YAPPL: Yet Another Probabilistic Programming Language

David Hu
Jonathan Huggins
Hans Hyttinen
Harley McGrew

December 22, 2011

Contents

1 Introduction

Probabilistic programming languages (PPLs) have grown increasingly popular in the machine learning research in recent years because they allow for the concise definition of complex statistical models. PPLs provide mechanisms for defining generative Bayesian models and conditionally sampling from them. The two key features the PPLs have to accomplish this are conditional evaluation and memoization. Conditional evaluation allows a function to be evaluated conditional on some predicate of its return value being true (this corresponds to conditional sampling from the model). A memoized function remembers what value it returned for previously evaluated argument values and always returns the same value in the future given those arguments. Memoization is useful because it allows elements from “infinite” random structures (like lists or matrices) to be generated lazily (as needed).

YAPPL (Yet Another Probabilistic Programming Language) is inspired primarily 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, however, which can be 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. In particular, conditional evaluation and memoization are directly built into the YAPPL language. This makes it easier to write and understand the probabilistic models written in YAPPL.

YAPPL is an almost pure functional programming language. The only functions that produce side-effects are those explicitly built into the language: printing, seeding the random number generator, generating random numbers, and functions defined via memoization.

We hope YAPPL will serve to demonstrate how writing a PPL from the the ground up can produce that is accessible and useful to a wide audience.

2 Language Tutorial

YAPPL was designed to make it easy to work with probabilities. The following tutorial will give a quick tour of the basics of the language and eventually build up to define a probability distribution and draw samples from it. At the end we highlight constructs unique to YAPPL.

2.1 Basic values

Let's begin by exploring the simple concepts of the language. There are three basic data types: integers, floating-point numbers, and booleans. Here's how to declare values as each one of these:

Figure 2.1 Value declarations.

```
int:num
float:number_with_a_dot
bool:truthful
```

It would be helpful if we could give these names values and actually use them. To do so, we need to discuss scope. When a value is bound to a name, you must also specify where it is defined. This is done explicitly through the `let...in` construction:

Figure 2.2 Value definition and scoping.

```
let int:num = 5 in
  num + 5
```

The scope of `num` is the expression after `in`.

2.2 Functions

Function declaration and definition looks similar to its value counterparts. Scoping works in a similar way, though a function's scope begins immediately after the `=`, so it may be called recursively. There is no need to explicitly define a function as being recursive.

Figure 2.3 A simple function.

```
fun int:add int:a int:b =
   a + b
in
  ~print_line ~add 1 2
```

This creates a function `add` that returns an integer and takes in two integers as arguments. The body of the function is after the `=` sign. A function returns whatever value its body expression evaluates to.

Functions are called using the function evaluation operator, tilde (~).

2.2.1 Built-in functions

The above code also demonstrates a built-in function of YAPPL, `print_line`, which takes in a single argument to print. The other built-in functions are `rand` and `seed`, which we will use later in this tutorial.

2.3 Expressions

Function and value declarations and definitions are expressions, and so are function evaluations. Sequences of expressions are separated by semicolons. The `if` statement will prove to be useful.

Figure 2.4 An `if` statement.

```
if x < 10 then
  ~print_line x
else
  ~print_line 10
```

For a more formal description, please see the Language Reference Manual.

2.4 A simple distribution

We now know enough to construct a function that takes a sample from a distribution. We will use the geometric distribution because it is a simple enough construct to use in an exemplary program, but still interesting. To remind the reader, a sample from a geometric distribution will return the number of Bernoulli trials needed to obtain a success, given some probability of success p.

Our approach will generate a random number for a number of iterations. Each time a random number is generated, we will increase our integer return value. The function will continue to iterate until the random number is within $[0, p)$. We return the final value. We implement the increment recursively:

Figure 2.5 The basics of sampling from a geometric distribution.

```
fun int:geom float:p int:i =
    if ~rand < p then
      i
    else
      ~geom p (i+1)
```

This is a good start, except we now mention a nuance of YAPPL. For `rand` to return unique numbers on each execution of a program, `seed` must first be called. Furthermore, the interface is not very clean: it would be better to have a function that takes a single parameter that is some probability p. Finally, we would actually like to call this and run the program. The final code for `geom.ypl` is below.

Figure 2.6 The full `geom.ypl`.

```
~seed;
fun int:geom float:p =
  fun int:geom_helper float:orig_p int:i =
    if ~rand < orig_p then
      i
    else
      ~geom_helper orig_p (i+1)
  in
    ~geom_helper p 1
in
  ~print_line ~geom 0.1
```

2.5 Compilation and execution

To compile a YAPPL file, pass it into the YAPPL compiler, which produces an executable. The following is the basic breakdown of compiling and executing a `.ypl` file from the command line:

Figure 2.7 Compilation and execution.

```
$ ./yapplc geom.ypl
$ ./geom
$ 8
```

Note: To compile and run the included `geom.ypl`, you must do `./yapplc tutorials/geom.ypl`.

2.6 Conditionals

With the basics of the language covered, we now give an overview of special constructs built into the language and how to use them.

Below we explain conditionals, which can be specified after any function evaluation. The `given` keyword specifies a condition that must be met. The special symbol `$` references the return value of a function within the context of `given`.

Below is a line of code that returns a random number less than .5 and the last line of `geom.ypl` after modifying it to say we want `geom` to return a value greater than some other value `a`.

Figure 2.8 Two conditionals with `given`.

```
~rand given $ < .5

~geom 0.1 given $ > a
```

2.7 Memoization & returning functions

You can create a memoized function using the `:=` operator instead of the `=` operator during function definition. A memoized function will remember the value it returned for previously evaluated parameter values, and it will always return this same value. Below we give a memoized version of our geometric sampler.

The code below also highlights another useful feature of YAPPL: the ability to pass around and return functions. The notation `(fun int int):geom_list_gen` indicates that the return type of `geom_list_gen` is a function that returns an integer value and takes a single integer value as a parameter.

Figure 2.9 A memoized `geom.ypl`.

```
fun int:geom float:q =
  fun int:geom_helper float:orig_q int:i =
    if ~rand < orig_q then
      i
    else
      ~geom_helper orig_q (i+1)
  in
    ~geom_helper q 1
in
fun (fun int int):geom_list_gen float:p =
    fun int:geom_list int:n :=
     ~geom p
    in
    geom_list
in
~seed;
let (fun int int):g = ~geom_list_gen 0.5 in
~print_line [~g 1, ~g 1,~g 3, ~g 3]
```

Sample output: `[4, 4, 2, 2]`

Note that `geom` is the same as before; however, we define a `geom_list_gen` function that creates and returns a memoized version of `geom`. `g` is an instance of `geom_list_gen`. Although we are taking four different samples, because of memoization, the first two elements of the printed list will always have the same value, as well as the last two elements.

2.8 Pattern matching

YAPPL also supports pattern matching. The following code uses the standard library function `fflip`, which returns either `true` or `false`. However, instead of printing these boolean values, we use pattern matching to print `0` or `1` instead.

Figure 2.10 Basic pattern matching.

```
~seed;
match ~fflip with
    true -> ~print_line 1
  | false -> ~print_line 0
```

Another example of pattern matching is this function which calculates the sum of a list of integers.

Figure 2.11 Pattern matching with lists.

```
fun int:sum int[]:nums =
   match nums with
       n :: rest -> n + ~sum rest
     | []        -> 0
in
~print_line ~sum [1,2,3,4,5]
```

3 Language Reference Manual

3.1 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.

3.2 Definition of a program

A program is a sequence of one or more expressions.

3.3 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. They have no other semantic meaning. Comments are also ignored.

3.3.1 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.

3.3.2 Identifiers

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

3.3.3 Keywords

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

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

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

3.3.4 Constants

The reserved boolean constants are `true` and `false`.

3.3.5 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`.

3.3.6 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`.

3.4 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 `[ ]`

3.4.1 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
```

3.4.2 Function Type Declarations

Function declarations consists of `fun` followed by a type declaration for the function in the format *fun-type* `:` *identifier*. This is followed by zero or more type declarations for arguments of function, which are separated by whitespace. The return types for a function may be a basic type or a function type in the format `( fun` *return-type arg-types* `)`

Below are examples of function type declarations:

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

3.5 Operations

3.5.1 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 the order of declaration and binds the values of those expressions to the names specified in *value-decl*^{1} … *value-decl*^{n}. The scope of a value declaration begins directly to the right of the declaration.

3.5.2 Function binding

The syntax for function binding is identical to that for value binding, except the *value-decl* is replaced by a *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 a function is evaluated on those values, it will always return the same value without being reevaluated.

A literal can only be bound to a function or value once within a program.

3.5.3 Function evaluation

Functions are evaluated with the following construct:

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

where *expr*^{1} … *expr*^{n} are arguments passed to the function. The arguments must match the number and type of the arguments in the declared function being called. The `given` *expr* portion specifies an optional condition that the return value of the function must fulfill. The return value of the function may be referenced within the conditional statement by the special variable `$`. Below is a sample function evaluation that utilizes this construct. The random function `geom`, which generates a draw from a geometric random variable, takes a single argument between 0 and 1.

