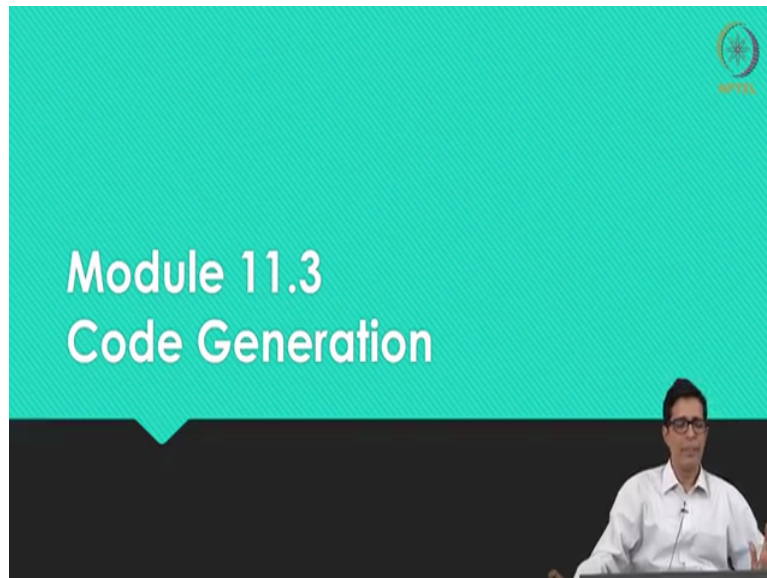


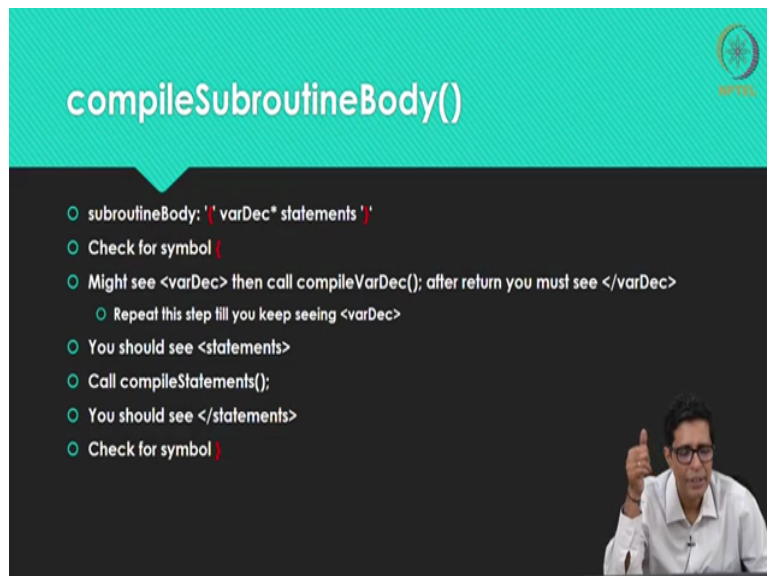
Foundations To Computer Systems Design
Professor V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology, Madras
Module 11.3
The Jack Compiler Backend: Code Generation - 1

(Refer Slide Time: 00:16)



Ok, welcome to module 11 point 3, in this we will now start seeing the code generation.

(Refer Slide Time: 00:23)



So what we have seen so far is when we the last time when we started compileSubroutineBody first it finish the compile varDec then it went to compile statements, right.

(Refer Slide Time: 00:45)

compileSubroutineBody()

- o subroutineBody: "varDec" statements "
- o Check for symbol
- o Might see <varDec> then call compileVarDec(); after return you must see </varDec>
 - o Repeat this step till you keep seeing <varDec>
- o CODE: function className.currentSubroutineName local_count
- o You need local_count to output the above
- o If current subroutine is "constructor" (Creating the object)
 - o CODE:
 - o push constant <field_count>
 - o Call Memory.alloc 1
 - o pop pointer 0
- o If current subroutine is "method" (initializing 'this')
 - o CODE:
 - o push argument 0
 - o pop pointer 0
- o You should see <statements>; Call compileStatements(); You should see </statements>
- o Check for symbol

