ANL/MCS-TM--173

DE93 007270

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

ANL/MCS-TM-173

Accessing Integrated Genomic Data Using GenoBase: A Tutorial. Part 1

by

Ross Overbeek and Morgan Price

Mathematics and Computer Science Division

Technical Memorandum No. 173

January 1993

This work was supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38.

Contents

Accessing Integrated Genomic Data Using GenoBase: A Tutorial. Part 1

by

Ross Overbeek and Morgan Price

Abstract

GenoBase integrates genomic information from many existing databases, offering convenient access to the curated data. This document is the first part of a two-part tutorial on how to use GenoBase for accessing integrated genomic data.

1 Introduction

GenoBase is a database that integrates information from a number of existing databases. Enormous work has gone into developing many, many carefully curated databases that contain information relating to genomic sequences. Now, there are a number of efforts taking place around the world attempting to offer integrated access to this growing body of valuable information. GenoBase is one of these projects. Our goal in developing the system is simply to offer more convenient access to the curated data; as such, it builds directly on the efforts of many individuals. We will not attempt to list them all, but we will try to mention the individuals and groups that have developed the databases that we utilize, as we cover specific categories of data in later chapters.

GenoBase is an object-oriented database. By this, we mean that the user should think of GenoBase as containing information about *objects*. Objects have *attributes*. All of the objects that we encounter in the initial stages of this tutorial will be *typed objects*; that is, each object will have a *type* that categorizes the object (e.g., we will have objects of type *sequence_fragment, cds, peptide, enzyme*, and so forth).

GenoBase has been implemented using the logic programming language Prolog, and many of the details of how to use the system will reflect this fact. We believe that a modest knowledge of Prolog is probably useful for most scientists, and we would encourage you

study the language at some point; however, we will attempt to present the system in a fashion that will not require you to understand Prolog at all.

We intend to eventually offer versions of GenoBase on PCs, and we will certainly integrate it into a graphics interface (our current plan is to use GDE, a freely distributed X-windows based system). However, for now we believe that many users will be able to start making effective use of the system using our current, relatively primitive interface. Initially, our recommendation is that you run GenoBase from within an emacs window on a Sun workstation; this is not absolutely required, but it is the framework that we use and the one which will be discussed within this tutorial.

This report constitutes the first of two that develop the basic environment supported by GenoBase.

1.1 Getting Started

Our database is a collection of objects, each of which has some set of attributes. The database offers you the ability to find objetcs which meet specified criteria, operate on the objects, and display the results. To bring up the databases initially you need to position yourself at the appropriate "home directory", start Prolog, and initiate the database. On our current system this would be done using

```
cd ~/Bacterial/Version3
prolog
[startup].
```

The startup procedure will prompt you to ask which aspects of the environment you wish access to (it does take as much as 10–20 minutes to get the entire environment initialized, so sometimes we do not load the entire database). If you simply take the default settings, you will eventually reach the point where the system prompts you with

```
?-
```

and you are ready to begin.

Well, what can you do now? The first step in becoming familiar with GenoBase will be to explore the different types of objects maintained within the database and the relationships between them. Let's start by just looking at the data associated with the enzyme 4.1.3.1:

```
| ?- eval(obj([type=enzyme,name='4.1.3.1'])).

['4.1.3.1',enzyme]
```

We show what the user typed in with boldface. The system response follows. There are a number of things to be learned from the previous request. First, the basic structure of such a request is

```
eval(TermToBeEvaluated)
```

where

```
obj([type=enzyme,name='4.1.3.1'])
```

is the term that gets evaluated. This type of term stands for "give me an object with type *enzyme* that has the name 4.1.3.1". The word 'enzyme' and the name '4.1.3.1' are *atoms*. If an atom begins with a lowercase letter and contains no special characters, it need not be enclosed by apostrophes; else, they are necessary.

Next, note that we got back only a short term identifying the object. To get a complete version of the attributes, you would ask that the expression be evaluated and the result "pretty-printed":

```
| ?- evalpp(obj([type=enzyme,name='4.1.3.1'])).

enzyme 4.1.3.1
    desc('ISOCITRATE LYASE')
    alternate_names: [ISOCITRASE,ISOCITRITASE,ISOCITRATASE,ICL]
    catalytic_activity:
      [1]: ISOCITRATE = SUCCINATE + GLYOXYLATE.

    Comments:
  1. THE ISOMER OF ISOCITRATE INVOLVED IS (1R,2S)-1-HYDROXYPROPANE-1,2,3-
TRICARBOXYLATE.

===
```

Objects like *enzyme 4.1.3.1* also relate to other objects (which have been extracted from a number of other databases). If you wished to see these relationships, you would ask for

the object to be pretty-printed, along with the relationships:

```
| ?- evalppr(obj([type=enzyme,name='4.1.3.1'])).
```

```
enzyme 4.1.3.1
    desc('ISOCITRATE LYASE')
    alternate_names: [ISOCITRASE,ISOCITRITASE,ISOCITRATASE,ICL]
    catalytic_activity:
      [1]: ISOCITRATE = SUCCINATE + GLYOXYLATE.

    Comments:
  1. THE ISOMER OF ISOCITRATE INVOLVED IS (1R,2S)-1-HYDROXYPROPANE-1,2,3-
TRICARBOXYLATE.

==============<< Related To >==============
enzyme_to_cds    -->    [aceA,cds,'E.coli']
enzyme_to_peptide    -->    ['P25248',peptide]
enzyme_to_peptide    -->    ['P20014',peptide]
enzyme_to_peptide    -->    ['P05313',peptide]
enzyme_to_peptide    -->    ['P17069',peptide]
enzyme_to_peptide    -->    ['P20699',peptide]
enzyme_to_peptide    -->    ['P15479',peptide]
===
```

Finally, you could ask for a pretty-print of the object, its relationships, and a pretty-print of all objects that it relates to (which can produce a prodigious amount of output) using

```
| ?- evalpprpp(obj([type=enzyme,name='4.1.3.1'])).
```

```
enzyme 4.1.3.1
    desc('ISOCITRATE LYASE')
    alternate_names: [ISOCITRASE,ISOCITRITASE,ISOCITRATASE,ICL]
    catalytic_activity:
      [1]: ISOCITRATE = SUCCINATE + GLYOXYLATE.

    Comments:
  1. THE ISOMER OF ISOCITRATE INVOLVED IS (1R,2S)-1-HYDROXYPROPANE-1,2,3-
TRICARBOXYLATE.

==============<< Related To >==============
enzyme_to_cds    -->    cds aceA of E.coli
    accession('EG10022')
```

```
    desc("isocitrate lyase; utilization of acetate")
    swissprot('P05313')
-------------------------
enzyme_to_peptide   -->   peptide P25248
    id('ACEA_BRANA')
    desc('ISOCITRATE LYASE (EC 4.1.3.1) (ISOCITRASE) (ISOCITRATASE) (ICL)')
    data_class('STANDARD')
    species('BRASSICA NAPUS (RAPE)')
    classification: [EUKARYOTA,PLANTA,EMBRYOPHYTA,ANGIOSPERMAE,DICOTYLEDONEAE,
        CAPPARALES,CRUCIFERAE]
Features:
    SITE              574  576  GLYOXYSOMAL SORTING SEQUENCE (POTENTIAL).

    peptide sequence of length 576:
        0 MAASFSVPSM IMEEEGRFEA EVAEVQTWWS SERFKLTRRP YTARDVVALR
       50 GHLKQGYASN EMAKKLWRTL KSHQANGTAS RTFGALDPVQ VTMMAKHLDT
      100 IYVSGWQCSS THTSTNEPGP DLADYPYDTV PNKVEHLFFA QQYHDRKQRE
      150 ARMSMSREER AKTPFVDYLK PIIADGDTGF GGTTATVKLC KLFVERGAAG
      200 VHIEDQSSVT KKCGHMAGKV LVAVSEHINR LVAARLQFDV MGTETVLVAR
      250 TDAVAATLIQ SNIDSRDHQF ILGVTNPSLR GKSLSSLLAE GMAVGNNGPA
      300 LQAIEDQWLS SARLMTFSDA VVEALKRMNL SENEKSRRVN EWLNHARYEN
      350 CLSNEQGREL AAKLGVTDLF WDWDLPRTRE GFYRFQGSVT AAVVRGWAFA
      400 QIADLIWMET ASPDLNECTQ FAEGVKSKTP EVMLAYNLSP SFNWDASGMT
      450 DQQMMEFIPR IARLGYCWQF ITLAGFHADA LVVDTFAKDY ARRGMLAYVE
      500 RIQREERSNG VDTLAHQKWS GANYYDRYLK TVQGGISSTA AMGKGVTEEQ
      550 FKETWTRPGA AGMGEGTSLV VAKSRM
    mol_wt(64325)
    Comments:
  1. FUNCTION: INVOLVED IN STORAGE LIPID MOBILIZATION DURING THE GROWTH
OF HIGHER PLANT SEEDLING.
  2. CATALYTIC ACTIVITY: ISOCITRATE = SUCCINATE + GLYOXYLATE.
  3. PATHWAY: FIRST STEP IN GLYOXYLATE BYPASS, AN ALTERNATIVE TO THE
TRICARBOXYLIC ACID CYCLE (IN BACTERIA AND PLANTS).
  4. SUBUNIT: HOMOTETRAMER.
  5. SUBCELLULAR LOCATION: GLYOXYSOME.
  6. SIMILARITY: TO THE BACTERIAL AND FUNGAL ENZYME.

   01-MAY-1992:     (REL. 22, CREATED)
   01-MAY-1992:     (REL. 22, LAST SEQUENCE UPDATE)
   01-MAY-1992:     (REL. 22, LAST ANNOTATION UPDATE)

    keywords: [[GLYOXYLATE BYPASS,TRICARBOXYLIC ACID CYCLE,LYASE,GLYOXYSOME]]
    References:
      [1]: PLANT CELL 1:293-300(1989).
           COMAI L., DIETRICH R.A., MASLYAR D.J., BADEN C.S., HARADA J.J.;


Cross reference to PIR [JQ1105,JQ1105]
```

```
Cross reference to PROSITE [PS00161,ISOCITRATE_LYASE]
Cross reference to PROSITE [PS00342,PEROXISOMAL]

--------------------------
.
.
.
===
```

