# Homework 8

Due Thursday, Nov 8 by 10pm

## Turn-In Instructions

For text response questions, provide a LaTeX source file and pre-built PDF. For full credit, your LaTeX file should build properly. Be sure to `git add` all necessary files (e.g., images) if your LaTeX depends on it.

For each coding question in this assignment, create a project directory. You should be able to `cd` into this directory and then run the program by typing the command "`dotnet run`". Each program should be split into two pieces: a "`Program.fs`" file that contains the `main` method and associated program-startup routines (like argument parsers), and another "`Library.fs`" file that contains the function(s) of interest in the question. All library code should be in a module named "`CS334`". Be sure to provide usage output (defined in `main`) for all programs that require arguments. For full credit, your program should both build and run correctly.

Turn in your work using the Github repository assigned to you. The name of the Github repository will have the form `cs334hw8_<your user name>`. For example, my repository would be `cs334hw8_dbarowy`. You should have received an invite to commit in the repository in your email. If you did not receive this email, please contact me right away!

A good way to check that you submitted your assignment correctly is to `git clone` your repository to a new folder and then try building/running everything.

**Pair Programming and Honor Code:** You may optionally collaborate with one other person on the submission for this assignment. If you work with a partner, choose one of the two partner repositories for your submission. In the other repository, be sure to leave a `collaborators.txt` file that states who you worked with and the name of the repository where the code may be found. If you would like me to pair you with a partner, please let me know. You may not share code with any student who is not your designated partner.

This assignment is due on Thursday, Nov 8 by 10pm.

## Problems

### Q1. (*70 points*) ..... Project Proposal

For this assignment, you will propose your final project: a programming language of your own design. You are strongly encouraged to work with a partner on this assignment, however you will be permitted to work by yourself if you feel up to the extra challenge.

Many programming language specifications begin as informal proposals, and this is the template that we will follow in this class. With each stage of your project, you will revisit your document, making it clearer and more precise as you work. It will be a "living document." By the end of this class, your final specification should include a formal syntax and an informal, but precise, description of your language's semantics. It should clearly document your software artifact, which will be an interpreter for the language.

#### Structure of the Proposal

Version 1.0 of your specification should include the following sections.

(a) Introduction

**2+ paragraphs.** What problem does your language solve? Why does this problem need its own programming language?

(b) Design Principles

**1+ paragraphs.** Languages can solve problems in many ways. What are the guiding aesthetic or technical principles that underpin its design?

(c) Examples

**3+ examples.** Keeping in mind that your syntax is informal, sketch out 3 or more sample programs in your language.

(d) Language Concepts

**1+ paragraphs.** What are the core concepts a user needs to understand in order to write programs? Think in terms of both "primitives" and "combining forms." What are the key ideas and how are they combined?

(e) Syntax

**As much as is needed.** Sketch out the syntax of the language. For now, this can be an English description of the key syntactical elements and how they fit together. We will eventually transform this into a formal syntax section written in Backus-Naur Form.

(f) Semantics

**5+ paragraphs.** How is your program interpreted? This need not be formal yet, however, you should demonstrate that you've thought about how your program will be represented and evaluated on a computer. It should answer the following questions.

i. What are the primitive kinds of values in your system? For example, a primitive might be a number, a string, a shape, a sound, and so on.

ii. What are the "actions" or compositional elements of your language? In other words, how are values used? For example, your system might have "commands" like "move pen" or compositional elements like "sequence of notes."

iii. How is your program represented? If it helps you to think about this using ML algebraic data types, please use them. Otherwise, rough sketches such as class hierarchy drawings are fine.

iv. How do elements "fit together" to represent programs as abstract syntax? For the three example programs you gave earlier, provide sample abstract syntax trees.

v. How is your program evaluated? In particular,

   A. Do programs in your language read any input?

   B. What is the effect (output) of evaluating a program?

   C. Evaluation is usually conceived of as a depth-first, post-order traversal of an AST. Describe how such a traversal yields the effect you just described and *provide illustrations for clarity.* Demonstrate evaluation for at least one of your example programs.

## Goals

Your language need not be (and I discourage you from trying to build) a Turing-complete programming language. Instead, focus on solving a small class of problems. In other words, design a *domain-specific programming language.*

Your project should eventually achieve all of the following objectives:

(a) It should have a grammar capable of generating either an infinite or practically-infinite number of possible programs.

(b) It should have a parser that recognizes a grammatically-correct program, outputting the corresponding abstract syntax tree.

