Datalog User Manual

Next: Introduction

This manual documents Datalog version 2.2, a lightweight deductive database system.

Contents

- Introduction: Why `datalog`?
- Tutorial: A quick start.
- Interpreter: Invoking and using.
- Primitives: Predicates implemented by code.
- Syntax: Some language details.
- Import/Export: Using tab separated values.
- Extensions: Adding primitive predicates.
- Stand-Alone: Using the Datalog module in Lua.
- Library: Embedding the Datalog interpreter.
- Copying: Terms for copying the software.
- Index: A concept index.

The software described in this manual was written by John D. Ramsdell of The MITRE Corporation. The work was supported by the MITRE-Sponsored Research Program.

Next: Tutorial, Previous: Top, Up: Top

1 Introduction

The Datalog package contains a lightweight deductive database system. Queries and database updates are expressed using Datalog—a declarative logic language in which each formula is a function-free Horn clause, and every variable in the head of a clause must appear in the body of the clause. The use of Datalog syntax and an implementation based on tabling intermediate results, ensures that all queries terminate.

The components in this package are designed to be small, and usable on memory constrained devices. The package includes an interactive interpreter for Datalog, and a library that can be employed to embed a small deductive database into C programs. The library uses the tabled logic programming algorithm described in “Efficient Top-Down Computation of Queries under the Well-Founded Semantics”, Chen, W., Swift, T., and Warren, D. S., *J. Logic Prog.*, Vol. 24, No. 3, Sep. 1995, pp. 161-199. Another

important reference is “Tabled Evaluation with Delaying for General Logic Programs”, Chen, W., and Warren, D. S., *J. ACM*, Vol. 43, No. 1, Jan. 1996, pp. 20-74. Datalog is described in “What You Always Wanted to Know About Datalog (And Never Dared to Ask)”, Stefano Ceri, Georg Gottlob, and Letizia Tanca, *IEEE Transactions of Knowledge and Data Engineering*, Vol. 1, No. 1, March 1989.

Next: Interpreter, Previous: Introduction, Up: Top

2 Tutorial

Datalog is introduced by showing a sequence of interactions with a Datalog interpreter. The interpreter is started with the `datalog` command.

```
$ datalog
Datalog 2.2

>
```

Facts are stored in tables. If the name of the table is ‘`parent`’, and ‘`john`’ is the parent of ‘`douglas`’, store the fact in the database with this:

```
> parent(john, douglas).
```

Each item in the parenthesized list following the name of the table is called a *term*. A term can be either a logical variable or a constant. Thus far, all the terms shown have been constant terms.

A query can be used to see if a particular row is in a table. Type this to see if ‘`john`’ is the parent of ‘`douglas`’:

```
> parent(john, douglas)?
parent(john, douglas).
```

Type this to see if ‘`john`’ is the parent of ‘`ebbon`’:

```
> parent(john, ebbon)?
```

The query produced no results because ‘`john`’ is not the parent of ‘`ebbon`’. Let's add more rows.

```
> parent(bob, john).
> parent(ebbon, bob).
```

Type the following to list all rows in the ‘`parent`’ table:

```
> parent(A, B)?
parent(john, douglas).
parent(bob, john).
parent(ebbon, bob).
```

Type the following to list all the children of ‘`john`’:

```
> parent(john, B)?
parent(john, douglas).
```

A term that begins with a capital letter is a logical *variable*. When producing a set of answers, the

Datalog interpreter lists all rows that match the query when each variable in the query is substituted for a constant. The following example produces no answers, as there are no substitutions for the variable ‘A’ that produce a fact in the database. This is because no one is the parent of oneself.

```
> parent(A, A)?
```

A deductive database can use rules of inference to derive new facts. Consider the following rule:

```
> ancestor(A, B) :- parent(A, B).
```

The rule says that if *A* is the parent of *B*, then *A* is an ancestor of *B*. The other rule defining an ancestor says that if *A* is the parent of *C*, *C* is an ancestor of *B*, then *A* is an ancestor of *B*.

```
> ancestor(A, B) :- \
>>   parent(A, C),  \
>>   ancestor(C, B).
```

In the interpreter, a line can be continued by ending it with the backslash character.