Here, we deleted most of the output, since it displays a complete version of all of the related peptides. You need not get at the related objects in this "shotgun" fashion; rather, you might just use *evalppr/1* (which means the version *evalppr* that takes one argument) to get a listing of the relationships, and then pursue individual related objects with more queries:

```
| ?- evalppr(obj([type=enzyme,name='4.1.3.1'])).


enzyme 4.1.3.1
    desc('ISOCITRATE LYASE')
    alternate_names: [ISOCITRASE,ISOCITRITASE,ISOCITRATASE,ICL]
    catalytic_activity:
      [1]: ISOCITRATE = SUCCINATE + GLYOXYLATE.

    Comments:
  1. THE ISOMER OF ISOCITRATE INVOLVED IS (1R,2S)-1-HYDROXYPROPANE-1,2,3-
TRICARBOXYLATE.

==============<< Related To >==============
enzyme_to_cds   -->   [aceA,cds,'E.coli']
enzyme_to_peptide   -->   ['P25248',peptide]
enzyme_to_peptide   -->   ['P20014',peptide]
enzyme_to_peptide   -->   ['P05313',peptide]
enzyme_to_peptide   -->   ['P17069',peptide]
enzyme_to_peptide   -->   ['P20699',peptide]
enzyme_to_peptide   -->   ['P15479',peptide]
===
```

```
| ?- evalppr(obj([type=peptide,name='P05313'])).


peptide P05313
    id('ACEA_ECOLI')
    desc('ISOCITRATE LYASE (EC 4.1.3.1) (ISOCITRASE) (ISOCITRATASE)')
```

```
    data_class('STANDARD')
    species('ESCHERICHIA COLI')
    gene name(s): ACEA
    classification: [PROKARYOTA,GRACILICUTES,SCOTOBACTERIA,
     FACULTATIVELYANAEROBICRODS,ENTEROBACTERIACEAE]
Features:
    CONFLICT          101  117  LAASMYPDQSLYPANSV -> WRPACIRISRSIRQTRC
       (IN REF. 2).
    CONFLICT          215  215  A -> P (IN REF. 2).
    CONFLICT          338  338  Q -> E (IN REF. 2).

    peptide sequence of length 434:
        0 MKTRTQQIEE LQKEWTQPRW EGITRPYSAE DVVKLRGSVN PECTLAQLGA
       50 AKMWRLLHGE SKKGYINSLG ALTGGQALQQ AKAGIEAVYL SGWQVAADAN
      100 LAASMYPDQS LYPANSVPAV VERINNTFRR ADQIQWSAGI EPGDPRYVDY
      150 FLPIVADAEA GFGGVLNAFE LMKAMIEAGA AAVHFEDQLA SVKKCGHMGG
      200 KVLVPTQEAI QKLVAARLAA DVTGVPTLLV ARTDADAADL ITSDCDPYDS
      250 EFITGERTSE GFFRTHAGIE QAISRGLAYA PYADLVWCET STPDLELARR
      300 FAQAIHAKYP GKLLAYNCSP SFNWQKNLDD KTIASFQQQL SDMGYKFQFI
      350 TLAGIHSMWF NMFDLANAYA QGEGMKHYVE KVQQPEFAAA KDGYTFVSHQ
      400 QEVGTGYFDK VTTIIQGGTS SVTALTGSTE ESQF
    mol_wt(47521)
    Comments:
  1. CATALYTIC ACTIVITY: ISOCITRATE = SUCCINATE + GLYOXYLATE.
  2. PATHWAY: FIRST STEP IN GLYOXYLATE BYPASS, AN ALTERNATIVE TO THE
TRICARBOXYLIC ACID CYCLE (IN BACTERIA AND PLANTS).
  3. SUBUNIT: HOMOTETRAMER.
  4. SUBCELLULAR LOCATION: CYTOPLASMIC.
  5. SIMILARITY: TO THE PLANT AND FUNGAL ENZYME.

   01-NOV-1988:     (REL. 09, CREATED)
   01-NOV-1988:     (REL. 09, LAST SEQUENCE UPDATE)
   01-FEB-1991:     (REL. 17, LAST ANNOTATION UPDATE)

    keywords: [[GLYOXYLATE BYPASS,TRICARBOXYLIC ACID CYCLE,LYASE]]
    References:
      [1]: NUCLEIC ACIDS RES. 16:10924-10924(1988).
           BYRNE C.R., STOKES H.W., WARD K.A.;
            STRAIN=K12;
           medline=89083515

      [2]: NUCLEIC ACIDS RES. 16:5689-5689(1988).
           RIEUL C., BLEICHER F., DUCLOS B., CORTAY J.-C., COZZONE A.J.;
            STRAIN=K12;
           medline=88262573

      [3]: J. BACTERIOL. 170:2763-2769(1988).
           KLUMPP D.J., PLANK D.W., BOWDIN L.J., STUELAND C.S., CHUNG T.,
```

```
   LAPORTE D.C.;
          medline=88227861


Cross reference to EMBL [X12431,ECACEB]
Cross reference to EMBL [X07543,ECACEA]
Cross reference to EMBL [M20714,ECIDHKPA]
Cross reference to PIR [S00931,WZECIC]
Cross reference to PIR [S05692,S05692]
Cross reference to ECOGENE [EG10022,ACEA]
Cross reference to PROSITE [PS00161,ISOCITRATE_LYASE]

==============<< Related To >==============
peptide_to_cds    -->    [aceA,cds,'E.coli']
peptide_to_enzyme    -->    ['4.1.3.1',enzyme]
peptide_to_prosite('F')    -->    ['PS00030',prosite]
peptide_to_prosite('T')    -->    ['PS00161',prosite]
===
```

| ?- evalppr(obj([type=cds,name=aceA])).

```
cds aceA of E.coli
    accession('EG10022')
    desc("isocitrate lyase; utilization of acetate")
    swissprot('P05313')
==============<< Related To >==============
object_to_piece(1,14191,15495,direct)    -->
[hydGecoM,sequence_fragment,'E.coli']
cds_to_enzyme    -->    ['4.1.3.1',enzyme]
gene_to_map(90.942)    -->    ['Bachmann',map,'E.coli']
cds_to_peptide    -->    ['P05313',peptide]
===
```

One small detail is worth noting in this list set of queries: some of the relationships included data fields. Note the following lines (where the first two appeared as relationships from *peptide* P05313, and the last from *cds* aceA):

```
peptide_to_prosite('F')    -->    ['PS00030',prosite]
peptide_to_prosite('T')    -->    ['PS00161',prosite]
gene_to_map(90.942)    -->    ['Bachmann',map,'E.coli']
```

The meanings of the data fields in relationships depend on the particular relationship. The field in a *peptide_to_prosite* relationship indicates whether or not the occurrence of the prosite pattern in the peptide is a "real positive" or a "false positive", and the field in the *gene_to_map* relationship gives the position of the gene in the corresponding map.

Here, we have started exposing you to some of the basic types of objects included in the system:

- *genome* objects represent an entire genome (and relate to the chromosomes, plasmids, etc.) for specific orgainisms.
- *chromosome* objects represent specific chromosomes for specific organisms.
- *sequence_fragment* objects represent a section of DNA sequence that has been captured. This data comes from GenBank.
- *enzyme* objects represent an "abstract enzyme" in they can relate to many distinct peptides and genes (from many organisms). The data associated with this type of object comes from the Enzyme Data Bank created by Amos Bairoch.
- *peptide* objects represent specific peptide sequences, and most of this data currently comes from the Swiss Protein Data Bank (again, developed by Amos Bairoch) and the Protein Identification Resource (PIR).
- *prosite* objects represent the peptide motifs compiled by Amos Bairoch.
- Objects of type *cds, rRNA, tRNA,* and *misc_RNA* represent specific genes (from specific organisms). Much of the data associated with these types of objects comes directly from GenBank.
- *map* objects are used to represent physical or genetic maps (and most of the information one gets from accessing these objects will be through relationships to the objects contained in the map). The system includes a map of the *E. coli* chromosome published by Barbara Bachmann, as well as maps of several other bacterial organisms.
- Objects of type *eco2dbase* capture the data provided by Fred Neidhardt's project to develop data relating to expression of *E. coli* genes.
- *rebase_entry* objects describe the sites cut by restriction enzymes. This data comes from the database distributed by Rich Roberts.
- *pdb_entry* objects contain data relating to the coordinates of atoms within a specific peptide for which the crystal structure has been determined. This data has been extracted from the Protein Data Base distributed by Brookhaven National Laboratory.
- *peptide_alignment* objects contain alignments of peptides. Most of the currently available alignments were acquired from the Protein Identification Resource.
- *nucleotide_alignment* objects contain alignments of DNA or RNA sequences. These have come from several sources including those distributed by the Ribosomal Database Project and the Berlin Data Bank.

- Objects of type *compound* and *reaction* are used to encode the reactions in metabolic pathways. The compound information is largely from Peter Karp's database, and the metabolic pathway information was assembled by Murali Raju (who started from a set of reactions provided by Ray Ochs).

This is only a partial list, and we find that we add new object types frequently, since the body of curated data is expanding so rapidly. If at any point you wish to know what types of objects are accessible, you can use

```
eval(help(types))
```

to get help with exactly what is being represented by a particular type, you would use

```
eval(help(SomeType))
```

where *SomeType* is a specific type (like *eco2dbase* or *cds*) to get a short summary of what a given type of object represents. To see what topics help is available for, use

```
eval(help(help))
```

That is, ask for help about *help*!

There are some details that need to be explained about expression evaluation. First, an expression can often be successfully evaluated to any of a set of acceptable objects. Thus,

```
| ?- evalpp(obj([type=cds,name=gap])).

cds gap of E.coli
    desc("glyceraldehyde-3-phosphate dehydrogenase")
=== ;
cds gap of Salmonella
    codon_start(1)
    product("glyceraldehyde-3-phosphate dehydrogenase")
=== ;

yes
| ?-
```

if you were to run the preceding query, GenoBase would respond with one acceptable object (*gap* of *E.coli*) and then pause waiting for you to respond. If you were to hit a carriage return, that would end the query (and the system would reply with "yes", meaning that it had successfully processed your query); on the other hand, responding with a semicolon causes GenoBase to attempt to produce an alternative response. We have glossed this aspect of the interaction over in our previous discussion – the system always pauses after presenting an answer, and the user responds indicating whether or not GenoBase should attempt to find alternative solutions.

