**Lecture 21:**

- Quiz 5
- Semantic Analysis (wrap up)

**Announcements:**

- HW-4 out
- Proj Part 1 due Mon after Spring Break

**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 ... `exists_in_curr_env(...)`
- built-in functions ... as special `CallExpr` 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 ... won't be picky about this

## User-Defined Types (Structs) in MyPL

Four places where structs are used ...

1. struct declarations ... e.g., `struct T { int a1, ... }`
2. object creation ... e.g., `T 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_struct_def(...)`

(2) For object creation:

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

Store result of variable declaration in the symbol table:

```
# curr_type inferred in this case from visit_new_rvalue(...)
self.symbol_table.add(var_name, self.curr_type);
```

(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)
lhs_type = self.symbol_table.get(var_name)

# check type is a struct
if lhs_type.type_name.lexeme not in self.structs:
  # ... var_name not a struct type ...

# get struct definition
struct_def = self.struct_defs[lhs_type.type_name.lexeme];

# check that the field is in in the struct ("a1" in "T")
if not self.get_field_type(struct_def, field_name):
   # ... not a field ...

# get the field info (DataType)
field_type = self.get_field_type(struct_def, field_name)
```

## Function (Signature) Types in MyPL

Two places where function type information is used:

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

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

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

```
def visit_call_expr(self, expr):
  fun_name = expr.fun_name.lexeme

  # check built-in types
  if fun_name == "print":
    ... check the call ...
  elif fun_name == "input":
    ... check the call ...
  ... and so on ...
```

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

```
  else:
    if fun_name not in self.functions:
      # ... function being called not defined ...

    fun_def = self.functions[fun_name]

    # make sure same number of args as params
    if len(expr.args) != len(fun_def.params):
      # ... bad call ...

    # check each arg type:
    for i in range(len(expr.args)):
      param_type = fun_def.params[i].data_type
      expr.args[i].accept(this)
      ... etc ...
    }
  }
  curr_type = fun_def.return_type
```

### Special Case: Handling return statements

Consider the following MyPL function:

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

Note that `return` statements handled in their own visitor functions:

- `return` handled in `visit_return_stmt(self, return_stmt)`
- whereas `FunDef` handled in `visit_fun_def(self, fun_def)`

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

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

To address this, can safely add a `"return"` variable to function's environment:

```
  ...
  # check the body statements (with special "return" var name)
  self.symbol_table.push_environment()
  # add the return type
  self.symbol_table.add("return", fun_def.return_type)
  ...
  for stmt in fun_def.stmts:
    stmt.accept(self)
  self.symbol_table.pop_environment();
```

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

```
...
return_type = self.symbol_table.get("return")
... check the type against the expression, etc.  ...
```