LabBase User Manual, Version 1.0

Steve Rozen Lincoln Stein
Nathan Goodman

{steve,lstein}@genome.wi.mit.edu

Whitehead Institute for Biomedical Research
One Kendall Square
Cambridge MA 02139

April 3, 1997

Abstract

LabBase is a generic database management system for implementation of laboratory information systems. This manual constitutes both a tutorial introduction to and a reference manual for LabBase.

1 Data Model

The primary abstractions supplied by LabBase are *materials* and *steps*: materials are things one works on in the laboratory, such as genetic markers or expressed sequence tags, and steps record both actions taken on a material and the information they generate. For example, at the Whitehead/MIT Center for Genome Research (CGR), the discovery of a mouse genetic marker involves many steps:

1. sequencing a small DNA fragment,
2. determining if it contains a simple repeat (microsatellite),
3. checking to see that we have not already used this sequence,
4. selecting polymerase chain-reaction (PCR) primers,

and so on. In this example, the material is the potential genetic marker, and the steps correspond to the actions 1 through 4. Our experiences with MapBase [3, 4, 2], as well as

other literature on laboratory information systems [5, 6], suggest that the notions of material and step are ubiquitous in these systems.

Additional material describing LabBase appears in [8] (which discusses the user view of LabBase), and [7] (which focuses on LabBase's implementation). The CGR's approach to workflow management and CGR's workflow-management software are described in [9]. Documentation on the perl API to LabBase is found with the LabBase distribution.

In LabBase, almost all information about a material is stored as part of a step. In our work with MapBase we found this to be a robust organization, because we want almost all attributes of a material to be associated with information about the processing step that produced them. For example, when we record the sequence of a potential marker, we also record the file that contains the output of the automated DNA sequencer, as well as the user that entered the sequence and the time a which the sequence was entered.

In LabBase, each material is associated with

- a *material kind*, such as `marker` or `est` (expressed sequence tag), and
- a history (list) of steps that record the chronology of real-world operations performed on that material.

A step is associated with a *step kind*. For example, step 1 above might have the step kind `sequence_step`. In addition to its step kind, a step consists of a set of *tag*/value pairs, for example `sequence='ACCG...'` or `sequence_file='/usr/local/seq_output/3851'`. Tags serve as attribute names, and each tag has a unique type that defines the values that can be associated with that tag. Examples of types for a tag are `'STRING'` or`'SET'('INTEGER')`.

The current version of LabBase does not automatically support ordering constraints among steps, though this is a capability that would probably be useful.

In addition to materials and steps, LabBase provides *data-dictionary entries* that record what sorts of materials, steps, and tags are known to the system.

The remainder of this manual is organized so that the information most users will require appears first, followed by information that is of interest to smaller numbers of users. Therefore, the next section describes how to pose read-only LabBase queries. Subsequent sections describe how to pose queries that update the database and how to define new kinds of materials and steps. Sections 4 and 5 act as a reference manual: they describes all the predicates available in LabBase and define the grammar of the LabBase query language.

2 Queries

Queries (and updates) are posed in a non-recursive logic programming language: the syntax and semantics are essentially those of a subset of Prolog [1]. The essential idea is that LabBase automatically defines certain predicates over the materials, steps, and tags constituting the state of a LabBase database. For example, if t is a tag, then the predicate t`(M,S)` binds `S` to the most recent value associated with t in the step history associated with the material bound to `M`.

Each query is a non-empty sequence of comma-separated *terms*, terminated by a period. For example

```
marker(M),sequence(M,Seq).
```

This query prints, for each marker, `M`, that has a sequence, the most recent sequence recorded for `M`. The two terms in this query are `marker(M)` and `sequence(M,Seq)`. To answer this query, each marker in the database is successively bound (assigned) to `M`, and then the `sequence` predicate binds `S` to the value associated with the most recent `sequence` tag in any step belonging to that particular marker.

Assuming there are four markers in the database, the output would look something like this:

```
M=B123,Seq=ACTTG...
M=Q443,Seq=GGATTG...
M=Z1N4,Seq=CCAAG...
M=L96,Seq=CTTTA...
YES
```

In this output, each variable is printed with a binding that made the query true. (A variable is an alphanumeric identifier beginning with a upper-case letter.) If the query has any bindings that make it true, LabBase prints `YES` after printing out the bindings. This is the default output. It is possible to tailor the form of the output by using the `only` predicate.

In the query above, `marker` is a material kind, and `sequence` is a tag. For any material kind k, the predicate k`(X)` binds `X` to each material of kind k. For any tag, t, the predicate t`(M,V)` requires `M` to be bound to a material. Then the predicate t binds `V` to each value associated with t in the most recent step of `M` that has an instance of t. Because the first argument to the predicate t must be bound, and the second argument can be free, we say that its *adornment* is [bf][1]

Tags can be set- (or list-) valued. For example, in the mouse genetic mapping schema at CGR, the tag `duplicate_id_set` is associated with a set of (string, integer) pairs, where each of the strings is the id of a marker, and each of the integers is a measure similarity between two sequences.

Sometimes many steps have the same tag; for example, every step has a `when` and a `who` tag. To restrict LabBase to the most recent step of a particular step kind, one can pose a query such as the following:

```
marker(M),sequence_step(material=M,sequence=Seq,when=W).
```

In this query, `sequence_step` is a step kind, and `Seq` and `W` are bound to the values of the `sequence` and `when` tag (respectively) of the most recent `sequence_step`. If, for example, there were no `sequence` tag associated with marker m's most recent `sequence_step`, m would not appear as a binding for `M` in the output of from this query.

