Homework 4

Please submit this work by Wednesday, April 30 at 11:59pm.

Problem 1: Evaluation Semantics

TaPL problem 3.5.18.

Problem 2: Proofs about Programming Languages

TaPL problems 8.2.3 and 8.3.4.

Problem 3: The Untyped Lambda Calculus

Review Chapter 5 in *TaPL*. Then, in the untyped lambda calculus, give encodings of the following:

- the binary exclusive-or operator `xor`,
- a constructor for rational numbers (where both numerator and denominator are naturals), and
- a reciprocal operator `recip` on your rational number encoding.

You may use abbreviations for common terms as the text does (`tru` and `fls`, for example), but include definitions of any abbreviations in your presentation.

Problem 4: Type Systems

Here is a grammar for an expression language of characters and character strings, and some common operations on them:

```
t ::= nil        -- the empty string
    | c          -- a single character drawn from a finite alphabet
    | put(c,t)   -- attach the character c to the front of string t
    | get(t)     -- return the first character of the string t
    | cat(t,t)   -- string concatenation
    | del(c,t)   -- delete all occurrences of c from string t
    | rev(t)     -- reverse the string t
```

Design a type system for this language by both giving a grammar of types and a set of typing judgments assigning types to typable terms. What constitutes a typable term is suggested and nearly determined by the grammar (and especially the comments). Nevertheless, there are cases that typing judgments must guard against: for example, the argument to `rev` must be a string and not a character, both arguments to `cat` must be strings, etc.

Problem 5: Visualizing Typing Derivations

Chapter 8 of *TaPL* gives a type system for the language NB. We will provide a framework for lexing and parsing terms in NB in a directory`hw4/nb-typing` in your repository. In our front end code, we have added parentheses to the syntax, so that you can wrap any expression in parentheses for clarity. (Parentheses in NB are never necessary, but they are very helpful, for example, when typing expressions in a brace-matching editor.)

A typing derivation tree looks like this:

```
                                    0 : Nat

        true : Bool   0 : Nat    succ 0 : Nat

         if true then 0 else (succ 0) : Nat
```

Working within the framework we provide you, write a program to consume an NB program, and produce an "HTML picture" of the typing derivation. You may imitate my example (above) of how to draw a typing derivation in HTML (check the source of this page), but you do not have to; if you have a better idea, implement it. More specifically, if you have a stronger grasp on the various mechanisms of HTML and CSS than I do (which is entirely too possible), then by all means generate the output in some other way. The result should look like a typing derivation in *TaPL*, to a reasonable approximation.

Note that even if a typing derivation goes wrong, your program should produce an informative picture of the failed derivation. For example:

```
                                  true : Bool

        true : Bool   0 : Nat    succ true : X

         if true then 0 else (succ true) : X
```

Again, you have freedom as to how you draw a failed derivation, and how much of the failed derivation you display; my example is just one possibility. Even in the case of an ill-typed* term, the program must run to completion (and not, for example, terminate with an unhandled exception).

**To clarify: we're talking here about terms that are ill-typed but otherwise scannable and parseable. What your program does with strings that cannot be scanned or parsed is undefined.*