

The BLOG Language Reference

Lei Li
Stuart J. Russell

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-51

<http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-51.html>

May 8, 2013



Copyright © 2013, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

The BLOG Language Reference

(BLOG version 0.6)

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

May 8, 2013

Abstract

This document introduces the syntax of BLOG, a probabilistic programming language, for describing random variables and their probabilistic dependencies. BLOG defines probabilistic generative models over first-order structures. For example, all Bayesian networks can be easily described by BLOG. BLOG has the following features: (a) it employs open-universe semantics; (b) it can describe relational uncertainty; (c) it can handle identity uncertainty; and (d) it is empowered by first-order logic. The syntax as described in this document corresponds to BLOG version 0.6. The current version represents a significant redesign and extension to previous versions of BLOG, based on the principles of usability and implementation efficiency.

Contents

1	Basic language concepts	4
2	Declaring types	4
3	Fixed functions	5
3.1	Constants	5
4	Distinct symbols	6
5	Random functions	6
6	Number statements	7
6.1	Origin functions	7
7	Dependency statement	8
8	Observing evidence	8
9	Issuing queries	9
10	Expressions	9
10.1	Fixed expression	10
10.2	Quantified formula	10
10.3	Set expressions	11
11	Array type	11
11.1	Constant array	11
11.2	Constant list literals	12
11.3	Multi-dimensional array	12
12	Map	12
12.1	Multi-dimensional map	13
13	Probability Distribution Library	13
13.1	Elementary Distribution	13
13.2	Categorical distribution as defined by probability mass table	13
13.3	TabularCPD	14
13.3.1	Multiple dependent variables	14
14	Extending BLOG	14
14.1	User defined distribution	14
15	A comprehensive example	16

Appendices	17
A Summary of changes since version 0.3 (2008)	17
B Built-in operators and functions	18
C Built-in distributions	19

1 Basic language concepts

A BLOG program consists a list of statements. Each statement ends with semicolon(;). Statements include

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.

BLOG is strongly typed, therefore every variable and function should explicitly specify a type. Each BLOG program defines a set of random variables and their probabilistic dependencies. A toy example of defining a random variable in BLOG is:

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

which states that a real-valued random variable x is distributed according to the standard normal distribution. To specify a dependent variable $y|x \sim \mathcal{N}(x, 1)$:

```
random Real y ~ Gaussian(x, 1);
```

2 Declaring types

BLOG is a strongly typed language. Each variable should have an associated type. BLOG has the following built-in types: `Boolean`, `Integer`, `Real`, `String`, `Timestep`, and `array` (which is described later). There are *literals* in built-in types, e.g. `1.0`, `''abc''`, `true`, `false`.

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

```
type typename;
```

For example, the following line of BLOG declares a `User` type:

```
type User;
```

3 Fixed functions

A fixed function always has the same semantic interpretation, i.e. its value does not change over possible worlds. To declare a function with fixed interpretation for all satisfying possible worlds:

```
fixed type0 funcname(type1) = expression;
```

This statement defines a function with name `funcname` with one argument, of type `type1`, and with return type `type0`.

The function body is an `expression`, which can be

- a literal of built-in types;
- an expression using built-in operators such as `a+b`, (see later sections);
- a function call to an external interpretation implemented in a Java class, with passing of fixed term as arguments.

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

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

Functions can have zero or multiple arguments as well. Functions without arguments are *constants*.

3.1 Constants

A *constant* is a zero-ary fixed function. Constants are defined in the following form:

```
fixed typename name = expression;
```

where `expression` does not contain any free variables.

For example, constants can be defined as type literals:

```
fixed Real a = 1.0;  
fixed Boolean b = true;
```

Constants may also be defined using built-in functions (note that `a` is already defined above):

```
fixed Real c = 1.0 + a;
```

Such names can be referred to anywhere that fixed zero-ary function can appear. For example:

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

Here `x` follows a Poisson distribution with the parameter set to `a`.