If you had wanted only the gene in *Salmonella*, the appropriate expression would have been

```
| ?- evalpp(obj([type=cds,name=gap,genome='Salmonella'])).
```

Sometimes, you will wish GenoBase to collect an entire set of objects; in this case, you should use the *all/1* operator:

```
| ?- evalpp(all(obj([type=cds,name=gap]))).

[
cds gap of E.coli
    desc("glyceraldehyde-3-phosphate dehydrogenase")
===,
cds gap of Salmonella
    codon_start(1)
    product("glyceraldehyde-3-phosphate dehydrogenase")
===
] ;
```

You may well find that typing *evalpp* becomes a bit tiring. To ease the pain slightly, you can simply type it once, and GenoBase will prompt you for objects to be evaluated:

```
| ?- evalpp.

|: obj([type=compound]).

compound (-)o-acetylcarnitine
    Stochiometry C9H17N1O4
    Molecular Weight = 203.238
    Sources: ['AEPCO','BOEMAN']
    Generalizes to ['COMPOUNDS']
```

```
    Built from ['CARNITINE']
===
```

```
|: obj([type=rebase_entry]).
```

```
rebase_entry AaaI
    organism('Acetobacter aceti ss aceti')
    enzyme_type([
'R2'
])
    sites([
cuts_at("CGGCCG",1)
])
    References:
    references("[1]
Tagami H., Tayama K., Tohyama T., Fukaya M., Okumura H., Kawamura Y.,
Horinouchi S., Beppu T.;
FEMS Microbiol. Lett. 56:161-166(1988).
")
===
```

```
|: all(obj([type=misc_RNA,genome='E.coli'])).
```

```
[
misc_RNA ssrA of E.coli
    accession('EG30100')
===,
misc_RNA ffs of E.coli
    accession('EG30027')
    desc("4.5S RNA")
===,
misc_RNA micF of E.coli
    accession('EG30063')
    desc("regulatory antisense RNA affecting ompF expression")
===,
misc_RNA rnpB of E.coli
    accession('EG30069')
    desc("RNase P; RNA subunit; M1 RNA")
===,
misc_RNA spf of E.coli
    accession('EG30098')
    desc("Spot 42 RNA")
===,
```

```
misc_RNA ssr of E.coli
    accession('EG30099')
    desc("Stable 6S RNA")
===
]
```

|: quit.

```
yes
| ?-
```

Note that you end your sequence of prompts with *quit.* If you find that you wish to see all of the solutions, rather than typing in a semicolon at the pause, you can just type in the character 'a' and GenoBase will automatically display all of the alternatives.

Just as *evalpp/0* is used to process a sequence of evaluations, pretty-printing the results, versions of *eval/0, evalppr/0,* and *evalpprpp/0* also exit.

Now, let us proceed to the topic of traversing relationships between objects. Suppose, that you knew that the *E.coli* gene *gap* were related to an *eco2dbase* object. Then, the expression

```
obj([type=cds,name=gap,genome='E.coli']) * cds_to_eco2dbase
```

evaluates to the *eco2dbase* object related to the gene. Thus,

| ?- evalpp(obj([name=gap,genome='E.coli']) * cds_to_eco2dbase).

```
eco2dbase 515
    spot id: H034.3
    mol. wt. = 35376
    pI as predicted from seq = 7.07
    X coordinate in gel in Fig. 1 = 0.0
    Y coordinate in gel in Fig. 1 = 0.0
    X coordinate in gel in Fig. 2B = 77
    Y coordinate in gel in Fig. 2B = 76
    X coordinate in gel in Fig. 3B = 83
```

```
    Y coordinate in gel in Fig. 3B = 78
    protein name = Glyceraldehyde-3-phosphate dehydrogenase
        spot identified by comigration with purified protein
    donor: D. Fraenkel
    'EC'('1.2.1.12')
    'SWISSPROT'('G3P1_ECOLI')
    'GENE'(gap)
    'GENBANK'('Ecogap')
    'SEQREF'('EJB150;61')
    location on genetic linkage map: 39.3
    'DIR'('F')
    occurs in Kohara clone 330
===
```

Here, the exact meanings of the attributes will make sense only if you have read the paper describing the 2-d protein gel system being used to explore expression of *E. coli* genes or are familiar with the widely distributed database. To help, we provide summaries of the attributes for each type of object in appendicies to this tutorial (or you could try the *help* facility and see if it contains a summary of the possible attributes).

You may occasionally wish to traverse several relationships. For example, to get a listing of the alignments that contain at least one peptide corresponding to a gene in *E. coli*, one might use

```
| ?- eval.

|: all(obj([type=cds,genome='E.coli']) * cds_to_peptide * peptide_to_alignment).


[
['FA0293',alignment],
['FA0500',alignment],
['FA0016',alignment],
['FA0302',alignment],
['FA0299',alignment],
['FA0072',alignment],
['FA0268',alignment],
['FA0361',alignment],
['FA0381',alignment],
['FA0402',alignment],
['FA0380',alignment]
]
|:
```

At this point, you have seen the basic mechanisms for accessing objects and for traversing relationships between objects. We will return to these topics in later sections, exploring some of the more advanced options. However, before continuing, we recommend that you simply try to become fluent with these few features; then, once you can at least migrate through the available data, it will be appropriate to continue and learn how to gain access to much broader capabilities.

1.2 More on Attributes and Relationships

In the last section, you learned how to locate objects and traverse relationships to other objects. Now let us fill in two small details – how to ask for the set of attributes of an object and the relationships for an object. At this point, these are minor embellishments, since you can get this information using *evalppr* as discussed before.

The term

```
attributes(Object)
```

evaluates to the list of attributes associated with the object, and the term

```
relationships(Object)
```

evaluates to the list of relationships associated with the object.

To illustrate,

```
| ?- eval(attributes(obj([type=cds,name=aceA,genome='E.coli']))).

[
accession('EG10022'),
desc("isocitrate lyase; utilization of acetate"),
swissprot('P05313')
]
```

```
| ?- eval(relationships(obj([type=cds,name=aceA,genome='E.coli']))).
```

```
[
annotate( [hydGecoM,sequence_fragment,'E.coli'],
   object_to_piece(1,14191,15495,direct)),
annotate( ['4.1.3.1',enzyme], cds_to_enzyme),
annotate( ['Bachmann',map,'E.coli'], gene_to_map(90.942)),
annotate( ['P05313',peptide], cds_to_peptide)
]
```

To cause an expression to evaluate to the value of just one specific attribute, one would use

```
| ?- eval(attribute(obj([type=cds,name=aceA,genome='E.coli']),desc)).
```

```
"isocitrate lyase; utilization of acetate"
```

As we compose more complex expressions, we will find this ability to get at specific attributes to be critical.

1.3 Restricting a Set of Objects with Constraints

Often, one wishes to access a set of objects based on the value of one or more specific attributes. For example, suppose that you wished to access the set of genes within *E.coli* that are expressed during heat shock. Since the *eco2dbase* objects have encoded within them attributes that relate to expression, this is a reasonable goal. In particular, the attribute '46C' gives "Cellular abundance of the protein spot at 13.5oC relative to cellular abundnace at 46oC (J. Bacteriol. 139:185-194)." Thus, we might reasonably wish to see all genes that have a value greater than (say) 2.0 in the attribute '46C'. To do these, one would evaluate the expression

```
require(obj([type=eco2dbase]), X in attribute(X,'46C') > 2.0) * eco2dbase_to_cds.
```

Here, the subexpression

```
require(obj([type=eco2dbase]), X in attribute(X,'46C') > 2.0)
```

evaluates to an *eco2dbase* object with the desired property. The general form of the *require/2* operator is

```
require(ObjectExpression,Requirement)
```

which will find an object satisfying the ObjectExpression for which Requirement evaluates to *true*. The format of the Requirement expression is

```
X in ExpressionInX
```

That is, *X* indicates a variable that takes on the value of the desired object, and then X appears in the condition that gets evaluated.

Think of the overall form of *require/2* as saying "find an object X such that ExpressionInX is *true*".

Finally, in the complete expression, GenoBase finds such an object and then traverses the relationship to the desired gene. The outcome of such a query looks like

```
| ?- eval.

|: require(obj([type=eco2dbase]),X in attribute(X,'46C') > 2.0) * eco2dbase_to_cds.

[grpE,cds,'E.coli'] a
[mopA,cds,'E.coli'] ;
[dnaK,cds,'E.coli'] ;
[htpE,cds,'E.coli'] ;
[mopB,cds,'E.coli'] ;
[recA,cds,'E.coli'] ;
[htpG,cds,'E.coli'] ;
[lysU,cds,'E.coli'] ;
[glpK,cds,'E.coli'] ;
[carB,cds,'E.coli'] ;
[ompA,cds,'E.coli'] ;
[ilvE,cds,'E.coli'] ;
[sucB,cds,'E.coli'] ;
[sdhA,cds,'E.coli'] ;
[clpB,cds,'E.coli'] ;
[carA,cds,'E.coli'] ;
[glyA,cds,'E.coli'] ;
```

You can use multiple constraints within a *require/2* expression by using the *and/2* operator. Thus,

```
require(obj([type=eco2dbase]),X in and(attribute(X,'46C') > 2.0,
                                       attribute(X,'46C') < 3.0)).
```

evaluates to objects in which the 46C attribute has a value between 2.0 and 3.0.

If you do not really care what value an attribute has, but only wish to ensure that the object actually has the attribute, you should use the *has_attribute* operator. For example, had we defined "heat shock gene" as one for which there was a '50C' attribute in the *eco2dbase* object, we would have used an expression of the form

```
require(obj([type=eco2dbase]),X in has_attribute(X,'50C')).
```

Now, before proceeding, let us briefly summarize the key points that have been discussed so far:

- Expressions evaluate to one of a set of values.
- One uses expressions that evaluate to objects to access objects within the database.
- *Object1 * Relationship* is an expression that evaluates to an *Object2* that stands in the given relationship to *Object1*. That is, one uses the asterisk operator (which is called an "infix operator") to traverse relationships.
- You can constrain the set of objects that a given expression E evaluates to by using *require(E, X in Condition)*. Here X is an arbitrary name of a variable that is bound to an object that results from evaluating E (i.e., you could use any uppercase letter in place of X).
- When evaluating expressions, GenoBase attempts to find a single suitable value. If you then request alternative values, it will try to locate those, as well.

At this point, you have access to the basic tools to locate sets of objects from the database. However, this basic notion of obtaining answers by "evaluating expressions" can be used in far more powerful ways. They may well seem a bit unnatural to you, until you get the hang of what is going on, but it is a paradigm that supports a very strong expressive power. In later sections we will dramatically expand the expressive capabilities by introducing numerous specialized operators. However, before you go on to those points, you really do need to get comfortable with the basic operations used to navigate among the objects in the database.

1.4 Saving Results in Temporary Variables

You will find that it is sometimes convenient to retain the values of an expression. As an example of how this is done,

|: $hsg := require(obj([type=eco2dbase]),X in attribute(X,'46C') > 1.0) * eco2dbase_to_cds.

```
[crr,cds,'E.coli'] a
[grpE,cds,'E.coli'] ;
.
.
.
```

evaluates the expression, saving the result in *$hsg* (all of these "user variables" that play the role of accumulating temporary sets must begin with a $). The variable takes on the entire set of returned values for the expression, so

```
eval($hsg).
```

would produce the same output as the previous evaluation.

Finally, one can clear a user variable with

```
eval(clear($hsg)).
```

2 Intervals and Points

You will often find it useful to be able to refer to subintervals of objects (e.g., the region preceding a gene, a region of a chromosome, etc.). To do this, one can use an expression of the form

```
interval(Object,Begin,End,Direction)
```

Thus,

```
interval(obj([name=aceA,type=cds,genome='E.coli']),-18,18,direct)
```

evaluates to a 37-character region starting 18 characters upstream of *aceA* and continuing through the 19th character of the gene (i.e., the beginning and ending positions are given as offsets from the first character of the object – 0 corresponds to the first character).