Error Reporting When LabBase detects a syntax or evaluation error it prints out a string of the form

[1]Some Prolog manuals label an argument with '-' to indicate that it can be free (corresponding to LabBase's 'f'), and with '+' to indicate that it must be bound (corresponding to LabBase's 'b').

ERROR=*message* near line *n*, column *m*.

where *message* tries to explain what went wrong, *n* is a line number near the error, and *m* is the column position near the error. If you keep each term in a query to single line, *n* should be accurate. For evaluation errors, *m* is often at the end of the term which could not be evaluated. For a single query, it is possible, for some valid bindings to be printed before LabBase detects an evaluation error.

Queries can be interrupted with control-C (SIGINT), in which case an error message is printed. However, the SIGINT must directed to either the `lbserv` or `lbback` process. For more details see the man pages for these programs and also [7].

It is often desirable to have a query print out an error message when no bindings can be found to make a term true. The `insist` predicate can do this. For example

```
marker(M), sequence_step(material=M),
insist(sequence_step(material=M,sequence=Seq,when=W)).
```

which prints an error for those markers that have no `sequence` or `when` tag on in their most recent `sequence_step`. The `insist` predicate succeeds each time its argument succeeds, put prints out an error message if its argument does not succeed at least once.

Example Queries We close the section on queries with a few more example queries:

- Print all markers and selected `typing_step` information for those markers that have been typed on more than 2 typing panels.

```
marker(M),
count( typing_step(material=M,the_typing_panel=P), C),
C > 2,typing_step(material=M,the_typing_panel=Q).
```

- Print the number of markers that have been typed on more than 3 typing panels.

```
count( marker(M),
       count(typing_panel(P), typing_string(M,P,S), C),
       C > 3, D ).
```

- Find all markers with no step with a `sequence` tag.

```
marker(M),not(sequence(M,D)).
```

- Find all markers whose most recent `typing_step` *follows* a `mapmaker_step` (two versions).

```
marker(M),next(M,typing_step,mapmaker_step),
   mapmaker_step(material=M).


marker(M), mapmaker_step(material=M,when=W1),
   typing_step(material=M,when=W2), W2 > W1.
```

- Print all markers that have *any* typing step following a mapmaker step.

```
marker(M),all_steps(M,S1),all_steps(M,S2),
  mapmaker_step(S1),typing_step(S2),when(S1,W1),
  when(S2,W2),W2 > W1.
```

3 Writes

Tag associations are a key concept in the expression of writes to LabBase. A tag associations is a syntactic representation of a tag/value pair to be inserted into the database, and is of the form

t=v

where t is a tag (as determined by the set of tags in data dictionary), and v is a single value.

3.1 Inserts

Inserts of a new instance of a material have the form

`insert(`x`(`*args*`))`

where x is a material kind, and *args* is a comma-separated list of tag associations. The *args* must contain tag associations with tags x`_id` (whose value becomes the id of the newly created material) and `who`. The result is the creation of a new material with a first step of kind `create` with the specified tag/value pairs, and, in addition the a `created_material` tag associated with the newly created material itself.

Here is code to insert a new marker:

```
insert(marker(marker_id='A1', who='', when=1991:06:12:09:23:47)).
```

Here is code to create a screening panel:

```
insert(
  screening_panel(
   screening_panel_id='standard mouse screening panel',
   who=steve, when=1991:01:01:00:00:00
   screening_panel_abbreviations
       =[ob, cast, spr, a, b6, c3h, dba, balb, akr, non, nod, lp],
   screening_panel_members
       =['C57/6J-Ob/Ob', 'Castaneus', 'Spretus', 'A/J',
         'C57bl/6J', 'C3H/HEJ', 'DBA/2J', 'BALB/CJ', 'AKR/J',
         'NON', 'NOD', 'LP/J' ] )).
```

Inserts of a new step have the same form as insert of a new material instance, except that *x* must a step kind; *args* must contain a `who` tag, and must contain at least one tag of type `'MATERIAL_POINTER'`.

Here is an example of inserting a step of kind `external_choice_step`.

```
insist(marker_id(M,'L59')),
insert(external_choice_step(
  material=M,who='',when=1991:07:09:10:45:08,
  left_primer='ATGGGTACCACCCTATCATACCTA',
  right_primer='TTATACACTGATATCTTGATAGCC',
  product_length=48,
  external_choice_source='First WIBR Bluescript Library')).
```

3.2 Value Sets

Value sets (also called "material sets" for historical reasons) are sets of values that occur "at the top level"—not as part of a step. They are often used to hold partial results of a multi-statement query, or to represent the state of materials in a laboratory production line.

The predicates `value_set` and `temp_value_set` allow one to manipulate permanent and temporary value sets. Currently `value_set`'s are completely persistent: they survive the shutdown and restart of the database server, while `temp_value_set`'s disappear when the database server is shut down. The predicates `material_set` and `temp_material_set` are retained for backward compatibility with earlier releases of LabBase; they are *synonyms* for `value_set` and `temp_value_set`, respectively. In addition `lbserv` (see the `lbserv` man page) provides `temp_material_set`'s that are visible only within a single database session.

