Today
- Evaluation (Interpretation)

Assignments
- HW5 out (due Thurs)
- Quiz on Thurs
Review: Structured Types in MyPL

Four places where structured types are used ...

1. User-defined type declarations ... e.g., type T var a := ... end
2. Object creation ... e.g., var s := new T
3. Rvalues (path expressions) ... e.g., set x := s.a
4. Lvalues (path expressions) ... e.g., set s.a := v

For simplicity, we represent a structured type using a Java Map:

- For a struct:
  ```
  type T
  var a1 := 0
  var a2 := ""
  end
  ```
- We use the type:
  ```
  Map<String,String> theType = new HashMap<>();
  theType.add("a1", "int");
  theType.add("a2", "string");
  ```

During a declaration, add type to symbol table ...

  symbolTable.addName("T");
  symbolTable.setInfo("T", theType);

For object creation, store the type name as the type ...

  symbolTable.addName("s");
  symbolTable.setInfo("s", "T");

For rvalues and lvalues, we have two steps ...

  String typeName = (String)symbolTable.getInfo("s"); // returns "T"
  Map<String,String> typeMap = (Map<String,String>)symbolTable.getInfo(typeName);
Review: Function Types in MyPL

Two places where function types are used

1. Function declarations

   ... e.g., fun int f(int x) ... end

2. Function calls

   ... e.g., var r := f(0)

For simplicity we represent function types using basic lists:

- For the function

```plaintext
fun bool cmp(string x, string y)
  var r := x == y
  return r
end
```

- We use the type

```plaintext
List<String> funType = new ArrayList<>();
funType.add("string");  // param type 1
funType.add("string");  // param type 2
funType.add("bool");    // return type
```

To add a function type to the symbol table:

```plaintext
symbolTable.addName("cmp");
symbolTable.setInfo("cmp", funType);
```
To type check the body of a function ... rough sketch:

```java
// add the function type first
... as above but from AST nodes ...

// add a new environment and a "special" return name
symbolTable.pushEnvironment();
symbolTable.addName("return");
symbolTable.setInfo("return", returnType);

// add parameters to environment
for (FunParam param : node.params) {
    // check for dups
    // add param name and types to environment
}

// finally, check the body
node.stmtList.accept(this);
symbolTable.popEnvironment();

• in the return statement visitor, ensure types align

Function calls ...

String funName = callRvalue.funName.lexeme();
List<String> funType = (List<String>)symbolTable.getInfo(funName);

• check that function was defined
• check that parameter and argument types align
• set the current type to the function return type
Variable “Shadowing”

- Redefining the same variable within a block leads to issues ... 
- Most languages disallow this (or more general) “shadowing”

An example of the problem in MyPL:

```plaintext
var x := 0
type T
  var y := x + 1
end
var x := 1
var t := new T # what should t.y be?
```

Your type checker should disallow block-level variable shadowing:

```java
private void visit(VarDeclStmt node) {
  node.varExpr.accept(this); // type check expression
  String expType = currType; // remember the result type
  String varName = node.varId.lexeme(); // variable we're adding

  // check for shadowing
  if (symbolTable.nameExistsInCurrEnv(varName)
    error("duplicate variable definition in current block", node.varId);
...
}
```
Some general hints:

- get each test file to work one at a time ... start with `p3.myp1`
- start with expressions, then var decls, then assignment stmts
- to start, don’t worry about “path expressions”
- move on to if, while, for
- do type declarations, then new, then path expressions
- do return, then function declarations, then function calls
- ask questions on piazza! ... (but don’t give away large sections of code)
HW6: Implement an Interpreter for MyPL

Writing a “pure (AST) interpreter” for MyPL:

- Overall similar to type checking
- We’ll again use the visitor pattern ... over AST nodes
- We’ll also use the symbol table ... this time, var -> value
- Instead of a current type, we’ll keep track of the current value
- Also need to maintain “heap space” for structured types (for HW7)

The basic class structure for HW 6

- In HW6 we won’t handle functions, structured types, or return statements
- We’ll add these as the last step in HW7

```java
public class Interpreter implements Visitor {
    SymbolTable symbolTable = new SymbleTable();
    Object currVal = null;
    // in HW7, we'll add the heap ...

    // assumes stmtList type checks
    public int run(StmtList stmtList) throws MyPLException {...

    // TODO: visitor functions
}
```

Our plan:

- start with basic evaluation hints (HW 6) ... next time
- go over hints for structs and functions (HW 7)
- move on to functional programming while you finish up the interpreter (today)