A *point* is a specific location on an object (i.e., a specific base pair given as an offset into an object). There are a variety of ways to specify points:

```
point(obj([name=thrA,type=cds,genome='E.coli']),10)
start(obj([name=thrA,type=cds,genome='E.coli'])) + 10
```

are equivalent ways to specify the 11th character of the gene *thrA* (and, for the record, there is also an *end/1* operator that evaluates to the point at the end of an object – i.e., the last bp in the object).

Finally, before we illustrate these concepts with a specific sequence of operations, we will introduce the operation that evaluates to the DNA sequence of an object:

```
dna_sequence(Object)
```

evaluates to the actual string of characters that make up the sequence of the object (assuming that the object has been sequenced – if not, evaluation of the expression will just fail).

In the following short sequence of commands, we illustrate these notions.

```
| ?- evalpp.
|: interval(obj([name=aceA,type=cds,genome='E.coli']),-18,18,direct).
INTERVAL FROM -18 TO 18 ON cds aceA of E.coli
    accession('EG10022')
    desc("isocitrate lyase; utilization of acetate")
    swissprot('P05313')
===
```

```
|: dna_sequence(interval(obj([name=aceA,type=cds,genome='E.coli']),
                         -18,18,direct)).
DNA_SEQUENCE of length 37
        0 ACTATGGAGC ATCTGCACAT GAAAACCCGT ACACAAC
|: $x := obj([name=aceA,type=cds,genome='E.coli']).
cds aceA of E.coli
    accession('EG10022')
    desc("isocitrate lyase; utilization of acetate")
    swissprot('P05313')
===
|: start($x).
POINT AT 0 ON cds thrA of E.coli
    accession('EG10998')
    desc("aspartokinase I-homoserine dehydrogenase I")
    swissprot('P00561')
===
|: interval(start($x),start($x)+10).
INTERVAL FROM 0 TO 10 ON cds thrA of E.coli
    accession('EG10998')
    desc("aspartokinase I-homoserine dehydrogenase I")
    swissprot('P00561')
===
|: dna_sequence(interval(start($x),start($x)+6)).
DNA_SEQUENCE of length 7
        0 ATGCGAG
```

3 More on Sequences and Pattern Matching

In the last section, we introduced an operator (*dna_sequence/1*) that can be used to get the actual DNA sequence of an object. There is a corresponding operator to get the sequence of a protein. For example,

```
| ?- evalpp.
|: $aceA := obj([name=aceA,type=cds,genome='E.coli']).
cds aceA of E.coli
    accession('EG10022')
    desc("isocitrate lyase; utilization of acetate")
    swissprot('P05313')

|: protein_sequence($aceA).
PROTEIN_SEQUENCE of length 435
        0 MKTRTQQIEE LQKEWTQPRW EGITRPYSAE DVVKLRGSVN PECTLAQLGA
       50 AKMWRLLHGE SKKGYINSLG ALTGGQALQQ AKAGIEAVYL SGWQVAADAN
```

```
100 LAASMYPDQS LYPANSVPAV VERINNTFRR ADQIQWSAGI EPGDPRYVDY
150 FLPIVADAEA GFGGVLNAFE LMKAMIEAGA AAVHFEDQLA SVKKCGHMGG
200 KVLVPTQEAI QKLVAARLAA DVTGVPTLLV ARTDADAADL ITSDCDPYDS
250 EFITGERTSE GFFRTHAGIE QAISRGLAYA PYADLVWCET STPDLELARR
300 FAQAIHAKYP GKLLAYNCSP SFNWQKNLDD KTIASFQQQL SDMGYKFQFI
350 TLAGIHSMWF NMFDLANAYA QGEGMKHYVE KVQQPEFAAA KDGYTFVSHQ
400 QEVGTGYFDK VTTIIQGGTS SVTALTGSTE ESQF
```

Less commonly used functions for translating either DNA sequence (or an object X for which *dna_sequence(X)* can be evaluated to DNA sequence) to a peptide sequence exist. *translate/1* can be used to translate any string of DNA characters that has an appropriate length (i.e., a multiple of 3); *translate_nostop/1* will translate any string in which there are no embedded stop codons, and it will remove a terminal stop codon. Thus,

```
|: translate(dna_sequence("ATGTAA")).
protein_sequence(2,"M*")
|: translate_nostop(dna_sequence("ATGTAA")).
PROTEIN_SEQUENCE of length 2
        0 M
|: translate(dna_sequence("ATGTAAATG")).
protein_sequence(3,"M*M")
|: translate_nostop(dna_sequence("ATGTAAATG")).
|: translate($aceA).
|: translate(dna_sequence($aceA)).
PROTEIN_SEQUENCE of length 435
        0 MKTRTQQIEE LQKEWTQPRW EGITRPYSAE DVVKLRGSVN PECTLAQLGA
       50 AKMWRLLHGE SKKGYINSLG ALTGGQALQQ AKAGIEAVYL SGWQVAADAN
      100 LAASMYPDQS LYPANSVPAV VERINNTFRR ADQIQWSAGI EPGDPRYVDY
      150 FLPIVADAEA GFGGVLNAFE LMKAMIEAGA AAVHFEDQLA SVKKCGHMGG
      200 KVLVPTQEAI QKLVAARLAA DVTGVPTLLV ARTDADAADL ITSDCDPYDS
      250 EFITGERTSE GFFRTHAGIE QAISRGLAYA PYADLVWCET STPDLELARR
      300 FAQAIHAKYP GKLLAYNCSP SFNWQKNLDD KTIASFQQQL SDMGYKFQFI
      350 TLAGIHSMWF NMFDLANAYA QGEGMKHYVE KVQQPEFAAA KDGYTFVSHQ
      400 QEVGTGYFDK VTTIIQGGTS SVTALTGSTE ESQF*
```

Note that the two requests

```
|: translate_nostop(dna_sequence("ATGTAAATG")).
|: translate($aceA).
```

failed to evaluate; the first because the sequence contained an embedded stop codon, and the second because these operators take *dna_sequence* as arguments (not objects that can be evaluated to DNA sequence).

Now, let us proceed to discuss the issue of how to search for patterns in either DNA or peptide sequences. We will begin by talking about how to construct a pattern. A pattern is composed of a sequence of *pattern units.* When a pattern is matched against a sequence, each of the pattern units must successfully match a subsequence. For example, the pattern

```
AATG 2...4 CATT
```

is composed of three pattern units; the first and third are just character sequences, and the second matches any subsequence of 2 to 4 characters. Thus, this simple pattern could match

```
AATGCCCATT or
AATGCATTCATT or
AATGTTTCATT
```

or any of a variety of similar subsequences. In this example, we illustrated just two of the basic types of pattern units. There is a fairly limited set of types of pattern units, but it is rich enough to allow you to express a wide variety of possible patterns. Here is a basic set (we will enrich this basic set substantially as we continue, but it is a good starting point):

- an *exact match* pattern unit is just a sequence of characters. For a pattern constructed to match nucleotide sequences, the characters must be from the alphabet {A,C,G,T,U,M,R,W,S,Y,K,B,D,H,V,N}, which is the standard set for representing nucleotides (with the ability to represent ambiguous characters). The meanings of the ambiguity codes are

  ```
  M means one of {A,C}
  R means one of {A,G}
  W means one of {A,T,U}
  S means one of {C,G}
  Y means one of {C,T,U}
  K means one of {G,T,U}
  B means one of {C,G,T,U}
  D means one of {A,G,T,U}
  H means one of {A,C,T,U}
  V means one of {A,C,G}
  N means one of {A,C,G,T,U}
  ```

  For a pattern used to match peptide sequences, you must select the characters from the set {A,C,D,E,F,G,H,I,K,L,M,N,P,Q,R,S,T,V,W,Y}, which are the standard 1-character codes for amino acids.

- An *elipses* pattern unit is of the form X...Y, and it matches any subsequence of length X to Y (where X is less than or equal to Y, and X is nonnegative).
- an *any* pattern unit is used to match any of a set of amino acids. Thus, *any(RFY)* matches a single character (which must be R, F, or Y).
- a *notany* pattern unit matches a single amino acid character, as long as it is not one of the set given. Thus, *notany(RFY)* matches any single character that is not an R, F, or Y.

With just this basic set, you can express a fairly wide class of patterns. For example, the pattern

```
any(ST) 1...1 any(RK)
```

can be used to search for the pattern identified by Amos Bairoch in his ProSite collection as recognizing a protein kinase C phosphorylation site (see the description associated with PDOC00005, which is a *prosite_doc* object created from Bairoch's collection – that is, pretty print the object that *obj([name='PDOC00005',type=prosite_doc])* evaluates to).

In fact, this modest set of basic pattern units offers the ability to search for most of the ProSite patterns. However, there are many types of structural features that cannot be expressed in this simple set. For example, one would like to be able to look for repeating sequences or palindromes in DNA sequences, as well as the ability to look for "fuzzy" matches (where the string matched must be "close", but not necessarily an exact match). So, now let us proceed the mechanisms included to support these more advanced types of pattern matching.

The first capability that we need to introduce is the ability to "name" a pattern unit and then refer back to the name. Let us illustrate:

```
p1=6...6 0...20 p1
```

is a pattern composed of three pattern units. The first is an *elipses* that matches any 6 characters. Furthermore, the first pattern unit is assigned the "name" *p1*. The third pattern unit specifies an exact match against whatever *p1* matched. Thus, the pattern recognizes an exact repeat of 6 characters separated by 0 to 20 intervening characters. Names must be one *p0, p1, p2, p3 ... p9*; that is, *p* followed by a single numeric digit.

The next addition to our basic pattern matching repertoire is the ability to match reverse-complements of previously matched subsequences in a DNA-matching pattern; this

will be useful for identifying structures like hairpin loops and palindromes. To do this, one would use a pattern like

```
p1=8...10 3...15 ~p1
```

which specifies three pattern units. The first is named *p1* and the third matches an exact reverse-complement of whatever matched *p1* (which allows one to look for hairpin loops with stems of 8 to 10 characters that bond perfectly). To match a more complex pattern, like a pseudo-knot, one might use a pattern like

```
p1=6...6 2...4 p2=6...6 0...6 ~p1 8...10 ~p2
```

However, a reader that has actually sought such structures in DNA will find the requirement that the reverse complement be precise to be too confining; it is often the case that a few mismatches can be tolerated, or even insertions and deletions will occur. Hence, our next extension of the basic set of tools is to generalize some of our pattern units to allow imprecise matches. Thus, we allow one to append *[M,D,I]* to an exact match pattern unit or to a complement pattern unit. Thus,

```
TATAAGTT[1,0,1]
GATCGATC[0,1,0]
```

are valid. The three integers give the number of tolerated mismatches, deletions, and insertions, respectively. A "mismatch" is when a character in the pattern unit mismatches a character in the subsequence, a deletion refers to a case in which a character in the pattern is simply omitted in the subsequence, and an insertion is when a character in the subsequence is omitted in the pattern. Thus,

```
CATT[0,1,0]
```

would match the subsequence CTT (due to the tolerance of a single deletion). Now, one can relax the requirements for a match and still detect fairly complex structures in DNA.

