Notes:

- We have two following projects: both about code generation.
- You can probably re-write pcode to work on intel.
- Compiler is nothing but a code generator in general.
- Compiler is a program that takes programs and emits code.
- Our compiler takes pl0 code and translates it to pcode.
- It is more complicated to write code that generates code.
- We have a very similar architecture to RISC.
- It is very important to make sure that we stick to the specifications while generating code.
- Code generation is not about intelligence - it is about being detail oriented.
- The machine does not think for you - it does whatever you program it to do. Therefore, you should be very careful about details.

Today, we are going to go over function calls, function definitions, variable declarations and variable usage.

Function Calls:

- Whenever you have a function call, there is a stack layout.
- Remember frame pointer(fp): it is a register in the machine that is pointing to the function currently being executed

- When we generate code, we need to make sure that the generated code uses the stack layout correctly.
- Creation of the stack layout broke down into caller and callee.
- Prologue / epilogue: code to be emitted before/after emitting code for block.

- You push enough information to the stack so that you can restore the state before calling a function.

Example:

```
var global : int
function f(x : int)
  var local : int
  write x
f(global)
```

- Formals (the parameters to the function, in this specific case global is passed to f) are passed to the function via stack. At the compile time, we don't know what the exact address of a formal. Hence, we use offsets.
- Static link is set by the caller.
- Dynamic link is nothing but the old frame pointer.
- We are not managing the stack: we are generating code that manages the stack.
- Whenever we see a function declaration, we dump out the code for that function only once.
- Compiler needs to know addresses of symbols. In our symbol table, we will save addresses of symbols, so that, we can refer to them when they are used.
- Everything is offset from the frame pointer
- We emit jumps at the beginning of functions to jump over nested functions since we should not execute them immediately - they are executed when called. We emit jump before emitting nested function code, hence, we don't know where to jump initially. We will update them after generating below code.
- Tearing down (popping) stack is just updating the stack pointer. The previous values in memory will stay still there until they are overwritten.
- Call statements and function factors are AST nodes that represent function calls.

Variable Declarations:

- Whenever we see a function call, we set up the beginning of the stack frame before making an actual branch to our function code
- From your symbol table, you know how deeply nested your function is.
- Variables are stored in memory. Whenever we see a variable declaration, you create a space for variable in memory and save the offset in the symbol table.

Example:

```
var global : int     ← global variable 'global'
function f(x : int) ← f takes a input parameter: x
  var local : int    ← local variable of f
  write x            ← body for function f
f(global)            ← main func is composed of just one func call
```

```
0.  addi sp sp 1
1.  br 12
2.  psh ln sp
3.  psh fp sp
4.  mov fp sp
5.  addi sp sp 1
6.  br 1
7.  ld r0 fp -4
8.  wr r0
9.  mov sp fp
10. pop fp sp
11. pop ln sp
12. ret ln
13. addi sp sp 1
14. ld r0 fp 1
15. st r0 sp 0
16. addi sp sp 1
17. psh fp sp
18. bl -16
19. subi sp sp 1
20.pop r0 sp
21. subi sp sp 1
22. hlt
```

0: When we see a variable, we just emit a code that creates a space for the variable
1: Jump over the code regarding the function since we want to start running from main func

Function declaration itself generates code for prologue/epilogue

6: Since there is no nested functions inside f, we just do br 1

7: When you see usage of a variable, a load instruction gets generated to load the value of variable into a register
The caller supplies the parameters to the function. The reason why we have -4 in this instruction (a negative value) is that we go back to stack frame of caller.
Typical convention is to push the parameters in reverse order to the stack - we do the same
We don't know the exact addresses of the symbols but addresses relative to frame pointer

9 - 11: Epilogue.
14 - 21: This whole thing is the function call (setting up the stack frame)

Notes:

- To make code generation simpler, we don't do optimizations.
- Before you have optimal code, have correct code first.
- The only things that we should worry about AST and symbol table in code generation step is filling scope and address.
- For a function, in stack, we have: parameters, return value, static link, return address, and dynamic link and local variables. First two are placed by caller and the rest is placed by callee.
- If you look at intel ABI or intel stack frame, you will see the convention for intel.
- Function addresses: address in the code, not the address in the memory. In our architecture, we have separate memory for code.
- All the local variables begin with offset 1.