Notes:

- We are in the last phase of our compiler.
- You are writing a program that generates another program which does what input program is supposed to do.
- You use the stack to freeze the stack of a function to run another function.
- A lot of pseudo-code and actual pcode you need to emit are included in codegen.md.

Today, we are going to talk about how to go from AST to generating source code.

ASts and Code generation:

- For each AST node, we will have a corresponding way to generate code.
- Grammar defines the syntax of the language and AST is just a version for that syntax.
- Computer does not know what a language is - we can.
- So, we will write little compilers for each AST nodes and let the computer generate the code for the whole program.
- Our compiler is a runnable specification for our language.

What is prologue and epilogue?

prologue/epilogue are the code that you insert before and after the actual code for the function to construct/tear down the stack frame for the function

Local variables and parameters:

- Local variables are set up by the callee.
- In terms of the source language, parameters are inside the scope of the language - they are part of the function.
- It is the compiler that describes meaning for the syntax.
- Our compiler says that (like other languages), parameters should be passed by the callee

Note: It is kind of a 'fake' language, it is invented by us, machine cannot run it directly. Therefore, we are defining the meaning of it.

Writing lots of code does not mean you are a great programmer - but writing lots of correct code means.
Code generation will be more tricky when compared to previous assignments.
You need to make sure that everything works as expected to make whole code generation step work as expected.

Any seperate functions that you use together will have some kind of 'contract'.
We know what a function is supposed to do without knowing the inside implementation.
We will do the same in code generation - we will assume functions for different AST nodes do what they are supposed to do.

Static link usage:
We use static link to keep track of parent function's stack frame. We will see how it works with nested functions.
It is useful to have static scoping - some languages do not have it. It is an invented feature, we use it in our language.

Frame pointer usage:
Frame pointer points to the currently active frame.
When you want to freeze the state and run another function, you push your 'state' (which also includes frame pointer) to the stack, and, we restore when we return from the function.

Symbol table usage:

- Symbol table tells you how many static links to follow a symbol (difference in nesting depth)
- If the caller is an ancestor, things go more complicated. You need to find the frame pointer for the ancestor. Maybe it is not the immediate ancestor, It is 'some' ancestor. You follow the static links to reach that. (current level - level of the symbol you want to reach) times.
- You can think of the global scope as a special function scope. You can use the same exact algorithm to find symbols for other functions.
- Global scope is half of a standard stack frame.

Var declaration:
We store variable declarations in the stack. For each variable declaration, we allocate a space on the stack (using addi sp sp 1)

Backpatching: we generate a branch but until we generate the code for the nested function, we don't actually know where to branch. We emit that branch instruction and save where that instruction lies. After we emit function related code, we update where branch is going to take.

Function declaration:

- Inside function, we have lots of different things.
- Caller makes space in the stack for the function
- Code generation is where we figure out the offsets for the variables. We save them in the symbol table.
- Prologue and epilogue are generated by the funcdecls
- bl instruction (Branch and link) sets the link register to return address and branches.

In **prologue**, we do:
1- Save the return address (which is in ln) in the stack
2- Save the dynamic link in the stack
3- Update frame pointer

Ershov numbering is a algorithm to figure out optimize register usage.

If you don't have any explicit return or your function does not return anything, funcdecls will generate epilogue anyway. If you use a return statement, it will also create another epilogue.

In **epilogue**, we do:
1- Tear down the stack frame
2- Restore the stack pointer (sp)
3- Restore the frame pointer (fp)
4- Retrieve the return address (into link register ln)

There is whole bunch of code around function call. It is for setting up / tearing down the stack.

In **caller setup**, we do:
1- Allocate space for the parameters.
2- Store the parameter on stack.
3- Allocate space for the return value.
4- Save the static link.
5- Call the function (using bl instruction).
6- Retrieve the static link.
7- Collect the return value.
8- Remove the parameters from the stack (deallocate the allocated space).

Notes:

- There are lots of different programs that behave equivalently (program equivalence).
- If you want to optimize your code, don't forget to let us know because we will grade your code. generation project by checking if it exactly matches with the suggested pattern. Otherwise, it will be graded manually.
- In code generation assignment, again we will have visitor functions that will emit code.