Section 4.2 describes how to use `value_set` and `temp_value_set` to query value sets. To insert a value into a value set one would write something like

```
marker_id(M,'D1118'),insert(value_set('My Set',M)).
```

which inserts the marker with id `D1118` into value set `'My Set'`. To create a temporary value set containing all markers with no sequence write

```
marker(M),not(sequence(M,S)),insert(temp_value_set(no_sequence,M)).
```

Here is an example of deleting a *particular* value from a material set

```
hybrid_screening_panel(P,panel_X),delete(value_set(good_panels,P)).
```

This query deletes the panel with `panel_X` from the set `good_panels`.

It will soon be impossible to delete an entire value set (or temporary value set) with name X using the form

```
delete(value_set(X))
```

Instead, use

```
delete_entire_value_set(X).
```

and

```
delete_entire_temp_value_set(X).
```

It is safe to insert to or delete from a value set while iterating over it. LabBase makes a copy of the value set before finding bindings for `value_set` [bf] or `temp_value_set` [bf].

3.3 Deletes and Updates

We expect deletes and updates to be rare, and used only to make corrections to data when no workflow step is needed to record the correction. Since MapBase currently offers no facilities for updates or deletes, their implementation in LabBase (except for value sets) has been deferred. Current practice is to edit the ASCII roll-forward logs to perform the updates and deletes when they are absolutely required.

4 Reference Manual

4.1 Types

The legal atomic types are: `'STRING'`, `'INTEGER'`, `'FLOAT'`, `'DATE'`, `'DNA_SEQUENCE'`, `'BOOLEAN'`, `'MATERIAL_POINTER'`, `'STEP_POINTER'`, and `'TERM'`. Every atomic type is a legal type. The legal type constructors are: `'LIST'`, `'SLIST'` and `'SET'`. `'SLIST'` is a space-efficient representation for lists that have mostly zero elements. (The zero elements are the empty string for `'STRING'`, 0 for `'INTEGER'` and `BOOLEAN`, 0.0 for `'FLOAT'`, the date corresponding to the Unix `time_t` 0 for `'DATE'`, the empty sequence for `'DNA_SEQUENCE'`, the empty set for `'SET'`, and the empty list for `'LIST'` and `'SLIST'`. An `'SLIST'` cannot have `'MATERIAL_POINTER'`, `'TERM'`, or `'STEP_POINTER'` elements.)

For any legal types *t1, ..., tn*, $n>0$, and for any legal type constructor U, U(*t1,..., tn*) is a legal type. Neither `'STEP_POINTER'` nor `'TERM'` can be used as part of the type of a tag.

4.2 Predicates

The available predicates are either built in (like `insert` and `not`), or are defined by the contents of the data dictionary.

As of April, 1997, it is possible to logically delete steps. The effect is to simply *mark* a step as deleted; no storage space is released, and no index entries for associated identifier tags are removed. It is possible to see logically deleted steps if the predicate `see_deleted_steps` has previously been evaluated in the current query.

Definition: A step is *visible* unless it is logically deleted or `see_deleted_steps` has been evaluated in the current query.

- *material_kind*(X) [f],[b] True if X is a material with kind *material_kind*.
- *tag*(R,*Id*) [fb], [bb] Provided *tag* is an identifier-tag, true if R is a material with *Id* associated with *tag* at some step. (An identifier-tag is one with the tag `id_tag` set to 1 in its data-dictionary entry.)
- *tag*(*M1,...,Mk,V*) [b...bf] (For cases not subsumed by the previous entry.) We must have k>0. Each *Mi* must be bound to a value of type `'MATERIAL_POINTER'`. Search the intersection of the histories of *M1,...,Mk* from the most recent step backward, until a visible step is found with tag *tag*. Bind V to the corresponding associated value. (The tag *tag* must not be a data-dictionary tag—one with the tag `id_tag` set to 1 in its data-dictionary entry).
- *tag*(S,V) [bf] Bind V to the value associated with *tag* in step S. (*tag* must not be a data-dictionary tag.)
- *step_kind*(S) [b], True if S is a step of kind *step_kind*.
- *step_kind*(*t1=x1,...,tn=xn*) [b...b] Predicates of this form allow greater precision than those based on tags: *step_kind* predicates can determine which tags are collected in a particular step, and can determine the tags with which materials are associated, thereby differentiating the roles of the materials. At least one *ti* must have type `'MATERIAL_POINTER'` and the associated *xi* must be bound. In the current implementation there can be no more than one unbound *ti* with type `'MATERIAL_POINTER'`. Bindings for unbound *xi*'s are found as follows:
  - Let S be the set containing exactly those steps, s, such that all of the following obtain:
    - s is visible.
    - The step kind of s is *step_kind*.
    - For all the *ti*'s of type `'MATERIAL_POINTER'`, s is the most recent step with any particular mapping of the *ti*'s to material pointers.
    - For every bound *xi*, *ti* is associated with *xi*'s value.
  - For each s in S do:

∗ Bind each free *xi* to the value associated with *ti* in *s* and return true.

- `+`, `-`, `*`, `/` These symbols are not the names of predicates, but can be the principal functor of a term argument to the `is` predicate.

- `>`,`<`,`>=`, `<=`, `<>`, `==` [bb] Binary comparison operators. `<>` is the inequality predicate, and `==` is the equality predicate, needed because underbound methods are not yet supported. May be used as infix operators.

- *Pattern*`~`*String* [bb] True if *String* contains the regular expression *Pattern*. *Pattern* and *String* must have type `'STRING'`. The syntax of *Pattern* is that of the Unix editor `ed`(1), except that newlines are allowed in *Pattern*; for documentation use the Unix command `man ed`. Please note that in order to get a backslash into *Pattern*, is necessary to use two backslashes in the quoted string. For example, `'^\\(.).*\\1$' ~ uveweru`. (The pattern matches any string of length at least 2 with the same character at the beginning and the end.)