```
~ geom q given $ > 5
```

Functions are evaluated when they are called.

3.5.3.1 A word of caution about function evaluation As with function declaration, the arguments passed in are separated by whitespace only.

We make a special note here to remind the reader that whitespace is ignored (except for purposes token delineation) and to be mindful when calling functions with multiple arguments. As an example, if `add` expects two integer arguments, when dealing with negative numbers, one must call `~add (-1) (-1)`. Neglecting to do so in this situation would have resulted with YAPPL interpreting the second unary minus as a binary minus operator and reducing `~add -1 -1` to `~add -2`, producing an error.

3.5.4 List construction

Lists can be constructed using the syntax

`[` $expr^1$ `,` ... `,` $expr^n$ `]`

Each expression must have the same type.

3.5.5 Patterns

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

3.5.5.1 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.

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

3.5.5.3 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 for patterns is left-associative.

3.6 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.

3.6.1 Primary expressions

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

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

3.6.1.3 *identifier* `[` *expr* `]` An identifier followed by an expression in square brackets is a primary expression that yields the value at the *expr* index of a list, where the expression evaluate to an integer between 0 and one less than the length of the named list. The behavior is unspecified if the index is outside of that range.

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

3.6.2 Unary operators

The boolean operator `!` (negation) and numerical operator `-` (minus) are unary and group right-to-left.

`-` *expr*
`!` *expr*

3.6.3 Exponential operator

The exponential operator `**` is a binary operator that groups right-to-left.

expr `**` *expr*

Both expressions must be of type `float`. The operator evaluates to a `float`.

3.6.4 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. For multiplication and dilation, both operands must be of type `int` or `float` and the result is of the same type. Modulus takes integers and returns an integer. The remainder has the same sign as the dividend.

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

3.6.5 Additive operators

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

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

3.6.6 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*

3.6.7 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*

3.6.8 Boolean binary operators

The boolean operators `&&` (conjunction) and `||` (disjunction) are binary and group left-to-right, with the latter having higher precedence. The second operand of `or` is not evaluated if the value of the first is false.

expr `&&` *expr*
expr `||` *expr*

3.6.9 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*

3.6.10 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*`[]`. The list building operator `::` is left-associative.

3.6.11 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*

3.6.12 Pattern match expression

The case expression notation yields the expression paired with the first pattern matching the expression to be matched. Each *pattern*^{1}... *pattern*^{n} should be of the same type.

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

3.6.13 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*

3.7 Built-in Functions

There are four built-in functions in YAPPL: `rand`, `seed`, `print`, and `print_line`. These are reserved keywords. They are called by using the tilde operator just like any other function.

3.7.1 Random values

The function `rand` takes no arguments and returns a random or pseudo-random number between 0 and 1. `seed` also takes no arguments. If it is called before `rand`, all subsequent calls of `rand` will be seeded with a value based on the current system time. Otherwise all calls to `rand` use the same default seed.

3.7.2 Printing

Since YAPPL does not currently support the `string` type or string literals, or allow for side-effects (excepting `rand` and `seed`), printing must be achieved explicitly within the language. The `print` function takes a single expression of one of the three basic types (`int`, `bool`, and float) or a list of one of those three types as an argument and prints a string representation of that argument to standard output. `print_line` functions like `print` but appends a newline.

3.8 Grammar

A summary of the grammar for YAPPL.

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

type-decl =
   type : identifier

value-binding =
    value-decl = expr

value-decl =
    type-decl

function-binding =
   function-decl assignment-op expr

function-decl =
   type : identifier { type-decl }

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

pattern-matching =
   [|] pattern -> expr { | pattern -> expr }

pattern =
```

```
_
identifier
constant-of-base-type
( pattern )
pattern :: pattern
```

4 Project Plan

4.1 Process

The initial planning and specification phase took place during in-person weekly meetings. Real-time collaboration was achieved with a whiteboard as well as Google Docs (in-person and remotely). Everyone contributed to the high-level language design.

A Git repository on GitHub was used for version control and code storage. Individually, various text editors were used for development. The compiler was written in OCaml and compiles down to OCaml.

Tests were started early and each team member contributed to the test suite as modules were completed. A test script was used to quickly run all tests during development.

4.2 Style Guide

Two spaces per tab

Value_and_type_names_with_underscores

Modules And Constructors With First Letter Uppercase

TOKENS_ALLCAPS

Pattern matching lines and other logical groupings kept aligned

4.3 Project Timeline

Nov. 24	Finish scanner and overall design
Dec. 13	Complete most of parser, translator
Dec. 20	Complete presentation and report
Dec. 22	Squash all bugs, final report edits

4.4 Roles and Responsibilities

Everyone contributed to writing the LRM and the final document. During actual implementation there was a lot of cross-contribution. Here are the assigned responsibilities:

David	• primary on lexer/parser
Hans	• regression tests • bug squashing
Harley	• translator: arrays, array printing, pattern matching • parser: pattern matching, value binding
Jonathan	• language conception and team leader • primary on translator

4.5 Tools and Languages

OCaml was used for the compiler and bash scripts for testing and building. Various text editors were used for writing code. Git was used for version control. Email and Google Docs were used for remote discussions and real-time collaboration.

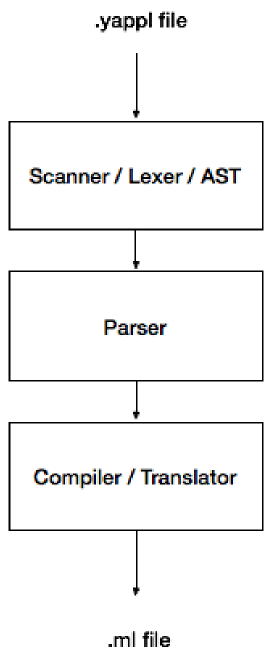
4.6 Project Log

Sep. 13	First meeting, discussing ideas
Sep. 20	Second meeting, deciding on PPL idea
Sep. 27	Decided language features and scope
Oct. 18	Grammar decided, skeleton of scanner implemented
Oct. 26	Skeleton of parser working
Nov. 25	Initial tests written in advance
Dec. 14	Complete most of parser, translator
Dec. 16	Translation working
Dec. 21	Completed tutorial and presentation
Dec. 22	Wrote last examples, finished final report edits

5 Architectural Design

5.1 Block Diagram

YAPPL (it is appropriate to shout when pronouncing the all-caps name) is implemented using a standard model for compilers. The input is tokenized, parsed, and then translated into OCaml. The components are implemented in ocamllex, ocamlyacc and OCaml. The generated .ml can then be compiled into executable code. `yapplc` automates this process.



5.2 Components

5.2.1 Symbol Table

The translator builds an initial table for storing user specified identifiers and their type. The table can also point to a parent table, allowing for scoping. This scoping occurs, for example, in a declaration of a `(` *pattern* `::` *pattern* `)` match. To detect a problematic pattern such as `(a::a)`, the first `a` is added to a blank symbol subtable. When converting the second `a` into output, the compiler will notice that an `a` has already been created in the subtable and raise an error.

5.2.2 expr_to_string

This is the main function for the evaluation of the program expression and returns both a string representation of the parse tree it is given, and its type. It is called recursively to evaluate the program parse tree. As part of the output generation, types are checked for validity. When the function converts identifiers to strings, it will first check if that identifier is built-in, such as `print`, before performing a lookup with the symbol table.

5.2.3 Memoization

Memoization is implemented using a hash table that is populated when a memoized function is called with a new parameter. In the generated code, all user-defined identifiers (not just those in the memoized function) will be prepended with `yappl_`, to prevent namespace collisions with OCaml identifiers, such as hash tables, used by the YAPPL compiler.

5.2.4 Compiling to executable

The `ocamlc` utility links the compiled generated code with built-in OCaml functionality such as `rand`. It also prepends `stdlib.ypl`, if it exists, to function as a standard library.

6 Test Plan

6.1 Example Programs

The Dirichlet process (DP) is a stochastic process. Draws from a DP are discrete distributions with some other distribution as their "mean" distribution. In the example below, we create a draw from a DP with a geometric distribution as its mean. We are then able to take samples from this distribution. We use the "stick breaking" representation of the Dirichlet process. Memoization is used to keep track of the weights on the atoms in the draw from the DP. A distribution drawn from a DP has a "rich-get-richer" property. So when the code below is run, you will see that numbers that appear early in the sequence tend to reappear.

tutorials/dpmem.ypl

```
###
An implementation of the Dirichlet process (DP) using memoization. For an explanation of DPs, see

Teh et. al. Hierarchical Dirichlet processes. Journal of the Am. Stat. Assoc., 101(476):1566--1581, 2006.
###

# placeholder for a function that would generate a draw from the beta distribution (so this is a draw from the Beta(1,1) distribution, no matter what a and b are
fun float:beta float:a float:b = ~rand in

# get a stick, breaking more if necessary
fun int:pickastick (fun float int):sticks int:j =
    if ~rand < ~sticks j then j else ~pickastick sticks j+1
in

# generic Dirichlet process code
fun (fun int):DP float:alpha (fun int):proc =
    fun float:sticks int:x := ~beta 1.0 alpha in
    fun int:atoms  int:x := ~proc in
    fun int:f = ~atoms ~pickastick sticks 1 in
    f # return f
in

fun (fun (fun int) float):DPmem float:alpha (fun int float):proc =
    fun (fun int):dps float:arg :=
      fun int:apply = ~proc arg in
      ~DP alpha apply
```

```
    in
    fun (fun int):dp float:arg = ~dps arg in
    dp
in

# this function will create Dirichlet process draws with geometric base
    distribution
let (fun (fun int) float):geom_dp = ~DPmem 1.0 geom in

# this is a DP draw with geometric base distribution with q = .2
let (fun int):mydraw = ~geom_dp .2 in

# use a tail-recursive loop to generate some samples from the Dirichlet Process
fun bool:loop int:i =
    ~print ~mydraw;
    if i > 0 then ~loop i - 1 else true
in
~seed;
~loop 30; ~print_line ~mydraw
```

