Homework 4

━━━ Turn-In Instructions ━━━

For this assignment, create one LaTeX file for the entire assignment called `hw4.tex`. Please also upload a PDF called `hw4.pdf`. You may use the LaTeX template for the assignment which is in your starter repository.

Turn in your work using the Github repository assigned to you. The name of the Github repository will have the form `cs334hw4_<your user name>`. For example, my repository would be `cs334hw4_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!

Honor code: You may collaborate with one or more people on this assignment, but you may not write code together. All submitted work must be your own original work. If you work with a partner, please submit a `collaborators.txt` file that includes their names.

This assignment is due on Thursday, October 4 by 10pm.

━━━ Reading ━━━

1. **(Required)** "Introduction to the Lambda Calculus," Parts 1 and 2

2. **(Required)** Concepts in Programming Languages, "Grammars and Parse Trees"

━━━ Problems ━━━

Q1. (*10 points*) .... Parse Tree

Draw the parse tree for the derivation of the expression "`10-15+12`" described in section 4.1.2 (bottom of page, Mitchell p. 53). Is there another derivation for "`10-15+12`"? Is there another parse tree?

Q2. (*10 points*) .... Lambda Calculus Reduction

Use lambda calculus reduction to find a shorter expression for $(\lambda x.\lambda y.xy)(\lambda x.xy)$. Begin by renaming bound variables. You should do all possible reductions to get the shortest possible expression. What goes wrong if you do not rename bound variables?

Q3. (*15 points*) .... Parsing and Precedence

Draw parse trees for the following expressions, assuming the grammar and precedence described in Example 4.2 (Mitchell, p. 56):

(a) `1 + 1 * 1`

(b) `1 + 1 - 1`.

(c) `1 - 1 + 1 - 1 * 1`, if `+` is given higher precedence than `-`.

Q4. (*30 points*) .... Symbolic Evaluation

The Python program fragment

```
def f(x):
  return x + 4

def g(y):
  return 3 - y

f(g(1))
```

can be written as the following lambda expression:

$$\left(\underbrace{(\lambda f.\lambda g.f\ (g\ 1))}_{\text{main}}\ \underbrace{(\lambda x.(+\ x\ 4))}_{f} \right)\ \underbrace{(\lambda y.(-\ 3\ y))}_{g}$$

Reduce the expression to a normal form in two different ways, as described below.

(a) (*5 points*) Reduce the expression by choosing, at each step, the reduction that eliminates a λ as far to the *left* as possible.

(b) (*5 points*) Reduce the expression by choosing, at each step, the reduction that eliminates a λ as far to the *right* as possible.

(c) (*5 points*) In pure λ-calculus, the order of evaluation of subexpressions does not affect the value of an expression. However, that is not the case for a language with side effects like Python or Java.

    i. Write a Python or Java *instance method* f and expressions e1 and e2 for which evaluating arguments left-to-right and right-to-left produces different results. (Hint: Recall that in Python/Java, an instance method may refer to variables declared outside of the scope of the function definition.)

    ii. What evaluation order is used by Java or Python? (you choose the language you prefer)

Q5. (*20 points*) .... Translation into Lambda Calculus

A programmer is having difficulty debugging the following Python program. In theory, on an "ideal" machine with infinite memory, this program would run forever. In practice, this program crashes because it runs out of memory, since extra space is required every time a function call is made.

```
def f(g):
    g(g)

def mymain():
  x = f(f)
  print x
```

Explain the behavior of the program by translating the definition of f into lambda calculus and then reducing the application f(f). Note that an equivalent program in a statically typed language like Java or ML would not compile.

Q6. (*10 points*) .... Bonus: Lambda Reduction with Sugar

Lambda expressions can be made easier to understand by the use of "syntactic sugar." Syntactic sugar is additional syntax that simplifies readability while leaving the meaning (semantics) of a language expression unchanged.

For example, here is a "sugared" lambda expression using some extra syntax known as a `let` declaration:

$$\text{let } foo = \lambda x.\lambda y.(+ \; x \; y) \text{ in}$$
$$foo \; 2 \; 3$$

The above expression may be "desugared" by replacing each `let` $z = U$ `in` V with $(\lambda z. V) \, U$. First, we identify z, U, and V:

$$
\begin{aligned}
z &= foo \\
U &= \lambda x.\lambda y.(+ \; x \; y) \\
V &= foo \; 2 \; 3
\end{aligned}
$$

which yields:

$$(\lambda foo.(foo \; 2 \; 3))(\lambda x.\lambda y.(+ \; x \; y)))$$

and after reducing this expression, the value 5.

(a) Desugar the following expression:

$$\text{let } compose = \lambda f. \lambda g. \lambda x. \, f(g \, x) \text{ in}$$
$$\text{let } h = \lambda x. \, (+ \; x \; x) \text{ in}$$
$$((compose \, h) \, h) \, 3$$

(b) Simplify the desugared lambda expression using reduction. Briefly explain why the simplified expression is the answer you expected.

Q7. (*0 points*) .... Optional Feedback

How hard was this assignment on a scale of 1 to 5? (where $1 =$ "very easy" and $5 =$ "very hard")

Do you have any additional comments or feedback that you would like me to know?

Please supply your answer as a `feedback.txt` file.

Q8. (*1 point*) .... Bonus

Does the reading "Introduction to the Lambda Calculus, Part 2" have any errors? One bonus point will be awarded for every verified problem that you find and report.

Submit as a text file called `errors.txt`.