Homework 4

Due Sunday, March 7 by 10:00pm

Handout 8
CSCI 334: Spring 2022

Turn-In Instructions

Turn in your work using the Github repository assigned to you. The name of the Github repository will have the form `cs334lab4_<your user name>`. For example, my repository would be `cs334lab4_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

This is a partner lab. You may work with another classmate if you wish, and you may co-develop solutions. Remember: although you can work on code together, you must each independently write up and submit your solution. No code copying is allowed. Be sure to tell me who your partner is by committing a `collaborators.txt` file to your repository (2 points).

Lambda reductions should be typed using the LaTeX starter template in your repository.

Lisp files should have a "`.lisp`" suffix. For example, the pair programming problem **Q4b** should appear as the file `q4b.lisp`. Your code should be documented using comments. Comment lines start with a "`;`".

Honor code: You may collaborate with your partner on all aspects of this assignment, including discussing the problems, work on paper, and writing code. You may also work with students in the class other than your partner, but you may only discuss the problems at a high level (e.g., what the problems mean or general approaches to problems that do not involve code).

This assignment is due on Sunday, March 7 by 10:00pm.

Reading

1. **(As needed)** "Introduction to the Lambda Calculus," Parts 1 and 2 from the course packet.
2. **(Required)** "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I," from the course packet.
3. **(Required)** LISP Notes.

Problems

Q1. (10 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)

Q2. (10 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)

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.

Q3. (20 points) .... Church Numerals

Church encoding is a means of representing data and operators in the lambda calculus. The data and operators form a mathematical structure which is embedded in the lambda calculus. The Church numerals are a representation of the natural numbers using lambda notation. The method is named for Alonzo Church, who first encoded data in the lambda calculus this way.

The natural numbers are written using Church numerals as follows.

Number	Lambda Expression
0	$\lambda f.\lambda x.x$
1	$\lambda f.\lambda x.fx$
2	$\lambda f.\lambda x.f(fx)$
3	$\lambda f.\lambda x.f(f(fx))$
...	...
n	$\lambda f.\lambda x.f^n x$

Addition by one can be achieved using the successor function, defined as

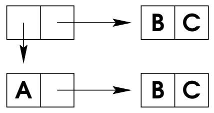
$$\text{succ} \equiv \lambda n.\lambda f.\lambda x.f(nfx)$$

Prove that $0 + 1 = 1$.

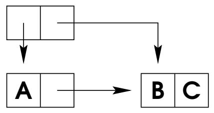
Q4. (18 points) .... Cons Cell Representations

(a) (4 points) Draw the list structure created by evaluating `(cons 'A (cons 'B 'C))`. Include this answer in your LaTeX file.

(b) (7 points) In a file called `q4b.lisp` write a pure Lisp expression that results in the representation to the right, with no sharing of the `(B . C)` cell. In the comments, explain why your expression produces this structure.



(c) (7 points) In a file called `q4c.lisp` write a pure Lisp expression that results in the representation to the right, with sharing of the `(B . C)` cell. In the comments, explain why your expression produces this structure.



While writing your expressions, use only these Lisp constructs: `lambda` abstraction, function application, the atoms `'A 'B 'C`, and the basic list functions (`cons, car, cdr, atom, eq`).

Q5. (20 points) .... Reverse

In a file called `q5.lisp` write a function `rev` that takes one argument. If the argument is an atom it remains unchanged. Otherwise, the function returns the elements of the list in reverse order:

```
* (rev nil)
nil

* (rev 'A)
A

* (rev '(A B C D))
```

```
(D C B A)

* (rev '(A (B C) D))
(D (B C) A)

* (rev '((A B) (C D)))
((C D) (A B))
```

Q6. (20 points) .... Recursive List Manipulation

In a file called `q6.lisp` write a function `merge-list` that takes two lists and joins them together into one large list by alternating elements from the original lists. If one list is longer, the extra part is appended onto the end of the merged list. The following examples demonstrate how to merge the lists together:

```
* (merge-list '(1 2 3) nil)
(1 2 3)

* (merge-list nil '(1 2 3))
(1 2 3)

* (merge-list '(1 2 3) '(A B C))
(1 A 2 B 3 C)

* (merge-list '(1 2) '(A B C D))
(1 A 2 B C D)

* (merge-list '((1 2) (3 4)) '(A B))
((1 2) A (3 4) B)
```

Before writing the function, you should start by identifying the base cases (there are more than one) and the recursive case.

Q7. (5 bonus points) .... Optional: Deep Reverse

In a file called `q7.lisp` write a function `deep-rev` that performs a "deep" reverse. Unlike `rev`, `deep-rev` not only reverses the elements in a list, but also deep-reverses every list inside that list.

```
* (deep-rev 'A)
A

* (deep-rev nil)
NIL

* (deep-rev '(A (B C) D))
(D (C B) A)

* (deep-rev '(1 2 ((3 4) 5)))
((5 (4 3)) 2 1)
```

Q8. (1 bonus 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.