ANL-92/1

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

An Integrated Database to Support Research on *Escherichia coli*

by Alexandra Baehr, George Dunham,
Adam Ginsburg, Ray Hagstrom,
David Joerg, Toni Kazic, Hideo Matsuda,
George Michaels, Ross Overbeek,
Kenneth E. Rudd, Cassandra Smith,
Ron Taylor, Kaoru Yoshida,
and Dave Zawada

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

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

Argonne National Laboratory, with facilities in the states of Illinois and Idaho, is owned by the United States government, and operated by The University of Chicago under the provisions of a contract with the Department of Energy.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

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)

ANL-92/1

ANL--92/1

DE92 007982

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

An Integrated Database to Support Research on *Escherichia coli*

by

*Alexandra Baehr, George Dunham (1), Adam Ginsburg (2),
Ray Hagstrom (3), David Joerg, Toni Kazic (2),
Hideo Matsuda, George Michaels (1), Ross Overbeek,
Kenneth E. Rudd (4), Cassandra Smith (5), Ron Taylor (1),
Kaoru Yoshida (5), Dave Zawada (6)*

Mathematics and Computer Science Division

January 1992

(1) Division of Computer Research and Technology, National Institutes of Health, Bethesda, Md.
(2) Department of Genetics, Washington University, St. Louis, Mo.
(3) High Energy Physics Division, ANL
(4) National Center for Biotechnology Information, National Institutes of Health, Bethesda, Md.
(5) Department for Molecular and Cellular Biology, University of California and Lawrence Berkeley Laboratory, Berkeley, Calif.
(6) Environmental Assessment and Information Sciences Division, ANL

This work was supported in part by the Office of Health and Environmental Research and in part by the Office of Energy Research, U.S. Department of Energy, under Contract W-31-100-Eng-38.

Contents

An Integrated Database to Support Research on *Escherichia coli*

Alexandra Baehr, George Dunham, Adam Ginsburg,
Ray Hagstrom, David Joerg, Toni Kazic,
Hideo Matsuda, George Michaels, Ross Overbeek,
Kenneth E. Rudd, Cassandra Smith, Ron Taylor,
Kaoru Yoshida, Dave Zawada

Abstract

We have used logic programming to design and implement a prototype database of genomic information for the model bacterial organism *Escherichia coli*. This report presents the fundamental database primitives that can be used to access and manipulate data relating to the *E. coli* genome. The present system, combined with a tutorial manual, provides immediate access to the integrated knowledge base for *E. coli* chromosome data. It also serves as the foundation for development of more user-friendly interfaces that have the same retrieval power and high-level tools to analyze complex chromosome organization.

1 Introduction

Two recent advances in biotechnology have produced a pressing need to integrate and make accessible large volumes of genomic information. First, large-scale chromosome mapping strategies [1, 5, 6, 9, 10, 12, 15, 20, 21, 25, 26, 27, 28, 33, 34] are now being successfully used to determine the chromosome locations of specific DNA sequences. Second, the development of automated DNA fragment analysis and sequencing machines [11] has made it possible to determine the complete DNA sequence for any organism with a small genome in a reasonable amount of time. Large-scale efforts at determining the complete DNA sequence of several model organisms have been targeted by the joint DOE/NIH Human Genome Project (HGP) [9]. Though relatively little gene sequence data has been produced by its component projects so far, approximately three gigabases of human DNA sequence will be determined in the next fifteen years. This number translates, at two bits per base. into approximately 750 megabytes of data, or about the size of the database that can fit onto a relatively cheap, commercially available hard disk drive for any desktop workstation. Thus, the scientific issue is not storage per se, but a mechanism for providing flexible access to stored sequence information in order to analyze it. For example, consider the process of determining large DNA sequences. A sequencing project requires extensive manipulation of the data for sequences and clones to keep track of experimental details. Systematic computational analysis of these data is also required to determine the course of continued experimentation, diagnose discrepancies and errors in the data, and evaluate progress toward the goal of completing the sequenced DNA fragment. Such systematic analysis requires reliable and flexible access to the clone and sequence information. Finally, there must be a continuing effort to interpret the data, which often necessitates manipulation of the data using novel methods. Yet because the methods used in determining sequences and the underlying conceptual framework for analysis are changing almost daily, an adaptable system is required that is easy and natural for practicing biologists to use when analyzing the data and designing experiments.

1.1 Flexibility of a Chromosome Analysis System

The answers to many challenging questions in biology require an analysis facility that combines information from different subdisciplines to form a coherent picture of the genetic basis of a biological process. Indeed, a key element in successfully interpreting the biological "meaning" of genomic sequence data hinges on the availability of a wide spectrum of information. For example, in the assignment of chromosomal locations of a specific sequence of an organism, a researcher may wish information on the clonal origin of the sequenced fragment, as well as access to high-resolution physical and genetic maps for the chromosome.

Recent improvements in experimental technologies have facilitated a shift in focus to larger-scale projects aimed at integrating more global biological information. Existing DNA sequence and restriction map data have been consolidated into a coherent representation [4, 23, 29, 30]. Large-scale physical mappings of several organisms, including the yeast *S. cerevisae* [16, 22, 14], the fruit fly *Drosophila melanogaster* [1, 15, 33], the nematode *C. elegans* [13, 31], and all human chromosomes [2, 6, 8, 9, 11, 34] are in progress. The most complete collection of genomic data is for the bacterium *E. coli*: approximately 30% of the chromosome has been sequenced, complete low- and high-resolution restriction maps are available, approximately half of the genes have been identified, and several ordered libraries of clones are available. This rich information base provides an excellent platform to explore the principles fundamental to manipulating sequences, performing comparative analysis of multiple maps, and resolving the chromosomal location of the new sequence information.

Integrating and reconciling these different data with DNA sequence data into a knowledge base to support both broadly based research and the genome projects poses substantial challenges. One major challenge is that information is in a continual state of flux. New data are being added, and experimental errors are corrected. Moreover, this state of flux goes far beyond the automatic updating of previous information required after every transaction. The biological concepts that underlie the organization of the database are in constant revision. There are changes even to the questions users wish to pose: as new experimental protocols are invented, the data types, the inferences drawn, and the questions all change.

Diversity presents another major challenge for the integration of scientific databases. Data can be of widely different quality and even contradictory. Multiple values, or none at all (null values), for a given attribute can occur. Further, the user community is diverse—including DNA sequencing project managers, biochemists, and population geneticists, each with a customized set of algorithms and queries.

Thus, any chromosome analysis system that seeks to accommodate biological information from multiple sources must be extremely flexible in both design and use.

1.2 Ease of Access

The second issue is the ease of user access. While many different algorithms exist for the analysis of gene sequence information, each software package implements those algorithms using different data formats and requires the user to learn yet another set of conventions for constructing queries. Posing even relatively simple queries can require substantial effort. To ease this burden, various groups of departmental "experts" have been formed, groups to whom other scientists come for help and instruction. However, since few departments can afford professional database managers, or even formal training for their "experts," many interesting questions go unaddressed.

Therefore, any new system should allow users to formulate new queries as easily and as intuitively as possible. Such a system should also interface with existing packages, in order to maximize the amount of genome information available.

1.3 Reconciliation of Data

The final issue is the reconciliation of different interpretations of the data. Genetic information and gene sequence data come from multiple sources in different formats. Such sources may disagree even on the usage of common terms. A gene in one database may be understood to be the sequence data coding for a protein, while in another context it may include adjoining regulatory regions. While synonyms are easy to recognize since most databases include suitable pointers or tables, homonyms require a knowledge of the biological literature to determine whether two terms represent the same entity.

These difficulties complicate the normal task of assuring data integrity. Since the data should be biologically appropriate, integrity checks can and should be performed. For example, determining potential protein coding regions (open reading frames, or ORFs) in a DNA sequence and comparing these with genetic data can be quite powerful in assigning a chromosome position. Such tasks, however, require expert knowledge. Clearly, a system is needed that enables the automatic comparison of multiple interpretations of chromosome organization.

1.4 Current Systems

Currently, data sent to a centrally supported distribution mechanisms (e.g., Genbank or EMBL) are accessed by one of two techniques. The researcher may use a limited set of tools to locate sequences similar to a specified sequence. Alternatively, the researcher can hire a programmer to write special-purpose programs designed to answer specific, but unpredictable questions. The former technique is limited by the number and type of tool available. The latter technique is limited by its cost and its applicability to only a few specific organisms. What is needed is an environment that is extremely flexible, enables data to be readily incorporated, and is relatively easy for biologists to use.

1.5 Prototype Database Based on Logic Programming

We have developed such an environment [18, 24] Specifically, we have used logic programming to design and implement a prototype database of genomic information for the model bacterial organism *Escherichia coli.*

We have based our approach on logic programming for two principal reasons. First, logic programming enables rapid prototyping and adaptable data retrieval. The technical problems outlined above make it particularly important to experiment in a restricted domain before proceeding to more complex databases involving multiple genomes. Second, logic programming enables the straightforward inclusion of the query capabilities of a relational database with the ability to do pattern-matching operations against sequence data in a single declarative framework.

The virtues of logic programming to support flexible access to data are well understood. We have developed a logic programming workbench for genome analysis based on the language Prolog. This prototype environment was designed to facilitate the exploration of chromosome structure and organization. While the primitives we describe for accessing the data do require some computational education of the user, most queries can be formulated easily with minimum instruction. Furthermore, we have already constructed a natural-language interface that demonstrates the utility of the underlying primitives, and several graphical display interfaces written in C to visualize the spatial relationships of the integrated data and chromosome analysis features. We shall describe these interfaces in separate documents. We believe that the features included in our current system, along with the relatively short time required to construct the system, support our decision to base our implementation on logic programming.

This report presents the fundamental database primitives that can be used to access and manipulate data relating to the *E. coli* genome. The present system, combined with a tutorial manual, provides immediate access to the integrated knowledge base for *E. coli* chromosome data. It also serves as the foundation for development of more user-friendly interfaces that have the same retrieval power and high-level tools to analyze complex chromosome organization.

2 Conceptual Framework

Like the data in all experimental biological databases, the data here should be understood to be tentative, in a temporary state of validation. Some items are believed to be almost certain, while others are far less determined and reflect the views of the curator. Any database provides a more or less accurate model of reality that can be queried. The conclusions drawn from the model inherently reflect the degree of certainty in the incorporated data. The goal of our work is to make the interrogation of the model as straightforward and as flexible as possible.

The *E. coli* chromosome for this work is represented as a double-stranded piece of DNA of fixed length. The current implementation defines this length at 4,672,600 bases pairs. This length is an extrapolation based on the high-resolution physical map of the *E. coli* chromosome and the known lengths of assembled sequenced portions of the chromosome represented in the EcoSeq data collection. Oriented sequence fragments containing 1,332,986 bases have been assigned positions that account for 28.5% of the chromosome [29, 30].

2.1 Objects with Positions on the Chromosome

The system supports queries relating to various types of object. One general category involves objects that have been assigned or mapped to positions on the chromosome. The system supports queries concerning the locations, directional arrangements, and distributions of such objects. Initially, the objects with positions on the chromosome that can be queried fall into the following categories:

1. *Kohara's clones* – the cloned DNA fragments used by Kohara [19] to determine the high-resolution physical map of the *E. coli* chromosome.

2. *Kohara's restrictions sites* – the estimated positions of restriction enzyme cut sites within Kohara's cloned *E. coli* DNA fragments, used to assemble the high-resolution physical map for the *E. coli* genome. Those restriction enzyme sites are BamHI, Bgl1, EcoR1, EcoR5, Hind3, Kpn1, Pst1, and Pvu2.

3. *Fragments of sequence* – the DNA sequence contigs and individual sequences that make up the Rudd EcoSeq database. Many of the sequences have been assigned genome positions

based on a comparison of the distribution of restriction enzyme sites in sequences and the physical map.

4. *Restriction sites that occur within sequence fragments* – the same eight restriction enzyme DNA sequence recognition sites that were used by Kohara and have been identified by pattern analysis of the DNA sequence data. The sites are BamHI, GGATCC; Bgl1, GCCnnnnnGGC; EcoR1, GAATTC; Eco R5, GATATC; Hind3, AAGCTT; Kpn1, GGTACC; Pst1, CTGCAG; and Pvu2, CAGCTG.

5. *Structural genes that have been identified by direct DNA sequencing* – DNA sequence regions for structural RNAs (such as tRNA and rRNAs) and protein coding regions. All genes have a length and a direction of information content that corresponds to the direction of transcription.

Some of these objects have been assigned to sections of the chromosome that have been sequenced (e.g., all "fragments of sequence," six of Kohara's clones, and some structural genes); others have been partially sequenced or not sequenced at all.

In the following subsections, we illustrate some of the basic queries that can be used to access data about these objects. The Appendix contains a summary of the Prolog predicates that were developed to organize and manipulate this *E. coli* knowledge base. In a later section, we use these basic techniques to illustrate the level of interaction required to answer more complex questions typical of those that might be made by a molecular biologist.

