Lecture 24:

- Quiz 6 (!)
- Code Generation (getting started)

Announcements:

- HW-5 due Friday
- Proj Part 2 next Mon

IR Code Generation

The Plan

- Our last step is to convert ASTs to MyPL VM instructions
- We'll use the Visitor pattern for this
- I'll go over examples, the basic setup, and tips and tricks

The `CodeGenerator` class:

```
public class CodeGenerator implements Visitor {
  // vm to add frames to
  private VM vm;
  // current frame template being generated
  private VMFrameTemplate currTemplate;
  // variable -> index mappings with respect to environments
  private VarTable varTable = new VarTable();
  // struct defs for field names
  private Map<String,StructDef> structs = new HashMap<>();

  // ... helper functions for adding instructions, executing body statements

  // ... visitor functions ...

}
```

IR Code Generation

The `VarTable` class:

- helps keep track of variable offsets as code is being generated
- push and pop environments (like symbol table)
- add variable, get back it's corresponding offset
- pop reclaims the variable offsets

```
public class VarTable {
  // ... normal stuff (stack of environments) ...

  public void pushEnvironment() {...}
  public void popEnvironment() {...}

  public void add(String varName) {...} // assigns index automatically
  public int get(String varName) {...}  // get the index for the var

}
```


IR Code Generation

(1) Getting started: `Program` nodes ... just visit struct and function defs

```
public void visit(Program node) {
  // record each struct definitions and check for duplicate names
  for (StructDef s : node.structs)
    s.accept(this);
  // generate each function
  for (FunDef f : node.functions)
    f.accept(this);
}
```

(2) For structs, just add to `structs` map for later initialization

```
public void visit(StructDef node) {
  structs.put(node.structName.lexeme, node);
}
```

(3) Generating Functions: `FunDef` nodes

- Create a new frame template (as `currTemplate`)
- Push a new variable environment (via `varTable`)
- Store each argument provided on operand stack (from the `CALL`)
- Visit each statement node (to generate its code)
- Add a return (`PUSH`, `RET`) if last statement isn't a return
- Pop the variable environment
- Add the frame template to the VM