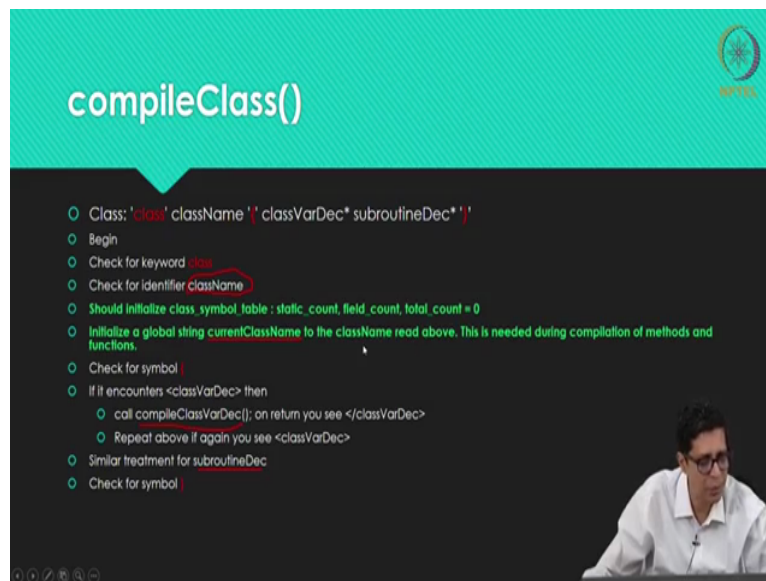
Finished VarDec and before entering compileStatements()

(A small video inset shows a man in a white shirt speaking.)

So what does happens, so we were in compileSubroutineBody, so we have finished of the all the variable declarations, right and then after we finish compile varDec then we said it has to do compile statements before it goes to compile statements, this compileSubroutineBody will do certain functionalities for generating the code. So what are those functionalities we will see, what the compileSubroutineBody will do.

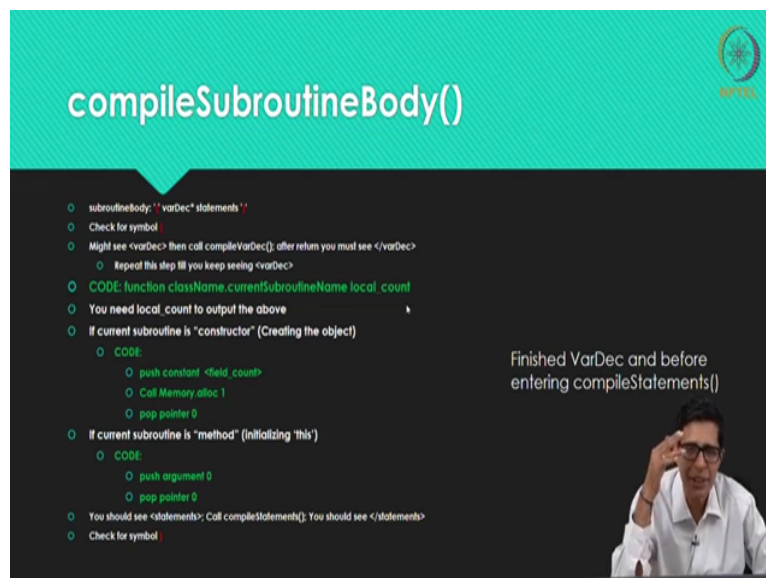
First thing is, it is needs to done the code it there is a VM code we should say function right, so function className dot currentSubroutineName local count, as I know the className as to current className, so the className has to be stored in a global variable I told you earlier we are going to use this, so this function is a from this className in this class and currentSubroutineName these are global variables that we already saved.

(Refer Slide Time: 01:49)



compileClass()

- Class: `'class' className '!' classVarDec* subroutineDec* '!'`
- Begin
- Check for keyword `class`
- Check for identifier `className`
- 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 methods and functions.
- 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



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>`
- CODE: `function className.currentSubroutineName local_count`
- You need `local_count` to output the above
- If current subroutine is "constructor" (Creating the object)
 - CODE:
 - push constant `<field_count>`
 - Call `Memory alloc 1`
 - pop pointer 0
- If current subroutine is "method" (initializing 'this')
 - CODE:
 - push argument 0
 - pop pointer 0
- You should see `<statements>`; Call `compileStatements()`; You should see `<statements>`
- Check for symbol

Finished VarDec and before entering `compileStatements()`

By `compileClass`, `compileClass` I have shown that initialize a global string `currentClassName` and assign the `className` to this, so that is what we are going to use here, right. So `className` dot `currentSubroutineName` which we have again accounted for and then `local count`, this is a total number of local variables there are that. So actually instead of `local count` we should say it should reach `field count`, sorry correct, the `local count`.

So we need to have this do you see the number of local variables that are there, so go back to VM implementation whenever you see a function immediately the VM actually creates `local count` number of variables, right so the function VM call if you go and repeat it actually uses the number of local variables. So now you should also understand why I am dumping this code here? Why are we doing it here? Because, so we need to know the number of local


variables before we output this function, this particular VM this is a VM code, we need to have the number of local variables and that is why we are doing it after your compileVarDec is finished, right.

So this is the first thing you generate now, right this is a so when you compile a class the first code that you will come is function some className of the first subroutine that you see followed by the number of local variables it has, ok. Now after this if the current subroutine is constructor right, this also we have noted somewhere here if you very carefully looked at, right.