- *term0*`;;` *term1* [bb] Evaluate *term0*. If there are any bindings that make it true, then return true for every binding that makes *term0* true. If *term0* is never true, then return true for every binding that makes *term1* true. (`;;` is a short-circuit or operator.) In the query-language grammar, `;;` binds more tightly than `,`.

- `all_steps(`*R*`,`*S*`)` [bf],[fb] True if *S* is a visible step associated with material *R*. The bindings are guaranteed to produced in the order in which the steps appear in *R*'s step history.

- `baseline_rusage` [] Create a baseline for measuring resource usage. (Resource usage is measured in lbback only.) See `incremental_rusage`.

- `cardinality`(*Set,Cardinality*) [bf] True if *Cardinality* is the cardinality of *Set*.

- `commit` [] Execute a commit in the underlying storage manager.

- `count`(*term1,..., termn,C*) [b...bf] Binds *C* to the number of times *term1,...,termn* is true. Any bindings produced in evaluating *term1,...,termn* are undone before evaluating the term following the `count` term.

- `db_size`(*Bytes*, *Blocks*) [ff] Bind *Bytes* to the number of bytes in the database, and *Blocks* to the number of blocks (as returned by Unix `stat`). Only the main database file or files are considered. Log files are excluded.

- `delete`(*term*) [b]
The only legal argument is a *term* of the form `value_set`(*string,value*) (or `material_set`(*string,valu*
)), where both *string* and *value* are bound. The result is to delete *value* from the value set with name *string*.

- `delete_entire_material_set`(*Set_name*) [b]
  Delete the material set with name *Set_name* (which must be a `'STRING'`). Always true (whether or not there is material set named *Set_name*).

- `delete_entire_temp_material_set`(*Set_name*) [b]
  Like `delete_entire_material_set`, except or temporary material sets.

- `delete_step`(*Step*) [b]
  Logically delete *Step*. (No error is raised if *Step* is already deleted.) See also `undelete_step`.

- `element`(*collection,V1,...,Vn*) [b*x*...*x*], *x* in {b,f} The *collection* must be of type `'LIST'`, `'SLIST'`, or `'SET'`. If *n*=1, each element of *collection* is bound (in order, if *collection* is of type `'LIST'` or `'SLIST'`) to *V1*. If *n*>1, each element of *collection* must be a list, [*x1,...,xn*] containing exactly *n* elements. For each element of *collection* (in order, if *collection* is of type `'LIST'`) bind *Vj* to *xj*.

- `exists`(*term1,..., termn*) [b...] True if *term1,..., termn* evaluated as a query (using any already-established bindings) is true. The argument query is not evaluated after one set bindings is found which makes it true; thus using `exists` might be more efficient than evaluating the argument query directly. It is an error to use the second kind of *step_kind* query within the argument of an `exists` predicate. In the current implementation the `exists` predicate can cause some storage leakage in the `lbback` server, so it should be used only when there is a compelling efficiency rationale.

- `gather_in_list`(*term1,..., termn,Element,List*) [b...bff] For each set of bindings for which *term1,..., termn* is true, take the value bound to *Element* and make it an element of *List*. The order of elements in *List* is determined by the order in which they are bound to *Element* by evaluating *term1,..., termn*.

- `gather_in_set`(*term1,..., termn,Element,Set*) [b...bff]
  For each set of bindings for which *term1,..., termn* is true, take the value bound to *Element* and make it an element of *Set*. All the values bound to *Element* must be comparable as if by `==`.

- `hex_escape`(*c*) [b] *Please use* `hex_escape_and_quote` *for all future coding.* `hex_escape` *is maintained only for backward compatibility.* When printing out strings from the database, replace the characters

  ```
  [,],{,},comma,newline,
  ```

  and *c* itself by *c* followed by the character's 2-digit hex code. The intent of this predicate is to make it easy to parse LabBase output by means of simple regular expressions. The effect of evaluating this predicate is limited to the current query.

- `hex_escape_and_quote`(*c*) [b] When printing out strings from the database, replace the characters

```
[,],{,},(,),comma,newline,single quote,
```

and *c* itself by *c* followed by the character's 2-digit hex code. In addition enclose all strings in single quotes (including material-kind names and principal material ids when printing a `MATERIAL_POINTER` and step-kind names when printing a `STEP_POINTER`).

The intent of this predicate is to make it easy to parse LabBase output by means of simple regular expressions. The effect of evaluating this predicate is limited to the current query. The local perl module `../site_lisp/LabBase.pm` is designed to turn the output of a query produced using this predicate into a normal perl5 data structure.

The format in which step "identifiers" are printed out is designed to make parsing by `../site_lisp/LabBase.pm` reliable; they are printed out in the form `step('<kind>''%2c'(when=<timestamp>))` (%2c is the hex code for comma) (or `step('<kind>''%2c'(when=<timestamp>' %5bdeleted%5d'))` if the step is logically deleted).

- `incremental_rusage`(*List*) [f] Bind *List* to a list of triples. The first element of each triple is a string describing the resource. The second element is the amount of resource used since the most recent evaluation of `baseline_rusage`, and the third element is (ususally) the amount of resource used since lbback was invoked. The first three elements of *List* are user CPU time, system CPU time, and elapsed time (as in the default for `csh`'s `time` command). Remaining elements (if any) are from `getrusage`(2).

