Useful functions.

Problem 1 assumes the following Python functions:

```python
def add(D, x, y):
    """Returns a new dictionary, d’, that is the same as D, except that
d’[X] == Y.""
    dp = D.copy()
    dp[x] = y
    return dp

def add_dict(D0, D1):
    """Returns a new dictionary, d, that is the same as D0 except that
d[x] == D1[x] for all keys x in D1.""
    d = D0.copy()
    d.update(D1)
    return d
```
1. [6 points] In the following grammar, each rule has two semantic actions, labeled A and B. The terminal symbol ‘ID’ has a string as its semantic value, and ‘NUM’ has an integer as its value. Other nodes have dictionaries as values. (For brevity, we’ll abbreviate $N.value()$ as $N$.)

**Answer:** There really should have been an A part for stmt, allowing one to write $\$$ = \{\}. Assume that is done.

```plaintext
program : region
stmt : "print" ID A: { $\$$ = {} } B: { if $\$$ID not in ENV:
                                    error()
                                    else: print(ENV[$ID]) } |
| "dcl" ID "=" NUM A: { $\$$ = { $ID : $NUM } } |
| "{" region "}" A: { $\$$ = {} } B: { } |

| stmts: /* Empty */ A: { $\$$ = {} } B: { } |
| stmts stmt A: { $\$$ = add_dict($stmts, $stmt);
                     if (len($stmt) == 1 && len($stmts))
                       error(); } B: { } |

| region: stmts A: { $\$$ = $stmts } B: { ENV = add(ENV, $\$$) } |
```

The scope rules for this minilanguage are mostly analogous to those of Project 2. Each declaration (dcl) statement declares the identifier on its left to have the value on its right throughout the innermost declarative region (region) enclosing the declaration, not
including nested regions that define the same identifier. (Unlike the project, the outer region is not handled differently.) A given identifier may be declared only once immediately within its declarative region.

Continues...
Thus, the following is supposed to print 4, 2, 5, and 1, and the commented-out statements would cause errors.

```plaintext
dcl x = 1
dcl y = 2
{
    print x
    print y
    print z
    dcl h = 3
    dcl x = 4
}
print x
dcl z = 5
# dcl x = 6
# print h
```

All A-actions are applied, then all B-actions. Fill in the actions so that the correct values are printed:

a. Fill in the A-actions so that the semantic value of each ‘region’ is a (Python-style) dictionary that maps identifiers defined immediately within that region to their declared values. Rather than using side-effects on dictionaries, use the operation `add(D, x, y)` defined on page 1. Other nodes may also have values, as you see fit.

b. Whereas the A-actions are applied in the usual post-order bottom-up fashion and refer to their children’s values, the B-actions are applied top-down and refer to their node’s value (which we’ve been calling $$ in projects), as previously set by the A-action. Each B-action is executed before those of its node’s children. ENV is a global variable to which you may assign dictionaries in the B-actions, as needed. When the children of the node are all processed, ENV is returned to its previous value. Again, rather than use side effects on dictionaries, use the operation `add_dict(D1, D2)` defined on page 1.
2. [2 points] Consider the following program:

```python
def g(x, y, p):
    def h():
        print(x)
        if y > 0:
            g(x+1, y-1, h)
        else:
            p()

x = 0
g(5, 1, None)
```

a. What does this print under normal Python scope rules?

**Answer:** 5

b. What would this print if Python were to use dynamic scoping?

**Answer:** 6

3. [1 point] Barnum Brown, Cosimo Alessandro Collini, Marjorie Courtenay-Latimer, John Estaugh Hopkins, Edward Lhuyd, Othniel Charles Marsh, and Richard Verstegen of Antwerp are all associated with a particular kind of discovery. But that of Ms. Courtenay-Latimer differed distinctly from that of the others in one important way. What was the difference?

**Answer:** They all made contributions to paleontology. Courtney-Latimer, however, discovered an ancient species that was not extinct (the Coelacanth).
4. [6 points] A certain language’s definition includes the following type rules. As in Lecture #12, the ‘x notation denotes a type variable (a new one for each invocation of the rule):

<table>
<thead>
<tr>
<th>Construct</th>
<th>Type</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Integer literal</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>2. hd (L)</td>
<td>‘a</td>
<td>L: ‘a list</td>
</tr>
<tr>
<td>3. tl (L)</td>
<td>‘a list</td>
<td>L: ‘a list</td>
</tr>
<tr>
<td>4. E₁-E₂</td>
<td>int</td>
<td>E₁: int, E₂: int</td>
</tr>
<tr>
<td>5. E₁ = E₂</td>
<td>bool</td>
<td>E₁: ‘a, E₂: ‘a</td>
</tr>
<tr>
<td>6. if E₁ then E₂ else E₃ fi</td>
<td>‘a</td>
<td>E₁: bool, E₂: ‘a, E₃: ‘a</td>
</tr>
</tbody>
</table>

Given these rules, consider the expression

\[
\text{if } x = 0 \text{ then } f(tl(L), x-1) \text{ else } hd(L) \text{ fi}
\]

If we denote the types of \(x\), \(f\), and \(L\), respectively, as type variables ‘x, ‘f, and ‘L, then:

a. After type inference, what are those variables bound to?