2.1.1 Kohara's Clones and Restriction Sites

Each of Kohara's clones has a unique identifier. One can access the object corresponding to a specific identifier and display it using the following Prolog query:

```
| ?- kohara_clone('[629B]18C4',Clone),display_object(Clone).
```

```
4240715/4243455   2741      [629B]18C4        (Kohara clone)
```

Here, the system displays the position (beginning/end), length, and identifier of the clone. To list the set of Kohara restriction sites that occur in a given clone, one might use a query of the form

```
| ?- kohara_clone('[531B]3C5',Clone),
setof(Site,(kohara_rsite(Site),contains(Clone,Site)),Sites),
display_objects(Sites).
```

```
4234059/4234064      6             EcoR5           (Kohara site)
4234092/4234097      6             EcoR5           (Kohara site)
4234292/4234297      6             EcoR5           (Kohara site)
4234440/4234450     11              Bgl1           (Kohara site)
4235157/4235162      6             EcoR5           (Kohara site)
4236072/4236082     11              Bgl1           (Kohara site)
4236533/4236538      6             EcoR5           (Kohara site)
4236665/4236675     11              Bgl1           (Kohara site)
```

```
4236848/4236853     6          EcoR1          (Kohara site)
4237609/4237614     6          Hind3          (Kohara site)
4238177/4238182     6          Hind3          (Kohara site)
4238203/4238208     6          EcoR1          (Kohara site)
4238367/4238377    11           Bgl1          (Kohara site)
4240268/4240273     6          EcoR1          (Kohara site)
```

This query retrieves exactly those Kohara physical map sites associated with clone [531B]3C5 and displays their locations and lengths. (For further explanation of the **display** capability, see Section 2.2.)

In the preceding example, we used **kohara_rsite(Site)** to retrieve an arbitrary Kohara restriction site. The following Prolog predicate retrieves a Kohara restriction site corresponding to a specific restriction enzyme:

```
| ?- kohara_rsite(Beg,End,Enzyme).
        Beg = 600,      End = 610,      Enzyme = 'Bgl1' ;
        Beg = 1458,     End = 1468,     Enzyme = 'Bgl1' ;
        Beg = 2611,     End = 2616,     Enzyme = 'Pvu2' ;
        Beg = 3709,     End = 3714,     Enzyme = 'EcoR1';
   .
   .
   .
```

By invoking **kohara_rsite/3** with the third argument instantiated, one can extract restriction sites for a specific enzyme:

```
| ?- kohara_rsite(Beg,End,'Not1').

        Beg = 25087,    End = 25094 ;
        Beg = 679216,   End = 679223 ;
        Beg = 786494,   End = 786501 ;
   .
   .
   .
```

To collect all Kohara clones or restriction enzyme map sites, we provide the predicates **all_kohara_clones(Clones)** and **all_kohara_rsites(Rsites)**. In both cases, the objects are sorted based on starting location on the chromosome.

2.1.2 Fragments of Sequence

Knowledge about the *E. coli* genome has progressed to the point where many of the isolated sequence entries in Genbank can be assigned locations on the chromosome [29, 30]. Our database includes those nonoverlapping entries from the EcoSeq database, each of which has an associated unique identifier. To access the position and length of a specified object, one uses a Prolog query of the following form:

```
| ?- dna_fragment('ECOPROC',Fragment),display_object(Fragment).

411369/412336      968          ECOPROC        (DNA fragment)
```

Note that what we are calling a "fragment" is a specified section of the chromosome that has been sequenced; to access the sequence associated with the fragment, one uses the tools described in Section 2.2.

To access the complete set of DNA sequence fragments, one uses the predicate all_dna_fragments(Fragments). As with the predicates for Kohara clones and restrictions sites, the objects are ordered based on starting location.

2.1.3 Computed Restriction Sites

For each section of the chromosome that has been sequenced, we can compute the position of restriction sites that occur in that region. This capability is extremely useful for comparing the arrangement of sites in a new DNA fragment against a physical map of the Kohara restriction sites. The alignment of such restriction sites was one of the main methods of positioning fragments of sequence on the genome [29, 30]. The predicates for computed restriction sites are similar to those used to access Kohara restriction sites:

```
| ?- dna_frag_rsite(Obj).
        Obj = dna_frag_rsite(1973976,1973981,'Acc1') ;
        Obj = dna_frag_rsite(1974741,1974746,'Acc1') ;
        Obj = dna_frag_rsite(1974347,1974352,'Acy1') ;
        Obj = dna_frag_rsite(1974329,1974334,'Af13')

| ?- dna_frag_rsite(Beg,End,Enz).
        Beg = 1973976,  End = 1973981,  Enz = 'Acc1' ;
        Beg = 1974741,  End = 1974746,  Enz = 'Acc1'

| ?- dna_frag_rsite(Beg,End,'EcoR1').
        Beg = 335988,   End = 335993 ;
        Beg = 338631,   End = 338636 ;
        Beg = 338989,   End = 338994
```

We have a large list of restriction enzymes sites that are known to the system. To compute positions any restriction enzyme site, one can use the following:

```
| ?- restriction_site('Not1',Pattern,Cuts), format('~s~n',
[Pattern]).
        GCGGCCGC

| ?- restriction_site('AlwN1',Pattern,Cuts), format('~s~n',
[Pattern]).
        CAGnnnCTG
```

To compute the set of restriction sites corresponding to a set of restriction enzymes in a given object, one uses restriction_sites_in_object/3:

```
| ?- gene(aceE,Gene),
restriction_sites_in_object(Gene,['EcoR1','BamH1','BbvS1'],
Sites),display_objects(Sites).

123370/123375          6         GGATCC          (BamH1)
123625/123629          5          GCTGC          (BbvS1)
123826/123830          5          GCAGC          (BbvS1)
123899/123904          6         GAATTC          (EcoR1)
124129/124133          5          GCAGC          (BbvS1)
124246/124250          5          GCTGC          (BbvS1)
124376/124380          5          GCAGC          (BbvS1)
                 .
                 .
                 .
```

2.1.4 Occurrences of Genes

The database includes information about genes that have been sequenced, along with genes that have been assigned positions but have not yet been sequenced. The basic notions of gene that we have implemented are as follows:

structural gene - a section of the chromosome that corresponds to a "mature product." That is, if the gene codes for a protein, the section of the chromosome corresponding to the structural gene will begin with a valid start codon and end with a valid stop codon. Otherwise, it will correspond to a mature RNA product such as tRNA or rRNA. Each gene has an associated "direction of expression," which has two possible values – "clockwise" or "counterclockwise."

translated gene - a structural gene believed to encode a polypeptide. It will always be a multiple of 3 in length, will begin with a valid start codon, and will end with a valid stop codon.

mapped gene - a gene that has been approximately positioned by using genetic mapping [3], but has not yet been sequenced.

known gene - either a structural gene or a mapped gene. Since the lengths of mapped genes are not known, we represent them as points on the chromosome, while structural genes all have known lengths and are thought of as a contiguous section of the chromosome (the complexities associated with the distinction of exons and introns are absent in the restricted case of *E. coli*).

To access structural genes, one uses the gene/2 or gene/4 predicates:

```
| ?- gene(Id,Obj).

Id = thrA,
Obj = gene(thrA,207,2669,clockwise) ;

Id = thrB,
Obj = gene(thrB,2671,3600,clockwise) ;
```

```
Id = thrC,
Obj = gene(thrC,3601,4887,clockwise)

| ?- gene(Id,Beg,End,Direction).

Id = thrA,
Beg = 207,
End = 2669,
Direction = clockwise ;

Id = thrB,
Beg = 2671,
End = 3600,
Direction = clockwise ;

Id = thrC,
Beg = 3601,
End = 4887,
Direction = clockwise
```

To access a gene with a specified Id or Direction, one invokes these predicates with the appropriate arguments instantiated.

To access all genes, one uses **all_genes(Genes)**, which binds Genes to the set of all genes, ordered by starting location (i.e., the start of the gene on the chromosome, irrespective of direction of expression).

To access translated genes, one uses either **translated_gene/2** or **trans-lated_gene/4**:

```
| ?- translated_gene(aceE,Obj).

Obj = gene(aceE,123344,126004,clockwise)

| ?- translated_gene(Id,Beg,End,counterclockwise).

Id = gef,
Beg = 16867,
End = 17019 ;

Id = apaH,
Beg = 50814,
End = 51656
```

To get a list of all genes thought to be translated, one uses

```
all_translated_genes(Genes)
```

To access a mapped gene, one uses **mapped_gene/2**:

```
| ?- mapped_gene(Id,Gene).

Id = tolJ,
Gene = mapped_gene(tolJ,'Bach.',unknown,4.0E-02,6099) ;

Id = tolI,
Gene = mapped_gene(tolI,'Bach.',unknown,5.0E-02,6645) ;

Id = popD,
Gene = mapped_gene(popD,'Bach.',unknown,8.0E-02,8284) ;
   .
   .
   .
```

Note that the second argument is bound to a structure of the form

mapped_gene(Id,Map,Direction,PositionOnMap,PositionOnChromosome)

Here, 'Bach.' is a reference to the digitized Bachmann genetic map [3], 4.0E-02 is a position in the units chosen by the person constructing the map (in this case, minutes), and 6099 is the best estimate of the position on the chromosome (in terms of base pairs).

To access known genes (both structural genes and mapped genes), one uses **known_gene/2**:

```
| ?- known_gene(Id,Gene).

Id = thrA,
Gene = gene(thrA,207,2669,clockwise) ;

Id = thrB,
Gene = gene(thrB,2671,3600,clockwise) ;
   .
   .
   .
```

To access entire collections of either known or mapped genes, one uses the predicates **all_known_genes/1** and **all_mapped_genes/1**.

2.2 Predicates Common to All Objects Located on the Chromosome

To access the location of any object on the chromosome, one uses the `location/3` predicate:

```
| ?- gene(entA,Obj), location(Obj,Beg,End).

Obj = gene(entA,636874,637620,clockwise),
Beg = 636874,
End = 637620
```

Alternatively, one can use start_of/2 and end_of/2:

```
| ?- gene(entA,Obj), start_of(Obj,Beg), end_of(Obj,End).

Obj = gene(entA,636874,637620,clockwise),
Beg = 636874,
End = 637620
```

To determine whether an object has been sequenced, one uses the predicate sequenced/1. Thus,

```
| ?- gene(Id,Obj), sequenced(Obj).

Id = thrA,
Obj = gene(thrA,207,2669,clockwise)
```

is guaranteed to set Obj to a sequenced gene.

The length of an object is computed with

```
| ?- gene(entA,Obj), length_obj(Obj,Ln).

Obj = gene(entA,636874,637620,clockwise),
Ln = 747
```

The sum of the lengths of a list of objects can be computed by using length_objects/2:

```
| ?- all_translated_genes(AllTranslated),
     length_objects(AllTranslated,Ln).

AllTranslated = [gene(thrA,207,2669,clockwise), ...]
Ln = 764226
```

It is often extremely useful to be able to check whether one object contains another. This check can be done with contains/2. For example, to locate the Kohara clone that contains gene phnL, one can use the query

```
| ?- gene(phnL,Gene), kohara_clone(_,Clone),
     contains(Clone,Gene).

Gene = gene(phnL,4354686,4355366,clockwise),
Clone = kohara_clone('[643]12H2',4337800,4358195)
```

To display an object, one uses display_object/1; to display a set of objects, one uses display_objects/1:

```
| ?- gene(phnL,Gene), kohara_clone(_,Clone),
     contains(Clone,Gene),
     display_object(Gene),
     display_objects([Gene,Clone]).

4354686/4355366    681      phnL      (gene)    clockwise

4337800/4358195   20396   [643]12H2   (Kohara clone)
4354686/4355366    681      phnL      (gene)    clockwise
```

We note that display_objects/1 sorts the objects to be displayed into ascending order based on their starting locations. Hence, the Kohara clone appears before phnL in the displayed list.

In Section 2.1.3, we discussed how to locate restriction sites in an object (using restriction_sites_in_object/3). For sequenced objects, one can compute a restriction map of the object (e.g., here gene) and display the object using code similar to the following:

```
 | ?- gene(aceE,Gene),
      map_restriction_fragments(Gene,
      ['EcoR1','Af13','BamH1'],Map),display_objects(Map).

123371/123899     529   [BamH1,EcoR1]   (computed rest. frag.)
123900/124020     121   [EcoR1,Af13]    (computed rest. frag.)
```

To create and display a restriction map based on Kohara restriction sites (which can be done for either sequenced or unsequenced objects), one uses code similar to the following:

```
| ?- kohara_clone('[101]9E4',Clone),
     kohara_map(Clone,['EcoR1','Hind3','EcoR5'],Map),
     display_objects(Map).

3710/5063      1354   [EcoR1,EcoR5]    (Kohara rest. frag.)
5064/5879       816   [EcoR5,EcoR5]    (Kohara rest. frag.)
5880/6602       723   [EcoR5,EcoR5]    (Kohara rest. frag.)
6603/8602      2000   [EcoR5,EcoR5]    (Kohara rest. frag.)
8603/8900       298   [EcoR5,Hind3]    (Kohara rest. frag.)
8901/13006     4106   [Hind3,EcoR1]    (Kohara rest. frag.)
13007/13278     272   [EcoR1,EcoR5]    (Kohara rest. frag.)
13279/13710     432   [EcoR5,EcoR5]    (Kohara rest. frag.)
13711/14439     729   [EcoR5,Hind3]    (Kohara rest. frag.)
14440/15322     883   [Hind3,Hind3]    (Kohara rest. frag.)
```

2.3 The Use of Actual Sequence Data

A central goal of our prototype environment is not only to demonstrate a capability of manipulating relational data about the chromosome, but also to support an extensive sequence searching functionality. For example, one type of analysis involves the identification of regions in the DNA that could form a secondary structure known as a hairpin. Hairpin structures are characterized by a region of sequence that is followed by a complementary sequence. For example, the short section of sequence ACCGTTAGCAACGGT can form a hairpin, with ACCGTT pairing with the final AACGGT, and the three middle characters forming "the loop." These hairpin structures are often part of the genetic control mechanisms. With our prototype, one can easily write a query to extract all hairpins that occur near the end of any structural gene. One merely uses the relational capabilities discussed above to locate the sections of the chromosome that correspond to the notion "near the end of a structural gene" and then uses the pattern-matching functions to check for hairpins.