Rules are used to answer queries just as is done for facts.

```
> ancestor(A, B)?
ancestor(ebbon, bob).
ancestor(ebbon, douglas).
ancestor(bob, john).
ancestor(john, douglas).
ancestor(ebbon, john).
ancestor(bob, douglas).
> ancestor(X,john)?
ancestor(bob, john).
ancestor(ebbon, john).
```

A fact or a rule can be retracted from the database using tilde syntax:

```
> parent(bob, john)~
> parent(A,B)?
parent(john, douglas).
parent(ebbon, bob).
> ancestor(A,B)?
ancestor(john, douglas).
ancestor(ebbon, bob).
```

Unlike Prolog, the order in which clauses are asserted is irrelevant. All queries terminate, and every possible answer is derived.

```
> q(X) :- p(X).
> q(a).
> p(X) :- q(X).
> q(X)?
q(a).
```

Next: Primitives, Previous: Tutorial, Up: Top

3 Interpreter

The `datalog` program provides access to a deductive database in both interactive or batch mode. In

batch mode, the program reads a Datalog program, prints the answers to its query, and then exits. In interactive mode, the program treats each line of text it reads as a Datalog program, prints the answer to its query, and then prompts for more input.

The `datalog` interpreter accepts the following options:

```
$ datalog -h
Usage: datalog [options] [file]
Options:
  -o file -- output to file (default is standard output)
  -i      -- enter interactive mode after loading file
  -l file -- load extensions written in Lua
  -t      -- print output as tab separated values
  -v      -- print version information
  -h      -- print this message
Use - as a file name to specify standard input
```

When in interactive mode, if the first character of a line of input is the equals sign, the remaining text on the line is used as the name of a file that is loaded into the interpreter. An incomplete line of text can be continued by ending the line with the backslash character.

Next: Syntax, Previous: Interpreter, Up: Top

4 Primitive Predicates

The interpreter provides an equality predicate with infix notation. Two terms are equal if both terms are the same constant, or one is a constant and the other is a variable.

```
> 1 = 2?
> 1 = 1?
1 = 1.
> X = 1?
1 = 1.
> X = X?
```

Next: Import/Export, Previous: Primitives, Up: Top

5 Syntax

In Datalog input, whitespace characters are ignored except when they separate adjacent tokens or when they occur in strings. Comments are also considered to be whitespace. The character ‘`%`’ introduces a comment, which extends to the next line break. Comments do not occur inside strings.

The characters in Datalog input are collected into tokens according to the rules that follow. There are four classes of tokens: punctuations, variables, identifiers, and strings. The punctuation tokens are: ‘`(`’, ‘`,`’, ‘`)`’, ‘`=`’, ‘`:-`’, ‘`.`’, ‘`~`’, ‘`?`’, and ‘`"`’.

A variable is a sequence of Latin capital and small letters, digits, and the underscore character. A variable must begin with a Latin capital letter.

An identifier is a sequence of printing characters that does not contain any of the following characters: ‘`(`’, ‘`,`’, ‘`)`’, ‘`=`’, ‘`:`’, ‘`.`’, ‘`~`’, ‘`?`’, ‘`"`’, ‘`%`’, and space. An identifier must not begin with a Latin capital

letter. Note that the characters that start punctuation are forbidden in identifiers, but the hyphen character is allowed.

A string is a sequence of characters enclosed in double quotes. Characters other than double quote, newline, and backslash can be directly included in a string. The remaining characters can be specified using escape characters, '`\"`', '`\n`', and '`\\`' respectively.

