1. [5 points]

a. Even though the ~~Pyth~~ Python language assigns a static type to every variable and expression, Pyth can't properly be called statically typed, since checks are necessary at runtime to insure that certain statements obey the type rules. Give an example showing why such run-time checks are unavoidable.

```
class C(object): pass

x = C()
x.f()
```

← x has type any, so this is okay method f does not exist though

b. Pyuce is a very stripped-down dialect of Pyth that has no user-defined classes and only the built-in types Bool, Float, Int, and String (no type Any or Void, in particular). Furthermore, it enforces the following rules:

1. Every function or method definition must be given a type (e.g., `f: (Int)->Int`).
2. Every variable (instance variable, local variable) has the type of the first value assigned to it.
3. The dynamic type of any value assigned to a variable or passed as a parameter must have that variable's or parameter's type. So the following is illegal:

```
x = 0
x = "Hello"
```

4. Every function (including the main program) must start with assignment statements to all its local variables. Thus, the following program is illegal (because y is not first assigned to until after a non-assignment statement):

```
x = 2
g(x)
y = 3
```

5. The operators and and or yield True or False (as opposed to Pyth and Scheme, where they yield an operand).

Pyuce compilers never need to generate run-time type checks. They can either tell that a statement will never cause a type error, or that it always will (and therefore can be compiled into a call to some error routine). Give an example of a Pyth program (in the Pyuce subset) that requires a run-time type check, but for which a Pyuce compiler could detect statically the statement that would cause a type error.

```
x = "hello"
y = x+1
```

c. Show (with an example) why rule 4 in part b above is necessary (if Puce compilers are to be able to eliminate all run-time type checks).

```
def f():
    print x
    x = 1
```

With no runtime checks, you need to ensure statically that every variable is defined before use.

d. Show why rule 5 is necessary.

```
x :: Bool = ...
y = "hi"
z = x or y
```

Without rule 5, there would be no way to tell statically if z is a bool or a string. So the operation z[0] might be valid or not, depending on x.

2. [5 points] This problem involves coming up with new typing rules that is, new rules concerning the predicate `typeof(`E, T, Env`)`. You may assume that we already have rules for the rest of the language, and that there are rules for predicate `subtype(`T_0`, `T_1`)` that make it mean "T_0 is a subtype of T_1."

I wish to describe type rules for a map (dictionary) type. A map in this language takes keys of some particular type (call it the *domain type*) and produces values of some other type (call it the *codomain type*). So `m[k]`, where `m` has a map type, produces a value of its codomain type (or some subtype of it) as long as `k` has the appropriate domain type (or some subtype of it). We'll use the notation `map(`D, C`)` to mean "the type of maps with domain type D and codomain type C."

a. Give appropriate rules for the type of `m[k]`, that is, appropriate Prologish rules for `typeof(index(m,k), T, E)` (where `index(m,k)` is the AST for `m[k]`).

```
typeof(index(M,K), T, E) :- typeof(M, map(D, T), E),
                            typeof(K, T1, E),
                            subtype(T1, D).
```

We can omit this predicate and use D instead of T1 if we have a rule "typeof(E, T0, Env) :- subtype(T1, T0), typeof(E, T1, Env)" as in HW6.

b. Give appropriate rules for assignment to `m[k]`. Assignment always produces a Void value, so the problem is to give correctness rules for `typeof(assign(m,k,v), Void, E)`, where `assign(m,k,v)` is the AST for `m[k]=v`.

```
typeof(assign(M,K,V), void, E) :-
    typeof(M, map(D, C), E),
    typeof(K, T1, E),
    typeof(V, T2, E),
    subtype(T1, D),
    subtype(T2, C).
```

} ditto

3. [1 point] Name a Berkeley professor who contributed a key algorithm used in correcting errors in transmitted data.

Elwyn Berlekamp

4. [5 points] The following exercises involve scoping rules for variables (including parameters). Do not consider type declarations or the names of functions or constants introduced by **def**.

a. Do we need multiple passes over the AST to bind names to declarations of ~~Pyth~~ Python variables? If so, give an example showing why. If not, briefly say why not.

Yes. For example:

```
x = 3
def f(y):
    if y:
        return x
    x = 9
    return x
```

When we see x in the body of the "if" statement, we haven't seen the assignment that declares the local x inside f. We need a first pass to see if there are declarations for local variables, then a second pass to perform the binding.

b. How would your answer to part (a) above change if ~~Pyth~~ Python had no assignment statements?

It depends. If we count "for" statements as assignment statements because they perform assignments, then they are also disallowed and so the only way to introduce variable names is as function parameters. In this case, variable names always precede their uses, and so a single pass is sufficient for binding.

If "for" statements are allowed, we can construct an equivalent example that requires multiple passes.

c. Suppose ~~Pyth~~ Python used dynamic scoping for variables. How would this change the compiler's processing of declarations?

All variables would effectively be global, except that their types could change. There would be no need for binding at compile time; instead all references to variables would have to be resolved at run time (e.g., by checking a stack of declarations associated with the name).

d. Consider now Java's rules for variables (not including static or instance variables). What would be your answer to part (a) for Java?

A single pass is sufficient for binding, because Java requires that all variables are declared before they are used.

5. [5 points] Consider the following class definitions:

```
class A(Object):
    x = 3
    def f (self, a):
        Sf1
    ~~class~~ def p (x):
        Sp1
    def g (self):
        Sg1

class B(A):
    y = 7
    def g (self):
        Sg2

class C(B):
    def f (self, a):
        Sf2
    def h (self):
        Sh1

aC: C
aC = C ()
```

Diagram the object pointed to by aC, showing its contents and any other runtime data structures used to make the object-oriented features of ~~Pyth~~ Python work.