YAPPL
Yet Another Probabilisitic Programming Language

David Hu
Jonathan Huggins
Hans Hyttinen
Harley McGrew

Columbia University

December, 2011

Overview: Introduction

Inspiration: functional, **probabilistic programming languages**

- **Church**: PPL based on pure subset of Scheme
- HANSEI: PPL based on Ocaml
- OCaml: inspiration for syntax

Church and HANSEI code can be difficult to read and understand

Overview: Probabilistic Programming

What is probabilistic programming about?

- allows for the concise definition of complex statistical models
- in particular, we are interested in defining generative **Bayesian models** and conditionally sampling from them
- to accomplish these goals, use **conditional evaluation** and **memoization**
- 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 lets you have "infinite" things (like lists or matrices), but only **lazily generate** items from the list

Overview: Goals

Improving on HANSEI and Church by...

- implementing a functional, natively probabilistic programming language with modern, Ocaml-like syntax
- build conditional evaluation and memoization directly into the language
- making syntax cleaner and more readable

Tutorial 1

tutorials/add.ypl

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

Tutorial 2

tutorials/geom_cond.ypl

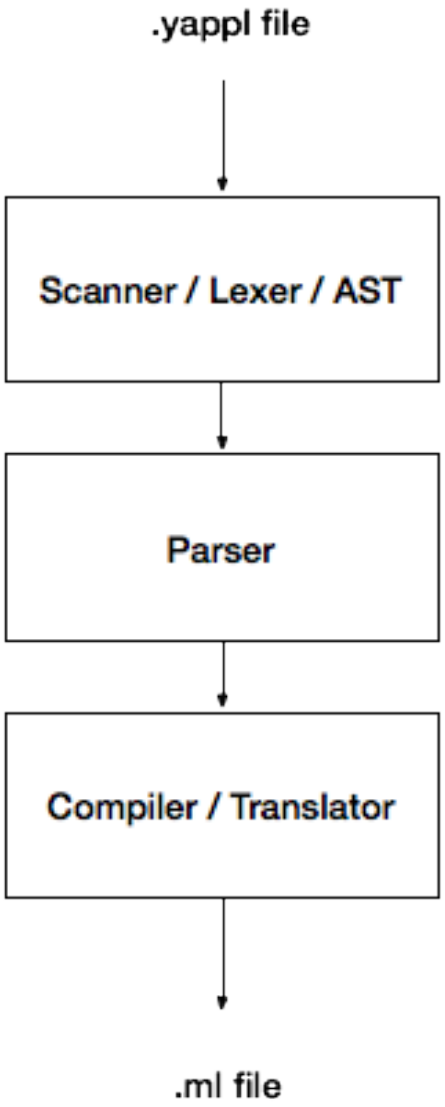
```
~seed;
fun int:geom float:q =
  fun int:geom_helper float:orig_q int:i =
    if ~rand < orig_q then i
    else ~geom_helper orig_q (i+1)
  in
    ~geom_helper q 1
in
```

Tutorial 3

```
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
```

Block Diagram



YAPPL code conversion to AST

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

YAPPL code conversion to AST

```
FuncBindings
    FuncBind =
        FuncDecl(t1, ValType(Int), Decl(a, ValType(Int))
        Binop +
            Id a
            IntLit 3
    Eval print_line
        Eval t1
            IntLit 2
            Noexpr
        Noexpr
```

Code Generation

Important steps

- Generate symbol table
  - tracks identifiers and type
  - can point to parent symbol table for scoping
- expr_to_string
  - main function for evaluation of ast
  - resolves reserved identifiers before using symtable
- Compile OCaml to executable
  - links with builtin (includes functions like rand)

Summary

- Yet Another Probabilisitic Programming Language, but
  - Cleaner syntax
  - Built-in constructs: memoization, conditionals
- `.ypl` → translation → `.ml` → execution
  - Condensed: `./yapplc program.ypl ; ./program`

Summary

Lessons learned

- Start early
- Parallelize work structure
- Project scope
  - Big: potential to do cool stuff
  - Small: it will probably actually work
- Unit testing
- Learn debug tools
  - OCAMLRUNPARAMS
  - ocamlyacc -v