Other escape characters can be used to improve the readability of the input. If a string is too long to fit conveniently on one line, all but the final line containing the string can be ended with a backslash character, and each backslash newline pair is ignored. The character escape codes from the C programming language are allowed—'`\a`', '`\b`', '`\f`', '`\n`', '`\r`', '`\t`', '`\v`', '`\'`', and '`\?`'. The numeric escape codes consist of one, two, or three octal digits. Thus the ASCII character newline is '`\012`', and zero is '`\000`'. A printed Datalog string is also a string constant in C.

5.1 Literals

A *literal*, is a predicate symbol followed by an optional parenthesized list of comma separated terms. A *predicate symbol* is either an identifier or a string. A term is either a variable or a constant. As with predicate symbols, a *constant* is either an identifier or a string. The following are literals:

```
parent(john, douglas)
zero-arity-literal
aBcD(-0, "\n\377")
"="(3,3)
""(-0-0-0,&&&,***,"\0")
```

5.2 Clauses

A *clause* is a *head* literal followed by an optional body. A *body* is a comma separated list of literals. A clause without a body is called a *fact*, and a *rule* when it has one. The punctuation '`:-`' separates the head of a rule from its body. A clause is *safe* if every variable in its head occurs in some literal in its body. The following are safe clauses:

```
parent(john, douglas)
ancestor(A, B) :-
    parent(A, B)
ancestor(A, B) :-
    parent(A, C),
    ancestor(C, B)
```

5.3 Programs

A Datalog reader consumes a Datalog program. A *program* is a sequence of zero or more statements, followed by an optional query. A statement is an assertion or a retraction. An *assertion* is a clause followed by a period, and it adds the clause to the database if it is safe. A *retraction* is a clause followed by a tilde, and it removes the clause from the database. A *query* is a literal followed by a question mark. The effect of reading a Datalog program is to modify the database as directed by its statements, and then to return the literal designated as the query. If no query is specified, a reader returns a literal know to have no answers. The following is a program:

```
edge(a, b). edge(b, c). edge(c, d). edge(d, a).
path(X, Y) :- edge(X, Y).
```

```
path(X, Y) :- edge(X, Z), path(Z, Y).
path(X, Y)?
```

Next: Extensions, Previous: Syntax, Up: Top

6 Import/Export

Tab separated values is a simple text format for a database table. Each record in the table is one line of text. Fields within the record are separated by the tab character. The script below translates a file of tab separated values into Datalog syntax.

```
#! /bin/sh
sed 's/\\/\\\\/g;s/"/\\"/g' -- "$@" | awk -F'\t' '
{
    if (NF == 0)
        print "data()."
    else {
        printf("data(\"%s\"", $1)
        for (i = 2; i <= NF; i++)
            printf(", \"%s\"", $i)
        print ")."
    }
}'
```

The ‘`-t`’ option causes the `datalog` program to print answers to queries as tab separated values. The following script removes quoting introduced by `datalog`.

```
#! /bin/sh
sed 's/"\([^\t]*\)"/\1/g' -- "$@" | awk '
BEGIN { print "BEGIN {" }
{ print "print \""$0"\"" }
END { print "}" }' | awk -f -
```

An example using `/etc/passwd` as a database table follows. The example shows a convoluted way of determining my home directory, but never the less demonstrates the integration of `datalog` with other Unix tools. In this example, assume the Datalog import script above has been placed into the file `datalogimport`.

```
sed 's/:/\t/g' /etc/passwd \
| datalogimport \
| sed '
$a\home(A,F):-data(A,B,C,D,E,F,G).\
home(ramsdell,Dir)?' \
| datalog -t - \
| sed 's/\t/:/g'
```

The first `sed` command converts `/etc/passwd` to tab separated values, the second `sed` command adds a rule and a query to the Datalog program, and the last `sed` command converts the output to colon separated values. On my machine, the above script produces `ramsdell:/home/ramsdell`.

