Written Homework.

Read the following in the textbook and then answer the questions below.

- Ch. 4: 4.3

(1). Give concrete examples in MyPL demonstrating each of the three “static” scope rules described in Definition 4.4 of Section 4.3.

(2). In your own words, explain how “dynamic” scoping differs from “static” scoping.

(3). Give an example in MyPL of a function that returns different values depending on whether static or dynamic scoping rules are used. Explain the differences and generally how they work using your example.

(4). Repeat question (3) but for structs.

Programming Homework: Syntax Analysis.

The two goals of this assignment are to: (a) modify your parser implementation to return an Abstract Syntax Tree (AST) object structure representing the program being analyzed; and (b) to take the AST object structure returned from a parse and navigate it (using the Visitor pattern) to output a “pretty print” version of the corresponding source code. You must use the AST class design provided below (shown graphically as well as the class definitions in mypl_ast.py). When writing your “pretty printer”, you must use the following MyPL code styling rules (also see the test cases provided separately).

1. Indent all statements within a block by four spaces.

2. Each statement should be on a separate line without blank lines before or after the statement. The exceptions to this rule are struct and function declarations.

3. Format variable declarations without explicit types using one space between var and the identifier, and one space before and after the equals sign, e.g., var id = expr;. There also should not be a space between the end of the expression and the ending semicolon.

4. Format variable declarations with explicit types the same as for implicit types, but with one space after the colon symbol used to designate the type (but not before the colon), e.g., var id: type = expr;. There should be one space after the type before the equals sign.

5. Format variable assignments with one space between set and the identifier, and one space before and after the equals sign, e.g., set id = expr;.

6. Format struct declarations such that struct and the type name appears on one line, with one space between the two, each variable declaration indented (four spaces from the start of struct) and on a separate line (with no blanks between), and end on the next immediate line after the last variable declaration and aligned with struct. There should be one blank line before and after the struct declaration. Here is an example:
struct Person
  var age = 0;
  var name = "";
  var mother: Person = nil;
  var father: Person = nil;
end

7. Format function declarations such that `fun`, the return type, the function name, and the parameter list are all on the same line, the body of the function is indented appropriately, and the `end` is on a separate line, immediately following the last body statement, aligned with the `fun` reserved word. There should be one space between `fun` and the type, one space between the type and the function name, no spaces between the name and the open parenthesis, parameters of the form `name: type`, each of which is separated by a comma followed by one space before the next parameter name. There should be one blank line before and after each function declaration. Here is an example:

```c
fun int add(x: int, y: int)
  set sum = x + y;
  return sum;
end
```

8. Format while statements such that `while` and `do` occur on the same line, there is one space between the start and end of the Boolean expression, the body of the while loop is appropriately indented, and `end` is aligned with `while` and occurs on the line immediately after the last statement of the body.

9. Format if-elif-else statements similar to while statements such that the body of each section is indented, `elif` and `else` statements appear on separate lines (with no blank lines before or after), `then` appears on the same line as its corresponding `if` or `elif`, and `end` statements appear on separate lines with no preceding blank lines.

10. Format simple expressions without any extra spaces. Path expressions should not contain spaces between corresponding dots (e.g., `x.y.z`).

11. Format complex expressions with spaces between their corresponding parts and fully enclose them in parentheses (regardless of whether there were parentheses in the original source code). For example, if the original was written as `3+4+5` the pretty-printed version should be written as `(3 + (4 + 5))`.

12. Boolean expressions should follow the same rules as for complex expressions. For example, `not x>1 and y>1` should print as `not ((x > 1) and (y > 1))`.

13. Format struct object creation such that there is one space between `new` and the struct type name, but no space between the type name and the closing semicolon.

14. Format function calls such that the function name is immediately followed by an opening parenthesis, followed by a comma-separated list of expressions, followed by a closing parenthesis. There should be one space after each comma.
On the due date, **hand in** a cover sheet together with hard copy of your implementation (all new or edited source code files), print outs of tests showing your program works properly, a write up of your testing strategies and implementation issues, and your test cases (test files). In addition, submit your program files and test cases to the online dropoff site ([https://www.cs.gonzaga.edu/dropoff/](https://www.cs.gonzaga.edu/dropoff/)). Note that the hard-copy print out of your tests must include both the input files and the output produced after running your pretty printer over the files. For this assignment you do not need syntactically incorrect tests.

