Foundations To Computer Systems Design

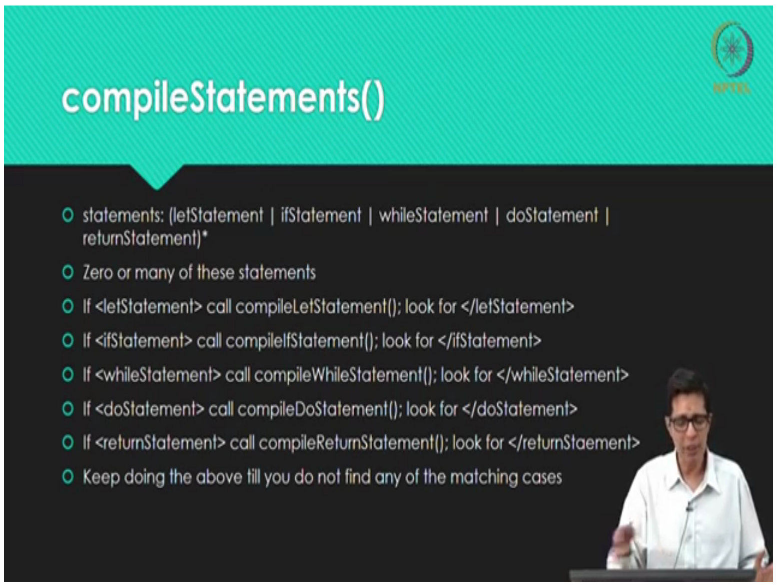
Professor V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Module 11.4

The Jack Compiler Backend: Code Generation - 2

So welcome to module 11 point 4, this is code generation 2.

(Refer Slide Time: 00:25)



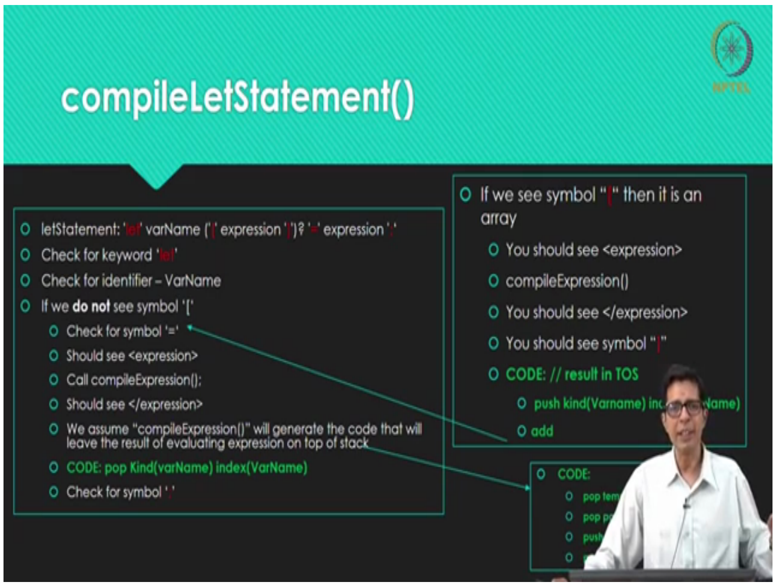
compileStatements()

- 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

What we have seen in the end of the previous module was about compileStatements where compileStatements will be a collection of five different statements 0 or many of these five statements let, if, while, do, return and so it will just a keep looking for these statements and if those statements if you see (())(00:44) rule the statement then compileLetStatement would be called if you see ifStatement and then (ifstay) compileIfStatement would be call and so on so forth.

Now in this module we will see in this module and subsequent modules we will see how compileLetStatement is code that, ifStatement is code that etcetera.

