ProbL: Probabilistic Modeling Language
Language Reference Manual

Nir Grinberg, Andrew Wong, Diana Liskovich, Sam Tkach
{ng2470,aw2192,dl2956,st2794}@columbia.edu

June 16, 2015

Contents

1 Introduction

The vast amounts of data readily available for processing (a.k.a the buzz around "Big Data") have created an even greater need for careful modeling and analysis of these large datasets. One of the most common techniques for modeling data is using probabilistic models, in particular, using Directed Graphical Models (DGM's), which describe the dependencies between random variables in a probabilistic model. DGM's are Directed Acyclic Graphs (DAG's) where nodes are random variables and directed edges, connecting node u to v, represent parenting such that: $P(u, v) = P(u)P(v|u)$. Despite the somewhat confining definition, DGM's are suitable for modeling a wide range of important artificial intelligence and machine learning models: from generalized linear regression, factor analysis and PCA, through hidden markov models, time-series models, Kalman filters, Bolzman Machines and hierarchical mixture models.

Still, specifying a probabilistic graphical model in C, Java or even R is not native to the language, which oftentimes implies that people re-implement common statistical inference algorithms (e.g. Gibbs Sampling) for each model. The use of third-party packages like *mcmc* in R is restrictive and does not easily allow for user-defined distributions or functions. Third-party modeling software such as WinBUGS [1] does allow for model specification, but suffers from the same restrictive shortcomings as mentioned before.

2 Types

2.1 Primitive Types

- int - The 32-bit int data type can hold integer values in the range of -2,147,483,648 to 2,147,483,647.
- float - The float data type stores real number values as double-precision floating-point numbers. Its minimum value is no greater than 1e37 and its maximum value is no less than 1e37.
- string - The string data type represents a sequence of characters.
- bool - The bool data type represents a boolean value. It takes one of two values: true or false.
- enum - The enum data type represents a variable that takes one value from a pre-defined set of values.

Here are some examples of declaring and defining a variable of type int:

```
int i = 100;
```

Here are some examples of declaring and defining a variable of type float:

```
float f = .101;
```

Here are some examples of declaring and defining a variable of type string:

```
string s = 'abc';
```

Here are some examples of declaring and defining a variable of type bool:

```
bool b = true;
bool b = false;
```

Here are some examples of declaring and defining an enum:

```
enum DIRECTIONS = {north, south, west east};
enum DIRECTIONS = {north, south, west, east} dir;
enum DIRECTIONS dir = north;
```

2.2 Collections

- array - The array data structure allows you store one or more elements consecutively in memory. Array elements are indexed beginning at position 0. Arrays can be one dimensional or two dimensional.

Here are some examples of declaring and defining an array:

```
int a[3] = [10, 20, 30];
bool b[2][2] = [[true, true],[false, false]];
b = [[true, true],[false, false]];
a[1] = 5;
b[0][0] = false;
```

3 Lexical Conventions

The objective of ProbL is to allow for easy specification of Directed Graphical Models and efficient inference of such models.

3.1 Control Flow

ProbL would support standard looping, conditional statements and block control flows as in *C*. The table below summarizes this.

;	end line
/* */	begin/end comment block
for, while	standard looping constructs
if, else	standard conditional statement
{ }	code block start and end (respectively)

Table 1: Control Flow

3.2 Comments

ProbL has the same multi-line comment methods as C.

3.3 Identifiers

Identifiers are used to create variables and functions. Each identifier will be a combination of digits, letters, and symbols. Letters can be lowercase or uppercase ASCII characters, ProbL will be case sensitive. Digits will be the ASCII characters of 0-9.

3.4 Keywords

ProbL reserves the following words for various purposes. As such, they should not be used as identifiers.

int	data type for an integer
float	data type for a floating point number
enum	data type taking one value from a pre-defined set of values.
string	data type for a sequence of ASCII characters
if, else	conditional statement
bool	data type for a Boolean value
true, false	boolean literals
while	execute and loop the contents until the condition is false
and, or, not	boolean logical operators
fun	defines a standard function taking arguments and returning a value of a given type
print	prints information to standard out
return	returns a value
input, output, model	denote data, parameters and model section of the program

Table 2: Keywords

3.5 Operators

Our most important operator is the $\sim$ operator, which defines the distribution of a random variable. For instance, in the sample program below we define y to be normally distributed using the $\sim$ operator with parameters that correspond to a linear regression model. The operators are:

!=, ==, <, <=, >, >=	Numerical relational
+, -, *, /, %, ˆ	Arithmetic
and, or, not	Logical
$\sim$	Sample a value from the distribution on the RHS
=	Assignment

Table 3: Operators

Mathematical operators act differently in the context of the $\sim$ operator.

3.6 Constants

Constants are Boolean types or sequences of digits. For a Boolean type, a constant is true or false. For a sequence of digits, a minus sign indicates negative value.

3.7 Whitespace

Blank characters represent Whitespace. Whitespace is ignored by the compiler and is used to separate tokens from one another, unless they are inside a string literal of course.

3.8 Scope

Scope of variables is straightforward in ProbL. If a variable is declared outside a block (i.e.- loop, conditional, or function), it is considered as a global variable in that file. Variables declared inside blocks are local to those blocks. In nested blocks, variables can be referenced in the block in which it was defined as well as in any block inside that block.