Finally, we need to introduce the ability to search for "fuzzy" matches using "weight matricies". Suppose that you had aligned all known versions of a given type of feature that was 5-characters long. Suppose that you counted up the number of times an A, C, G, and T occurred in each column of the alignment giving

```
A C G T   A C G T   A C G T   A C G T   A C G T

1 0 9 1   0 5 1 5   10 0 1 0   11 0 0 0   2 5 0 4
```

Then, to search for an occurrence of such a structure, one might use a pattern of the form

```
{(1,0,9,1),(0,5,1,5),(10,0,1,0),(11,0,0,0),(2,5,0,4)} > 34
```

This pattern unit can only match 5-character subsequences. To determine whether or not it matches a specific subsequence, you simply take the integer corresponding to the character in the subsequence from each 4-tuple and compute the sum; if the sum exceeds 34, a match has occurred. While it is a bit tedious to type in such a long pattern unit, and it requires some experimentation to determine the appropriate threshhold, this can be a powerful tool – particularly when it is embedded with other pattern units in a complex pattern.

The example that we used to illustrate weight matricies used a pattern unit to search DNA (i.e., we used 4-tuples). To form a pattern unit to search a peptide string, one uses 20-tuples (where the entries refer to the amino acid codes in alphabetic order). These are almost impossible to accurately type at a terminal, and this capability is normally used with patterns that have been encoded as objects. We will defer this discussion to a later section.

The final comment that we must make concerning the formation of patterns is that one encloses complete patterns within single apostrophes, as you will see in the examples below.

Now, we need to illustrate the actual use of patterns. Consider the following short interaction with GenoBase:

```
| ?- evalpp.
|: protein_sequence(obj([name='P11447',type=peptide])).
PROTEIN_SEQUENCE of length 51
        0 MALWGGRFTC AADQRFKQFN DSLRFDYRLA EQDIVGSVAW SKALVTVGVL
       50 T
|: matches(protein_sequence(obj([name='P11447',type=peptide])),'Q 0...6 Q').
[
Q AAD Q  at offset 9,
Q RFK Q  at offset 13
]
|: matches(obj([name=aceA,type=cds,genome='E.coli']),'ARRYAG 2...8 RR').
```

```
[
AGGTAG CGGC GG  at offset 280,
AGGTAG CGGCG GA  at offset 280
]
|: halt.

yes
| ?- eval.
|: matches(obj([name=aceA,type=cds,genome='E.coli']),'ARRYAG 2...8 RR').
|: [
annotate( dna_sequence(12,"AGGTAGCGGCGG"), vector([ 280, 286, 290, 292 ])),
annotate( dna_sequence(13,"AGGTAGCGGCGGA"), vector([ 280, 286, 291, 293 ]))
]
```

Here, we introduce the *match/2* operator. Its first argument can be a protein sequence, a DNA sequence, or a term for which *dna_sequence/1* can be applied to produce a DNA sequence (which is what we did in the second use of *matches/2*. The second argument is a pattern. Evaluation of the term produces a *list* of *annotated objects*. To show you what an annotated object looks like, we reissued the second query under *eval*; you should compare it to what got pretty-printed when we used *evalpp*. An annotated object is of the form

```
annotate(Object,Annotation)
```

In the example above, the annotation is a vector of integers indicating where each pattern unit matched. The first integer is the offset of where the first pattern unit matched, the second where the second matched, and so forth (the last integer is the offset just past the subsequence matched by the last pattern unit). The pretty-printing uses the annotation to give a more readable presentation of the matched subsequence.

Now, the most common type of pattern matching involves searching through sets of objects for specific patterns. Creating just the right expression to give you what you want can be quite tricky, until you become comfortable with the overall paradigm of expression evaluation. Initially, you will probably need to pattern your attempts after some examples. We will now present a few, and we intend to build a growing set in the help presentation for *pattern_examples*. So, to begin: suppose that you wished to search for large hairpin loops in the region around the start of genes that code for proteins (i.e., around the start of sequenced objects of type *cds*). To do this, you need an expression that

1. finds an object of type *cds*,
2. locates the interval centered around the start of the object (let's say 15 characters on either side, to be specific),

3. computes the set of hairpins in that region (let's insist on stems of at least 9 characters, for now), and

4. displays the results only if the set is nonempty.

Of course, you will wish to iterate over all possible *cds* objects, and you will wish to see which objects were used in successfully detecting the hairpins.

The following session illustrates such a search

```
|: obj([type=cds]) with X in
require(matches(interval(start(X)-15,start(X)+15),
                'p1=9...9 3...10 ~p1'),Y in size(Y) > 0).
ANNOTATE cds fucK of E.coli
    accession('EG10350')
    desc("L-Fuculose kinase")
    swissprot('P11553')
=== BY [
TAGCCGGAT AAGCAATGTT ATCCGGCTA  at offset 1,
AGCCGGATA AGCAATGT TATCCGGCT  at offset 2,
GCCGGATAA GCAATG TTATCCGGC  at offset 3
] a
ANNOTATE cds pyrL of E.coli
    accession('EG11278')
    swissprot('P17776')
=== BY [
AAGGCGACT GATG AGTCGCCTT  at offset 5
] ;
```

Note that the overall expression is of the form

```
Expression1 with X in Expression2
```

This type of expression has not been covered before. It means "evaluate *Expression1* to get a value and call it *X*, and then try to evaluate *Expression2*; if *Expression2* can be evaluated, then return the value of *Expression1* annotated with the value of *Expression2*." Here, *Expression1* evaluates to a *cds*.

Expression2 evaluates as follows:

1. The basic form of the expression is *require(Expression3,Y in Expression4)*. As we discussed earlier, this means "evaluate *Expression3* and when you get a value, call it *Y*. Then, evaluate *Expression4*. If the result is *true*, then the value of the whole *require/2* expression is the value of *Expression3*; else, try to determine an alternative evaluation of *Expression3*, and so forth."

2. *Expression3* evaluates to the list of matches for the pattern over an object that can be evaluated using *dna_sequence/2*. That is, we could have used the slightly longer (but equivalent) *matches(dna_sequence(interval(start(X)-15,start(X)+15)) ...)*.

3. *Expression4* evbaluates to *true* exactly when the list of matches is nonempty.

The reader should study this example carefully, since many common types of searches can be made by just altering the interval to be scanned and the pattern.

4 Codon Usage and Kmer Statistics

Occasionally, it will be useful to example codon usage in a single object or a set of objects (i.e., the *cds* objects in a given genome). This can be done as follows:

```
| ?- eval.
|: codon_usage(obj([name=aceA,type=cds])).
annotate( vector([ 18, 10, 3, 2, 2, 12, 5, 8, 0, 6,
        0, 2, 0, 9, 10, 11, 8, 7, 21, 1,
        5, 0, 11, 1, 0, 4, 1, 11, 0, 3,
        27, 0, 19, 8, 8, 15, 10, 12, 26, 8,
        1, 22, 0, 12, 2, 7, 11, 6, 1, 6,
        0, 11, 3, 4, 8, 2, 0, 4, 8, 1,
        3, 13, 0, 6 ]), codon_usage) ;
annotate( vector([ 23, 8, 9, 4, 7, 0, 4, 9, 1, 0,
        0, 1, 0, 10, 10, 14, 20, 3, 4, 2,
        3, 0, 4, 9, 0, 7, 0, 10, 3, 1,
        0, 12, 26, 7, 7, 13, 12, 5, 10, 15,
        10, 5, 1, 10, 7, 3, 7, 4, 1, 8,
        0, 5, 2, 0, 1, 7, 0, 0, 3, 0,
        8, 10, 0, 7 ]), codon_usage) ;
|:

| ?- evalpp.
|: codon_usage(obj([name=aceA,type=cds])).
number codons = 435
    alanine: 56          12.87%
```

GCA:	10	2.30%
GCC:	12	2.76%
GCG:	26	5.98%
GCT:	8	1.84%
arginine:	16	3.68%
AGA:	0	0.00%
AGG:	0	0.00%
CGA:	0	0.00%
CGC:	4	0.92%
CGG:	1	0.23%
CGT:	11	2.53%
asparagine:	12	2.76%
AAC:	10	2.30%
AAT:	2	0.46%
aspartic_acid:	23	5.29%
GAC:	8	1.84%
GAT:	15	3.45%
cysteine:	5	1.15%
TGC:	4	0.92%
TGT:	1	0.23%
glutamic_acid:	27	6.21%
GAA:	19	4.37%
GAG:	8	1.84%
glutamine:	29	6.67%
CAA:	8	1.84%
CAG:	21	4.83%
glycine:	35	8.05%
GGA:	1	0.23%
GGC:	22	5.06%
GGG:	0	0.00%
GGT:	12	2.76%
histidine:	8	1.84%
CAC:	7	1.61%
CAT:	1	0.23%
isoleucine:	20	4.60%
ATA:	0	0.00%
ATC:	9	2.07%
ATT:	11	2.53%

leucine: 33	7.59%
CTA: 0	0.00%
CTC: 3	0.69%
CTG: 27	6.21%
CTT: 0	0.00%
TTA: 3	0.69%
TTG: 0	0.00%
lysine: 21	4.83%
AAA: 18	4.14%
AAG: 3	0.69%
methionine: 10	2.30%
ATG: 10	2.30%
phenylalanine: 19	4.37%
TTC: 13	2.99%
TTT: 6	1.38%
proline: 17	3.91%
CCA: 5	1.15%
CCC: 0	0.00%
CCG: 11	2.53%
CCT: 1	0.23%
serine: 25	5.75%
AGC: 6	1.38%
AGT: 2	0.46%
TCA: 3	0.69%
TCC: 4	0.92%
TCG: 8	1.84%
TCT: 2	0.46%
stop: 1	0.23%
TAA: 1	0.23%
TAG: 0	0.00%
TGA: 0	0.00%
threonine: 27	6.21%
ACA: 2	0.46%
ACC: 12	2.76%
ACG: 5	1.15%
ACT: 8	1.84%
tryptophan: 8	1.84%
TGG: 8	1.84%
tyrosine: 17	3.91%

```
        TAC: 6          1.38%
        TAT: 11         2.53%

    valine: 26          5.98%
        GTA: 2          0.46%
        GTC: 7          1.61%
        GTG: 11         2.53%
        GTT: 6          1.38%

|: codon_usage(all(obj([type=cds,genome='E.coli']))).
number codons = 420695
    alanine: 40323      9.58%
        GCA: 8373       1.99%
        GCC: 10157      2.41%
        GCG: 14780      3.51%
        GCT: 7013       1.67%

    arginine: 24668     5.86%
        AGA: 590        0.14%
        AGG: 420        0.10%
        CGA: 1248       0.30%
        CGC: 9895       2.35%
        CGG: 1947       0.46%
        CGT: 10568      2.51%

    asparagine: 16102   3.83%
        AAC: 10207      2.43%
        AAT: 5895       1.40%

    aspartic_acid: 22633 5.38%
        GAC: 9346       2.22%
        GAT: 13287      3.16%

    cysteine: 4721      1.12%
        TGC: 2764       0.66%
        TGT: 1957       0.47%

    glutamic_acid: 26314 6.25%
        GAA: 18457      4.39%
        GAG: 7857       1.87%

    glutamine: 18248    4.34%
        CAA: 5487       1.30%
        CAG: 12761      3.03%

    glycine: 31641      7.52%
        GGA: 2604       0.62%
        GGC: 13391      3.18%
```

