The BLOG Language Reference

(BLOG version 0.9)

Lei Li
Computer Science Division
University of California Berkeley
leili@cs.berkeley.edu

Stuart Russell
Computer Science Division
University of California Berkeley
russell@cs.berkeley.edu

July 15, 2014

Abstract

This document provides a reference for the syntax and semantics of BLOG, a probabilistic programming language that represents one possible form of unification for probability and first-order logic. A BLOG program consists of a set of assertions that jointly determine a probability distribution over the space of first-order possible worlds (structures) definable using the vocabulary of the program. In addition to allowing quantification over logical variables denoting objects and relational uncertainty, BLOG has *open-universe* semantics, meaning that it can represent uncertainty about the existence and identity of objects. Hence, it is suitable for applications in which the existence of objects and events must be inferred from raw data. The syntax as described in this document corresponds to BLOG version 0.9. This version implements many improvements in syntax, usability, and efficiency.

Contents

1 Introduction: Open-universe probability models and BLOG

The initial syntax and semantics of BLOG was described in [MMR04], and in [MMR+05]. The authors of these two papers contributed to the first implementation of the BLOG language and inference algorithms. Full details of the formal semantics are given in the Brian Milch's PhD thesis [Mil06].

This document is primarily concerned with the detailed syntax of BLOG, which has changed considerably from the initial version; it is helpful, nonetheless, to understand the key semantic ideas, which are based on first-order logic and have remained unchanged. A BLOG program defines a probability distribution over a space of *possible worlds*. It does so by means of a set of type declarations, number statements, and dependency statements. These statements describe the objects that may exist in each world and introduce function symbols[1] that carry information about the objects. BLOG also supplies several built-in types and built-in functions that operate on those types. The possible worlds are constructed from the function symbols and the objects, and the probability of each world can be calculated from the information supplied by the dependency statements.

1.1 Example: Writing a BLOG program

A simple example serves to illustrate these concepts. Suppose that an urn contains unknown numbers of blue and green balls; balls are drawn one at a time and then replaced after the observed color is noted. Color observation is 80% accurate. First, we define the types:

```
type Ball;
type Draw;
```

We declare two types and use semicolons (`;`) to terminate the type declarations. Then, we talk about the objects that exist for each type. In this case, we know there are two colors and (say) eight draws; by writing

```
distinct Draw Draw[8];
```

we name these objects and assert that the names refer to distinct `Draw`'s. On the other hand, the number of balls is (say) equally likely to be anywhere from 1 to 4; for this we use a number statement and `UniformInt` distribution:

```
#Ball ~ UniformInt(1,4);
```

Having made these assertions, we declare and write dependencies for the function symbols that we need to describe the domain. First, the true color of a ball is equally likely to be blue or non-blue; for this, we use a boolean predicate `isBlue`. It requires the built-in `BooleanDistrib` distribution, which takes a positive real number as the probability of being true:

[1]Function symbols include as special cases predicate symbols (Boolean-valued functions) and constant symbols (zero-ary functions).

```
random Boolean isBlue(Ball b) ~ BooleanDistrib(0.5);
```

The ball drawn on each draw is chosen at random from the balls in the urn:

```
random Ball ballDrawn(Draw d) ~ UniformChoice({b for Ball b});
```

`{b for Ball b}` is the Set comprehension expression. Finally, we say that the observed color is the same as the true color with 80% probability, i.e. the chance of observing blue balls being 80% if the true color of the ball is blue. Here we use the `if-then-else` expression:

```
random Boolean obsBlue(Draw d) ~
  if isBlue(BallDrawn(d)) then
    BooleanDistrib(0.8)
  else
    BooleanDistrib(0.2);
```

Having written dependency statements for all the function symbols, we are done.

1.2 Example, contd.: What does the program mean?

As noted earlier, the BLOG program defines a probability model over worlds. What are the worlds in this case, and what are their probabilities?

Each world has a specific number of balls, $\#Ball = k$, and thus contains balls $Ball_1, \ldots, Ball_k$ in addition to eight draws $Draw_1, \ldots, Draw_8$. (Notice we will use italics for objects in worlds, as distinct from typewriter font for symbols in the BLOG language.) A particular world is fixed by setting the values of every function symbol for every possible tuple of objects of the appropriate types. We call the value of a function symbol applied to a particular tuple of objects a *basic random variable* or BRV. For this domain, the BRVs are:

- $isBlue_{Ball_i}$, either $true$ or $false$, for $i = 1, \ldots, k$;
- $ballDrawn_{Draw_j}$, one of $Ball_1, \ldots, Ball_k$, for $j = 1, \ldots, 8$;
- obs_{Draw_j}, either $true$ or $false$, for $j = 1, \ldots, 8$.

For each k, there are 2^k values for the *isBlue* BRVs, k^8 values for the *ballDrawn* BRVs, and 2^8 values for the *obsBlue*s. The total number of worlds is thus $\sum_{k=1}^{4} 2^k \cdot k^8 \cdot 2^8 = 282,135,040$. The probability of any particular world is just the product of the probabilities of the choices made in "constructing" the world, i.e., choosing values for the number variables and BRVs in topological order. For example, we might choose

- $\#Ball = 1$ with probability 0.25;
- $isBlue_{Ball_1} = true$ with probability 0.5;
- $ballDrawn_{Draw_j} = Ball_1$ with probability 1 (the only choice!) for $j = 1, \ldots, 8$;

- $\ldots ballDrawn_{Draw_8} = \ldots = Ball_1$ with probability 1;
- $obsBlue_{Draw_j} = Blue$ with probability 0.8, for $j = 1, \ldots, 8$.

The probability of this world is $0.25 \times 0.5 \times 1^8 \times 0.8^8 = 0.02097152$.

1.3 Example, contd.: Evidence and queries

Once a model has been defined by a BLOG program, evidence can be supplied by evidence statements. Evidence is supplied by asserting observed values for expressions. arguments (which may themselves be complex terms). In our case, we might assert the following:

```
obs obsBlue(Draw[0]) = true;
obs obsBlue(Draw[1]) = false;
obs obsBlue(Draw[2]) = true;
obs obsBlue(Draw[3]) = false;
obs obsBlue(Draw[4]) = true;
obs obsBlue(Draw[5]) = false;
obs obsBlue(Draw[6]) = true;
obs obsBlue(Draw[7]) = false;
```

Given evidence, a query statement indicates a posterior probability of interest. We can query the value of any expression by adding a query statement to the file. Each such statement will cause the inference engine to be invoked and the answers—probabilities for each possible value of the expression—to be returned to the standard output stream. Alternative output format is available but it is out of the scope. Readers may refer to BLOG user's manual [2].

```
query size({b for Ball b});
```

generates the following answers:

```
Distribution of values for size({b for Ball b : true})
1 0.07696467881871667
2 0.2749155437572819
3 0.3066400745347743
4 0.34147970288922114
```

```
query ballDrawn(Draw[0]) == ballDrawn(Draw[1]);
```

generates answers

```
Distribution of values for (ballDrawn(Draw[0]) = ballDrawn(Draw[1]))
false 0.7070034886998044
true 0.29299651130018706
```

[2] http://bayesianlogic.cs.berkeley.edu/pages/user-manual.html

2 Syntax description conventions

In describing the syntax of BLOG, we use the following conventions:

- Keywords of the language appear in bold green text:

  ```
  type Ball;
  ```

- User-defined names (e.g., `Ball`) and all literal values appear in plain black text.
- Built-in types appear in bold red text:

  ```
  fixed Real x = 4.567;
  ```

- Built-in distributions appear in plain blue text:

  ```
  random Integer N ~ Poisson(12);
  ```

- Function names appear in blue text:

  ```
  random Real height(Person p) ~ Gaussian(6, 1);
  ```

- The dot-dot-dot `...` notation indicates that additional elements can be added, similar to those preceding it.
- Optional elements appear in angle brackets:

  ```
  if boolean-expression then expression <else expression>;
  ```

3 BLOG Program

A BLOG program consists a sequence of statements, each ending with a semicolon(;). The following kinds of statements are recognized:

1. Type declarations,
2. Distinct symbol declarations,
3. Fixed function declarations,
4. Random function declarations,
5. Origin function declarations,
6. Number statements,
7. Evidence statements, and,
8. Query statements.

Type declarations, distinct symbol declarations, number statements and their associated origin function declarations serve to populate worlds with objects. Fixed and random function declarations introduce properties of objects and relationships among objects and define distributions over the worlds that can be constructed using these functions.

A BLOG program can be written in a single file. However, it is also typical to write a large BLOG program in multiple files. By convention, `.blog` suffix is used for BLOG program, and `.dblog` for those with `Timestep`. Within a single file, the ordering of statements does not affect their semantic interpretation. However, while a BLOG program consists of multiple files, they should be organized into an order such that statements in later files only refer to symbols defined within the same file or before. Such order is called *loading order* of a BLOG program.

4 Lexical tokens

BLOG tokens are case sensitive.

