Homework 3

Due Sunday, February 27 by 10:00pm

Handout 7
CSCI 334: Spring 2022

Turn-In Instructions

For this assignment, upload a PDF called `hw3.pdf` and a LaTeX file called `hw3.tex`. You must use the LaTeX starter template for the assignment, which is in your repository.

Turn in your work using the Github repository assigned to you. The name of the Github repository will have the form `cs334lab3_<your user name>`. For example, my repository would be `cs334lab3_dbarowy`.

Honor code: This assignment must be completed you, individually; you may not collaborate with another students on a solution. In other words, all submitted work must be your own original work. However, you are permitted work with a classmate for study-related activies, including: understanding the meaning of a question or learning how to perform techniques needed to complete this assignment. If you study with a partner, please submit a `collaborators.txt` file that includes their name.

This assignment is due on Sunday, February 27 by 10:00pm.

Reading

1. **(Required)** "Introduction to the Lambda Calculus," Parts 1 and 2 from the course packet.
2. **(Required)** "Grammars and Parse Trees" from the course packet.

Problems

Q1. (20 points) .... Parse Tree

Draw the parse tree for the derivation of the expression "`1 - 5 + 24`". Is there another derivation for "`1 - 5 + 24`"? If so, draw the other parse tree. Refer to the grammar on the bottom of page 77 of the course packet.

Q2. (40 points) .... Lambda Calculus Reduction

Use lambda calculus reductions 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. Your reduction should be in the two-column format shown in the course packet.

What goes wrong if you do not rename bound variables? Perform a second reduction (in two-column format) that shows what happens when you fail to rename.

Q3. (40 points) .... Parsing and Precedence

Draw parse trees for the following expressions, assuming the grammar and precedence described in Example 4.2 (course packet, pp. 81–82):

(a) `1 + 1 * 1`

(b) `1 + 1 - 1`

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

Q4. (points on Lab 4) .... 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. (points on Lab 4) .... 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)

f(f)
```

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. (5 points) .... Bonus: Lambda Reduction with Sugar

A common technique in programming language design is to provide extra syntax to make programs easier to understand. This extra syntax, called "syntactic sugar," improves readability but adds no real features to a language. In fact, an expression using syntactic sugar can be mechanically translated—or "desugared"—into another expression with the equivalent meaning (semantics).

To make this idea clear, we will add some sugar to the lambda calculus. For example, here is a lambda expression using a feature called a "`let` declaration":

$$\begin{array}{l} \texttt{let}\ \mathit{foo} = \lambda x.\lambda y.(+\ x\ y)\ \texttt{in} \\ \quad \mathit{foo}\ 2\ 3 \end{array}$$

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

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

which yields:

$$(\lambda \mathit{foo}.(\mathit{foo}\ 2\ 3))(\lambda x.\lambda y.(+\ x\ y)))$$

and after reducing this expression, the value 5.

(a) Desugar the following expression:

$$\begin{array}{l} \texttt{let}\ \mathit{compose} = \lambda f.\, \lambda g.\, \lambda x.\, f(g\, x)\ \texttt{in} \\ \quad \texttt{let}\ h = \lambda x.\, (+\ x\ x)\ \texttt{in} \\ \qquad ((\mathit{compose}\, h)\, h)\, 3 \end{array}$$

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

Q7. (1 point) .... Optional: Feedback

How hard was this assignment on a scale of 1 to 5? (where 1 is easy and 5 is difficult).

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

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