Written Homework.

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

- Ch. 2: 2.5
- Ch. 5: 5.1, 5.2

1. Briefly describe the difference between "operational semantics" and "pragmatics".

2. In your own words and in one sentence, describe the challenge placed on interpreters when it comes to languages that allow recursion.

3. Define (operational semantics) transitions for the following three constructs in MyPL.
   - Modulus operation \( n \% m \)
   - Function calls (of the form \( f(n_1, n_2, \ldots, n_m) \))
   - If statements (which differ from the books in that we allow \texttt{elif} clauses).

4. Consider the following command in the mini language of Section 2.5.

   \[
   \text{if } \neg (Y == Z) \text{ then } X := (X + (Y - Z)) \text{ else } X := 0
   \]

   Use the operational semantics given in 2.5 and the initial state \( \sigma = [(X, 1), (Y, 2), (Z, 1)] \) to compute the result of the command.


The goal of this assignment is to implement a static “type checker” for MyPL that checks for type errors, use before def errors, and function call errors. Your job is to implement a new visitor class that performs type checking over the AST generated from your parser. Your implementation should follow the typing rules for MyPL given below. The following list describes “hints” and things to be aware of as you develop your type checker.

1. You should implement your type checker incrementally, starting with simple and complex expressions, followed by variable declarations and assignments, followed by if statements and while loops. Once you have these working, you should move on to struct declarations and struct object creation. Then finally, move on to function declarations and function calls. You might also wait to handle \texttt{nil} values and types for the end as well (since \texttt{nil} also adds a number of special cases during type checking).

2. Within the symbol table, a struct type is represented using a Python dictionary of the form \{\( a_1 : t_1, \ldots, a_n : t_n \)\}, where each \( a_i \) is a member variable and \( t_i \) is the corresponding type of the variable. Each struct declaration will have an entry in the symbol table that associates the struct type name with its corresponding type.
3. Within the symbol table, a function type is represented using a Python list of the form 
\([t_1, \ldots, t_n], t\], where the first element is a list of the parameter types, and the second argument is the return type. Each function declaration will have an entry in the symbol table that associates the function name with the function’s declared type.

4. For a statement such as “\texttt{var obj = new S;}”, the type stored in the symbol table for the variable \texttt{obj} is \texttt{S}. To type check a statement such as “\texttt{set obj.a = v;}” for some value \texttt{v}, we first obtain the type of \texttt{obj}, which in this case is \texttt{S}, and use the symbol table to obtain the struct type for \texttt{S} to ensure that \texttt{a} is a member of \texttt{S} and that the type of \texttt{v} and \texttt{S}’s member attribute \texttt{a} align. A similar process is used when \texttt{obj.a} is used within an expression.

5. While return statements are primarily for the purpose of implementing functions, it is possible to have a return statement outside of a function in \texttt{MyPL}. We require that such a return statement must return either an integer value or \texttt{nil}.

6. You are free to write any helper functions you feel are useful in your visitor class. It might be useful, e.g., to have a function to get the first token associated with an expression for reporting errors.

On the due date, \textbf{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 (\url{https://www.cs.gonzaga.edu/dropoff/}).

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.
class SymbolTable(object):
    """A symbol table consists of a stack of environments, where each
    environment maps a (variable) name to its associated information
    """

    def __init__(self):
        self.scopes = []  # list of {id_name:info}

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

    def id_exists(self, identifier):
        return self.__environment(identifier) != None

    def add_id(self, identifier):
        # can't add if no environments
        if not self.scopes:
            return
        # add to the most recently added environment
        self.scopes[-1][identifier] = None

    def get_info(self, identifier):
        env = self.__environment(identifier)
        if env is not None:
            return env[identifier]

    def set_info(self, identifier, info):
        env = self.__environment(identifier)
        if env is not None:
            env[identifier] = info

