Designing The Code Generator

- Given:
  - A target language – The EVM in our case.
  - An intermediate representation in the form of a tree – the AST in our case
- Find:
  - For each node in the source tree find a combination of instructions in the target language.

The Generator

```
public class Generator implements
 Visitor, expressive.evm.Instructions {
  final Global global;
  final PrintWriter out;
  final Code code = new Code();

  public Generator(Global global,
                   PrintWriter out) {
    this.global = global;
    this.out = out;
  }

  public void generate(Tree tree) {
    tree.apply(this);
  }
…
```


Bottom up Design

- Lets try to design the code generator from the bottom up. That is, we will start with leaves in the AST and then work our way up.
- Since the target machine is a stack machine we will want to get the result of each expression on the top of the stack.
- We will use the Visitor pattern over the AST.
- We assume we have a function emit which produces code for the target.

Literals

- The literals in the AST can contain either a string or an integer.
- Since string literals can only occur in a `prints` node we will ignore them for now.
- For a value what we want to do is to push it onto the top of the stack.

Code for Literals

```
public void caseLiteral(Literal tree) {

  if (tree.str != null) // String or int lit?
    global.fatal(tree.pos,
      "Cannot generate code for a string literal");

  code.emit(PUSH, tree.value);
}
```


Identifiers

- Identifiers represents uses of variables.
- Variables will be stored in memory.
- Each identifier is associated with a symbol.
- The symbol can keep track of where in memory the variable is stored – let's assume it does.

Identifiers (cont)

- We need to get the contents of the variable from the memory to the top of the stack.
- The LOAD instruction does this, assuming we know the memory position.

Code for Identifiers

```
public void caseIdent(Ident tree) {
  // id

  code.emit(LOAD,
            tree.symbol.getOffset());

}
```


Assignment

- The assignment node consists of an identifier and an expression:
  `id := expr`
- The statement should store the value of `expr` to the memory location of `id`.
- We assume we can call `generate` to generate code for an expression.
- We assume that the value of the evaluated expression will be on the top of the stack.
- All we need to do is emit code for storing the top of the stack to memory with `STORE mempos`.

Code for Assignment

```
public void caseAssign(Assign tree) {
  // id := expr

  // Recursive call to generate code
  //  for the expression expr.
  generate(tree.rhs);

  // Code for storing the result in memory
  //  at the location of id.
  code.emit(STORE, tree.symbol);
}
```


Code

- We have assumed that we have an emit function and that each symbol is allocated to a memory position.
- This is handled by the Code class.
- We keep a vector of emitted instructions, and a count of used memory positions.

The Code Class

```
public class Code implements
   expressive.evm.Instructions {

 private final Vector/*<int>*/ code = new Vector();
 private int memSize = 0;
…
 public Code emit(int instr) {
   code.add(new Integer(instr));
   return this;
 }

 public Code emit(int instr, int arg) {
  emit(instr);
  return emit(arg);
 }
```

Part VI

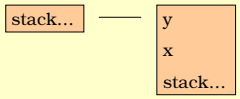
The Magic in Code

```
public Code emit(int instr, Symbol sym) {
 return emit(instr, declareVar(sym));
}

public int declareVar(Symbol sym) {
 // Offset -1 indicate unallocated symbol.
 if (sym.getOffset() < 0)
  sym.setOffset(memSize++);

 return sym.getOffset();
}
```


Part VI

Operators

- The operators can be constant (`TRUE`, `FALSE`), unary (`NOT expr`) or binary (`expr ADD expr`, etc).
- For the binary operators we want the right hand side (y) on the top of the stack and the left hand side (x) below that on the stack.



Part VI

Operators

- We can use `generate` to generate code for the sub trees, first left (x), then right (y).
- Then we just have to figure out what each operation should do.
- Most ops have a direct correspondence in the EVM.

Part VI

Booleans

- In the EVM Booleans are represented by the integer 0 for false and any other integer for true.
- This means that we can implement the Boolean operators:
  - `! expr` (`NOT x => x = 0`)
  - `Expr | expr` (`x OR y => x + y`)
  - `Expr & expr` (`x AND y => x * y`)
- As:

NOT x:	x OR y:	x AND y:
`PUSH 0`	`ADD`	`MUL`
`CMPE`		

Part VI

If

- To generate code for if we will need to use control flow instructions.
- For

  "`if Cond {Statements1} else {Statements2}`"

  we need to first generate code that evaluates `Cond`.

  Then we need to generate code that checks if `Cond` is false (=0) and if so jumps over all the statements in Statements1.

Part VI

Labels

- In order to easily jump to some point in the code the Code class provides labels.
  - `getLabel()` returns a new label.
  - `markLabel(Label label)` marks the label to point to the current location in the code.
- For the implementation look in Code.java.

...If again

- Armed with labels we can make the following outline for the generated code for `if Cond {Statements1} else {Statements2}`

```
Code for Cond
JZ elseL // Jump if Cond=0 i.e. false
Code for Statements1
elseL:
Code for Statements2
```


...and again

- We do not want to execute the code for `Statements2` after `Statements1` if `Cond` is true:

```
Code for Cond
JZ elseL // Jump if Cond=0 i.e. false
Code for Statements1
GOTO exitL // Skip Statements2
elseL:
Code for Statements2
exitL:
```


Code for if

```
public void caseIf(If tree) {
  // if Cond {Statements1} [else {Statements2}]
  Code.Label elseL = code.getLabel();
  Code.Label exitL = code.getLabel();

  generate(tree.cond);        // Code for Cond
  code.emit(JZ, elseL);       // JZ elseL // Jump if Cond=0 i.e. false
  generate(tree.thenp);       // Code for Statements1
  if (tree.elsep.length > 0)
      code.emit(GOTO, exitL);// GOTO exitL // Skip Statements2
  code.markLabel(elseL);      // elseL:
  if (tree.elsep.length > 0) {
      generate(tree.elsep);   // Code for Statements2
      code.markLabel(exitL); // exitL:
  }
}
```


That's about it

- You will get Code.java and a skeleton for Generator.java.
- You will have to figure out how to handle `caseProg`, `caseWhile`, and `caseIO` nodes by yourself.
- You will also have to finish `caseOP`.