tutorials/dpmem.ml

```
open Builtin
open Hashtbl

let _ =
let rec yappl_flip yappl_bias unit =
 ( Builtin.rand  () ) <= ( yappl_bias )
in
let rec yappl_fflip  unit =
 yappl_flip ( 0.5 ) ()
in
let rec yappl_geom yappl_q unit =
 let rec yappl_geom_helper yappl_orig_q yappl_i unit =
 if ( ( Builtin.rand  () ) < ( yappl_orig_q ) ) then ( yappl_i ) else (
     yappl_geom_helper ( yappl_orig_q ) ( ( yappl_i ) + ( 1 ) ) () )
in
yappl_geom_helper ( yappl_q ) ( 1 ) ()
in
let rec yappl_beta yappl_a yappl_b unit =
 Builtin.rand  ()
in
let rec yappl_pickastick yappl_sticks yappl_j unit =
```

```
 if ( ( Builtin.rand  () ) < ( yappl_sticks ( yappl_j ) () ) ) then ( yappl_j
    ) else ( yappl_pickastick ( yappl_sticks ) ( ( yappl_j ) + ( 1 ) ) () )
in
let rec yappl_DP yappl_alpha yappl_proc unit =
 let rec table_yappl_sticks tabl yappl_x unit =
 let rec no_mem_yappl_sticks yappl_x unit =
 yappl_beta ( 1. ) ( yappl_alpha ) ()
in
try Hashtbl.find tabl ( yappl_x ) with Not_found ->
let result = no_mem_yappl_sticks yappl_x () in
Hashtbl.add tabl yappl_x result; result
in
let hash_table_for_yappl_sticks = Hashtbl.create 50 in
let yappl_sticks = table_yappl_sticks hash_table_for_yappl_sticks in
let rec table_yappl_atoms tabl yappl_x unit =
 let rec no_mem_yappl_atoms yappl_x unit =
 yappl_proc  ()
in
try Hashtbl.find tabl ( yappl_x ) with Not_found ->
let result = no_mem_yappl_atoms yappl_x () in
Hashtbl.add tabl yappl_x result; result
in
let hash_table_for_yappl_atoms = Hashtbl.create 50 in
let yappl_atoms = table_yappl_atoms hash_table_for_yappl_atoms in
let rec yappl_f  unit =
 yappl_atoms ( yappl_pickastick ( yappl_sticks ) ( 1 ) () ) ()
in
yappl_f
in
let rec yappl_DPmem yappl_alpha yappl_proc unit =
 let rec table_yappl_dps tabl yappl_arg unit =
 let rec no_mem_yappl_dps yappl_arg unit =
 let rec yappl_apply  unit =
 yappl_proc ( yappl_arg ) ()
in
yappl_DP ( yappl_alpha ) ( yappl_apply ) ()
in
try Hashtbl.find tabl ( yappl_arg ) with Not_found ->
let result = no_mem_yappl_dps yappl_arg () in
Hashtbl.add tabl yappl_arg result; result
in
let hash_table_for_yappl_dps = Hashtbl.create 50 in
let yappl_dps = table_yappl_dps hash_table_for_yappl_dps in
let rec yappl_dp yappl_arg unit =
 yappl_dps ( yappl_arg ) ()
in
yappl_dp
```

```
in

 let yappl_geom_dp = yappl_DPmem ( 1. ) ( yappl_geom ) () in
 (
 let yappl_mydraw = yappl_geom_dp ( 0.2 ) () in
 ( let rec yappl_loop yappl_i unit =
 (ignore ( print_int ( yappl_mydraw  () ); print_char ' '; true ));
if ( ( yappl_i ) > ( 0 ) ) then ( yappl_loop ( ( yappl_i ) - ( 1 ) ) () ) else
   ( true )
in
(ignore ( Builtin.seed  () ));
(ignore ( yappl_loop ( 30 ) () ));
ignore ( print_int ( yappl_mydraw  () ); print_char ' '; true );
 print_newline (); true ) )
```

tutorials/fib.ypl

```
### Calculate Fibonacci numbers  ###

# recursive implementation of calculating the nth number
# in the Fibonacci sequence.
fun int:fib int:n =
    if n <= 1 then n
    else (~fib n-1) + (~fib n-2)
in

# memoized version of the above.
fun int:fib_memo int:n :=
    if n <= 1 then n
    else (~fib_memo n-1) + (~fib_memo n-2)
in

# inefficient, since each recursive call recomputes all the previous numbers.
~print_line ~fib 5;
~print_line ~fib 20;
~print_line ~fib 35;
~print_line ~fib 40;

# the memoized version is much faster, as lookups are performed instead.
~print_line ~fib_memo 5;
~print_line ~fib_memo 20;
~print_line ~fib_memo 35;
~print_line ~fib_memo 40
```

tutorials/fib.ml

```
open Builtin
open Hashtbl

let _ =
let rec yappl_flip yappl_bias unit =
 ( Builtin.rand  () ) <= ( yappl_bias )
in
let rec yappl_fflip  unit =
 yappl_flip ( 0.5 ) ()
in
let rec yappl_geom yappl_q unit =
 let rec yappl_geom_helper yappl_orig_q yappl_i unit =
 if ( ( Builtin.rand  () ) < ( yappl_orig_q ) ) then ( yappl_i ) else (
     yappl_geom_helper ( yappl_orig_q ) ( ( yappl_i ) + ( 1 ) ) () )
in
yappl_geom_helper ( yappl_q ) ( 1 ) ()
in
let rec yappl_fib yappl_n unit =
 if ( ( yappl_n ) <= ( 1 ) ) then ( yappl_n ) else ( ( yappl_fib ( ( yappl_n )
     - ( 1 ) ) () ) + ( yappl_fib ( ( yappl_n ) - ( 2 ) ) () ) )
in
let rec table_yappl_fib_memo tabl yappl_n unit =
 let rec no_mem_yappl_fib_memo yappl_n unit =
 if ( ( yappl_n ) <= ( 1 ) ) then ( yappl_n ) else ( ( table_yappl_fib_memo
     tabl ( ( yappl_n ) - ( 1 ) ) () ) + ( table_yappl_fib_memo tabl ( (
     yappl_n ) - ( 2 ) ) () ) )
in
try Hashtbl.find tabl ( yappl_n ) with Not_found ->
let result = no_mem_yappl_fib_memo yappl_n () in
Hashtbl.add tabl yappl_n result; result
in
let hash_table_for_yappl_fib_memo = Hashtbl.create 50 in
let yappl_fib_memo = table_yappl_fib_memo hash_table_for_yappl_fib_memo in
(ignore ( ignore ( print_int ( yappl_fib ( 5 ) () ); print_char ' '; true );
 print_newline (); true ));
(ignore ( ignore ( print_int ( yappl_fib ( 20 ) () ); print_char ' '; true );
 print_newline (); true ));
(ignore ( ignore ( print_int ( yappl_fib ( 35 ) () ); print_char ' '; true );
 print_newline (); true ));
(ignore ( ignore ( print_int ( yappl_fib ( 40 ) () ); print_char ' '; true );
 print_newline (); true ));
(ignore ( ignore ( print_int ( yappl_fib_memo ( 5 ) () ); print_char ' '; true
    );
 print_newline (); true ));
```

```
(ignore ( ignore ( print_int ( yappl_fib_memo ( 20 ) () ); print_char ' '; true );
 print_newline (); true ));
(ignore ( ignore ( print_int ( yappl_fib_memo ( 35 ) () ); print_char ' '; true );
 print_newline (); true ));
ignore ( print_int ( yappl_fib_memo ( 40 ) () ); print_char ' '; true );
 print_newline (); true
```

6.2 Test Suites

Our testing strategy was "bottom-up": we began by writing tests for the simplest functionality possible, namely printing an integer literal. We increased the complexity of our tests as more of the grammar was implemented. We tried to ensure that each language feature–such as data types, functions, pattern matching, and memoization–was covered by at least one test. Testing the builtin `rand`, however, can only be done manually, as random numbers have no guarantees. As bugs were found, regression tests for those bugs were written so they would not crop up again. Once most of the language was indeed implemented, we wrote slightly more involved tests, such as creating toy functions.

The testing script was based on the MICROC test suite's `testall.sh` script provided to the class. Our tests work rather simply; the printed output is compared to the expected output file's contents.

Additional "testing" was performed via writing and running several examples programs for the tutorial (Section 2) and test plan (Section 6.1). We also added several tests for errors; we expect each error example to fail to compile, and check if an error was produced. This is important so that we know if the grammar is too inclusive.

7 Lessons Learned

David:

- It is important to be familiar with the language we're working with, especially the debugging tools. The `OCAMLRUNPARAM` environment variable and running `ocamlyacc` with `-v` were invaluable tools at our disposal.
- We should code to the LRM. This means having the LRM finalized before coding begins. During a lot of situations when we were implementing the language, we realized something in the LRM was incomplete, incompatible, or just inaccurate. It is important to realize these issues early on. In general, having a complete and well thought-out spec is one of the more important parts of the software development cycle.

Jonathan:

- Everyone needs to be able to work independently and be productive. But joint coding sessions have a lot of benefits too. Not only does it force everyone to set aside some time to work on the project for a few hours, but everyone can provide each other with immediate feedback and help with debugging, figuring out how to implement something etc. These factors tend to make such group sessions very productive.
- Unit tests are amazing. There's a good reason software engineers use them. Particularly in the middle of the development process, writing even 2 or 3 non-trivial tests is pretty much guaranteed to find at least one bug.

Hans:

- Possibly the only reason we have a complete and working compiler–turned in on time–is that we recognized how lofty our initial dreams were and chose to limit our project's scope. Through careful planning we kept our grammar small and left frivolous niceties, such as strings, unimplemented unless we somehow finished early. Our small language is now complete and well-tested.
- Code ownership can lead to "blocking"–preventing someone else from completing their part, because they depend on your finishing. Using Git and communicating early and often with your team will make collaboration much easier–across modules and files. Pair programming is sometimes the perfect productivity boost you need to eliminate blocking.

Harley:

- The LRM is always authoritative and reliable— *except when it's not.* Then, it needs to be corrected. Testing is largely validating that your language follows the LRM, but part of testing is also verifying the completeness and consistency of the LRM.
- Don't forget to test for parsing and compilation errors you **want** to happen. If you only test the output of well-formed programs, you won't find out that your compiler happily compiles invalid code into a meaningless blob of executable gubbins.

7.1 Advice for the Future

- **On getting a head start**
  It is, of course, easy for everyone to say "start early." It's similarly easy to think "ok, we're going to start early." It's definitely harder to actually get started as early as you intend. Some ideas:
  - Create some deadlines early on for your team for meeting and creating some initial tests, and skeletal implementations of your compiler.
  - Send email to the team about the project when the discussion goes quiet. It can be about anything: questions you have, or when the next meeting is occurring. Don't wait for your leader or others to send an email; often your activity will motivate or oblige your team members to become active as well.
- **Choose something you care about**
  Or at least choose something that you find interesting. It will make the whole experience much more fun!

