Lecture 17:

- Semantic analysis (cont)

Announcements:

- HW-4 out
- Proj. Part 1 due Fri

Symbol Table

Stores variable state in a "stack" of environments as program is being checked

```
public class SymbolTable {
  private Deque<Map<String,DataType>> environments = new ArrayDeque<>();

  // add and remove environments
  public void pushEnvironment() { ... }
  public void popEnvironment() { ... }

  // check if variable name is bound
  public void exists(String name) { ... }
  public boolean existsInCurrEnv(String name) { ... }

  // add a binding (overwrites existing name binding)
  public void add(String name, DataType type) { ... }

  // returns the name's binding
  public DataType get(String name) { ... }

  // print the symbol table (for debugging)
  public String toString() { ... }
}
```

Symbol Table

New environments created/removed when we visit new "blocks"

```
symbolTable.pushEnvironment();
for (var stmt : node.stmts)
  stmt.accept(this);
symbolTable.popEnvironment();
```

The plan:

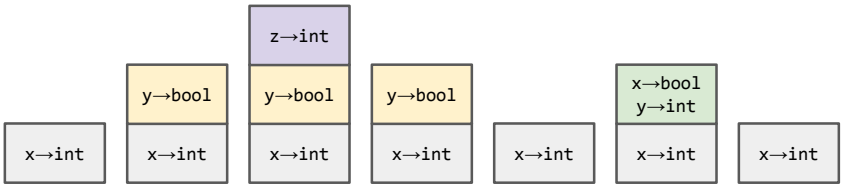
- basic idea of the type checker implementation for HW-4
- typing rules for MyPL
- hints on type checker (visitor) implementation

Symbol Table

Symbol table tracks "sub environments" (update as AST is navigated) ...

```
 1: var x = 1
 2: while x < 10 {
 3:   var y = true
 4:   if y {
 5:     x = x * 2
 6:     var z = 50
 7:   }
 8: }
 9: if x > 10 {
10:   var x = false
11:   var y = 11
12: }
```

Code Snippet



Symbol Table (type bindings) ...

Calls to `exists(name)` and `get(name)` search ...

1. in the local ("current") environment first ... top of "stack"
2. then the parent environment ... environment before top of "stack"
3. and so on

Semantic Checker

The `SemanticChecker` implements the visitor pattern

- includes a symbol table and the "current" inferred type (as `DataType`)

```
public class SemanticChecker implements Visitor {
  // for tracking function and struct definitions:
  private Map<String,FunDef> functions = new HashMap<>();
  private Map<String,StructDef> structs = new HashMap<>();
  // for tracking variable types:
  private SymbolTable symbolTable = new SymbolTable();
  // for holding the last inferred type:
  private DataType currType;

  // helper functions:
  private boolean isBaseType(String type) { ... }
  private boolean isBuiltInFunction(String name) { ... }
  private void error(String msg, Token token) { ... }
  private void error(String msg) { ... }
  private boolean isStructField(String fieldName, StructDef structDef) { ...
       }
  private DataType getStructFieldType(String fieldName, StructDef structDef)
       { ... }

  // visit functions ...
```


Semantic Checker

Inferred types recorded in `currType`, which is a `DataType` object ...

```
class DataType implements AST {
  public boolean isArray;
  public Token type;
  public void accept(Visitor v) {v.visit(this);}
}
```

Example for simple (literal) rvalues:

```
public void visit(SimpleRValue node) {
  TokenType literalType = node.literal.tokenType;
  int line = node.literal.line;
  int column = node.literal.column;
  Token typeToken = null;
  if (literalType == TokenType.INT_VAL)
    typeToken = new Token(TokenType.INT_TYPE, "int", line, column);
  else if (literalType == TokenType.DOUBLE_VAL)
    typeToken = new Token(TokenType.DOUBLE_TYPE, "double", line, column);
  // ... similar needed for string and bool here ...
  else if (literalType == TokenType.NULL_VAL)
    typeToken = new Token(TokenType.VOID_TYPE, "void", line, column);
  currType = new DataType();
  currType.type = typeToken;
}
```

Semantic Checker

Inferred types help check more complex statements and expressions

For example, for expression checking:

```
public void visit(BasicExpr node) {
  node.rvalue.accept(this);
}

public void visit(UnaryExpr node) {
  node.expr.accept(this);
  if (node.unaryOp.lexeme.equals("not"))  // not is the only choice
    if (!currType.type.lexeme.equals("bool") || currType.isArray)
      error("expecting boolean expression", currType.type);
}

public void visit(BinaryExpr node) {
  // set up the lhs
  node.lhs.accept(this);
  DataType lhsType = currType;
  // set up the rhs
  node.rhs.accept(this);
  DataType rhsType = currType;
  // get the operator
  String op = node.binaryOp.lexeme;
  // check the cases ...
}
```