4 Distinct symbols

There is a special type of functions, distinct symbols. Distinct symbols are fixed zero-ary functions without function bodies. Distinct symbols may be defined as follows:

```
distinct typename name1, name2, name3, [...];
```

This statement defines several symbols of type `typename`, `name1`, `name2`, `name3` These symbols will have a fixed interpretation across all satisfying possible worlds. In addition, all these symbols will have *different* interpretations from each other. We can have multiple `distinct` statements in one BLOG model, and all distinct symbols for a given type will have distinct interpretations in possible worlds. In this sense, these symbols are equivalent to “objects” in model structures. Indeed, the `distinct` keyword is usually used to define possible values for a user-defined type.

In addition, we can define arrays of distinct symbols with the following statement:

```
distinct typename name[int];
```

For example: the following BLOG code declares one hundred `Person` symbols.

```
type Person;  
distinct Person P[100];
```

We can use `P[0]`, `P[1]` to refer to these symbols later. Note that the symbols are indexed starting from 0.

Built-in distinct symbols: literals There are predefined distinct symbols for Boolean, Integer, Real, and String, including all integers, all real numbers, and text strings, e.g. 1, 3.14, “hello”. These predefined symbols are also called *literals*. Also, Timestep has its own notion of time tick: `@0`, `@1`, `@2`, etc.

5 Random functions

Random functions may have different interpretations across possible worlds. Random functions are defined in a similar way as fixed functions, but with the `random` keyword. To declare a random function, use the following:

```
random type0 funcname(type1 x) dependency-expression;
```

This statement defines a random function with name `funcname` with one argument, of type `type1`, and with return type `type0`. The notion of a dependency statement will be introduced later, but for now consider this to be a probability distribution.

As a simple example, we can declare the height of a Person with the following BLOG model:

```
type Person;
distinct Person Alice, Bob;
random Real height(Person p) ~ Gaussian(1.70, 0.25);
```

6 Number statements

BLOG supports open world semantics, i.e. the number of objects in possible worlds can be declared in the language itself. Traditional graphical models are constrained to a known, fixed number of objects in all worlds, and thus do not support open world semantics. For a user declared type, number statements specify how many objects there are of each type, and how they are generated:

```
#typename dependency-expression;
```

For example, the following example declares the number of Persons according to a Poisson distribution:

```
type Person;
#Person ~ Poisson(10.0);
```

6.1 Origin functions

Origin functions specify related groups of generated objects in possible worlds. They may be defined as follows:

```
origin type0 funcname(type1);
```

An origin function has exactly one argument type and one return type. Once specified, objects may be generated from an origin function as follows:

```
#typename(origin_function=x, ...) dependency-expression;
```

Below is one example of a number statement with an origin function. It declares that the total number of aircraft follows a Poisson distribution, that each Blip has a source, which is an Aircraft object, and that the number of Blips generated by a given Aircraft follows a Bernoulli distribution.

```
type Aircraft;
type Blip;
#Aircraft ~ Poisson(10.0);
```

```
origin Aircraft Source(Blip);
#Blip(Source=a) ~ Bernoulli(0.5);
```

7 Dependency statement

In both nonrandom function declarations and number statements, the main body consists of dependency statements, which specify a generative process. A dependency statement can be of one of the following forms: simple distribution clause, simple operator clause, or conditional clauses.

A distribution clause consists of the symbol \sim , representing sampling, followed by the distribution name and arguments. Arguments must match the types of the distribution's parameters.

```
~ Distribution(args)
```

For example, below is a dependency statement to sample values from a Poisson distribution.

```
~ Poisson(10.5)
```

The operator clause is specified as

```
= expression
```

where `expression` is an arithmetic or relational operation.

Conditional clauses use logical branches of the form `if then else`,

```
if cond then clause1
else clause2
```

where `cond` is a Boolean valued expression, and `clause1` and `clause2` can be one of the three types of clauses: simple distribution clause, operator clause, or conditional clause.