GGG:	4041	0.96%
GGT:	11605	2.76%
histidine:	9688	2.30%
CAC:	4721	1.12%
CAT:	4967	1.18%
isoleucine:	24067	5.72%
ATA:	1135	0.27%
ATC:	11558	2.75%
ATT:	11374	2.70%
leucine:	42655	10.14%
CTA:	1247	0.30%
CTC:	4268	1.01%
CTG:	23963	5.70%
CTT:	3920	0.93%
TTA:	4327	1.03%
TTG:	4930	1.17%
lysine:	19526	4.64%
AAA:	14965	3.56%
AAG:	4561	1.08%
methionine:	11567	2.75%
ATG:	11567	2.75%
phenylalanine:	15537	3.69%
TTC:	7725	1.84%
TTT:	7812	1.86%
proline:	18615	4.42%
CCA:	3349	0.80%
CCC:	1762	0.42%
CCG:	10929	2.60%
CCT:	2575	0.61%
serine:	23143	5.50%
AGC:	6600	1.57%
AGT:	2799	0.67%
TCA:	2441	0.58%
TCC:	3961	0.94%
TCG:	3417	0.81%
TCT:	3925	0.93%
stop:	1351	0.32%
TAA:	827	0.20%
TAG:	85	0.02%

```
    TGA: 439          0.10%

threonine: 22264      5.29%
    ACA: 2217         0.53%
    ACC: 10529        2.50%
    ACG: 5536         1.32%
    ACT: 3982         0.95%

tryptophan: 5677      1.35%
    TGG: 5677         1.35%

tyrosine: 11786       2.80%
    TAC: 5711         1.36%
    TAT: 6075         1.44%

valine: 30169         7.17%
    GTA: 4791         1.14%
    GTC: 6092         1.45%
    GTG: 11133        2.65%
    GTT: 8153         1.94%

|: halt.
```

There are several points worth mentioning in the last short dialogue:

1. The *codon_usage/1* operator evaluates to an annotated object. The object is a vector of 64 integers (representing the number of occurrences of AAA, AAC, AAG, AAT, ACA,... TTT). The annotation records the fact that the vector represents codon usage.
2. When the argument of the *codon_usage/1* operator can evaluate to multiple values (i.e., there are multiple possible objects that one could apply the operator on to get a codon usage vector), it will return codon usages for them one-at-a-time.
3. Pretty-printing the annotated vector will produce a more readable report of the codon usage.
4. To get the codon usage of more than a single gene, one must apply the *codon_usage/1* operator to a *list*. The *all/1* operator can be used to construct the list of all possible evaluations of its argument. In the last expression evaluated, we used this to construct the set of all *cds* genes in *E. coli* and then took the codon usage of the set.

When a vector giving the occurrence of each oligo of some specified length is desired, you should use *kmers/2*. The first argument evaluates to a DNA sequence, and the second

gives the size of the oligos (in the following example, we used 3, which produces counts for the 64 possible 3-mers).

```
| ?- eval.
|: kmers(dna_sequence(obj([type=cds,name=aceA,genome='E.coli'])),3).
annotate( vector([ 29, 23, 24, 13, 18, 19, 13, 21, 12, 31,
        13, 18, 7, 22, 26, 17, 22, 18, 40, 12,
        17, 7, 28, 17, 23, 27, 35, 22, 14, 13,
        37, 12, 35, 14, 8, 24, 32, 24, 47, 26,
        16, 42, 15, 22, 13, 21, 23, 16, 4, 16,
        2, 22, 25, 19, 19, 12, 30, 29, 32, 11,
        10, 19, 16, 9 ]), kmers(3))
|: halt.

| ?- evalpp.
|: kmers(dna_sequence(obj([type=cds,name=aceA,genome='E.coli'])),3).
 AAA: 29                2.23%
 AAC: 23                1.77%
 AAG: 24                1.84%
 AAT: 13                1.00%
 ACA: 18                1.38%
 ACC: 19                1.46%
 ACG: 13                1.00%
 ACT: 21                1.61%
 AGA: 12                0.92%
 AGC: 31                2.38%
 AGG: 13                1.00%
 AGT: 18                1.38%
 ATA: 7                 0.54%
 ATC: 22                1.69%
 ATG: 26                2.00%
 ATT: 17                1.30%
 CAA: 22                1.69%
 CAC: 18                1.38%
 CAG: 40                3.07%
 CAT: 12                0.92%
 CCA: 17                1.30%
 CCC: 7                 0.54%
 CCG: 28                2.15%
 CCT: 17                1.30%
 CGA: 23                1.77%
 CGC: 27                2.07%
 CGG: 35                2.69%
 CGT: 22                1.69%
 CTA: 14                1.07%
 CTC: 13                1.00%
 CTG: 37                2.84%
```

```
CTT: 12                 0.92%
GAA: 35                 2.69%
GAC: 14                 1.07%
GAG: 8                  0.61%
GAT: 24                 1.84%
GCA: 32                 2.46%
GCC: 24                 1.84%
GCG: 47                 3.61%
GCT: 26                 2.00%
GGA: 16                 1.23%
GGC: 42                 3.22%
GGG: 15                 1.15%
GGT: 22                 1.69%
GTA: 13                 1.00%
GTC: 21                 1.61%
GTG: 23                 1.77%
GTT: 16                 1.23%
TAA: 4                  0.31%
TAC: 16                 1.23%
TAG: 2                  0.15%
TAT: 22                 1.69%
TCA: 25                 1.92%
TCC: 19                 1.46%
TCG: 19                 1.46%
TCT: 12                 0.92%
TGA: 30                 2.30%
TGC: 29                 2.23%
TGG: 32                 2.46%
TGT: 11                 0.84%
TTA: 10                 0.77%
TTC: 19                 1.46%
TTG: 16                 1.23%
TTT: 9                  0.69%
```

It is wise to use some caution when displaying the output of such an evaluation, since a request for even 8-mers will produce over 64,000 counts in the vector! Most often, one will wish to compare kmer usage or codon usage from different genes or organisms. In such a case, one would evaluate the vectors, and then use them in more complex expressions. We will illustrate the power of this approach later, when we develop operators for processing vectors. One small point might be worth noting here: if you need the unannotated vector, use *unannotate/1* as in the following example.

```
| ?- eval.
|: unannotate(kmers(dna_sequence(obj([type=cds,name=aceA,genome='E.coli'])),3)).
```

```
vector([ 29, 23, 24, 13, 18, 19, 13, 21, 12, 31,
        13, 18, 7, 22, 26, 17, 22, 18, 40, 12,
        17, 7, 28, 17, 23, 27, 35, 22, 14, 13,
        37, 12, 35, 14, 8, 24, 32, 24, 47, 26,
        16, 42, 15, 22, 13, 21, 23, 16, 4, 16,
        2, 22, 25, 19, 19, 12, 30, 29, 32, 11,
        10, 19, 16, 9 ])
```

5 Searching for Common Subsequences

Another common class of queries involves looking for short subsequences that occur several times within one or more objects. For example, one might reasonably look for all sequences of length 6 or more that occur more than once in a region just upstream of a gene; or, even better, one might look for sequences that occur multiple times in front of two or more genes expressed during heat shock. In this section, we will introduce ways to find such sequences.

The basic operator that is used to search for common sequences is *common/2*, where the first argument is a list of objects to be scanned, and the second is the minimum size of common strings that you wish to see. If you want to insist that one of the objects to be scanned contain multiple occurrences of the common string, then that object is given as

```
[Object,NumberOccurrences]
```

To illustrate,

```
| ?- eval.
|: common([dna_sequence("ACGTACGTACGT"),dna_sequence("CGTACGTACGT")],3).
[
annotate( dna_sequence(3,"CGT"), [
vector([ 1, 5, 9 ]),
vector([ 0, 4, 8 ])
]),
annotate( dna_sequence(4,"ACGT"), [
vector([ 0, 4, 8 ]),
vector([ 3, 7 ])
]),
annotate( dna_sequence(6,"GTACGT"), [
vector([ 2, 6 ]),
vector([ 1, 5 ])
```

```
])
]

|: common([obj([name=aceA,type=cds,genome='E.coli']),
          obj([name=aceE,type=cds,genome='E.coli'])],11).
[
annotate( dna_sequence(11,"CTACATCAACA"), [
vector([ 191 ]),
vector([ 176 ])
]),
annotate( dna_sequence(11,"GTCGAAAAAAG"), [
vector([ 179 ]),
vector([ 290 ])
]),
annotate( dna_sequence(12,"ATCTGGAACTGG"), [
vector([ 880 ]),
vector([ 223 ])
])
]

|: common([obj([name=aceA,type=cds,genome='E.coli']),
           [obj([name=aceE,type=cds,genome='E.coli']),2]
         ],8).
[
annotate( dna_sequence(8,"AGCAGCTG"), [
vector([ 1012 ]),
vector([ 481, 2219 ])
]),
annotate( dna_sequence(8,"CGAAAGAT"), [
vector([ 1168 ]),
vector([ 979, 2236 ])
]),
annotate( dna_sequence(8,"CTACATCA"), [
vector([ 191 ]),
vector([ 176, 2060 ])
]),
annotate( dna_sequence(8,"CTTCGAAG"), [
vector([ 551 ]),
vector([ 842, 2519 ])
]),
annotate( dna_sequence(8,"TCGAAAAA"), [
vector([ 180 ]),
vector([ 291, 1297 ])
]),
annotate( dna_sequence(8,"TGGGCGGC"), [
vector([ 592 ]),
vector([ 310, 1833 ])
]),
```

```
annotate( dna_sequence(9,"ATCTGGAAC"), [
vector([ 880 ]),
vector([ 223, 631 ])
])
]
```

These examples illustrate the basic techniques of how to search for common subsequences.