8 Appendix

ast.ml

```
module StringMap = Map.Make(String)

type binop = Add | And | Sub | Mult | Div | Expon | Equal | Neq | Less | Leq |
    Greater | Geq | Or | Mod | ListConcat | ListBuild
type unop = Neg | Not
type assignop = Assign | MemoAssign

type expr =
  (* Literal of literal*)
    IntLiteral of int
  | BoolLiteral of bool
  | FloatLiteral of float
  | Id of string
  | CondVar
  | ExprSeq of expr * expr
  | Eval of string * expr list * expr (* id args predicate *)
  | Binop of expr * binop * expr
  | Unop of unop * expr
  | If of expr * expr * expr
  | Match of expr * pattern_match
  | ValBind of val_bind list * expr
  | FuncBind of func_bind list * expr
  | ListBuilder of expr list
  | GetIndex of string * expr
  | Noexpr

and pattern =
    Ident of string
  | IntPatt of int
  | BoolPatt of bool
  | FloatPatt of float
  | ListPatt of expr list
  | Wildcard
  | Concat of pattern * pattern

and pattern_match =
    Pattern of pattern * expr * pattern_match
  | NoPattern
```

```
(* let <type> : <name> = <expr> *)
and val_bind = {
    vdecl : decl;
    vexpr : expr;
}

(* <type> : <name> *)
and decl = {
    dtype : fv_type;
    dname : string;
}

(* <fdecl> <assignop> <expr>*)
and func_bind = {
    fdecl : func_decl;
    op : assignop;
    body : expr;
}

(* <type> : <fname> { <type> : <arg> } *)
and func_decl = {
    freturn : fv_type;
    fname : string;
    fargs : decl list
}

(* For the symbol table *)
and sym_table = {
    table : fv_type StringMap.t;
    parent : sym_table option;
}

(* Types *)
and fv_type = FuncType of func_type | ValType of t
and func_type = {
    args_t : fv_type list;
    return_t : fv_type;
}
and t = Void | Bool | Int | Float | List of t

type program = expr

(* to string ... *)

let rec string_of_expr indent expr =
  let more_indent = "    " ^ indent in
  match expr with
```

```
      IntLiteral(i) -> indent ^ "IntLit␣" ^ (string_of_int i) ^ "\n"
    | BoolLiteral(b) -> indent ^ "BoolLit␣" ^ (string_of_bool b) ^ "\n"
    | FloatLiteral(f) -> indent ^ "FloatLit␣" ^ (string_of_float f) ^ "\n"
    | Id(id) -> indent ^ "Id␣" ^ id ^ "\n"
    | CondVar -> indent ^ "CondVar\n"
    | ExprSeq(e1, e2) ->
  indent ^ "ExprSeq\n" ^ (string_of_expr more_indent e1) ^ (string_of_expr more_indent e2)
    | Eval(id, args, p) ->
  let expr_strs = String.concat "" (List.map (string_of_expr more_indent) args) in
  indent ^ "Eval␣" ^ id ^ "\n" ^ expr_strs ^  (string_of_expr more_indent p)
    | Binop(e1, op, e2) -> indent ^ "Binop␣" ^ (string_of_binop op) ^ "\n" ^ (string_of_expr more_indent e1) ^ (string_of_expr more_indent e2)
    | Unop(op, e) -> indent ^ "Unop␣" ^ (string_of_unop op) ^ "\n" ^ (string_of_expr more_indent e)
    | If(pred, e1, e2) ->  indent ^ "IfThenElse\n" ^  (string_of_expr more_indent pred) ^ (string_of_expr more_indent e1) ^ (string_of_expr more_indent e2)
    | ValBind(bindings, e) -> indent ^ "ValBindings\n" ^ (String.concat "" (List.map (string_of_val_bind more_indent) bindings)) ^ (string_of_expr more_indent e)
    | FuncBind(bindings, e) -> indent ^ "FuncBindings\n" ^ (String.concat "" (List.map (string_of_func_bind more_indent) bindings)) ^ (string_of_expr more_indent e)
    | Noexpr -> indent ^ "Noexpr\n"
    | ListBuilder(l) -> indent ^ "ListBuilder\n" ^ (String.concat "\n" (List.map (string_of_expr more_indent) l))
    | _ -> raise Not_found


and string_of_val_bind indent fb =
  ""

and string_of_func_bind indent fb =
  let more_indent = "␣␣␣␣" ^ indent in
  indent ^ "FuncBind␣" ^ (string_of_assignop fb.op) ^ "\n" ^ (string_of_func_decl more_indent fb.fdecl) ^  (string_of_expr more_indent fb.body)

and string_of_func_decl indent fd =
  indent ^ "FuncDecl(" ^ fd.fname ^ ",␣" ^ (string_of_fv_type fd.freturn) ^ ",␣" ^ (String.concat "␣" (List.map string_of_decl fd.fargs)) ^ "\n"

and string_of_decl d =
  "Decl(" ^ d.dname ^ ",␣" ^ (string_of_fv_type d.dtype) ^ ")"

```

```
and string_of_fv_type = function
    FuncType(ft) -> "FuncType(" ^ (string_of_fv_type ft.return_t) ^ ",␣" ^
       (String.concat ",␣" (List.map string_of_fv_type ft.args_t)) ^ ")"
  | ValType(vt) -> "ValType(" ^ (string_of_t vt) ^ ")"

and string_of_t = function
    Void -> "Void"
  | Bool -> "Bool"
  | Int  -> "Int"
  | Float -> "Float"
  | List typ -> (string_of_t typ) ^ "[]"

and string_of_binop = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Equal -> "="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | Mod -> "%"
  | Expon -> "**"
  | ListConcat -> "@"
  | ListBuild -> "::"
  | Or -> "||"
  | And -> "&&"

and string_of_unop = function
  | Neg -> "-"
  | Not -> "!"

and string_of_assignop = function
  | Assign -> "="
  | MemoAssign -> ":="
```

builtin.ml

```
(* builtin ocaml functionality that yappl-generated code needs to access *)

open Random
open Ast
```

```
open Unix

module Builtin =
  struct
    let pred_special_var = "pred_var"
    let builtins = ["print", FuncType { args_t = []; return_t = ValType Bool };
        "rand", FuncType  { args_t = []; return_t = ValType Float };
        "seed", FuncType  { args_t = []; return_t = ValType Bool }   ]

    let rec cond_eval pred f =
      let x = f ()
      in
      if pred x then
  x
      else
  cond_eval pred f

    let rand () = Random.float 1.0

    let seed () = Random.init (int_of_float(10000. *.
        fst(modf(Unix.gettimeofday())))); true
  end
```

builtin.mli

```
open Ast

module Builtin :
    sig
      val pred_special_var : string
      val builtins : (string * fv_type) list

      val cond_eval : ('a -> bool) -> (unit -> 'a) -> 'a
      val rand : unit -> float
      val seed : unit -> bool
    end
```