4.1 Keywords

Keywords in BLOG have preserved semantics. Currently there are the following keywords:

```
type     distinct     fixed     random     origin
if       then         else      case       in
obs      query        true      false      null
```

4.2 Operators and Punctuation

BLOG supports the following punctuations:

```
;    {    }    (    )    [    ]
:    =    ~    ,    #    .
```

Note `.` is different from `.` in real numbers. It is only used in referring the underlying class name for a distribution. These punctuations are used to separating expressions or statements. `{ }` and `( )` always appear in pairs.

BLOG supports the following operators,

```
+    -    *    /    ^    %
>    <    >=   <=   ==   !=
!    &    |    ->   =>
```

The semantic interpretation of these operators are described in Section B.

4.3 Identifiers

An identifier starts with either 26 alphabetical characters (`a` to `z`, `A` to `Z`) or underscore (_). Then followed by any number of the 26 alphabetical characters, underscores, or digits. An identifier should not collide with keywords. Identifiers are used to refer type names, functions, distribution, distinct symbols, and logical variables. For example, `x`, `_abc1def`, and `abs` are all valid identifiers. The first two can be used as type names, distinct symbol names, or logical variables. The last one, `abs` refers to the library function `abs()` which returns absolute value of the input argument.

4.4 Literals

Boolean literal There are two `Boolean` literals: `true` and `false`.

Integer literals `Integer` literals are defined for all integers, ranging from -2^{31} to $2^{31}-1$, e.g. `-123`, `0`, `7`, `50000`.

Real literals `Real` literals are defined for real numbers. Each real number is a double-precision 64-bit IEEE 754 floating point. The maximal positive `Real` value is $1.7976931348623157 \times 10^{308}$, while the minimal positive `Real` value is $4.9406564584124654 \times 10^{-324}$. In real literals, a zero on either side of the decimal point can be omitted: `123.`, `123.45`, `0.123`, `.123`, `-45.6`. Scientific notation is also supported: `123.45e-2`, `123.45E-2`, `123.45e1`, `123.45e+1`.

Character and String literals Character literals are delimited by single quotes: `'s'`, `'\n'`. String literals are delimited by double quotes: `"hello\n\nworld"`.

Timestep literals `Timestep`'s correspond to nonnegative integers, but are a distinguished type to allow for inference algorithms that are specialized for temporal models. `Timestep` literals are distinguished by an `@` prefix. They are defined for all non-negative timesteps: `@0`, `@1`, `@654`, etc.

Null `null` is a special literal that can be assigned to any user declared type.

4.5 Whitespace and Comments

Tokens may be separated by whitespace characters: ␣, tab-character, and new-line.

Comments are in the following two forms:

- A single-line comment is a sequence of characters starts with `//` and extends to the end of the line, or
- A multi-line comment is a sequence of characters between `/*` and `*/`. Note any character after the first `/*` and before the first `*/` is interpreted as comment. Therefore `/* /* */` is a valid comment, while `/* /* */ */` is not because it has extra `*/`, which BLOG does not know how to interpret.

5 Declaring types

BLOG is a strongly typed language. At present the type system is extremely simple: there is no type hierarchy and each object has exactly one type. Types must be specified for the arguments and value of every function symbol. The types can be built-in-types or user-defined types from a type declaration statement.

5.1 Built-in types

BLOG has the following built-in types:

- `Boolean`
- `Integer`
- `Real`
- `Character`
- `String`
- `Timestep`
- `RealMatrix`

Section 4.4 describes the literals defined for each type.

Section 5.4.2 describes how to construct a `RealMatrix`.

5.2 User-defined types

Additionally, a user may define his or her own types. The syntax for declaring a type in BLOG is:

```
type type_name;
```

Here `type_name` is an identifier. For example, the following line of BLOG declares a Citation type:

```
type Citation;
```

5.3 Distinct symbols

Information about the objects that are elements of a user-defined type is usually provided via a number statement, but there are cases where the objects are known and individually distinguishable, perhaps through observations. Names for such objects can be introduced as follows:

```
distinct type_name identifier1, identifier2, ...;
```

For example, an information extraction system might have 5 citations to work with; these can be introduced by

```
distinct Citation Cite0, Cite1, Cite2, Cite3, Cite4;
```

This statement defines symbols naming 5 objects of type `Citation`. Technically, these are zero-ary function symbols whose value type is `Citation`; they are necessarily distinct (i.e., `Cite1` $\neq$ `Cite2`, etc.) and each refers to the same object in all worlds. In this sense, distinct symbols play the same role as built-in literals.

With a large number of objects, explicit naming of each is inconvenient. In such cases, one can use an implicit construction to define *indexed symbols*:

```
distinct type_name prefix[int];
```

where `prefix` is an identifier and `int` is a nonnegative integer. For example, the following BLOG code declares one hundred symbols for `Citations`:

```
distinct Citation Cite[100];
```

Now the names `Cite[0]`, `Cite[1]`, etc., can be used to refer to these `Citations`. Note that the symbols are indexed starting from 0.

5.4 Array and Matrix

Array types in BLOG include one-dimensional vectors and two-dimensional matrices. Functions with integer arguments can play many of the roles usually played by arrays in ordinary programming languages, but arrays in BLOG are nonetheless useful to represent values of sets of variables whose interaction is best described by linear algebra operations on multiple values simultaneously, rather than one variable at a time. For example, the velocity of an object in 3-D space can be modeled as a vector-valued variable rather than three scalar variables. In such cases, distributions will tend to have vector and matrix parameters—e.g., the mean and covariance parameters of a multivariate Gaussian.

Currently, only Integer matrix and Real matrix are fully supported. Arrays of user declared type are only supported in distinct symbols and symbol reference as described in Section 5.3.

Arrays can be used as return type of functions, but not as arguments of functions. However, some distributions take arrays as arguments.

5.4.1 Constructing List

As we have already seen, we can use square brackets, `[]` to construct a List. Elements in a List are separated by commas (,). Lists can also be nested within other Lists. A shorthand notation is to use semicolons (;) to separated multiple Lists. Thus, the following two Lists are equivalent:

```
[1, 2, 3; 4, 5, 6];
[[1, 2, 3], [4, 5, 6]];
```

Lists are used to assign values to matrix, or to pass parameters to functions.

5.4.2 Matrices

To define a RealMatrix:

```
fixed RealMatrix table = [[1, 2, 3], [4, 5, 6]];
```

The following shorthand syntax is also allowed:

```
fixed RealMatrix table = [1, 2, 3; 4, 5, 6];
```

Linear algebra operations on vectors and matrices are listed in in Appendix B, B.4.

5.4.3 Matrix indexing

The index of matrices starts from 0.

To access first row of the matrix:

```
a[0]
```

To access the first column of the matrix:

```
transpose(a)[0]
```

To access first element in the matrix:

```
a[0][0]
```

Note that `[]` is an operator that applies on a `RealMatrix` and returns a `RealMatrix`. The general form is `expression1[expression2]`, where both `expression` can be any kind of expression as long as the first is of `RealMatrix` type, and the second is of `Integer` type. In order to obtain scalar value of a singleton matrix, use `toReal`. For example:

```
random Integer x ~ UniformInt(0, 2);
random Integer y ~ UniformInt(0, 2);
fixed RealMatrix a = [1, 2, 3; 4, 5, 6; 7, 8, 9];
fixed RealMatrix b = [1, 3, 5; 7, 9, 2; 4, 6, 8];
random Real z ~ toReal((a + b)[x][y]);
query z;
```

6 Dependency statements

A *dependency statement* constrains the value of functions applied to objects. In the case of ordinary programming languages, as well as in the case of built-in functions in BLOG, the value of a function applied to a given tuple of input objects is *fixed*, i.e., it is the same in all worlds. Dependency statements for fixed functions are described in Section 6.1. On the other hand, a *random* function is one about whose values there is uncertainty, so that the values may vary across possible worlds. Dependency statements for random functions are described in Section 6.2.

6.1 Fixed functions

BLOG allows the user to declare that there is *no* uncertainty concerning the value of a given function; that is, the value of the function applied to given arguments is the same in all worlds. Such a fixed function is declared as follows:

```
fixed type_name0 function_name(type_name1 var1, ...) =
  fixed_expression;
```

This statement defines a fixed function with name `function_name` whose arguments are `var1` (of type `type_name1`), etc., and whose return type is `type_name0`. The logical variables `var1`, `var2`, etc., are implicitly universally quantified over all elements of the corresponding types.

The function body is a fixed expression, which may be

- a literal from one of the built-in types or a declared distinct symbol;
- a logical variable from the argument list;
- a built-in operator, fixed function, or externally defined function applied to fixed expressions.
- other expressions containing no random function applications nor Distributions.

The following example defines a function to calculate the sum of squares:

```
random Real sumsquare(Real x, Real y) = x^2 + y^2;
```

When a function has zero arguments, the resulting empty parentheses may be dropped in both the declaration statement and in occurrences within expressions. A fixed function with no arguments is called a *fixed constant*; for example:

```
fixed Real pi = 3.14159;
fixed Real CircleArea(Real r) = pi * r^2;
```

More types of expressions are in Section 8.

6.1.1 List interpretations

The `ListInterp` construct allows defining a relation by exhaustively listing the tuples for which it holds. The first argument of `ListInterp` is the number of arguments to the relation. The remaining arguments list the tuples for which the relation is true. The relation is implicitly false for any tuple not listed. For example:

```
fixed Boolean Teaches(Professor p, Course c)
    = ListInterp(2, Smith, CS106,
                    Jones, CS106,
                    Moriarty, Phil80,
                    Jones, Stat10);
```

With this definition, `Teaches(Jones, CS106)` evaluates to `true`, while `Teaches(Jones, Math101)` evaluates to `false`.