    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)
mypl_type_checker.py

import mypl_token as token
import mypl_ast as ast
import mypl_error as error
import mypl_symbol_table as symbol_table

class TypeChecker(ast.Visitor):
    """A MyPL type checker visitor implementation where struct types
    take the form: type_id -> {v1:t1, ..., vn:tn} and function types
    take the form: fun_id -> [[t1, t2, ..., tn,], return_type]
    ""

def __init__(self):
    # initialize the symbol table (for ids -> types)
    self.sym_table = symbol_table.SymbolTable()
    # current_type holds the type of the last expression type
    self.current_type = None
    # global env (for return)
    self.sym_table.push_environment()
    # set global return type to int
    self.sym_table.add_id('return')
    self.sym_table.set_info('return', token.INTTYPE)
    # load in built-in function types
    self.sym_table.add_id('print')
    self.sym_table.set_info('print', [[token.STRINGTYPE], token.NIL])
    # put remaining built-in function types here ...

    def visit_stmt_list(self, stmt_list):
        # add new block (scope)
        self.sym_table.push_environment()
        for stmt in stmt_list.stmts:
            stmt.accept(self)
        # remove new block
        self.sym_table.pop_environment()

    def visit_expr_stmt(self, expr_stmt):
        expr_stmt.expr.accept(self)

    def visit_var_decl_stmt(self, var_decl):
        ... you need to define this one ...

    def visit_assign_stmt(self, assign_stmt):
        assign_stmt.rhs.accept(self)
        rhs_type = self.current_type
        assign_stmt.lhs.accept(self)
        lhs_type = self.current_type
        if rhs_type != token.NIL and rhs_type != lhs_type:
            msg = 'mismatch type in assignment'
            self.__error(msg, assign_stmt.lhs.path[0])

    # finish remaining visit calls ...

4
#!/usr/bin/python3
#
# Author: 
# Assignment: 5
# Description:
#   Simple script to execute the MyPL type checker.
#----------------------------------------------------------------------

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_type_checker as type_checker
import sys

def main(filename):
    try:
        file_stream = open(filename, 'r')
        hw5(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 hw5(file_stream):
    the_lexer = lexer.Lexer(file_stream)
    the_parser = parser.Parser(the_lexer)
    stmt_list = the_parser.parse()
    the_type_checker = type_checker.TypeChecker()
    stmt_list.accept(the_type_checker)

if __name__ == '__main__':
    if len(sys.argv) != 2:
        sys.exit('Usage: %s file' % sys.argv[0])
    main(sys.argv[1])
Type Inference Rules. The following rules help clarify type inference in MyPL. While the rules are more formal than a text-based description, some of the rules take liberties in terms of notation. Note that below we assume nil has type nil (i.e., nil : nil). In general, we use e to denote expressions, t to denote types, x to denote variable names, s to denote struct names, f to denote function names, and block to denote statement lists (e.g., the body of a function or an if clause).

\[
\begin{align*}
\text{(1)} & \quad \Gamma \vdash c : t \\
\text{(2)} & \quad \Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t \quad t \in \{\text{string, int, float}\} \\
\quad \quad \Gamma \vdash e_1 + e_2 : t \\
\text{(3)} & \quad \Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t \\
\quad \quad \Gamma \vdash e_1 - e_2 : t \\
\text{(4)} & \quad \Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t \\
\quad \quad \Gamma \vdash e_1 \times e_2 : t \\
\text{(5)} & \quad \Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t \\
\quad \quad \Gamma \vdash e_1 / e_2 : t \\
\text{(6)} & \quad \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \\
\quad \quad \Gamma \vdash e_1 \% e_2 : \text{int} \\
\text{(7)} & \quad \Gamma \vdash e : t \quad t \neq \text{nil} \\
\quad \quad \Gamma, \ var \ x = e; \vdash x : t \\
\text{(8)} & \quad \Gamma \vdash e : t \quad t \neq \text{nil} \\
\quad \quad \Gamma, \ var \ x : t = e; \vdash x : t \\
\text{(9)} & \quad \Gamma \vdash e : \text{nil} \\
\quad \quad \Gamma, \ var \ x : t = e; \vdash x : t \\
\text{(10)} & \quad \Gamma, \ var \ x_1 : d_1; \vdash x_i : t_i \quad \ldots \quad \Gamma, \ var \ x_n : d_n; \vdash x_n : t_n \\
\quad \quad \Gamma, \ \text{struct} \ s \ var \ x_1 : d_1; \ldots \ \var \ x_n : d_n; \ \text{end} \vdash \{x_1 : t_1, \ldots, x_n : t_n\} \\
\text{(11)} & \quad \Gamma, \ \text{block} \vdash e : t \\
\quad \quad \Gamma, \ \text{return} \ e; \in \text{block} \vdash \text{block} : t \\
\text{(12)} & \quad \Gamma, \ \text{return}; \in \text{block} \vdash \text{block} : \text{nil} \\
\text{(13)} & \quad \Gamma, \ \text{return}; \notin \text{block} \vdash \text{block} : \text{nil} \\
\text{(14)} & \quad \Gamma \vdash \text{block} : t' \quad t' \in \{t, \text{nil}\} \\
\quad \quad \Gamma, \ \text{fun} \ t \ f(p_1 : t_1, \ldots, p_n : t_n) \ \text{block} \ \text{end} \vdash f(t_1, \ldots, t_n) \rightarrow t
\end{align*}
\]
\[
\begin{aligned}
\Gamma \vdash s\{x_1 : t_1, \ldots, x_n : t_n\} \\
\Gamma \vdash \text{new } s : s\{x_1 : t_1, \ldots, x_n : t_n\}
\end{aligned}
\] (15)