Notice that input to `datalog` is obtained from `sed` using the ‘`-`’ option rather than implicitly via standard input. Using the ‘`-`’ option avoids prompt printing and other features of the interactive loop.

Next: Stand-Alone, Previous: Import/Export, Up: Top

7 Extensions

Adding primitive predicates to the Datalog interpreter requires knowledge of its internals. Much of the functionality of the interpreter is implemented using an embedded language called Lua, available at http://www.lua.org. Lua provides the needed infrastructure and a small footprint. As an added bonus, the Lua language is a gem. Knowledge of Lua is a prerequisite for what follows.

The mechanics of adding primitives is straightforward. Put your extensions in a file and load it using the '`-l`' option. The more difficult part is to decide what code should be loaded.

The source file for the Datalog specific Lua code is installed in the Datalog package's data directory. The path name of the source file typically ends with `share/datalog/datalog.lua`. The section with a comment containing the words "PRIMITIVES" is relevant.

There is good support for the case in which a primitive can be defined in terms of a Lua iterator. The file `datalog.lua` defines the `datalog` module, and it contains the function `add_iter_prim`. This function defines a primitive when given a predicate symbol, its arity, and an iterator.

When given a literal, the iterator generates a sequences of answers. Each answer must be an array. Each element in the array must be either a number or a string. The length of the array must be equal to the arity of the predicate.

What follows is a predicate defined by an iterator that produces one answer, and the answer has one element in it—the number three.

```
local function all_threes(literal)
  return function (s, v)
    if v then
      return nil
    else
      return {3}
    end
end
datalog.add_iter_prim("three", 1, all_threes)
```

This primitive for addition shows how to ask if term `t` is a constant via the construct `t:is_const()`.

```
local function add(literal)
   return function(s, v)
             if v then
                return nil
             else
                local x = literal[1]
                local y = literal[2]
                local z = literal[3]
                if y:is_const() and z:is_const() then
                   return {y.id + z.id, y.id, z.id}
                elseif x:is_const() and z:is_const() then
                   return {x.id, x.id - z.id, z.id}
                elseif x:is_const() and y:is_const() then
                   return {x.id, y.id, x.id - y.id}
                else
                   return nil
                end
             end
          end
```

```
end
datalog.add_iter_prim("add", 3, add)
```

The above primitive fails when given a non-numeric argument. A more robust implementation of the predicate ensures that arguments are numbers before performing arithmetic.

```
local function add(literal)
   return function(s, v)
             if v then
                return nil
             else
                local x = literal[1]
                local y = literal[2]
                local z = literal[3]
                if y:is_const() and z:is_const() then
                   local j = tonumber(y.id)
                   local k = tonumber(z.id)
                   if j and k then
                      return {j + k, j, k}
                   else
                      return nil
                   end
                elseif x:is_const() and z:is_const() then
                   local i = tonumber(x.id)
                   local k = tonumber(z.id)
                   if i and k then
                      return {i, i - k, k}
                   else
                      return nil
                   end
                elseif x:is_const() and y:is_const() then
                   local i = tonumber(x.id)
                   local j = tonumber(y.id)
                   if i and j then
                      return {i, j, i - j}
                   else
                      return nil
                   end
                else
                   return nil
                end
             end
          end
end
```

Next: Library, Previous: Extensions, Up: Top

8 Stand-Alone

The Datalog module can be used by the stand-alone Lua interpreter. Load it with

```
require "datalog"
```

In the remainder of this chapter, a use of the Datalog module is demonstrated by translating the following Datalog program into Lua.

```
q(X) :- p(X).
q(a).
p(X) :- q(X).
```

```
q(X)?
```

Abbreviations make the code more readable.

```
mv = datalog.make_var
mc = datalog.make_const
ml = datalog.make_literal
mr = datalog.make_clause
```