In this section, we discuss the fairly low-level operations to access and search a sequence. We also discuss how to search for patterns, translate genes, and search for patterns in translated genes. We believe that these capabilities go beyond those normally offered by chromosomal databases and that they are extremely useful for supporting active research about the contents of the chromosome.

2.3.1 Accessing the Sequence of an Object

To access the sequence of the fragment, one can use the following:

```
| ?- dna_fragment('ECOPROC',Fragment),
     sequence_of(Fragment,Seq),
     display_object(Seq).

411369/412336:            sequence
   411369     GGTTAAATTGAAATTTGCATAAAAATTGCGGCCTATATGGATGTTGGAAC
   411419     CGTAAGAGAAAATGAATTTCACGGCAGGAGTGAGGCAATGGAAAAGAAAA
   411469     TCGGTTTTATTGGCTGCGGCAATATGGGAAAAGCCATTCTCGGCGGTCTG
   411519     ATTGCCAGCGGTCAGGTGCTTCCAGGGCAAATCTGGGTATACACCCCCTC
   411569     CCCGGATAAAGTCGCCGCCCTGCATGACCAGTTCGGCATCAACGCCGCAG
   411619     AATCGGCGCAAGAAGTGGCGCAAATCGCCGACATCATTTTTGCTGCCGTT
   411669     AAACCTGGCATCATGATTAAAGTGCTTAGCGAAATCACCTCCAGCCTGAA
   411719     TAAAGACTCTCTGGTCGTTTCTATTGCTGCAGGTGTCACGCTCGACCAGC
   411769     TTGCCCGCGCGCTGGGCCATGACCGGAAAATTATCCGCGCCATGCCGAAC
   411819     ACTCCCGCACTGGTTAATGCCGGGATGACCTCCGTAACGCCAAACGCGCT
   411869     GGTAACCCCAGAAGATACCGCTGATGTGCTGAATATTTTCCGCTGCTTTG
   411919     GCGAAGCGGAAGTAATTGCTGAGCCGATGATCCACCCGGTGGTCGGTGTG
   411969     AGCGGTTCTTCGCCAGCCTACGTATTTATGTTTATCGAAGCGATGGCCGA
   412019     CGCCGCCGTGCTGGGCGGGATGCCACGCGCCCAGGCGTATAAATTTGCCG
   412069     CTCAGGCGGTAATGGGTTCCGCAAAAATGGTGCTGGAAACGGGAGAACAT
   412119     CCGGGGGCACTGAAAGATATGGTCTGCTCACCGGGAGGCACCACCATTGA
   412169     AGCGGTACGCGTACTGGAAGAGAAAGGCTTCCGTGCTGCAGTGATCGAAG
   412219     CGATGACGAAGTGTATGGAAAAATCAGAAAAACTCAGCAAATCCTGATGA
   412269     CTTTCGCCGGACGTCAGGCCGCCACTTCGGTGCGGTTACGTCCGGCTTTC
   412319     TTTGCTTTGTAAAGCGCT
```

Here, only the sequence of the clockwise strand of DNA is displayed. That is,

```
sequence_of(Object,Seq)
```

sets Seq to a "sequence object" representing the sequence of Object, and

```
        display_object(AnyObject)
```

displays any object, including a "sequence object." One can also extract any sequence by absolute coordinates. Thus, the following works as well.

```
| ?- sequence_at(123344,126004,Seq),display_object(Seq).

123344/126004:          sequence
   123344    ATGTCAGAACGTTTCCCAAATGACGTGGATCCGATCGAAACTCGCGACTG
   123394    GCTCCAGGCGATCGAATCGGTCATCCGTGAAGAAGGTGTTGAGCGTGCTC
   123444    AGTATCTGATCGACCAACTGCTTGCTGAAGCCCGCAAAGGCGGTGTAAAC
      .
      .
      .
```

To access subsequences of a sequence, one can use subseq(Position,Length, SubSequence,Sequence) Specifically, this can be used either to find the subsequence at a given position in a sequence or to search for where a given subsequence occurs in a sequence. For example, the following query computes all of the ten character sequences that occur at least twice in the gene aceE.

```
        | ?- gene(aceE,Gene),sequence_of(Gene,Seq),
             subseq(Pos1,10,SubSeq,Seq),
             subseq(Pos2,10,SubSeq,Seq), Pos2 > Pos1,
             format('~d/~d: ~s~n',[Pos1,Pos2,SubSeq]),
             fail.

        123541/124860: TGAAGAACAA
        123575/123604: CTGGAACGCC
        123744/125084: GCGGCGACCT
        124190/125450: GAAGGTGCTG
        124281/125715: TGATGAACGA
        124631/125972: GATGCAGATA
        124747/125623: CTTCACCGAG
        125545/125851: CCTGCGTCAC

        no
        | ?-
```

This is such a common request that we have included a predicate that computes the set of such common sequences:

```
| ?- gene(aceA,Gene),
     common_seqs_at_least_k_long([Gene,Gene],10,Seqs),
     display_objects(Seqs).
```

```
4246610/4246619:         sequence
               4246610   TCCTGAATGC
4246984/4246993:         sequence
               4246984   TCCTGAATGC

4246902/4246914:         sequence
               4246902   GCGGGCATTGAGC
4247289/4247301:         sequence
               4247289   GCGGGCATTGAGC
```

Notice that, in this case, matches are extended as far as possible (thus, the second reported match is 13 characters long). One would normally use this with distinct objects, for example,

```
| ?- gene(thrA,Gene),
     start_of(Gene,Start),
     StartPre is Start-100, EndInit is Start+80,
     common_seqs_at_least_k_long([region(StartPre,Start),region
                                  (Start,EndInit)],5,Seqs),
     display_objects(Seqs).
```

```
129/133:                 sequence
               129       GTACA
229/233:                 sequence
               229       GTACA

134/138:                 sequence
               134       GGAAA
278/282:                 sequence
               278       GGAAA

141/145:                 sequence
               141       CAGAA
247/251:                 sequence
               247       CAGAA

147/151:                 sequence
               147       AAAGC
280/284:                 sequence
               280       AAAGC

177/181:                 sequence
               177       TTTTC
254/258:                 sequence
               254       TTTTC
```

We also allow one to look for the longest common subsequence.

```
| ?- gene(aceE,Gene),location(Gene,Beg,End),EndPt is Beg+99,
     sequence_at(Beg,EndPt,Prefix),
     longest_common_subseq(Prefix,Prefix,Common,Pos1,Pos2),
     format('~d/~d ~s~n',[Pos1,Pos2,Common]).

123375/123402 CGATCGAA
```

The answer from this query indicates that the displayed eight-character string is the longest string that occurs twice in the first hundred characters of the gene aceE.

2.3.2 Higher-Level Predicates to Support Scanning for Patterns in Objects

To properly handle requests to search for structures like hairpins or repeats, we implemented the ability to scan for patterns. Here, we think of a pattern as a sequence of pattern units, each of which can be

1. a string of DNA characters (including the codes to represent ambiguous characters);
2. a pattern unit that matches an arbitrary string of characters, where the length of the string varies between specified bounds;
3. a pattern unit that "matches" the reverse complement of a string matched by a previous pattern unit; and
4. a pattern that matches a string identical to a previously matched pattern unit.

The last two types of pattern unit allow one to specify an allowable number of mismatches, insertions, and deletions (which gives an "approximate" matching capability).

For example, we think of the pattern

```
p1=AYGG 3...5 ~p1 p1
```

as capable of matching a sequence like

```
ACGGTTCGCCGTACGG
```

We encode such patterns as Prolog terms. Thus, the preceding pattern is encoded as

```
[pvar(p1,dna("AYGG")),
 ellipses(3,5),
 complement(p1,0,0,0),
 repeat(p1,0,0,0)]
```

The rules for a term encoding a pattern are as follows:

1. A pattern is a list of pattern units.
2. A pattern unit can be a "raw" pattern unit or can have the form

```
pvar(Id,RawUnit)
```

When an Id is specified, it is used to allow following pattern units to refer back to the string matched by this pattern unit.

3. A raw pattern unit must be one of the following:

   (a) dna(String)

   (b) ellipses(Min,Max), where Min and Max give the bounds on the length of the string matched;

   (c) complement(Id,Mis,Ins,Del), where Mis gives the number of allowed mismatches, Ins specifies the number of indels that can be inserted into the string matched, and Del specifies the number of characters in the string being matched that can be deleted; or

   (d) repeat((Id,Mis,Ins,Del), where the parameters are just as for complement.

To scan a section of the chromosome for the occurrence of a pattern, one uses `scan_mem_for_pattern_occurrence/4`:

```
| ?- gene(aceE,Gene),start_of(Gene,Beg),end_of(Gene,End),
     scan_mem_for_pattern_occurrence(Beg,End,
         [pvar(p1,dna("RYRYRY")),
          ellipses(0,400),
          repeat(p1,1,1,0)],Occ),
          display_object(Occ).

123436/123464:            sequence
                 123436   GCGTGC TCAGTATCTGATCGACCA ACTGC
```

By computing the set of such matches, one can rapidly acquire all matches of fairly complex patterns (the actual pattern matching is achieved by invoking an underlying routine written in C).

2.3.3 A Predicate to Support Scanning for Patterns in Translated Genes

We have found that users wish to scan for patterns in the translated genes, as well as for patterns in the DNA sequences. Hence, we have provided a predicate to support this capability:

```
find_pp_match(+Pat,+Gene,-PolyPepTide)
```

Both Pat (a list of the pattern units to scan) and Gene (the gene to be translated) must be specified. Pat is a list of pattern units. Each unit is one of the following:

1. a string of one-character amino acid codes, with ? to represent an arbitrary amino acid (e.g., "CP???H"); or

2. the alternative of two patterns P1 and P2, which is represented as P1;P2.

PolyPepTide is the section of the translation of the Gene that matches Pat. The following example will illustrate:

```
| ?- gene(thrA,Gene),
     find_pp_match(["RE?E",("H";"L")],Gene,Match),
     display_object(Match).


2280/2294       15       thrA  (expressed) clockwise
                        RELE L
```

2.3.4 Predicates for Computing Codon Usage, K-mer Counts, and GC Content

The database provides a facility for computing codon usage for any set of translated genes. This is achieved by using the predicate

```
codon_usage(Objects,Counts)
```

where Objects is a list of translated genes, and Counts is set to a list of 65 integers. The first integer is a count of the number of "invalid" codons (i.e., those that are ambiguous or unsequenced characters). The remaining 64 correspond to the counts of AAA, AAC, AAG, AAT, ACA,...TTT To display the counts in a meaningful way, one can use

```
print_codon_usage(Counts)
```

For example, one can obtain the codon usage statistics for the genes currently placed on the genome by using

```
| ?- all_translated_genes(Genes),
     codon_usage(Genes,Counts),
     print_codon_usage(Counts).

number valid codons = 254740
number invalid codons = 2

    alanine: 24676      9.69%
        GCA: 5133       2.01%
        GCC: 6189       2.43%
        GCG: 9146       3.59%
        GCT: 4208       1.65%

    arginine: 14841     5.83%
        AGA: 291        0.11%
        AGG: 215        0.08%
        CGA: 713        0.28%
        CGC: 5914       2.32%
        CGG: 1130       0.44%
        CGT: 6578       2.58%

    asparagine: 9740    3.82%
    AAC: 6237        2.45%
    AAT: 3503        1.38%

aspartic_acid: 13829 5.43%
    GAC: 5739        2.25%
    GAT: 8090        3.18%

cysteine: 2736      1.07%
    TGC: 1592        0.62%
    TGT: 1144        0.45%

glutamic_acid: 15961 6.27%
    GAA: 11170       4.38%
    GAG: 4791        1.88%

glutamine: 11235     4.41%
    CAA: 3329        1.31%
    CAG: 7906        3.10%

glycine: 19285       7.57%
    GGA: 1490        0.58%
    GGC: 8191        3.22%
    GGG: 2442        0.96%
    GGT: 7162        2.81%

histidine: 5762      2.26%
    CAC: 2819        1.11%
    CAT: 2943        1.16%

isoleucine: 14551    5.71%
    ATA: 604         0.24%
    ATC: 7132        2.80%
    ATT: 6815        2.68%

leucine: 25943      10.18%
    CTA: 747         0.29%
    CTC: 2596        1.02%
    CTG: 14682       5.76%
    CTT: 2392        0.94%
    TTA: 2563        1.01%
    TTG: 2963        1.16%

lysine: 11835        4.65%
    AAA: 9040        3.55%
    AAG: 2795        1.10%
```

```
methionine: 6885      2.70%
    ATG: 6885         2.70%

phenylalanine: 9369   3.68%
    TTC: 4653         1.83%
    TTT: 4716         1.85%

proline: 11145        4.38%
    CCA: 1973         0.77%
    CCC: 1030         0.40%
    CCG: 6609         2.59%
    CCT: 1533         0.60%

serine: 13923         5.47%
    AGC: 3925         1.54%
    AGT: 1698         0.67%
    TCA: 1398         0.55%
    TCC: 2442         0.96%
    TCG: 2050         0.80%
    TCT: 2410         0.95%

stop: 697             0.27%
    TAA: 451          0.18%
    TAG: 49           0.02%
    TGA: 197          0.08%

threonine: 13465      5.29%
    ACA: 1304         0.51%
    ACC: 6436         2.53%
    ACG: 3297         1.29%
    ACT: 2428         0.95%

tyrosine: 7040        2.76%
    TAC: 3403         1.34%
    TAT: 3637         1.43%

valine: 18436         7.24%
    GTA: 2873         1.13%
    GTC: 3724         1.46%
    GTG: 6816         2.68%
    GTT: 5023         1.97%
```

The database also includes the capability of rapidly accumulating statistics on the occurrences of k-mers. In the most trivial case, one can obtai. and display the number of occurrences of each of the four nucleotides by using

```
| ?- all_dna_fragments(Frags),
     kmer_usage(Frags,1,Counts),
     print_kmer_usage(Counts,1),
     print_gc_content(Counts).

 A: 354898               24.29%
 C: 375714               25.71%
 G: 377757               25.85%
 T: 352961               24.15%

        Gs, Cs: 753471    51.56%
        As, Ts: 707859    48.44%

Counts = [354898,375714,377757,352961]
```

The system can accumulate counts for k-mers of any size (although the user will probably not wish to go above 10-mers).

