Homework 4

Due Tuesday Nov 14 at 8pm

1. *Semantic equivalence.* Write a transition rule in 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:

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

   Also fill in the rule for the base case:

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

   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.

2. *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 transition rule describing this behavior below:

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

3. *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 $\vdash$.

4. *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()
```

Explain why this is the result by drawing out the vtables for `A` and `B` and how the compiler translates the method calls in `main()`.

5. *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:

```
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.

6. *Java generics.*

   a) The file `Array.java` contains the definition of a multidimensional-array class that can hold elements of type `String`. Read through the provided code. Then make whatever modifications are necessary to make `Array` a generic class, so that it can hold elements of arbitrary class type.

      The `Array` class is built by storing elements in a single-dimensional built-in Java array. Variadic methods are used to allow the `Array` to have any valid *rank* (i.e. dimensionality), and the `indexOf()` method translates a multidimensional index to a single-dimensional location in the underlying Java array.

      The `main()` method of `ArrayTest` provides some tests, and the expected output from running the test is in `ArrayTest.expected`.

      In order to install Java on your machine, download the installer for your platform, selecting the Java SE Development Kit. Once it is installed and added to your path, you will be able to compile `ArrayTest.java` with:

      ```
      > javac ArrayTest.java
      ```

      This produces the `Array.class` and `ArrayTest.class` files, and you can then run the `main()` function of the `ArrayTest` class with:

      ```
      > java ArrayTest
      ```

      Make sure that you are not using OpenJDK or Java EE.

   b) Once you have a working generic `Array` class, implement the `indexOfMax1D()` method of the `Util` class in `Util.java`. This method takes in a generic `Array`, which must be rank one and have at least one element. It returns the index of the maximum element in the array.

      In order to be able to compare elements to each other, the element type must implement the appropriate `Comparable` interface. You will need to modify the header of `indexOfMax1D()` to enforce that this is the case.

      The `main()` method of the `UtilTest` class contains some tests. Compile and run as follows:

      ```
      > javac UtilTest.java
      > java -ea UtilTest
      ```

      The `-ea` flag tells Java to enable assertions, which are otherwise disabled by default.

Submission

Place your solution to question 5 in the provided `hw4.py` file, question 6a in `Array.java`, and question 6b in `Util.java`. Write your answers to questions 1-4 in a PDF file named `hw4.pdf`. Make sure to list any other students with whom you discussed the homework, as per course policy in the syllabus. Submit `hw4.py`, `Array.java`, and `Util.java` to the autograder before the deadline. Submit `hw4.pdf` to GradeScope before the deadline. **Pushing your work to GitHub does not submit it to the autograder!**