A translation of `q(X) :- p(X).`

```
do
   local head = ml("q", {mv("X")})
   local body = {ml("p", {mv("X")})}
   datalog.assert(mr(head, body))
end
```

A translation of `q(a).`

```
do
   local head = ml("q", {mc("a")})
   datalog.assert(mr(head, {}))
end
```

A translation of `p(X) :- q(X).`

```
do
   local head = ml("p", {mv("X")})
   local body = {ml("q", {mv("X")})}
   datalog.assert(mr(head, body))
end
```

Given a literal, the `ask` function uses it as a Datalog query and then prints the answers.

```
function ask(literal)
   local ans = datalog.ask(literal)
   if ans then
      for i = 1,#ans do
         io.write(ans.name)
         if ans.arity > 0 then
            io.write("(")
            io.write(ans[i][1])
            for j = 2,ans.arity do
               io.write(", ")
               io.write(ans[i][j])
            end
            io.write(").\n")
         else
            io.write(".\n")
         end
      end
   end
   return ans
end
```

A translation of the query `q(X)?`

```
ask(ml("q", {mv("X")}))
```

The answer printed.

```
q(a).
```

Next: Copying, Previous: Stand-Alone, Up: Top

9 Library

The Datalog library makes the services provided by a Datalog interpreter available to C programs as a set of C functions and types. The interface is declared in the header file `datalog.h`. The header file contains useful comments.

The state of a Datalog interpreter is maintained by a dynamically allocated structure. A pointer to this structure is passed as the first argument to every C function in the library, except to `dl_open`, which allocates the structure, and returns it as a pointer of type `dl_db_t`.

```
dl_db_t dl_open(void);
```

If an error occurs, `dl_open` returns the null pointer. To release the resources associated with a Datalog interpreter, call `dl_close`.

```
void dl_close(dl_db_t);
```

The package name followed by the version number of the library is returned by `dl_version`.

```
const char* dl_version(void);
```

9.1 Asserting

There are two ways to build literals and clauses, and assert and retract clauses. This section describes the low-level interface. It builds items by pushing each of its component on a stack, and then inserting it into the result. The high-level Datalog program loader is described later.

The remaining int returning functions in this interface return zero on success, unless othewise noted. The maximum size of the stack manipulated by the functions is four, but implementations may provide a larger stack.

9.1.1 Strings

Push a string on the top of the stack. The string may contain zeros.

```
int dl_pushlstring(dl_db_t db, const char *s, size_t n);
```

Push a string on the top of the stack. The string must be zero terminated.

```
int dl_pushstring(dl_db_t db, const char *s);
```

Concatenate two strings. Pops two strings off the top of the stack and then pushes the concatenation of the two strings on the top of the stack.

```
int dl_concat(dl_db_t db);
```

9.1.2 Literals

Start a literal. Pushes an incompleted literal on the top of the stack.

```
int dl_pushliteral(dl_db_t db);
```

Make a string and then use this to add a predicate symbol to a literal. Pops the string from stack. For each literal, this must be done once, and can be done after some number of terms have been added.

```
int dl_addpred(dl_db_t db);
```

Make a string and then use this to add a variable to the list of terms of a literal. Pops the string from stack.

```
int dl_addvar(dl_db_t db);
```

Make a string and then use this to add a constant to the list of terms of a literal. Pops the string from stack.

```
int dl_addconst(dl_db_t db);
```

Finish making a literal after adding all terms and one predicate symbol. Leaves a completed literal on the top of the stack.

```
int dl_makeliteral(dl_db_t db);
```

9.1.3 Clauses

Make a literal and then use this to start a clause. Pops a literal from the stack and leaves a newly created, incomplete clause on top of the stack. The head of the clause is the literal.

```
int dl_pushhead(dl_db_t db);
```

Make a literal and then use this to add it to the clause's body. Pops a literal from the stack, and inserts it into the incomplete clause on the top of the stack.

```
int dl_addliteral(dl_db_t db);
```

Finish the clause. Leaves a completed clause on the top of the stack.

```
int dl_makeclause(dl_db_t db);
```

9.1.4 Actions

Asserts the clause on the top of the stack. The clause is added to the database and popped off the stack. Returns -1 when a clause is not safe.

```
int dl_assert(dl_db_t db);
```

Retracts the clause on the top of the stack. The clause is removed from the database and popped off the stack.

```
int dl_retract(dl_db_t db);
```

9.2 Querying

A literal on the stack can be used to query the database.

```
int dl_ask(dl_db_t db, dl_answers_t *a);
```

This function asks for a list that contains all ground instances of a literal that are a logical consequence of the clauses stored in the database. An instance of a literal is *ground*, if it is the result of substituting some constant for each variable that occurs in the literal. The function always pops the literal from the stack. If some answers are found, they are returned by assigning to the `a` parameter, a pointer to a freshly allocated list of answers of type `dl_answers_t`. The `a` parameter is assigned the null pointer when there are errors or no answers. The `dl_ask` function returns a non-zero value on error.

```
void dl_free(dl_answers_t a);
```

The `dl_free` function frees the space associated with a list of answers.

A description of the functions that access the components of a list of answers completes this section.

```
char *dl_getpred(dl_answers_t a);
```

This function gets the predicate symbol associated with the answers. If the length of the predicate is n, $n + 1$ character locations are returned, and the last location is zero. Other character locations in the predicate may be zero. This function returns the null pointer if given it.

```
size_t dl_getpredlen(dl_answers_t a);
```

This function gets the length of the predicate symbol associated with the answers. This function returns zero if given the null pointer.

```
size_t dl_getpredarity(dl_answers_t a);
```

This function gets the number of terms that makes up each answer. This number is called the predicate's arity. It returns zero if given the null pointer.

```
char *dl_getconst(dl_answers_t a, int i, int j);
```

This function gets the constant associated with term `j` in answer `i`. Zero-based indexing is used throughout. If the length of the constant is n, $n + 1$ character locations are returned, and the last location is zero. Other character locations in the constant may be zero. If there is no specified term or no answers at all, the null pointer is returned.

```
size_t dl_getconstlen(dl_answers_t a, int i, int j);
```

This function gets the length of the constant associated with term `j` in answer `i`. Zero-based indexing is used throughout. If there is no specified term or no answers at all, zero is returned.

9.3 Printing

The functions in this section provide support for printing using the loader's syntax for constants.

```
void dl_putlconst(FILE *out, const char *s, size_t n);
```

Print a constant of a given length. The function assumes the constant has $n + 1$ character locations, and the last location is zero. Other character locations in the constant may be zero.

```
void dl_putconst(FILE *out, const char *s);
```

Print a zero terminated constant.

```
size_t dl_widthoflconst(const char *s, size_t n);
```

Determine the columns needed for a constant with a given length. The function assumes the constant has $n + 1$ character locations, and the last location is zero. Other character locations in the constant may be zero.

```
size_t dl_widthofconst(const char *s);
```

Determine the columns needed for a zero terminated constant.

9.4 Loading

The loader reads a Datalog program. A program is a sequence of zero or more assertions or retractions, followed by an optional query expressed as a literal. While reading, the loader performs the assertions and retractions requested by the program. It finishes by pushing the query literal on the Datalog interpreter's stack. If no query is specified, it pushes a literal known to have no answers.

The loader obtains buffers containing characters by calling a function given to it of type `dl_reader_t`.

```
typedef const char *(*dl_reader_t)(void *data, size_t *size);
```

When called by the loader, the reader is given the data supplied to the loader as its first argument. If there is no more input available, the reader returns a null pointer, otherwise it assigns the size of the buffer to the `size` parameter, and then returns a pointer to the buffer.

