Lecture 27:

- Intro to IR code generation

Announcements:

- HW-5 out
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:

```python
class CodeGenerator(Visitor):
    def __init__(self, vm):
        # the vm to add frames to
        self.vm = vm
        # the current frame template being generated
        self.curr_template = None
        # for var -> index mappings wrt to environments
        self.var_table = VarTable()
        # struct name -> StructDef for struct field info
        self.struct_defs = {}

    def add_instr(self, instr):
        # helper to add to curr template
        # ...

    def visit_program(self, self, program):
        # ...

    # etc
```
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

```python
class VarTable:
    def __init__(self):
        self.environments = []
        self.total_vars = 0
    def __len__(self):
        # number of environments
    def __repr__(self):
        # to print environments
    def push_environment(self):
    def pop_environment(self):
    def add(self, var_name):
        # add var to current environment
    def get(self, var_name):
        # get offset for var (or None)
```
(1) **Getting started:** Program nodes

... just visit struct and function def

```python
def visit_program(self, program):
    for struct_def in program.struct_defs:
        struct_def.accept(self)
    for fun_def in program.fun_defs:
        fun_def.accept(self)
```

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

```python
def visit_struct_def(self, struct_def):
    # remember the struct def for later
    self.struct_defs[struct_def.struct_name.lexeme] = struct_def
```

(3) **Generating Functions:** FunDef nodes

- Create a new frame (as `curr_frame`)
- Push a new variable environment (via `var_table`)
- Store each argument provided on operand stack (from `CALL`)
- Visit each statement node (to generate its code)
- Add a return (PUSH, RET) if last statement wasn’t a return
- Pop the variable environment
- Add the frame to the VM
Basic function code generation examples:

(a) “Empty” function (adds a return)

```c
void f() {
}
```

Frame 'f'

0: PUSH(None)
1: RET()

(b) One-parameter function (store into function variable)

```c
void f(int x) {
}
```

Frame 'f'

0: STORE(0) // parameter x
1: PUSH(None)
2: RET()

(c) Two-parameter function

```c
void f(int x, bool y) {
}
```

Frame 'f'

0: STORE(0) // parameter x
1: STORE(1) // parameter y
2: PUSH(None)
3: RET()

(d) Simple return

```c
int f(int x) {
    return x + 1;
}
```

Frame 'f'

0: STORE(0) // parameter x
1: LOAD(0) // load var x
2: PUSH(1)
3: ADD()
4: RET()

Note: We are not doing any optimization! ... just straightforward translation