Example 1 (Uneven coin). *There are two coins, one evenly weighted and one skewed. However, there is no visually distinction between the two. Each time we pick a coin, we flip it and check which face appears.*

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

8 Observing evidence

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

```
obs expression1 = expression2;
```

where `expression1` should be 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 form:

```
obs {type type0 : expression(x)} = { x1, x2, ... }
```

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

```
obs {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.

9 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.

Example 2 (Uneven coin (continued)). *There are two coins, one evenly weighted and one skewed. However, there is no way to visually distinguish the two. Each time we pick a coin, we flip it and check which face lands facing up. What is the probability of the coin being even after we observe a head?*

```
random Boolean even ~ Bernoulli(0.5);
random Boolean head
  if even then ~ Bernoulli(0.5)
  else ~ Bernoulli(0.8);
obs head = true;
query even;
```

10 Expressions

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

- a reference to a literal of a built-in type, e.g. Integer, Real, String, Boolean and Timestep.
- a reference to a distinct symbol;
- a reference to a constant symbol;
- a proper reference to an element in Array, with index of general expression;
- a reference to a fixed, random or origin function of form `randomfun(p1, p2, ...)`, where arguments `p1, p2, ...` are *expressions*;
- an arithmetic operation on numerical type: `e1 + e2, e1 - e2, e1 * e2, e1 / e2, -e1, + e1, - e1, (e1)`, where `e1` and `e2` are also *expressions* of type Integer or Real;
- a logical expression on Boolean type: `e1 & e2, e1 | e2, ! e1, (e1)` where `e1` and `e2` are also *expressions* of type Boolean;
- a relational expression: `e1 > e2, e1 >= e2, e1 < e2, e1 <= e2`, where `e1` and `e2` are *expressions* of comparable types, or;
- an equality expression: `e1 == e2, e1 != e2`, where `e1` and `e2` are themselves *expressions*;
- a quantified expression: `forall t1 e1, exists t2 e2`, where `t1` and `t2` are types in this BLOG model, and `e1` and `e2` are themselves *expressions*;

10.1 Fixed expression

A fixed expression is an expression that does not contain any random function symbols. For example:

```
1.0 + 2.0 * 3.0
a - 2.0
Twice(10.0) * 5.5
```

Where `Twice(·)` is declared as

```
fixed Real Twice(Real x) = x * 2;
```

10.2 Quantified formula

BLOG allows quantified formulas, as in first-order logic. To specify a universal quantified formula,

```
forall typename x expression
```

To specify an existential quantified formula,

exists typename x expression

10.3 Set expressions

A set expression is a special type of expression which can only be used as an argument in a function call, and as observed symbol evidence.

```
{typename x:condition(x)}
```

For example, to specify uniform choice from all balls,

```
type Ball;  
#Ball ~ Poisson(10.0);  
random Ball choice() ~ UniformChoice({Ball b});
```

To specify symbol evidence,

```
obs {Ball b} = {B1, B2, B3};
```

11 Array type

To declare an array type

```
type []
```

Currently, only Integer arrays and Real arrays are fully supported. Arrays of other types are partially supported. Arrays are zero-indexed. Arrays can be used as return type of functions, but not as arguments of functions. However, some distributions take arrays as arguments.

11.1 Constant array

To declare a constant array, use the following form:

```
fixed type [] name = List_literal;
```

For example, to declare an array of natural numbers:

```
fixed Integer [] c = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
```

An element of an array can be referenced as `c[0]`, `c[1]`, `c[2]`, etc.

11.2 Constant list literals

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

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

Constant list literals are used to assign values to arrays, or to pass parameters to functions.

11.3 Multi-dimensional array

Since `Array` is a type, itself could be nested in `Array` declaration, thus yielding multi-dimensional array.

```
fixed type[][] table = [...];
```

For example, a two dimensional array of `int` will be

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

The following syntax in short hand is also correct:

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

For example, a transition matrix in Kalman filters with Newton dynamics can be declared as:

```
fixed Real[][] A = [1, 1, .5; 0, 1, 1; 0, 0, 1];
```

An element in such a dimensional array can be referred as `A[0][0]`.

12 Map

Maps are specified using braces.

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

For example,

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

A Map's key must be some constant, while its value can be evaluated as the value of a *expression*, as long as the type matches.