The loader reports errors by calling a function given to it of type `dl_loaderror_t`.

```
typedef void (*dl_loaderror_t)(void *data, int lineno,
                               int colno, const char *msg);
```

When a loader error occurs, the loader calls this function with the data supplied to the loader, the location of the error as a line number and column number, and an error message.

```
int dl_load(dl_db_t database, dl_reader_t reader,
            dl_loaderror_t loaderror, void *data);
```

This function loads a Datalog program. It uses a reader to obtain a sequence of buffers that contain the program. The load error reporter is called when the loader detects an error, and then a non-zero value is returned, otherwise, the loader returns zero. The value in the data parameter is passed to both of the supplied functions whenever they are called.

Upon success, the loader leaves a literal on the Datalog interpreter's stack. To ignore this literal, call `dl_pop`.

```
int dl_pop(dl_db_t db);
```

The `dl_loadbuffer` function loads a Datalog program contained in one buffer.

```
int dl_loadbuffer(dl_db_t database, const char *buffer,
                  size_t size, dl_loaderror_t loaderror);
```

Appendix A Copying the Software

The software described by this manual is covered by the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. The terms are contained in the file `COPYING.LIB` in the source distribution of the software, or at http://www.gnu.org/licenses/lgpl.txt.

Index

- `add`, primitive: Extensions
- `add`, primitive with type checking: Extensions
- `add_iter_prim`, `datalog` function: Extensions
- `ask`, Lua function: Stand-Alone
- assertion: Syntax
- atomic formula: Syntax
- body: Syntax
- clause: Syntax
- clause order, example: Tutorial
- comment syntax: Syntax
- constant: Syntax
- continuation line, interpreter: Interpreter
- continuation line, interpreter: Tutorial
- `datalog`, Lua module: Extensions
- `dl_addconst`, library function: Library
- `dl_addliteral`, library function: Library
- `dl_addpred`, library function: Library
- `dl_addvar`, library function: Library
- `dl_ask`, library function: Library
- `dl_assert`, library function: Library
- `dl_close`, library function: Library
- `dl_concat`, library function: Library
- `dl_db_t`, library type: Library
- `dl_free`, library function: Library
- `dl_getconst`, library function: Library
- `dl_getconstlen`, library function: Library
- `dl_getpred`, library function: Library
- `dl_getpredarity`, library function: Library
- `dl_getpredlen`, library function: Library
- `dl_load`, library function: Library
- `dl_loadbuffer`, library function: Library
- `dl_loaderror_t`, library type: Library
- `dl_makeclause`, library function: Library

- `dl_makeliteral`, library function: Library
- `dl_open`, library function: Library
- `dl_pop`, library function: Library
- `dl_pushhead`, library function: Library
- `dl_pushliteral`, library function: Library
- `dl_pushlstring`, library function: Library
- `dl_pushstring`, library function: Library
- `dl_putconst`, library function: Library
- `dl_putlconst`, library function: Library
- `dl_reader_t`, library type: Library
- `dl_retract`, library function: Library
- `dl_version`, library function: Library
- `dl_widthofconst`, library function: Library
- `dl_widthoflconst`, library function: Library
- equals sign, interpreter: Interpreter
- equals sign, primitive predicate: Primitives
- fact: Syntax
- fact, example: Tutorial
- file loading, interpreter: Interpreter
- ground instance of a literal: Library
- head: Syntax
- identifiers, token class: Syntax
- literal: Syntax
- literal, ground instance: Library
- Lua: Extensions
- predicate symbol: Syntax
- program: Syntax
- punctuations, token class: Syntax
- query: Tutorial
- reader: Syntax
- retraction: Syntax
- retraction, example: Tutorial
- rule: Syntax
- rule, example: Tutorial
- safe: Syntax
- strings, token class: Syntax
- tab separated values: Import/Export
- term: Tutorial
- `three`, primitive: Extensions
- token classes: Syntax
- variable: Tutorial
- variables, token class: Syntax
- whitespace: Syntax