6.1.2 Tabular interpretations

The `TabularInterp` construct allows defining a function by exhaustively listing the mapping it performs from arguments to return value. The first argument to `TabularInterp` is the number of arguments to the function. The remaining arguments list the arguments and return value for which the function is defined. The function will return `null` for any other tuple of arguments. For example:

```
fixed Integer Score(Student s, Course c)
  = TabularInterp(2, Alice, CS106, 80,
                     Alice, Phil80, 55,
                     Bob, CS106, 65,
                     Bob, Stat10, 35);
```

With this definition, `Score(Alice, Phil80)` evaluates to `55`, while `Score(Bob, Phil80)` evalutes to `null`.

6.2 Random functions

To declare a random function, there are two possible forms. The first describes a probabilistic conditional dependency:

```
random type_name0 function_name(type_name1 var1, ...) ~
  distribution_expression;
```

This form defines a random function with name `function_name` whose arguments are `var1` (of type `type_name1`), etc., and whose return type is `type_name0`. The logical variables `var1`, `var2`, etc., are implicitly universally quantified over all elements of the corresponding types and may appear in the distribution expression. The statement asserts that for any possible instantiation of the logical variables with objects, the resulting random variable has a conditional probability distribution described by the corresponding instantiation of the `distribution_expression`. The `distribution_expression` must return a concrete distribution.

For example, the following statement says the height of a tree has a Gaussian distribution whose mean depends linearly on the tree's age and the growth rate of its species:

```
random Real height(Tree x) ~
    Gaussian(Growthrate(Species(x))*Age(x), 4.0);
```

The full syntax of distribution expressions is described in Section 9.

The second form describes a deterministic dependency:

```
random type_name0 function-name(type_name1 var1, ...) = expression;
```

As before, the logical variables may appear in the expression. Such a declaration is distinct from a fixed function declaration because, although the dependency is deterministic, the expression may contain other random function symbols. For example, to express the fact that the observed value `Y(t)` of some temporal process is the underlying state `X(t)` plus an additive Gaussian noise term, one may write

```
random Real Epsilon(Timestep t) ~ Gaussian(0.0,1.0);
random Real Y(Timestep t) = X(t) + Epsilon(t);
```

7 Number statements

As noted in Section 1, BLOG supports open-universe semantics, i.e., different worlds may contain different numbers of objects and hence different numbers of random variables. For a user-defined type, a number statement specifies a probability distribution over the number of objects of that type, possibly depending on other aspects of the world. For example, intuitively the number of fleas in the world depends on the number of dogs.

The simplest form of number statement omits any direct dependency on other objects:

```
#type_name ~ count-distribution-expression;
```

The `count-distribution-expression` should be a distribution over the nonnegative integers (see Section 9.1). For example, the following example declares the number of `Balls` according to a Poisson distribution:

```
#Ball ~ Poisson(10.0);
```

As with random function declarations, a number statement can use a deterministic dependency:

```
#type_name ~ count-expression;
```

The `count-expression` should evaluate to a nonnegative integer.

7.1 Origin functions

The general form for a number statement provides a link between the objects generated by the statement and the objects on whose existence the new objects depend. For example, each flea's existence depends on the existence of its host dog. This link is expressed by an *origin function*. Origin functions are declared as follows:

```
origin type_name0 function-name(type_name1);
```

An origin function has exactly one argument type and one return type. For example, we can declare Host as an origin function:

```
origin Dog Host(Flea);
```

In the number statement, the origin functions are attached to the type as follows:

```
#type_name(origin-function1=var1, ...) ~ distribution-expression;
```

For example, the number statement

```
#Flea(Host=d) ~ Poisson(2*Weight(d));
```

says that the number of fleas whose `Host` is dog `d` has a Poisson distribution whose mean is twice the weight of the dog (bigger dogs have more fleas). In the formal semantics of BLOG, the objects in each possible world contain their origins.

An object may have multiple origins. For example, in a radar system an aircraft may generate a blip at each time step; the blip's origins are the aircraft and the time step.

```
origin Aircraft Source(Blip);
origin Timestep Time(Blip);
#Blip(Source=a,Time=t) ~ Bernoulli(0.8);
```

This says that the number of blips generated by aircraft `a` at time `t` is drawn from a `Bernoulli` distribution that returns a 1 with probability 0.8 and a 0 otherwise.

There can be at most one number statement for a given type *with a given set of origin functions*. More than one number statement can be provided as long as the set of origin functions for each is distinct. For example, suppose some radar blips are false alarms that are not generated by any aircraft:

```
#Blip(Time=t) ~ Poisson(FalseAlarmRate*DeltaT);
```

8 Expressions

An expression can include both fixed and random terms. Expressions are of the following forms:

- A literal of a built-in type, e.g., `Integer`, `Real`, `String`, `Boolean`, or `Timestep`.
- A user-defined symbol, one of the following:
  - A symbol declared in a distinct symbol declaration (including indexed symbols),
  - A fixed constant, i.e., a zero-ary function symbol declared as fixed.
  - A random constant, i.e., a zero-ary function symbol declared as random (often called a random variable).
  - A logical variable within the scope of a function declaration with that variable as an argument or within the scope of a quantifier or set expression with that variable in the prefix.
- A function application expression `function-name(e1, e2, ...)`, where `function-name` is a fixed, random, origin, or user defined function symbol and the arguments `e1`, `e2,`, etc., are expressions of the appropriate types.
- A reference to an array element `A[e1]` or matrix element `A[e1][e2]` where `e1` and `e2` are arbitrary expressions of type `Integer`.References outside the array size will cause a runtime error.
- A numerical expression such as `e1 + e2`, `e1 - e2`, `e1 * e2`, `e1 / e2`, `+ e1`, `- e1`, or `(e1)`, where `e1` and `e2` are also expressions of type `Integer` or `Real`; with the exception of `/`, the type of the expression will be `Integer` if all arguments are of type `Integer`, and will be of type `Real` otherwise. The full list of arithmetic operators is given in Appendix B. Expressions of the form `det(M)`, where `M` is an expression denoting a square matrix, and `size(S)`, where `S` is a set expression, are also numerical expressions.
- A matrix expression such as `e1 + e2`, `e1 - e2`, `e1 * e2`, `e1 / e2`, `+ e1`, `- e1`, `inv(e1) (e1)`, where `e1` and `e2` are [[vectors or matrices]] of the appropriate types and sizes. The full list of matrix operators is given in Appendix B, B.4.

- A Boolean expression in one of the following forms:
  - A logical expression: `e1 & e2`, `e1 | e2`, `! e1`, `(e1)` where `e1` and `e2` are expressions of `Boolean` type.
  - A comparative expression: `e1 > e2`, `e1 >= e2`, `e1 < e2`, `e1 <= e2`, where `e1` and `e2` are expressions of comparable types.
  - An equality expression: `e1 == e2`, `e1 != e2`, where `e1` and `e2` are any expressions;
  - A quantified formula (see Section 8.2).
- A set expression (see Section 8.3).
- A map expression (see Section 8.4).
- An if-then-else expression

  ```
  if condition then expression1 else expression2
  ```

  where `condition` is any Boolean expression and `expression1` and `expression2` are any expressions. The expression has the value of `expression1` when `condition` has the value *true* and `expression2` when `condition` has the value *false*.
- A case expression

  ```
  case expr in map
  ```

  where `expr` is any Boolean expression and `map` is a map expression.

8.1 If-then-else expressions

An if-then-else expression allows one to use different expressions depending on aspects of the conditioning context. The general form is

```
if condition then expression1 else expression2
```

where `condition` is a Boolean expression and `expression1` and `expression2` are expressions. For example, suppose that a `Coin` is exactly `Fair` with probability 0.99, and that a fair coin comes up heads half the time and a biased coin comes up heads roughly 80% of the time:

```
random Boolean Fair(Coin c) ~ BooleanDistrib(0.99);
random Boolean Bias(Coin c) ~
  if Fair(c) then Exactly(0.5)
  else Beta(80,20);
```

The built-in distribution `BooleanDistrib(`p`)` assigns probability p to `true` and $1-p$ to `false`. It is equivalent to `Categorical(true ->` p`, false ->` $1-p$`)`. Note that the `Bernoulli(`p`)` distribution returns 0 or 1, which are not Boolean values.

Note that if-then-else can be nested.

8.2 Quantified formula

BLOG allows quantified formulas, as in typed first-order logic. Each such formula constitutes a Boolean expression, i.e. its result type is `Boolean`.

A universally quantified formula has the form

```
forall type_name var expression
```

where `type_name` is any type and `var` is the name of a logical variable. The `expression` is any Boolean expression, possibly containing `var`. The formula has value *true* in a given world iff `expression` has value *true* in *every* extended interpretation in which the value of `var` is an object in the world of the given type. For example, the formula

```
forall Boolean b (b => b)
```

is true in all worlds, because the Boolean objects in all worlds are just *true* and *false*, and the built-in function => has value *true* for both (*true* => *true*) and (*false* => *false*).