2.4 Interface to External Systems

Our objective is to support the capability of storing and retrieving genetic data; it is certainly not our ambition to recreate the standard tools required to analyze the retrieved sequence data. That is, our system must be able to extract data that can later be processed by standard statistical packages or data that support graphical exploration. This ability to interface to external packages can be achieved in two basic ways:

1. For a very limited set of tools that require efficient transmission of data to and from the tool, it is possible to install the C or Fortran code as "foreign predicates" which can be invoked directly from the Prolog environment. This is how we have integrated the version of the Smith-Waterman algorithm written by Xiaoqiu Huang et al. [17].
2. More commonly, to invoke an external tool, one simply extracts the data, writes it to a file, and invokes a Unix shell script that invokes the desired tool and reformats the produced data in a form accessible by the Prolog system. This is, for example, how we interface to external systems to plot data and how we invoke FASTA [32] (the system for rapid similarity searches, distributed by Bill Pearson).

The second approach is clearly more flexible and offers the most painless way to integrate new capabilities. Tools that perform multiple-sequence alignment and motif searching must be integrated into systems that compute the energetic stability of secondary structures.

3 Encoding of Biologically Relevant Queries

In this section, we illustrate the query facility with the predicates discussed in the preceding section. We have collected questions typical of those asked by molecular biologists. To illustrate the level of difficulty, we provide short routines that will produce the desired answers In each case, the predicates have been implemented in a straightforward manner based on the predicates presented in the Appendix. Specifically, we present a collection of 21 questions about the *E. coli* chromosome, including the query, the answer, and the Prolog solution.

3.1 Physical Map Sites in Objects

The first three queries deal with identifying physical map sites in clones and sequences.

In determining a physical map for a chromosome and in establishing the chromosome positions of genes, it is useful to know which gene regions would be interrupted once by digestion with specific restriction enzymes.

```
Query 1:  For a specified restriction enzyme Not1, find all
          sequenced genes in which Not1 occurs precisely once.
```

```
% | ?- query1('Not1',Genes),display_objects(Genes).
%
% 785627/786892    1266     tolA     (gene)   clockwise
% 816181/817473    1293     bioA     (gene)   counterclockwise
% 1251391/1253088  1698     treA     (gene)   clockwise
% 2011366/2012091   726     orf      (gene)   counterclockwise
% 4083713/4084762  1050     glnL     (gene)   counterclockwise
%
```

```
query1(E,Genes) :-
        set_of_all(Gene,
                   Id^Sites^
                   (gene(Id,Gene),
                    computed_restriction_sites_in_object(Gene,
                    [E],[Sites])),Genes).
```

Subcloning operations designed to manipulate a gene sequence often require a list of restriction enzymes whose cut sites occur exactly once in that gene.

```
Query 2:  For a given sequenced gene thrA, find all restriction
          enzymes that occur precisely once in thrA.
```

```
% | ?- query2(thrA,Enzymes).
%
% Enzymes = ['Ava1','Bbv2','Bcl1','BsaB1','BstX1','Cla1','Dde1',
%            'Drd1','Ear1','EcoA','EcoP1','HgiC1','Mae1','Mst1',
%            'Nae1','Nsp3','NspC1','Pvu1','Pvu2','SgrAI','SnaB1',
%            'Ssp1']
```

```
query2(GeneId,Enzymes) :-
        gene(GeneId,Gene),
        set_of_all(Enz,
                   Pattern^CutPoint^Sites^
                   (restriction_site(Enz,Pattern,CutPoint),
                    computed_restriction_sites_in_object
                   (Gene,[Enz],[Sites])), Enzymes).
```

The enzymes to use in isolating intact genes on single DNA fragments are those whose restriction sites do not cut those genes. The following query allows us to identify that set of restriction enzymes.

```
Query 3.  For a given sequenced gene G, find the set of Kohara
          enzymes that do not occur in G.

% | ?- query3(thrA,Enz).
%
% Enz = ['BamH1','EcoR1','EcoR5','Hind3','Kpn1','Pst1']

query3(GeneId,Enzymes) :-
        gene(GeneId,Gene),
        set_of_all(Enz,
                   Kenz^
                   (kohara_enzymes(Kenz),
                    member(Enz,Kenz),
                    computed_restriction_sites_in_object
                    J(Gene,[Enz],[])),Enzymes).
```

3.2 Identifying Sequence Features

The next collection of queries involves searching for patterns in DNA sequences.

Much of the current work in the molecular biology involves some "reverse engineering." That is, one can often predict a short DNA sequence fragment (also known as a primer) that is characteristic of some genetic or structural trait. These primers can be used as probes to determine which clones contain the potential target genes. However, to find interesting clones for further study, we need to identify the sequenced clones that contain the primers. The following query identifies such clones.

```
Query 4:  For a given sequence X, list all Kohara clones that
          contain X.

% | ?-
query4("GATTGCCAGTTCGCCATAATCACTCTTC",Clones),display_objects
   (Clones).
%
% 1957500/1977500      20001      [337]20H4      (Kohara clone)
% 1969800/1988245      18446      [338]12C7      (Kohara clone)


query4(Seq,Clones) :-
        set_of_all(Clone,
                   Id^Occs^
                   (kohara_clone(Id,Clone),
                    subseqs_in_obj(Clone,Seq,Occs)
                   ),
                   Clones).
```

Conversely, we might like to identify those clones that do not contain a specific target sequence.

```
Query 5:  For a given string X, list all Kohara clones that are
          not known to contain X.


% | ?- query5("GATTGCC",Clones).
%
% Clones = [kohara_clone('[102]6H3',9400,24157),...]
%

query5(Seq,Clones) :-
        set_of_all(Clone,
                   Id^Occs^
                   (kohara_clone(Id,Clone),
                    \+ subseqs_in_obj(Clone,Seq,Occs)
                   ),
                   Clones).
```

Subcloning or probing projects often seek to identify those short unique sequences that are diagnostic for a particular DNA segment. The following query allows us to identify diagnostic sequences of a specific length within a target clone.

```
Query 6:  Given a length K and a clone Clone, produce a
          sequence S that occurs just once in Clone.

% | ?- query6(6,'[116]15A7',S), format('~s~n',[S]).
%
% CGCCTA

query6(K,CloneId,S) :-
        kohara_clone(CloneId,Clone),
        sequence_of(Clone,SeqObj),
        subseq(Pos,K,S,SeqObj),
        \+ (member(Char,S), \+ base(Char)),
        \+ (subseq(Pos2,K,S,SeqObj), Pos2 =\= Pos).
```

To confirm that the sequence is diagnostic of the fragment, we can use the following query to check that the sequence does not occur in any other sequenced clone.

```
Query 7:  Given a length K and a clone Clone, produce a
          sequence that occurs just once in Clone, and
          never in any other Clone.  Check both strands.

% | ?- query7(12,'[116]15A7',S), format('~s~n',[S]).
% ATCGCCTAATGC
%

query7(K,CloneId,S) :-
        kohara_clone(CloneId,Clone),
        sequence_of(Clone,SeqObj),
        subseq(Pos,K,S,SeqObj),
        \+ (member(Char,S), \+ base(Char)),
        domain(ecoli_genome,Beg,End),
        \+ (subseq_both(Pos2,K,S,seq(Beg,End),_), Pos2 =\= Pos).
```

Certain sequences must stand in spatial relationship to one another in order for certain biological mechanisms to take place. For example, genes that are regulated through a coordinated control mechanism using a common control protein usually have common control sequence motifs that occur in specific spatial relationships to those genes. The following query searches for a potential control sequence with a particular spatial requirement. In a relational database, identifying sequence level features such as these normally requires an extensive, specialized programming effort.

```
Query 8: List genes that contain sequence X exactly once, and
        the occurrence is at least a distance of Y away from
        each end of the gene.


% | ?- query8("TGATTTGCT",60,Genes),display_objects(Genes).
%
% 14285/15415       1131   dnaJ  (gene)   clockwise
% 572030/573193     1164   int   (gene)   counterclockwise
% 631876/632832      957   fepB  (gene)   counterclockwise
% 995234/996436     1203   pncB  (gene)   counterclockwise
% 1408669/1409421    753   fnr   (gene)   counterclockwise
% 2104525/2105829   1305   hisD  (gene)   clockwise
% 2448989/2449477    489   dedE  (gene)   counterclockwise
% 2465017/2466087   1071   aroC  (gene)   counterclockwise
% 2699918/2703805   3888   purL  (gene)   clockwise
% 3610926/3611813    888   ugpA  (gene)   counterclockwise
% 3903261/3904334   1074   recF  (gene)   counterclockwise
% 4014398/4015594   1197   hemY  (gene)   counterclockwise
%

query8(X,Y,Genes) :-
   length(X,Ln),
   set_of_all(Gene,
       Id^SeqObj^Pos^Pos2^Dir^Dir2^Beg^End^
       (gene(Id,Gene), sequence_of(Gene,SeqObj),
        subseq_both(Pos,Ln,X,SeqObj,Dir),
        \+ (subseq_both(Pos2,Ln,X,SeqObj,Dir2), Pos =\= Pos2),
        location(Gene,Beg,End),
         Pos-Beg >= Y, End-Pos >= Y
        ),
        Genes).
```

The presence of localized repeated sequences often reflects a common heritage of those chromosome regions. The following query demonstrates how to search for repeats of a definite size within a specific clone.

```
Query 9: List all repeats of length N in Kohara clone C.

% | ?- query9('[102]6H3',13,Repeats),display_objects(Repeats).
%
% 14556/14568:                   sequence
%                  14556         GCGATATTTTTGG
% 14580/14592:                   sequence
%                  14580         GCGATATTTTTGG
%
% 18932/18944:                   sequence
%                  18932         TATGCCGATAAAA
% 19486/19498:                   sequence
%                  19486         TATGCCGATAAAA
%
% 19062/19074:                   sequence
%                  19062         ACGCCGCAGTGGT
% 23657/23669:                   sequence
%                  23657         ACGCCGCAGTGGT
%

query9(CloneId,N,Repeats) :-
        kohara_clone(CloneId,Clone),
        common_seqs_at_least_k_long([Clone,Clone],N,Repeats).
```

Another possibly interesting region ("hot spot") for transcriptional control features, whether sequences or structural features, is the region between convergent genes. The following query searches for such hot spots.

```
Query 10: What is the longest common sequence between two
          convergent transcripts?

% | ?- query10(G1,G2,Common),
%      display_objects([G1,G2]), display_objects(Common),nl,fail.
%
% 15562/16836     1275     orf2  (gene)  clockwise
% 16867/17019      153     gef   (gene)  counterclockwise
%
% 16844/16847:         sequence
%           16844     GGGA
% 16852/16855:        sequence
%           16852     TCCC
%
% 16846/16849:        sequence
%           16846     GATC
%
%
%
% 18719/19507      789    orf3   (gene)  clockwise
% 20833/21096      264    rpsT   (gene)  counterclockwise
%
% 20158/20169:        sequence
%           20158     GCCAGCGCTGGC
%
%
%
% 50257/50736      480    folA    (gene)  clockwise
% 50814/51656      843    apaH    (gene)  counterclockwise
%
% 50761/50767:        sequence
%           50761     GCCGGAT
% 50787/50793:        sequence
%             50787     ATCCGGC
%     .
%     .
%     .

query10(Gene1,Gene2,Longest) :-
        convergent_genes(Gene1,Gene2),
        gap(Gene1,Gene2,Gap),
        (   (common_seqs_at_least_k_long_both_strands
             ([Gap,Gap],8,Common),
             Common \== []) ->
            true
        ;
            common_seqs_at_least_k_long_both_strands
            ([Gap,Gap],4,Common)
        ),
        keep_max(Common,Longest).

keep_max([H|T],Longest) :-
        H=common_sequence([S1|_]),
        length_obj(S1,Ln1),
        keep_max(T,Ln1,[H],Longest).
keep_max([],_MaxLn,Longest,Longest).
keep_max([H|T],MaxLn,MaxSet,Longest) :-
        H=common_sequence([S1|_]),
        length_obj(S1,Ln1),
        (   Ln1 < MaxLn ->
            keep_max(T,MaxLn,MaxSet,Longest)
        ;
            (   Ln1 =:= MaxLn ->
                keep_max(T,MaxLn,[H|MaxSet],Longest)
            ;
                keep_max(T,Ln1,[H],Longest)
            )
        ).
```