As part of your implementation you must use the code provided (see below) as initial class/program design. Also be sure to add comments to your code! Within your classes, you can define any helper functions you see fit.
import mypl_token as token

class ASTNode(object):
    """The base class for the abstract syntax tree."""
    def accept(self, visitor): pass

class Stmt(ASTNode):
    """The base class for all statement nodes."""
    def accept(self, visitor): pass

class StmtList(ASTNode):
    """A statement list consists of a list of statements."""
    def __init__(self):
        self.stmts = [] # list of Stmt
    def accept(self, visitor):
        visitor.visit_stmt_list(self)

class Expr(ASTNode):
    """The base class for all expression nodes."""
    def accept(self, visitor): pass

class ExprStmt(Stmt):
    """A simple statement that is just an expression.""
    def __init__(self):
        self.expr = None # Expr node
    def accept(self, visitor):
        visitor.visit_expr_stmt(self)

class VarDeclStmt(Stmt):
    """A variable declaration statement consists of a variable identifier,
    an (optional) type, and an initial value.
    ""
    def __init__(self):
        self.var_id = None # Token (ID)
        self.var_type = None # Token (STRINGTYPE, ..., ID)
        self.var_expr = None # Expr node
    def accept(self, visitor):
        visitor.visit_var_decl_stmt(self)

class AssignStmt(Stmt):
    """An assignment statement consists of an identifier and an expression.
    ""
    def __init__(self):
        self.lhs = None # LValue node
        self.rhs = None # Expr node
    def accept(self, visitor):
        visitor.visit_assign_stmt(self)

class StructDeclStmt(Stmt):
    """A struct declaration statement consists of an identifier, and a
    list of variable declarations.
    ""
    def __init__(self):
self.struct_id = None  # Token (id)
self.var_decls = []  # [VarDeclStmt]
def accept(self, visitor):
    visitor.visit_struct_decl_stmt(self)

class FunDeclStmt(Stmt):
    """A function declaration statement consists of an identifier, a list of
parameters (identifiers with types), a return type, and a list of
function body statements.
"""
def __init__(self):
    self.fun_name = None  # Token (id)
    self.params = []  # List of FunParam
    self.return_type = None  # Token
    self.stmt_list = StmtList()  # StmtList
def accept(self, visitor):
    visitor.visit_fun_decl_stmt(self)

class ReturnStmt(Stmt):
    """A return statement consist of a return expression and the corresponding return token (for printing line and column numbers).
"""
def __init__(self):
    self.return_expr = None  # Expr
    self.return_token = None  # to keep track of location (e.g., return;)
def accept(self, visitor):
    visitor.visit_return_stmt(self)

class WhileStmt(Stmt):
    """A while statement consists of a condition (Boolean expression) and a statement list (the body of the while).
"""
def __init__(self):
    self.bool_expr = None  # a BoolExpr node
    self.stmt_list = StmtList()
def accept(self, visitor):
    visitor.visit_while_stmt(self)

class IfStmt(Stmt):
    """An if stmt consists of a basic if part, a (possibly empty) list of else ifs, and an optional else part (represented as a statement list).
"""
def __init__(self):
    self.if_part = BasicIf()
    self.elseifs = []  # list of BasicIf
    self.has_else = False
    self.else_stmts = StmtList()
def accept(self, visitor):
    visitor.visit_if_stmt(self)

class SimpleExpr(Expr):
    """A simple expression consists of an RValue.
"""
def __init__(self):
    self.term = None  # RValue

def accept(self, visitor):
    visitor.visit_simple_expr(self)

class ComplexExpr(Expr):
    """A complex expression consist of an expression, followed by a
    mathematical operator (+, -, *, etc.), followed by another
    (possibly complex) expression.
    """
    def __init__(self):
        self.first_operand = None  # Expr node
        self.math_rel = None  # Token (+, -, *, etc.)
        self.rest = None  # Expr node
    def accept(self, visitor):
        visitor.visit_complex_expr(self)

class BoolExpr(ASTNode):
    """A boolean expression consists of an expression, a Boolean relation
    (==, <=, !=, etc.), another expression, and possibly an 'and' or
    'or' followed by additional boolean expressions. An entire boolean
    expression can also be negated. Note that only the first_expr is
    required.
    """
    def __init__(self):
        self.first_expr = None  # Expr node
        self.bool_rel = None  # Token (==, <=, !=, etc.)
        self.second_expr = None  # Expr node
        self.bool_connector = None  # Token (AND or OR)
        self.rest = None  # BoolExpr node
        self.negated = False  # Bool
    def accept(self, visitor):
        visitor.visit_bool_expr(self)

class LValue(ASTNode):
    """A lvalue consist of a simple id or a path expression.
    """
    def __init__(self):
        self.path = []  # [Token (ID)] ... one implies simple var
    def accept(self, visitor):
        visitor.visit_lvalue(self)

