The goal of this homework assignment is to write a type checker for MyPL. You will need to create a new class TypeChecker in the file mypl_type_checker.py. Your TypeChecker must implement the methods defined in the Visitor class from mypl_ast.py and you must use the SymbolTable class to implement your type checker (the symbol table code is given below). You will also need to modify your main program to run the type checker:

```python
...  
stmt_list = the_parser.parse()  
checker = type_checker.TypeChecker()  
stmt_list.accept(checker)
```

Like in HW 3, your program should only produce output if there is a type error in the MyPL program. As a simple example of what your program should do, given the MyPL program test.txt:

```plaintext
x = "foo";
y = 5 + 4;
z = y + x;
```

your program should output the following error.

```
$ python hw5.py test.txt  
error: expecting INT found STRING at line 3 column 9
```

This homework is trickier than the previous ones. You should make sure to give yourself enough time to ensure you finish the assignment. Because this homework is a bit harder:

1. To receive full points, you only need to perform type checking for non-list types. In addition to finding type errors, your program must also find variable "used before defined" errors.

2. If you are able to finish (1) and would like an additional challenge, you can receive extra credit by implementing type checking for list types. (This is really worth doing to go deeper into some of the complexities of type checking.)

An informal list of the typing rules we’ll use for MyPL are given below.

**Typing Rules (informal).** In addition to checking for "use before def" errors, the type checker must employ the following rules related to types.

1. When a variable first appears on the left-hand side (LHS) of an assignment it is considered declared/defined, and is assigned the type on the right-hand side (RHS) of the assignment. (Note that this rule does not include the case when index expressions are used on the LHS of an assignment; for more information on index expressions, see below). This rule implies the following program results in a type error: “x = 4; x = "a";”.

2. MyPL employs static scoping. A variable is only valid in the environment (block) it is declared in, or sub environments (sub blocks) of this environment. A block is defined by a statement list, namely, the entire program forms a block; the body of an if, elseif, or else forms a block; and the body of a while statement forms a block.
3. Lists are homogeneous, i.e., they can only contain values of the same type. So, e.g., [5,"hi"] results in a type error. The empty list [] is considered an untyped list, meaning it can take on any type.

4. The type of a list is written “[τ]” for a type τ. For example, the list [10,20,30] has type “[‘INT’]” and the list ["a","b"] has type “[‘STRING’]”. Lists of lists are allowed (if all sublists have the same type), so the lists [[0,1],['a']] and [[0,1],[[10],[20]]] result in type errors, whereas the lists [[0,1],[0]] and [[0,1],[]] are type safe with both having the type “[‘INT’]”.

5. Indexed expressions (e.g., xs[0]) on the LHS of an assignment require the variable name to be declared/defined prior to the assignment. The variable (e.g., xs) must be a list type. If the variable’s type is [τ], the type on the RHS of the assignment must be τ.

6. If an indexed expression (e.g., xs[0]) over a variable of type [τ] is used within an expression, the indexed expression has type τ.

7. Both operands of comparison and mathematical operators must be of the same type. So, e.g., "hi" + 5 and 5 <= true result in type errors. Integers can be added, subtracted, multiplied, divided, and used in a modulus operator. Strings can be added (which results in string concatenation, e.g., "foo" + "bar" evaluates to "foobar"). No mathematical operations are permitted over Boolean values. The operands of the “and” and “or” operators must be Boolean values.

8. Lists can be concatenated using the + operator. For example, the expression [4] + [5] results in a new list with the value [4,5]. Concatenation requires that both operands are of the same type. Note that [4] + [] is a valid expression (the result of which, in this case, is “[‘INT’]”). No other mathematical operators can be used with lists.

On the due date, hand in a cover sheet together with hard copy of your implementation (all source code files), hard copy of your test program, print outs of tests showing your parser works properly, and a design document. In addition, submit all files required to run your program as a single zip file to the online dropoff site (https://www.cs.gonzaga.edu/dropoff/). Note that in your hard copy, your tests must include both the input files and the output produced after running your program over the files.
class SymbolTable(object):
    """A symbol table consists of a stack of environments, where each
    environment maps a variable name to the variable’s type and value.
    """
    def __init__(self):
        self.scopes = [] # list of {var_name:{'type':t, 'value':v}}

    def __environment(self, var_name):
        # search from last (most recent) to first environment
        for i in range(len(self.scopes)-1, -1, -1):
            if var_name in self.scopes[i]:
                return self.scopes[i]

    def variable_exists(self, var_name):
        return self.__environment(var_name) != None

    def add_variable(self, var_name):
        # can’t add if no environments
        if len(self.scopes) == 0:
            return
        # add to the most recently added environment
        self.scopes[-1][var_name] = {'type':None, 'value':None}

    def get_variable_type(self, var_name):
        env = self.__environment(var_name)
        if env != None:
            return env[var_name]['type']

    def set_variable_type(self, var_name, var_type):
        env = self.__environment(var_name)
        if env != None:
            env[var_name]['type'] = var_type

    def set_variable_value(self, var_name, var_value):
        env = self.__environment(var_name)
        if env != None:
            env[var_name]['value'] = var_value

    def get_variable_value(self, var_name):
        env = self.__environment(var_name)
        if env != None:
            return env[var_name]['value']

    def push_environment(self):
        self.scopes.append({})

    def pop_environment(self):
        if len(self.scopes) > 0:
            self.scopes.pop()

    def __str__(self):
        return str(self.scopes)