Some transcriptional control sequences occur just upstream of a gene. If one conjectured that a particular transcriptional control signal were composed of a single occurrence of a sequence in the gene, together with two identical sequences at different positions upstream of that gene, the following query would extract the desired data.

```
Query 11: For a gene G, find all strings of length at least 6
          that occur at least twice in the first 150
          characters upstream and at least once in the first
          100 characters of G.
```

```
% | ?- query11(Id,Strings),gene(Id,Gene),display_object(Gene),
%      display_objects(Strings).
%
% 84435/85307     873     leuO      (gene)     clockwise
%
% 84407/84415:        sequence
%           84407     GGAGTTAAG
% 84425/84433:        sequence
%           84425     GGAGTTAAG
% 84470/84478:        sequence
%           84470     GGAGTTAAG
%    .
%    .
%    .
```

```
query11(GeneId,Strings) :-
        gene(GeneId,Gene),
        upstream(Gene,150,Upstream),
        initial(Gene,100,Initial),
        common_seqs_at_least_k_long([Upstream,Upstream,Initial],
        6,Strings),Strings \== [].
```

```
upstream(Gene,Ln,region(Pt1,Pt2)) :-
        direction(Gene,Dir), location(Gene,Beg,End),
        (   Dir == clockwise ->
            Pt1 is Beg-Ln, Pt2 is Beg-1
        ;
            Pt1 is End+1, Pt2 is End+Ln
        ).

initial(Gene,Ln,region(Pt1,Pt2)) :-
        direction(Gene,Dir), location(Gene,Beg,End),
        (   Dir == clockwise ->
            Pt1 is Beg, Pt2a is Beg+Ln, min(End,Pt2a,Pt2)
        ;
            Pt1 is End, Pt2a is End-Ln, max(Beg,Pt2a,Pt2)
        ).
```

3.3 Structure-Related Features

The following four queries ask about the arrangement of genes on the chromosome and about potential structural features, such as hairpins, that may be related to gene positions.

According to one well-known hypothesis, there is a correlation between the direction of replication and the strand on which genes are predominantly found [7]. The following query retrieves the data available to test this hypothesis.

```
Query 12: Give the counts of clockwise genes in the region
          just preceding the origin of replication and just
          following it, along with the percentage of each
          region that is sequenced.  Then, do the same for
          counterclockwise genes.

% | ?- query12(100000).
% 3853061/3953061  100001                          (region)
% 1 cw genes; 33 ccw genes; 39% sequenced
% 3953061/4053061  100001                          (region)
% 35 cw genes; 8 ccw genes; 49% sequenced

query12(Dist) :-
        oriC(ecoli,Origin),
        Left is Origin-Dist, Right is Origin+Dist,
        report_on_region(region(Left,Origin)),
        report_on_region(region(Origin,Right)).

report_on_region(Region) :-
        genes_in_object(Region,clockwise,CWG),
        genes_in_object(Region,counterclockwise,CCWG),
        length(CWG,CWcount), length(CCWG,CCWcount),
        kmer_usage([Region],1,[A,C,G,T]),
        length_obj(Region,Ln),
        PerCent is integer(100 * ((A+C+G+T) / Ln)),
        display_object(Region),
        format('~d cw genes; ~d ccw genes; ~d% sequenced~n',
               [CWcount,CCWcount,PerCent]).

genes_in_object(Object,Direction,Genes) :-
        set_of_all(Gene,
                   Id^
                   (gene(Id,Gene),
                    direction(Gene,Direction),
                    contains(Object,Gene)
                   ),Genes).
```

Similarly, one may wish to know whether there is a correlation between the direction of replication and the frequencies of occurrences of different sequences of length four (4-mers).

```
Query 13: Consider the set of 4-mers that occur in clockwise
          genes just to the left of the origin of replication
          and in clockwise genes just to the right.  Are the
          frequencies of occurrence for each 4-mer about the
          same?  In particular, give the set of 4-mers that
          occur more than twice as often (as a percentage of
          the length of the sequence of clockwise genes) on
          one side or the other.

% | ?- query13(200000).
% CCTT: left=0.0012 right=0.0027
% CTAG: left=0.0002 right=0.0004
% TAGG: left=0.0005 right=0.0012
%

query13(Dist) :-
        oriC(ecoli,Origin),
        Left is Origin-Dist, Right is Origin+Dist,
        get_adjusted_counts(region(Left,Origin),LeftCounts),
        get_adjusted_counts(region(Origin,Right),RightCounts),
        report_disparity(LeftCounts,RightCounts).

get_adjusted_counts(Region,Counts) :-
        genes_in_object(Region,clockwise,CWG),
        kmer_usage(CWG,4,[Counts1]),
        sumL(Counts1,Sum),
        adjust_to_give_fraction(Counts1,Sum,Counts).

sumL(L,Sum) :- sumL(L,0,Sum).
sumL([],Sum,Sum).
sumL([H|T],SoFar,Sum) :- SoFar1 is SoFar+H, sumL(T,SoFar1,Sum).

adjust_to_give_fraction([],_,[]).
adjust_to_give_fraction([H|T],Sum,[Ha|Ta]) :-
        Ha is H / Sum,
        adjust_to_give_fraction(T,Sum,Ta).

report_disparity(Left,Right) :- report_disparity(Left,Right,0).

report_disparity([],[],_).
report_disparity([Lh|Lt],[Rh|Rt],Which) :-
        (   (Lh >= 2*Rh ; Rh >= 2*Lh) ->
            conv_kmer(4,Which,String),
            format('~s: left=~4f right=~4f~n',[String,Lh,Rh])
        ;
            true
        ),
        Which1 is Which+1,
        report_disparity(Lt,Rt,Which1).
```

Hairpin loops are often proposed to be structural signals for transcriptional regulation. To find transcriptional signals common to a set of genes, we might wish to identify a set of hairpin loops that occur at the beginning of genes. The following query identifies the genes that contain hairpins within 20 bases of the start of the gene.

```
Query 14: Find all hairpin loops with that occur at the start
          of genes.


% | ?- query14(20,9).
% 27228/28142       915    orf    (gene)    clockwise
% 27208/27231:          sequence
%           27208       GCATTTTTT ATGGAG AAAACATGC
%
% 98459/99703       1245   ftsW   (gene)    clockwise
% 98442/98479:          sequence
%           98442       GCGAAGGAG TTAGGTTGATGCGTTTATCT CTCCCTCGC
%
% 108335/111040     2706   secA    (gene)    clockwise
% 108327/108347:        sequence
%           108327      ATTTTATTA TGC TAATCAAAT
%
% 231921/233462     1542   rrsH    (gene)    clockwise
% 231909/231938:        sequence
%           231909      CATCAAACT TTTAAATTGAAG AGTTTGATC
%
```

```
% 736274/738322    2049    kdpB    (gene)  counterclockwise
% 738321/738342:      sequence
%          738321     ATATTCAGT GCTC ACTCAATAT
%
% 1303723/1306398  2676    adhE    (gene)   clockwise
% 1303706/1303735:   sequence
%          1303706   ACCTTCTAC ATAATCACGACC GTAGTAGGT
%
% 1320555/1321094   540    orf     (gene)   counterclockwise
% 1321074/1321097:   sequence
%          1321074   AAAATCAAG AAACTG CTTCATTTT
%
% 2272115/2273806  1692    fruA     (gene)   clockwise
% 2272098/2272131:   sequence
%          2272098   CAATCAGGC ATTTATCGACATAAAC GCCAGATTG
%              .
%              .
%              .

query14(Dist,Stem) :-
        gene(_Id,Gene),
        once((
            around_start(Gene,Dist,Pt1,Pt2),
            scan_mem_for_pattern_occurrence(Pt1,Pt2,
                [pvar(p1,ellipses(Stem,Stem)),
                 ellipses(3,20),
                 complement(p1,1,0,0)
                ],
                Occ),
            display_object(Gene), display_object(Occ),nl)
        ),
        fail.
query14(_,_).

around_start(Gene,Dist,Pt1,Pt2) :-
        (   direction(Gene,clockwise) ->
            start_of(Gene,Start),
            Pt1 is Start-Dist, Pt2 is Start+Dist
        ;
            end_of(Gene,End),
            Pt1 is End-Dist, Pt2 is End+Dist
        ).
```

It is also possible to query the knowledge base about structural features of RNA molecules. Double-stranded hairpin stems in RNA molecules consist of the complementary base pairs A-U, G-C, and G-U. In investigating the potential structure of an RNA molecule transcribed from a known gene in another species, we detected complementary sequences as long as 18 bases. Such complementary sequences could form hairpins in the transcribed RNA molecules. How often do such complementary sections occur?

```
Query 15: Find all hairpins with stems 18 bases in length with
          loops that could be as large as 300 bases, allowing
          for G-T as a "match."
```

```
% | ?- query15(N).
%
% 85385/85402:                 sequence
%                85385         TGCAGAATAGGTCAGACA
% 85407/85424:                 sequence
%                85407         TGTCTGGTTTATTCTGCA
%
% 123257/123274:               sequence
%                123257        GAACCTGTCTTATTGAGC
% 123287/123304:               sequence
%                123287        GTTCAATGGGACAGGTTC
%
% 123258/123275:               sequence
%                123258        AACCTGTCTTATTGAGCT
% 123286/123303:               sequence
%                123286        AGTTCAATGGGACAGGTT
%
% 123259/123276:               sequence
%                123259        ACCTGTCTTATTGAGCTT
% 123285/123302:               sequence
%                123285        GAGTTCAATGGGACAGGT
%                   .
%                   .
%                   .
```

```
query15(N) :-
        set_of_all(HairPin,rna_hairpin(18,HairPin),L),
        length(L,N).

rna_hairpin(Ln,hairpin(seq(B1,B1e)-Occ)) :-
     all_dna_fragments(Frags),
     member(Frag,Frags), format('checking ~w~n',[Frag]),
     location(Frag,Beg,End), End1 is End-21,
     subseq(B1,Ln,DNA,seq(Beg,End1)),
     S2 is B1+(Ln+3), E2 is S2+300, min(E2,End,E2a),
     to_look_for(DNA,RNAcomp),
     scan_mem_for_pattern_occurrence(S2,E2a,[dna(RNAcomp)],Occ),
     B1e is B1+(Ln-1),
     display_objects([seq(B1,B1e),Occ]).
```

```
to_look_for(DNA,RNAcomp) :- reverse(DNA,DNAr}, rna_comp
    (DNAr,RNAcomp).

rna_comp([],[]).
rna_comp([H|T],[H2|T2]) :- rna_comp_char(H,H2), rna_comp(T,T2).

rna_comp_char(65,84).    % A/T
rna_comp_char(67,71).    % C/G
rna_comp_char(71,89).    % G/Y
rna_comp_char(84,82).    % T/R
```

3.4 Questions about the Overall Project Status

This final group of queries is directed toward assessing the current status of the assembly of the total genome sequence.

In the management of a large-scale sequencing project, one must know the current status with respect to project completion. The following query identifies which clones have been completely sequenced.

```
Query 16:  List all clones that are completely sequenced.

% | ?- query16(Clones),display_objects(Clones).
%
% 96594/105701        9108      [110]6F3      (Kohara clone)
% 3444102/3447540     3439      [630A]5F12    (Kohara clone)
% 3936168/3952263    16096      [560]2A1      (Kohara clone)
% 4233865/4240715     6851      [531B]3C5     (Kohara clone)
% 4240030/4240715      686      [530B]6G9     (Kohara clone)
% 4240715/4243455     2741      [629B]18C4    (Kohara clone)
%

query16(SequencedClones) :-
        all_dna_fragments(Frags),
        set_of_all(Clone,
                   Id^Frag^
                   (kohara_clone(Id,Clone),
                    member(Frag,Frags),
                    contains(Frag,Clone)
                   ),
                   SequencedClones).
```

We can also construct queries to assess progress in sequencing any chromosome region or clone.

```
Query 17:  List all clones that are greater than 90% sequenced.

% | ?- query17(90,L),member(Clone-PerCent,L),
%      format('~n~3f% sequenced:~n',PerCent),display_object
%     (Clone),fail.
%
% 100.000% sequenced:
% 96594/105700      9107     [110]6F3       (Kohara clone)
%
% 90.136% sequenced:
% 760100/775499    15400     [176]7E10      (Kohara clone)
%
% 100.000% sequenced:
% 4240030/4240714    685     [530B]6G9      (Kohara clone)
%
% 100.000% sequenced:
% 4233865/4240714   6850     [531B]3C5      (Kohara clone)
%
% 93.674% sequenced:
% 4188805/4206684  17880     [534]E11C11    (Kohara clone)
%
% 100.000% sequenced:
% 3936168/3952262  16095     [560]2A1       (Kohara clone)
%
% 93.768% sequenced:
% 3611044/3627299  16256     [613]1B6       (Kohara clone)
%
% 98.882% sequenced:
% 3606153/3617239  11087     [614]5B10      (Kohara clone)
%
% 100.000% sequenced:
% 4240715/4243454   2740     [629B]18C4     (Kohara clone)
%
% 100.000% sequenced:
% 3444102/3447539   3438     [630A]5F1      (Kohara clone)
%
% no

query17(X,ClonesAndPerCent) :-
        set_of_all(Clone-PerCent,
                   Id^A^C^G^T^Ln^
                   (kohara_clone(Id,Clone),
                    kmer_usage([Clone],1,[A,C,G,T]),
                    length_obj(Clone,Ln),
                    PerCent is ((A+C+G+T) / Ln) * 100,
                    PerCent >= X
                   ),
                   ClonesAndPerCent).
```