(Refer Slide Time: 03:44)

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>



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>
- CODE: function className.currentSubroutineName local_count
- You need local_count to output the above
- If current subroutine is "constructor" (Creating the object)
 - CODE:
 - push constant <field_count>
 - Call Memory.alloc 1
 - pop pointer 0
- If current subroutine is "method" (initializing "this")
 - CODE:
 - push argument 0
 - pop pointer 0
- You should see <statements>; Call compileStatements(); You should see </statements>
- Check for symbol '!'

Finished VarDec and before entering compileStatements()



I have, sorry I have asked the saying currentSubroutineType is constructor, function or method please assign, right so depending on this. So why I need this is done by

compileSubroutineDec and these are all global variables why I need this currentSubroutineType I need it here, ok. So if the current subroutine is a constructor then I have to create that object, ok so you dump this code push constant field count, right the number field count call memory dot alloc 1, so memory dot alloc is an (0)(04:26), right.

So I am (ca) this is usually equivalent in the lock that takes only one argument basically this field count, so what will memory dot alloc do? It will taking one argument and that argument is field count, so it will create field count number of space, so all the field variables will be given a space and what will memory dot alloc return you? Memory dot alloc will return you a pointer it will on the top of the stack you will get a pointer pointing to this memory dot alloc whatever allocated memory but you pop pointer 0 that means you assign that base address to this segment pop pointer 0 will make the this segment point to this object, so you are created an object and you have made the this segment point to that.

So this is what you need to do moment you see that this is a constructor, so this memory dot alloc is basically used for this. So then if then next if the current subroutine is a method, right if it is the method then somebody is calling this so this the first 0th argument will be so this method will be associated with an object, right and the 0th argument will be the pointer to that object because this method will be using the field variables etcetera of that object, so the 0th argument as such we have seen will be the 0th argument will be the pointer to the object for which we are executing this particular subroutine.

So what you do is you push argument 0, so that pointer to the object will become a top of stack and I say pop pointer 0, so the current this segment will be assign to this object. So when the subroutine is executing whatever field variables we are see those field variables will be taken from the object that is associated with that we have seen in the previous thing then this we do after that we see statements, we will saw Call compileStatements and when it is returns we should see (scla) slash statements and then we check for symbol to this and that is where coompileSubroutineBody finishes, ok so this is basically the contribution of compile coompileSubroutineBody to this.

Now I want you to point whenever there is a constructor function, see I have two types of variables field variables and static variables but I am creating space only for the field variables here, right because all the statics variables will be in the static segment, right. Now always, so for the entire application the static variable should be put into the static segment it will not be part of the object it will be common to a class, right we have already defined.

Now there are multiple jack files that we will compile each will have static variables and then we now say that this when we are compiling this, this static variable is 0, 1, so for every jack file I am assigning the index 0, 1, 2 for the static variables into the other. So every class every jack file will have static variables with index 0, 1, and 2. So how do you resolve this? That is where you should remember what we have done on the virtual machine.

In the virtual machine whenever I say pop static 0 or something like that immediately a label is created with the file name dot that index. So if I have j a 1 dot jack and that has two static variables say s 1, s 2 that will become that will be assign 0 and 1, so that will become a 1 dot 0 will be s 1 and a 1 dot 1 will be s 2. Similarly if I have another j 2 dot jack in which I have some r 1, r 2 are static, j 1 dot 0 will be r 1 and j 1, sorry j 2 dot 0 will be r 1 and j 2 dot 1 will be r 2, right.

So this is how the VM takes the static things, right. So the VM the way VM handles static variables it will make it file independent, so every file can start with 0, 1, 2 the VM will label it in such a way that these static variables are mapped on to distinct memory locations, they have given different names and the assembler if you remember will mapped the static variables on to different locations in the static segment we starts from 16th to 255.

So we need to understand how the VM works for us to start dumping the coding that is why we say that if you are just an application programmer you do not nothing about anything below that if you are just a compiler writer and you do not know anything below that then basically you are losing something. So the basic intention for this course is to give you the overall picture.