- `insert`(*term*) [b] The `insert` predicate is discussed in various sections above.

- `insist`(*term*) [b] True whenever *term* is true. If there are no bindings that make *term* true, then `insist` prints out an error message.

- *V* `is` *expression* [fb] If *expression* is a non-term value, binds *V* to *expression*. If *expression* is a term with one of principal functors `+`, `-`, `*`, or `/` `is` evaluates expression according to C-like rules of arithmetic. In particular, `/` operating on 'INTEGER's yields an 'INTEGER', and the result of any expression containing a 'FLOAT' will be a 'FLOAT'. For the current implementation, you must use the standard (i.e. parenfix) syntax for `+`, `-`, `*`, and `/`, which are all binary operators.

- `ith`(*List,I,V*) [bbf], [bff] For adornment [bbf] the second bound argument must be of type 'INTEGER'. *List* must be of type `'LIST'`, `'SLIST'`, or `'SET'`. Bind each element, *ei*, in *List* (in order) to *V*, and bind *i* to *I*.

  Simple array subscription is performed by `ith`. For example, `ith([a,b,c],0,Z)` binds `Z` to `a`. Some more examples are:

```
1> ith([a,b,c],I,Z).
I=0,Z=a
I=1,Z=b
```

```
I=2,Z=c
YES
2> ith([[a,b],[1,2]],1,Z).
Z=[1,2]
YES
3> ith([[a,b],[1,2]],I,Z)
I=0,Z=[a,b]
I=1,Z=[1,2]
YES
```

- {left,right}_primer_sequence(*M,Primer_sequence*) [bf]
  Special-purpose predicates for the CGR. *Primer_sequence* becomes bound to the left or right primer sequence of *M* (a 'MATERIAL_POINTER') according to the rules detailed below. (Also see pcr_product_length.) These predicates signal an error in database state by binding *Primer_sequence* to the empty string. They signal an incompatible database schema by the usual LabBase error mechanism (which aborts all query processing for the current query).
  - Let *A* be left or right.
  - Let *s* be an arbitrary step in the history of *M* such that *s* contains either the *A*_start or the *A*_primer tag, and such that no later step contains either tag.
  - Signal an error in database state if *s* contains *both* *A*_start and *A*_primer.
  - If *s* contains *A*_start then
    - Let *i* be the value associated with *A*_start, and let *j* be the value associated with *A*_length tag in *s*. Signal an error in database state if *A*_length is absent in *s*.
    - Find the most recent insert_start, insert_length and sequence tags in any step at or before *s*. If any of these tags is missing signal an error in database state.
    - Let *q* be the value associated with insert_start, let *r* be the value associated with insert_length, and let, *S*, bet the value associated with the sequence tag, respectively.
    - For left_primer_sequence, bind *Primer_sequence* to the substring of *S* starting at position *i* (0-based) and of length *j*.
    - For right_primer_sequence, bind *Primer_sequence* to the reverse complement of the substring of *S* starting at *i*-*j*+1 and of length *j*.
  - Otherwise, if *s* contains an *A*_primer tag, bind its value to *Primer_sequence*.

- length(*X,Length*) [bf] True if *Length* is the length of *X* (which must be of type 'LIST', 'SLIST', 'STRING', or 'DNA_SEQUENCE'.

- make_list(*V1,...,Vn,L*) [b...bf] Bind *L* to the list containing *V1,...,Vn,* where *n* must be greater than 0 and each *Vi* must be a value or a variable bound to a value.

- `material_set`(*Set_name*) [f],[b] True if *Set_name* (which must be a `'STRING'`) is a material set.

- `material_set`(*Set_name*, *R*) [bf],[bb] True if material *R* is in permanent material set *Set_name* (a `'STRING'`). With adornment bb, can also be the argument to `insert` and `delete`.

- `next`(*M*,*Step_kind1*,*Step_kind2*) [bbb] True iff the step history of material *M* contains a step of *Step_kind1* and the most recent step of kind *Step_kind1* in the history is *not* followed by a step of kind *Step_kind2*. The value of `next` is false if the most recent steps of both kinds have the same `when` value

- `not`(*term1,..., termn*) [b...b] True if *term1,..., termn* has no true bindings. Any bindings produced in evaluating *term1,..., termn* are undone before evaluating the term following the `not` term.

- `only`(*Variable1,...,Variablen*) (Adornment is irrelevant.) Cause the binding of *only* the argument variables to be printed. The effects of the side-effecting predicate persist during the evaluation of the query. Multiple evaluations cause the union of the variables in all the evaluations to be printed.

- `pcr_product_length`(*M*,*Length*) [bf] A special-purpose predicate for the CGR. *Length* becomes bound to the PCR product size (in base pairs) of *M* (a `'MATERIAL_POINTER'`) according to the rules detailed below.

  (Also see `left_primer_sequence`, and `right_primer_sequence`.)

  `pcr_product_length` binds *Length* to either:

  - The value associated with the most recent `product_length` tag in *M*'s history, provided that there is no more or equally recent `left_start` or `right_start` tag.
  - Otherwise the PCR product length computed from the most recent `left_start` and `right_start` tags in *M*'s history.

  `pcr_product_length` binds *Length* to -1 if there is a `left_start` (or `right_start`) tag at least as recent as the most recent `product_length` tag, but no `right_start` (or, respectively, `left_start`) tag. `pcr_product_length` signals an incompatible database schema by the usual LabBase error mechanism (which aborts all query processing for the current query). `pcr_product_length` does *not* treat as an error nonsensical values for `left_start` and `right_start`; for example if `right_start` is less than `left_start` `pcr_product_length` will silently bind *Length* to a non-positive value.