class FunParam(Stmt):
    """A function declaration parameter consists of a variable name (id)
    and a type."""
    def __init__(self):
        self.param_name = None  # Token (id)
        self.param_type = None  # Token (id)
    def accept(self, visitor):
        visitor.visit_fun_param(self)

class BasicIf(object):
    """A basic if holds a condition (Boolean expression) and a list of
    statements (the body of the if)."""
def __init__(self):
    self.bool_expr = None  # BoolExpr node
    self.stmt_list = StmtList()

class RValue(ASTNode):
    """The base class for rvalue nodes."""
    def accept(self, visitor): pass

class SimpleRValue(RValue):
    """A simple rvalue consists of a single primitive value."
    def __init__(self):
        self.val = None  # Token
    def accept(self, visitor):
        visitor.visit_simple_rvalue(self)

class NewRValue(RValue):
    """A new rvalue consists of a struct name (id)"
    def __init__(self):
        self.struct_type = None  # Token (id)
    def accept(self, visitor):
        visitor.visit_new_rvalue(self)

class CallRValue(RValue):
    """A function call rvalue consists of a function name (id) and a list of arguments (expressions)"
    def __init__(self):
        self.fun = None  # Token (id)
        self.args = []  # list of Expr
    def accept(self, visitor):
        visitor.visit_call_rvalue(self)

class IDRvalue(RValue):
    """An identifier rvalue consists of a path of one or more identifiers."
    def __init__(self):
        self.path = []  # List of Token (id)
    def accept(self, visitor):
        visitor.visit_id_rvalue(self)

class Visitor(object):
    """The base class for AST visitors."
    def visit_stmt_list(self, stmt_list): pass
    def visit_expr_stmt(self, expr_stmt): pass
    def visit_var_decl_stmt(self, var_decl): pass
    def visit_assign_stmt(self, assign_stmt): pass
    def visit_struct_decl_stmt(self, struct_decl): pass
    def visit_fun_decl_stmt(self, fun_decl): pass
    def visit_return_stmt(self, return_stmt): pass
    def visit_while_stmt(self, while_stmt): pass

def visit_if_stmt(self, if_stmt): pass
def visit_simple_expr(self, simple_expr): pass
def visit_complex_expr(self, complex_expr): pass
def visit_bool_expr(self, bool_expr): pass
def visit_lvalue(self, lval): pass
def visit_fun_param(self, fun_param): pass
def visit_simple_rvalue(self, simple_rvalue): pass
def visit_new_rvalue(self, new_rvalue): pass
def visit_call_rvalue(self, call_rvalue): pass
def visit_id_rvalue(self, id_rvalue): pass
import mypl_token as token
import mypl_ast as ast

class PrintVisitor(ast.Visitor):
    """An AST pretty printer""
    def __init__(self, output_stream):
        self.indent = 0  # to increase/decrease indent level
        self.output_stream = output_stream  # where printing to

def __indent(self):
    """Get default indent of four spaces""
    return ' ' * self.indent

def __write(self, msg):
    self.output_stream.write(msg)

def visit_stmt_list(self, stmt_list):
    for stmt in stmt_list.stmts:
        stmt.accept(self)

def visit_expr_stmt(self, expr_stmt):
    self.__write(self.__indent())
    expr_stmt.expr.accept(self)
    self.__write(';
')

def visit_expr_list(self, expr_list):
    for expr in expr_list.exps:
        expr.accept(self)
        self.__write(' ,
')

def visit_expr(self, expr):
    if isinstance(expr, token.InfixOp):
        self.__write(expr.op)
        expr.left.accept(self)
        expr.right.accept(self)
        self.__write(')
    else:
        expr.accept(self)
        self.__write(')

... etc. ...
import mypl_error as error
import mypl_lexer as lexer
import mypl_token as token
import mypl_parser as parser
import mypl_ast as ast
import mypl_print_visitor as ast_printer
import sys

def main(filename):
    try:
        file_stream = open(filename, 'r')
        hw4(file_stream)
        file_stream.close()
    except FileNotFoundError:
        sys.exit('invalid filename %s' % filename)
    except error.MyPLError as e:
        file_stream.close()
        sys.exit(e)

def hw4(file_stream):
    the_lexer = lexer.Lexer(file_stream)
    the_parser = parser.Parser(the_lexer)
    stmt_list = the_parser.parse()
    print_visitor = ast_printer.PrintVisitor(sys.stdout)
    stmt_list.accept(print_visitor)

if __name__ == '__main__':
    if len(sys.argv) != 2:
        sys.exit('Usage: %s file' % sys.argv[0])
    main(sys.argv[1])