Introduction
Last time we went over expressions: traversing trees for binary expressions, ershov numbering and allocating registers
Today, we will talk about statements.
Bottomline: nearly everything is tree traversal in this course
It is a special type of tree traversal, where each node in tree has a different type
For each type, we have different function that visits it

We use ershov numbers to decide number of registers and the register to use

We have ershov numbers, register base and return registers
The return value (return register) of an expression visitor function shows the register that the value evaluated by that function will be put

We are not actually interpreting the program, we are generating the code that evaluates it

Leaves are base cases
Once we get to the leaf node, the base register is the register we use for storing the value
We will use the formula (reg_base - ershov + 1) to compute return register.
Just stick with the algorithm, then, right values will be passed to and returned from the functions

The ershov numbering scheme decides which child to use/visit first.
If the ershov numbers are the same, we start from left. Otherwise, we start with the one that has larger ershov number
The conclusion: the traversal is post-order but not purely from left to right (or right to left). We decide that by looking at the ershov numbers

When you have difficult algorithms, it is best to work through the model of what the algorithm is doing

Project5 is for generating code for functions and variables (we are not going to try any expressions/statements while grading unless it is already given in the skeleton)
Project6 is for generating code expressions and statements

Somebody found a bug in the skeleton code, which is fixed now. Make sure to download the new codegen.c for your skeleton. There are very few changes.

Statements
There are only a few functions to implement for statements - a bunch of them are given

The tricky parts are: if statement and the while statement
If statement is just like generating code for boolean operations
Example pl0 codes and equivalent VM codes are given in markdown files in syllabus repository

If you do some compare operations on VM, it will set some flags in the machine (two flags: z and n). Following conditional branch operations will observe these flags and behave accordingly

Technically, each architecture you have, you need a new compiler since they use different instruction sets

Modern compilers are based on compiler framework, which separates the front-end and the back-end, which makes it easier

```
if true
  write 1
else
  write 0
```

See the diagram included in this class notes folder to see the tree for the above pl0 code.

Generating code boolean factor is same as the number factor: just emit a move instruction (movi).

1. Visit expression (the condition of if statement)
2. if_statement first emits *comparison* and *branch-if-not-equal* instructions.
3. We visit the statement, if_statement (it could be any statement. It is write statement in our example).
4. Emit an unconditional branch just after statement’s code for jumping over else branch - we will skip else branch if if_statement runs. We don’t know where to jump yet, since, we didn’t generate to code for else_branch. It will be backpatched later.
5. The statement emitted in step 3 could have been two lines or a hundred lines - we couldn’t have known. Now we know since generated code for if_statement. Update the displacement of the branch for the branch instruction emitted in step 2.
6. Generate code for else_branch
7. Update displacement of unconditional branch generated in step 4.

The pcode for above pl0 code:

```
movi r0 1
cmpi r0 1
bne 4
movi r0 1
wr r0
br 3
movi r0 0
wr r0
```

A while statement, in terms of machine code, is virtually identical to an if-then statement.
It is just an if statement with an unconditional branch at the bottom.

A simple while loop example in pl0:

```
while true do
  write 1
```

Corresponding pcode:

```
movi r0 1
cmpi r0 1
bne 4
movi r0 1
wr r0
br -5
```

You can figure out some information at compile time to increase performance: a whole research field on compilers - static analysis

Start coding so that you can come up with more questions for

There are two AST nodes used for function calls: visitCallStatement and visitFunctonFactor
Inside the skeleton given to you, there is a helper function: setupFunctionCall(), which will be used for both them. setupFunctionCall() will be implemented by you.

You are given the function for emitting code: emit(). Just put the instruction you want to emit as the parameter. It will return you the index on the code array that the instruction is stored
For backpatching, you are given a helper function: backpatch()

See the given visitBlock() code to understand how backpatch() function is used

We should freeze our expression evaluation in the stack before calling a function because function might use registers that are already in use. It is done in visitFunctionFactor(), which is given.