\[
\begin{aligned}
\Gamma \vdash e : s\{\ldots, x_i : t_i, \ldots\} \\
\Gamma \vdash e.x_i : t_i
\end{aligned}
\] (16)

\[
\begin{aligned}
\Gamma \vdash e_1 : t_1 \quad \ldots \quad \Gamma \vdash e_n : t_n \\
\Gamma \vdash f(t_1, \ldots, t_n) \rightarrow t \\
\Gamma \vdash f(e_1, \ldots, e_n) : t
\end{aligned}
\] (17)

\[
\begin{aligned}
\Gamma \vdash e : t \\
\Gamma, \text{ set } x = e; \vdash e : t \lor e : \text{nil}
\end{aligned}
\] (18)

\[
\begin{aligned}
\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t \quad t \in \{\text{int, float, bool, string}, s\{\ldots\}, \text{nil}\} \\
\Gamma \vdash: e_1 \{=, !=\} e_2 : \text{bool}
\end{aligned}
\] (19)

\[
\begin{aligned}
\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : \text{nil} \quad t \in \{\text{int, float, bool, string}, s\{\ldots\}\} \\
\Gamma \vdash: e_1 \{=, !=\} e_2 : \text{bool}
\end{aligned}
\] (20)

\[
\begin{aligned}
\Gamma \vdash e_1 : \text{nil} \quad \Gamma \vdash e_2 : t \quad t \in \{\text{int, float, bool, string}, s\{\ldots\}\} \\
\Gamma \vdash: e_1 \{=, !=\} e_2 : \text{bool}
\end{aligned}
\] (21)

\[
\begin{aligned}
\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t \quad t \in \{\text{int, float, bool, string}\} \\
\Gamma \vdash: e_1 \{<, >, <=, >=\} e_2 : \text{bool}
\end{aligned}
\] (22)

\[
\begin{aligned}
\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool} \\
\Gamma \vdash e_1 \text{ and } e_2 : \text{bool}
\end{aligned}
\] (23)

\[
\begin{aligned}
\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool} \\
\Gamma \vdash e_1 \text{ or } e_2 : \text{bool}
\end{aligned}
\] (24)

\[
\begin{aligned}
\Gamma \vdash e : \text{bool} \\
\Gamma \vdash \text{not } e : \text{bool}
\end{aligned}
\] (25)

\[
\begin{aligned}
\Gamma, \text{while } e \text{ do } \ldots \text{ end} \vdash e : \text{bool}
\end{aligned}
\] (26)

\[
\begin{aligned}
\Gamma, \text{if } e \text{ then } \ldots \text{ end} \vdash e : \text{bool}
\end{aligned}
\] (27)

\[
\begin{aligned}
\Gamma, \text{if } \ldots \text{ else if } e \text{ then } \ldots \text{ end} \vdash e : \text{bool}
\end{aligned}
\] (28)