parser.mly

```
%{ open Ast %}

```

```
%token SEMI LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK
%token COMMA COLON CONCAT ATTACH BAR GIVEN TILDE
%token PLUS MINUS TIMES DIVIDE MOD EXPON
%token NOT AND OR IN LET BIND_SEP
%token EQSYM NEQ LT LEQ GT GEQ MEMOEQ
%token IF ELSE THEN INT FLOAT BOOL FUN COND_VAR IN
%token MATCH WITH ARROW WILDCARD
%token STRING // unused but reserved
%token <bool> BOOL_LITERAL
%token <float> FLOAT_LITERAL
%token <int> INT_LITERAL
%token <string> ID
%token EOF

%nonassoc IN
%nonassoc below_SEMI
%nonassoc SEMI
%nonassoc above_SEMI
%nonassoc LET FUN
%nonassoc WITH
%nonassoc BIND_SEP
%nonassoc THEN
%nonassoc ELSE
%nonassoc IF MATCH
%left COLON
%nonassoc below_BAR
%left BAR
%nonassoc GIVEN
%nonassoc ARROW
%right OR
%right AND
%nonassoc below_EQUAL
%left MEMOEQ EQSYM NEQ LT GT LEQ GEQ
%right CONCAT
%right ATTACH
%left PLUS MINUS
%left TIMES DIVIDE MOD
%right EXPON
%nonassoc NOT
%nonassoc TILDE
%nonassoc LPAREN RPAREN
%nonassoc ID COND_VAR
%left LBRACK
%nonassoc RBRACK BOOL_LITERAL FLOAT_LITERAL INT_LITERAL LBRACE COMMA USCORE
    INT FLOAT BOOL

%start program
```

```
%type <Ast.program> program

%%

program:
   /* nothing  { None }*/
  seq_expr          { $1 }

seq_expr:
| expr        %prec below_SEMI { $1 }
| expr SEMI              { $1 }
| expr SEMI seq_expr     { ExprSeq($1,$3) }

expr:
    expr_core { $1 }
  | binop { $1 }


binop:
  | expr PLUS   expr { Binop($1, Add,    $3) }
  | expr MINUS  expr { Binop($1, Sub,    $3) }
  | expr TIMES  expr { Binop($1, Mult,   $3) }
  | expr DIVIDE expr { Binop($1, Div,    $3) }
  | expr MOD    expr { Binop($1, Mod,    $3) }
  | expr EXPON  expr { Binop($1, Expon,   $3) }
  | expr EQSYM  expr { Binop($1, Equal,  $3) }
  | expr NEQ    expr { Binop($1, Neq,    $3) }
  | expr LT     expr { Binop($1, Less,   $3) }
  | expr LEQ    expr { Binop($1, Leq,    $3) }
  | expr GT     expr { Binop($1, Greater,$3) }
  | expr GEQ    expr { Binop($1, Geq,    $3) }
  | expr CONCAT expr { Binop($1, ListConcat, $3) }
  | expr ATTACH expr { Binop($1, ListBuild, $3) }
  | expr OR     expr { Binop($1, Or, $3) }
  | expr AND    expr { Binop($1, And, $3) }

expr_core:
  | BOOL_LITERAL     { BoolLiteral($1) }
  | INT_LITERAL      { IntLiteral($1) }
  | FLOAT_LITERAL    { FloatLiteral($1) }
  | LPAREN seq_expr RPAREN { $2 }
  | ID               { Id($1) }
  | COND_VAR         { CondVar }
  | NOT expr         { Unop(Not, $2) }
  | MINUS expr       %prec TIMES { Unop(Neg, $2) }
  | TILDE ID expr_seq_opt cond_opt  { Eval($2, $3, $4) }
  | IF seq_expr THEN expr ELSE expr { If($2, $4, $6) }
```

```
  | FUN func_bind IN seq_expr { FuncBind($2, $4) }
  | LBRACK expr_list_opt RBRACK { ListBuilder($2) }
  | LET val_bind_list IN seq_expr {ValBind($2,$4) }
  | MATCH seq_expr WITH pattern_match  { Match($2, $4) }
  | ID LBRACK expr RBRACK { GetIndex($1,$3) }

/* Function binding */

func_bind:
  function_decl assn_op seq_expr
      { [{ fdecl = $1;
   op = $2;
   body = $3}] }

function_decl:
  fvtype COLON ID args
    { { freturn = $1;
        fname = $3;
        fargs = List.rev $4} }

assn_op:
    EQSYM { Assign }
  | MEMOEQ { MemoAssign }

fvtype:
  | LPAREN fvtype RPAREN { $2 }
  | FUN fvtype fvtype_list_opt { FuncType({args_t = List.rev $3; return_t =
      $2}) }
  | t { ValType $1 }

fvtype_list_opt:
     /* nothing */ { [] }
  |  fvtype_list_opt fvtype { $2 :: $1 }

t:
    INT { Int }
  | FLOAT { Float }
  | BOOL { Bool }
  | t LBRACK RBRACK { List $1 }

args:
  /* nothing */ {[]}
  | args decl { $2 :: $1 }

decl:
  fvtype COLON ID
    { { dtype = $1;
```

```
      dname = $3 } }


/* Function evaluation */

expr_seq_opt:
    /* nothing */   %prec above_SEMI { [] }
  | expr_seq       %prec above_SEMI { List.rev $1 }

expr_seq:
     expr          %prec above_SEMI  { [$1] }
   | expr_seq expr %prec above_SEMI { $2 :: $1 }


cond_opt:
  /* nothing*/ %prec below_BAR { Noexpr }
  | GIVEN expr  { $2 }


/* Lists */

expr_list_opt:
    /* nothing */ { [] }
  |expr_list          {$1}

expr_list:
    expr                         {[$1]}
  | expr_list COMMA expr          { $3 :: $1 }


/* Value binding */

val_decl:  decl  { $1 }

val_bind_list:
   val_bind {[$1]}
  | val_bind_list BIND_SEP val_bind { $3 :: $1 }

val_bind:
   val_decl EQSYM seq_expr {{vdecl = $1; vexpr = $3}}


/* Pattern matching */

pattern_match:
  | bar_opt pattern ARROW seq_expr { Pattern($2, $4, NoPattern) }
  | pattern_match BAR pattern ARROW seq_expr   { Pattern($3, $5, $1) }
```

```

bar_opt:
  | /* nothing */ {}
  | BAR  {}

pattern:
  | LPAREN pattern RPAREN { $2 }
  | ID %prec below_EQUAL { Ident($1) }
  | INT_LITERAL {IntPatt $1}
  | BOOL_LITERAL {BoolPatt $1}
  | FLOAT_LITERAL {FloatPatt $1}
  | LBRACK RBRACK { ListPatt [] }
  | WILDCARD { Wildcard }
  | pattern ATTACH pattern { Concat($1, $3) }
```

scanner.mll

```
{ open Parser }

let digit = [’0’-’9’]
let exp = ’e’ [’-’ ’+’]? digit+
let opt1 = digit+ ’.’ digit* exp?
let opt2 = digit+ exp
let opt3 = ’.’ digit+ exp?

rule token = parse
  [’ ’ ’\t’ ’\r’ ’\n’] { token lexbuf } (* Whitespace *)
| "###"    { comment lexbuf }           (* Comments *)
| "#"      { line_comment lexbuf }
| ’(’      { LPAREN }
| ’)’      { RPAREN }
| ’{’      { LBRACE }
| ’}’      { RBRACE }
| ’[’      { LBRACK }
| ’]’      { RBRACK }
| ’$’      { COND_VAR }
| ’|’      { BAR }
| "given"  { GIVEN }
| ’~’      { TILDE }
| "@"      { CONCAT }
| "::"     { ATTACH }
| ":"      { COLON }
| ’;’      { SEMI }
| ’,’      { COMMA }
```

```
| '!'    { NOT }
| '+'      { PLUS }
| '-'      { MINUS }
| "**"     { EXPON }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '%'      { MOD }
| '='      { EQSYM }
| ":="     { MEMOEQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "if"     { IF }
| "else"   { ELSE }
| "then"   { THEN }
| "&&"     { AND }
| "and"    { BIND_SEP }
| "||"     { OR }
| "in"     { IN }
| "let"    { LET }
| "fun"    { FUN }
| "int"    { INT }
| "float"  { FLOAT }
| "bool"   { BOOL }
| "true"   { BOOL_LITERAL(true) }
| "false"  { BOOL_LITERAL(false) }
| "match"  { MATCH }
| "with"   { WITH }
| "string" { STRING } (* our LRM states this is reserved, but it's not being
    used *)
| "->"     { ARROW }
| "_"      { WILDCARD }
| (opt1 | opt2 | opt3) as lxm { FLOAT_LITERAL(float_of_string lxm) }
| digit+ as lxm { INT_LITERAL(int_of_string lxm) }
| "$" as lxm      { ID(String.make 1 lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal␣character␣" ^ Char.escaped char)) }

and comment = parse
  "###" { token lexbuf }
| _     { comment lexbuf }

and line_comment = parse
  ['\n' '\r'] { token lexbuf }
```

```
| _           { line_comment lexbuf }
```