6 Summary

This tutorial is actually an evolving draft that we give to our friends to help them get started. We have found it impossible to keep up with the tasks required to integrate new databases, update the documentation on GenoBase, and also to adequately describe the basic features and technology used to create the system. Our current plan is to publish this document as a technical report, and then to begin work on Part 2, which will attempt to fill in the missing pieces.

We hope that the basic view of data extraction via evaluation has been conveyed to the reader. Once a firm grasp of this approach has been achieved, it does become possible to experiment with a much wider range of operators fairly quickly. To support those adventurous souls that are willing to make such an effort, we include a very brief description of most of the current operators in the Appendix.

Appendix

Summary of Expressions That Can Be Evaluated

You can evaluate expressions with

```
eval(Expression, Value).
```

There are a number of more convenient user interfaces based on this predicate, including

The expressions that follow can be evaluated at the Prolog prompt with four basic types of predicates – *eval, evalpp, evalppr,* and *evalpprpp* . Each of these predicates takes an optional single argument. The 0-ary versions cause the user to be prompted for an expression, which is evaluated and displayed (actually, the user is repeatedly queried, until he indicates that no more expressions are to be evaluated). The 1-ary versions take the input argument as the expression to be evaluated. *eval* prints out a fairly minimal description of the output of evaluation. *evalpp* pretty-prints the evaluated expression, *evalppr* pretty-prints the resulting object(s) and displays relationships to other objects, *evalpprpp* pretty-prints the evaluated expression, along with any related objects. Finally, *eval/2* allows a user to evaluate an expression and bind the second argument to the result. Thus, we summarize this with

```
eval/0,1      print value as Prolog term
evalpp/0,1    pretty print the value
evalppr/1     pretty print the value plus its relationships
evalpprpp/1   pretty print the value and pretty print its
              related objects
eval/2       bind the second argument to the result
```

The arity one versions of the predicates accept an expression as an argument; the arity zero versions read an expression from the user, process it, and loop. Once a value has been printed out, the programs prompt the user for one of ¡RETURN¿ (which means accept the value), ";" (which means try to backtrack for another), or "a" (which means backtrack through all remaining values). When there is no remaining value or the user hits ¡RETURN¿, the predicate terminates or prompts the user for another expression.

SEMANTICS OF EXPRESSIONS

Expressions are Prolog terms. Various operators are defined which evaluate their arguments as expressions and then produce a value. There are also various meta-operators, which do things like get all values for an expression or substitute a value into an expression containing a variable.

There is a special notion of *VariableExpression* (described below in FILE variable.pl) that allows substitution of a value into an expression.

The predicates that evaluate operations and meta-operations succeed, backtrack, and fail like any other Prolog predicate. If a term is not described by any operation or meta-

operation clause, it evaluates to itself. There is one special meta-operator, *val/1*, which returns its argument without evaluating it.

Here is a session with *eval/0* demonstrating a few examples of simple expressions involving the operators '+' and member and the meta-operators *val, obj, all,* and []. The lines starting with "—: " are prompts for the user to type in an expression; the lines ending with " ;" are prompts where the user typed ";" to look at additional values.

```
| ?- eval.
|: asdf.
asdf ;
|: -1.
-1 ;
|: 3.0.
3.0 ;
|: 2+3.
5 ;
|: 2+o.
|: val(2+3).
+(2,3) ;
|: all(val(4)).
[
4
] ;
|: member([4,5,2+3]).
4 ;
5 ;
5 ;
|: all(member([4,5,2+3])).
[
4,
5,
5
] ;
|: obj([name=araA,type=cds]).
[araA,cds,'E.coli'] ;
[araA,cds,'Salmonella'] ;
|: all(obj([name=araA,type=cds])).
[
[araA,cds,'E.coli'],
[araA,cds,'Salmonella']
] ;
|: exit.

yes
| ?-
```

atoms and numbers evaluate to atoms and numbers, which are printed out in fairly standard fashion. Operations that produce Boolean output produce the atoms *true* and *false*.

An arbitrary Prolog term that may or may not be a legitimate structure for use with eval is called a Value. Any Prolog term can be printed out by *eval/[0,1]*, *evalpp/[0,1]*, etc.

A Prolog list of values of arbitrary type is referred to as a *list*, and is printed out as either

"abcdefg"

or

[Value1, . . ValueN]

A list of two objects may have the special meaning of being a pair of objects.

obj(Object) is the structure used to store an object. It is printed out as *[Name, Type, optional Genome]. Object* is currently of the form *Name(Type, optional Genome)*, but that may change. Terms that are built out of Objects always include the object as *obj(Object)*.

annotate(Value, AssociatedValue) is the structure used to hold a value together with associated information. It is printed out as *annotate(Value, AssociatedValue)*

interval(Object, IntegerBeginning, IntegerEnd, AtomicDirection) is the structure used to hold an interval on an object. *Beginning* and *End* are inclusive, counted from 0 at the start of the object, and *Beginning* should always be =¡ *End*. *AtomicDirection* is one of *direct, complement, unknown.*

intervals(List) is a structure used to hold a sequence of intervals on an object. *List* should be nonempty. Each element of *List* should be a directed interval, that is, an interval such that *Direction* is not *unknown*. *intervals(List)* is printed out as *intervals(List)*.

dna_sequence(Length, ListOfCharacters) is the structure that represents a DNA or RNA sequence. A *dna_sequence* is printed out as *dna_sequence(Length, ListOfCharacters)*.

protein_sequence(Length, ListOfCharacters) is the structure that represents a peptide sequence. A *protein_sequence* is printed out as *protein_sequence(Length, ListOfCharacters)*.

mask(Length, ListOfCharacters) is used to store simple information relating to a sequence. For example, a mask showing secondary structure for a peptide sequence typically includes the characters

```
"." No info at this position
"-" No info at this position due to alignment
"H" This position is in an alpha-helix
"S" This position is in a beta-strand
"T" This position is in a turn.
```

point(Object, Offset) is the structure that represents a *point*. The *Offset* is relative to the start of the *Object* (counting from zero). A *point* is printed out as *point(Object, Offset)* .

vector(ListOfNumbersOrVectors) is a structure that represents a *vector* or an *array*. A *Vector* is printed out like *vector(ListOfNumbersOrVectors)* but in a more compact form.

SOME OF THE CURRENTLY IMPLEMENTED OPERATIONS AND META-OPS

FILE: simple.pl

simple operations are operations that act on a single *number* or *atom* or on a *pair* of numbers, and have been defined so that they extend to *vectors* and *arrays*. For example, *Vector*Number* means convert *Number* to a *vector* and do a pairwise multiply. Simple operations are "typed", and, if they return numeric results, then *ListOfValues* will be returned as *vector(ListOfValues)* and *ListOfVectors* will be returned as *vector(ListOfVectors)*. In additon, a *vector* on input is always converted to a *list*.

Here is a complicated example using the simple numeric operator *:

```
vector([vector([1,2]),vector([2,3])]) * [3,4]
-> [vector([1,2]),vector([2,3])] * [3,4]
-> [ vector([1,2]) * 3, vector([2,3]) * 4 ]
-> [ [1,2] * 3, [2,3] * 4 ]
-> [ [1*3,2*3], [2*4,3*4] ]
-> [ vector([4,6]), vector([8,12]) ]
-> vector([ vector([4,6]), vector([8,12]) ])
```

and with the Boolean operator and

```
and([true,false], true) ->
[and(true,true), and(false,true)] ->
[true, false]
```

Simple operations perform type checking - * expects its arguments to be numberic, *mod* expects its arguments to be *integer, not* expects its argument to be *boolean*, etc. If the types are not as expected, the evaluation will fail, otherwise it should always succeed. Therefore, expressions like *2+o* or *and(true,3)* or *2.0 mod 3.0* or *lowercase(-1)* fail to evaluate.

The simple operations are

```
boolean(Term)
is\_integer(Term)
number(Term)
and(Bool1, Bool2) % logical and
or(Bool1, Bool2) % logical or
xor(Bool1, Bool2) % exclusive or
not(Bool) % not
```

```
\+  Bool % not
Number1 =:= Number2 % numerical equality
Number1 =:= Number2 % numerical equality
Number1 < Number2
Number1 =< Number2
Number1 > Number2
Number1 >= Number2
Number1 =\= Number2 % numerical inequality
Number1 + Number2
Number1 - Number2
-Number2
Number1*Number2
Number1/NonzeroNumber
integer( Number) % rounds towards 0
abs(Number) % absolute value
min(Number1,Number2)
max(Number1,Number2)
Integer1 << Integer2 % binary shift left
Integer1 >> Integer2 % binary shift right
Integer1 // NonzeroInteger % integer divide
Integer1 div NonzeroInteger % integer divide, same as //
Integer1 mod NonzeroInteger
lowercase(Character)
uppercase(Character)
```

FILE: toplevel.pl

val(Term) evaluates to *Term*

once(Expr) evaluates *Expr* and cuts (eliminating backtracking).

Term1=Term2 returns *true* and unifies the two terms if *Term1* will unify with *Term2*, else returns *false*.

require(Value, VariableExpression) returns *Value* if *VariableExpression* returns *true* on it and fails otherwise.

FILE: variable.pl

variable(Value, VariableExpression) returns *VariableExpression* with *Value* substituted into it. A *VariableExpression* can be a normal expression or it can be of the form

```
PrologVar in ExpressionContainingPrologVar
```

FILE: list.pl

list(ListOfExpressions) returns *ListOfValues*, one for each *Expression*. It uses *get_vals* to do a "Cartesian product", so you needn't worry about the subexpressions being re-evaluated unnecessarily.

all(Expr) returns all values of *Expr* in a *list*.

all(Expr,Requirement) means *all(require(Expr,Requirement))*.

sort(List) sorts *List*.

foreach(List, VariableExpression) returns a *list* of values of *VariableExpression* on members on *List*, one for each member of *List*.

member(List) runs *list/1* on *List* and returns members.

size(List) runs *list/1* on *List* and returns the number of members.

FILE: vector.pl

vector(List) produces *Vector* if *List* is a list of *numbers*. *vector(Vector)* produces the *Vector* if it is a valid one (a list of *numbers*).

list(Vector) produces a *List* of the elements in *Vector*.

min(Vector) produces the smallest element in *Vector*.

max(Vector) produces the largest element in *Vector*.

total(Vector) produces the sum of the elements in *Vector*.

average(Vector) produces the average of the elements in *Vector* (the empty vector's average is 0).