To keep track of unsequenced regions, we need to identify gaps between known sequence fragments.

```
Query 18: Compute the gaps between sequence fragments.

%
% | ?- query18(Gaps),display_objects(Gaps).
%
% 5933/12279         6347                               (gap)
% 34340/49698       15359                               (gap)
% 54148/62852        8705                               (gap)
% 71729/83533       11805                               (gap)
%          .
%          .
%          .

query18(Gaps) :-
        all_dna_fragments(Frags),
        gaps(Frags,Gaps).
```

Knowing the unsequenced regions in the chromosome, we can now identify the Kohara clones that should be used to complete the sequencing.

```
Query 19: For any unsequenced region, give the Kohara clones
          that overlap the region.



% | ?- query18(Gaps), member(Gap,Gaps),
%      query19(Gap,Clones), display_object(Gap),
%      display_objects(Clones).
%
% 5933/12279      6347                      (gap)
%
% 383/17253        16871     [101]9E4     (Kohara clone)
% 9400/24157       14758     [102]6H3     (Kohara clone)
%
%       .
%       .
%       .

query19(Region,Clones) :-
    set_of_all(Clone,
               Id^
               (kohara_clone(Id,Clone),overlaps(Region,Clone)),
               Clones).
```

One might wish to locate the blocks of unknown sequence that could be determined with relatively small effort.

```
Query 20: Find all gaps between sequenced fragments that are
          less than 700 bp long.

% | ?- query20(L), member(X,L),display_objects(X),nl,fail.
%
% 779858/783702     3845     ECOCYD       (DNA fragment)
% 783703/783891      189                  (gap)
% 783892/788928     5037     tolQecoM     (DNA fragment)
%
%
% 408099/410813     2715     ECOPHOAA     (DNA fragment)
% 410814/411367      554                  (gap)
% 411368/412335      968     ECOPROC      (DNA fragment)
%                .
%                .
%                .
% no

query20(ClonesAndGaps) :-
        all_dna_fragments(L),
        domain(ecoli_genome,Beg,End),
        set_of_all([X,Y,Gap],
                   Ln^
                   (adjacent(X,Y,L),
                    contains(region(Beg,End),X),
                    contains(region(Beg,End),Y),
                    gap(X,Y,Gap),
                    length_obj(Gap,Ln),
                    Ln < 700
                   ),
                   ClonesAndGaps).
```

Given a region bounded by known sequence, one can use "primers" (strings that occur only once in a specified clone) to start the sequencing reaction. The following query identifies the primers that, used in a DNA sequencing reaction, will supply the sequence to "fill in" the gaps identified above.

```
Query 21: Given the output of the last query, find the
          sequencing primers on the counterclockwise and
          clockwise strands that can be used to complete
          the sequence.

% | ?- query21(L),member(X,L),display_closure(X),fail.
% CCW sequencing primer AACACCAGACCCGCGACAAA(410783)
% 408099/410813     2715     ECOPHOAA    (DNA fragment)
%
```

```
% CW sequencing primer GTAACCGCACCGAAGTGGCG(411398)
% 411368/412335       968      ECOPROC     (DNA fragment)
%
% will close the following gap:
% 410814/411367       554                  (gap)
%
% The following clones contain the above gap and primers:
%
% 399200/415299     16100    [142]1A10    (Kohara clone)
% 409727/425480     15754    [143]6A12    (Kohara clone)
% --------------------
% CCW sequencing primer CAACACGGCCACCGGTAGCA(4155544)
% 4151732/4155574    3843     cytRecoM     (DNA fragment)
%
% CW sequencing primer CCTACAAGTTCGTGCAAATT(4156143)
% 4156113/4164654    8542     metJecoM     (DNA fragment)
%
% will close the following gap:
% 4155575/4156112     538                  (gap)
%
% The following clones contain the above gap and primers:
%
% 4146365/4163864   17500    [538]12E3     (Kohara clone)
% --------------------
% CCW sequencing primer CCCTTCGGAGTTTTAGTCAC(3493602)
% 3490087/3493632    3546     tufAecoM     (DNA fragment)
%
% CW sequencing primer TAATGCCCCCATTAAGGTCT(3494112)
% 3494082/3495097    1016     ECOSTR1      (DNA fragment)
%
% will close the following gap:
% 3493633/3494081     449                  (gap)
%
% The following clones contain the above gap and primers:
%
% 3487500/3502699   15200     [626]3F8     (Kohara clone)
% --------------------

% no

query21(GapClosure) :-
        query20(FragsAndGaps),
        set_of_all([Seq1,Pos1,Seq2,Pos2,Frag1,Frag2,Gap,Clones],
                   Id^Clone^MustBeBefore^MustBeAfter^
                   (member([Frag1,Frag2,Gap],FragsAndGaps),
                    kohara_clone(Id,Clone), contains(Clone,Gap),
                  once ccw_primer(Frag1,Clone,Seq1,Pos1),
                  once cw_primer(Frag2,Clone,Seq2,Pos2),
                  MustBeBefore is Pos1-20, MustBeAfter is
                  Pos2+20,clones_that_contain(region
                  (MustBeBefore,MustBeAfter), Clones)
                 ),
                 GapClosure).

ccw_primer(Object,Clone,Seq,CCWpos) :-
        sequence_of(Clone,CloneSeq),
        location(Object,Beg,End),
        Start is End-30,
        pick(CCWpos,Start,Beg),
        subseq_backwards(CCWpos,20,Seq,CloneSeq),
        \+ (subseq_both(Pos,20,Seq,CloneSeq,_), Pos =\= CCWpos).

cw_primer(Object,Clone,Seq,CWpos) :-
        sequence_of(Clone,CloneSeq),
        location(Object,Beg,End),
        Start is Beg+30,
        pick(CWpos,Start,End),
        subseq(CWpos,20,Seq,CloneSeq),
        \+ (subseq_both(Pos,20,Seq,CloneSeq,_), Pos =\= CWpos).

clones_that_contain(Obj,Clones) :-
        set_of_all(Clone,
                Id~(kohara_clone(Id,Clone),contains(Clone,Obj)),
                Clones).

display_closure([Seq1,Pos1,Seq2,Pos2,Frag1,Frag2,Gap,Clones]) :-
        format('CCW sequencing primer ~s(~d)~n',[Seq1,Pos1]),
        display_object(Frag1),nl,
        format('CW sequencing primer ~s(~d)~n',[Seq2,Pos2]),
        display_object(Frag2),nl,
        format('will close the following gap:~n',[]),
        display_object(Gap),nl,
        format('The following clones contain the above gap
            and primers:~n',[]),
        display_objects(Clones),
        format('--------------------~n',[]).
```

This set of example queries has been included to illustrate some of the capabilities of our system. of course, biologists routinely make many more queries. We believe that the set we have chosen accurately reflects the level of effort required to extract a broad range of information.

4 Summary

Although enormous resources are going into the effort of accumulating raw sequence data, no effective means yet exists for allowing a biologist to query the data without employing a computing technician. As the volume of available sequence data increases, and as complete genomes begin to be assembled, the need for flexible access to the data is becoming increasingly acute.

A variety of database technologies can be used to achieve flexible access. We have selected logic programming, and we have implemented a prototype system for answering queries about the *E. coli* genome. This system provides numerous capabilities that are not available under any other system. It allows biologically relevant queries to be answered in small fractions of the time required with more conventional tools.

This system was developed as the initial step toward an environment that supports comparative analysis of chromosomes. It will be extended to provide the database services to support queries relating to several chromosomes. We shall then create user interfaces that make access to the data possible without special-purpose programming. At this point, we have developed one such interface, based on a restricted use of natural language, and we anticipate that other groups will wish to experiment with other such interfaces.

We believe that an approach based on an extension of the work presented in this document offers the most cost-effective strategy for making the benefits of database technology accessible to the biologist. Logic programming, by integrating database queries with ease of computation, creates an appropriate foundation for building user interfaces that will enable biologists to directly pose the questions required to interpret genetic data.

References

[1] Ajioka, J. W.; Smoller, D. A.; Jones, R. W.; Carulli, J. P.; Vellek, A. E. C.; Garza, D.; Linnk, A. J.; Duncan, I. W.; and Hartl, D. L., *Drosophilia genome project: One-hit coverage in yeast artificial chromosomes,* Chromosoma 100: 495–509 (1990)

[2] Adams, M. D.; Kelley, J. M.; Gocayne, J. D.; Dubnick, M.; Polymeropoulos, M. H.; Xiao, H; Merril, C. R.; Wu, A.; Olde, B.; Moreno, R. F.; Kerlavage, A. R.; McCombie, W. R.; and Venter, J. C., *Complementary DNA sequencing: Expressed sequence tags and human genome project*, Science 252: 1651–6 (1991)

[3] Bachmann, B. J., *Linkage map of Escherichia coli K-12, edition 8*, Microbiol. Rev. 54: 130–97 (1990)

[4] Birkenbihl, R. P., and Vielmetter, W., *Cosmid-derived map of E. coli strain BHB2600 in comparison to the map of strain W3110*, Nucleic Acids Res. 17: 5057–69 (1989)

[5] Billings, P. R.; Smith, C. L.; and Cantor, C. R., *New techniques for physical mapping of the human genome*, FASEB J. 5: 28–34 (1991)

[6] Brandriff, B.; Gordon, L.; and Trask, B., *A new system for high-resolution DNA sequence mapping interphase pronuclei*, Genomics 10: 75–82 (1991)

[7] Brewer, B. J., *When polymerases collide: Replication and the transcriptional organization of the E. coli chromosome*, Cell 53: 679–86 (1988)

[8] Branscomb, E.; Slezak, T.; Pae, R.; Galas, D.; Carrano, A. V. ; and Waterman, M., *Optimizing restriction fragment fingerprinting methods for ordering large genomic libraries*, Genomics 8: 351–66 (1990)

[9] Cantor, C. R., *Orchestrating the Human Genome Project*, Science 248: 49–51 (1990)

[10] Carrano, A. V., *Establishing the order of human chromosome-specific DNA fragments*, Basic Life Sci. 46: 37–49 (1988)

[11] Carrano, A. V., et al., *A high-resolution, fluorescence-based, semiautomated method for DNA fingerprinting*, Genomics 4: 129–36 (1989)

[12] Carrano, A. V.; de Jong, P. J.; Branscomb, E.; Slezak, T.; and Watkins, B. W., *Constructing chromosome- and region-specific cosmid maps of the human genome*, Genome 31: 1059–65 (1989)

[13] Coulson, A.; Sulston, J. E.; Brenner, S.; and Karn, J., *Towards a physical map of the genome of the nematode C. elegans*, Proc. Natl. Acad. Sci. U.S.A. 83: 7821–5 (1986)

[14] Coulson, A; Waterston, R.; Kiff, J.; Sulston, J.; and Kohara, Y., *Genome linking with yeast artificial chromosomes*, Nature 335: 184–6 (1988)

[15] Garza, D.; Ajioka, J. W.; Burke, D. T.; and Hartl, D. L., *Mapping the Drosophila genome with yeast artificial chromosomes*, Science 246: 641–6 (1989)

[16] Green, E. D.; and Olson, M. V., *Systematic screening of yeast artificial-chromosome libraries by use of the polymerase chain reaction*, Proc. Natl. Acad. Sci. U.S.A. 87: 1213–7 (1990)

[17] Huang, X. Q.; Hardison, R. C.; and Miller, W., *A space-efficient algorithm for local similarities*, Comput. Appl. Biosci. 6: 373–81 (1990)

[18] Kazic, T.; Michaels, G. S.; Overbeek, R.; Zawada, D.; Dunham, G.; and Rudd, K. E., *An integrated database of E. coli chromosomal information to support queries and rapid prototyping*, AAAI Workshop on Approaches to Classification and Pattern Recognition in Molecular Biology, Anaheim, Calif., July 12, 1991

[19] Kohara, Y.; Akiyama, K.; and Isono, K., The physical map of the whole E. coli chromosome: Application of a new strategy for rapid analysis and sorting of a large genomic library, Cell 50:495–508 (1987)

[20] Komine, Y.; Adachi, T.; Inokuchi, H.; and Ozeki, H., *Genomic organization and physical mapping of the transfer RNA genes in Escherichia coli K12*, J. Mol. Biol. 212: 579–98 (1990)

[21] Love, J. M.; Knight, A. M.; McAleer, M. A.; and Todd, J. A., *Towards construction of a high resolution map of the mouse genome using PCR-analysed microsatellites*, Nucleic Acids Res. 18: 4123–30 (1990)

[22] Link, A. J.; and Olson, M. V., *Physical map of the Saccharomyces cerevisae genome at 110-kilobase resolution*, Genetics 127: 681–98 (1991)

[23] Medigue, C.; Henaut, A.; and Danchin, A., *Escherichia coli molecular genetic map (1000 kbp): Update I*, Mol. Microbiol. 4: 1443–54 (1990)

[24] Michaels, G., Kazic, T.; Overbeek, R.; Zawada, D.; Dunham, G.; Rudd, K.; and Smith, C. L., *Logic programming-based system for querying E. coli chromosomal information*, Cold Spring Harbor Genomic Mapping and Sequencing meeting, May 8–12, 1991

[25] McKusick, V. A., *Current trends in mapping human genes*, FASEB J. 5: 12–20 (1991)

[26] Noda, A.; Courtright, J. B.; Denor, P. F.; Webb, G.; Kohara, Y.; and Ishihama, A., *Rapid identification of specific genes in E. coli by hybridization to membranes containing the ordered set of phage clones*, Biotechniques 10: 474, 476–7 (1991)

[27] Olson, M. V.; Dutchik, J. E.; Graham, M. Y.; Brodeur, G. M.; Helms, C.; Frank, M.; MacCollin, M.; Scheinman, R.; and Frank, T., *Random-clone strategy for genomic restriction mapping in yeast*, Proc. Natl. Acad. Sci. U.S.A 83: 7826–30 (1986)

[28] Olson, M. V.; Hood, L.; Cantor, C.; and Botstein, D., *A common language for physical mapping of the human genome*, Science 245: 1434–40 (1985)

[29] Rudd, K. E.; Miller, W.; Ostell, J.; and Benson, D. A., *Alignment of Escherichia coli K12 DNA sequences to a genomic restriction map*, Nucleic Acids Res. 18: 313–21 (1990)

[30] Rudd, K. E.; Miller, W.; Werner, C.; Ostell, J.; Tolstoshev, C.; and Satterfield, S. G., *Mapping sequenced E. coli genes by computer: software, strategies and examples*, Nucleic Acids Res. 19: 637–47 (1991).