translate.ml

```
(* translate to ocaml *)

open Ast
open Builtin
open Str

exception No_such_symbol_found of string
exception Function_identifier_expected of string
exception Argument_count_mismatch
exception Argument_type_mismatch
exception Error of string

(* utility functions *)

let match_num_types  = function
    ValType(Int), ValType(Int) -> Some(Int)
  | ValType(Float), ValType(Float) -> Some(Float)
  | ValType(x), ValType(y) -> print_endline ((string_of_t x) ^ ",␣" ^
      (string_of_t y)); None
  | _ -> None

let listtype_to_single_type = function
  | ValType(List(Int)) -> ValType(Int)
  | ValType(List(Float)) -> ValType(Float)
  | ValType(List(Bool)) -> ValType(Bool)
  | _ -> ValType(Void)

(* lookup a identifier in the symbol table, recursing upward as necessary *)
let rec sym_table_lookup table id =
  try
    StringMap.find id table.table
  with Not_found  ->
    match table.parent with
      Some(p) -> sym_table_lookup p id
    | None    -> raise (No_such_symbol_found id)

let id_to_ocaml_id = function
    "rand" | "seed" as id -> "Builtin." ^ id
  | _ as id ->
      if id = Builtin.pred_special_var then
```

```
  id
      else
  "yappl_" ^ id

(* expr to string functions *)

let rec ident_to_string table id =
  id_to_ocaml_id id, (sym_table_lookup table id)

and seq_to_string table e1 e2 =
  let (s1, _) = expr_to_string table e1 in
  let (s2, t) = expr_to_string table e2 in
  ("(ignore␣(␣" ^ s1 ^ "␣));\n" ^ s2, t )

and type_to_string vt =
 match vt with
   ValType(Int) -> "int"
 | ValType(Bool) -> "bool"
 | ValType(Float) -> "float"
 | ValType(Void) -> "void"
 | _ -> raise (Error "unexpected␣type")

(* "expected ... got" string *)
and eg2s s1 s2 =
 ("␣Expected␣" ^ type_to_string s1 ^ ",␣surprised␣by␣" ^ type_to_string s2 )

and ocaml_lstring_to_yappl ls t =
   let pc = match t with
        Int -> "print_int"
       |Bool -> "print_bool"
       |Float -> "print_float"
       |_     -> raise (Error("Unsupported␣type␣for␣printing:␣" ^ (string_of_t (List t))))
   in
   "(␣print_char␣'['␣)␣;␣␣(␣match␣(␣" ^ ls ^"␣)␣with␣|␣[]␣->␣()␣|␣h::t␣->␣(␣" ^ pc ^ "␣h)␣;" ^
   "ignore␣(␣List.map␣(fun␣i␣->␣print_char␣','␣;" ^ pc ^ "␣i)␣t␣)␣);␣␣(␣print_char␣']')"


and eval_to_string table id args p =
  match id with
    "print_line" ->
       let p,t = (eval_to_string table "print" args p ) in
         "ignore␣(␣" ^ p ^ "␣);\n␣print_newline␣();␣true", ValType Bool
   | "print" ->
      (match p with
```

```
Noexpr ->
  let arg =
    match args with
      arg :: [] -> arg
    | _ -> raise (Error("invalid␣number␣of␣args␣to␣print"))
  in
  let ret_t = ValType Bool in
  let (arg_s, arg_t) = expr_to_string table arg in
  (match arg_t with
    ValType Bool -> "print_string␣(string_of_bool␣(␣" ^ arg_s ^ "␣));␣
        print_char␣'␣';␣true", ret_t
  | ValType Int -> "print_int␣(␣" ^ arg_s ^ "␣);␣print_char␣'␣';␣true", ret_t
  | ValType Float -> "print_float␣(␣" ^ arg_s ^ "␣);␣print_char␣'␣';␣true",
      ret_t
        | ValType List(t) -> (ocaml_lstring_to_yappl arg_s t), ret_t
  | _ -> raise (Error("unsupported␣print␣expression␣type")))
    | _ -> raise (Error("print␣does␣not␣support␣predicates")))
| _ ->
  match sym_table_lookup table id with
    FuncType ft ->
let rev_args_and_types = List.rev_map (expr_to_string table) args in
let check b ea at =
  let (_, et) = ea in
  b || et <> at
in
let err = try (* check that arg and actual expr types match *)
  List.fold_left2 check false rev_args_and_types (List.rev ft.args_t)
with Invalid_argument s ->
  raise Argument_count_mismatch
in
if err then
  raise Argument_type_mismatch
else
  let eval_str_no_unit = (id_to_ocaml_id id) ^ "␣" ^ (String.concat "␣"
      (List.rev_map (fun (s, _) -> "(␣" ^ s ^ "␣)") rev_args_and_types)) in
  let str =
    match p with
      Noexpr -> eval_str_no_unit ^ "␣()"
    |  _ ->
  let temp_table = { table with table = StringMap.add
      Builtin.pred_special_var ft.return_t table.table } in (* add special
      predicate value *)
  let (pred, ptype) = expr_to_string temp_table p in
  if ptype <> ValType(Bool) then
    raise (Error "predicate␣does␣not␣evaluate␣to␣boolean")    (*predicate
        does not evaluate to boolean*)
  else
```

```
      "Builtin.cond_eval␣(␣fun␣" ^ Builtin.pred_special_var ^ "␣->␣" ^  pred ^ "␣)␣(␣" ^ eval_str_no_unit ^ "␣)"
    in
    str, ft.return_t
    | _ -> raise (Function_identifier_expected id)

and binop_to_string table e1 e2 op =
  let (s1, t1) = expr_to_string table e1
  and (s2, t2) = expr_to_string table e2
  in
  let ocaml_op, return_t =  (* there should be a better way to do this *)
    match op with
      Add ->
  (match match_num_types (t1, t2) with
    Some(Int) -> "+", ValType(Int)
  | Some(Float) -> "+.", ValType(Float)
  | _ -> raise (Error("Type␣mismatch␣for␣+")))
    | Sub ->
  (match match_num_types (t1, t2) with
    Some(Int) -> "-", ValType(Int)
  | Some(Float) -> "-.", ValType(Float)
  | _ -> raise (Error("Type␣mismatch␣for␣-")))
    | Mult ->
  (match match_num_types (t1, t2) with
    Some(Int) -> "*", ValType(Int)
  | Some(Float) -> "*.", ValType(Float)
  | _ -> raise (Error("Type␣mismatch␣for␣*")))
    | Div ->
  (match match_num_types (t1, t2) with
    Some(Int) -> "/", ValType(Int)
  | Some(Float) -> "/.", ValType(Float)
  | _ -> raise (Error("Type␣mismatch␣for␣/")))
    | Expon ->
  (match match_num_types (t1, t2) with
    Some(Float) -> "**", ValType(Float)
  | _ -> raise (Error("Type␣mismatch␣for␣**")))
    | Equal ->
  if t1 = t2 then
    "=", ValType(Bool)
  else
    raise (Error("Type␣mismatch␣for␣="))
    | Neq ->
  if t1 = t2 then
    "<>", ValType(Bool)
  else
    raise (Error("Type␣mismatch␣for␣!="))
    | Less ->
```

```
  (match match_num_types (t1, t2) with
    Some(_) -> "<", ValType(Bool)
  | None -> raise (Error("Type␣mismatch␣for␣<")))
    | Leq ->
  (match match_num_types (t1, t2) with
    Some(_) -> "<=", ValType(Bool)
  | None -> raise (Error("Type␣mismatch␣for␣<=")))
    | Greater ->
  (match match_num_types (t1, t2) with
    Some(_) -> ">", ValType(Bool)
  | None -> raise (Error("Type␣mismatch␣for␣>")))
    | Geq ->
  (match match_num_types (t1, t2) with
    Some(_) -> ">=", ValType(Bool)
  | None -> raise (Error("Type␣mismatch␣for␣>=")))
    | Mod ->
  if t1 = ValType(Int) && t2 = ValType(Int) then
    "mod", ValType(Int)
  else
    raise (Error("Invalid␣types␣for␣%"))
    | Or ->
        (match (t1,t2) with
         (ValType(Bool),ValType(Bool)) -> "||", ValType(Bool)
        | _ -> raise (Error("Type␣mismatch␣for␣or")))
    | And ->
        (match (t1,t2) with
         (ValType(Bool),ValType(Bool)) -> "&&", ValType(Bool)
        | _ -> raise (Error("Type␣mismatch␣for␣and")))
    | ListConcat ->
  (match (t1,t2) with
    ValType(List(lt1)), ValType(List(lt2)) when lt1 = lt2 -> "@", t1
  | _ -> raise (Error("Type␣mismatch␣for␣@")))
    | ListBuild ->
  (match (t1,t2) with
    ValType(lt1), ValType(List(lt2)) when lt1 = lt2 -> "::", t2
  | ValType(lt1), ValType(List Void) -> "::", ValType(List lt1)
  | _ -> raise (Error("Type␣mismatch␣for␣::␣" ^ (string_of_fv_type t1) ^ "␣" ^
      (string_of_fv_type t2))))
  in
  "(␣" ^ s1 ^ "␣)␣" ^ ocaml_op ^  "␣(␣" ^ s2 ^ "␣)", return_t

and unop_to_string table e op =
  let (s, et) = expr_to_string table e in
  let opstr =
    match op, et with
      Not, ValType(Bool) -> "not␣(␣"
    | Neg, ValType(Int)
```

```
    | Neg, ValType(Float) -> "(-␣"
    | _ -> raise (Error("Type␣mismatch␣with␣unary␣operator"))
  in
     opstr ^ s ^ "␣)", et

and if_to_string table pred e1 e2 =
    let (pred_str, pt) = expr_to_string table pred in
    if pt <> ValType(Bool) then
      raise (Error("Predicate␣for␣if␣expression␣not␣a␣boolean"))
    else
      let (s1, t1) = expr_to_string table e1
      and (s2, t2) = expr_to_string table e2 in
      if t1 = t2 then
  "if␣(␣" ^ pred_str ^ "␣)␣then␣(␣" ^ s1 ^ "␣)␣else␣(␣" ^ s2 ^ "␣)", t1
      else if s2 = "" then (* in case there was no else *)
  "if␣(␣" ^ pred_str ^ "␣)␣then␣(␣" ^ s1 ^ "␣)", t1
      else
  raise (Error("Type␣mismatch␣of␣if␣expressions"))


and list_to_string table l =
    match l with
      []  -> "[]", ValType(List Void)
    | _   -> let head = List.hd (List.rev l) in
             let (_,vt) = expr_to_string table head in
             let sl = List.map ( fun e ->
          (match expr_to_string table e with
                 (s1, t1) -> if t1 = vt then s1
                 else raise (Error("Type␣mismatch␣in␣list" ^ (eg2s vt t1)))
                                )
             ) l in
             (* Need type t to construct a list type from the enumerated type *)
             match (vt) with
              (ValType ty) -> let t = ty in
                 ("[␣" ^ (String.concat "␣;␣" (List.rev sl)) ^ "␣]"), ValType(List(t))
             |_ -> raise (Error("Functions␣not␣allowed␣in␣lists."))

and string_at_index table s e =
     try
      let vt =  (sym_table_lookup table s) in
      let es,et = expr_to_string table e in
      if (et <> ValType(Int)) then
        raise(Error("Invalid␣index.␣Must␣be␣integer."))
      else
```

```
        ("(List.nth␣" ^ (id_to_ocaml_id s) ^ "␣(␣" ^ es ^ "␣))" ), (listtype_to_single_type vt)
      with No_such_symbol_found id ->
        raise (Error("Unbound␣symbol␣" ^ id ^ "␣referenced"))

(* pattern matching *)
and match_to_string table e p =
    let es,mt = expr_to_string table e in
    let match_table = { table = StringMap.empty; parent = Some(table) } in
    let (pls,pmt) = pattlist_to_string match_table p mt in
    "␣match␣(␣" ^ es ^ "␣)␣with␣" ^ pls, pmt

(* mt = match type, for type inference *)
and pattlist_to_string table pl mt =
   match (pl) with
     Pattern (pat , exp, pmatch) -> let (patstring, new_table) = pat_to_string table pat mt in
                                      let (es, pt) =  expr_to_string new_table exp in
              let (ps, _ ) = pattlist_to_string table pmatch mt in
                                      ps ^ "\n|␣" ^ patstring ^ "␣->␣" ^ es, pt
    | NoPattern -> "", ValType(Void)

and pat_to_string table p mt =
    match (p) with
    | ListPatt lp -> "[]", table
    | Ident s -> patid_to_string table s mt
    | IntPatt i  -> string_of_int i, table
    | BoolPatt b -> string_of_bool b, table
    | FloatPatt f -> string_of_float f, table
    | Wildcard -> "_", table
    | Concat (p1, p2) -> let (p1s, table1) = pat_to_string table  p1 (listtype_to_single_type mt ) in
                           let (p2s, table2) = pat_to_string table1 p2 mt in
                           p1s ^ "::" ^ p2s, table2

(* adds symbol to table, clobbers existing symbols *)
and patid_to_string table s mt =
      try
      ignore (sym_table_lookup table s);
      raise (Error("Type␣mismatch␣in␣concatenation"))
      with  No_such_symbol_found _ ->
        let new_table = { table with table = StringMap.add s mt table.table } in
        id_to_ocaml_id s, new_table