- `polymorphic`(*Avg_allele_sizes,I0,I1,Delta*) [bbbb] *Avg_allele_sizes* is a list of integers, each at least -1. Yield true if the *I0*th and *I1*th (0-based) elements of *Avg_allele_sizes* differ by at least *Delta* and neither element is -1, or if one but not both of *I0* and *I1* is -1. (See also `strictly_polymorphic`).

- `reversec`(*X,Y*) [bf] [fb] Bind the free argument to the reverse complement of the bound argument. The bound argument must have type `'STRING'` or `'DNA_SEQUENCE'`, and an error is reported if a `'STRING'` argument contains any character other than A, T, G, C, or N. The free argument becomes bound to a value of type `'DNA_SEQUENCE'`.

- `right_primer_sequence`(*M, Primer_sequence*) [bf] A special-purpose predicate for the CGR. Documented under `left_primer_sequence`.

- `see_deleted_steps` [b]
  Make logically deleted steps visible for the remainder of the current query.

- `strictly_polymorphic`(*Avg_allele_sizes,I0,I1,Delta*) [bbbb] *Avg_allele_sizes* is a list of integers, each at least -1. Yield true if the *I0*th and *I1*th (0-based) elements of *Avg_allele_sizes* differ by at least *Delta* and neither element is -1. (See also `polymorphic`).

- `substring`(*String,Start,Length,Substring*) [bbbf] Bind *Substring* to the substring of *String* starting at (0-based) index *Start* and of length *Length*. It is an error if any part of the specified *Substring* falls outside of *String*. *String* must be bound to a value of type `'STRING'` or `'DNA_SEQUENCE'`, and *Substring* becomes bound to a value of the same type as that of *String*.

- `tag_value`(*step,List*) [bf] Bind *List* to a list containing all the tag-and-value pairs (each represented as a list) in `'STEP_POINTER'` *step*.

- `tag_value`(*material,List*) [bf] Bind *List* to a list containing all the *most-recent* tag-and-value pairs (each represented as a list) in `'MATERIAL_POINTER'` *material*.

- `temp_material_set`(*Set_name*) [f],[b] True if *Set_name* (which must be a `'STRING'`) is a temporary material set.

- `temp_value_set`(*Set_name*, *R*) [bf],[bb] Same as `material_set`, except that the material set dissappears at the end of a database session—that is, when `lbserv` is used to shut down the database server.

- `undelete_step`(*Step*) [b]
  Undo the logical deletion of *Step*. (No error is raised if *Step* is not logically deleted.)

- `var`(*X*) [b],[f] True iff *X* is an unbound variable.

4.3 Built-In Tags and Step Kinds

Built-In Tags Certain tags are required by LabBase, and are built-in. Some are *data-dictionary tags*, because they are tags in data-dictionary entries. These are `dd_identifier`, `type`, `dd_tag`, `id_tag`. The other built-in tags are: `who`, `when`, `created_material`, `material`.

Built-In Step Kinds The step kind `create` is built-in.

4.4 Proposed but Unimplemented Predicates

LabBase users who require predicates not available above should contact:

`steve@genome.wi.mit.edu`.

5 Grammar

The grammar of the LabBase query language is similar to those of standard Prologs, with some exceptions. The most notable exceptions are:

- LabBase does not support the definition of rules.
- All infix operators are right associative and all have the same precedence, and parentheses cannot be used to group. When in doubt, use the parenfix syntax: for example, `;(3 < 5, foo(X))` rather than `3 < 5 ; foo(x)` (equivalent to `<(3,;(5,foo(x)))`). The infix operator `=` plays a special role in LabBase, in arguments to the `insert` predicate and in certain queries based on step kinds. It is not possible to define new operators in the LabBase query language; they are all built in.
- Beware of the following:

  ```
  3 < 4.
  ```

  The term above compares the integer 3 with the *float* 4. The dot does *not* end a query.

5.1 Lexical Elements

Null characters (ASCII code 0) are illegal anywhere in the input. Control-A (ASCII code 1) is used as a synchronization character between clients and the LabBase server: it terminates the current query in any context. The perl API sees to it that each query sent from a client is terminated with a control-A.

5.1.1 String Literals

A string literal is one of the following:

- A (maximal) sequence of the characters `[a-zA-Z0-0_]` beginning with a lower-case letter (`[a-z]`).
- A sequence of the characters from the set `+-*\^<>=~:.?@#$&`.
- A sequence of characters enclosed in single quotes.

Escapes are available within quoted strings: `\t`, `\n`, `\b` expand to tab, newline, and back space, respectively, `\`*c* causes the whitespace character *c* to be ignored, for example `'ab\ c'` is another way of writing `'abc'`. Every other character preceded by a `\` is replaced by itself, so `'\foo'` is another way of writing `'foo'`. To write a the string `ab'c` write `'ab''c'` or `'ab\'c'`. Octal escapes are not yet implemented—you cannot write `'\1'` to get a string consisting of the character control-A, but they can be implemented quickly on request.

6 User Extension

It is possible to extend LabBase by adding new builtin predicates or new types to LabBase executable. Please refer to [7] for an overview of LabBase's system architecture before reading the material in this section.

6.1 Adding New Builtin Predicates

If you want to add a builtin that takes terms as arguments, you can find examples in `builtin1.C`. The remainder of this section discusses a simple interface for adding builtins that use only values (as opposed to terms) as input or output arguments. Examples can be found in `builtin2.C`.