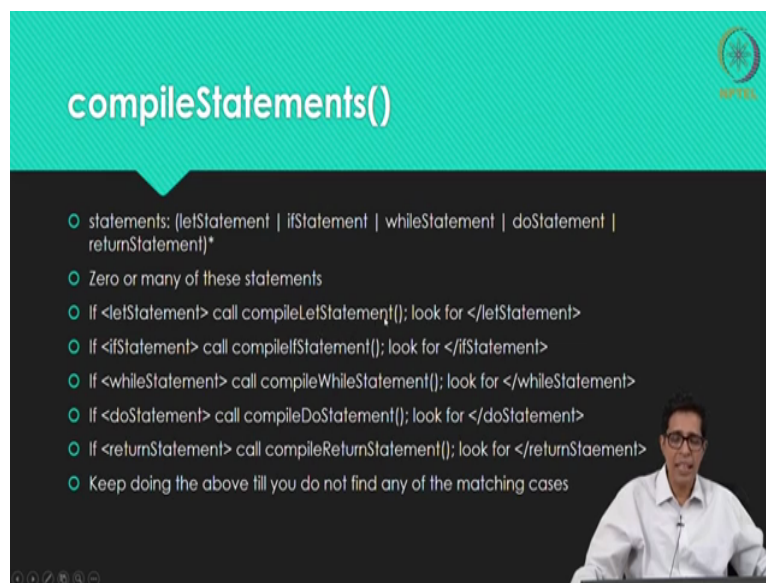
So what I have what I am basically telling you here is that when we are seeing a constructor we are allocating space only for the field variables there, so and for all the static variable will be assigned somewhere in the static segment and every file every jack file I compile the static variable should be assign index 0, 1, 2. So there will be multiple jack files and each one will have static variables is index 0, 1 and 2.

Now and I will just say push static 0, push static 0 in both cases whenever I want to access that variable. Now the VM will take care of labeling these static variables of different files in different manner and since they are labeled differently the static variables there will be allocated different locations in the memory. So this is these whole burden of handling static

has been carefully push to the virtual machine and that is one basic thing that you need to understand otherwise this whole thing would have become much more chaotic, ok.

So that is one thing that, so this is also trying to answer the question that if you want to become a great compiler writer you should also know how the virtual machine works, right. So this is what subroutine body does and, so then the next thing is that we will go to, so it will see statements, so and so it will call compile statements.

(Refer Slide Time: 11:12)



The slide features a teal header with the title 'compileStatements()' and a small logo in the top right corner. Below the header, a list of statements is provided, each with a corresponding compile method and a note on what to look for. A small video inset in the bottom right shows a man speaking.

- 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 </returnStatement>
- Keep doing the above till you do not find any of the matching cases

So what will compile statements do? Compile statements is zero or more of these letStatement, ifStatement, whileStatement, doStatement, returnStatement zero or more of these. So if the compile statements see letStatement it will call compileLetStatement and after finishing it will look for slash letStatement, if you see ifStatement it will call compileIfStatement and after compileIfStatement returns back it returns back to compileStatements it will look for slash ifStatement.

Similarly while, do return and it will be keep doing the same thing above till you do not find any of these letStatement or ifStatement or whileStatement, doStatement etcetera, right and that is what compileStatements does. So compileStatements basically takes you to the corresponding statement routine it does not directly contributes, so I such you do not see a green line here, it does not directly contribute to the code, right to the code generation.

So this is one part of, so what we will do is for all the fifteen subroutines that we have seen so far, we will basically start writing the code for them, right so that is what. So we have seen already we have seen some six of them, so in the remaining lectures of this module and the

subsequent module we will be basically explaining you what is the structure of each of these subroutine and if you code exactly what we are say ultimately the compiler will come back, right.

(Refer Slide Time: 12:52)

The slide titled "compileStatements()" contains the following text:

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

The slide titled "compileSubroutineBody()" contains the following text:

- 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

So now before we wind up his particular module I will just take you here, so this is the subroutine body, so the subroutine body will first see varDec, right so when you just open this varDec it is var int i comma j, right then the next varDec would be var string s, the next varDec is var array a, so all varDec three varDec are over and each of these varDec this varDec will put i and j as local variable between this as 0 and 1 and this varDec will put s as a local variable with type string index 2 and this will make a as a of type array local variable kind local and 3, ok.

So this is what slash varDec would do after varDec finishes as I told you, you will see statements and these statements can be you have an ifStatement then you have returnStatement, so you have an ifStatement and then a returnStatement then statement sent then, right. So this is how compileStatements work, so you see statements and then you see ifStatement, returnStatement and then you do not see anything then ultimately you come back to compileStatements and you should see a slash compileStatements, right.

So in the next lecture we will go on to more of these subroutines and we will finish it. So what I want you to do is to just take down these slides I have described here, take those notes and start writing each of these subroutines as we are seeing here and every time you will find for example and say in these subroutine body check for symbol this one, if you do not find the symbol so it will be symbol parentheses slash symbol, so that is the it is the something like this, right.

So symbol parentheses slash symbol, so if you do not find this you write you put an error, this particular thing is missing, so we were expecting this but this is missing. So that is how, so importantly one of the important thing in compiler is that when there is an error you should point back to the programmer saying here is the error, right. So that type of an error resilience you need to build inside the compiler and that is going to be a very important job for this.

So I want you to start coding and so you can come up to compileStatements at least you can start populating your symbol table, so we will meet in the next module with more details, thank you very much.