(Refer Slide Time: 01:05)



compileLetStatement()

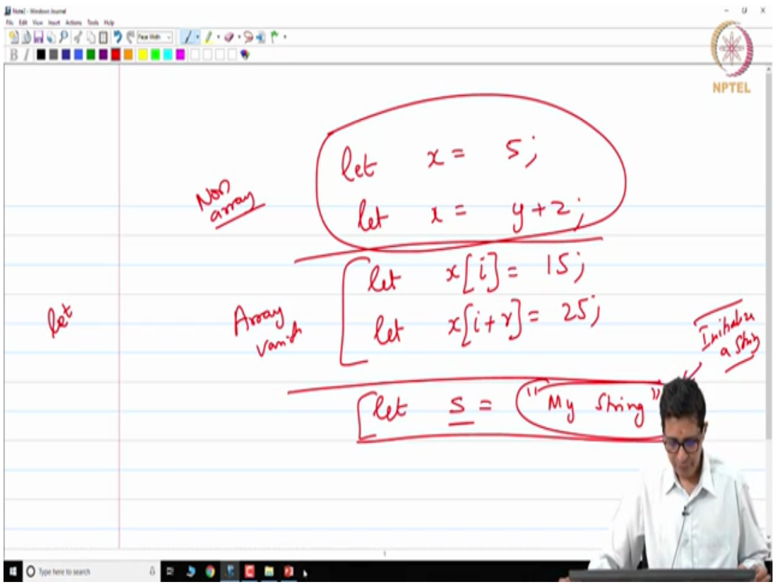
- letStatement: 'let' varName ('[' expression ']')? '=' expression ';'
- Check for keyword 'let'
- Check for identifier – VarName
- If we **do not** see symbol '['
  - Check for symbol '='
  - Should see <expression>
  - Call compileExpression();
  - Should see </expression>
  - We assume "compileExpression()" 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) inc Name)
    - add

- CODE:
  - pop tem
  - pop p
  - push

We started compileLetStatement, so letStatement is of two types.

(Refer Slide Time: 01:14)



let

Non array

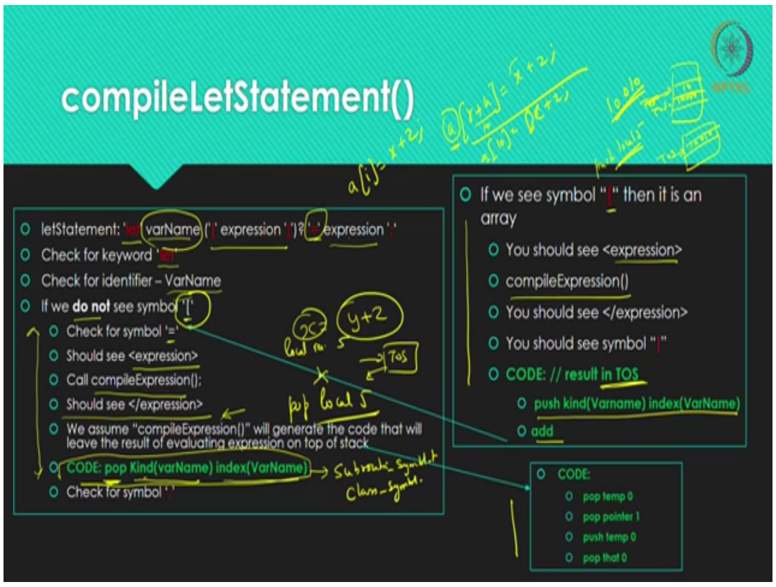
```
let x = 5;
let x = y+z;
```

Array variant

```
let x[i] = 15;
let x[i+y] = 25;
```

```
let s = "My String"
```