An existentially quantified formula has the form

```
exists type_name var expression
```

and has value *true* in a given world iff `expression` has value *true* in *some* extended interpretation in which the value of `var` is an object in the world of the given type. For example, the formula

```
exists Boolean b (b & b)
```

is true in all worlds, because the Boolean objects in all worlds are just *true* and *false*, and the built-in function `&` has value *true* for `(true &  true)`.

Quantifier expressions may be nested, in which case the `expression` may contain logical variables from any of the enclosing quantifier expressions.

Currently, BLOG's inference algorithms have no mathematical theorem-proving capability; hence, quantification over infinite types leads to expressions that cannot be finitely evaluated.

8.3 Set expressions

A set expression denotes a set of objects satisfying a certain Boolean condition. All such sets exist implicitly in every world, since every world contains objects and every Boolean expression has a truth value for those objects. Normally it is not necessary to construct set objects explicitly, since the same work can be done using predicate expressions, but sets are useful in two contexts:

- When choosing values for functions, it is useful to refer to the set of candidate values, e.g., as an argument to a distribution.
- When observing that certain distinct objects exist and constitute all objects satisfying a certain condition, a set expression can be used as the observation (see Section 10).

As with quantified formulas, set expressions have a logical variable that ranges over a given type. The most general form is

```
{expression for type_name var: condition}
```

where `expression` is any expression, possibly including `var`; `type_name` is any declared or built-in type; `var` is a logical variable; and `condition` is any Boolean expression, possibly containing `var`. The value of the set expression in any world is the set of objects that are possible values of `expression` as `var` ranges over objects of the given type that satisfy `condition`. For example, the set

```
{(x * x) for Integer x: x>0 & x<5}
```

contains the elements 1, 4, 9, 16.

If the `: condition` part is omitted, the condition is assumed to be `true`, i.e., the variable ranges over all values of the given type. Thus, the set

```
{(b => b) for Boolean b}
```

contains just *true*, since `(b => b)` is true for both values of `b`.

8.4 Maps

A map expression, used as one term in a case expression, has the form

```
{key1 -> value1, key2 -> value2, ...}
```

For example, the following map expression could be used as the argument to a ategorical distribution]:

```
 {true -> 0.3, false -> 0.7}
```

Each key must be a literal expression, while the values may be any expressions, as long as all value expressions are of the same type. In particular, the keys in a map can literal arrays; for example, the map

```
{[0, 0] -> 0, [0, 1] -> 0, [1, 0] -> 0, [1, 1] -> 1}
```

describes the AND of two one-bit inputs.

In a context where a distribution is required—for example, in a case distribution expressions—the value expressions may all be distribution expressions. Distribution-valued maps with literal arrays as keys are useful for describing conditional distributions with multiple discrete parents.

8.5 Case expressions

In Section 8.1, the if-then-else expression constructs a simple two-element mixture distribution whose index variable is `Fair(c)`. More complex mixture distributions and other context-specific dependencies can be constructed using nested-if-then-else expressions. Alternatively, one may use a *case expression*, which has the form

```
case expr in map
```

where `expr` is an arbitrary expression and `map` is a mapping from keys to distribution expressions (see Section 8.4). The case expression returns the distribution from the map whose key matches the value of `expr`. For example, the following describes a mixture of three Gaussians:

```
random Integer Z ~ Categorical({0 -> 0.4, 1 -> 0.5, 2 -> 0.1});
random Real X ~
  case Z in {
    0 -> Gaussian(0,1),
    1 -> Gaussian(0,9),
    2 -> Gaussian(5,1)
  };
```

In the terminology of Bayesian networks, `Z` in this example is a parent of `X`. If there are multiple parents, the conditional distribution can be written using a `case` expression with a list of indices. Consider an example due initially to Judea Pearl: the alarm in a house goes off in response to a burglary or an earthquake, but is somewhat unreliable. We might write the following model:

```
random Boolean Burglary(House h) ~ BooleanDistrib(0.003);
random Boolean Earthquake ~ BooleanDistrib(0.002);
random Boolean Alarm(House h) ~
  case [Burglary(h), Earthquake] in {
    [false, false] -> BooleanDistrib(0.01),
    [false, true]  -> BooleanDistrib(0.40),
    [true, false]  -> BooleanDistrib(0.80),
    [true, true ]  -> BooleanDistrib(0.90)
  };
```

9 Distribution expressions

In both random function declarations and number statements, probabilistic dependencies are specified by *distribution expressions*.

A distribution expression resembles a function application, with the name of the distribution followed by its parameter arguments in parentheses. For example, a Bernoulli distribution with parameter 0.8 returns 1 with probability 0.8, 0 otherwise:

```
Bernoulli(0.8)
```

The arguments may be any expressions and may include random functions. For example, we might use a highly concentrated Beta prior for the bias of a normal coin, i.e., the probability that the coin comes up 1 (heads) rather than 0 (tails).

```
random Real Bias(Coin c) ~ Beta(50,50);
random Integer Outcome(Coin c, Toss t) ~ Bernoulli(Bias(c));
```

Currently, many standard distributions are supported by BLOG. A full list of built-in distributions and their parameter specifications appears in Appendix C. Distribution expressions may also use user-defined distributions (Section 12.2).

9.1 Distributions over specific sets of interest

The *support* of a distribution is the set of values for which the distribution assigns non-zero probability. For example, the support of a Gaussian distribution is the entire real line, i.e., the type `Real`. Two classes of distributions have special significance in BLOG:

- *Boolean distributions*: any distribution whose support is the set {`true`,`false`}. Such distributions are suitable for random functions used as conditions in if-then-else expressions. Among the built-in distributions, `BooleanDistrib` has this property.
- *Count distributions*: any distribution whose support is (a subset of) the nonnegative integers $\{0,1,2,\ldots\}$. Such distributions are suitable for the right-hand sides of number statements. Among the built-in distributions, `Bernoulli`, `Binomial`, `Geometric`, `NegativeBinomial`, `Poisson`, and `UniformInt` (with a nonnegative lower bound) have this property.

9.2 Categorical distribution

One particular built-in distribution of interest is the *categorical distribution*, which specifies probabilities for each of a finite, discrete set of elements (all of which should be of a single type). The mapping from elements to probabilities is described by a map expression with elements as keys and probabilities as values. For example, the expression

```
Categorical({true -> 0.3, false -> 0.7});
```

defines a distribution with a 0.3 probability for `true` and a 0.7 probability for `false`.

BLOG will automatically normalize the probability values in the map, so that they sum to 1.0.

9.3 UniformChoice from Set

It is possible to choose from a set using the following distribution:

```
UniformChoice({x for Item x : weight(x) > 10})
```

It is uniformly choosing from a set of items with weight over 10.

10 Observing evidence

Evidence statements may be declared in two ways. The first way is known as value evidence, and is of the form:

```
obs expression1 = expression2;
```

where `expression1` should be a random function application expression without free variables. For example:

```
random Real x ~ Gaussian(1.0);
obs x = 0.5;
```

The second way is known as symbol evidence, and is of the form:

```
obs {x for type x : expression(x)} = { x1, x2, ...}
```

For example, in the aircraft example, blips may be specified in symbol evidence as follows:

```
obs {b for Blip b} = {b1, b2, b3};
```

This defines three blips with names `b1`, `b2`, and `b3`. These names can be used as expressions in queries, which are described next.

11 Issuing queries

To specify a query, use the form:

```
query expression;
```

where `expression` is a function application expression without free variables or formulas. The result will be the posterior distribution given the observations.

```
random Boolean even ~ BooleanDistrib(0.5);
random Boolean head ~
  if even then BooleanDistrib(0.5)
  else BooleanDistrib(0.8);
```

```
obs head = true;
query even;
```

12 Extending BLOG

12.1 User-defined functions

A user-defined function must extend `blog.model.AbstractFunctionInterp` and provide a constructor that takes a single `List` argument. When compiling the class defining the user-defined function, you must add the BLOG JAR file to the class path.

Here is an example user-defined function computing the logarithm in base 10 of a number:

```
package my_package;

import java.util.List;

import blog.model.AbstractFunctionInterp;

public class Log10Interp extends AbstractFunctionInterp {
  public Log10Interp(List args) {
  }

  public Object getValue(List args) {
    final double value = ((Number) args.get(0)).doubleValue();
    return Math.log10(value);
  }
}
```

To use this function from a model, declare it like this:

```
fixed Real log10(Real val) = my_package.Log10Interp();
```

Then `log10` can be used just like any other fixed function, e.g. `query log10(100.0)`.

Make sure `my_package` is on the class path. If you get an error like "No definition found for non-random function log10", you most likely have a class-path problem. If `my_package` is in the current working directory, you need not do anything. Otherwise, if `my_package` lives at `/some/absolute/path/my_package`, launch BLOG with an environment variable: `CLASSPATH=/some/absolute/path blog model.blog`.

For more advanced examples of user-defined functions, see `blog.distrib.ListInterp` and `blog.distrib.TabularInterp`.

