: -UCU----------------G-GUG--GCGAGU
    # of bases correctly placed: 13

Number of correctly placed bases: 1533

Total number of bases in species <private seq>: 1535

Percentage of bases correctly placed: 99.87


Number of structural units completely correct: 389

Total number of structural units: 392

Percentage of structural units correctly placed: 99.23

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Species <private seq>  is an aligned urbana seq not in our current
alignment.
Commentary on the correctness of the insertion, broken down by
structural unit, re already present version follows.
(Only structural units where the parse differs from the aligned
version are displayed.)

struc_unit(b66_82): not ok
              new version:       GCGCUG---AAG------------GUUNGUA
    aligned private seq version: GCGCUG-A--------------AGGUUNGUA
    # of bases correctly placed: 13
struc_unit(b87_103): not ok
              new version:       UACCGAC---------------UGG--AUGAGC
    aligned private seq version: UACCGACUGGA---------------U-GAGC
    # of bases correctly placed: 11
struc_unit(b184_186): not ok
              new version:       AACUU--UA---AAC
    aligned private seq version: AACUU--U-A--AAC
```

```
    # of bases correctly placed: 9
struc_unit(b191_193): not ok
              new version:       GUN---NN-AAGUU
    aligned private seq version: GUN--NNA-AGUUU
    # of bases correctly placed: 7
struc_unit(b194_197): not ok
              new version:       UGAAA-
    aligned private seq version: -GAAA-
    # of bases correctly placed: 4
struc_unit(b455_462): not ok
              new version:       UGGUGAGAG
    aligned private seq version: UGGUGAGA-
    # of bases correctly placed: 8
struc_unit(b463_469): not ok
              new version:       -UGGAA-
    aligned private seq version: GUGGAAA
    # of bases correctly placed: 5
struc_unit(b470_477): not ok
              new version:       AGCUNAU-NA
    aligned private seq version: -GCUNAU-NA
    # of bases correctly placed: 8
struc_unit(b841_845): not ok
              new version:       ---AUA---
v    aligned private seq version: --AUA----
    # of bases correctly placed: 0
struc_unit(b999_1003): not ok
              new version:       C-UNGUG
    aligned private seq version: C-UNGU-
    # of bases correctly placed: 5
struc_unit(b1004_1005): not ok
              new version:       --CU
    aligned private seq version: -GCU
    # of bases correctly placed: 2
struc_unit(b1033_1036): not ok
              new version:       GG-----A
    aligned private seq version: GG----AC
    # of bases correctly placed: 2
struc_unit(b1037_1041): not ok
              new version:       CACGGG
    aligned private seq version: -ACGGG
    # of bases correctly placed: 5

Number of correctly placed bases: 1561

Total number of bases in species <private seq>: 1582

Percentage of bases correctly placed: 98.67


Number of structural units completely correct: 379

Total number of structural units: 392

Percentage of structural units correctly placed: 96.68


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Species <private seq>  is an aligned urbana seq not in our current
alignment.
Commentary on the correctness of the insertion, broken down by
structural unit, re already present version follows.
```

```
(Only structural units where the parse differs from the aligned
version are displayed.)

struc_unit(b66_82): not ok
              new version:       (GGG-GCAG-CANG---------------GGNAC)
              new ver stripped : GGG-GCAG-CANG---------------GGNAC
    aligned private seq version: GGG-GCAG-CAN--------------GGGNAC
    new ver and ali with asterisks added to align with each other:
      new ver:GGG-GCAG-CANG---------------GGNAC
          ali:GGG-GCAG-CAN--------------G**GGNAC
    # of bases correctly placed: 15
struc_unit(b87_103): not ok
              new version:       (GUUCU----------------A-GUG--GCGACC)
              new ver stripped : GUUCU----------------A-GUG--GCGACC
    aligned private seq version: GUUCUA---------------GUG--GCGACC
    new ver and ali with asterisks added to align with each other:
      new ver:GUUCU----------------A-GUG--GCGACC
          ali:GUUCUA---------------*GUG--GCGACC
    # of bases correctly placed: 14
struc_unit(b191_193): not ok
              new version:       (NGU--ACUA-UUU)GU
              new ver stripped : NGU--ACUA-UUUGU
    aligned private seq version: NGU--ACUAUUUGU
    new ver and ali with dashes added to align with each other:
      new ver:NGU--ACUA-UUUGU
          ali:NGU--ACUA*UUUGU
    # of bases correctly placed: 12
struc_unit(b455_462): not ok
              new version:       (CUGCU-A---)
              new ver stripped : CUGCU-A---
    aligned private seq version: CUGCUA---
    new ver and ali with asterisks added to align with each other:
      new ver:CUGCU-A---
          ali:CUGCU*A---
    # of bases correctly placed: 6
struc_unit(b1006_1012): not ok
              new version:       (-GU-ACCGA)
              new ver stripped : -GU-ACCGA
    aligned private seq version: GUA-CCGA
    new ver and ali with asterisks added to align with each other:
      new ver:-GU-ACCGA
          ali:*GUA-CCGA
    # of bases correctly placed: 6
struc_unit(b1136_1138): not ok
              new version:       --GUUA---
    aligned private seq version: --GUUAAG-
    # of bases correctly placed: 4
struc_unit(b1139_1144): not ok
              new version:       ----AGCUGGG
    aligned private seq version: ------CUGGG
    # of bases correctly placed: 5
struc_unit(b1174_after_and_before_1175): not ok
              new version:       A-
    aligned private seq version: -A
    # of bases correctly placed: 0
struc_unit(b1435_1449): not ok
              new version:       G-AUGG-------U-G-AC-
    aligned private seq version: G-AUGG-------UGA-CCG
    # of bases correctly placed: 7
struc_unit(b1450_1453): not ok
              new version:       CGUCAAA---
    aligned private seq version: ---UCAA---
    # of bases correctly placed: 2
```

```
struc_unit(b1454_1466): not ok
              new version:       -GGAG----CCGU-U
    aligned private seq version: AGGAG----CCGU-U
    # of bases correctly placed: 9

Number of correctly placed bases: 1519

Total number of bases in species <private seq>: 1532

Percentage of bases correctly placed: 99.15


Number of structural units completely correct: 381

Total number of structural units: 392

Percentage of structural units correctly placed: 97.19
```

Appendix 6: Symbols Used to Flag Anomalies in Insertion Run Results

The symbols shown below were used to surround anomalies (and thus flag them for human attention) in the aligned sequence output where one or more column positions were added relative to the current alignment.

1) [[ ]] - Indicates that the parse failed in a region consisting of one or more structural units and that the number of bases placed in the region of localized parse failure exceeds the number of alignment columns for that region.

2) [ ] - Indicates that the parser placed within a structural unit or a subinterval of a structural unit a number of bases that exceeds the number of alignment columns for that unit or subinterval.

3) ( ) - Indicates that the parser placed within a structural unit or a subinterval of a structural unit a number of bases less than or equal to the number of alignment columns for that unit or subinterval but, due to the insertion of bases in the new species relative to the closest match in the alignment, enough indels were added to the new species so that the number of indels and bases combined exceeds the number of alignment columns for that unit or subinterval. (This is by far the most common of the three types of column addition anomalies.)

Appendix 7: Discussion of Scoring Method for Insertion Run Results

As stated in Section 6, I graded the output of the alignment insertion runs on the number of bases correctly placed. This is an obvious scoring method. Here I state why I chose the exact definition that I did for "correctly placed". That is, I will state why the definition of a base being "correctly placed" is that the correct base type (A,C,G,U,N) shows up in the correct alignment column.

Ideally, I would have liked to make such calculations as follows: I would have gone through the original input base sequence base by base, noting whether the base occupying position such and such in the original sequence showed up in the same alignment column in the insertion output and in the preexisting aligned version. If so, we would have added one to the total of correctly placed bases; if not, zero would have been added. Unfortunately, this simple method cannot be used because of the possible insertion of extra columns in the run output relative to the alignment. For example, suppose an extra column is added in the unit occupying alignment columns 400 through 405 in the alignment, so that this unit now actually takes up columns 400 through 406 in the run output. Then (assuming no indels occupy columns in the next unit), the first base in the following unit will be assigned to column 407 rather than 406, the second base will be assigned to 408 rather than 407, and so on. All positions would be shifted by one, so the program would report back that no bases were correctly inserted after column 405. Now, I could correct for the extra columns if I knew precisely where they were. However, at present our insertion tool reports only the structural unit or subinterval of the unit in which the inserted columns lie. I do not know how many bases within the unit lie on one side of an inserted column, and how many lie on the other. After examining my program, I have concluded that the precise location of the extra columns could indeed be passed to the section of my program that outputs the results, but it would involve, I believe, a rather drastic recoding of the indel insertion section of the parser. Since the current method (described in detail below) works pretty well, and since reporting on the percentage of bases correctly placed is not the primary goal of our tool (no such reporting will be done at all on the truly new species coming in, of course), I have left the program as is. If the need arises in the future, the change can be made to the program.

The current method to implement the definition of "correctly placed" uses calculations that are broken down by structural unit, with the results for all the units added together to get the net score. If (as is usually the case), the run has placed in a unit a grouping of bases and indels that combined equal the number of alignment columns assigned to that unit, then we simply count off the matches when the run output is lined up the previously existing aligned version. Here are five examples from the output for one of the runs in Section 6.1:

```
struc_unit(b191_193): not ok
   new version:       AA--AA-C-ACAUU
   alignment version: AAA----A-CACAU
   # of bases correctly placed: 3

struc_unit(b194_197): not ok
   new version:       -UAAA-
   alignment version: UUAAA-
   # of bases correctly placed: 4
```

```
struc_unit(b1029_1032): not ok
   new version:       GAGGCU--
   alignment version: GAGGCUAA
   # of bases correctly placed: 6

struc_unit(b1033_1036): not ok
   new version:       A-------
   alignment version: --------
   # of bases correctly placed: 0

struc_unit(b1037_1041): not ok
   new version:       ACAGAU
   alignment version: -CAGAU
   # of bases correctly placed: 5
```

However, if (as sometimes happens) the number of indels and bases combined that are placed in a unit or a subinterval of a unit (a subinterval being defined as the alignment columns lying between two pinned bases) exceeds the number of columns in the alignment for that unit or subinterval (that is, if one or more columns are inserted relative to the alignment), then a more complex calculation is done to find the number of bases correctly placed. Suppose, for example, our program put **A-G–A-GA** into a unit, while in the existing aligned sequence that unit contained **A–GA-GA**. One extra column has been added, so the program would place a pair of parentheses around the characters in the output like this to flag the anomaly: **(A-G–A-GA)**. (See Appendix 6 for a more detailed explanation of the use of parentheses.) Lining up the strings for comparison (with the pair of parentheses thrown out), we would have

```
new version:        A-G--A-GA
alignment version:  A--GA-GA
```

If we simply matched on the above, the score would be only one base (the first *A*) correctly placed. However, the misalignment really looks like it is confined to the area around the first *G*. What we do is use the Smith-Waterman algorithm to line the sequences up (bases and indels combined), and only then count the matches. After alignment we have

```
new version:        A-G--A-GA
alignment version:  A--G*A-GA
```

The score now becomes four bases, not one, correctly placed in this unit. I believe this provides a more accurate count of the number of bases correctly positioned. Two examples from the output of the third private species in Appendix 5 are shown below:

```
struc_unit(b87_103): not ok
         new version:       (GUUCU----------------A-GUG--GCGACC)
         new ver stripped : GUUCU----------------A-GUG--GCGACC
aligned private seq version: GUUCUA---------------GUG--GCGACC
new ver and ali with asterisks added to align with each other:
  new ver:GUUCU----------------A-GUG--GCGACC
      ali:GUUCUA---------------*GUG--GCGACC
# of bases correctly placed: 14
```

```
struc_unit(b1006_1012): not ok
          new version:       (-GU-ACCGA)
          new ver stripped : -GU-ACCGA
aligned private seq version: GUA-CCGA
new ver and ali with asterisks added to align with each other:
  new ver:-GU-ACCGA
      ali:*GUA-CCGA
# of bases correctly placed: 6
```

The method is not perfect. In certain cases it can produce an overcount. In the example below from Appendix 4, it would more correct to state that zero bases were correctly placed in unit *b1025_1028* than that two bases were correctly placed.

```
struc_unit(b1025_1028): not ok
    new version:       (U-----CUC)
    new ver stripped : U-----CUC
    alignment version: CU----CC
    new ver and ali with asterisks added to align with each other:
      new ver:U-----CUC
          ali:CU----C*C
    # of bases correctly placed: 2
```

However, having tried to produce counts both with and without alignment of units where columns were inserted relative to the alignment, I believe that use of the Smith-Waterman algorithm to align the two versions brings us substantially closer to the true number of bases correctly positioned.

Appendix 8: Discussion of Cap Motif Use

As stated in the body of this paper, when using a loop_unit in the grammar I place a weight on certain of the cap types, based on their base composition. From a conversation with Dr. Steve Winker on July 16, 1990, and from the results collected in reference [49], it was brought to my attention that particular cap motifs (i.e., tetra-loops, tri-loops, and penta-loops of particular base compositions) were very frequently seen. Thus the idea of using such information naturally came to mind as a means to improve the accuracy of the parse. I bias the parse to choose a configuration with such a cap base composition (since such a configuration appears to be more likely to occur in nature) by adding a certain number of points (a weight) to the Smith-Waterman score for that configuration to get a net score, and then comparing the net scores.

If 1 of the 24 predominant tetra-loop motifs exists in a (prokaryotic) RNA loop region, it is very likely that the bases in the tetra-loop motif (or perhaps the tetra-loop motif + one additional base) will form the cap of the loop. Also, quoting from reference [49]: "in the vast majority of tetra-loops the composition of the tetra-loop is independent of its position in the molecule, and conforms to one of three general motifs: GNRA, UNCG, and (more rarely) CUUG." (The position held by "N" can be filled by using any of the four possible bases.)

Hence we bias the parser toward recognizing such predominant tetra-loop motifs. We also bias the parser toward recognizing penta-loop motifs whose first four bases fit one of the tetra-loop motifs described above. (Although this approach has not yet been tried, it might also be worthwhile to bias the parser toward recognizing tri-loops consisting of *uuu* with a pyrimidine:purine base pair just before the cap.)

Note that tetra-loops are quite common, accounting for the caps of about 55% (i.e., 17) of all stem-loops in the 16S rRNA molecule in *E. coli*, with the next most prevalent (13%) stem-loop cap size being five nucleotides (penta-loops).

The 24 patterns used below to recognize a tetra-loop make up 24/256 or slightly less than 1 in 10 of all possible 4-base patterns. Statistically, if the parser finds one of these predominant motifs, then we are more likely to have found the cap (according to the information encoded in the 16S alignment, from which these tetra-loop motifs are derived).

A tetra-loop cap type is weighted so that it adds three points more than two good bond matches ($39 = 2 * 18 + 3$) to the net score for that configuration. In addition, if the last bond pair before a tetra-loop cap is one of the strongest three bond types (g-c, c-g, u-a, a-u, u-g, g-u) then we add another 18 points to the net score, for a total of 57 points added. The other bond types or mismatches (a-g, g-a, c-a, a-c, u-c, c-u, u-u, g-g, a-a, c-c) add zero points.

A penta-loop cap type is weighted so that it adds one point more than one mediocre bond match ($14 = 1 * 13 + 1$) to the net score for that configuration, if the first four bases in the penta-loop match one of the 24 predominant tetra-loop patterns. (Dr. Winker believes that the other way around is not a good penta-loop motif, that is, that a base followed by a tetra-loop pattern is rare in a cap and has no value in recognizing penta-loops.) With the current weighting scheme, if we have a penta-loop of *uucgg*, for example, then the parser should return that as the cap rather than *ucg* with a u-g bond added to the sides of the loop_unit.

Note: Currently I do not try to incorporate how unknown bases (*n*s) would fit into this scheme.

The encoding of these weights into Prolog clause format is as follows:

```
cap_type("uucg",tetraloop,39).
cap_type("ggaa",tetraloop,39).
cap_type("cgau",tetraloop,39).
cap_type("uuua",tetraloop,39).
cap_type("gcaa",tetraloop,39).
cap_type("auau",tetraloop,39).
cap_type("ucag",tetraloop,39).
cap_type("gaga",tetraloop,39).
cap_type("ggga",tetraloop,39).
cap_type("uacg",tetraloop,39).
cap_type("uccg",tetraloop,39).
cap_type("gaaa",tetraloop,39).
cap_type("ggaa",tetraloop,39).
cap_type("uuag",tetraloop,39).
cap_type("cuug",tetraloop,39).
cap_type("cucg",tetraloop,39).
cap_type("gaag",tetraloop,39).
cap_type("guga",tetraloop,39).
cap_type("gcga",tetraloop,39).
cap_type("guaa",tetraloop,39).
cap_type("gcau",tetraloop,39).
cap_type("acau",tetraloop,39).
cap_type("uaac",tetraloop,39).
cap_type("aaac",tetraloop,39).

cap_type(Bases,pentaloop,14) :-
     length(Bases,5),
     Bases = [Base1,Base2,Base3,Base4,_Base5],
     cap_type([Base1,Base2,Base3,Base4],tetraloop,_).