4 Syntax

Much of the ProbL syntax is similar to that of C. Function definitions, conditional statements, and loops work largely the same way. There are, however, several syntactic qualities unique to ProbL. The following sections will cover the syntax for all aspects of the language.

4.1 Operators and Symbols

Assignment in ProbL works as follows. The = operator is used for all non-probabilistic datatypes, while the $\sim$ operator is used to assign a distribution to a random variable or to link random variables conditionally. Probabilistic assignment is evaluated differently from regular assignment in a few ways.

```
y ~ distribution(/* parameters */); /* sets the distribution of y */
z ~ y;  /* sets z to be conditioned on y */
```

Arithmetic symbols for addition, subtraction, multiplication, division, and exponentiation are binary operators. So are the two assignment operators, and
The following are miscellaneous (non-operator) symbols used for various things in the language:

/* */	begin and end comment block
;	signifies end of a line of code
:	used in function declaration
,	used to separate arguments, elements of collections
{ }	begin or end a block of code, define enums
( )	used for arithmetic grouping, functions, loops, and conditionals
[ ]	used for array element access and instantiation, define collections
' '	used to denote string literals

Table 4: Symbols

4.2 Function Definitions

Functions are defined in the following form. When writing a function, use the "return" keyword to make it return a value.

```
fun function_name (type1 arg1, type2 arg2,...):return_type
{
        /* ... function code here ... */
        return output_value;
}
```

All functions must return a value.

4.3 Loops and Conditionals

ProbL supports for and while loops as well as if-else statements similar to those in C. while and if take a boolean expression, and for uses a more complicated syntax. Here is an example:

```
for(int i ; i<10 ; i = i + 1) /* iterate int i from 0 to 10 */
{                                      /* with step size 1 */
        int j = 0;
        while(j < i)
        {
                j = j + 1;
        }

        if(j == i)
        {
                j = j - 1;
        }
        else
        {
                j = i;
        }
}
```

4.4 Modeling

The core functionality of the language is its modeling capabilities. Each program needs to have a modeling block, which defines the graphical model to apply to data. Using the optional "input" and "output" keywords we let programmers of ProbL specify what data to use and which parameters to estimate. Since reading input and estimating parameters is the core of our language we'll provide standard ways for our users to read in and output a csv file in a simple format. Once compiled, executing a program that uses the optional input and/or output blocks would require an argument passed in that specify where in the file system the input/output should be. For instance, here is the skeleton of a program that makes use of these features on 2 arrays of data, x and y:

```
input {
        /* data */
  bool[] x;
  int[] y;
}
output {
  /* variable declarations of parameters to estimate */
}
model {
        y ~ /* distribution function to use for estimation */
}
```

This would return an array of float containing the estimated parameters in the order in which they were declared. These features can also be used outside of a function. See the sample program for a concrete example.

5 Standard Library Functions

ProbL provides a minimal set of functions that are at the core of any statistical model represented and inferred using ProbL. These include the basic `rand` function to generate random samples from a uniform distribution, basic I/O functions and a `len` function to find the length of an array or a string.

5.1 Randomization

The most essential building block for randomization is the function `rand`:

- `rand():float` - generates a float in the range 0 to 1 uniformly at random
- `srand(int seed):int` - initialize the pseudo-random process using the provided seed.

Using these building blocks, more complex statistical distributions can be defined. A common sampling technique is using the inverse cumulative distribution function (CDF) with a uniform sample from $[0, 1]$, which can be easily obtained using `rand`.

5.2 I/O functions

We will support a few simple I/O functions:

- `print(string str):int` - print the provided string to programs output.
- `open(string filename, string mode):int` - returns a file descriptor for the function opened in the specified mode ('r' for reading text files or 'w' for writing text files).
- `close(int file):int` - closes the file specified using `file`.
- `read(int file):string` - read till the end the previously opened file.
- `write(int file, string content):int` - writes `content` into previously opened file.

5.3 Miscellaneous

- `len(string str):int` - returns the length of the provided string.
- `len(int[] a):int` - returns the length of the first dimension of an array (of integers in this example).

6 Sample Programs

6.1 Linear Regression

Below is an example program that given two vectors (x and y) of observed variables would infer the parameters of the linear regression model (a, b and $sigma$):

```
input {
        float[] x;
        float[] y;
}
output {
        float a;
        float b;
        float sigma;
}
model {
        y ~ norm(a + b * x, sigma);
}
```

6.2 Baysian Model of Coin Tosses

Below is an example program that given a vector (y) of observed outcomes of a coin toss infers the probability the coin is biased (*theta*):

```
input {
        bool[] y;
}
output {
        float theta;
}
fun beta_dist(float a, float b): float {
        /* implementation of the beta distribution, */
        /* returns a number in (0,1) */
}
fun gamma_dist(float alpha, float beta): float {
        /* implementation of the gamma distribution, */
        /* returns a number in (0,inf) */
}
fun binomial(float theta): bool {
        /* implementation of the binomial distribution, */
        /* returns a boolean. */
}
```

```
model {
        a ~ gamma_dist(0.5, 0.8);
    b ~ gamma_dist(1.0, 2.0);
    theta ~ beta_dist(a, b);
        y ~ binomial(theta);
}
```

References

[1] David J Lunn, Andrew Thomas, Nicky Best, and David Spiegelhalter. Winbugs-a bayesian modelling framework: concepts, structure, and extensibility. *Statistics and computing*, 10(4):325–337, 2000.