12.2 User-defined distributions

Probability distributions are implemented in Java. Distribution classes should implement the interface `blog.distrib.CondProbDistrib`. By default, the BLOG engine will look up distribution classes in the package `blog.distrib`. In addition, it will look up distribution classes under the default empty package.

Below is one example of a uniform distribution on Integers.

```java
package blog.distrib;

import blog.common.Util;

public class UniformInt implements CondProbDistrib {

  /**
   * set parameters for UniformInt
   * distribution
   *
   * @param params
   *          An array of [Integer, Integer]
   *          <ul>
   *          <li>params[0]: <code>lower</code> (Integer)</li>
   *          <li>params[1]: <code>upper</code> (Integer)</li>
   *          </ul>
   *
   * @see blog.distrib.CondProbDistrib#setParams(java.util.List)
   */
  @Override
  public void setParams(Object[] params) {
    if (params.length != 2) {
      throw new IllegalArgumentException("expected two parameters");
    }
    setParams((Integer) params[0], (Integer) params[1]);
  }

  /**
   * For a non-null value of method parameter lower, sets the
   * distribution parameter <code>lower</code> to method
   * parameter lower. Similarly for <code>upper</code>.
   * Then checks to see if assignment of parameters is legal.
   * In other words, an assignment of parameters is legal
   * if <code>lower <= upper</code>.
   *
   * @param lower
   *          parameter <code>lower</code>
   * @param upper
   *          parameter <code>upper</code>
   */
  public void setParams(Integer lower, Integer upper) {
    if (lower != null) {
      this.lower = lower;
      this.hasLower = true;
    }
    if (upper != null) {
      this.upper = upper;
      this.hasUpper = true;
    }
    if (this.hasLower && this.hasUpper) {
      if (this.lower > this.upper) {
        throw new IllegalArgumentException(
            "UniformInt distribution requires that lower <= upper");
      }
      this.prob = 1.0 / (this.upper - this.lower + 1);
      this.logProb = Math.log(this.prob);
    }
  }

  private void checkHasParams() {
    if (!this.hasLower) {
      throw new IllegalArgumentException("parameter lower not provided");
    }
    if (!this.hasUpper) {
      throw new IllegalArgumentException("parameter upper not provided");
    }
  }

  /*
   * (non-Javadoc)
   *
   * @see blog.distrib.CondProbDistrib#getProb(java.lang.Object)
   */
  @Override
  public double getProb(Object value) {
    return getProb(((Integer) value).intValue());
  }

  /**
   * Returns the probability of the UniformInt distribution having
   * outcome <code>value</code>.
   */
  public double getProb(int value) {
    checkHasParams();
    return (value >= lower) && (value <= upper) ? prob : 0;
  }

  /*
   * (non-Javadoc)
   *
   * @see blog.distrib.CondProbDistrib#getLogProb(java.lang.Object)
   */
  @Override
  public double getLogProb(Object value) {
    return getLogProb(((Integer) value).intValue());
  }

  /**
   * Returns the log probability of the UniformInt distribution having outc
   * <code>value</code>.
   */
  public double getLogProb(int value) {
    checkHasParams();
    return ((value >= lower) && (value <= upper)) ? logProb
        : Double.NEGATIVE_INFINITY;
  }

  /*
   * (non-Javadoc)
   *
   * @see blog.distrib.CondProbDistrib#sampleVal()
   */
  @Override
  public Object sampleVal() {
    return sample_value();
  }

  public int sample_value() {
    checkHasParams();
    return lower + Util.randInt(upper - lower + 1);
  }

  @Override
  public String toString() {
    return getClass().getName();
  }

  /** Parameter <code>lower</code>. */
  private int lower;
  /** Flag indicating whether <code>lower</code> has been set. */
  private boolean hasLower;
  /** Parameter <code>upper</code>. */
  private int upper;
  /** Flag indicating whether <code>upper</code> has been set. */
  private boolean hasUpper;
  /**
   * The probability of an outcome between <code>lower</code> and
   * <code>upper</code> inclusive.
   */
  private double prob;
  /**
   * The log probability of an outcome between <code>lower</code> and
   * <code>upper</code> inclusive.
   */
  private double logProb;
}
```

13 Comprehensive examples

13.1 Burglary

The following example was due initially to Judea Pearl: the alarm in a house goes off in response to a burglary or an earthquake, but is somewhat unreliable. We might write the following model:

```
type House;

distinct House Maryhouse, Johnhouse, Cathyhouse, Rogerhouse;
```

```
random Boolean Burglary(House h) ~ BooleanDistrib(0.003);
random Boolean Earthquake ~ BooleanDistrib(0.002);
random Boolean Alarm(House h) ~
  case [Burglary(h), Earthquake] in {
    [false, false] -> BooleanDistrib(0.01),
    [false, true]  -> BooleanDistrib(0.40),
    [true, false]  -> BooleanDistrib(0.80),
    [true, true ]  -> BooleanDistrib(0.90)
  };

obs Alarm(Maryhouse) = true;
obs Alarm(Johnhouse) = true;
obs Alarm(Cathyhouse) = true;
obs Alarm(Rogerhouse) = false;

query Earthquake;
```

BLOG inference engine will produce the following likelihood of earthquake.

```
Number of samples: 10000
Distribution of values for Earthquake
true 0.967140181036552
false 0.03285981896344725
```

13.2 A hidden Markov model for genetic sequences

Example 1 (Hidden Markov models)**.** *The following represents a hidden Markov model for genetic sequences with four states and four output symbols. The state at each time step transitions to another with respect to a conditional distribution specified by a TabularCPD. Each state at each time step emits an observation with respect to another CPD. After making a few observations, we can query the states for each time step.*

```
type State;
distinct State A, C, G, T;

type Output;
distinct Output ResultA, ResultC, ResultG, ResultT;

random State S(Timestep t) ~
  if t == @0 then
    Categorical({A -> 0.3, C -> 0.2, G -> 0.1, T -> 0.4})
  else case S(prev(t)) in {
    A -> Categorical({A -> 0.1, C -> 0.3, G -> 0.3, T -> 0.3}),
    C -> Categorical({A -> 0.3, C -> 0.1, G -> 0.3, T -> 0.3}),
    G -> Categorical({A -> 0.3, C -> 0.3, G -> 0.1, T -> 0.3}),
    T -> Categorical({A -> 0.3, C -> 0.3, G -> 0.3, T -> 0.1})
  };

random Output O(Timestep t) ~
  case S(t) in {
    A -> Categorical({
      ResultA -> 0.85, ResultC -> 0.05,
      ResultG -> 0.05, ResultT -> 0.05}),
    C -> Categorical({
      ResultA -> 0.05, ResultC -> 0.85,
      ResultG -> 0.05, ResultT -> 0.05}),
    G -> Categorical({
      ResultA -> 0.05, ResultC -> 0.05,
      ResultG -> 0.85, ResultT -> 0.05}),
    T -> Categorical({
      ResultA -> 0.05, ResultC -> 0.05,
      ResultG -> 0.05, ResultT -> 0.85})
  };

/* Evidence for the Hidden Markov Model.
 */

obs O(@0) = ResultC;
obs O(@1) = ResultA;
obs O(@2) = ResultA;
obs O(@3) = ResultA;
obs O(@4) = ResultG;

/* Queries for the Hiddem Markov Model, given the evidence.
 * Note that we can query S(5) even though our observations only
 * went up to time 4.
 */

query S(@0);
query S(@1);
query S(@2);
query S(@3);
query S(@4);
query S(@5);
```

BLOG will generate the following results using the particle filtering algorithm.

```
Distribution of values for S(@0)
C 0.8128524436090175
T 0.09473684210526356
A 0.06905545112781902
G 0.023355263157894345
Distribution of values for S(@1)
A 0.8700181159420364
G 0.05365942028985556
T 0.05143115942029019
C 0.024891304347826236
Distribution of values for S(@2)
A 0.7075761624799592
C 0.09974612506680965
G 0.0965058792089793
T 0.0961718332442546
Distribution of values for S(@3)
A 0.7633727477477481
C 0.08085585585585582
G 0.07908220720720713
T 0.07668918918918921
Distribution of values for S(@4)
G 0.8739530096536214
C 0.050912123793299964
T 0.05014906303236865
A 0.02498580352072714
Distribution of values for S(@5)
A 0.2989778534923363
C 0.2949673480976736
T 0.2767248722316938
G 0.12932992617830938
```

