Weekly Reading. Ch. 6: 6.1–6.11 (Skim/Review), 6.12–6.15

Concepts to understand after the readings:
- Review of basic ideas about types
- Static, fixed stack-dynamic, fixed heap-dynamic, and heap-dynamic arrays
- Associative arrays
- Type types versus List types
- Union types (including discriminated unions)
- Type coercion versus casting
- Type errors
- Strong typing / strongly typed
- Name (nominal) vs Structure type equivalence (e.g., typedef vs struct)


The goal of this assignment is to implement a static “type checker” for MyPL that checks for type errors, use before def errors, function call errors, and variable shadowing 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 partial set of 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 structured type declarations and structured object creation. Then finally, move on to function declarations and function calls. You might also wait to handle nil values and types for the end as well (since nil also adds a number of special cases during type checking).

2. Within the symbol table, structured types should be represented using a Java HashMap such that variable names map to their corresponding types. Each structured type declaration will have an entry in the symbol table that associates the structured type name with its corresponding type.

3. Within the symbol table, a function type is represented using a Java ArrayList such that the formal parameter types of the function are members of the list (in corresponding order) and the last element of the list is the function return type.
4. For a statement such as "var obj := new T", the type stored in the symbol table for the variable obj is T. To type check a statement such as "set obj.a = v" for some rvalue v, we first obtain the type of obj, which in this case is T, and use the symbol table to obtain the type information for T to ensure that a is a member of T and that the type of v and T's member attribute a align. A similar process is used when obj.a is used within an expression (as an rvalue).

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 MyPL. We require that such a return statement must return either an integer value or 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.

7. You will need to “preload” the symbol table with each built-in function type. This is best handled by defining a default constructor for your visitor implementation that loads each built-in function type information. This can be added within a “super” global environment (i.e., an environment that the global environment is within).

8. You will need to pay special attention to how to report an error from the semantic analysis phase in terms of reporting line and column numbers (which are stored in the Tokens within the AST).

9. As part of your implementation you must use the code provided as an initial class/program design. Within your visitor implementation, however, you can define any helper functions you see fit.

What to Turn In: You must hand in the following by the due date for your assignment to be considered complete.

- A cover sheet with your name, the assignment number, and the date filled in
- A hard copy print out of your test files
- A hard copy print out showing your code runs correctly over all tests
- A hard copy print out of your “discussion” write up (see cover sheet)
- All program source code submitted through GitHub (instructions provided separately)
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. We use the notation $e : t$ to say that expression $e$ has type $t$. We assume $\text{nil}$ has type $\text{nil}$ (i.e., $\text{nil} : \text{nil}$). In general, we use $e$ to denote expressions, $t$ to denote types, $x$ to denote variable names, $s$ to denote structured type names, $f$ to denote function names, and $\text{block}$ to denote statement lists (e.g., the body of a function).

\[
\begin{align*}
\text{c is a constant with type } t & \quad \Gamma \vdash c : t \\
\Gamma \vdash e_1 : t & \quad \Gamma \vdash e_2 : t \quad t \in \{\text{int, double}\} \quad \text{op} \in \{+, -, *, \}, \\
\Gamma \vdash e_1 \text{ op } e_2 : t & \\
\Gamma \vdash e : t & \quad t \in \{\text{int, double}\} \\
\Gamma \vdash \text{neg } e : t & \\
\Gamma \vdash e_1 : \text{int} & \quad \Gamma \vdash e_2 : \text{int} \\
\Gamma \vdash e_1 \% e_2 : \text{int} & \\
\Gamma \vdash e : t & \quad t \neq \text{nil} \\
\Gamma, \text{ var } x := e & \vdash x : t \\
\Gamma \vdash e : t & \quad t \neq \text{nil} \\
\Gamma, \text{ var } t \ x := e & \vdash x : t \\
\Gamma \vdash e : \text{nil} & \\
\Gamma, \text{ var } t \ x := e & \vdash x : t \\
\Gamma, \text{ type } s \ \text{var } \tau_1 x_1 d_1 \ldots \Gamma, \text{ type } s \ \text{var } \tau_n x_n d_n \vdash x_n : t_n \\
\Gamma, \text{ type } s \ \text{var } \tau_1 x_1 d_1 \ldots \text{ var } \tau_2 x_2 d_n \text{ end } \vdash s : \{x_1 \to t_1, \ldots, x_n \to t_n\} \\
\Gamma, \text{ return } e \in \text{block} & \vdash \text{block} : t \\
\Gamma, \text{ return } \in \text{block} & \vdash \text{block} : \text{nil} \\
\Gamma, \text{ return } \notin \text{block} & \vdash \text{block} : \text{nil} \\
\Gamma \vdash \text{block} : t' & \quad t' \in \{t, \text{nil}\} \\
\Gamma, \text{ fun } f(t_1 p_1, \ldots, t_n p_n) \ \text{block} \ \text{end} & \vdash f : \{t_1, \ldots, t_n, t\} \\
\Gamma \vdash \text{new } s : s \\
\Gamma \vdash e : s & \quad \Gamma \vdash s : \{\ldots, x_i \to t_i, \ldots\} \\
\Gamma \vdash e. x_i : t_i
\end{align*}
\]
\[
\frac{\Gamma \vdash e_1 : t_1 \ldots \Gamma \vdash e_n : t_n \quad \Gamma \vdash f : \{t_1, \ldots, t_n, t\}}{\Gamma \vdash f(e_1, \ldots, e_n) : t}
\]

\[
\frac{\Gamma \vdash x : t}{\Gamma, \text{ set } x := e \vdash e : t \vee e : \text{nil}}
\]

\[
\frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t \quad t \in \{\text{int, double, bool, string, char, nil, s}\}}{\Gamma \vdash e_1 \{=, \neq\} e_2 : \text{bool}}
\]

\[
\frac{\Gamma \vdash e_1 : \text{nil} \quad \Gamma \vdash e_2 : t \quad t \in \{\text{int, double, bool, string, char, s}\}}{\Gamma \vdash e_1 \{=, \neq\} e_2 : \text{bool}}
\]

\[
\frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t \quad t \in \{\text{int, double, char, string}\}}{\Gamma \vdash e_1 \{<,\leq,\geq\} e_2 : \text{bool}}
\]

\[
\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \text{ and } e_2 : \text{bool}}
\]

\[
\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \text{ or } e_2 : \text{bool}}
\]

\[
\frac{\Gamma \vdash e : \text{bool}}{\Gamma \vdash \text{not } e : \text{bool}}
\]

\[
\frac{\Gamma, \text{while } e \text{ do } \ldots \text{ end } \vdash e : \text{bool}}{}
\]

\[
\frac{\Gamma, \text{if } e \text{ then } \ldots \text{ end } \vdash e : \text{bool}}{}
\]

\[
\frac{\Gamma, \text{if } \ldots \text{ elif } e \text{ then } \ldots \text{ end } \vdash e : \text{bool}}{}
\]

\[
\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma, \text{for } x := e_1 \text{ to } e_2 \text{ do } \ldots \text{ end } \vdash x : \text{int}}
\]