(c) It should have an evaluator capable of "running" any valid program.

(d) The language should do useful computational work.

If it is convenient to solve your problem by reducing to or extending a lambda calculus interpreter, you may propose such a solution. However, in most cases, it will likely be easier to design a purpose-built interpreter.

### Sample Specifications

The following documents, which are available on the course website, are useful to see how others structure their language specifications. Skim these as necessary in order to get a feel for your proposal. Note that Standard ML, Smalltalk, and Pascal are large, extensively developed general-purpose languages and therefore have long and very detailed specifications. Piet and Logo have short and imprecise specifications as they are both special-purpose languages. VoxPL has a short, precise specification as is common in CS research publications.

Your specification should be as long as is necessary and no longer. Writing good technical documentation is a real art. In general, try to be concise and precise, but not so concise or precise that you sacrifice understandability. The Logo documentation, especially the section titled "A Logo Primer" is an example of lucid, but informal, technical writing.

(a) "The Standard ML Core Language (Revised)" (Formal syntax; formal semantics)

(b) "PASCAL User Manual and Report" pp. 133-168 (Formal syntax; informal semantics)

(c) "Smalltalk-80: The Language and its Implementation" (A tour-de-force of both formal syntax/semantics and informal syntax/semantics... with lots of diagrams and artwork!)

(d) "VoxPL: Programming with the Wisdom of the Crowd" (Informal syntax; formal semantics)

(e) "Piet Programming Language"(Informal syntax; informal semantics)

(f) "What is Logo?" (Very informal, but beautiful, presentation of syntax and semantics)

## Q2. (*30 points*) ..... Alpha Normal Form

This question asks you to write F# code that performs alpha reduction on lambda calculus expressions.

Recall that alpha reduction is the process of renaming bound variables. Renaming variables simplifies the process of understanding a lambda expression, because the relationship between a variable binding site (a lambda abstraction) and its use (a lambda variable) can be understood without having to reason about nested scopes.

Your end goal is to write the following function, which reduces an arbitrary lambda expression to *alpha normal form*,

```
alphanorm : e:Expr -> b:Set<char> -> r:Map<char,char> -> Expr * Set<char>
```

where `e` is a lambda expression, `b` is a set of variable name bindings, `r` is a set of variable renamings (or, more precisely, a `Map` from one name to another), and the function returns (1) the alpha normalized expression and (2) an updated set of variable name bindings. In an alpha normalized expression, all bound and free variables in a lambda expression are unique.

For example, given the following lambda calculus expression written in our expression language,

```
(Lx.(Lx.y))
```

the `alphanorm` function returns the rewritten expression `(Lx.(La.y))` and the updated set of bindings `{ a, x, y }`.

The intuition behind `alphanorm` is that, with the exception of free variables (which should never be renamed), bound variables are renamed whenever conflicts are detected. The algorithm works from left to right. Therefore, the first `x` remains as is because it is the first `x` encountered. The second `x` is renamed because it conflicts with the first `x`. It is given the name `a`, because this is the next available

letter in the alphabet, starting from `a`, and `a` is not currently in use. The variable `y` is not renamed because it is a free variable.

These rules can a little subtle. For example, the expression `(Lx.(Lx.(La.x)))` should be alpha-normalized to `(Lx.(La.(Lb.a)))`. The second bound `x` needs to be renamed, and `a` is chosen because it is the "next available new variable." But the third bound variable, which is also an `a`, should be renamed to the "next available new variable," which is `b` because `a` was previously claimed by an earlier renaming.

We use use the same AST data types as in your parsing assignment,

```
type Expr =
| Variable of char
| Abstraction of char * Expr
| Application of Expr * Expr
```

therefore, you should allow the user to input lambda expressions as plain text, using a parser. You will be supplied with starter code for a complete lambda expression parser, but you may use your own code if you wish.

We will begin by decomposing this problem into a small set of helper functions in order to write the `alphanorm` function.

(a) Write the function

```
lambdaprint : e:Expr -> string
```

That prints an AST in exactly the same form in which a user supplies a lambda expression. For example, the result of the following function should always be `true`.

```
let check input =
  match parse input with
  | Some ast -> input = (lambdaprint ast)
  | None -> false
```

(b) Write the function

```
fv : e:Expr -> Set<char>
```

that returns the set of free variables in the expression `e`. The general strategy for finding free variables is to traverse the AST, keeping track of variables bound in lambda abstractions. If a variable is found which was not previously bound in a lambda abstraction, it must be free.