A BLOG Grammar Definition

```
program ::= opt_statement_lst ;

opt_statement_lst ::=
    /* EMPTY */
  | statement_lst ;

statement_lst ::=
    statement SEMI statement_lst
  | statement statement_lst
  | statement SEMI
  | statement ;

statement ::=
    declaration_stmt
  | evidence_stmt
  | query_stmt ;

declaration_stmt ::=
    type_decl
  | fixed_func_decl
  | rand_func_decl
  | origin_func_decl
  | number_stmt
  | distinct_decl
  | parameter_decl
  | distribution_decl ;

type_decl ::= TYPE ID ;

type ::=
    refer_name
  | list_type
  | array_type
  | map_type ;

list_type ::= LIST LT refer_name GT ;

array_type_or_sub ::= refer_name LBRACKET ;

array_type ::=
    array_type_or_sub RBRACKET
  | array_type  LBRACKET RBRACKET ;

map_type ::= MAP LT type COMMA type GT ;

opt_parenthesized_type_var_lst ::=
    /* EMPTY */
  | parenthesized_type_var_lst
  | type_var_lst ;

parenthesized_type_var_lst ::=
    LPAREN RPAREN
  | LPAREN type_var_lst RPAREN ;

extra_commas ::=
    COMMA COMMA
  | extra_commas COMMA ;

type_var_lst ::=
    type ID COMMA type_var_lst
  | type ID
  | type ID extra_commas type_var_lst
  | type ID type_var_lst
  | type COMMA type_var_lst ;

fixed_func_decl ::=
    FIXED type_type ID opt_parenthesized_type_var_lst
    EQ expression ;

rand_func_decl ::=
    RANDOM type_type ID opt_parenthesized_type_var_lst
    dependency_statement_body ;

number_stmt ::=
    NUMSIGN refer_name opt_parenthesized_origin_var_list
  dependency_statement_body
  | NUMSIGN opt_parenthesized_origin_var_list
  dependency_statement_body ;

opt_parenthesized_origin_var_list ::=
    /* Empty */
  | LPAREN origin_var_list RPAREN ;
```

```
origin_var_list ::=
    ID EQ ID COMMA origin_var_list
  | ID EQ ID extra_commas origin_var_list
  | ID EQ COMMA origin_var_list
  | ID EQ ID origin_var_list
  | ID EQ ID
  | ID ID ;

origin_func_decl ::=
  ORIGIN type_type ID LPAREN type_type RPAREN
 | ORIGIN type_type LPAREN type_type RPAREN
 | ORIGIN type_type ID LPAREN type_type
 | ORIGIN type_type ID type_type RPAREN ;

distinct_decl ::=
    DISTINCT refer_name id_or_subid_list ;

id_or_subid_list ::=
   id_or_subid
 | id_or_subid COMMA id_or_subid_list
 | id_or_subid id_or_subid_list
 | id_or_subid extra_commas id_or_subid_list ;

id_or_subid ::=
   ID
 | ID LBRACKET INT_LITERAL RBRACKET ;

distribution_decl ::=
    DISTRIBUTION ID EQ refer_name
    LPAREN opt_expression_list RPAREN ;

refer_name ::=
    ID
  | ID DOT refer_name ;

dependency_statement_body ::= DISTRIB expression ;

parameter_decl ::=
   PARAM type ID
 | PARAM type ID COLON expression ;
```

```
expression ::=
    operation_expr
  | literal
  | function_call
  | list_expr
  | map_construct_expression
  | quantified_formula
  | set_expr
  | number_expr
  | if_expr
  | case_expr ;

literal ::=
   STRING_LITERAL
 | CHAR_LITERAL
 | INT_LITERAL
 | DOUBLE_LITERAL
 | BOOLEAN_LITERAL
 | NULL ;

operation_expr ::=
  expression PLUS expression
 | expression MINUS expression
 | expression MULT expression
 | expression DIV expression
 | expression MOD expression
 | expression POWER expression
 | expression LT expression
 | expression GT expression
 | expression LEQ expression
 | expression GEQ expression
 | expression EQEQ expression
 | expression NEQ expression
 | expression AND expression
 | expression OR expression
 | expression DOUBLERIGHTARROW expression
 | expression LBRACKET expression RBRACKET
 | unary_operation_expr ;

unary_operation_expr ::=
   MINUS expression %prec UMINUS
 | NOT expression
 | AT expression
 | LPAREN expression RPAREN ;

quantified_formula ::=
   FORALL type ID expression
 | EXISTS type ID expression ;

function_call ::=
   refer_name LPAREN opt_expression_list RPAREN
 | refer_name ;

if_expr ::=
   IF expression THEN expression ELSE expression
 | IF expression THEN expression ;

case_expr ::= CASE expression IN map_construct_expression ;

opt_expression_list ::=
    expression_list
  | /*EMPTY*/ ;

expression_list ::=
   expression COMMA expression_list
 | expression
 | expression extra_commas expression_list ;

semi_colon_separated_expression_list ::=
   semi_ending_expression_list semi_colon_separated_expression_list
 | semi_ending_expression_list expression_list ;

semi_ending_expression_list ::=
   expression_list SEMI
 | semi_ending_expression_list SEMI ;

map_construct_expression ::= LBRACE expression_pair_list RBRACE ;

expression_pair_list ::=
   expression RIGHTARROW expression COMMA expression_pair_list
 | expression RIGHTARROW expression ;

number_expr ::=
   NUMSIGN set_expr
 | NUMSIGN type ;

list_expr ::=
   LBRACKET opt_expression_list RBRACKET
 | LBRACKET semi_colon_separated_expression_list RBRACKET
 | LBRACKET comprehension_expr RBRACKET ;

set_expr ::=
    explicit_set
  | tuple_set ;

explicit_set ::= LBRACE opt_expression_list RBRACE ;

comprehension_expr ::= expression_list FOR type_var_lst opt_colon_expr ;

opt_colon_expr ::=
    /* EMPTY */
  | COLON expression ;

tuple_set ::= LBRACE comprehension_expr RBRACE ;

evidence_stmt ::= OBS evidence ;

evidence ::= value_evidence ;

value_evidence ::= expression EQ expression ;

query_stmt ::= QUERY expression ;
```

B Built-in operators and functions

B.1 Unary operators

The operators are listed in order of precedence, with highest precedence at the top.

Table 1: Unitary Operators

operator	argument type	result type	meaning
−	Integer	Integer	minus
−	Real	Real	minus
−	RealMatrix	RealMatrix	minus
!	Boolean	Boolean	negation

B.2 Binary operators

The operators are listed in order of precedence, with highest precedence at the top.

Table 2: Binary Operators

left-hand type	operator	right-hand type	result type	meaning
Integer	*	Integer	Integer	multiply
Real	*	Real	Real	multiply
Timestep	*	Integer	Timestep	multiply
Integer	/	Integer	Integer	divide
Real	/	Real	Real	divide
Timestep	/	Integer	Timestep	divide
Integer	%	Integer	Integer	modulus
Real	^	Real	Real	power
Integer	+	Integer	Integer	plus
Real	+	Real	Real	plus
Timestep	+	Integer	Timestep	plus
String	+	String	String	concatenate
Integer	-	Integer	Integer	minus
Real	-	Real	Real	minus
Timestep	-	Integer	Timestep	minus
Integer	<	Integer	Boolean	less than
Real	<	Real	Boolean	less than
Timestep	<	Timestep	Boolean	less than
Integer	>	Integer	Boolean	greater than
Real	>	Real	Boolean	greater than
Timestep	>	Timestep	Boolean	greater than
Integer	<=	Integer	Boolean	less than or equal
Real	<=	Real	Boolean	less than or equal
Timestep	<=	Timestep	Boolean	less than or equal
Integer	>=	Integer	Boolean	greater than or equal
Real	>=	Real	Boolean	greater than or equal
Timestep	>=	Timestep	Boolean	greater than or equal
Any	==	Any	Boolean	equal to
Any	!=	Any	Boolean	not equal to
Boolean	&	Boolean	Boolean	and
Boolean	|	Boolean	Boolean	or
Boolean	=>	Boolean	Boolean	implies

B.3 Constants

Real e

The base of the natural logarithm (approximately equal to 2.71828).

`Real` `pi`

The π constant (approximately equal to 3.14159).

B.4 Matrix operations

For functions, a type signature like `RealMatrix eye(Integer dim)` means that the function `eye` takes a single `Integer` and produces a `RealMatrix`.

`RealMatrix` `inv(`**`RealMatrix`** `x)`

Inverse of the matrix `x`.

`RealMatrix` `transpose(`**`RealMatrix`** `x)`

Transpose of the matrix `x`.

`Real` `det(`**`RealMatrix`** `x)`

Determinant of the matrix `x`.

`Real` `trace(`**`RealMatrix`** `x)`

Trace of the matrix `x`, i.e. the sum of the elements on the diagonal.

`RealMatrix` `diag(`**`RealMatrix`** `vals)`

Diagonal matrix with the given values on the diagonal. `vals` must be a column vector.

`RealMatrix` `repmat(`**`RealMatrix`** `m,` **`Integer`** `rows,` **`Integer`** `cols)`

Return the matrix `m` tiled `rows` times vertically and `cols` times horizontally.

`RealMatrix` `sum(`**`RealMatrix`** `m)`

Column-wise sum of a matrix. For example, `sum([1, 2; 3, 4])` returns `[4, 6]`.

`RealMatrix` `hstack(`**`RealMatrix`** `arg1, ...)`

`RealMatrix` `hstack(`**`Real`** `arg1, ...)`

Stack scalars or matrices horizontally. Accepts an arbitrary number of arguments. For example, `hstack(1, 2, 3)` returns the row vector `[1, 2, 3]`.

`RealMatrix` `vstack(`**`RealMatrix`** `arg1, ...)`

`RealMatrix` `vstack(`**`Real`** `arg1, ...)`

Stack scalars or matrices vertically. Accepts an arbitrary number of arguments. For example, `vstack(1, 2, 3)` returns the column vector `[1; 2; 3]`.

