YAPPL: Yet Another Probabilistic Programming Language

David Hu
Jonathan Huggins
Hans Hyttinen
Harley McGrew

October 31, 2011

Contents

1 Introduction

Probabilistic programming languages have grown increasingly popular in recent years because they allow for the concise definition of complex statistical models. They also provide tools for sampling the (usually Bayesian) models. YAPPL is inspired by the probabilistic programming language Church, an implementation of a pure subset of Scheme (a dialect of Lisp) for generating models using probabilistic functions. Church relies on the standard Lisp syntax, which is unintuitive and difficult to read. The syntax of YAPPL is inspired by OCaml and contains special constructs for the probabilistic elements of the language, which makes it more approachable and human-readable than Church.

2 Language Reference Manual

2.1 Notation

Through the document, *nonterminals* are in brown italics and `terminals` are in light blue monospace. Regular expression-like constructs are used to simplify grammar presentation and are in black. Brackets `[]` are used to indicate optional parts of productions, curly braces `{}` indicate portions of productions that can appear zero or more times, and parentheses `()` indicate grouping, with a vertical bar `|` separating options.

2.2 Lexical conventions

As syntax of YAPPL is inspired by OCaml, many of the lexical conventions follow those of that language. YAPPL has four kinds of tokens: identifiers, keywords, constants, and expression operators. Whitespace such as blanks, tabs, and newlines are ignored and serve to separate tokens. Comments are also ignored.

2.2.1 Comments

A single `#` indicates that all succeeding characters shall be considered part of a comment and ignored until a newline is encountered.

Immediately following a newline, a series of three `###` indicates that all succeeding characters shall be considered part of a comment until another series of three `###` is encountered. Note that newlines are ignored following the `###`, which essentially delimits multi-line comments.

2.2.2 Identifiers

An identifier is a series of alphabetical letters and digits; the first character must be alphabetic.

2.2.3 Keywords

The following identifiers are reserved as keywords/special function and may not be used otherwise:

`fun`	`if`	`match`
`int`	`then`	`with`
`bool`	`else`	`case`
`float`	`in`	`string`
`true`	`false`	`print`
`rand`	`and`	`or`

The keyword `string` is not currently used, but is reserved for future use.

2.2.4 Constants

The reserved boolean constants are `true` and `false`. The empty list constant is `[]` (for all types).

2.2.5 Integer Literals

An integer literal is a sequence of one or more digits, optionally preceded by a minus sign.

Examples of integer literals are `1337` and `-42`.

2.2.6 Floating-point Literals

Floating-point decimals consist in an integer part, a decimal part and an exponent part. The integer part is a sequence of one or more digits, optionally preceded by a minus sign. The decimal part is a decimal point followed by zero, one or more digits. The exponent part is the character `e` or `E` followed by an optional `+` or `-` sign, followed by one or more digits. The decimal part or the exponent part can be omitted, but not both to avoid ambiguity with integer literals.

Examples of floating-point constants are `9000.1`, `2e-5`, and `1.4e9`.

2.3 Types

The following are the basis data types in YAPPL:

`int`	an integer.
`float`	double-precision floating point.
`bool`	a boolean value (either `true` or `false`).
`fun`	a function.

In addition there are derived array types denoted

type `[ ]`

2.3.1 Non-function Type Declarations

All bindings must either be declared within a function declaration or declared when bound. A non-function declaration specifies a type and an identifier in the format *type* `:` *identifier*. Spaces around the colon are optional. Examples of non-function type declarations:

```
int:temp
float[]:data
bool : flag
```

2.3.2 Function Type Declarations

Function declarations consists of `fun` followed by a type declaration for the return type, followed by zero or more type declarations for arguments of function. Optionally, parentheses may surround the type declarations:

`fun` *type-decls* ... *type-decls*

where

fun-type-decls =
fun-type-decls *type-decl*
(*fun-type-decls*)

Examples of function type declarations:

```
fun int:add int:a int:b
fun bool:contains (float:a float[]:list)
```

2.4 Operations

2.4.1 Value binding

Values are bound to names through the construct

value-decl^1 = *expr*^{1} `and` ... `and` *value-decl*^{n} = *expr*^{n} `in` *expr*

which evaluates *expr*^{1} ... *expr*^{n} in an unspecified order and binds the values of those expressions to the names specified in *value-decl*^{1} ... *value-decl*^{n}.

2.4.2 Function binding

The syntax for function binding is identical to that for value binding, except *value-decl* is replaced by *function-decl* and any number of `=` symbols may be replaced by `:=` symbols. The `:=` symbol defines a special memoization function. A memoized function is only evaluated once for a set of input values. Once function is evaluated on those values, it will always return the same value without being reevaluated.

2.4.3 Function evaluation

Functions are evaluated with the following construct:

`~` *identifier* [*expr*^{1} ... *expr*^{n}] [`|` *expr*]

where *expr*^{1} ... *expr*^{n} are optional arguments passed to the function and `|` *expr* specifies an optional condition that the return value of the function must fulfill. The return value of the function may be referenced within the condition by the special variable `$`.

2.4.4 List construction

Lists can be constructed using the syntax

`[` *expr*^{1} `,` . . . `,` *expr*^{n} `]`

Each expression must have the same type.

