Review:

- Today, we will have a Q&A session
- We conclude the lectures on back-end
- At the end, we will go over OS, optimization and software security
- The skeleton changed with the last fix
  - You can look at the diff if you have already started implementing the code generator
- You can formally specify compiler using logic with no ambiguity
- Project 5 includes functions and variables - and the given expressions and statements
- Project 6 involves the rest
- You should create your own test cases
- Variable access part of codegen.md will be fixed

What does static link do?
First let's see what is stack frame:
It is used for freezing functions. In order to do that we need to capture all the information about the function. Such as, parameter values, return address, return value, who called us (dynamic link), local variable, static link (tells where the stack frame for the parent function is).
Static scoping:
Is a rule in the language telling that we can reach to symbols defined in the static scope we are in. So, we can also access any variables in the parent scope.
There is nothing in the hardware that handles static scope - we are implementing it in the compiler.
For example, if you didn't have static scoping in Java, you would need to give a unique name to every variable you have.
Static link is meant to point to the stack frame of the **parent** function - so that, we can use it to access variables in parent scopes
Whenever we create a stack frame, we preserve a pointer from our stack frame to the parent function's stack frame.
Address of the stack frame is a run time property - hence, we (generally) don't know where stack frame is going to be placed.
Dynamic link is different - this time, we point to **caller** function - not to parent function.

How do we know how many static links we need to follow to find a variable in a parent scope?
In our symbol table, we know depths of the symbols. The difference between the current depth and the depth of the symbol we want to reach will show us how many static links we need to follow.

Let's see how this code generation works in pcode?
st: store from register to memory
ld: load from memory to register

pointer: a special value that pointing to an address.

In the positive offsets we store local variables.

- `FP + (1 + i)`: ith local variable

You can check these in codegen.md

To follow the static links, we take the frame pointer of the parent scope from static link, and replace the current frame pointer with it.

We are mixing values and pointers in machine: we are using registers for both values and pointers for example. This is kind of confusing - hence, you need to interpret your values depending on your context.
For example here, registers can be used to keep an address or value.
mov r0 fp → r0 = fp
ld r0 r0 -2 → r0 = static link
ld r0 r0 1 → r0 = first local variable

Code is another data - we use the same address space for code and data. However, in our virtual machine, we use a different architecture: code and data are lying in seperated memories.