[31] Ruvkun, G.; Ambros, V.; Coulson, A.; Waterston, R.; Sulston, J.; Horvitz, H. R., *Molecular genetics of the Caenorhabditis elegans heterochronic gene lin-14*, Genetics 121: 501–16 (1989)

[32] Pearson, W., *Rapid and sensitive sequence comparison with FASTP and FASTA*, Methods in Enzymology 183: 63–98 (1990)

[33] Siden-Kiamos, I.; Saunders, R. D. C.; Spanos, L.; Majerus, T.; Treanear, J.; and Savakis, C.; Louis, C.; aGlover, D. M.; Ashburner, M.;and Kafatos, F. C., *Towards a physical map of the D. melanogaster genome: Mapping of cosmid clones within defined genomic divisions,* Nucleic Acid Res. 18: 6261–70 (1990

[34] Stephens, J. C.; Cavanaugh, M. L.; Gradie, M. I.; Mador, M. L.; and Kidd, K. K., *Mapping the human genome: Current status*, Science 250: 237–44 (1990)

Appendix: Supported Predicates for Querying the *E. coli* Database

```
adjacent(-Object1,-Object2,+ListOfObjects)
    Object1 and Object2 are adjacent in ListOfObjects (and the
    last element in the list is considered to be adjacent to
    the first)

align_2_seqs(+String1,+String2,-Corr,-Score)
    Align the two lists of ascii DNA characters using a
    Smith-Waterman algorithm.  Corr is set to a list of terms
    of the form P1-P2 where P1 and P2 are displacements
    (integers from 0) into Seq1 and Seq2.

align_two_objects(+Obj1,+Obj2)
    aligns the sequence of Obj1 with that of Obj2 and prints
    the result

aligned_sequences(+String1,+String2,-Score,-Aligned1,-Aligned2)
    This is used to produce aligned versions of Seq1 and Seq2
    (i.e., the aligned sequences that are returned are lists
    of characters that have indels inserted at the appropriate
    locations).

alignment_parameters(-U,-V)
    returns current Smith-Waterman deletion cost parameters
    (mismatch is always -18, and a match is always +18)

all_dna_frag_rsites(-AllDna_FragRsites)
    gets a list of all restriction sites in sequenced
    fragments of DNA

all_dna_fragments(-AllFragments)
    gets a list of all sequenced fragments of DNA

all_genes(-AllGenes)
    gets a list of all the genes

all_known_genes(-AllKnownGenes)
    gets a list of all structural genes and mapped genes

all_kohara_clones(-AllClones)
    gets a list of all of the Kohara clones

all_kohara_rsites(-AllKoharaRsites)
    gets a list of all of the Kohara restriction sites
```

all_mapped_genes(-MappedGenes)
gets a list of unsequenced, but mapped genes

all_translated_genes(-TranslatedGenes)
gets a list of translated genes

amino_acid(?OneCharCode,?ThreeCharCode,?AminoAcid)
table of codes used to represent amino acids

between(+Point1,+Point2,+Point3)
succeeds if Point2 is between Point1 and Point3. This will be the case iff the shortest path on the circular chromosome from Point1 to Point3 goes through Point2

quick_sim(+Seq,+PrintFlag,+MaxMatches,-Matches)
Seq represent a sequence fragment to be quick_simed against the ecoli database. PrintFlag should be 0 or 1 (print). Matches comes back as a list of terms of the form

```
region(FragId,QueryBeg,QueryEnd,FragBeg,FragEnd,Score)
```

bp_to_min(?BasePairs,?Minutes)
converts (using a simple formula) between BasePair coordinates and Minutes on the genetic map

char_stats(+Object,+Size,-CharStats) For a given object (that may or may not have been sequenced), this goes through the sequence cutting it into pieces of length Size. Then it accumulates counts of each of the types of characters (A,C,G,T, and Other) for each interval. The list of CharStats is actually a "list of objects", which means that each interval has a location and can be displayed using display_object/1. Thus, you can get character count statistics and then just display them using display_objects/1. However, the more common use is to feed them into either gc_histogram/1 or gc_histogram_averaged_window/1.

clean_pins(+Pins,-CleanedPins)
Pins must be a list of pairs of the form P1-P2. CleanedPins is set to a list in which "pins" do not cross. Thus, [3-22,4-23,5-17,7-25] would produce [3-22,4-23,7-25] as the "cleaned" pins.

codon(?Char1,?Char2,?Char3,?ThreeCharCode,?OneCharCode)
Table of the genetic code, where Char1-3 are ascii numeric values.

codon_usage(+Objects,-Counts)
Objects is a list of objects. Counts is set to a list of 65 integers. The first is a count of the number of "invalid" codons (i.e., those that contain ambiguous or unsequenced characters). The remaining 64 correspond to the counts of AAA, AAC, AAG, AAT, ACA,...TTT.

common_seq_at_least_k_long(+Objects,+Min,-Seqs)
Locates a sequence that is at least Min long in all Objects and then finds all occurrences in the objects and sets Seqs to the set of occurrences.

common_seq_at_least_k_long_both_strands(+Objects,+Min,-Seqs)
Locates a sequence that is at least Min long in all Objects and then finds all occurrences in the objects and sets Seqs to the set of occurrences (looking at both strands).

common_seqs_at_least_k_long(+Objects,+Min,-SubSeqs)
Computes the set of values reurned by common_seq_at_least_k_long/3.

common_seqs_at_least_k_long_both_strands(+Objects,+Min,-SubSeqs)
Computes the set of values reurned by common_seq_at_least_k_long_both_strands/3.

common_sub_sequence(+SequenceObjects,+Length,-Common,-Positions)
SequenceObjects must be a list of sequence objects (produced by sequence_at/3 or sequence_of/2). Suppose this list has length N. Then Positions will be set to a list of N positions of occurrences of a Common string of the given Length.

common_sub_sequence_both_strands(+SequenceObjects,+Length,
-Common,-Positions)
SequenceObjects must be a list of sequence objects (produced by sequence_at/3 or sequence_of/2). Suppose this list has length N. Then Positions will be set to a list of N positions of occurrences of a Common string of the given Length. The search proceeds by picking a sequence in the "forwards" strand of the first object, and then by taking strings from either strand of the following objects. The positions are either integers (same strand) or i' (for

reverse strand).

compL(+String,?Complement)
produces the Watson-Crick complement of a string. Thus, compL("AACG",X) binds X to "TTGC"

complement(+String,-ReversedComplement)
produces the reversed complement of String. Thus, complement("AACG",X) binds X to "CGTT"

computed_dna_frag_rsite(+LB,+UB,?Beg,?End,-Cuts,+Enzyme)
LB and UB must be the bounds of a sequenced section of DNA. Beg and End are then the beginning and end of a restriction site for the designated enzyme.

computed_restriction_fragment(-Beg,-End,+Enzymes,
-UsedEnzymes,+LB,+UB)
Given bounds LB and UB and a list of restriction Enzymes, find Beg and End that delimit a restriction fragment, and bind UsedEnzymes to a list containing just the two cutting enzymes.

computed_restriction_sites_in_object(+Obj,+Enzymes,-Sites)
returns a list of computed restriction sites from the given set of Enzymes that occur in Obj.

cont_gc_histogram(+Object,+SizeOfWindow)
Given a sequenced Object and a size of a window, produce a histogram with one entry for each position in the object which can be the center of a window. The histogram gives the average GC content of the window.

contains(+ContainingObject,+ContainedObject)
succeeds if the first object contains the second

convergent_genes(-Gene1,-Gene2)
binds Gene1 and Gene2 to convergent genes (which are objects, not IDs)

direction(+Gene,?Direction)
Gene must be a gene, and direction gets bound to clockwise or counterclockwise.

disp_seqs(+Ids,+Strings)
This is used to display a set of sequences that might be over 50 characters long. Thus,

```
disp_seqs([seq1,seq2],[S1,S2])
```

would interleave 50 characters of each sequence in a visual display.

disp_seqs(+Ids,+Strings,+StartingLocations)
like disp_seqs/2, except that the positions of sequences can be specified.

display_object(+Object)
displays an arbitrary object (gene, dna_fragment, sequence object, etc.)

display_objects(+ListOfObjects)
displays a list of objects

dist(+Point1,+Point2,-Distance)
gets the Distance from Point1 to Point2 on the circular chromosome

divergent_genes(-Gene1,-Gene2)
gets two divergent genes (Gene1 and Gene2 are adjacent; Gene1 is expressed ccw and Gene2 cw)

dna_frag_rsite(?Beg,?End,?Enzyme)
Beg and End delimit a site that is matched by the cutting pattern for the designated Enzyme in a sequenced section of the genome

dna_frag_rsite(?Object)
Object is bound to an object representing a DNA fragment restriction site.

dna_fragment(?Id,?Beg,?End)
Id is the ID of a sequenced fragment of the genome beginning at Beg and ending at End

dna_fragment(?Id,?Object)
Id is the ID of a sequenced fragment represented by the object Object.

end_of(+Object,-EndLocation)
Equivalent to location(Object,_,EndLocation) for noncomposite objects. For composite objects, it gives the location of the last piece.

find_pp_match(+Pat,+Gene,-PolyPepTide)

Pat must be an encoding of a pattern to scan for in the translation of Gene. PolyPepTide is bound to a section of the translation that matches. Pat is a list of pattern units. Each unit is one of the following:

1. a string of 1-character amino acid codes, with ? to represent an arbitrary amino acid (e.g., "CP???H"),

2. the alternative of two patterns P1 and P2, which is represented as

```
          P1;P2
```

To illustrate,

```
| ?- gene(thrA,Gene), find_pp_match(["RE?E",("H";"L")],
     Gene,Match), display_object(Match).


2280/2294      15       thrA  (expressed) clockwise
                     RELE L
```

first_n(+List,+N,-ListOfFirstN,-AllButFirstN)

ListOfFirstN is set to be a list of the first N elements of List, and AllButFirstN is bound to a list of the remaining elements in List.

gap(+Object1,+Object2,-Gap)

Gap is bound to an object representing the gap between Object1 and Object2.

gaps(+Objects,-Gaps)

Gaps is bound to a list of any gaps that occur between the objects in the list Objects.

gc_histogram(+CharStats)

writes a histogram of the GC contents of the intervals described in CharStats (produced by char_stats/3).

gc_histogram_averaged_window(+CharStats)

gc_histogram/1 just produces a bar for the GC percentage for each interval, with the bar corresponding to the midpoint of the interval. This looks at adjacent intervals, setting the bar to represent the GC percentage for two adjacent intervals. Thus, there is an overlapping effect.

gene(?Id,?Beg,?End,?Direction)

Beg and End delimit a transcribed section of the genome, where Direction is either counterclockwise or clockwise, giving the direction of transcription.

gene(?Id,?Object)

Object is an object representing the gene with ID Id. This predicate is identical to structural_gene/2. To get only genes that are translated, use translated_gene/2.

genetic_code(?DNA,?AminoAcids)

DNA is a list of Ascii characters representing DNA, and AminoAcids is set to a list of 1-char-codes of the corresponding amino acids produced by translation of the code

group(+ListOfKeyValuePairs,-Groups)

This routine takes a list of sorted key-value pairs and groups them. For example

```
            group([3-a,3-b,4-c,5-a,5-c],X)
```

would bind X to [3-[a,b],4-[c],5-[a,c]]

helix(+StartLoop,+LoopMin,+LoopMax,-Ln,-SizeLoop)

StartLoop specifies a point in the genome. This routine considers all possible helices that could be formed with perfect pairing and loops containing LoopMin to LoopMax characters. Ln is set to the maximum length of the stem of a helix, and SizeLoop gets the size of the loop that produced the maximal stem length.

histogram(+ListOfPairs)

ListOfPairs must be a list of X-Y pairs. A histogram is printed on the terminal to represent the data (one line of asterisks for each pair).