To add a builtin predicate one must write a single function if the predicate produces no more than 1 set of bindings for a single input, or two functions if the predicate can produce more than 1 set of bindings for a single input.

To write a single-function predicate:

1. Let *first* be the name of the function implementing the predicate. (The name of the predicate is determined later, and need not be the same.) This function will set up the first bindings (if any). It returns an object of type `QL_Iterator_State*`, but a 1-function predicate should return 0 or 1 (suitably cast), to indicate whether the predicate is true or not.

2. Put the following line in a header file to be included by `dd_ops.C` (and which must also include `builtin.H`).

   `BUILTIN_FIRST_EXTERN(`*first*`);`

3. In the file that will contain the function definition, create a static, file-global array variable, *types*, that contains the types expected of an initial segment of the bound arguments:

   `static Value::tv_type_id` *types*`[] = {Value::`*T1*`,...,Value::`*Tk*`};`

   Each *Ti* is a type from the `enum tv_type_id` defined in `value.H`. The variable *types* must contain at least one value.

4. Use the macro (defined in `builtin.H`):
   `BUILTIN_FIRST_DEFINITION(`*first*`,`

bound_number, *free_number*, *types*, *type_number*, *unused*)
{
function_body
}

The arguments to `BUILTIN_FIRST_DEFINITION` are:

(a) *first* The C++ identifier for a global function to be created.

(b) *bound_number* The number of bound arguments expected. Use -1 to indicate a variable number of bound arguments.

(c) *free_number* The number of free arguments expected. Use -1 to indicate a variable number of free arguments.

(d) *types* The array of `Value::tv_type_id`s defined above. Use `Value::UNDEFINED` to indicate that any type is acceptable. Otherwise an error will be generated if the corresponding bound argument is of a different type.

(e) *type_number* The number of elements in *types*, which must at least 1 unless *bound_number*=0. If *type_number* is less than the number of bound arguments found at run time, the last element in *types* is taken to specify the required type of the remaining bound arguments.

(f) *function_body* Computes whether the predicate is true or false, and produces bindings for any unbound arguments. The following variables are available in *function_body*:

i. `QL_Term* t` is the term being evaluated; `t` would be the first argument in a call to `ql_eval_error`, which is how *first* should report a user error.

ii. `int m` is the number of bound arguments.

iii. `int n` is the number of free arguments.

iv. `const Value in_tv[]` has length `m`. The contents are the bound arguments. It is guaranteed that each element of `in_tv` is the address of a legal `Value`.

v. `Value out_tv[]` has length `m`. The elements of this array must be assigned to create the update bindings if *first* returns a non-0 result.

5. Add the following line to function `DD_Ops::add_builtins()` in `dd_ops.C`:

`add_builtin`(*predicate_name*, *predicate_adornment*, *first*);

where

- *predicate_name* is of type `char*`, and is the name by which the predicate will be known in the LabBase query language.
- *predicate_adornment* is either the C++ string `"*"`, indicating a variadic predicate, or a (possibly empty) string accepted by the regular expression `[bf]*`, which denotes the expected adornment of a predicate of fixed arity.
- *first* is the C++ identifier discussed above.

6. **Garbage Collection** Values produced using the static member functions *Value_Class*::`make` are automatically garbage collected. The constructors for `Value` classes are not public.

7. **Reporting User Errors** Use `ql_eval_error(QL_Term*, char*m1,...)` to report user errors. (There can be up to a total of 5 `char*` arguments.) The error message reported to the user is the concatenation of the `char*` arguments, followed by the text `" predicate '`*name*`' near line` *line_number*`, column` *column_number*`"`. As a consequence, most error messages should end in a preposition, such as "in" or "for". For example, calling `ql_eval_error` with `char*` arguments `"a total "` and `"disaster occurred in"`, the user will see `ERROR=a total disaster occurred in predicate` *name* `near line` *line_number*`, column` *column_number*. Do not end the error message with a period.

8. **Stick with class `Value`** You should be able to do everything you need to do in class `Value`. For most elements, *X* of the `enum Value::tv_type_id` there is a member function `down_to_`*X* that should be used for down-casts. These functions will call `abort` if the downcast is illegal based on the type of the `Value`. Also, for collection classes (lists and sets), stick with the class `TV_Collection_Value`, (which is a `Collection_Value` of `Values`). Lists and sets have the *same* C++ class! but do differ in how they respond to `Value::tv_type()`, and some member functions raise run-time errors if invoked on sets as opposed to lists (à la Smalltalk).

To write a two-function predicate, follow all steps for writing a one-function predicate, except step 5. In addition:

1. Let *next* be the name of the second function implementing the predicate. The *first* function must return some state that the `next` function receives as an argument, and that allows *next* to set up the next set of bindings (if any).

2. Put the following line in a header file to be included by `dd_ops.C` (and which must also include `builtin.H`).

   `BUILTIN_NEXT_EXTERN(`*next*`);`

3. Place the macro
   `BUILTIN_NEXT_DEFINITION(`*next, unused*`) {` *function_body* `}`
   in the same file as the use of the macro `BUILTIN_FIRST_DEFINITION`. Within *function_body* the variables `t`, `n`, and `out_tv` have the same types and semantics as for the *function_body* argument of `BUILTIN_FIRST_DEFINITION`. In addition, variable `s` has type `QL_Iterator_State*`, and contains state information returned by a call to *first* or a previous call to *next*.