`RealMatrix` `eye(`**`Integer`** `dim)`

Identity matrix of the given size. For example, `eye(5)` returns a 5x5 identity matrix.

`RealMatrix` `zeros(`**`Integer`** `rows,` **`Integer`** `cols)`

Matrix of the given size, filled with zeros. For example, `zeros(3, 4)` returns a 3x4 matrix of zeros.

`RealMatrix ones(Integer rows, Integer cols)`
Matrix of the given size, filled with ones. For example, `ones(3, 4)` returns a 3x4 matrix of ones.

`RealMatrix abs(RealMatrix m)`
Element-wise absolute value.

`RealMatrix exp(RealMatrix m)`
Element-wise exponential of a matrix.

Use square brackets to index into a matrix. If `m` is a two-dimensional matrix, `m[i]` returns the `i`-th row of `m`. If `m` is a row or column vector, `m[i]` returns the `i`-th element of `m`. Note that the resulting value is a `RealMatrix`. To get the i,j-th element, use `toInt(m[i][j])`.

It is a runtime error to perform matrix operations with dimensions that do not match.

B.5 Set operations

`Any min(Set s)`
Minimum of a set.

`Any max(Set s)`
Maximum of a set.

`Real sum(Set s)`
Sum of elements in a set of Real values.

`Integer size(Set s)`
Number of elements in a set.

`Any iota(Set s)`
Extract element from a singleton set. For example, `iota({6})` evaluates to `6`.

Note: `Any` is an internal BLOG type not meant to be used by the end user. It is necessary here because the BLOG type system does not distinguish between, for example, a `Set` of `Reals` and a `Set` of `Integers`. Assigning an `Any` value to a variable of the wrong type is a runtime error.

B.6 Conversions between types

`Integer round(Real val)`
Round a Real to the nearest Integer. For example, `round(1.6)` evaluates to `2`.

`Integer toInt(Real val)`
Convert to Integer, rounding towards zero.

`Integer toInt(Boolean val)`

Convert false to 0, true to 1.

`Integer toInt(RealMatrix val)`

Extract the element from a 1x1 RealMatrix, and convert it to an Integer by rounding towards zero.

`Real toReal(Boolean val)`

Convert false to 0.0, true to 1.0.

`Real toReal(Integer val)`

Convert Integer to Real.

`Real toReal(RealMatrix val)`

Extract the element from a 1x1 RealMatrix, and convert it to a Real.

B.7 Trigonometric functions

`Real sin(Real radians)`

Sine of the given value.

`Real cos(Real radians)`

Cosine of the given value.

`Real tan(Real radians)`

Tangent of the given value.

`Real atan2(Real y, Real x)`

Arctangent. Analogous to the `atan2` function in Java.

B.8 Miscellaneous functions

`Integer abs(Integer x)`

`Real abs(Real x)`

Absolute value.

`Real exp(Integer x)`

`Real exp(Real x)`

Exponential.

`Real log(Integer x)`

`Real log(Real x)`

Natural logarithm.

```
Timestep next(Timestep t)
```

The next timestep.

```
Timestep prev(Timestep t)
```

The previous timestep.

```
Boolean isEmptyString(String s)
```

True iff string is empty.

```
RealMatrix loadRealMatrix(String s)
```

Load a `RealMatrix` from a text file. The space-separated formats produced by numpy and Matlab are supported.

C Built-in distributions

This section lists the distributions included in BLOG standard library.

C.1 Bernoulli

Bernoulli distribution generates the value 1 with probability p and 0 with probability $1-p$.

Parameters:

`Real` $p, 0 \leq p \leq 1$

Support:

`Integer`, $k \in \{0,1\}$

Probability mass function:

$$P(X=k) = \begin{cases} p & k=1 \\ 1-p & k=0 \end{cases}$$

Example: The following code defines a random function symbol `x` distributed according to a Bernoulli distribution.

```
random Integer x ~ Bernoulli(0.5);
```

C.2 Beta

Parameters:

`Real` α, $\alpha > 0$
`Real` β, $\beta > 0$

Support:

`Real`, $0 \leq x \leq 1$

Probability density function:

$$f(x) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{\int_0^1 x^{\alpha-1}(1-x)^{\beta-1} dx}$$

Example: The following code defines a random function symbol `x` distributed according to a Beta distribution with $\alpha = 2$ and $\beta = 3$

```
random Real x ~ Beta(2, 3);
```

C.3 Binomial

Binomial distribution generates the number of successes in a sequence of n independent Bernoulli trials where each trial yields success with probability p.

Parameters:

`Integer` n, $n \geq 0$
`Real` p, $0 \leq p \leq 1$

Support:

`Integer`, $k \in \{0, 1, \ldots, n\}$

Probability mass function:

$$P(X = k) = \binom{n}{k} p^k (1-p)^k, 0 \leq k \leq n$$

Example: The following code defines a random function symbol `x` distributed according to a Binomial distribution.

```
random Integer x ~ Binomial(10, 0.5);
```

C.4 BooleanDistrib

BooleanDistrib distribution generates `true` with probability p and `false` with probability $1 - p$.

Parameters:

`Real` $p, 0 \leq p \leq 1$

Support:

`Boolean`, $k \in \{$`True`,`False`$\}$

Probability mass function:

$$P(X = k) = \begin{cases} p & k = \texttt{true} \\ 1 - p & k = \texttt{false} \end{cases}$$

Example: The following code defines a random function symbol `x` distributed according to a BooleanDistrib distribution.

```
random Boolean x ~ BooleanDistrib(0.5);
```

C.5 Categorical

Refer to Section 9.2.

C.6 Dirichlet

Parameters:

`RealMatrix` (Column Vector) $\bar{\alpha}$, s.t. $\alpha_i > 0, \forall i \in \{1, \ldots, K\}$

Support:

`RealMatrix`, $\bar{x} \in \mathbb{R}^K, s.t. \forall i \in \{1, \ldots, K\} x_i \geq 0, \sum_1^K x_i = 1$

Probability density function:

$$f(\bar{x}) = \frac{1}{Z(\bar{\alpha})} * \prod_{i=1}^{K} x_i^{\alpha_i - 1}$$

$$Z(\bar{\alpha}) = \frac{\prod_{i=1}^{K} \Gamma(\alpha_i)}{\Gamma(\sum_{i=1}^{K} \alpha_i)}$$

Example: The following code defines a random function symbol `x` distributed according to a Dirichlet distribution with $\bar{\alpha} = (1, 3, 5)$

```
fixed RealMatrix vector = [1; 3; 5];
random RealMatrix x ~ Dirichlet(matrix);
```

C.7 Discrete

The Discrete distribution generates an integer from 0 to $K-1$, where the likelihood of each integer is represented by the unnormalized probability vector $\bar{p}$. The Discrete distribution is a simplification of the Multinomial Distribution where $n = 1$.

Parameters:

`RealMatrix` (ColumnVector) $\bar{p}$, $\bar{p} \in \mathbb{R}$, $p_0 \geq 0, \ldots, p_{K-1} \geq 0, s.t. \sum_0^{K-1} p_i > 0$

Support:

`Integer` $k \in \{0, 1, \ldots, K-1\}$

Probability mass function:

$$P(X = k) = \frac{p_k}{\sum_{i=0}^{K-1} p_i}$$

Example: The following code defines a random function symbol `x` distributed according to a Discrete distribution with normalized probabilities $0.5, 0.3, 0.2$.

```
fixed RealMatrix vec = [5; 3; 2];
random Integer x ~ Discrete(vec);
```

C.8 Exponential

The exponential distribution generates the interarrival time between two events in a Poisson process.

Parameters:

`Real` λ, $\lambda > 0$

Support:

`Real`, $x \geq 0$

Probability density function:

$$f(x) = \lambda \exp(-\lambda x), x \geq 0$$

Example: The following code defines a random function symbol `x` distributed according to an Exponential distribution.

```
random Real x ~ Exponential(0.2);
```

C.9 Gamma

Parameters:

`Real` k, $k > 0$
`Real` λ, $\lambda > 0$

Support:

`Real`, $x > 0$

Probability density function:

$$f(x) = \frac{\lambda exp(-\lambda x)(\lambda x)^{k-1}}{\Gamma(k)}$$

$$\Gamma(k) = \int_0^\infty t^{k-1} exp(-t)dt$$

Example: The following code defines a random function symbol `x` distributed according to a Gamma Distribution.

```
random Real x ~ Gamma(2.0, 4.0);
```

C.10 Gaussian

Parameters:

mean, `Real` μ, $\mu \in \mathbb{R}$
variance `Real` σ^2, $\sigma^2 > 0$

Support:

`Real`, $x \in \mathbb{R}$

Probability density function:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} exp(\frac{-(x-\mu)^2}{2\sigma^2})$$

Example: The following code defines a random function symbol `x` distributed according to a Gaussian distribution with mean -1 and variance 2.

```
random Real x ~ Gaussian(-1, 2);
```

C.11 Geometric

Geometric distribution generates the number of failures before the first success in a sequence of independent Bernoulli trials with probability of success p.