```

```ocaml
and val_bindings_to_string table bindings e =
  let proc (tabl, s) vb =
    let (new_tabl, new_s) = val_bind_to_string tabl vb in
    new_tabl, s ^ "␣\n␣" ^ new_s
  in
  let (new_table, bstr) = List.fold_left proc (table, "") (List.rev bindings) in
  let (s, et) = expr_to_string new_table e in
  bstr ^ "\n␣(␣" ^ s ^ "␣)", et

and val_bind_to_string table vb =
  try
    ignore (sym_table_lookup table vb.vdecl.dname);  (* make sure id doesn't already exist *)
    raise (Error("Duplicate␣value␣identifier:␣" ^  vb.vdecl.dname))
  with No_such_symbol_found _ ->
    let (s, et) = expr_to_string table vb.vexpr in
    if et <> vb.vdecl.dtype then
      raise (Error("Incompatible␣type␣for␣value␣binding"))
    else
      let new_table = { table with table = StringMap.add vb.vdecl.dname et table.table } in
      new_table, "let␣yappl_" ^ vb.vdecl.dname ^ "␣=␣" ^ s ^ "␣in␣"


and func_bindings_to_string table bindings e  =
  let proc (tabl, s) fb =
    let (new_tabl, new_s) = func_bind_to_string tabl fb in
    new_tabl, s ^ new_s
  in
  let (new_table, bstr) = List.fold_left proc (table, "") (List.rev bindings) in
  let (s, et) = expr_to_string new_table e in
  bstr ^ s, et

and func_bind_to_string table fb =
   try
     ignore (sym_table_lookup table fb.fdecl.fname); (* make sure id doesn't already exist *)
     raise (Error("Duplicate␣function␣identifier:␣" ^  fb.fdecl.fname))
  with No_such_symbol_found _ ->
     let build_table (tabl, args_t) decl =
       let new_tabl = StringMap.add decl.dname decl.dtype tabl in
       new_tabl, decl.dtype :: args_t
     in
     let func_table, args_t = List.fold_left build_table (StringMap.empty, []) fb.fdecl.fargs in
```

```
      let func_t = FuncType { args_t = List.rev args_t; return_t = fb.fdecl.freturn } in
      let new_table = { table with table = StringMap.add fb.fdecl.fname func_t table.table } in
      let (body_s, body_t) = expr_to_string { table = func_table; parent = Some(new_table) } fb.body in
      if body_t <>  fb.fdecl.freturn  then
        raise (Error("mismatched␣return␣and␣function␣body␣types␣for␣" ^ fb.fdecl.fname ^ ":␣" ^ (string_of_fv_type body_t) ^ "␣" ^ (string_of_fv_type fb.fdecl.freturn)))
      else
        let arg_names = List.map (fun decl -> id_to_ocaml_id decl.dname) fb.fdecl.fargs in
        let oid = id_to_ocaml_id fb.fdecl.fname in
        let arg_str = String.concat "␣" arg_names in
        match fb.op with
    Assign ->
      new_table, "let␣rec␣" ^ oid  ^ "␣" ^ arg_str ^ "␣unit␣=␣\n␣" ^ body_s ^ "␣\nin\n"
        | MemoAssign ->
      let t_oid  = "table_" ^ oid and nm_oid = "no_mem_" ^ oid
      and tbl_name = "hash_table_for_" ^ oid and arg_tpl = String.concat ",␣" arg_names
      in
      let body_s_fix = Str.global_replace (Str.regexp ("␣" ^ oid ^ "␣")) ("␣" ^ t_oid  ^ "␣tabl␣") body_s
      in
      new_table, "let␣rec␣" ^ t_oid ^ "␣tabl␣" ^ arg_str ^ "␣unit␣=␣\n␣" ^
      "let␣rec␣" ^ nm_oid  ^ "␣" ^ arg_str ^ "␣unit␣=␣\n␣" ^ body_s_fix ^ "\nin\n" ^
      "try␣Hashtbl.find␣tabl␣(␣" ^ arg_tpl ^ "␣)␣with␣Not_found␣->\n" ^
      "let␣result␣=␣" ^ nm_oid ^ "␣" ^ arg_str ^ "␣()␣in␣\n" ^
      "Hashtbl.add␣tabl␣" ^ arg_tpl ^ "␣result;␣result\nin\n" ^
      "let␣" ^ tbl_name ^ "␣=␣Hashtbl.create␣50␣in\n" ^
      "let␣" ^ oid ^ "␣=␣" ^ t_oid ^ "␣" ^ tbl_name ^ "␣in\n"

and expr_to_string table = function
    IntLiteral(i) -> string_of_int i, ValType(Int)
  | BoolLiteral(b) -> string_of_bool b, ValType(Bool)
  | FloatLiteral(f) -> string_of_float f, ValType(Float)
  | Id(id) -> ident_to_string table id
  | CondVar -> ident_to_string table Builtin.pred_special_var
  | ExprSeq(e1, e2) -> seq_to_string table e1 e2
  | Eval(id, args, p) -> eval_to_string table id args p
  | Binop(e1, op, e2) -> binop_to_string table e1 e2 op
  | Unop(op, e) -> unop_to_string table e op
  | If(pred, e1, e2) -> if_to_string table pred e1 e2
```

```
  | ValBind(bindings, e) -> val_bindings_to_string table bindings e
  | FuncBind(bindings, e) -> func_bindings_to_string table bindings e
  | ListBuilder(l) -> list_to_string table l
  | GetIndex(l, e) -> string_at_index table l e
  | Match(e,p) -> match_to_string table e p
  | Noexpr -> "", ValType(Void)
  (*| _ -> raise (Error "unsupported expression type")*)

let translate prog =
  (*print_endline (string_of_expr "" prog);*)
  let init_table = List.fold_left (fun tabl (id, id_t) -> StringMap.add id id_t tabl) StringMap.empty Builtin.builtins in
  let global_sym_table = { table = init_table; parent = None } in
  let s, _ = expr_to_string global_sym_table prog in
  "open␣Builtin\nopen␣Hashtbl\n\nlet␣_␣=\n" ^ s
```

yappl.ml

```
let lexbuf = Lexing.from_channel stdin in
let program = Parser.program Scanner.token lexbuf in
print_endline (Translate.translate program);
```

errors.sh

```
#bin/bash

SUMMARY=$"PASS:␣All␣errors␣generated"

for i in $(\ls errors)
do
   ERROR=$(  ./yappl < errors/$i  2>&1 1>/dev/null)
   RESULT=$(echo $ERROR $i | grep -v error)

   if [ $RESULT ]
   then
     echo "Error␣not␣generated␣by␣:␣" $RESULT
     SUMMARY=$"FAIL:␣some␣errors␣not␣generated."
   fi
   unset RESULT
done

```

```
echo $SUMMARY
```

Makefile

```
OBJS = ast.cmo parser.cmo scanner.cmo builtin.cmo translate.cmo yappl.cmo

yappl : $(OBJS)
  ocamlc -o yappl str.cma unix.cma $(OBJS)

debug : $(OBJS)
  ocamlc -g -o yappl unix.cma $(OBJS)

scanner.ml : scanner.mll
  ocamllex scanner.mll

parser.ml parser.mli : parser.mly
  ocamlyacc parser.mly

%.cmo : %.ml
  ocamlc -g -c $<

%.cmi : %.mli
  ocamlc -g -c $<

.PHONY : clean
clean :
  rm -f yappl parser.ml parser.mli scanner.ml \
  testall.log *.cmo *.cmi

# Generated by ocamldep *.ml *.mli
ast.cmo:
ast.cmx:
builtin.cmo: ast.cmo builtin.cmi
builtin.cmx: ast.cmx builtin.cmi
parser.cmo: ast.cmo parser.cmi
parser.cmx: ast.cmx parser.cmi
scanner.cmo: parser.cmi
scanner.cmx: parser.cmx
translate.cmo: builtin.cmi ast.cmo
translate.cmx: builtin.cmx ast.cmx
yappl.cmo: translate.cmo scanner.cmo parser.cmi
yappl.cmx: translate.cmx scanner.cmx parser.cmx
builtin.cmi: ast.cmo
parser.cmi: ast.cmo
```

stdlib.ypl

```
# START STDLIB

# flip based on supplied probability
# ~flip .5 is a fair coin toss
fun bool:flip float:bias = ~rand <= bias in
fun bool:fflip = ~flip .5 in

# geometric distribution
fun int:geom float:q =
  fun int:geom_helper float:orig_q int:i =
    if ~rand < orig_q then i
    else ~geom_helper orig_q (i+1)
  in
    ~geom_helper q 1
in

# END STDLIB
```

testall.sh

```
#!/bin/sh

YAPPL="./yappl"

# Set time limit for all operations
ulimit -t 30

globallog=testall.log
rm -f $globallog
error=0
globalerror=0

keep=0

Usage() {
    echo "Usage:␣testall.sh␣[options]␣[.ypl␣files]"
    echo "-k␣␣␣␣Keep␣intermediate␣files"
    echo "-h␣␣␣␣Print␣this␣help"
    exit 1
}

```

```
SignalError() {
    if [ $error -eq 0 ] ; then
  echo "FAILED"
  error=1
    fi
    echo "␣␣$1"
}

# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to difffile
Compare() {
    generatedfiles="$generatedfiles␣$3"
    echo diff -b $1 $2 ">" $3 1>&2
    diff -b "$1" "$2" > "$3" 2>&1 || {
  SignalError "$1␣differs"
  echo "FAILED␣$1␣differs␣from␣$2" 1>&2
    }
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* || {
  SignalError "$1␣failed␣on␣$*"
  return 1
    }
}

Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.ypl//'`
    reffile=`echo $1 | sed 's/.ypl$//'`
    basedir="`echo␣$1␣|␣sed␣'s/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "######␣Testing␣$basename" 1>&2

    generatedfiles=""

#    generatedfiles="$generatedfiles ${basename}.i.out" &&
#    Run "$YAPPL" "<" $1 ">" ${basename}.i.out &&
#    Compare ${basename}.i.out ${reffile}.out ${basename}.i.diff

