Introduction

Last time we talked about how we implement static scoping, how functions and variables are implemented for code generation
Today, we will talk about how we generate code for expressions

Codegen.md: Second half is about expressions and statements

Tricky part: register allocation
We use ershov number that we assigned to each AST node for making register allocation much easier
Work through an example yourself to better understand how ershov numbering algorithm works

Machine representation of numbers and booleans are different
The compiler is smart enough to figure out if the expression is boolean expression or arithmetic expression and it decides to use different sets of instructions to handle each of the expression (arithmetic or boolean).

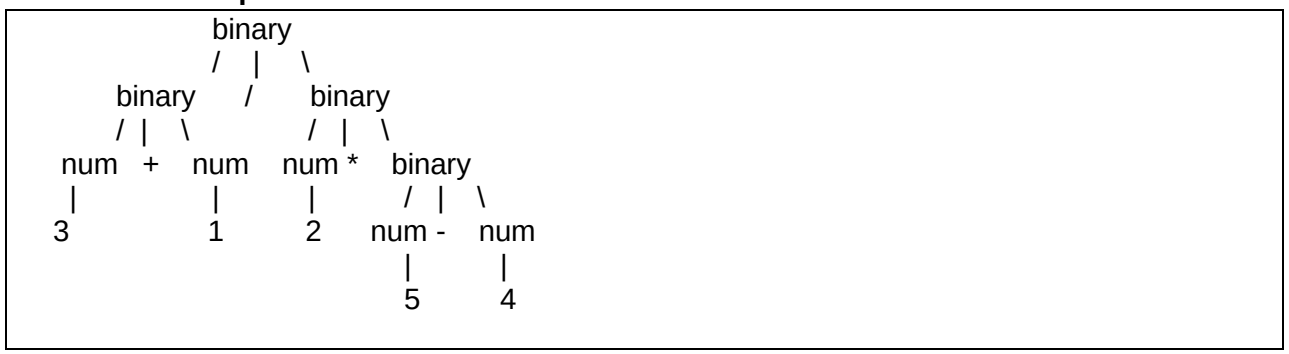
How can we actually take a boolean value and do a boolean operation using vm?
Because vm does not support direct and/or operations for boolean values, we will simulate it with our code generator
We have different visitors in our skeleton code to be used depending on the type of the expression

Code generation for expressions using ershov numbering

ershov.pl0 / ershov.pcode (look a bit complicated. AST is encoded in comments. Yet, includes just an expression)

Our main representation of the source code is a tree, and code generation is just a tree traversal
We will walk over the AST of ershov.pl0 to see how code is generated while tree traversal
We will do post-order traversal: first children, then parent
When encountered a number factor: move the machine representation of that value to a register (movi)

AST for ershov.pl0:



Initially, assume that we don't know which registers to use. Let's use [?] instead of an exact register id

We start from the last children (leaves) in the tree and we traverse the tree recursively for each child node (bottom up).

The pcode for ershov.pl0 ends up being the following (extracted from a tree traversal):

```
movi [?] 3
movi [?] 1
add [?] [?] [?]
movi [?] 2
movi [?] 5
movi [?] 4
sub [?] [?][?]
mult [?] [?] [?]
div [?] [?] [?]
```

There are lots of fancy ways to optimize this code. However, we will do it in a very simple way to make the code generation step easier
In fact, modern compilers would not generate any code for ershov.pl0 since we have all constants - no variables or functions. Hence, it could have been computed at compile time (constant folding) to make the program faster (we will not do that.)

There are some techniques called control flow analysis and data flow analysis, with which you could even do more optimization and analysis on the source code

Ershov numbering algorithm allows optimal register allocation: with that, you can use minimal possible number of registers to compute your expressions

We have a limit: limited number of registers. What if we need more than what we have as the number of registers? Our program won't compile. However, as a bonus exercise, you can implement a solution for it

Remember ershov numbering:

- All the leaf nodes have ershov number 1.
- If two children have the same ershov number, we add one
- If not, use the larger ershov number

Ershov numbering will allow us to reuse registers
The rules how to put ershov numbers to use as register ids are explained in codegen.md in detail

Ershov number basically means how many registers you need (for that expression)
Everything will end up in register 0: that is why we have slight modification on the algorithm explain in Dragon's book
All expression visitor functions take a register base as a parameter and returns a value

We start from top and we start with reg_base for the root node (reg_base = 2 in this example, and ershov = 3).
Then we go to right child first if the ershov is the same (ershov = 2, and reg_base = 2, so reg = reg_base - ershov + 1 = 2 - 2 + 1 = 1). If the ershov is not the same start with the bigger ershov number's node).
for leave nodes the reg is just the same reg_base
The detailed instruction is in codegen.md under Register Allocation section

The final pcode ershov.pl0 is:

```
movi r0 4
movi r1 5
sub r0 r1 r0 # r0 is destination register and we use the formula for
             # .. return reg
movi r1 2
mult r0 r1 r0
movi r1 1
movi r2 3
add r1 r2 r1
div r0 r1 r0
```

You are given a bunch of code for functions calls. You are going to save all the register on the stack before calling a function and restore them when returned from the function, so that, you will not lose your values inside the registers. This is done in FunctionFactor.

Simulation of boolean operations

We don't have some instructions in VM: and, or, negation etc. We will simulate these operations by using compare and branch instructions. Some examples are included in expressionsbool.pl0 test file (corresponding vmcode: expressionsbool.pcode)

Let's say we want to negate a boolean value in register r0. We use the following code:

```
cmpi r0 0   # compare the condition with 0
beq 3       # if the condition is false (0), branch 3
movi r0 0   # set condition to false (0). Will execute if the
            # .. condition was true
br 2       # skip the below code that makes condition true
movi r0 1  # set condition to true (1). Will execute if the
           # .. condition was false
```

To simulate less than operations (between r0 and r1) and write the result to r0:

```
cmp r1 r0
blt 3
movi r0 0
br 2
movi r0 1
```

AND operation(short circuiting happens here):

```
cmpi r1 0
beq 5      # short circuiting here
cmpi r0 0
beq 3
movi r0 1
br 2
movi r0 0
```

OR operation:

```
cmpi r1 1
beq 5       # short circuiting here
cmpi r0 1
beq 3
movi r0 0
br 2
movi r0 1
```

See visitBinaryExpressionBool function in codegen.c. Skeleton gives you example code, you will need to fill the rest of the boolean operations.