Parameters:

`Real` $p, 0 \leq p \leq 1$

Support:

`Integer`, $k \in \{0, 1, 2, \ldots\}$

Probability mass function:

$$P(X = k) = p(1-p)^k$$

Example: The following code defines a random function symbol `x` distributed according to a Geometric distribution where each Bernoulli trial has a probability of success $p = 0.2$

```
random Integer x ~ Geometric(0.2);
```

C.12 Laplace

Parameters:

`Real` $\mu, \mu \in \mathbb{R}$
`Real` $b, b > 0$

Support:

`Real`, $x \in \mathbb{R}$

Probability mass function:

$$f(x) = \frac{1}{2b} exp(\frac{-|x-\mu|}{b})$$

Example: The following code defines a random function symbol `x` distributed according to a Laplace distribution with mean $\mu = 1.0$ and scale $b = 2.0$

```
random Real x ~ Laplace(1.0, 2.0);
```

C.13 Multinomial

The multinomial distribution contains k categories. During a single trial, exactly one of the k categories is selected. The probability of selecting a particular category i is $\frac{p_i}{\sum_{i=1}^{k} p_i}$. The process is repeated independently n times, and the resultant counts of each category are returned.

Parameters:

`Integer` $n, n \in \{0,1,2,\ldots\}$
`RealMatrix` (Column Vector) $\bar{p}$, $p \in \mathbb{R}^k$ $p_1 \geq 0, p_2 \geq 0, \ldots, ..p_k \geq 0, s.t. \sum_{i=1}^{k} p_i > 0$

Support:

`RealMatrix` (Column Vector), $\bar{x} \in \mathbb{Z}^k, s.t. x_i \in \{0,1,2,\ldots,n\} \forall i \in \{0,1,2,\ldots,k\}$

Probability mass function:

$$P(X = x_1, \ldots, x_k) = \frac{n!}{x_1! \times \ldots \times x_k!} p_1^{x_1} \ldots p_k^{x_k}$$

Example: The following code defines a random function symbol `x` distributed according to a Multinomial distribution.

```
fixed RealMatrix m = [1; 1; 2];
random RealMatrix x ~ Multinomial(4, m);
```

C.14 MultivarGaussian

Parameters:

`RealMatrix` (Column Vector)$\bar{\mu}$, $\bar{\mu} \in \mathbb{R}^K$
`RealMatrix` (Column Vector) Σ, symmetric and semi-definite.

Support:

`RealMatrix`, $\bar{x} \in \mathbb{Z}^K$

Probability density function:

$$f(\bar{x}) = \frac{1}{\sqrt{(2\pi)^K |\Sigma|}} exp(-\frac{1}{2}(\bar{x}-\mu)^T \Sigma^{-1} (\bar{x}-\mu))$$

Example: The following code defines a random function symbol `x` distributed according to a Multivariate Gaussian with mean $\bar{\mu} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ and covariance $\Sigma = \begin{pmatrix} 1 & 0 \\ 0 & 3 \end{pmatrix}$

```
fixed RealMatrix mu = [1; 1];
fixed RealMatrix covariance = [1, 0; 0, 3];
random RealMatrix x ~ MultivarGaussian(mu, covariance);
```

C.15 NegativeBinomial

NegativeBinomial distribution generates the number of failures until the r^{th} success in a sequence of independent Bernoulli trials each with probability of success p.

Parameters:

`Integer` r, $r \in \{1,2,\ldots\}$
`Real` p, $0 \leq p \leq 1$

Support:

`Integer`, $k \in \{0,1,2,\ldots\}$

Probability mass function:

$$P(X = k) = \binom{k+r-1}{k} p^r (1-p)^k$$

Example: The following code defines a random function symbol `x` distributed according to a Negative Binomial distribution with probability of success $p = 0.8$ and number of failures $r = 2$.

```
random Integer x ~ NegativeBinomial(2, 0.8);
```

C.16 Poisson

Given a success probability p very close to 0, and n independent Bernoulli trials such that $\lambda = np$, the Poisson distribution generates a good approximation to the number of successful trials.

Parameters:

`Real` λ, $\lambda > 0$

Support:

`Integer`, $k \in \{0,1,2,\ldots\}$

Probability mass function:

$$P(X = k) = exp(-\lambda) * \frac{\lambda^k}{k!}$$

Example: The following code defines a random function symbol `x` distributed according to a Poisson distribution with mean $\lambda = 3.0$

```
random Integer x ~ Poisson(3.0);
```

C.17 UniformChoice

UniformChoice distribution chooses an element uniformly from the set S, whose objects are all of type T.

Parameters:

`Real` S, a set of objects

Support:

`T`, $k \in S$

Probability mass function:

$$P(X = k) = \frac{1}{|S|}$$

Example: The following code defines a random function symbol `x` that selects between the colors Green, Red, and Blue uniformly at random.

```
type Color;
distinct Color Green, Red, Blue;
random Color x ~ UniformChoice({c for Color c});
query x;
```

C.18 UniformInt

UniformInt distribution generates an `Integer` uniformly at random between the between a lower bound a and an upper bound b. a and b are both included in the set of integers which are uniformly sampled from.

Parameters:

`Integer` a, $a \in \mathbb{Z}$
`Integer` a, $b \in \mathbb{Z}$
$a \leq b$

Support:

`Integer`, $k \in \{a, a+1, \ldots, b-1, b\}$

Probability mass function:

$$P(X = k) = \begin{cases} \frac{1}{b-a+1} & a \leq k \leq b \\ 0 & \text{else} \end{cases}$$

]

Example: The following code defines a random function symbol `x` that selects uniformly at random from the set of integers $\{1,2,3\}$

```
random Integer x ~ UniformInt(1, 3);
```

C.19 UniformReal

UniformReal distribution generates a `Real` uniformly at random between the between a lower bound a and an upper bound b. The lower bound a is inclusive but the upper bound is exclusive.

Parameters:

`Real` $a, a \in \mathbb{R}$
`Real` $b, b \in \mathbb{R}$
$a < b$

Support:

`Real`, $x \in [a,b)$

Probability density function:

$$f(x) = \begin{cases} \frac{1}{b-a} & a \leq x < b \\ 0 & \text{else} \end{cases}$$

Example: The following code defines a random function symbol `x` that generates a `Real` uniformly at random between 0.0 and 1.0.

```
random Real x ~ UniformReal(0, 1);
```

C.20 UniformVector

v is an argument list of 1×2 `RealMatrix` row vectors with arbitrary length k. Denote $v_{i,L}$ as the first element in the i^{th} row vector and $v_{i,U}$ as the second element in the i^{th} row vector. UniformVector generates a `RealMatrix` column vector of dimension k where element i is uniformly distributed between $v_{i,L}$ and $v_{i,U}$.

Parameters:

`RealMatrix` $v_1, v_1 \in \mathbb{R}^2, v_{1,L} < v_{1,U}$
...
`RealMatrix` $v_k, v_k \in \mathbb{R}^2, v_{k,L} < v_{k,U}$

Support:

`RealMatrix`(Column Vector), $\bar{x} \in \mathbb{R}^k, \bigcap_{i=1}^{k} v_{i,L} \leq \bar{x}_i \leq v_{i,U}$

Probability density function:

$$f(\bar{x}) = \frac{1}{\prod_{i=1}^{k}(v_{i,U} - v_{i,L})}$$

Example: The following code defines a random function symbol `x` that generates a `RealMatrix` column vector of dimension 2 whose first element is a `Real` that is uniformly distributed between 0 and 0.5 and whose second element is a `Real` that is uniformly distributed between 5 and 10.

```
fixed RealMatrix a = [0, 0.5];
fixed RealMatrix b = [5, 10];
random RealMatrix x ~ UniformVector(a, b);
```

Table 3: Distributions in BLOG

distribution	argument type	value
Bernoulli	Real in [0,1]	binary 0/1
Beta	Positive Real, Positive Real	Real in [0,1]
Binomial	Nonnegative Integer, Real	Nonnegative Integer
BooleanDistrib	Real in [0,1]	Boolean
Categorical	Map	
Dirichlet	Column Vector	Column Vector
Discrete	ColumnVector	Integer
Exponential	Positive Real	Nonnegative Real
Gamma	Positive Real, Positive Real	Positive Real
Gaussian	Real, Real	Real
Geometric	Real in [0,1]	Nonnegative Integer
Laplace	Real, Positive Real	Real
Multinomial	Nonnegative Integer, ColumnVector	ColumnVector
MultivarGaussian	ColumnVector, MatrixReal	ColumnVector
NegativeBinomial	Positive Integer, Real in [0,1]	Integer
Poisson	Real	Nonnegative Integer
UniformChoice	Set	Element in Set
UniformInt	Integer, Integer	Integer
UniformReal	Real, Real	Real
UniformVector	MatrixReal's	RealMatrix

References

[Mil06] Brian Milch. *Probabilistic Models with Unknown Objects*. PhD thesis, Computer Science Division, University of California, Berkeley, 2006. 5

[MMR04] Brian Milch, Bhaskara Marthi, and Stuart Russell. BLOG: Relational modeling with unknown objects. In *ICML 2004 Workshop on Statistical Relational Learning and Its Connections*, pages 67–73, 2004. 5

[MMR$^+$05] Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. BLOG: probabilistic models with unknown objects. In *IJCAI'05: Proceedings of the 19th international joint conference on Artificial intelligence*, pages 1352–1359, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc. 5