Initialize a String

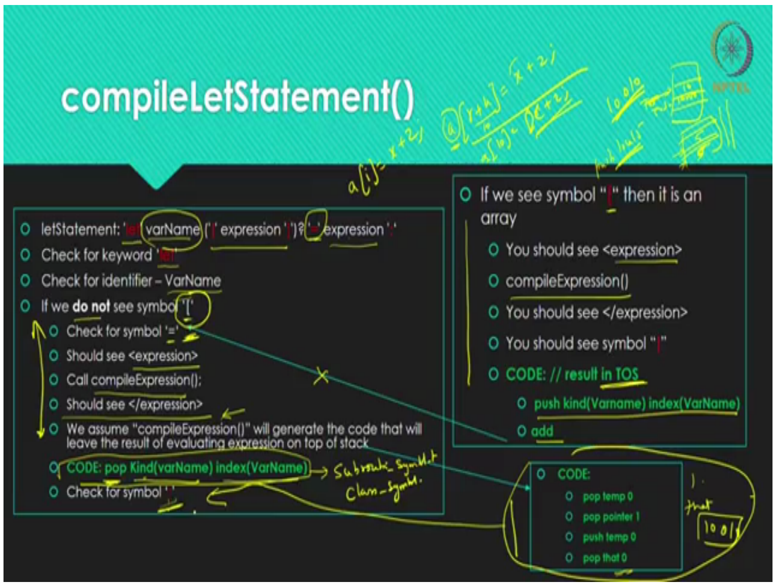


compileLetStatement()

- 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) index(VarName)
    - add

- CODE:
  - pop temp 0
  - pop pointer 1
  - push temp 0
  - pop that 0



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) index(VarName)
    - add

- CODE:
  - pop temp 0
  - pop pointer 1
  - push temp 0
  - pop that 0

So it can be of the form let x is equal to 5 or let x is equal to y plus z, it can also be of the form let x of i is equal to 15 or let x of i plus r is equal to 25, right it can also be of the form let s is equal to My String, right so these are all non-array variables where these are array variables of course this is a new type of array, string is a character array and you can use these type of statements to initialize the string, right we have seen all these examples in the past, right.

So essentially let will be of as you have seen here let will be of the form, so there is a let here followed by a varName, right so how we will be code the letStatement first we will see let we will just take check for the keyword let then we will have an identifier that is a varName then we will see whether there is this symbol square bracket or not, if the symbol square bracket is not there then this is the code that we will execute, if the (singer) symbol square bracket is there then these two are the code that we will execute when the symbol square bracket is not there then it is a very simple thing like let x is equal to y plus z etcetera.

So then immediately we will go and if the symbol square bracket is not there we will go and check for symbol equal to, right, so we have come to this stage of this rule then we will go and check then you will should see this expression, right as you have token next root then the moment you will see expression as (())(03:44), so you will see xxx I have to call compile xxx, so I will call compileExpression.

So compileExpression will give me two things, it will generate the code that is needed for you know implementing that expression and also that code will also ensure that the result of that evaluation of the expression is on the top of the stack. So we will assume now and that is what will happen as we will see compileExpression later, compileExpression will generate the code that will leave that will generate the code for that entire expression and also ensure that the result is on top of the stack.

So after you reach this point from the code generation point of view (compile generate) compileExpression would have generated all the code and when you reach this point the result of that execution of that expression is on the top of the stack and now that has to be, so as suppose I have x is equal to y plus z, y plus z evaluation would have been done and the result of y plus z will be on the top of the stack.

Now I have to pop this top of the stack on to whatever variable name is derived, so I have to put kind of variable name index of variable name, suppose it is say for example it is say local variable number 5, so I should say pop local 5, so the whatever evaluation of this, so if you are access local variable 5, if I say pop local 5 then whatever is the evaluated value of y plus z will get assign to pop local 5 which is x.

So this is the, so when I see compileLetStatement in which there is no array on the right left hand side then the only line of code that I need to add is after you finish this compileExpression slash expression etcetera at this point where your expression is actually already evaluated, the code for that is already dumped you have to put pop the kind of this variable name and the index of this variable name where will you get the kind of index? First you have to go and search in the subroutine symbol table, the current subroutine symbol table and if you do not find in that then go and check in the class symbol table.

So you give these varName as an input search the (cla) subroutine symbol table find out if that varName is matching with some of the varName is entered that and that then you know the kind and the index otherwise go back to the class subroutine table and find out if it is a class variable meaning it is a field or a static variable, right. So this is some this is how you get this kind and index, right.

