Lecture 19:

- Wrap up Semantic Analysis

Announcements:

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

HW-4 Details

For HW-4, your job is to finish the visitor implementation ...

- given your parser w/ AST generation
- given the symbol table implementation (provided)
- given `MyPL` "typing rules" (also provided)
- given various unit tests (kinds of errors to catch)

Lots of details and some aspects are tricky ...

- variable "shadowing" rules ... `existsInCurrEnv(...)`
- built-in functions ... as special `CallRValue` cases
- handling `return` statements ... special `"return"` var
- the various type inference cases ... type rules
- handling path expressions ... save for last in your implementation
- reporting good error messages ... must have, won't be picky

Structs (User-Defined Types) in MyPL

Four places where structs are used ...

1. struct declarations ... e.g., `struct T { a1: int, ... }`
2. object creation ... e.g., `var t = new T(4, ...)`
3. rvalues (path expressions) ... e.g., `x = t.a1`
4. lvalues (path expressions) ... e.g., `t.a1 = v`

(1) Struct info stored as `StructDef` AST objects (in `structs`)

- ensure declarations are well-typed in `visit(StructDef node)`

(2) For object creation (i.e., in `visit(VarStmt node)`):

```
var t: T = new T(...)
```

Store result of variable declaration in the symbol table:

```
// currType inferred here from visit(NewStructRValue node)
symbolTable.add(varName, currType);
```


Structs (User-Defined Types) in MyPL

(3, 4) For rvalues and lvalues, we require two steps to get the type info:

```
// get variable's type (e.g., for var name "t" above)
varType = symbolTable.get(varName)

// check type is a struct
if (!structs.containsKey(varType.type.lexeme))
  // ... varName not a struct type ...

// get the struct definition
StructDef structDef = structs.get(varType.type.lexeme);

// check that the field is in the struct ("a1" in "T")
if (!isStructField(fieldName, structDef)
  // ... not a field ...

// get the field info (DataType)
fieldType = getStructFieldType(fieldName, structDef);

// ... do the checking ...
```

User-Defined Functions in MyPL

Two places where function type information is used:

1. Function declarations ... e.g., `int f(x: int) { ... }`
2. Function calls ... e.g., `f(42)`

(1) Function info stored as `FunDef` AST objects (`functions`)

(2) For a function call, first check if calling a built-in function:

```
public void visit(CallRValue node) {
  String funName = node.funName.lexeme;
  // 1. check if function is built in or user defined
  if (!isBuiltInFunction(funName) && !functions.containsKey(funName))
    error("undefined function", node.funName)
  // 2. check for built ins
  else if (funName.equals("print") || funName.equals("println")) {
    // ... check the call (number of args and types), set return type
  }
  // ... and similar for rest of built-in functions
```


User-Defined Functions in MyPL

(3) Otherwise, grab the type information from `functions`:

```
    // 3. check signature vs call
    else {
      FunDef funDef = functions.get(funName);
      // make sure same number of args as params
      if (fundDefs.params.size() != node.args.size())
        error("incorrect number of arguments", node.funName);
      // check each arg type
      for (int i = 0; i < node.args.size(); ++i) {
        node.args.get(i).accept(this);
        DataType paramType = funDef.params.get(i).dataType;
        // ... and so on
      }
      // set up return type
    }
  }
```

Handling "return" Statements

Consider the following MyPL function:

```
int f(x: int) {
  if x > 0 {
    return x - 1
  }
  else {
    return false
  }
}
```

Note that return statements handled in their own visitor functions:

- return handled in visit(ReturnStmt node)
- whereas FunDef handled in visit(FunDef node)

So when visiting return statements, *we don't know what function we are in* ...

- and so don't know the type we are trying to match against

Handling "return" Statements

To solve, add "return" variable to function's environment: ... Q: Why safe?

```
// ... in FunDef
symbolTable.pushEnvironment();
// ... add params
// add special 'return' var
node.returnType.accept(this);          // check ``known'' type
symbolTable.add("return", currType);
// check body statements
for (Stmt s : node.stmts)
  s.accept(this);
// all done with symbol table
symbolTable.popEnvironment();
```

Then in return statement visitor, look up special "return" variable type

```
public void visit(ReturnStmt node) {
  node.expr.accept(this);
  DataType returnType = symbolTable.get("return");
  // ... ensure the two are compatible
}
```