Foundations To Computer Systems Design

Professor V Kamakoti

Department of Computer Science and Engineering

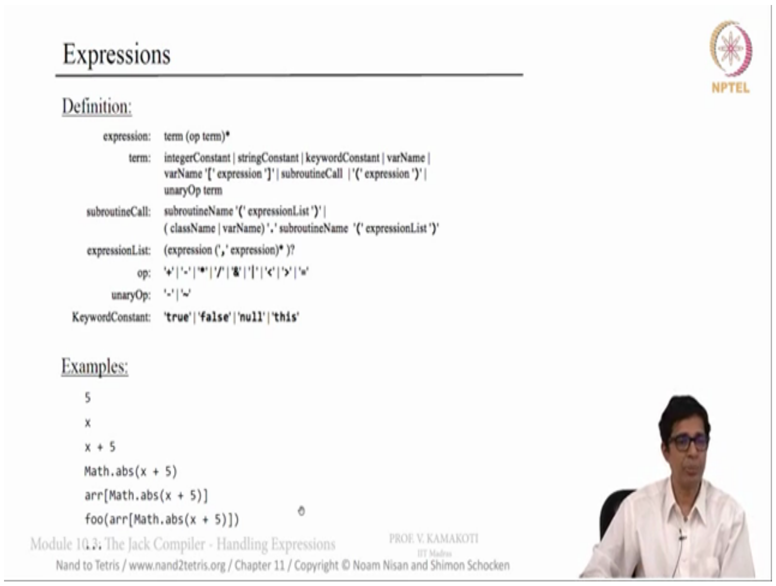
Indian Institute of Technology, Madras

Module 10.3

The Jack Compiler – Handling Expressions

So welcome to module 10.3, we are doing the Jack compiler back-end, so we have seen so far how to handle variables now we will now see of how to handle expressions.

(Refer Slide Time: 00:38)



Expressions

Definition:

```
       expression:  term (op term)*
             term:  integerConstant | stringConstant | keywordConstant | varName |
                    varName '[' expression ']' | subroutineCall | '(' expression ')' |
                    unaryOp term
   subroutineCall:  subroutineName '(' expressionList ')' |
                    ( className | varName) '.' subroutineName '(' expressionList ')'
   expressionList:  (expression (',' expression)* )?
               op:  '+' | '-' | '*' | '/' | '&' | '|' | '<' | '>' | '='
          unaryOp:  '-' | '~'
  KeywordConstant:  'true' | 'false' | 'null' | 'this'
```

Examples:

```
5
x
x + 5
Math.abs(x + 5)
arr[Math.abs(x + 5)]
foo(arr[Math.abs(x + 5)])
```

Module 10.3: The Jack Compiler - Handling Expressions   PROF. V. KAMAKOTI IIT Madras

Nand to Tetris / www.nand2tetris.org / Chapter 11 / Copyright © Noam Nisan and Shimon Schocken

Now well we start looking at expressions, we need to start looking at the grammar of how expressions are evolved at in this case, the expression is basically we have a term followed by an optional operator term pattern right, so this is how we define the first grammar right, so expression will have a term followed by an optional, if I put star it can be zero or more repetition so it can be just term or term of term, a term of term of term and it can go on like that.

And what is it term, a term can be just an integer constant like 150, 200, it can be a string constant like double quote something, it can be a keyword constant so what are all the keyword constants you can see it can be true, false, null or this right the last thing that you see here, true, false, null are this right, this can be a keyword constant it can be just an identifier right, I, c equal to I plus J, I is a term right, it could it can be an array expression, a of 10, a of 10 as you see here right, so a with followed by some expression here, a of 10.

It can be a subroutine call and the subroutine call will be just a subroutine name with an expression list or it can be a subroutine name dot sub, it can be a class name dot something or a var name dot something right, so just a subroutine name with expression means it belongs to the same class otherwise this belongs to some other class or so that can be this and then of course there is a subroutine call and then it can be an expression within parentheses or it can be an unary operator and a term and there are two unary operators, one is minus and another is negative right.

And what is an expression list, it will be expression, expression multiple times right and there can be subroutines without any expression also right, so these are all the things so let us give some examples of term, 5 is a term, x is a term, x plus 5 is actually a term right, x plus 5 is also a term right, it is term of term, x is a term, 5 is a term, plus is an operator, x plus 5 is a expression sorry, x plus 5 is an expression.

Now if we look at subroutine called Math, Math is a class name, if you look at this Math, dot abs, abs is a subroutine name absolute value, then there is an expression list right, this expression list here what you see here is an expression list, so the expression list here has only one expression and that name is x plus 5 so expression list has one expression and nothing more and that is x plus 5.

No I could have another term would be var name, var name is arr and then followed by this square brackets ending with the square brackets and within that I could have an expression and what is that expression here, the expression can be a term and what is a term, term can in addition be a subroutine call and the subroutine call in addition can derive to this class name dot subroutine name, Math dot abs of x plus 5.

