Homework 4

Due Friday, Nov 8, 2019 at 8pm

Use the following commands to download and unpack the distribution code:

```
$ wget https://amirkamil.github.io/eecs490/homework/hw4/starter-files.tar.gz
$ tar xzf starter-files.tar.gz
```

You may work alone or with a partner. Please see the syllabus for partnership rules. As a reminder, you may not share any part of your solution outside of your partnership. This includes code, test cases, and written solutions.

1. *Lambda calculus.* For this question, you may use the symbol λ or the capital letter L to signify a λ.

   a) Evaluate the λ-calculus term below until it is in normal form. You must follow the standard rules for normal-order evaluation. Show each α-reduction and β-reduction step, as in the following:

   $$\begin{aligned} & (\lambda x.\ x)\ (\lambda x.\ x) & \\ \rightarrow\ & (\lambda x.\ x)\ (\lambda y.\ y) & (\alpha\text{-reduction}) \\ \rightarrow\ & \lambda y.\ y & (\beta\text{-reduction}) \end{aligned}$$

   In plain text:

   ```
   (L x. x) (L x. x)
   -> (L x. x) (L y. y)                  (alpha-reduction)
   -> L y. y                             (beta-reduction)
   ```

   Term to evaluate:

   $$(\lambda x.\ \lambda y.\ x\ \lambda x.\ y\ x)\ (\lambda w.\ w)\ (\lambda x.\ x\ x)\ a$$

   Hint: Evaluating this term requires a total of five β-reductions and one α-reduction.

   b) The following function maps a pair containing numbers (m,n) to a pair containing $(m+1,m)$:

   $$pairincr\ =\ \lambda p.\ pair\ (incr\ (first\ p))\ (first\ p)$$

   Applying it to *pair m n* produces:

   $$\begin{aligned} pairincr\ (pair\ m\ n)\ &=\ (\lambda p.\ pair\ (incr\ (first\ p))\ (first\ p))\ (pair\ m\ n) \\ &\rightarrow\ pair\ (incr\ (first\ (pair\ m\ n))\ (first\ (pair\ m\ n))) \\ &\rightarrow\ pair\ (incr\ m)\ (first\ (pair\ m\ n)) \\ &\rightarrow\ pair\ (incr\ m)\ m \end{aligned}$$

   Using *pairincr*, define a *decr* function that decrements a Church numeral:

   $$decr\ =\ \lambda n.\ \text{[fill in your solution]}$$

   Hint: What is the result when *pairincr* is applied to the pair $(0,0)$? What about when it is applied twice?

2. *Semantic equivalence.* Write a transition rule in big-step operational semantics that specifies the evaluation of a `let*` form in Scheme in terms of `let` and `let*`. As an example, the expression

   ```
   (let* ((v1 e1) (v2 e2) (v3 e3)) body)
   ```

   is equivalent to

   ```
   (let ((v1 e1)) (let* ((v2 e2) (v3 e3)) body))
   ```

Fill in the recursive rule below:

$$\frac{}{\langle(\textbf{let*}\ ((v_1\ e_1)\ \ldots\ (v_k\ e_k))\ body), \sigma\rangle \Downarrow} \quad \text{if } k > 1$$

Also fill in the rule for the base case:

$$\frac{}{\langle(\textbf{let*}\ ((v\ e))\ body), \sigma\rangle \Downarrow}$$

For this question and Q2, you may write subscripts with or without a preceding underscore (e.g. `v_1` or `v1`, and `e_k` or `ek`), and you may use the word `sigma` instead of σ.

3. *Scope.* Suppose we wanted to add the **let** construct to the simple imperative language defined in lecture, with the following syntax:

$$S \rightarrow \textbf{let}\ V\ =\ A\ \textbf{in}\ S\ \textbf{end}$$

The semantics of this construct are to execute the body S of the **let** in the context of a state in which the result of evaluating the given expression A is bound to the variable V. After the **let** is executed, the variable should be restored to its previous value. Fill in the big-step transition rule describing this behavior below:

$$\frac{}{\langle\textbf{let}\ v = a\ \textbf{in}\ s\ \textbf{end}, \sigma\rangle \Downarrow}$$

4. *Type systems and recursion.* The language that we used to explore type systems does not have a direct mechanism for defining a recursive function. Suppose we wanted to add a **letrec** construct, which is similar in structure to **let**:

$$E \rightarrow (\ \textbf{letrec}\ V\ :\ T\ =\ E\ \textbf{in}\ E\ )$$

A syntactic difference is that the variable must be explicitly typed in a **letrec**. Then if the initializer is a function abstraction, it is allowed to refer to itself by name in its body. For example, the following defines a factorial function:

$$\begin{array}{l}(\textbf{letrec}\ fact : Int \rightarrow Int = \\ \quad (\textbf{lambda}\ n : Int. \\ \quad\quad (\textbf{if}\ (n <= 0)\ \textbf{then}\ 1\ \textbf{else}\ (n * (fact\ (n-1)))) \\ \quad) \\ \quad \textbf{in}\ (fact\ 5) \\ )\end{array}$$

Fill in the following typing rule for the **letrec** construct:

$$\frac{}{\Gamma \vdash\ (\textbf{letrec}\ v : T_1 = t_1\ \textbf{in}\ t_2)\ :}$$

For this question, you may write subscripts with or without a preceding underscore (e.g. `t_1` or `t1`), and you may use the word `Gamma` and the symbols `|-` instead of Γ and $\vdash$.

5. *Vtables.* Consider the following C++ code:

```
struct A {
  void foo() {
    cout << "A::foo()" << endl;
  }
  virtual void bar() {
    cout << "A::bar()" << endl;
  }
};

struct B : A {
  virtual void foo() {
    cout << "B::foo()" << endl;
  }
```

```
  void bar() {
    cout << "B::bar()" << endl;
  }
};

int main() {
  A *aptr = new B;
  aptr->foo();
  aptr->bar();
}
```

This code prints the following when run:

```
A::foo()
B::bar()
```

a) Draw the vtables for `A` and `B`.

b) Briefly explain how the compiler translates the method calls in `main()`.

6. *Dispatch dictionaries and inheritance.* In the course notes, we saw a definition of a bank account ADT using functions and dispatch dictionaries. The following is a version of this ADT using built-in Python dictionaries:

```python
def account(initial_balance):
    def deposit(amount):
        new_balance = dispatch['balance'] + amount
        dispatch['balance'] = new_balance
        return new_balance

    def withdraw(amount):
        balance = dispatch['balance']
        if amount > balance:
            return 'Insufficient funds'
        balance -= amount
        dispatch['balance'] = balance
        return balance

    def get_balance():
        return dispatch['balance']

    dispatch = {}
    dispatch['balance'] = initial_balance
    dispatch['deposit'] = deposit
    dispatch['withdraw'] = withdraw
    dispatch['get_balance'] = get_balance

    def dispatch_message(message):
        return dispatch[message]

    return dispatch_message
```

Implement an ADT for a checking account that is a derived version of a bank account but charges a $1 fee for withdrawal. Fill in the ADT definition for `checking_account()` in the `hw4.py` file.

Do **not** repeat code from `account()`. Instead, implement a scheme for deferring to `account()` where possible.

Submission

Place your solution to question 6 in the provided `hw4.py` file. Write your answers to questions 1-5 in a PDF file named `hw4.pdf`. Submit `hw4.py` to the autograder before the deadline. Submit `hw4.pdf` to Gradescope before the deadline. Be sure to register your partnership on the autograder and Gradescope if you are working with a partner.