Suppose you will see this square bracket then it is an array, so you will have something like a of i is equal to x plus z, so what we see so once you see this so this need not be just i it can be a of r plus k is equal to x plus z, right. So you have seen that variable name var where VarName will be a and what is a stands for? a stands for this starting location of that array. Now what will r plus k now gives? When I evaluate this expression this will give me a value which will be the index into that array, so if so a itself will be the starting address of that array and to that if I add the evaluated value of r plus k I will get the exact location in the memory where I have to store x plus z, right.

So this is comes to 10 then a of 10 is equal to x plus z that is what I need to do, so if a is at location 10 thousand, so at 10 thousand 10 I need to store the value of x plus z, right this is what. So what happens first I will have seen a here first I have seen let then I have seen a, a is the varName. Now I see this array expression, the moment I see this square brackets then I should see expression as a token the moment I see expression as per our understanding we will call compileExpression what will compileExpression do? compileExpression will generate the code for evaluating that expression and on execution of the code it also ensures that the result of that evaluation is on the top of the stack.

So on the top of the stack, now suppose let us say a r plus k, r plus k is 10 on the top of the stack 10 will be there, right now after this compileExpression code for compileExpression finishes then executing the 10 will be on the top of the stack. Now you should see slash expression, right and then you should see the close of that symbol, right this is all done. Now the next thing is so the result 10 is in the top of stack as you see here, now I have to push the kind and index of that variable, so a so if I say pull so a is say some local argument 5, so I then I will say now push local 5 what I am pushing here? Suppose a is started to store at 10 thousand I would have push 10 thousand into the stack here.

So I have pushed 10 thousand into the stack here, this is what this will do then I say add, so now on the top of the stack we will have 10 thousand 10 this is on the top of the stack and that is where you go back to this, so this particular arrow that you see I am just erasing when you have of this so that becomes clear, so after finishing this 10 thousand 10 I go back here, now I look for equal to then again I evaluate the expression whatever this.

So suppose I have evaluate x plus z and the x plus z turns out to be 5, so the answer of that evaluation of that expression is now push onto the stack, right. So it is on the top of the stack, so when I evaluate this expression x plus z this compileExpression is going to be called and end of that I should see slash expression that is I am doing on the right hand side of this. Now the compileExpression will ensure that if suppose x plus z is 5, 5 is on the top of the stack.

So that is this part of the code again, right we are executing this part of the code then after you finish here again if it is an array expression you come back here now what we do here there is 5 and 10 thousand 10 that 5 has to get stored in 10 thousand 10 as you see here 5 has to get stored in 10 thousand 10 how do you do? You pop temp 0, so into that temp which is a temporary stack your 5 will be going inside then pop pointer 1 when 5 goes off to top of the stack would be 10 thousand 10, so pop pointer 1, so your that segment will now get initialize to 10 thousand 10, now you push temp 0, so again 5 gets onto the top of the stack, so now you pop that 5 onto that comma zero that means 10 thousand 10 comma zeroes 10 thousand 10, so 10 thousand will get the value 5.

So this we are explain the theoretically in the previous module, module 10 now just we are seeing how it is getting implemented. So to sum up compileLetStatement works like this you first check for let, you check for variable name then if you do not see a square parentheses square bracket then you just check for symbol equal to then check for expression, so you should see expression token then call compileExpression then you should see now slash expression ending, right.

And then, right then you have to go and you write this code pop kind of varName and index of varName and value you will get the kind and index of this varName, varName is what you saw on the left, ok you will give you would search a subroutine symbol table followed by the class symbol table to get this and then finally checks for symbol semicolon but in this case if you are seeing a symbol there, if you are seeing a square bracket there then you after the square bracket you should see a expression then you call compileExpression then you should see slash expression then you see the end of the that square bracket then you write this code push kind VarName index VarName add then you come back to this part of the code again then you check for symbol equal to then you should see expression.

Now you are on the left right evaluating right hand side, you will call compileExpression, now you should see slash expression after that you go back to this part of the code whatever I am marking here on your right hand side bottom, you write pop temp 0, pop pointer 1, push temp 0, pop that 0 the come back here to check for the symbol semicolon, the semicolon should be checked again here. So this is how the entire compileLetStatement is coded.