/* Any cap type other than the above is not weighted (has a weight
   added of zero). */

cap_type(_,nil,0).
```

Appendix 9: Detailed Flow Diagrams for Parser Component

This appendix contains a detailed set of diagrams for the first section of the parser component, that is, for the section that performs the parse and assigns bases to structural units. Some Prolog terminology and symbols will be employed. Where judged useful, the actual names of the Prolog predicates used in the source code of the program are shown.

Diagram 1

Top-Level Flow of Control through the Prolog Predicates
of the Parser Component of the Insertion Tool

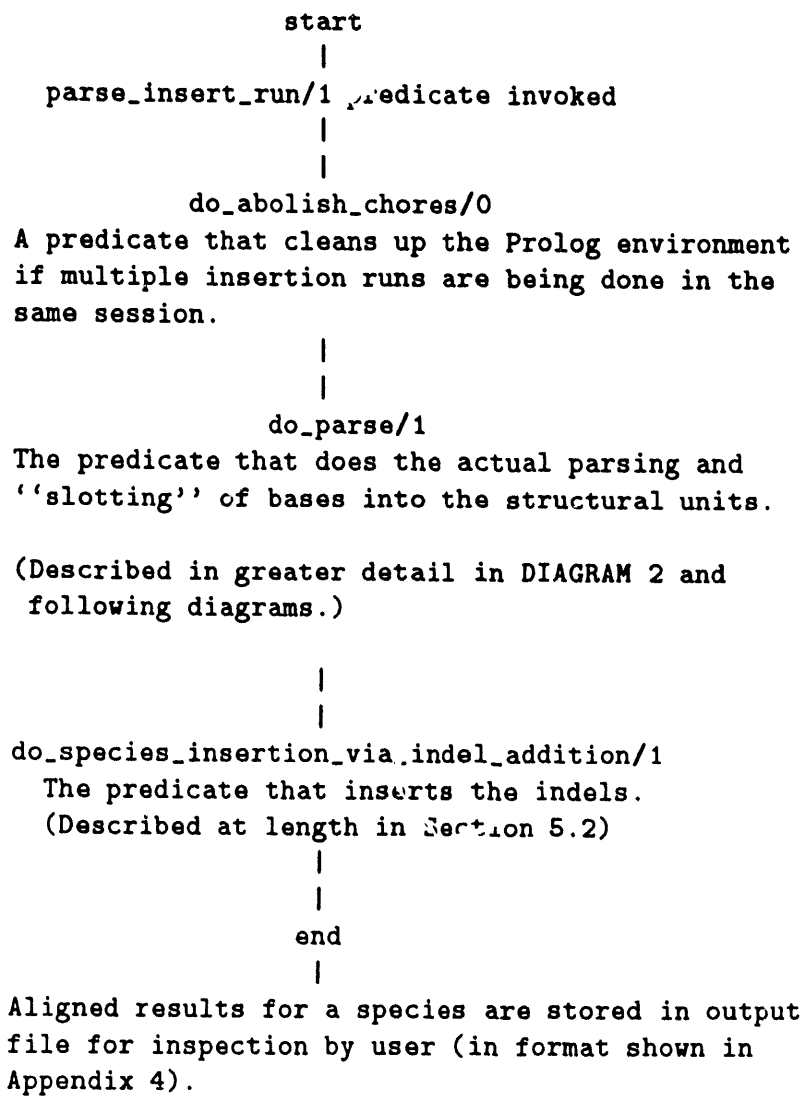
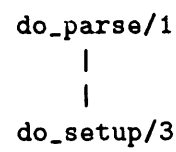
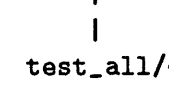


Diagram 2



The do_setup/3 predicate does the following:

1) It reads in the base sequence Original_Base_Seq from a file called new_species.pl. (The sequence for the new species is assumed to have been placed in this file by the user.)
2) It checks on the longest number of consecutive unknown bases in Original_Base_Seq. (The species can be rejected if above a preset number).
3) It reports on any ambiguous and nonstandard characters in Original_Base_Seq.
4) It ''normalizes'' Original_Base_Seq into BaseSeq by converting ambiguous and nonstandard characters into unknown characters (''n''s).
5) It uses the Prolog assert/1 predicate to store the normalized BaseSeq in a clause of this format:
   seq(MoleculeId,SpeciesId,ParseInterval, StartPos,
       BaseSeq).
   In particular, for the 16S molecule we have
       seq('16S',SpeciesId,b1_1542,1,BaseSeq).
   A seq_in_orig_format/5 clause using Original_Base_Seq in place of BaseSeq is also asserted into the database for later use by the
   do_species_insertion_via_indel_addition/1 predicate.



The test_all/4 predicate performs four tasks:

1) It calls setup_pinned_unit/2 to set up a list of the pinned units in PinnedUnits. Each member of PinnedUnits is of the form
   bases(UnitName,BasesInUnit).
2) It calls build_adjacency_info_facts/1 to construct a set of su_with_adjacency_info/10 clauses, one for each structural unit in the grammar. Each clause, among other data, states what the two neighboring units are for a given unit and whether the unit is pinned or not (PinBoolean = ''yes'' or ''no''). The clause format is
   su_with_adjacency_info(UnitName,AlignStart,AlignEnd,
     UnitType,RelatedUnit,PinList,PinBoolean,PinnedBases,
     PreviousUnit,NextUnit).

3) It calls parse_linear/10 to perform the parse.

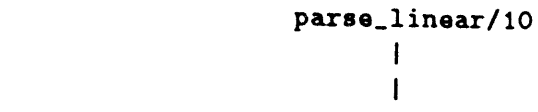
(See DIAGRAM 3 and following diagrams.)

Output from parse_linear/10 is contained in one parameter, a list called ParseOut. There is one entry in ParseOut for each structural unit that parsed and one for each unpinned unit group that failed to parse. Each entry contains the (normalized) bases slotted into that unit or region of localized parse failure, along with the start and end positions of that set of bases in the original incoming sequence.

4) It calls build_and_write_out_structural_prolog_clauses/3 to write out a set of su/11 clauses to a holding file using the data returned by parse_linear/10 in ParseOut. The holding file is later read in by the do_species_insertion_via_indel_addition/1 predicate. There is one su/11 clause per parsed structural unit and one per unparsed unpinned unit group (localized parse failure). The format of each su/11 clause looks like this:

```
su(MoleculeId,SpeciesId,EcoliUnitName,
   FamilyUsedToParse,SpeciesStartPos,SpeciesEndPos,
   UnitType,Bases,AlignStartOfUnit,AlignEndOfUnit,
   UnitPinList).
```

Diagram 3



sequence_parsed_by_family_set/6

The one sequence_parsed_by_family_set/6 clause present in the grammar (constructed by the automatic grammar generation program) is accessed to obtain the StrucUnitList. The list contains the names of all the structural units that must be processed, and in proper order for processing. The format of the list looks like this:
[b1_8, b9_13, b14_16, b17_19, b20, b21_25, ...].
We shall walk through this list, parsing the units one by one.

initialization is performed on input parameters to parse_by_units/7:

1) StartingDict = [ ] (Dict is a list that stores the name of each parsed lhs and the bases it contains for later use when the associated rhs is encountered. The Dict is set to the empty list at the start.)
2) StartingSeqPos = 1 (We keep a running track of where we are in the original sequence as we parse it. The start position is the first base in the sequence.)
3) P1 = BaseSeq (We start the parse using the entire sequence as input in a variable called P1.)

The lists PinnedUnits and StrucUnitList constructed above are also used as input parameters to parse_by_units/7.

parse_by_units/7

This predicate performs the parse unit-by-unit, returning the results in the ParseOut list, which is passed back up through parse_linear/10 to test_all/4. When called from within the parse_linear/10 predicate it looks like this:

```
parse_by_units(StrucUnitList,BaseSeq,PinnedUnits,
  StartingDict,StartingSeqPos,P1,ParseOut).
```

Note that P1 and BaseSeq are equivalent at this point. P1 will lose bases from its head as each pinned unit or unpinned unit group is processed. BaseSeq will

stay the same throughout the parse, keeping the entire original sequence available if needed. (Currently no use is made of the BaseSeq parameter after this point, but that might change in the future.)

(See DIAGRAM 4 and following diagrams.)

Diagram 4

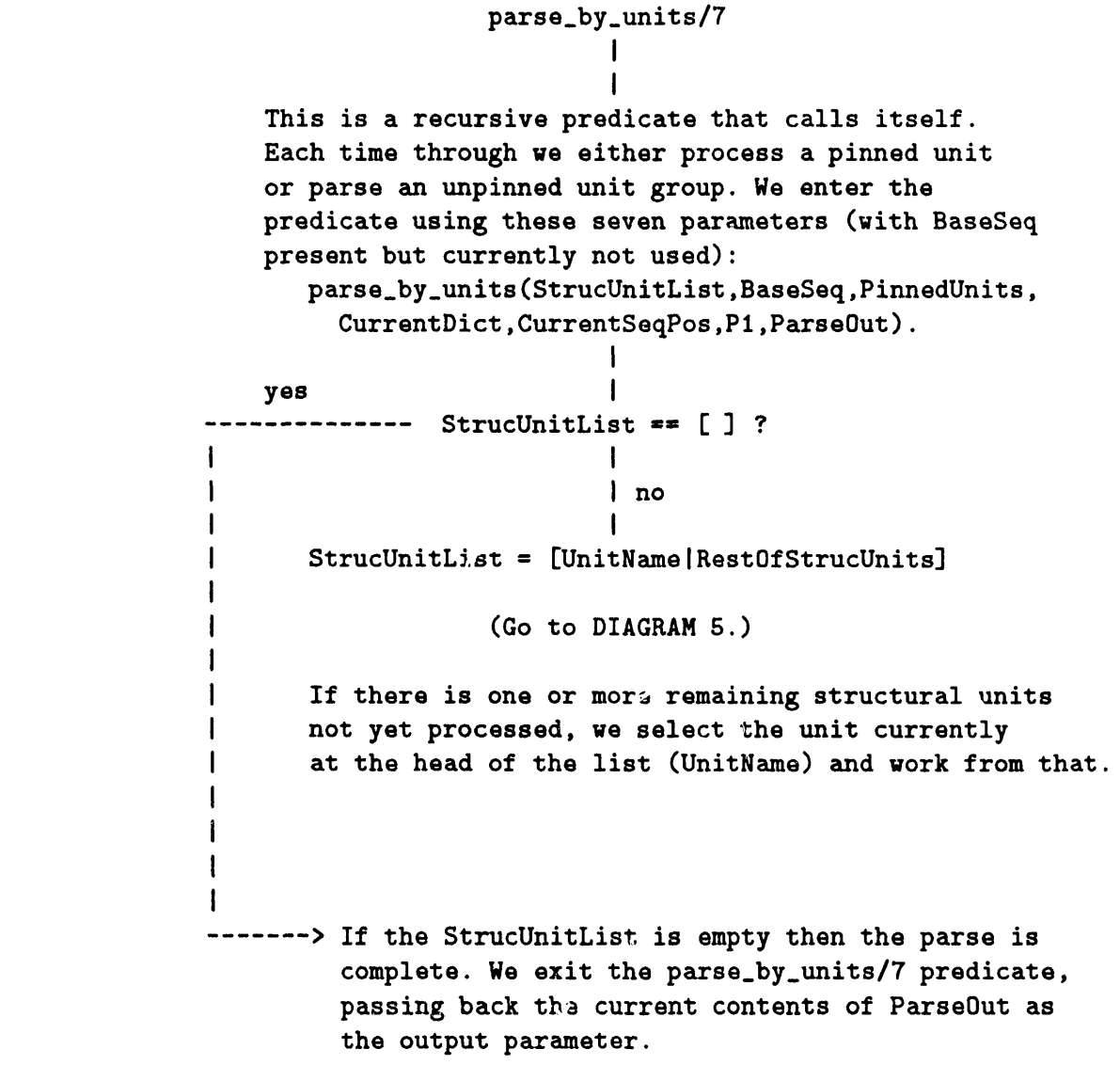
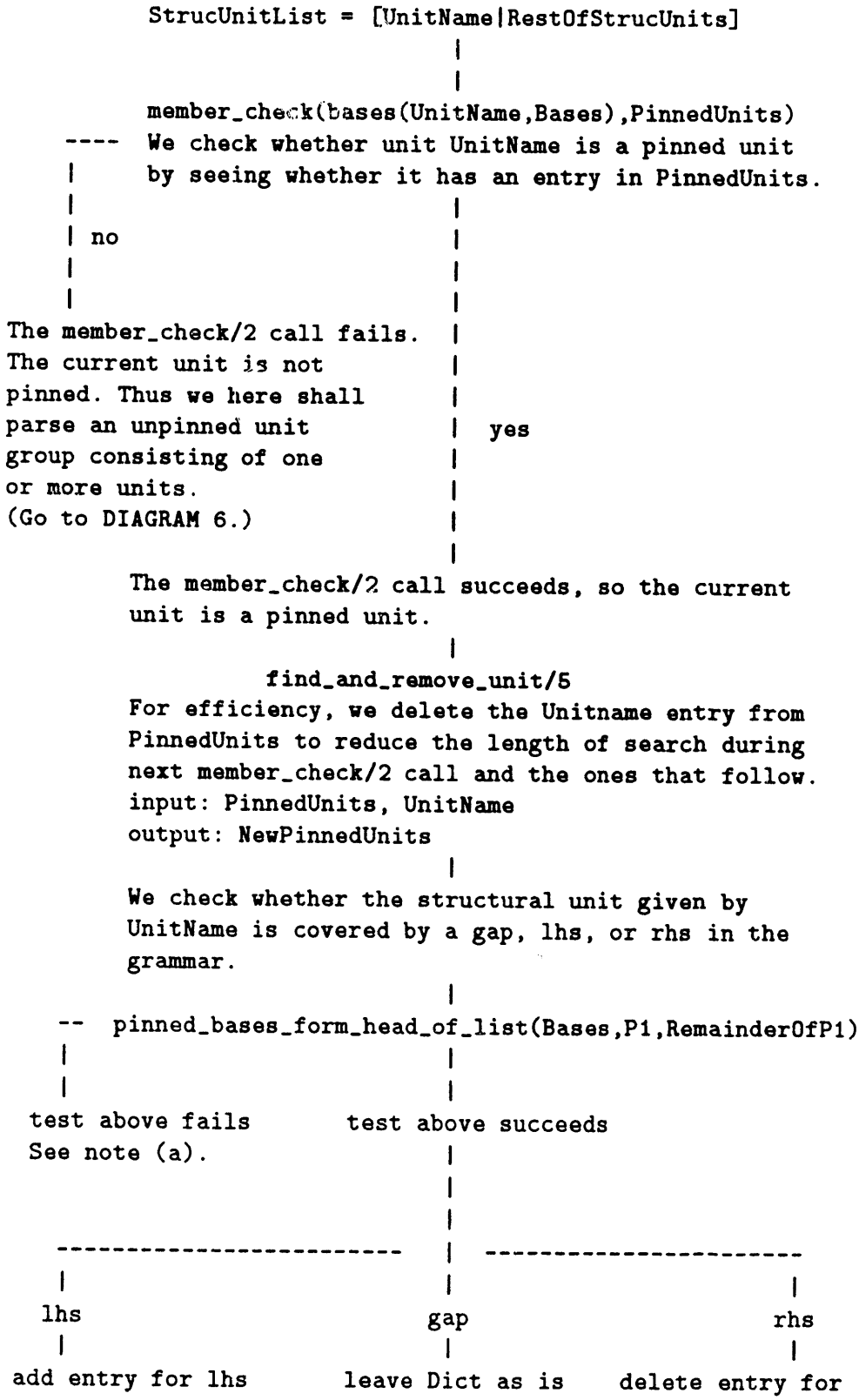
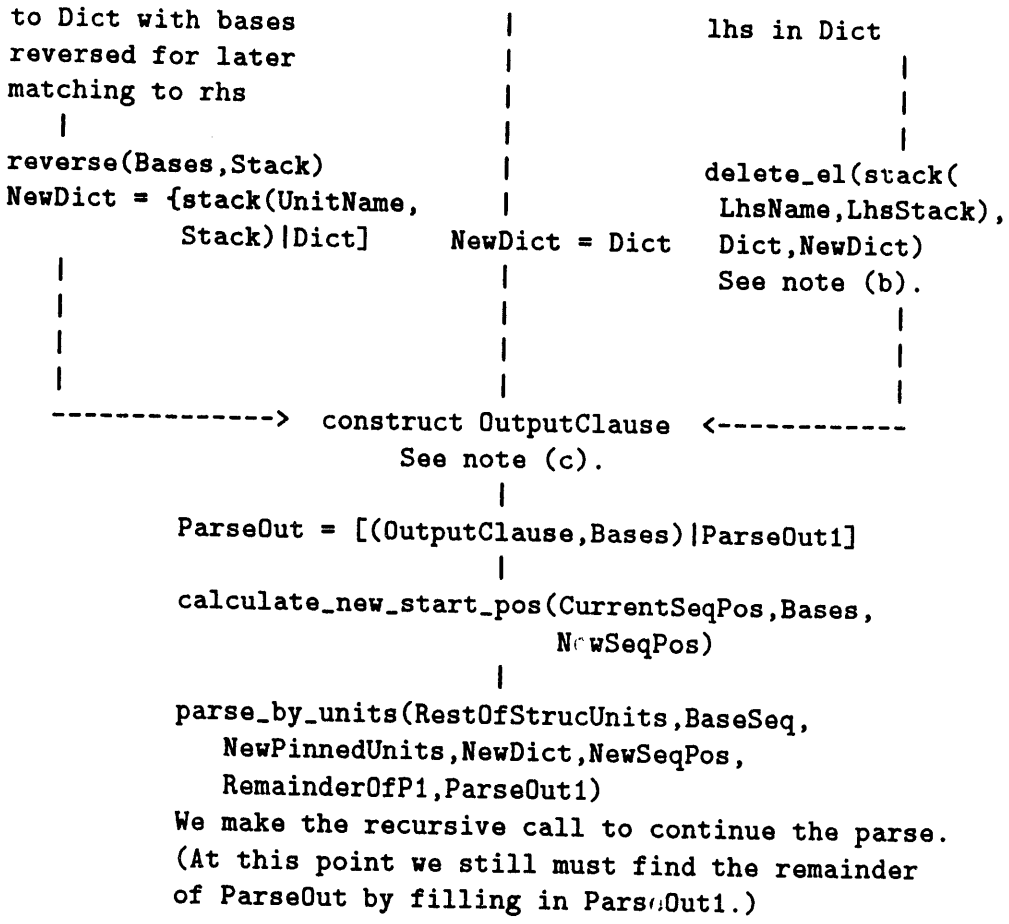

parse_by_units/7
This is a recursive predicate that calls itself.
Each time through we either process a pinned unit
or parse an unpinned unit group. We enter the
predicate using these seven parameters (with BaseSeq
present but currently not used):
parse_by_units(StrucUnitList,BaseSeq,PinnedUnits,
CurrentDict,CurrentSeqPos,P1,ParseOut).
yes
StrucUnitList == [ ] ?
no
StrucUnitList = [UnitName|RestOfStrucUnits]
(Go to DIAGRAM 5.)
If there is one or more remaining structural units
not yet processed, we select the unit currently
at the head of the list (UnitName) and work from that.
If the StrucUnitList is empty then the parse is
complete. We exit the parse_by_units/7 predicate,
passing back the current contents of ParseOut as
the output parameter.

Diagram 5


StrucUnitList = [UnitName|RestOfStrucUnits]
member_check(bases(UnitName,Bases),PinnedUnits)
We check whether unit UnitName is a pinned unit
by seeing whether it has an entry in PinnedUnits.
no
The member_check/2 call fails.
The current unit is not
pinned. Thus we here shall
parse an unpinned unit
group consisting of one
or more units.
(Go to DIAGRAM 6.)
yes
The member_check/2 call succeeds, so the current
unit is a pinned unit.
find_and_remove_unit/5
For efficiency, we delete the Unitname entry from
PinnedUnits to reduce the length of search during
next member_check/2 call and the ones that follow.
input: PinnedUnits, UnitName
output: NewPinnedUnits
We check whether the structural unit given by
UnitName is covered by a gap, lhs, or rhs in the
grammar.
pinned_bases_form_head_of_list(Bases,P1,RemainderOfP1)
test above fails
See note (a).
test above succeeds
lhs
gap
rhs
add entry for lhs
leave Dict as is
delete entry for



Note (a): If the pinned_bases_form_head_of_list(Bases, P1, RemainderOfP1) call fails, then a situation like the following must have occurred: Suppose that the input string for the unit before this one was "AGGCUUUGG ...". Suppose that unit was pinned, with the pins telling the parser to place "AGG" in the unit. The input string to the current (second) pinned unit then becomes "CUUUGG ...". If the pinning component found pins that told it to place "UUUGG", not "CUUUGG", in this pinned unit, then we have a problem: there is an extra "C" wedged between two pinned units and not assigned to either. Hence the pinned bases do not form the head of the current input string. This is an extremely rare occurrence. (It has happened exactly once in all the trials run so far.) However, we must allow for the possibility. In such a case the extra base(s) is inserted as the first base in the current pinned unit being processed, that is, in the second of the two units that it falls between. A message is also issued to the user notifying him or her of this situation. (Note that such a base would fall between the two adjacent columns in the current alignment occupied by the pinned base at the end of the first unit and the next pinned base at the start of the second unit.)