For this part of the assignment, you will be using the F# `Set<'a>` datatype. A *set* is a collection (a data structure) that stores *at most one* element with a given value, in no particular order. Remember that in a functional language, all values are immutable, therefore adding an element to a set using `Set.add` returns a *new set*. Other functions you will need to perform are `Set.empty`, `Set.contains`, and `Set.union`.

You can find documentation for `Set` on the F# collections reference page: `https://msdn.microsoft.com/en-us/visualfsharpdocs/conceptual/collections.set-module-%5bfsharp%5d`.

For example, `fv` run on the AST for the expression `(Lx.y)` returns the set `{ y }`.

(c) Write the function

```
freshvar : b:Set<char> -> char
```

that returns the next available unused variable given a set `b` of *used variable* bindings. This algorithm should choose the next variable name deterministically, and in alphabetical order.

For example, running `freshvar` with the set of used bindings `{ a, b, x, y, z }` should return the character `c`.

(d) Finally, write the function,

```
alphanorm : e:Expr -> b:Set<char> -> r:Map<char,char> -> Expr * Set<char>
```

where `e` is a lambda expression, `b` is a set of variable name bindings, `r` is a variable renaming map, and the function returns (1) the alpha normalized expression and (2) an updated set of variable name bindings.

The logic of the function should depend on the case of the `Expr` ADT.

- `Variable v`: If `r` contains a renaming for `v`, rename `v` and return a new `Variable`. Otherwise, return the original `Variable`.
- `Application(e1, e2)`: Alpha-normalize the subexpression `e1`, Alpha-normalize the subexpression `e2`, and then return a new `Application` using the alpha-normalized subexpressions.
- `Abstraction(v,e)`: If `b` contains a binding for `v`, obtain a fresh variable `v'`, add `v'` to `b`, add a renaming from `v` to `v'` to `r`, alpha-normalize the subexpression `e`, and return a new `Abstraction` using the renamed variable `v'` and the alpha-normalized subexpression. Otherwise, add `v` to `b`, alpha-normalize `e`, and return a new `Abstraction` using `v` and the alpha-normalized subexpression.

For each of these cases, you should also return the updated set of bindings, `b`. However, you should *not* return the updated set of renamings, `r`, as renamings should be *scoped* to a given subexpression.

In order for the algorithm to work correctly, `b` must be initialized with the set of free variables for the expression, and `r` must be initially an empty map.

For this part of the assignment, you will be using the F# `Map<'a,'b>` datatype. A `map` or `dictionary` is a collection of pairs. The first element of the pair, called the *key*, has type `'a` and serves as a *lookup value*. The second element of the pair, called the *value*, has type `'b` and stores arbitrary data. Therefore, a map is a *key-value store*, where values are obtained by indexing the data structure using their key. There can be at most one key-value pair for each distinct key.

You can find documentation for `Map` on the F# collections reference page: `https://msdn.microsoft.com/en-us/visualfsharpdocs/conceptual/collections.map-module-%5Bfsharp%5D`. Note that the `Map` datatype is an unrelated concept from the `map` higher-order function.

One useful check for a correct implementation is to call `alphanorm` using an alpha-normalized expression. In that case, the algorithm should simply return the same expression. Since F# uses structural equality, the following identity should hold:

```
a = fst (alphanorm a (fv a) Map.empty)
```

where `a` is an AST for an expression in alpha-normal form.

Here are some sample lambda expressions and their conversion to alpha-normal form.

| | |
|---|---|
| `(Lx.x)` | `(Lx.x)` |
| `(Lx.(Lx.x))` | `(Lx.(La.a))` |
| `(((Lx.x)(Lx.(xx)))(Lx.(xa)))` | `(((Lx.x)(Lb.(bb)))(Lc.(ca)))` |
| `(((Lx.x)(Ly.(yy)))(Lz.(za)))` | `(((Lx.x)(Ly.(yy)))(Lz.(za)))` |
| `(((Lx.(xx))(Lx.(yx)))z)` | `(((Lx.(xx))(La.(ya)))z)` |
| `((Lx.((Ly.(xy))y))x)` | `((La.((Lb.(ab))y))x)` |

(e) Be sure to document the above functions using XML comment blocks.

(f) The project directory for this question should be called "`q2`". You should be able to run your program on the command line by typing, for example, "`dotnet run "(Lx.(Lx.x))"`" and the program's output should be the alpha-normalized expression pretty-printed using `lambdaprint`.