```

```
    generatedfiles="$generatedfiles␣${basename}.o.out" &&
    Run "$YAPPL"  "<" $1 ">␣test.ml␣;␣ocamlc␣-o␣test_binary␣unix.cma␣
        builtin.cmo␣test.ml␣;␣./test_binary␣>␣" ${basename}.o.out &&
    Compare ${basename}.o.out ${reffile}.out ${basename}.o.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
  if [ $keep -eq 0 ] ; then
      rm -f $generatedfiles
  fi
  echo "OK"
  echo "######␣SUCCESS" 1>&2
    else
  echo "######␣FAILED" 1>&2
  globalerror=$error
    fi
}

while getopts kdpsh c; do
    case $c in
  k) # Keep intermediate files
      keep=1
      ;;
  h) # Help
      Usage
      ;;
    esac
done

shift `expr $OPTIND - 1`

if [ $# -ge 1 ]
then
    files=$@
else
    files="tests/fail-*.ypl␣tests/test-*.ypl"
fi

for file in $files
do
    case $file in
  *test-*)
      Check $file 2>> $globallog
      ;;
  *fail-*)
      CheckFail $file 2>> $globallog
```

```
    ;;
  *)
    echo "unknown file type $file"
    globalerror=1
    ;;
  esac
done

exit $globalerror
```

yapplc

```
#!/usr/bin/env bash
FILE="$1"
if [ $FILE -a -f $FILE ]
then
  name=$(echo "$FILE" | sed 's/\.[^\.]*$//')
  stdlib="stdlib.ypl"
  if [ -f $stdlib ]
  then
    cat $stdlib $FILE | ./yappl > ${name}.ml
  else
    ./yappl < $FILE > ${name}.ml
  fi

  ocamlc -w -24-26 -c ${name}.ml
  ocamlc -o ${name} unix.cma builtin.cmo ${name}.cmo
  echo "run with ./${name}"
else
  echo "file $FILE does not exist"
fi
```

tutorials/add.ypl

```
fun int:add int:a int:b = a + b in
~print_line ~add 1 2
```

tutorials/dpmem.ypl

```
###
An implementation of the Dirichlet process (DP) using memoization. For an
    explanation of DPs, see

Teh et. al. Hierarchical Dirichlet processes. Journal of the Am. Stat. Assoc.,
    101(476):1566--1581, 2006.
###

# placeholder for a function that would generate a draw from the beta
    distribution (so this is a draw from the Beta(1,1) distribution, no matter
    what a and b are
fun float:beta float:a float:b = ~rand in

# get a stick, breaking more if necessary
fun int:pickastick (fun float int):sticks int:j =
    if ~rand < ~sticks j then j else ~pickastick sticks j+1
in

# generic Dirichlet process code
fun (fun int):DP float:alpha (fun int):proc =
    fun float:sticks int:x := ~beta 1.0 alpha in
    fun int:atoms  int:x := ~proc in
    fun int:f = ~atoms ~pickastick sticks 1 in
    f # return f
in

fun (fun (fun int) float):DPmem float:alpha (fun int float):proc =
    fun (fun int):dps float:arg :=
      fun int:apply = ~proc arg in
      ~DP alpha apply
    in
    fun (fun int):dp float:arg = ~dps arg in
    dp
in

# this function will create Dirichlet process draws with geometric base
    distribution
let (fun (fun int) float):geom_dp = ~DPmem 1.0 geom in

# this is a DP draw with geometric base distribution with q = .2
let (fun int):mydraw = ~geom_dp .2 in

# use a tail-recursive loop to generate some samples from the Dirichlet Process
fun bool:loop int:i =
```

```
    ~print ~mydraw;
    if i > 0 then ~loop i - 1 else true
in
~seed;
~loop 30; ~print_line ~mydraw
```

tutorials/geom.ypl

```
~seed;
# named geom1 so as not to conflict with stdlib geom.
fun int:geom1 float:q =
  fun int:geom1_helper float:orig_q int:i =
    if ~rand < orig_q then
      i
    else
      ~geom1_helper orig_q (i+1)
  in
    ~geom1_helper q 1
in
  ~print_line ~geom1 0.1 given $ > 10
```

tutorials/geom-cond.ypl

```
~seed;
fun int:try_g = ~geom 0.1 given $ > 100 in
~print_line ~try_g;
~print_line ~try_g;
~print_line ~try_g;
~print_line ~try_g;
~print_line ~try_g;

fun int:try_g2 = ~geom 0.1 given $ > 10 in
~print_line ~try_g2;
~print_line ~try_g2;
~print_line ~try_g2;
~print_line ~try_g2;
~print_line ~try_g2;

fun int:try_g3 = ~geom 0.1 given $ + (~geom .7) > 50 in
~print_line ~try_g3;
~print_line ~try_g3;
```

```
~print_line ~try_g3;
~print_line ~try_g3;
~print_line ~try_g3;
```

tutorials/memogeom.ypl

```
fun (fun int int):geom_list_gen float:p =
  fun int:geom_list int:n := ~geom p
  in geom_list
in
~seed;
let (fun int int):g = ~geom_list_gen 0.5 in
~print_line [1, ~g 1, ~g 1];
~print_line [2, ~g 2, ~g 2];
~print_line [3, ~g 3, ~g 3];
~print_line [4, ~g 4, ~g 4];
```

tests/test-array-cons.ypl

```
~print_line 3 :: 4 :: [] @ [1, 2, 3]
```

tests/test-array-cons.out

```
[3,4,1,2,3]
```

tests/test-basic-operators.ypl

```
~print_line (-2.5e-6 + 1.0);
~print_line (-2 + 1000);
~print_line (8 * 5);
~print_line (90 -15);
~print_line (1000/10);
~print_line (2 * (-2) + 102 - 6 / 3);
~print_line (10 % 5)
# this is a single line comment.
```

```
### this
is
a
multiline
comment
###
```

tests/test-basic-operators.out

```
0.9999975
998
40
75
100
96
0
```

tests/test-binding.ypl

```
let int:a = 4 and int:b = 5 in ~print_line ( a + b );
let int[]:x = [1,2,3] in ~print_line x[2] + x[0];
let fun float:r = rand in ~rand;
fun (fun int int):f int:x =
    fun int:g int:y =
      x + y
    in
    g
in
let (fun int int):h = ~f 4 in
~print_line ~h 2
```

tests/test-binding.out

```
9
4
6
```

tests/test-functions.ypl

```
fun int:t1 int:a =
    a + 3
in
~print_line ~t1 2;
~print_line ~t1 -2;

fun int:t2 int:a int:b =
  a + 5 + b
in
~print_line ~t2 1 2;
~print_line ~t2 (-1) (-2)
```

tests/test-functions.out

```
5
1
8
2
```

tests/test-list-indexes.ypl

```
let int[]:fib = [1,1,2,3,5,8,13,21,34,55,89,144] in
let bool:piranha = (fib[3] + fib[4]) = (fib[5])  in
~print_line piranha;
~print_line fib[if ( fib[4] = 5 ) then 7 else 8]
```

tests/test-list-indexes.out

```
true
21
```

tests/test-memo.ypl

```
fun float:memod int:n := ~rand in
```

```
let float:a = ~memod 5 and float:b = ~memod 5
and float:c = ~memod 5 and float:d = ~memod 5
and float:e = ~memod 7 in
~print_line (a = b);
~print_line (b = c);
~print_line (c = d);
~print_line !(e = a)
```

tests/test-memo.out

```
true
true
true
true
```

tests/test-more-operators.ypl

```
~print_line (2 = 2);
~print_line (2 = 3);
~print_line (3 != 2);
~print_line (3 != 3);
~print_line !(3 = 2);
~print_line (-2 < 2);
~print_line (-2 < -2);
~print_line (100 > 100);
~print_line (100 > 50);
~print_line (19 <= 19);
~print_line (17 <= 19);
~print_line (21 >= 21);
~print_line (21 >= 20);
~print_line ((1 != 2) || (5 = 5));
~print_line ((3 != 0) && (8 != 1));
~print_line ([2, 3] @ [4, 5]);
~print_line (1 :: [2, 3])
```

tests/test-more-operators.out

```
true
```

```
false
true
false
true
true
false
false
true
true
true
true
true
true
true
[2,3,4,5]
[1,2,3]
```

tests/test-pattern-matching.ypl

```
let int[]:i = [3,2,1] and
    bool[]:b = [true,false,true] and
    float[]:f = [3.1415, 2.7182, 6.022e23 ]

in match f with
| foo::bar -> ~print_line (bar[1] / 1.434e22);
                ~print_line (b[0] && !false);
                ~print_line (i[2] > i[1])
| _ -> ~print_line 0
```

tests/test-pattern-matching.out

```
41.9944211994
true
false
```

tests/test-pattern-matching-elist.ypl

```
fun bool:partytime int[]:a =
```

```
    match a with
     b::c -> ((~print_line b) && (~partytime c ))
   | [] -> false
in

let int[]:x = [5,4,3,1,0] in

~print_line ( ~partytime x );

(match x with
    _ :: n -> ~print_line n
 | n -> ~print_line 0);

fun int:add int[]:nums =
    match nums with
        n :: rest -> n + ~add rest
  | []        -> 0
in
~print_line ~add x
```

tests/test-pattern-matching-elist.out

```
5
4
3
1
0
false
[4,3,1,0]
13
```

tests/test-pattern-matching-bug1.ypl

```
let int[]:b = [3,2,1,0] in match b with (c::d) -> ~print_line (c * d[0]) | _
    -> ~print_line (4)
```

tests/test-pattern-matching-bug1.out

```
6
```

tests/test-pattern-matching-bug2.ypl

```
let int[]:i = [3,2,1] and
    bool[]:b = [true,false,true] and
    float[]:f = [3.1415, 2.7182, 6.022e23 ]

in match f with
| foo::bar -> ~print_line (bar[1] / 1.434e22)
| _ -> ~print_line 3
```

tests/test-pattern-matching-bug2.out

```
41.9944211994
```

tests/test-print-line.ypl

```
~print_line 1;
~print_line -2.0;
~print_line false;
~print_line [104, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100]
```

tests/test-print-line.out

```
1
-2.
false
[104,101,108,108,111,32,119,111,114,108,100]
```

errors/boolBindPLUSint.ypl

```
let bool:cow=true in
```

errors/boolIndex.ypl

```
let bool[]:boolio = [true,false,false,false] in boolio[true]
```

errors/floatIndex.ypl

```
let float[]:floaty = [4.1, 3.0, 1.1] in floaty[9.1]
```

errors/intBindMINUSfloat.ypl

```
let int:cow=4 in cow - 3.0
```

errors/intBindPLUSfloat.ypl

```
int:cow=4 in cow + 3.0
```

errors/intORint.ypl

```
5 or 4
```

errors/intPLUSfloat.ypl

```
5 + .4
```

errors/notAnERror.ypl

```
4 + 4
```