Note (b): If the unpinned group containing the lhs failed to parse, then no Dict entry for the lhs would be created, and the delete_el/2 call would fail. In such a case we simply set NewDict = Dict and LhsStack = [ ].

note (c): If the unit is a gap, then we have

OutputClause = gap(UnitName, pinned, no_msg, CurrentSeqPos).

If the unit is an lhs, then we have

OutputClause = gap(UnitName, pinned, no_msg, CurrentSeqPos).

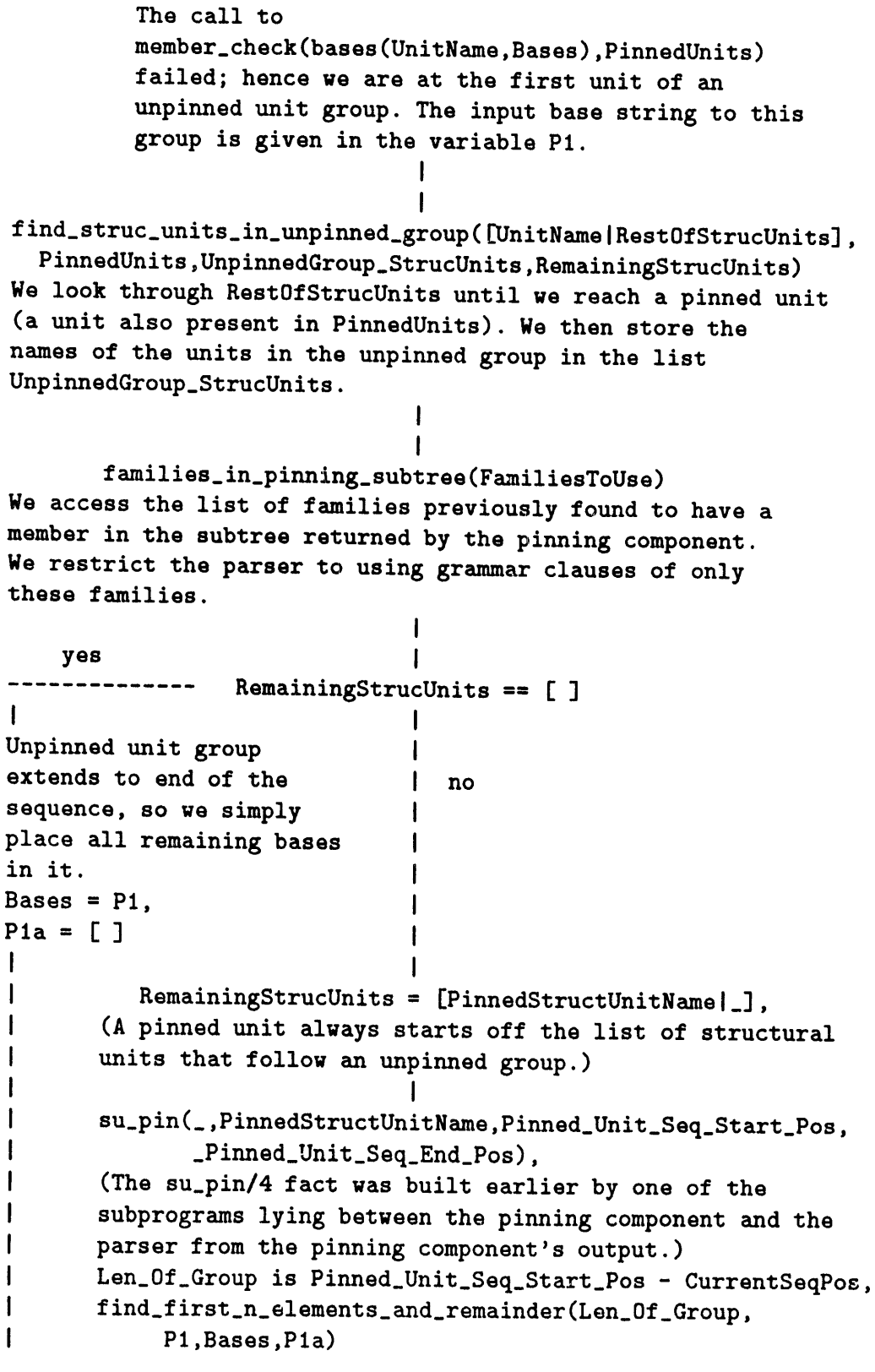
If the unit is an rhs, then we have

OutputClause = rhs(UnitName, pinned, no_msg, 0, 0, 0, LhsName, LhsStack, CurrentSeqPos).

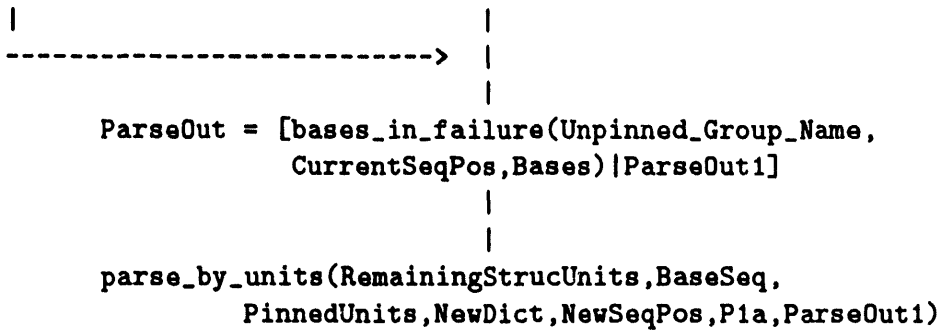
(The number of mismatches, deletions, and insertions used in a pinned rhs is set to zero. Those three arguments have true values placed in them only when an rhs is parsed in an unpinned unit group.)

Note that the third argument, which can carry messages, is not currently used and is filled in simply with "no_msg".

Diagram 6


The call to
member_check(bases(UnitName,Bases),PinnedUnits)
failed; hence we are at the first unit of an
unpinned unit group. The input base string to this
group is given in the variable P1.
find_struc_units_in_unpinned_group([UnitName|RestOfStrucUnits],
PinnedUnits,UnpinnedGroup_StrucUnits,RemainingStrucUnits)
We look through RestOfStrucUnits until we reach a pinned unit
(a unit also present in PinnedUnits). We then store the
names of the units in the unpinned group in the list
UnpinnedGroup_StrucUnits.
families_in_pinning_subtree(FamiliesToUse)
We access the list of families previously found to have a
member in the subtree returned by the pinning component.
We restrict the parser to using grammar clauses of only
these families.
yes
RemainingStrucUnits == [ ]
no
Unpinned unit group
extends to end of the
sequence, so we simply
place all remaining bases
in it.
Bases = P1,
P1a = [ ]
RemainingStrucUnits = [PinnedStructUnitName|_],
(A pinned unit always starts off the list of structural
units that follow an unpinned group.)
su_pin(_,PinnedStructUnitName,Pinned_Unit_Seq_Start_Pos,
_Pinned_Unit_Seq_End_Pos),
(The su_pin/4 fact was built earlier by one of the
subprograms lying between the pinning component and the
parser from the pinning component's output.)
Len_Of_Group is Pinned_Unit_Seq_Start_Pos - CurrentSeqPos,
find_first_n_elements_and_remainder(Len_Of_Group,
P1,Bases,P1a)

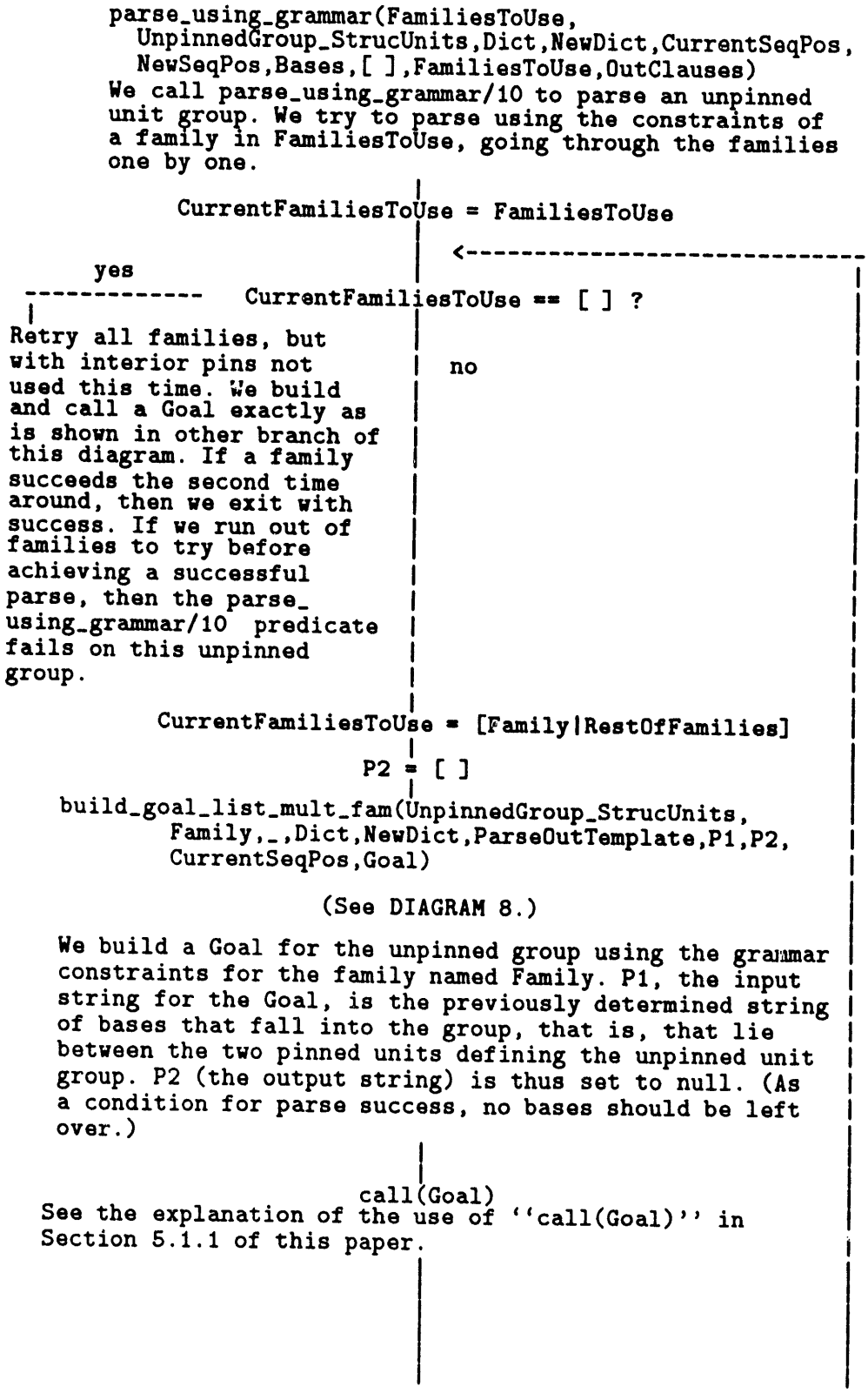
```
   |      The string returned in Bases is made up of the bases that
   |      must occupy that part of the molecule defined by the
   |      unpinned unit group. Note that input string P1 is equal
   |      to the combined string made by appending P1a to Bases.
   |                                |
   |--------------------------->    |
                                    |
          parse_using_grammar(FamiliesToUse,
            UnpinnedGroup_StrucUnits,Dict,NewDict,CurrentSeqPos,
            NewSeqPos,Bases,[ ],FamiliesToUse,OutClauses)
          We call parse_using_grammar/10 to parse the unpinned
          unit group.