FILE: annotate.pl

Expr with VariableExpression returns *annotate(ValueOfExpr, ValueOfVariableExpressionOnValue)* with full backtracking.

annotate(Value, VariableExpression) returns *annotate(Value, ValueOfVariableExpressionOnValue)* with full backtracking if *VariableExpression* succeeds on *Value* and *Value* otherwise.

unannotate(annotate(Value,Annot)) returns *Value*. In all other cases, *unannotate(NotAnnotation)* returns *NotAnnotation*.

annotation(annotate(Value,Annot)) returns *Annot*.

FILE: obj.pl

id(Atom) returns any *object* of name *Atom*. *id(List)* returns any *object* whose *Id* unifies with *List*

obj(List) returns an *object* satisfying the restrictions in *List*, where each element of *List* is one of

```
name = Expression
type = Expression
genome = Expression
```

where the *Expression* evaluates to an *atom*. For example,

obj([]) returns any *object*, *obj([name=member([thrA,thrB])])* returns any *object* with name *thrA* or *thrB*, and *obj([name=thrA,genome='E.coli'])* returns any *object* of name *thrA* in *genome* E.coli.

pathway(Object) returns *Object* if *Object* is a *pathway*, and fails otherwise. *pathway([])* returns any *pathway Object* with backtracking.

attributes(Object) returns the *list* of attributes of *Object*.

has_attribute(Object,Atom) returns *true* if *Object* has attribute *Atom*, false otherwise.

attribute(Object,Atom) returns the value of attribute *Atom* for *Object*, fails otherwise.

attribute_length(Object,Atom) returns the length of the string attribute *Atom* for *Object*, and fails otherwise.

name(Object) returns the name of *Object* (as an *atom*).

type(Object) returns the type of *Object* (as an *atom*).

genome(Object) returns the genome of *Object* (as an *atom*).

length(Object) returns the length of an object by calculating it from its binding or from a length attribute.

related(Object,Atom) returns *annotate(RelatedObject,Term)*. *related(annotate(Object,Annot),Atom)* returns *annotate(RelatedObject,Term)*.

*Object*Atom* returns a *RelatedObject* such that *Object* relates to *RelatedObject* by a *Term* with functor *Atom*.

*Object*Term* returns a *RelatedObject* such that *Object* relates to *RelatedObject* by *Term*.

relationships(Object) returns a *list* of all related *annotate(RelatedObject,Term)*.

FILE: point.pl

Point+Number returns *NewPoint* with the offset increased by *Number*.

Point-Number returns *NewPoint* with the offset decreased by *Number.*

prelower(Point) returns (by backtracking) any *point* which can be generated from *Point* by following precise bindings. The first one returned is always *Point.*

location(Point) returns a *point* which can be generated from *Point* by following all precise bindings (going all the way down).

FILE: interval.pl

complement(Direction) returns the complement of *Direction* (*direct -> complement, complement -> direct, unknown -> unknown).*

interval(Object) returns *interval(Object,0,LengthOfObjMinus1,direct).*

interval(Object,Direction) returns *interval(Object,0,LengthOfObjMinus1,Direction).*

interval(Point1,Point2) returns the same thing as *interval(Point1,Point2,direct).*

interval(Object,Beg,End) returns the same thing as *interval(Object,Beg,End,direct).*

interval(Point1,Point2,Direction) prelowers *Point1* and *Point2* to be on the same object and returns the corresponding *interval* with direction *Direction.* If the resulting interval has *Beg ¿ End* then *Beg* and *End* are reversed and the *Direction* is complemented.

interval(Object,Beg,End,Direction) returns *interval(Object,Beg,End,Direction).*

Interval ¿¿ Number shifts the "right" end of a directed *Interval* (the left end if its direction is *complement*) by *Number.* A positve *Number* increases the length of the interval.

Interval ¡¡ Number shifts the "left" end of a directed Interval (the right end if its direction is *complement)* by *Number.* A positve *Number* decreases the length of the interval (which is intuitively suspect and will probably be reversed at some point).

length(Interval) returns the length of *Interval.*

start(Interval) returns the *point* which is the beginning of a *direct interval,* the end of a *complement interval.*

start(Object) returns *start(interval(Object)).*

end(Interval) returns the point which is the end of a *direct interval,* the beginning of a *complement interval.*

end(Object) returns *end(interval(Object)).*

complement(Interval) returns *Interval* with the direction complemented.

prelower(Interval) returns (by backtracking) any *Intervals* which can be generated from *Interval* by following precise bindings. The first one returned is always *interval([Interval]).*

location(Interval) returns an *Intervals* which can be generated from *Interval* by following all precise bindings (going all the way down).

FILE: intervals.pl

intervals(Intervals) returns *Intervals.*

intervals(List) returns the *Intervals* structure containing members of *List.* Each element of *List* must be a valid *Interval* with known direction.

binding(Object) returns the precise binding of *Object* as an *Intervals* structure.

location(Object) returns *location(binding(Object)).*

location(Intervals) finds the location of each element of *Intervals* and appends the results into an *Intervals.*

list(Intervals) returns the List consisting of members of *Intervals.*

interval(Intervals, Beginning, End, Direction) returns the *SubIntervals* of *Intervals.*

complement(Intervals) returns the reverse complement of Intervals.

length(Intervals) returns the total length of *Intervals.*

start(Intervals) returns the start of the first element of *Intervals.*

end(Intervals) returns the end of the last element of *Intervals.*

prelower(Intervals) applies prelower to each member of *Intervals* and appends the result into an *Intervals* structure.

FILE: dna_sequence.pl

dna_sequence(Object) returns the DNA sequence of *Object,* either by looking at its *dna_sequence* attribute if its a *sequence_fragment,* or by following its binding and finding the *dna_sequence* of its binding.

dna_sequence(Interval or Intervals) prelowers *Interval* or *Intervals* to a *list* of *intervals* on sequence fragments, builds the substrings, and appends them to produce the *dna_sequence.*

dna_sequence(Dna_Sequence) returns *Dna_Sequence.*

dna_sequence(ListOfChars) returns *Dna_Sequence* corresponding to *ListOfChars.*

complement(Character) interprets Character as a DNA character and returns its complement.

complement(Dna_Sequence) returns the complement of *Dna_Sequence.*

protein_sequence(Dna_Sequence) returns *translate(Dna_Sequence).*

protein_sequence(Object) either calculates *translate(dna_sequence(Object))* or looks at the *peptide_sequence* or *protein_sequence* attribute of *Object.*

translate(Dna_Sequence) returns the result of translation on *Dna_Sequence* as a *peptide_sequence* using translate/3.

translate(Char1,Char2,Char3) interprets [Char1,Char2,Char3] as a codon and returns the appropriate protein character. If any of the Chars are ambiguous, the character "X" is returned. The stop codon is "*".

no_stop_codons_in(Protein_Sequence) returns *true* if there are no stop codons ("*" characters) in *Protein_Sequence* until the last character, *false* if there is one before that.

ok_start_codon(Protein_Sequence) returns *true* if the first character of *Protein_Sequence* is *Met*, *Leu*, or *Ile* (one of "MLImli").

FILE: sequence_ops.pl

matches(Value,AtomicPattern) evaluates *Value* to a *Dna_Sequence* or a *Protein_Sequence* and searches for hits to the punit pattern *AtomicPattern* in it. Returns a list of hits, each of which is of the form *annotate(Dna_or_Protein_Sequence, VectorOfStartPosOfPunits).*

common(List,Integer) evaluates each member of *List* to a *Dna_Sequence* or a *Protein_Sequence* and finds subsequences of length *Integer* that are common to each of them. *List* must have at least two members. Returns a *list* of *annotate(CommonSequence, ListOfVectorOfPositions).*

common(List,ListOfIntegers,Integer) is like *common/2* but *ListOfIntegers* is interpreted as a *list* of minimum numbers of occurrences in each sequence. *List* and *ListOfIntegers* should be of the same length with at least one element.

FILE: adjacent.pl

adjacent(ObjectExpression,BaseExpression) produces *[Object1,Object2]* such that *Object1* and *Object2* are values for *ObjectExpression* that are adjacent on a value of *BaseExpression.*

adjacent(ObjectExpression,BaseType,Genome) produces the same thing as *adjacent(ObjectExpression,obj([type=BaseType,genome=Genome]))* but is more efficient.

divergent(Type,Genome) produces all objects of type *Type* in genome *Genome* that are adjacent on an object of type *sequence_fragment* and are divergent.

divergent([Object1,Object2]) is *true* if *Object1* and *Object2* are divergent, and *false* otherwise.

convergent(Type,Genome) produces all objects of type *Type* in genome *Genome* that are adjacent on an object of type *sequence_fragment* and are convergent.

convergent([Object1,Object2]) is *true* if *Object1* ad *Object2* are convergent, and *false* otherwise.

parallel(Type,Genome) produces all objects of type *Type* in genome *Genome* that are adjacent on an object of type *sequence_fragment* and are parallel (parallel meaning in the same direction).

parallel([Object1,Object2]) is *true* if *Object1* ad *Object2* are parallel, and *false* otherwise.

between([Object1,Object2]) returns an interval between *Object1* and *Object2.*

FILE: structure.pl

has_direct_structure(Object) is true if *Object* has directly attached features delineating a secondary structure, false otherwise.

structure(Object) attempts to generate a secondary structure mask for an object by one of the following methods:

- getting it directly from the features attribute of the Object,
- getting it through an alignment from the relationship peptide_to_alignment and its inverse,
- getting it through an alignment from the relationship cds_to_alignment and alignment_to_peptide.

FILE: variants.pl

In this file *Sequence* means one of *Dna_Sequence, Protein_Sequence,* or *Mask.*

change_by(Sequence,ListOfChanges) returns a new *Sequence* operated on by *ListOfChanges.*

change_reverse(Sequence,ListOfChanges) returns a new *Sequence* operated on inversely by *ListOfChanges.*

changes_to(ListOfChanges,Sequence1,Sequence2) returns *true* if *ListOfChanges* converts *Sequence1* to *Sequence2; false* otherwise.

FILE alignment.pl:

align_seq(Object,Entry) returns the sequence of entry *Entry* (without dashes) in *alignment Object* by looking for *Name-CharList* in the *aligned_seqs* attribute.

aligned_seq(Object,Entry) is the same as *align_seq(Object,Entry)* but it leaves the dashes in.

align_by(Sequence,Object,Entry1,Entry2) treats *Sequence* as a *mask* or other sequence relating to *Entry1* (i.e, of the same length if the dashes in *Entry1* are removed) and converts it to something relating to *Entry2. Object* is an *alignment.*

align_by(Sequence, Object, Entry) inserts dashes in *Sequence* wherever there is a dash in *Entry* to produce a *mask* of the same length as the *alignment Object*. *unalign_by(Sequence, Object, Entry)* assumes that *Sequence* relates to the *alignment Object* and removes characters wherever *Entry* has a dash to produce a *Sequence* that relates to *Entry*.

END

DATE
FILMED
3/23/93