2.4.5 Patterns

Patterns are templates that allow selecting values of a given shape and binding identifier names to values. Patterns are used in pattern matching.

2.4.5.1 Variable Patterns A variable pattern consists of a value identifier. The pattern will match any value, and the value will be bound to the identifier. The pattern `_` will also match any value, but will not result in a binding. A value identifier can only appear once in a pattern.

2.4.5.2 Constant Patterns A pattern consisting of a constant matches the values equal to that constant.

2.4.5.3 Variant Patterns The pattern *pattern* `::` *pattern* matches non-empty lists whose heads match the first pattern and whose tails match the second pattern. The `::` operator is right associative.

2.5 Expressions

The precedence of expression operators is the same order as they are presented below. Operators in the same grouping (multiplicative, additive, relational etc.) are given the same precedence. Expressions on either side of binary operations must have the same type.

2.5.1 Primary expressions

2.5.1.1 *identifier* An identifier is a primary expression, provided it has been suitably bound. Its type is specified when bound.

2.5.1.2 *constant* A decimal or floating constant is a primary expression. Its type is `int` in the first case, `float` in the last.

2.5.1.3 *identifier*`[`*expr*`]` An identifier followed by an expression in square brackets is a primary expression that yields the value at the `int` index of a list.

2.5.1.4 `(` *expr* `)` A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression.

2.5.2 Multiplicative operators

The multiplicative operators `*` (multiplication), `/` (division), and `%` (modulus) are binary and group left-to-right. The binary % operator results in the remainder from the division of the first expression by the second. Both operands must be type `int` and the result is `int`. The remainder has the same sign as the dividend.

expr `*` *expr*
expr `/` *expr*
expr `%` *expr*

2.5.3 Additive operators

The additive operators `+` (sum) and `-` (difference) are binary and group left-to-right.

expr `+` *expr*
expr `-` *expr*

2.5.4 Relational operators

The relational operators `<` (less than), `>` (greater than), `<=` (less than or equal to) and `>=` (greater than or equal to) all yield false if the specified relation is false, and true if it is true.

expr `<` *expr*
expr `>` *expr*
expr `<=` *expr*
expr `>=` *expr*

2.5.5 Equality operators

The `=` (equal to) and the `!=` (not equal to) operators function as the relational operators above, but have a lower precedence. Therefore, “`a<b = c<d`” is true when `a<b` and `c<d` have the same truth value.

expr `=` *expr*
expr `!=` *expr*

2.5.6 Boolean operators

The boolean operators `and` (conjunction) and `or` (disjunction) are binary and group left-to-right. The boolean operator `!` (negation) is unary and groups right-to-left. The second operand or `or` may not be evaluated if the value of the first is false.

expr `and` *expr*
expr `or` *expr*
`!` *expr*

2.5.7 Concatenation operator

The concatenation operator yields an list that is the concatenation of the left list at the head of the right list. Both sides must be lists of matching type (e.g. `int[]` or `bool[]`).

expr `@` *expr*

2.5.8 List building operator

The building operation

expr^1 `::` *expr*^{2}

yields a list with *expr*^{1} as the head and *expr*^{2} as the tail. Thus, if *expr*^{1} is of type *type*, then *expr*^{2} must have type *type*`[]`.

2.5.9 Conditional expression

The conditional expression evaluates to the second expression if the first is true, otherwise it evaluates to the third expression. The else binds to the closest if.

`if` *expr* `then` *expr* [`else` *expr*]

2.5.10 Pattern match expression

The case expression notation yields the expression paired with the first pattern matching the expression to be matched.

`match` *expr* `with` *pattern*^{1} `->` *expr*^{1} `|` ... `|` *pattern*^{n} `->` *expr*^{n}

2.5.11 Expression sequencing

A pair of expressions separated by a semicolon is evaluated left-to-right and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left to right.

expr `;` *expr*

2.6 Built-in Functions

There are two built-in functions in YAPPL: `rand` and `print`. These are both reserved keywords.

2.6.1 rand

The function `rand` takes no arguments and returns a random or pseudo-random number between 0 and 1.

2.6.2 print

Since YAPPL does no currently support the `string` type or string literals, or allow for side-effects, printing must be achieved explicitly within the language. The `print` function takes a single expression of one of the three basic types as an argument and prints a string representation of that argument to standard output.

2.7 Grammar

A summary of the grammar for YAPPL.

```
expr =
    constant
    identifier
    ( expr )
    expr ; expr
    expr :: expr
    ~ identifier { expr } [ | expr ]
    prefix-op expr
    expr infix-op expr
    [ expr { , expr } ]
    if expr then expr [ else expression ]
    match expression with pattern-matching
    value-binding { and value-binding } in expr
    function-binding { and function-binding } in expr
    rand
    print expr

value-binding =
    value-decl = expr

function-binding =
    function-decl assignment-op expr

type-decl =
    var-decl
    function-decl

function-decl =
    fun fun-type-decls { fun-type-decls }

fun-type-decls =
    fun-type-decls type-decl
    ( fun-type-decls )

var-decl =
    type : identifier

type =
    type [ ]
    base-type
    fun type { type }
```

```
pattern-matching =
    [|]  pattern -> expression  { | pattern -> expression  }

pattern =
    _
    identifier
    constant
    ( pattern )
    pattern :: pattern
```