4. Add the following line to function `DD_Ops::add_builtins()` in `dd_ops.C`:

   `add_builtin(`*predicate_name*`,` *predicate_adornment*`,` *first*`,` *next*`);`

   where

- *predicate_name, predicate_adornment,* and *first* are as for the 1-function case, and
- *next* is the C++ identifier introduced above.

Please add appropriate regression tests to the directory `$LABBASE_ROOTDIR/src/lbback/tests`, and add a description of the new builtin to this document. Regression tests end in extension `.test`, and the baseline output is stored files with the same name and extension `.last`.

6.2 Adding New Types

This section is still *incomplete* and has not been debugged much. Please contact the authors for help in adding a new type.

All new types must be derived (in the C++ sense) from class `Value`. Let the new class be *New_Class*, and let the corresponding enum value for `Value::tv_type_id` be *NEW_CLASS*. For consistency with existing LabBase types, the enum name should be all upper-case, since the LabBase user will see this as the type name.

- In `value.H`:
  - Add the forward declaration "`class` *New_Class*`;`" just before the definition of class `value`.
  - In the definition of class `Value`:
    - Add the enum value named *NEW_CLASS* to the definition of `enum tv_type_id`.
    - Add the declaration "*New_Class* `*down_to_`*NEW_CLASS*`() const;`"
  - Add the necessary declaration of *New_Class*. This declaration should offer the following public member functions:
    - `compare` [more to come]
    - `print` [more to come]
    - `zero` [more to come]
    - `make` [more to come]
    - `save` [more to come]
    - [more to come]
- In `value.C`:
  - Define the function `down_to_`*New_Class*`()` by the top-level macro call DEFINE_DOWNTO_FUNCTION(STEP_POINTER,Step)
  - Add the macro call `V_PRINT(`*New_Class*`)` to the body of `Value::print()`.
  - Add the macro call `V_SAVE(`*New_Class*`)` to the body of `Value::save()` (assuming you want to save values of type *New_Class* in the database).
  - Add the macro call `V_COMPARE(`*New_Class*`)` to the body of `Value::compare()`.

  - Add an appropriate branch to the `switch` in the body of `delete_value()` to call the correct destructor for *New_Class*.
  - As necessary, add definitions for the member functions of *New_Class*. (These definitions can also be in a separate file, of course.)

- In `dd_ops.C`:
  - Add the appropriate call to `add_type_constructor` to the body of `DD_Ops::add_type_constructors`, to enable lbback to parse type expressions. The optional second argument to `add_type_constructor`, if non-0, indicates that the new type is in fact a type constructor that takes arguments.

- In `ql_term.C`:
  Fix the type-checking routines as necessary.

- In the implementations of builtin predicates:
  If any existing builtin predicates should operate on the new type, their implementations must be changed.

You will probably want to add a new set of regression tests for the new type.

References

[1] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, 1987.

[2] N. Goodman. An object oriented DBMS war story: Developing a genome mapping database in C++. In W. Kim, editor, *Modern Database Management: Object-Oriented and Multidatabase Technologies.* ACM Press, 1994.

[3] N. Goodman, M.-P. Reeve, and L. Stein. The design of MapBase: An object oriented database for genome mapping, Dec. 1992. Whitehead Institute for Biomedical Research, technical report.

[4] N. Goodman, S. Rozen, and L. Stein. Requirements for a deductive query language in the MapBase genome-mapping database. In R. Ramakrishnan, editor, *Applications of Logic Databases*, pages 259–278. Kluwer, 1994. Available at `ftp://genome.wi.mit.edu/pub/papers/Y1994/requirements.ps`.

[5] A. R. Kerlavage, M. D. Adams, J. C. Kelly, M. Dubnick, J. Powell, P. Shanmugam, J. C. Venter, and C. Fields. Analysis and management of data from high-throughput expressed sequence tag projects. In T. N. Mudge, V. Milutinovic, and L. Hunter, editors, *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, volume 1, pages 585–594. IEEE Computer Society Press, Jan. 1993.

[6] A. R. Kerlavage, W. FitzHugh, A. Glodek, J. Kelley, J. Scott, R. Shirley, G. Sutton, M. Wai-Chiu, O. White, and M. D. Adams. Data management and analysis for high-throughput DNA sequencing projects. *IEEE Engineering in Medicine and Biology*, Nov./Dec. 1995.

[7] S. Rozen, L. Stein, and N. Goodman. Constructing a domain-specific DBMS using a persistent object system. In M. Atkinson, V. Benzaken, and D. Maier, editors, *Persistent Object Systems*. Springer-Verlag and British Computer Society, Workshops in Computing Series, 1995. Presented at POS-VI, Sep. 1994. Available at `ftp://genome.wi.mit.edu/pub/papers/Y1994/labbase-design.ps.Z`.

[8] S. Rozen, L. Stein, and N. Goodman. LabBase: A database to manage laboratory data in a large-scale genome-mapping project. *IEEE Engineering in Medicine and Biology*, 14(6):702–709, Nov./Dec. 1995. Preprint available at `ftp://genome.wi.mit.edu/pub/papers/Y1995/labbase.ps.gz`.

[9] L. Stein, S. Rozen, and N. Goodman. Managing laboratory workflow with LabBase. In *Proceedings of the 1994 Conference on Computers in Medicine (CompMed94)*. World Scientific Publishing Company, 1995. In press. Available at `ftp://genome.wi.mit.edu/pub/papers/Y1995/workflow.ps.Z`.