                          (See DIAGRAM 7.)

                                    |
                                    |
       yes                          |
   -----------   does parse_using_grammar/10 succeed ?
   |                                |
   |                                |  no
append(OutClauses,                  |
    ParseOut1,ParseOut)             |
(This is the usual route;           |
the great majority of               |
unpinned groups will parse.)        |
   |                                |
   |                          See note (a).
   |                                |
parse_by_units(                     |
  RemainingStrucUnits,BaseSeq,      |
  PinnedUnits,NewDict,              |
  NewSeqPos,P1a,ParseOut1).         |
                                    |
                                    |
       yes                          |
   -------------  RemainingStrucUnits == [ ] ?
   |                                |
NewDict = Dict                      |
length(Bases,Base_Len)              |  no
NewSeqPos is CurrentSeqPos          |
             + Base_Len             |
   |                                |
   |                      NewDict = Dict
   |              NewSeqPos is CurrentSeqPos + Len_Of_Group
```



Note (a): The build_and_write_out_structural_prolog_clauses/3 predicate that will later write out an su/11 fact for the unit named by Unpinned_Group_Name will need a struc_unit/5 clause for the unit. (This unit has its UnitType argument set to "bases_in_failure".) We create a dynamically_asserted_struc_unit/5 clause here and insert it into our database using the Prolog *assert/1* predicate if the unpinned unit group fails to parse.

The build_and_write_out_structural _prolog_clauses/3 predicate was modified to check for a dynamically_asserted_struc_unit/5 fact if a struc_unit/5 fact did not exist. (The struc_unit/5 clauses are currently "static", so under Quintus Prolog we cannot use assert/1 to add clauses with that same name into the database.)

Diagram 7


parse_using_grammar(FamiliesToUse,
UnpinnedGroup_StrucUnits,Dict,NewDict,CurrentSeqPos,
NewSeqPos,Bases,[ ],FamiliesToUse,OutClauses)
We call parse_using_grammar/10 to parse an unpinned unit group. We try to parse using the constraints of a family in FamiliesToUse, going through the families one by one.
CurrentFamiliesToUse = FamiliesToUse
yes
CurrentFamiliesToUse == [ ] ?
Retry all families, but with interior pins not used this time. We build and call a Goal exactly as is shown in other branch of this diagram. If a family succeeds the second time around, then we exit with success. If we run out of families to try before achieving a successful parse, then the parse_using_grammar/10 predicate fails on this unpinned group.
no
CurrentFamiliesToUse = [Family|RestOfFamilies]
P2 = [ ]
build_goal_list_mult_fam(UnpinnedGroup_StrucUnits,
Family,_,Dict,NewDict,ParseOutTemplate,P1,P2,
CurrentSeqPos,Goal)
(See DIAGRAM 8.)
We build a Goal for the unpinned group using the grammar constraints for the family named Family. P1, the input string for the Goal, is the previously determined string of bases that fall into the group, that is, that lie between the two pinned units defining the unpinned unit group. P2 (the output string) is thus set to null. (As a condition for parse success, no bases should be left over.)
call(Goal)
See the explanation of the use of ''call(Goal)'' in Section 5.1.1 of this paper.

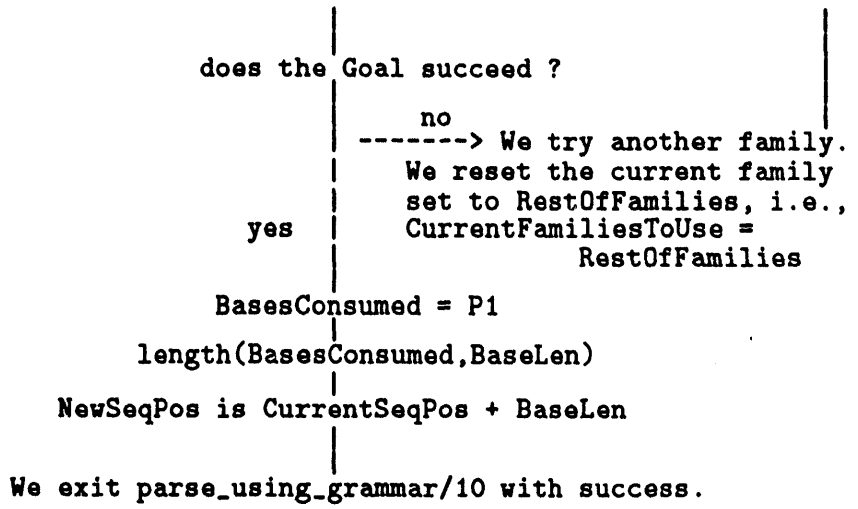

does the Goal succeed ?
no
We try another family.
We reset the current family
set to RestOfFamilies, i.e.,
CurrentFamiliesToUse =
RestOfFamilies
yes
BasesConsumed = P1
length(BasesConsumed,BaseLen)
NewSeqPos is CurrentSeqPos + BaseLen
We exit parse_using_grammar/10 with success.

Diagram 8