Answer:

\[
'x = \text{int} \\
'L = \text{list('a1)} \\
'f = \text{list('a1)} \rightarrow \text{int} \rightarrow 'a1
\]

where ‘a1 is a fresh type variable.

b. Show the steps needed to do this inference. Specifically, show which rules you use and which type equations get introduced as a result of the restrictions in the “Conditions” column of the table, and what all the type variables eventually get bound to (including the fresh ones introduced by applying the type rules from the table).

Answer: Numbering the rules for convenience, as shown:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
<th>Type</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>0</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>x = 0</td>
<td>bool</td>
<td>‘x = int</td>
</tr>
<tr>
<td>3.</td>
<td>tl(L)</td>
<td>list('a1)</td>
<td>‘L = list ‘a1</td>
</tr>
<tr>
<td>4.</td>
<td>x-1</td>
<td>int</td>
<td>‘x = int, int = int</td>
</tr>
<tr>
<td>7.</td>
<td>f(tl(L), x-1)</td>
<td>‘a4</td>
<td>‘f = ‘a2 ↦ ‘a3 ↦ ‘a4, list('a1) = ‘a2, int = ‘a3</td>
</tr>
<tr>
<td>2.</td>
<td>hd(L)</td>
<td>‘a5</td>
<td>‘L = list('a5)</td>
</tr>
<tr>
<td>6.</td>
<td>if · · · fi</td>
<td>‘a6</td>
<td>‘a4 = ‘a6, ‘a5 = ‘a6</td>
</tr>
</tbody>
</table>

As a result,

\[
'L = 'a2 = \text{list('a1)} \\
'x = 'a3 = \text{int} \\
'a4 = 'a6 = 'a5 = 'a1 \\
'f = \text{list('a1)} \rightarrow \text{int} \rightarrow 'a1
\]
5. [6 points] A *typeclass* resembles a Java interface; it is type variable coupled with a list of function signatures that restrict that type variable. For example, this declaration:

```
typeclass shape(s) { area: s->num, move: s->s }
```

is intended to mean that a type `s` is in the `shape` typeclass if there is a function `area` in the environment that takes an `s` and returns a `num` (some other type), and a function `move` that takes an `s` and returns an `s`. So, for example, in a given environment, the type `square` is in the `shape` typeclass if there are functions `area: square->num` and `move: square->square` in that environment.

For this problem, we’ll restrict ourselves to one-argument functions. We’ll represent environments in Prolog as usual: a list of items of the form `def(N, T)`. The `typeclass` declaration above, for example, would be represented in an environment like this:

```
def(shape, typeclass(s, [def(area, func(s, num)), def(move, func(s, s))]))
```

Just to have names, we’ll call `s` the *type parameter* of `shape`, and `[def(area, ...), ...]` the *class environment* of `shape`.

Complete the predicates below so that together they check whether a type belongs to a given typeclass. Assume that function overloading is possible, that there is no subtyping, and that there are no parameterized types. For example, it would be true that

```
inClass(square, shape, 
  [def(shape, 
    typeclass(s, [def(area, func(s, num)), def(move, func(s, s))]),
    def(move, func(square, square)),
    def(perimeter, func(square, num)),
    def(area, func(square, num))])
```

**Hint:** you may find `member(X, List)` and `not(X = Y)` useful.

*Problem continues on the next page.*
/** satisfiesClass(Type, TypeParam, ClassEnv, Env) if in environment Env,  
  * Type is a type that is in any typeclass whose type variable is  
  * TypeParam and whose class environment is ClassEnv. */

/** inClass(Type, TypeClass, Env) if in environment Env, TypeClass  
  * is the name of a typeclass and Type is the name of a type that is in  
  * typeclass TypeClass. */

Add rules for satisfiesClass and inClass here. Here’s one for free:

satisfiesClass(T, P, [], Env). /* Typeclass with no constraint  
  contains all types. */

Answer: [In retrospect, this problem was too hard.]  
inClass(T, TC, Env) :-  
  member(def(TC, typeclass(P, CE)), Env),  
  satisfiesClass(T, P, CE, Env).

satisfiesClass(T, P, [Def|R], Env) :-  
  findMatch(T, P, Def, Env), satisfiesClass(T, P, R, Env).

findMatch(T, P, def(F, func(P, C)), Env) :-  
  not(P=C), member(def(F, func(T, C)), Env).
findMatch(T, P, def(F, func(D, P)), Env) :-  
  not(P=D), member(def(F, func(D, T)), Env).
findMatch(T, P, F, func(D, C), Env) :-  
  not(P=D), not(P=C), member(def(F, func(D, C)), Env).
Login: