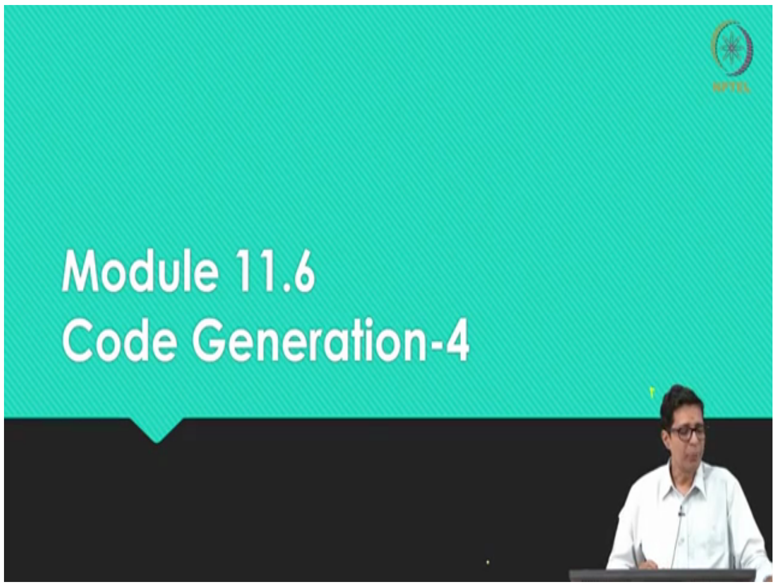
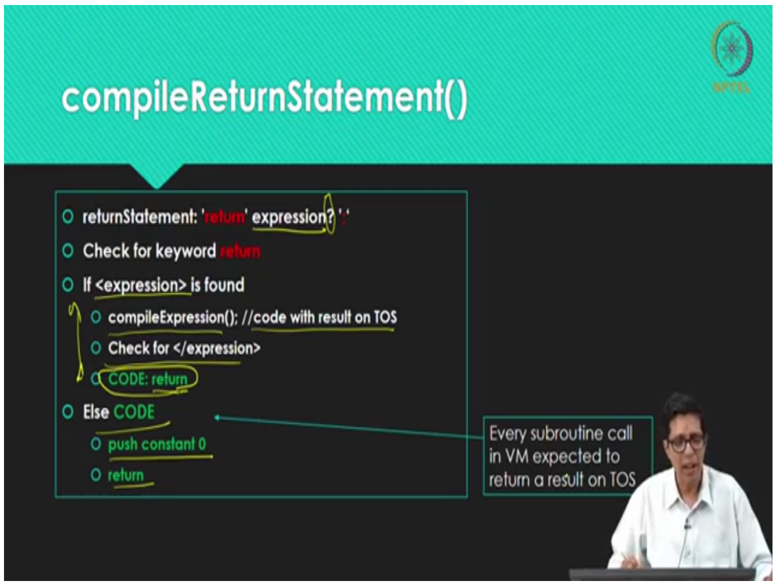
Foundations to Computer Systems Design
Professor V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology, Madras
Module 11.6
The Jack Compiler Backend: Code Generation - 4

(Refer Slide Time: 00:16)


Module 11.6
Code Generation-4

Welcome to module 11 point 6, we will do one more module here.

(Refer Slide Time: 00:20)


compileReturnStatement()

- returnStatement: 'return' expression? ';'
- Check for keyword return
- If <expression> is found
  - compileExpression(); //code with result on TOS
  - Check for </expression>
  - CODE: return
- Else CODE
  - push constant 0
  - return

Every subroutine call in VM expected to return a result on TOS

This is the compileReturnStatement right, so how does the return statement look it will be returned the keyword return followed by just a semicolon or an expression so the expression is actually optional here right, you may have the expression or not I have the expression. So how will you code this returnStatement you first check for the keyword return then if expression is found the next one is expression then you call compile expression then what will compile expression to it will the code with code will it will generate the code for the expression evaluation of the expression and the result of that execution will be on top of stack.

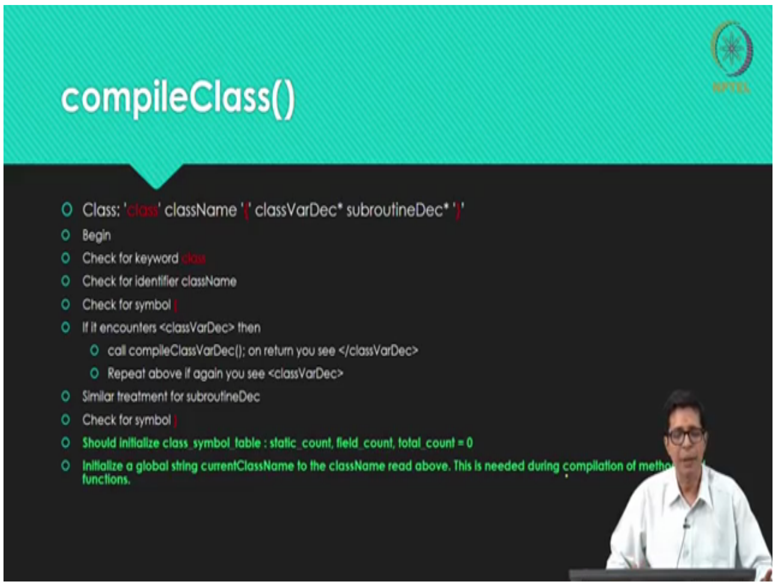
Now you check for slash expression right then you just say return the only code that you need to (re) be right here is return right. So when you return if you go back to the original VM implementation the calling function will take the result from the top of the stack right and that is essentially, so already this is on top of the stack so the calling function will take the result from the top of the stack.

So this is how the return basically works but if you do not see an expression then it means returning nothing (ret) just return right, so you do some action and just return. There the calling function will always as per the VM implementation the calling function will always see some result on the top of the stack right, so even if you return nothing there should be something on the top of the stack because the calling function after you return the calling function will pop the stack and start processing right.

So for that purpose if you do not see an expression if you do not see an expression if the expression is found do this part of the code if expression is not found then just push constant 0 and return, right that is all, right. So because every subroutine call in VM is expected to return a result on top of stack, so just push some constants here, right so that is (())(02:30), so this is how the return statement works

So these are the two lines of course, so you this is how you code the return statement, so what I have seen so far is for 11 different routines in this module starting from.

(Refer Slide Time: 02:53)

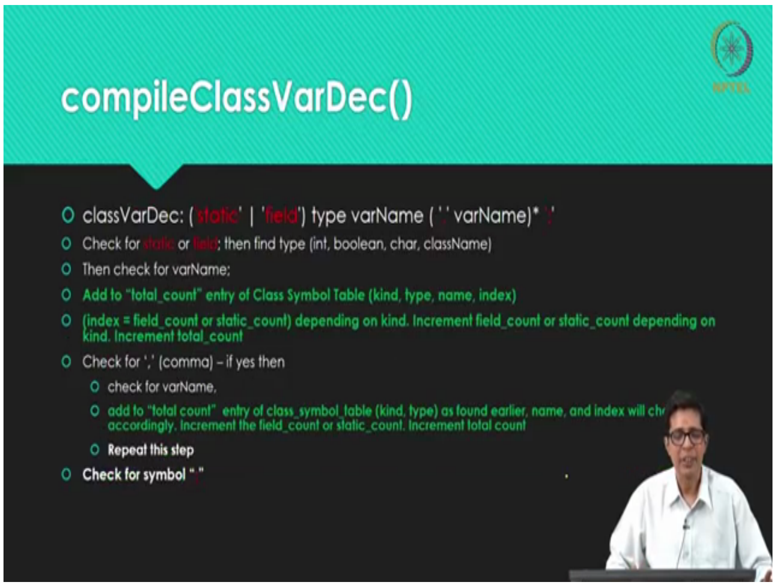


compileClass()

- Class: 'class' className '{' classVarDec* subroutineDec* '}'
- Begin
- Check for keyword class
- Check for identifier className
- Check for symbol {
- If it encounters <classVarDec> then
  - call compileClassVarDec(); on return you see </classVarDec>
  - Repeat above if again you see <classVarDec>
- Similar treatment for subroutineDec
- Check for symbol }
- **Should initialize class_symbol_table : static_count, field_count, total_count = 0**
- **Initialize a global string currentClassName to the className read above. This is needed during compilation of metho functions.**

So let us quickly go to this part for the 11 different modules starting from compileClass.

(Refer Slide Time: 02:55)

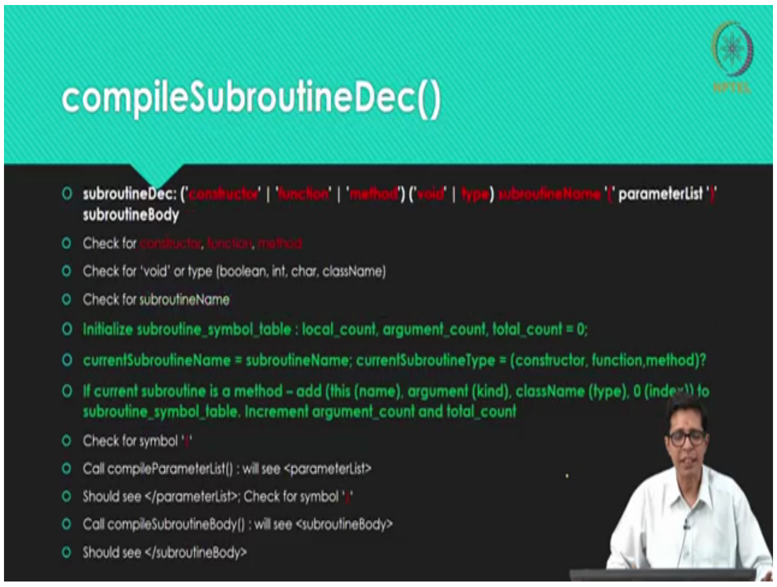


compileClassVarDec()

- classVarDec: ('static' | 'field') type varName (',' varName)* ';'
- Check for static or field; then find type (int, boolean, char, className)
- Then check for varName;
- **Add to "total_count" entry of Class Symbol Table (kind, type, name, index)**
- **(index = field_count or static_count) depending on kind. Increment field_count or static_count depending on kind. Increment total_count**
- Check for ',' (comma) – if yes then
  - check for varName,
  - **add to "total count" entry of class_symbol_table (kind, type) as found earlier, name, and index will ch accordingly. Increment the field_count or static_count. Increment total count**
  - **Repeat this step**
- **Check for symbol ";"**

compileClassVarDec.

(Refer Slide Time: 02:56)

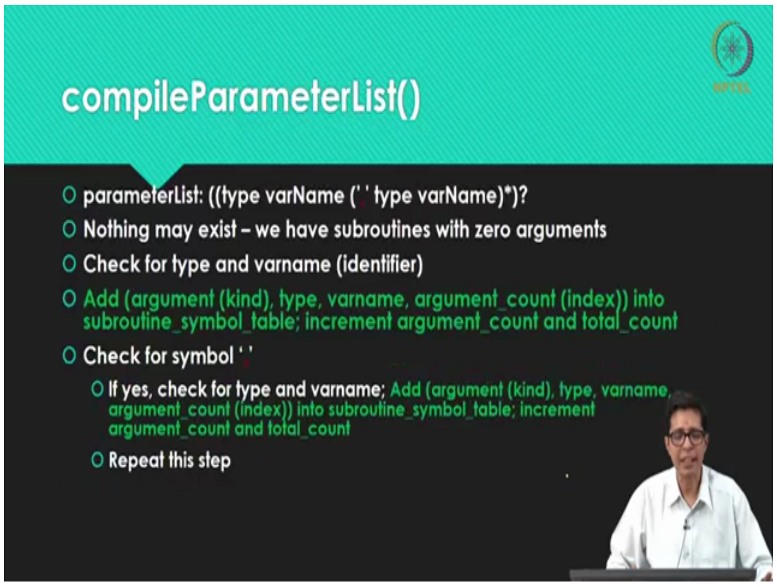


compileSubroutineDec()

- subroutineDec: ('constructor' | 'function' | 'method') ('void' | type) subroutineName '(' parameterList ')' subroutineBody
- Check for constructor, function, method
- Check for 'void' or type (boolean, int, char, className)
- Check for subroutineName
- Initialize subroutine_symbol_table : local_count, argument_count, total_count = 0;
- currentSubroutineName = subroutineName; currentSubroutineType = (constructor, function,method)?
- If current subroutine is a method – add (this (name), argument (kind), className (type), 0 (index)) to subroutine_symbol_table. Increment argument_count and total_count
- Check for symbol '('
- Call compileParameterList() : will see <parameterList>
- Should see </parameterList>; Check for symbol ')'
- Call compileSubroutineBody() : will see <subroutineBody>
- Should see </subroutineBody>

SubroutineDec.

(Refer Slide Time: 02:57)

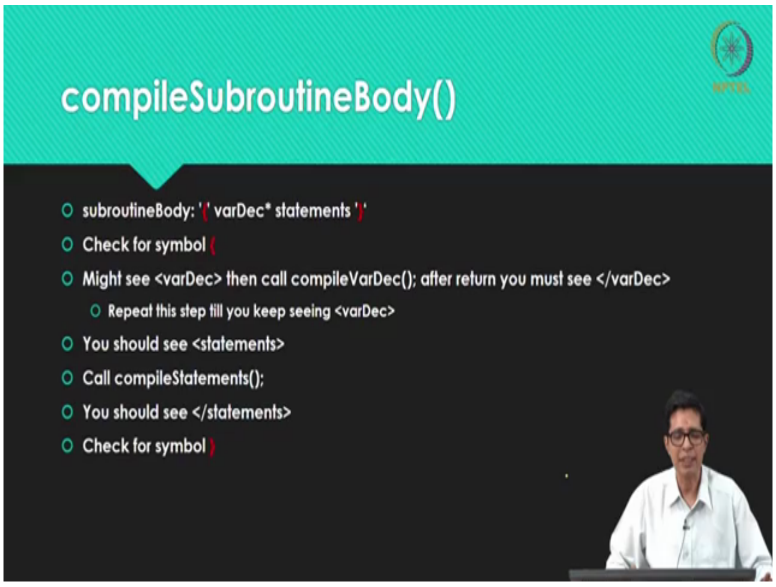


compileParameterList()

- parameterList: ((type varName (',' type varName)*)?
- Nothing may exist – we have subroutines with zero arguments
- Check for type and varname (identifier)
- Add (argument (kind), type, varname, argument_count (index)) into subroutine_symbol_table; increment argument_count and total_count
- Check for symbol ','
  - If yes, check for type and varname; Add (argument (kind), type, varname, argument_count (index)) into subroutine_symbol_table; increment argument_count and total_count
  - Repeat this step

ParameterList.

(Refer Slide Time: 02:58)

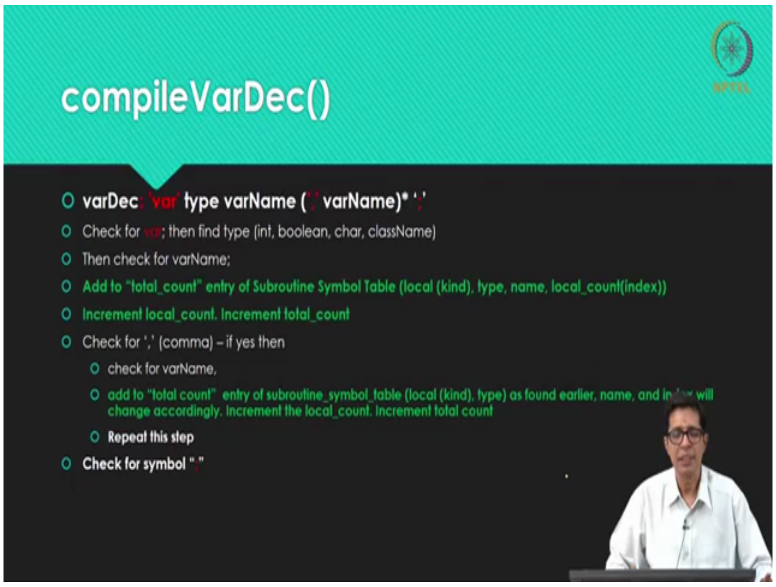


compileSubroutineBody()

- subroutineBody: '{' varDec* statements '}'
- Check for symbol {
- Might see <varDec> then call compileVarDec(); after return you must see </varDec>
  - Repeat this step till you keep seeing <varDec>
- You should see <statements>
- Call compileStatements();
- You should see </statements>
- Check for symbol }

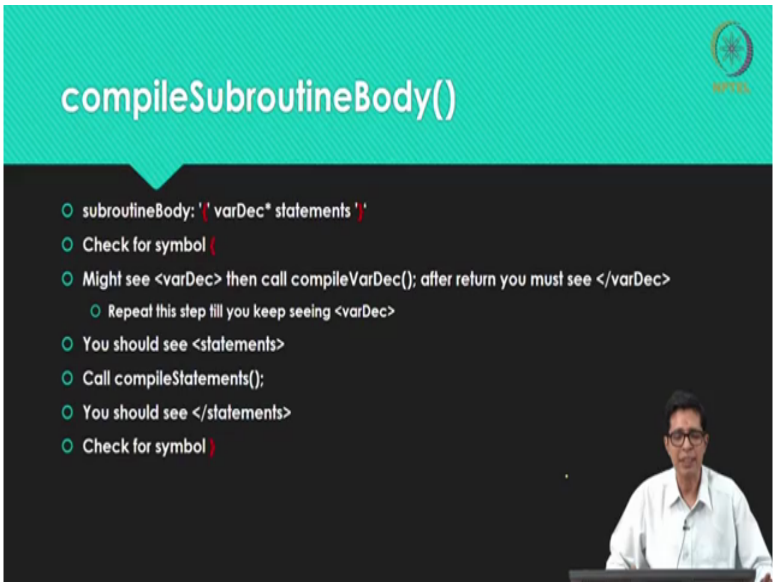
SubroutineBody.

(Refer Slide Time: 03:00)



compileVarDec()

- varDec: 'var' type varName (',' varName)* ';'
- Check for var; then find type (int, boolean, char, className)
- Then check for varName;
- Add to "total_count" entry of Subroutine Symbol Table (local (kind), type, name, local_count(index))
- Increment local_count. Increment total_count
- Check for ',' (comma) – if yes then
  - check for varName,
  - add to "total count" entry of subroutine_symbol_table (local (kind), type) as found earlier, name, and in[illegible] will change accordingly. Increment the local_count. Increment total count
  - Repeat this step
- Check for symbol ";"

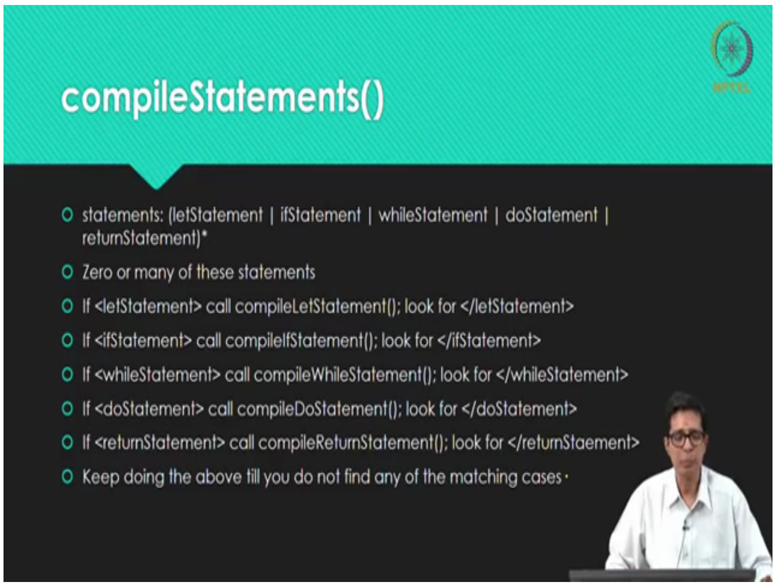


compileSubroutineBody()

- subroutineBody: '{' varDec* statements '}'
- Check for symbol {
- Might see <varDec> then call compileVarDec(); after return you must see </varDec>
  - Repeat this step till you keep seeing <varDec>
- You should see <statements>
- Call compileStatements();
- You should see </statements>
- Check for symbol }

VarDec, SubroutineBody again something little more on SubroutineBody.

(Refer Slide Time: 03:07)

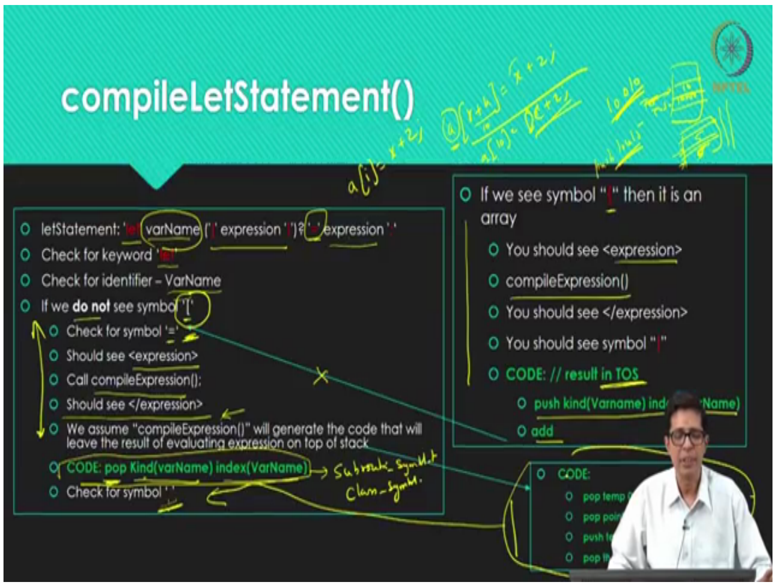


compileStatements()

- statements: (letStatement | ifStatement | whileStatement | doStatement | returnStatement)*
- Zero or many of these statements
- If <letStatement> call compileLetStatement(); look for </letStatement>
- If <ifStatement> call compileIfStatement(); look for </ifStatement>
- If <whileStatement> call compileWhileStatement(); look for </whileStatement>
- If <doStatement> call compileDoStatement(); look for </doStatement>
- If <returnStatement> call compileReturnStatement(); look for </returnStaement>
- Keep doing the above till you do not find any of the matching cases

Then compileStatements.

(Refer Slide Time: 03:10)



compileLetStatement()

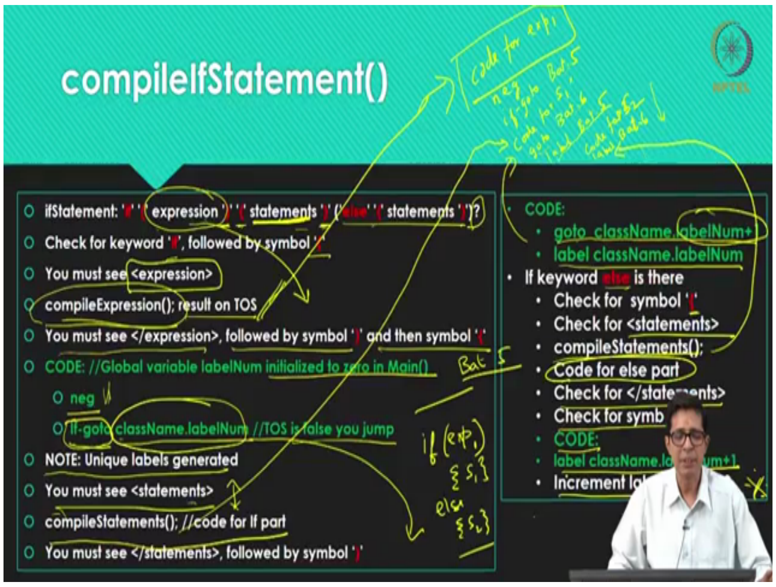
- letStatement: 'let' varName ('[' expression ']')? '=' expression ';'
- Check for keyword 'let'
- Check for identifier – VarName
- If we **do not** see symbol '['
  - Check for symbol '='
  - Should see <expression>
  - Call compileExpression();
  - Should see </expression>
  - We assume "compileExpression()" will generate the code that will leave the result of evaluating expression on top of stack
  - CODE: pop Kind(varName) index(VarName)
  - Check for symbol ';'

- If we see symbol "[" then it is an array
  - You should see <expression>
  - compileExpression()
  - You should see </expression>
  - You should see symbol "]"
  - CODE: // result in TOS
    - push kind(Varname) ind Name)
    - add

- CODE:
  - pop temp
  - pop poin
  - push t
  - pop th

compileLetStatement.

(Refer Slide Time: 03:12)



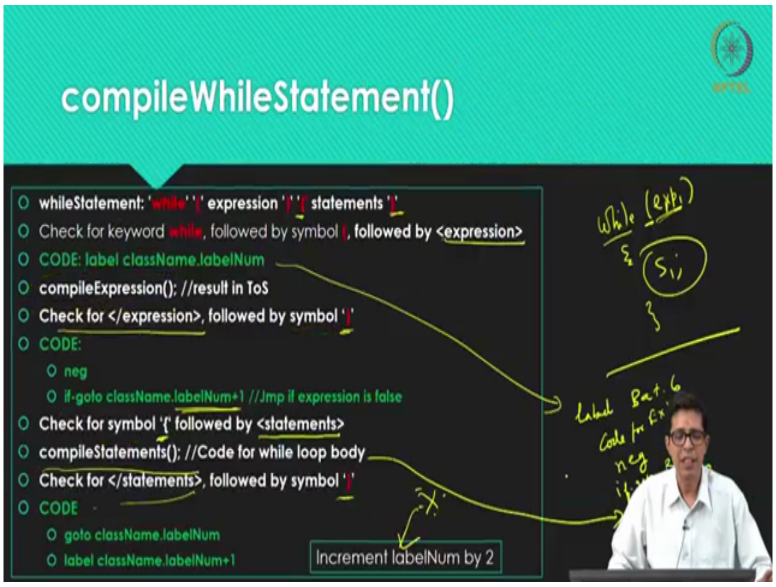
compileIfStatement()

- ifStatement: 'if' '(' expression ')' '{' statements '}' ('else' '{' statements '}')?
- Check for keyword 'if', followed by symbol '('
- You must see <expression>
- compileExpression(); result on TOS
- You must see </expression>, followed by symbol ')' and then symbol '{'
- CODE: //Global variable labelNum initialized to zero in Main()
  - neg
  - If-goto className.labelNum //TOS is false you jump
- NOTE: Unique labels generated
- You must see <statements>
- compileStatements(); //code for If part
- You must see </statements>, followed by symbol '}'

- CODE:
  - goto className.labelNum+1
  - label className.labelNum
- If keyword else is there
  - Check for symbol '{'
  - Check for <statements>
  - compileStatements();
  - Code for else part
  - Check for </state ents>
  - Check for symb
  - CODE:
  - label className.la um+1
  - Increment la

compileIfStatement.

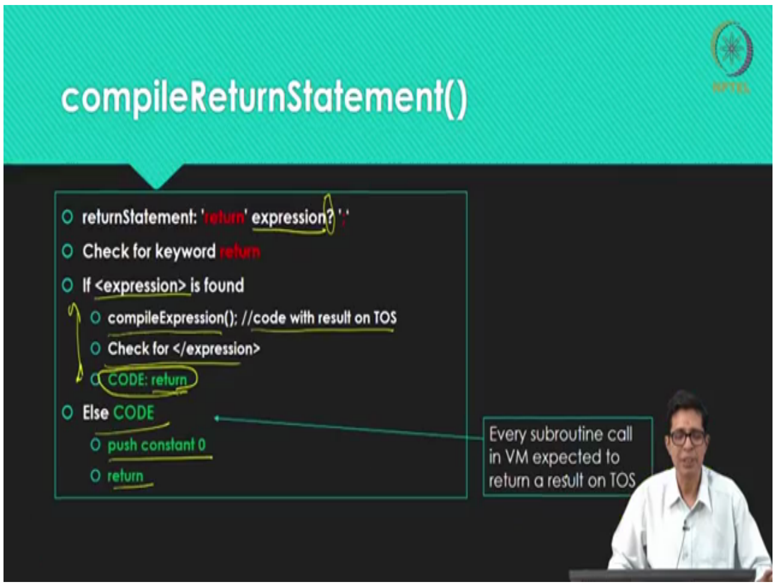
(Refer Slide Time: 03:13)



compileWhileStatement()

- whileStatement: 'while' '(' expression ')' '{' statements '}'
- Check for keyword while, followed by symbol (, followed by <expression>
- CODE: label className.labelNum
- compileExpression(); //result in ToS
- Check for </expression>, followed by symbol ')'
- CODE:
  - neg
  - if-goto className.labelNum+1 //Jmp if expression is false
- Check for symbol '{' followed by <statements>
- compileStatements(); //Code for while loop body
- Check for </statements>, followed by symbol '}'
- CODE
  - goto className.labelNum
  - label className.labelNum+1

Increment labelNum by 2



compileReturnStatement()

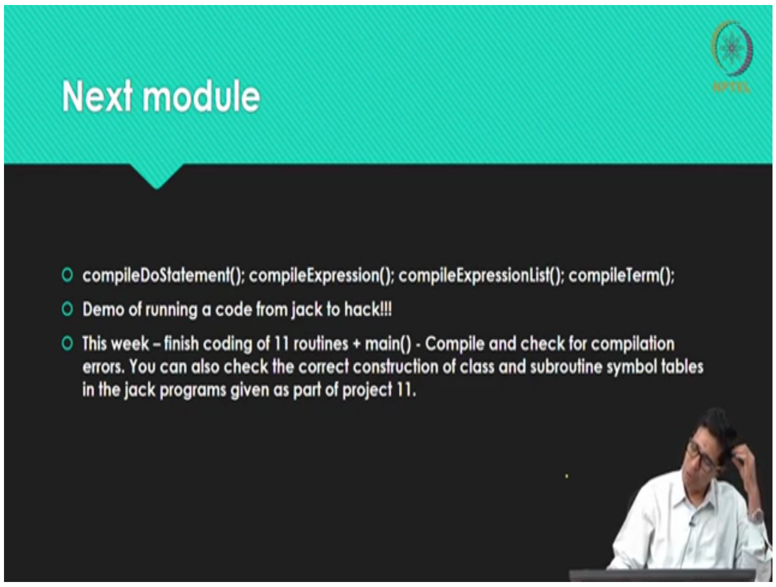
- returnStatement: 'return' expression? ';'
- Check for keyword return
- If <expression> is found
  - compileExpression(); //code with result on TOS
  - Check for </expression>
  - CODE: return
- Else CODE
  - push constant 0
  - return

Every subroutine call in VM expected to return a result on TOS

WhileStatement and ReturnStatement, so 11 such subroutines we have said what is the (contin) what should be done there and whatever we have put on green is what should go into an output file and that will be the dot VM file. So create a, so you take an XML file as an input and you generate a dot VM file as an output and whatever should go into the that VM file is basically whatever you are seeing in green plus of course some part which is dedicated for constructing your symbol tables namely your global class symbol table and your subroutine symbol table that happens in your class var declaration and your parameter list and your var declaration.

Three subroutines where who are responsible the classVarDec is responsible for all the class variables, your parameter list is responsible for all your arguments of your subroutine symbol table and your VarDec compileVarDec is responsible for your local variables there, right.

(Refer Slide Time: 04:25)



Next module

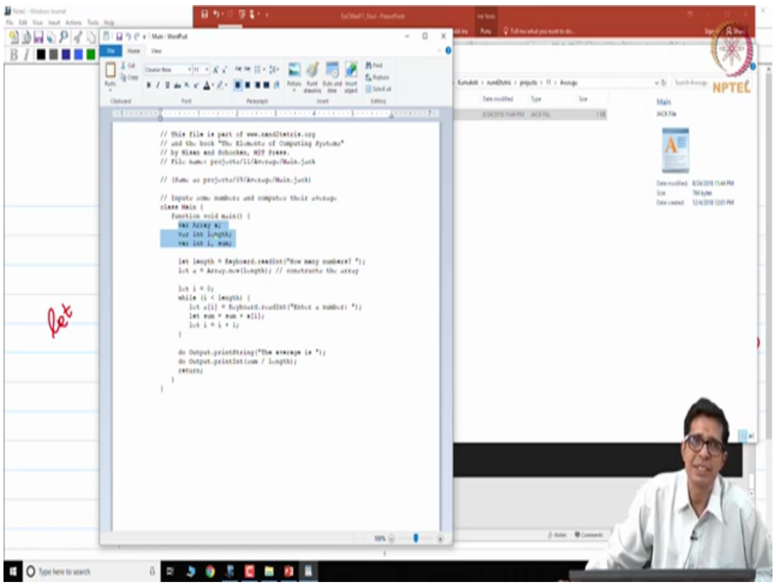
- compileDoStatement(); compileExpression(); compileExpressionList(); compileTerm();
- Demo of running a code from jack to hack!!!
- This week – finish coding of 11 routines + main() - Compile and check for compilation errors. You can also check the correct construction of class and subroutine symbol tables in the jack programs given as part of project 11.

So in the next module that is 12 we are left with only 4 subroutines namely compileDoStatement, expression, expression list and compileTerm but these are slightly involved you will, so you have seen maximum of 4 (co) 4 lines of code they involve involvement here would be a little more number of lines of code that we need to write for this and so we will finish those 4 in the next step starting of the next module and we will also give you a demo of running a code from jack to hack so, right, so the entire thing will see from jack to hack.

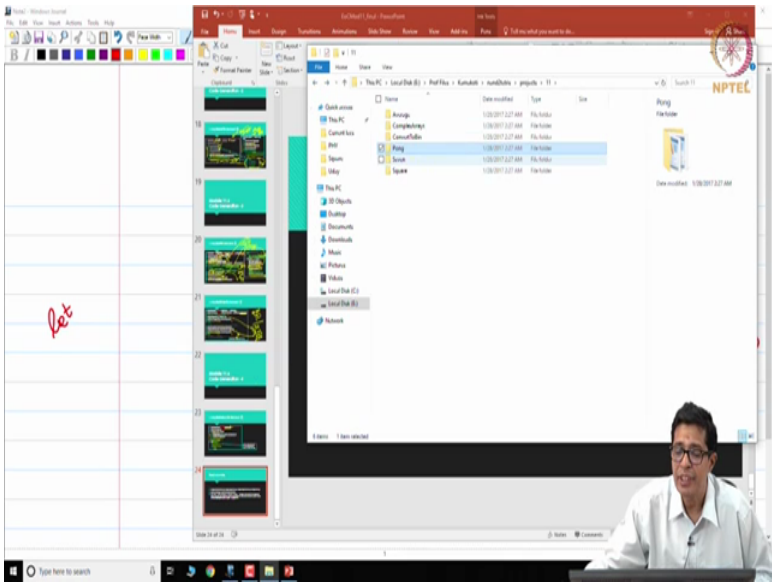
So what we are expected to do this is a very defining moment for you, you are writing a compiler on your own, so do not ditch this effort see and this module is very short and you have to just code what we have told right so and what we have understood, just keep coding, so finish the coding of the 11 routines here we will be just left with 4 more routines which we are going to discuss in the next module and plus of course the main right and you compile and check for any compilation errors, right I have just oriented it from a C programming point of view you could have written in C plus plus or Python or whatever but just check there is an there is no issue there in terms of compiler and you can also check the current construction of your class symbol table and at least your first (simp) subroutine symbol table.

So we will we may not cross the first subroutine but at least the first subroutine you are seeing in your class that symbol table you can check whether it is properly constructing you can print it out at correct position and see who this printing out, so and in project 11 you actually have a number of programs, right.

(Refer Slide Time: 06:21)



```
// This file is part of www.nand2tetris.org
// and the book "The Elements of Computing Systems"
// by Nisan and Schocken, MIT Press.
// File name: projects/11/Average/Main.jack

// (Same as projects/09/Average/Main.jack)

// Inputs some numbers and computes their average
class Main {
   function void main() {
     var Array a;
     var int length;
     var int i, sum;

     let length = Keyboard.readInt("How many numbers? ");
     let a = Array.new(length); // constructs the array

     let i = 0;
     while (i < length) {
        let a[i] = Keyboard.readInt("Enter a number: ");
        let sum = sum + a[i];
        let i = i + 1;
     }

     do Output.printString("The average is ");
     do Output.printInt(sum / length);
     return;
   }
}
```



So the you can basically see so if you go to nand2tetris and project 11, so you see a number of programs average, complex arrays, convert to bin, pong seven, square, so for all these things you can tokenize it then you can and a parse it to get the xml file and then start running this code for that and see whether the class symbol table for example if you take this main you can see whether this, so you will see there is no class variable here but you will see at least three variables being done here right, so there are three local variables.

So your subroutine your class symbol table should be null that is there should be no entry in their part your current subroutine table the first sub main has at least three entries, you can check whether this at least is formed there. So it is your symbol table properly constructed that much you can see right and that you can do for every jack file that you are seeing but there are for example pong has four jack files you can do for all the four jack files.

Similarly seven has one of course and then square has three jack files, so you can see for each of these Jack files so you have to first tokenize then create the XML file and then put your current code generator and see at least to this stage, so we will again meet in module 12 and we will take it forward, thank you very much.