```
build_goal_list_mult_fam(UnpinnedGroup_StrucUnits,
        Family,_,Dict,NewDict,ParseOutTemplateSet,P1,P2,
        StartPos,GoalSet)
The Goal described in Diagram 7 is really a set of
(sub)Goals, one for each unit in the unpinned group.
Hence we refer here to GoalSet, not Goal, for clarity. We
build a GoalSet for the unpinned group using the grammar
constraints for the family named Family. P1, the input
string for the GoalSet, is the the previously determined
string of bases that fall into the group, i.e., that lie
between the two pinned units defining the unpinned unit
group. P2 (the output string) is thus set to null. (As
a condition for parse success within the unpinned group,
no bases should be left over).
                              |
    yes                       |
<----------   UnpinnedGroup_StrucUnits == [ ] ?
 |                            |
We have run out of units.     |
Hence we construct one        |
last Goal here (testing       |
whether the remaining         |
bases equal the empty         |  no
list) and then exit.          |
 |                            |
NewDict = Dict                |
ParseOutTemplateSet = [ ]     |
GoalSet = (P1=P2)             |
 |                            |
exit the predicate            |
                              |
                              |
                              |
      UnpinnedGroup_StrucUnits = [UnitName|RemainingUnits]
                              |
      build_single_goal_mult_fam(UnitName,RemainingUnits,
           Family,_,ParseOutTemplate,P1,P1a,Goal,StartPos,
           NewStartPos,Dict,Dict1,_)
```

(See DIAGRAM 9.)

Each call to build_single_goal_mult_fam/13 constructs a Goal and a ParseOutTemplate (to report back the results) for the unit given by UnitName. Goal and ParseOutTemplate contain certain variables that share the same name. When

the (sub)Goal is executed as part of the GoalSet in ''call(GoalSet)'', each placeholder variable in ParseOutTemplate that has the same name as in Goal will be filled in (instantiated) with the value that satisfied the Goal, that is, that allowed the Goal to succeed.

Note that a single goal takes P1, StartPos, and Dict as input parameters and returns altered values in P1a, NewStartPos, and Dict1. These three altered values are then passed on to the rest of the goals to be constructed in the recursive call to build_goal_list_mult_fam/10 below.

The difference between the strings P1 and P1a (P1/P1a) represents the bases slotted or ''consumed'' by the goal for the structural unit UnitName. P1 is the input base string and P1a is the shortened output base string to be used as input to the remaining goals.

```
NewStartPos = StartPos + (the number of bases by
                          which P1 and P1a differ)
```

Dict1 differs from Dict if the goal is for a lhs or a rhs unit. In such cases, an entry is added or deleted from the Dict, respectively.

```
                          |
                          |
            GoalSet = [Goal|RemainingGoals]
      ParseOutTemplateSet = [ParseOutTemplate|
                             RemaiingParseOutTemplates]
                          |
                          |
   build_goal_list_mult_fam(RemainingUnits,
       Family,_,Dict1,NewDict,RemainingParseOutTemplates,
       P1a,P2,NewStartPos,RemainingGoals)
```

We recursively call this predicate to fill in a Goal and ParseOutTemplate for each unit.

Diagram 9

```
build_single_goal_mult_fam(UnitName,RemainingUnits,
      Family,_,ParseOutTemplate,P1,P1a,Goal,StartPos,
      NewStartPos,Dict,NewDict,_)
                          |
                          |
What is the UnitType of the unit UnitName? We try
accessing gap/5, lhs/5, rhs/7, and loop_unit/10 grammar
clauses for UnitName and Family. If access succeeds to
a clause of a given type, then we automatically have
our answer.
                          |
                          |
If UnitType = gap, then we set
     ParseOutTemplate = (gap(UnitName,Family,
                           Msg,StartPos),Bases)
     Goal = process_gap_mult_fam(UnitName,Family,Msg,
                    StartPos,NewStartPos,Bases,P1,P1a)
     NewDict = Dict
(Note: Msg is currently an argument that is not used
in any of these UnitTypes. It is nulled out to
''no_msg''.)
                          |
                          |
If UnitType = lhs, then we set
     ParseOutTemplate = (lhs(UnitName,Family,
                           Msg,StartPos),Bases)
     Goal = process_lhs_mult_fam(UnitName,Family,Min,
         Max,Stack,StartPos,NewStartPos,Bases,P1,P1a)
     NewDict = [stack(Name,Stack)|Dict]
(The values for the Min and Max bases allowed in the
 unit come from the lhs/5 clause in the grammar for
 UnitName and Family.)
                          |
                          |
If UnitType = rhs, then we set
     ParseOutTemplate = (rhs(UnitName,Family,Msg,
         MismatchesUsed,InsertionsUsed,DeletionsUsed,
         LhsName,LhsStack,StartPos),Bases))
     Goal = process_rhs_mult_fam(UnitName,Family,
      MismatchesAllowed,InsertionsAllowed,DeletionsAllowed,
      MismatchesUsed,InsertionsUsed,DeletionsUsed,LhsName,
      LhsStack,Dict,NewDict,StartPos,NewStartPos,
      Bases,P1,P1a),
(The values for MismatchesAllowed, InsertionsAllowed,
```

```
 and DeletionsAllowed come from the rhs/7 clause in the
 grammar for UnitName and Family.)
                          |
                          |
If UnitType = loop_unit, then we set
     ParseOutTemplate =
          loop_unit(UnitName,Family,Msg,
            WiggleFactor,Variance,
            Lhs_interval_name,Cap_interval_name,
            Rhs_interval_name,
            SimilarityScore,NetScore,BasesInOptimalLhs,
            BasesInOptimalCap,CapSubStructureList,
            BasesInOptimalRhs,
            Lhs_Rhs_Correspondence_Found,StartPos)),
     Goal =
          process_loop_unit(UnitName,Family,
            Lhs_interval_name,Cap_interval_name,
            Rhs_interval_name,
            LhsPin,CapPin,RhsPin,
            (Cap_interval_type,ListofCapSubIntervals),
            WiggleFactor,Variance,
            ListofStructuralUnitsForLocalLookAhead,
            SimilarityScore,NetScore,BasesInOptimalLhs,
            BasesInOptimalCap,CapSubStructureList,
            BasesInOptimalRhs,Lhs_Rhs_Correspondence_Found,
            StartPos,NewStartPos,P1,P1a)
     NewDict = Dict
(The ListofStructuralUnitsForLocalLookAhead was found by
walking forward from the loop_unit to the nearest pinned
unit. The values for Lhs_interval_name, Cap_interval_name,
Rhs_interval_name, WiggleFactor, and Variance were found
by checking the loop_unit clause in the grammar for
UnitName and Family.)
                          |
                          |
end of possibilities for UnitType; the predicate will
always set up variables for one of the four unit types
above and then terminate with success.
```

The four *process* predicates which can be set up as the Goal in the above diagram (process_gap_mult_fam/8, process_lhs_mult_fam/10, process_rhs_mult_fam/17, and process_loop_unit/23) have been designed to use the grammar constraints on both the primary and secondary structure. The work done by the first three of these predicates has been described at length in Section 5.1.1 of this paper. The tasks that the process_loop_unit/23 predicate has been built to handle have been stated in great detail in Section 3.1.

Use of interior pins has been designed into these four predicates, with a global flag that can be set in the database to turn interior pin use on or off.

A form of *chart* parsing (intermediate storage of results) has also been implemented in each of the *process* predicates. When the parse succeeds in a structural unit, the base string for that unit and the remaining string are combined into one string and then asserted into the database in a *struc_unit_config/2* clause. Then, if the parse later fails and we backtrack to the unit, we check to see whether a configuration has already been tried. If so, we skip it. Note that just storing the bases placed in the unit in the struc_unit_config/2 clause is not good enough to implement chart parsing. For example, suppose that the input string to some unit, say, *b404_409*, is *acuuuacugg*. Also suppose that we succeed in parsing unit *b404_409* with the *ac* at the head of the input string and thus store *ac* in a *struc_unit_config(ac, b404_409)* clause as the successful configuration that should not be retried. Now suppose that someplace further on the parse fails and we have to backtrack to *b404_409*. The substring *ac* occurs twice in the input string, but we shall never be able to try the second *ac* as a different route to success, even though such a route will pass on a different string to the rest of the parse and possibly allow the parse to overcome its later failure. The check against the struc_unit_config/2 clause, where *ac* would have been found to already have been tried, would prevent this. But if we uniquely identify the first trial as *ac_uuuacugg* and the second as *ac_ugg* then we shall be able to try both possibilities. Hence we concatenate the bases placed in the unit with the bases passed on as the input string to the rest of the parse, separated by an underscore, and place the combined string into a struc_unit_config/2 clause, like so: *struc_unit_config(ac_uuuacugg, b404_409)*. (Further discussion of chart parsing and the reason for its use can be found in Section 5.1.2.)

As a example of the *process* predicates, I show below the process_gap_mult_fam/10 predicate, the simplest of the four. (Unlike the other three, it performs a check solely against the primary structure constraints for a structural unit. All other *process* predicates check against both the grammar constraints on the unit's base composition and the grammar constraints on the secondary structure of the unit.) This is the complete Prolog predicate definition as it currently stands in the parser program. The input string to the unit is passed in by means of the P1 variable. The output string is returned in the P2 variable after being filled in by the predicate. The difference between the input string and the output string is the set of bases slotted into this particular unit. The first test that the predicate does is to check on whether the primary structure constraints can be fulfilled using some substring from the head of the input string. This is done by a direct call to the appropriate constraint/5 clause in the grammar. If this subgoal succeeds, then a second subgoal is tried (in interior_pins_agree_with_parse/3) to see if the interior pins test can be satisfied using the substring defined by the call to constraint/5. (The interior_pins_agree_with_parse/3 predicate checks on a global flag to see whether it should actually perform its test or simply always succeed, no matter what input it is given. The second option is used when we want to skip this subgoal.) Note that, if the interior_pins_agree_with_parse/3 subgoal fails, then we backtrack to the constraint/5 subgoal to find out whether a different substring can be found that will obey the base constraints. If so, then we can retry the interior pins test and still possibly succeed in the parse of this unit. This process of automatic backtracking upon failure should be kept firmly in mind whenever one examines any section of Prolog code where failures can occur.

```
process_gap_mult_fam(UnitName,FamilyUsed,
                     MsgUsed,StartPos,NewStartPos,
                     Bases,P1,P2) :-
        constraint(UnitName,FamilyUsed,MsgUsed,P1,P2),
        convert_dif_list_to_list(P1/P2,Bases),
        calculate_new_start_pos(StartPos,Bases,NewStartPos),
        interior_pins_agree_with_parse(UnitName,StartPos,NewStartPos),
        append(Bases,"_",TempBaseList),
        append(TempBaseList,P2,Unique_Trial_Bases),
        atom_chars(Unique_Trial_Id,Unique_Trial_Bases),
        ( struc_unit_config(Unique_Trial_Id,UnitName) ->
            % This config has been tried before, so there is no
            % point in trying it again and thus failing again later
            % on. Hence we force failure here.
            fail
        ;
            do_assert(struc_unit_config(Unique_Trial_Id,UnitName))
        ).
```

Distribution for ANL-91/29

Internal:

E. A. Baehr
J. M. Beumer (25)
F. Y. Fradin
H. G. Kaper
R. A. Overbeek
G. W. Pieper
D. P. Weber
C. L. Wilkinson
S. K. Winker
ANL Patent Department
ANL Contract File
TIS Files (3)

External:

DOE-OSTI, for distribution per UC-405 (58)
ANL Libraries
Manager, Chicago Operations Office, DOE
Mathematics and Computer Science Division Review Committee:
W. W. Bledsoe, The University of Texas, Austin
P. Concus, Lawrence Berkeley Laboratory
E. F. Infante, University of Minnesota
M. J. O'Donnell, University of Chicago
D. O'Leary, University of Maryland
R. E. O'Malley, Rensselaer Polytechnic Institute
M. H. Schultz, Yale University
J. Cavallini, Department of Energy - Energy Research
F. Howes, Department of Energy - Energy Research
G. Olsen, University of Illinois, Urbana
R. Taylor, National Institutes of Health (30)
C. Woese, University of Illinois, Urbana

END

DATE FILMED

2/03/92