```
{true -> x^2, false -> y/2}
```

In addition, `type2` in a map can be of the `Distribution` type, which will be introduced in Section 13.

12.1 Multi-dimensional map

The type in a map can be an array, which results in a multi-dimensional map. For example,

```
{[1, 1] -> 0.1, [1, 2] -> 0.3, [2, 1] -> 0.2, [2, 2] -> 0.4};
```

This will be useful in creating `TabularCPD` (see later sections) with multiple parent variables.

13 Probability Distribution Library

13.1 Elementary Distribution

Currently, many distributions are supported by BLOG. A full list of distributions is included in the appendix.

For example, the Gaussian distribution can be referenced via the form:

```
Gaussian(Real, Real)
```

13.2 Categorical distribution as defined by probability mass table

The Categorical distribution is defined as follows:

```
Categorical(Map_expression);
```

The map expression defines the probability mass over possible values of the distribution.

For example:

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

defines a distribution where sampling yields a 0.3 probability of drawing true, and 0.7 probability of drawing false.

The probability should sum up to 1.0; otherwise, it will by default add an entry `null` with probability equal to the residual probability. On the other hand, if the probabilities sum to more than 1.0, the BLOG compiler will produce a runtime error.

13.3 TabularCPD

To declare and construct a tabular conditional probability distribution, use the form:

```
TabularCPD(Map_expression, expression);
```

which evaluates `expression` as a key and generates values from the map. Note the `Map_expression` should be a map from literals or array of literals to a distribution expression.

For example, to draw from Bernoulli distribution according to the value of x ,

```
TabularCPD({true -> ~ Bernoulli(0.3),  
           false -> ~ Bernoulli(0.6)}, x);
```

With this comprehension, we can even declare a conditional mixture of Gaussians easily. For example:

```
random Integer z ~ Categorical({0 -> 0.4, 1 -> 0.6});  
random Real x ~ TabularCPD({0 -> ~Gaussian(5, 1.0),  
                          1 -> ~Gaussian(10, 1.0)}, z);
```

13.3.1 Multiple dependent variables

To declare that a TabularCPD is dependent on several parent variables, use a multi-dimensional map:

```
TabularCPD({[0, 0] -> ~ Gaussian(5, 1.0),  
           [0, 1] -> ~ Gaussian(10, 1.0),  
           [1, 0] -> ~ Gaussian(2, 4.0),  
           [1, 1] -> ~ Gaussian(20, 4.0)}, [x, y])
```

14 Extending BLOG

14.1 User defined distribution

Probability distributions are implemented in Java. Distribution classes should implement the interface `blog.distrib.CondProbDistrib`. Alternatively, distributions can be declared as subclass of `blog.distrib.AbstractCondProbDistrib`. 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.

Note: using a distribution class to implement a deterministic operation is supported but not recommended.

Below is one example of a uniform distribution on Integers.

```
import java.util.*;
import blog.*;
import blog.distrib.*;
import blog.common.Util;
import blog.model.Type;

public class UniformInt extends AbstractCondProbDistrib {
    public UniformInt(List params) {
        try {
            lower = ((Number) params.get(0)).intValue();
            upper = ((Number) params.get(1)).intValue();
            if ((lower > upper) || (params.size() > 2)) {
                throw new IllegalArgumentException();
            }
        } catch (RuntimeException e) {
            throw new IllegalArgumentException(
                "UniformInt CPD expects two integer arguments "
                + "[lower, upper] with lower <= upper. Got: " + params);
        }
    }

    public double getProb(List args, Object value) {
        if (!args.isEmpty()) {
            throw new IllegalArgumentException(
                "UniformInt CPD does not take any arguments.");
        }
        if (!(value instanceof Integer)) {
            throw new IllegalArgumentException(
                "UniformInt CPD defines distribution over objects of class "
                + "Integer, not " + value.getClass() + ".");
        }
        int x = ((Integer) value).intValue();

        if ((x >= lower) && (x <= upper)) {
            return 1.0 / (upper - lower + 1);
        }
        return 0;
    }

    public Object sampleVal(List args, Type childType) {
        if (!args.isEmpty()) {
```

```

    throw new IllegalArgumentException(
        "UniformInt CPD does not take any arguments.");
}

double x = lower + Math.floor(Util.random() * (upper - lower + 1));
return new Integer((int) x);
}

private int lower;
private int upper;
}

```

15 A comprehensive example

Example 3 (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 ~ TabularCPD(
    {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})},
    S(Prev(t)));

random Output O(Timestep t)
  ~ TabularCPD(
    {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})},
    O(Prev(t)));

```

```

                                ResultG -> 0.85, ResultT -> 0.05}},
T -> ~ Categorical({ResultA -> 0.05, ResultC -> 0.05,
                                ResultG -> 0.05, ResultT -> 0.85})),
S(t));

/* 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 Hidden 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);