init

an initialization routine that must be run before access to sequence data are made. The routine loads sequences from the file "sequences" into main memory, where C routines access the data.

is_left(+Point1,+Point2)

succeeds if the shortest path from Point2 to Point1 is counterclockwise ("Point1 is to the left of Pcint2")

is_right(+Point1,+Point2)
succeeds if the shortest path from Point2 to Point1 is clockwise ("Point1 is to the right of Point2")

kmer-usage(+Objects,+K,-Counts)
Accumulates a list of K-mer counts. For example,

```
| ?- gene(thrA,G), kmer_usage([G],1,L).

G = gene(thrA,207,2669,clockwise),
L = [0,553,614,692,604]
```

Here, there were

```
0   - invalid 1-mers (ambiguous or unsequenced)
553 - As
614 - Cs
692 - Gs
604 - Ts
```

known_gene(?Id,?Gene)
either a structural gene or a mapped gene

kohara_clone(?Id,?Object)
Object is an object representing the Kohara clone with ID Id.

kohara_clone(?Id,?Beg,?End)
The Kohara clone with ID Id begins at Beg and ends at End.

kohara_enzymes(?Enzymes)
the enzymes that Kohara used to construct his map

kohara_restriction_fragment(-Beg,-End,+Enzymes,-UsedEnzymes)
There is a Kohara restriction fragment from Beg to End bounded by cutting sites for the two enzymes in UsedEnzymes, which are both elements of Enzymes.

kohara_rsite(?Beg,?End,?Enzyme)
Beg and End bound a cutting site for Enzyme in the Kohara map.

kohara_rsite(?Object)

Object represents a Kohara restriction site.

kohara_rsites_in_object(+Object,-Rsites)
binds Rsites to the list of Kohara restriction sites that occur in Object

length_obj(+Object,-Ln)
Ln is the length of Object.

length_objects(+Objects,-Ln)
binds Ln to the sum of the lengths of the objects in the list Objects

location(+Object,?Beg,?End)
Object has a piece that begins at Beg and ends at End. Normally, objects are not composite, so this succeeds just once. However, for composite objects, it will succeed multiple times.

longest_common_subseq(+Seqs,-Common,-Positions)
Seqs must be a list of sequence objects (produced by sequence_at/3 and sequence_of/2). Suppose that the length of this list is N. Then, Common string and Positions are bound to a set of N unique positions (each from the corresponding sequence object). Thus,

```
longest_common_subseq([Prefix,Gene,Gene],Common,[P1,P2,P3])
```

would find the longest sequence that occurred in Prefix and twice in Gene. P1 would get the occurrence in Prefix. This call is determinate.

map_restriction_fragments(+Object,+Enzymes,-Map)
produces a list of restriction fragments (which are objects) which would be formed by Enzymes cutting Object. One can display the map using display_objects/1. Object must be sequenced.

mapped_gene(?Id,-Gene)
used to access genes that have been mapped, but not sequenced

mapped_gene(?Id,?Mapper,?Dir,?MapLoc,?BasePair)
Mapper is the name of the person who did the map (e.g., 'Bach.' for Barbara Bachmann); Dir is 'clockwise', 'counterclockwise', or 'unknown'; MapLoc is the

location on the map, using whatever units the Mapper gave; BasePair is the location on the chromosome that we computed by converting the MapLoc.

match(+Pattern,+String)

If Pattern is a string that may contain ambiguous characters (Ns, Rs, Ys, etc.) and String is a string of DNA, then this succeeds if each character in the pattern matches the corresponding character in the string. An ambiguous character in the pattern matches the appropriate values in the string. On the other hand, an ambiguous character in the string will match only that exact character in the pattern (preventing a string of Ns in the string from matching every restriction enzyme).

maxL(+List,-Maximum)

Maximum is the maximum element in List.

max_match(+Pattern,+String,-Matched)

Matched is set to the maximum number of characters that the pattern matches the string.

minL(+List,-Minimum)

Minimum is the minimum value in List.

minutes_to_bp(+Min,-Bp)

converts a coordinate given in minutes on the Bachmann genetic map to a base pair location (by interpolation between points that occur on both the genetic and physical maps).

on_circular_chromosome(+X,-XonChrom)

XonChrom is X modulo the length of the chromosome.

once(+Goal)

allows a single solution of Goal

overlaps(+Object1,+Object2)

succeeds iff Object1 overlaps Object2

overlaps(+Object1,+Object2,-OvBeg,-OvEnd)

like overlaps/2, except that the region of overlap is returned

pick(-X,+StartOfRange,+EndOfRange)

This clause allows you to pick a value of X in the range StartOfRange to EndOfRange. The values may be ascending or descending.

polypeptide(?Id,?PolyPepTide)

used to access translations of structural genes that code for proteins

polypeptide(?Id,?Beg,?End,?Dir,?AAs)

For the translated gene given by translated_gene (Id,Beg,End,Dir), AAs is a list of "chunks of the polypeptide", where each chunk is a list of the 1-character amino acid codes. This predicate always returns AAs as a list of one element, which is the translation of the region Beg/End. Other routines occasionally return the translation broken into sublists; these are separated by a space when the string is displayed.

print_codon_usage(+Counts)

displays the meaning of the 65 integers in the list Counts. For example,

```
| ?- gene(thrA,G),codon_usage([G],L),print_codon_usage(L).
number valid codons = 821
number invalid codons = 0

          alanine: 92         11.21%
              GCA: 15         1.83%
              GCC: 36         4.38%
              GCG: 27         3.29%
              GCT: 14         1.71%

          arginine: 47        5.72%
              AGA: 0          0.00%
              AGG: 2          0.24%
              CGA: 3          0.37%
              CGC: 19         2.31%
              CGG: 5          0.61%
              CGT: 18         2.19%

          asparagine: 40      4.87%
              AAC: 18         2.19%
              AAT: 22         2.68%
                .
                .
                .
```

print_gc_content(+Counts)

displays GC content represented by Counts returned by kmer_usage/2. For example,

```
  | ?- gene(thrA,G), kmer_usage([G],1,L), print_gc_content(L).
  invalid bases: 0
        Gs, Cs: 1306      53.02%
        As, Ts: 1157      46.98%

  G = gene(thrA,207,2669,clockwise),
  L = [0,553,614,692,604]
```

print_kmer_usage(+Counts,+K)

displays the Counts returned by kmer_usage/2. For example,

```
  | ?- gene(thrA,G), kmer_usage([G],1,L), print_kmer_usage(L,1).
  invalid 1mers: 0

        A: 553                   22.45%
        C: 614                   24.93%
        G: 692                   28.10%
        T: 604                   24.52%

       G = gene(thrA,207,2669,clockwise),
       L = [0,553,614,692,604]
```

restriction_site(+Enzyme,-Pattern,-DisplacementToCut)

returns the pattern and position of the cut for a specified restriction enzyme

restriction_sites_in_object(+Obj,+Enzymes,-Sites)

returns a list of restriction sites (both computed and Kohara sites) from the given set of Enzymes that occur in Obj. To get just the computed restriction sites, use computed_restriction_sites_in_object/3.

scan_mem_for_pat(+Pattern,+Beg,+End,-Matches)

To scan a section of the chromosome for the occurrence of a pattern, one uses the routine scan_mem_for_pattern_occurrence/4:

```
   | ?- gene(aceE,Gene),start_of(Gene,Beg),end_of(Gene,End),
        scan_mem_for_pattern_occurrence(Beg,End,
            [pvar(p1,dna("RYRYRY")),
             ellipses(0,400),
             repeat(p1,1,1,0)],Occ),
             display_object(Occ).

123436/123464:            sequence
                123436    GCGTGC TCAGTATCTGATCGACCA ACTGC

Gene = gene(aceE,123344,126004,clockwise),
Beg = 123344,
End = 126004,
Occ = seq(123436,123464,spaces([123442,123460]))
```

sequence_at(+Beg,+End,-SequenceObject)
produces a sequence object representing the section of the genome from Beg to End.

sequence_of(+Object,-SequenceObject)
produces a sequence object representing the sequence of a given object.

sequenced(+Object)
succeeds if Object has been entirely sequenced

set_sw_parameters(+U,+V)
set insertion costs for the Smith-Waterman alignment algorithm. Mismatches cost -18; matches have a similarity of +18. Insertion of n indels costs -(U + nV).

set_sw_parameters(U,V)
sets the costs of insertions for the Smith-Waterman algorithm. "Identical matches" are worth 18 points of similarity for DNA/RNA. The cost of a k-indel insertion is U+kV. Default settings for the DNA/RNA alphabet (which is the default alphabet) are U=0, V=18.

similarity_search(+String1,+Id1,+String2,+Id2,+MS,+Q,+R,+K,
+Print,-Sim)
This predicate invokes the similarity search generously contributed by Xiaoqiu Huang and Webb Miller. Seq1 and Seq2 are lists of ascii characters. Id1 and Id2 are atoms. MS, Q, R, and K are as described above. Print_flag == yes -> write out the report of similarities; anything else will suppress printing. Similarities are bound to a list in which each element is of the form

```
similarity(Score,NumCharMatched,LengthOfAlignmentWithIndels,
           NumberMisMatches,Start1,End1,Start2,End2)

     Here is a little example:

     :- similarity_search("aaaaaaaaacccccccccggggggggg",seq1,
                          "ccccaaccccccaaaaacccc",seq2,
                          -1.0,2.2,0.2,2,yes,Similarities).

     produces the following output:
         ==================================

Match   Mismatch   Gap-Open Penalty   Gap-Extension Penalty
  1.0     -1.0            2.1                 0.1

                 Upper Sequence : seq1
                         Length : 27
                 Lower Sequence : seq2
                         Length : 20

*********************************************************
      Number 1 Local Alignment
      Similarity Score : 9
      Match Percentage : 100%
      Number of Matches : 9
      Number of Mismatches : 0
      Total Length of Gaps : 0
      Begins at (5, 12) and Ends at (13, 20)

    0      .
    5 aaaaacccc
      |||||||||
   12 aaaaacccc

*********************************************************
      Number 2 Local Alignment
      Similarity Score : 8.4
      Match Percentage : 68%
      Number of Matches : 11
      Number of Mismatches : 0
      Total Length of Gaps : 5
      Begins at (8, 5) and Ends at (18, 20)

    0      .    :    .
    8 aaccccc      cccc
```

```
       |||||||-----||||
      5 aacccccaaaaacccc

  X = [similarity(90,9,9,0,5,13,12,20),similarity(84,11,
      16,0,8,18,5,20)]
```

similarity_search(+String1,+String2)

runs the local similarity search and displays the best 5 alignments

sites_in_object(+Object,-Sites)

Sites is set to a list of objects representing "interesting sites" that occur in Object. You can use display_objects/1 to display the objects.

sites_in_object_both(+Object,-Sites)

Sites is set to a list of objects representing "interesting sites" that occur in Object, looking at both strands.

start_of(+Object,-StartingLocation)

equivalent to location(Object,StartingLocation,_) for non-composite objects. For composite objects, it gives the location of the first piece.

sub_list(+Pattern,+String,-LocOfMatch)

finds a location in String (location values start from 1) for which Pattern matches.

sub_seq(+Position,+Ln,?String)

a predicate that takes some of the pain out of invoking subseq/4. Position is an expression that gets evaluated. Then, String is set to the Ln characters that occur at that position (on the clockwise strand) at that location.

subseq(?Position,?Length,?String,+SequenceObject)

as described in the tutorial

subseq_backwards(?Position,?Length,?String,+SequenceObject)

as described in the tutorial

subseq_both(?Position,?Length,?String,+SequenceObject,-Direction)

as described in the tutorial

subseqs_in_obj(+Object,+String,-Positions)

binds Positions to a list of all occurrences of String

```
    in the Object (which does not have to be a sequence object).
    This predicate fails if there are no occurrences.

sum_gaps(+ListOfGaps,-Sum)
    ListOfGaps must be a list of gap objects.  Sum is bound to
    the sum of the lengths of the gaps.

trans_to_polypeptide(+Beg,+End,+Dir,-AAs)
    translates the DNA string in the region Beg/End in the
    direction given by Dir, setting AAs to the list of
    1-character amino acid codes

translated_gene(?Id,?Object)
    Object is an object representing the gene with ID Id.
    Furthermore, the gene has a length that is a multiple
    of 3, and it begins with ATG or GTG and terminates with
    TGA, TAA, or TAG.

unique(+Beg,+End)
    succeeds if the region Beg/End has been sequenced, and
    if the value occurs just once.

write_list(+List)
    displays the list of Prolog terms
```

Distribution for ANL-92/1

Internal:

- E. A. Baehr
- J. M. Beumer (20)
- F. Y. Fradin
- R. T. Hagstrom
- D. Joerg
- H. Matsuda
- R. A. Overbeek (50)
- G. W. Pieper
- R. L. Stevens
- D. P. Weber
- C. L. Wilkinson
- D. G. Zawada

- ANL Patent Department
- ANL Contract File
- TIS Files (3)

Erternal:

- 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
- G. Dunham, National Institutes of Health
- A. Ginsburg, Washington University
- F. Howes, Department of Energy - Energy Research
- T. Kazic, Washington University
- G. Michaels, National Institutes of Health
- K. Rudd, National Institutes of Health
- C. Smith, University of California and LBL, Berkeley, Calif.
- R. Taylor, National Institutes of Health
- K. Yoshida, University of California and LBL, Berkeley, Calif.