Main Program Initialization and Termination

At main program initialization, we need to generate code for all the functions first, followed by the variables, followed by the statements. The program counter must be started after all the function declarations, but before the variable declarations.
Note: this is similar to what needs to be done during function initialization, where we must first generate all code from nested functions, then local variables, then statements. In such a case, the branching label for the main function will be after the generated code for all of its nested functions but before the local variable declarations.

Programs need to be terminated with the HALT instruction. The simplest way to do this is to add HALT as the last instruction after the rest of the code generation is complete.

so for a program:

```
{
        var a:integer
        func x (b: integer) : integer {
                result b+1
        }
        a := 3
        x(a)
}
```

the machine code would be:

```
FUNC_LABEL:
  ADDR 1  -1
  LOAD
  PUSH 1
  ADD
  ADDR 1 -2
  SWAP
  STORE
  PUSH 0 //for the number of local variables
  POPN // pop all local variables… in this case 0 of them
  BR
START:
  PUSH UNDEFINED //for local variable a
  ADDR 0 0 //load address of a into memory
  PUSH 3
  STORE //store 3 into a
  PUSH UNDEFINED // for return value of function
  ADDR 0 0
  LOAD // load a to the top of the stack
  PUSHMT //push the current stack pointer
  ADDR 1 0 //push the previous display address at the lexical level of function x
  SWAP
  SETD 1 //set the display value at lexical level one to the stack pointer we had pushed earlier
  PUSH RETURN_LABEL
  PUSH FUNC_LABEL
  BR
RETURN_LABEL:
  SETD 1 //set the display addr at LL 1 to be the previous display addr before the function call
  PUSH 1 //Num arguments in func x
  POPN
  HALT
```

Any Handling of Scopes not Described Above

All scope related issues have been discussed elsewhere in the document.

Other Relevant Information

Expressions of the form '(' expression '?' expression ':' expression ')'

We will first evaluate the conditional expression, if it is false we will jump to a label which evaluates the value if it's false and then continues execution, otherwise we will evaluate the value if it's true and jump to the end of the conditional expression.

Example:
e1 ? e2 : e3

The machine code is:

```
<eval e1>
push falseValue
BF
<evaluate e2>
push endConditional
BR
falseValue
<evaluate e3>
endConditional
```

LABELS:
Labels are used to allow us to branch in our code. The working of labels relies on the idea that we are going through an intermediate step before actually writing to the machine. We are going through a first pass of the AST and generating an instruction list that will need to be written to the machine after the pass has been completed.

In order to use labels, we will create 2 label classes:

1. LabelPlaceHolder: This will be a simple class that stores a unique string to represent the name of the label. We can keep a static integer in this class that gets concatenated to every label name as it is created and then incremented in order to ensure its uniqueness.
The label placeholder will be used in the intermediate step of code generation in order to branch to the location represented by that label. In our intermediate code, we can say something like:
PUSH “myLabelHolder”
BR
This label will get resolved later. This will be discussed soon.

2. LabelLocation: This class will act as an instruction. It will have a size value of zero. Objects of this class will be placed right before the instruction where anything branching to the label needs to go.
For example if BRANCH RET_LABEL needs to jump to instruction 40
then
LabelLocaltion(RET_LABEL) will be placed right before instruction 40.

In our second pass, we need to go through the code generated and count the size of instructions. Instructions such as HALT, LOAD, STORE that have no arguments have size 1. Instructions like SETD which takes 1 argument has size 2. Instructions such as ADDR which have 2 arguments will have size 3.

We must keep a running count of the sizes and every time we encounter a LabelLocation, we add it to a Label keymap that maps the label name to the the size of the instructions up to that point. This gives us the instruction location where each label can be found.

We need to then go through a third pass, and write all of this code to the machine. When doing this, everytime we encounter a LabelPlaceHolder object, we need to replace it with it’s instruction address that we can look up in the Label keymap we built. We also need to Omit any LabelLocation objects from being written to our machine.