```

A Summary of changes since version 0.3 (2008)

- Support for the Map data structure;
- Categorical and TabularCPD distribution redefined using Maps;
- Uniform argument representation in CPD;
- Nested if-then-else in dependency statements;
- Direct support of boolean operations;
- Arithmetic operators (+, -, *, /, $\hat{\cdot}$), which can be used in both fixed and random function statements;
- Support for Arrays, references to array elements, and basic matrix operators on Real Arrays (+, -, *);
- All internal operators are type checked;
- Support for additional operators and built-in functions (detailed in appendix);
- Without causing confusion, “nonrandom” is changed to “fixed”, “guaranteed” is changed to “distinct”;
- Complete redesign of parser and semantic translator, aimed at parsing efficiency and modularity.

B Built-in operators and functions

Table 1: Arithmetic operators on Integer and Real

operator	interpretation	example
+	plus	$x + y, 1.0 + 2$
-	minus	$x - y, 1.0 - 2$
*	multiply	$x * y, 1.0 * 2$
/	divide	$x / y, 1.0 / 2$
%	modulus (only applied to Integers)	$x \% y, 1.0 \% 2$
^	power	$x ^ y, 1.0 ^ 2$

Table 2: Logical operators on Boolean

operator	interpretation	example
&	and	$x \& y, (x > 3) \& (x < 5)$
	or	$x y, (x > 5) (x < 3)$
!	not	$! x, !(x > 3)$
=>	imply	$x => y, (x > 5) => (x > 3)$

Table 3: Quantified formula

operator	interpretation	example
forall	\forall	forall Person x height(x) > 1.0
exists	\exists	exists Person x height(x) > 1.0

Table 4: Relational operators on Integer, Real and other comparable types

operator	interpretation	example
>	greater than	$a > b, 2 > 1.0$
>=	greater than or equal to	$a >= b, 2 >= 1.0$
<	less than	$a < b, 1.0 < 2.0$
<=	less than or equal to	$a <= b, 1.0 <= 2.0$

Table 5: Equality operator on all types

operator	interpretation	example
<code>==</code>	equal to	<code>a == b</code>
<code>!=</code>	unequal to	<code>a != b</code>

Table 6: Operators on String

operator	interpretation	example
<code>+</code>	concatenate	<code>"hello " + "world"</code>
<code>==</code>	equal to	<code>"abc" == "def"</code>
<code>!=</code>	unequal to	<code>"abc" != "def"</code>
<code>IsEmptyString()</code>	returns True if the string is empty	<code>IsEmptyString(a)</code>

Table 7: Operators on Timestep

operator	interpretation	example
<code>Pred()</code>	previous Timestep	<code>Pred(@1)</code>

C Built-in distributions

- Bernoulli
- Beta
- Binomial
- Categorical
- Dirichlet
- Exponential
- Gamma
- Gaussian
- Geometric
- MultivarGaussian
- NegativeBinomial
- Poisson
- TabularCPD
- UniformChoice
- UniformInt
- UniformReal

Table 8: Distributions in BLOG

distribution	argument type	value	example
Bernoulli	Real in [0,1]	Boolean	Bernoulli(0.8)
Beta	Real, Real	Real in [0,1]	Beta(1.0, 1.0)
Binomial	Integer, Real	Integer	Binomial(10, 0.5)
Dirichlet	Array of Real	Array of Real	Dirichlet([1, 1, 1])
Exponential	Real	Real	Exponential(2.0)
Gamma	Real, Real	Real	Gamma(3, 2.0)
Gaussian	Real, Real	Real	Gaussian(2.0, 1.0)
Geometric	Real in [0,1]	nonnegative Integer	Geometric(0.5)
MultivarGaussian	Array, 2D Array	Array of Real	MultivarGaussian([0, 0], [1, 0; 0, 1])
NegativeBinomial	Integer, Real in [0,1]	Integer	NegativeBinomial(4, 0.5)
Poisson	Real	nonnegative Integer	Poisson(6.0)
UniformChoice	Set		UniformChoice({Person p})
UniformInt	Integer, Integer	Integer	UniformInt(0, 10)
UniformReal	Real, Real	Real	UniformReal(0, 1.0)