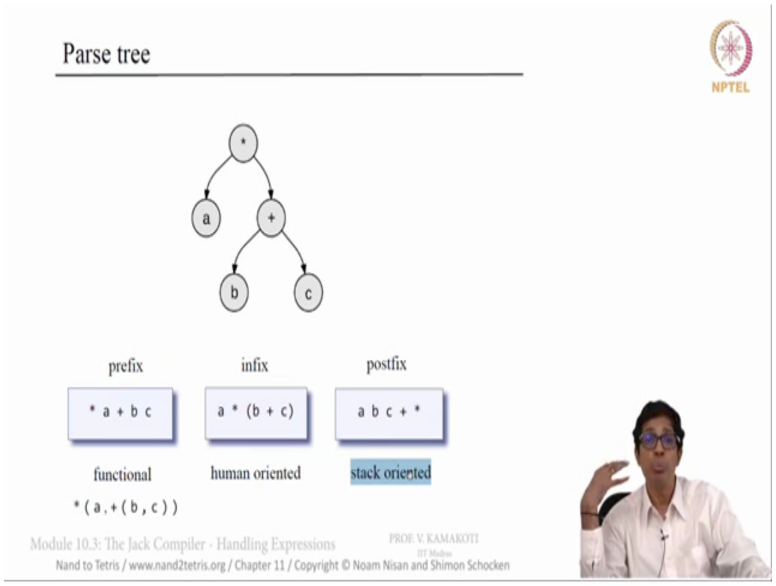
So this arr of this is actually a term where in r is, this r of this arr is var name followed by the square and ending with a square parenthesis and my Math dot abs and then var name and this Math dot abs of x plus y is an expression because this expression has one term and that term in turn is a subroutine call and that subroutine call in turn is a class name dot subroutine name math dot abs followed by an expression list, the expression list is an expression which has a term followed by op, followed by a term and that term is var name identifier x, this op is plus as you see here up can be anything here, op is plus and the next term is a integer constant which is 5 okay.

Now we can also have a subroutine call who is a subroutine name followed by an expression list followed by this parentheses, small parentheses and n parentheses as you see here, so this is a subroutine call and followed by an expression list, the expression list has one expression followed by, expression list can have an expression followed by many expressions, in this case I have only one expression that expression essentially has one term to start with and that term essentially is a bad name followed by a parenthesis.

So arr is the word name followed by parenthesis, followed by a square bracket ending with a square bracket and inside that there is an expression and that new expression is we have subroutine call because that new expression has a term that term essentially has a subroutine called, the term actually has a subroutine call and that subroutine call has a class name which is Math followed by a subroutine name dot abs and again there is an expression list that expression list has one expression and that expression has a term which is x and the term is an identifier var name with x and then op which is operand with operator with plus and another term which is an integer constant which is 5 right.

So this is the way we need to start interpreting expressions, so the grammar per se will capture all these expressions in this manner right, so try and understand this in this perspective.

(Refer Slide Time: 07:25)



Parse tree

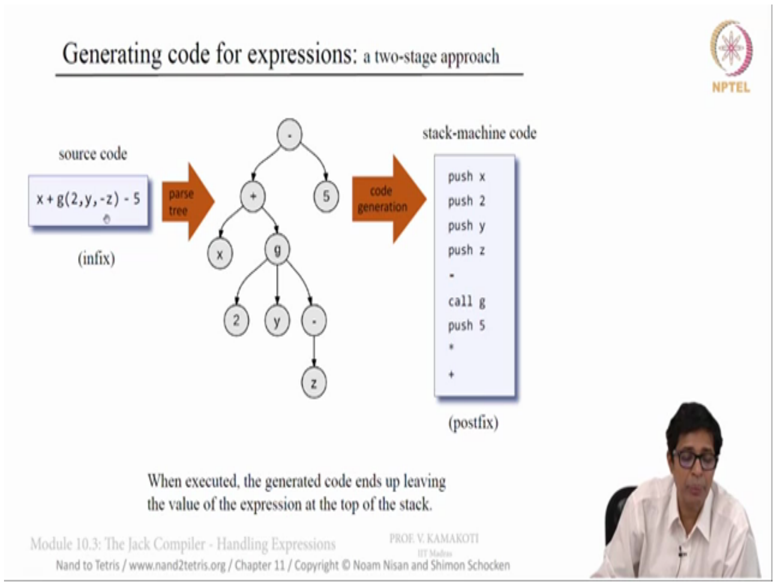
prefix	infix	postfix
* a + b c	a * (b + c)	a b c + *
functional	human oriented	stack oriented
*(a,+(b,c))		

Module 10.3: The Jack Compiler - Handling Expressions
PROF. V. KAMAKOTI IIT Madras
Nand to Tetris / www.nand2tetris.org / Chapter 11 / Copyright © Noam Nisan and Shimon Schocken

Now I want to do a into b plus c right, the infix is a into b plus c but we will not do the prefix here, prefixes I put the operand before, I am doing something called postfix which we have already seen on expression evaluation when we are doing the virtual machine, so what we do here we just put a b c plus star, so this means essentially push a, push b, push c then when you encounter a plus what you do, you do b plus c you just take the top two elements in the stack and add them and store the results so at the end after I see the plus what will be on the stack would be a followed by B plus c.

b plus c on the top and a in the bottom then I have star which will basically give you a into b plus c, so this is the stack oriented stuff that we need to do, so given an expression of any path, we need to convert it into this postfix notation and that is going to become extremely important, so remember when we are handling expressions we have to handle postfix notation.

(Refer Slide Time: 08:46)


Generating code for expressions: a two-stage approach
source code
x + g(2,y,-z) - 5
(infix)
parse tree
code generation
stack-machine code
(postfix)
NPTEL
Module 10.3: The Jack Compiler - Handling Expressions
PROF. V. KAMAKOTI
IIT Madras
Nand to Tetris / www.nand2tetris.org / Chapter 11 / Copyright © Noam Nisan and Shimon Schocken

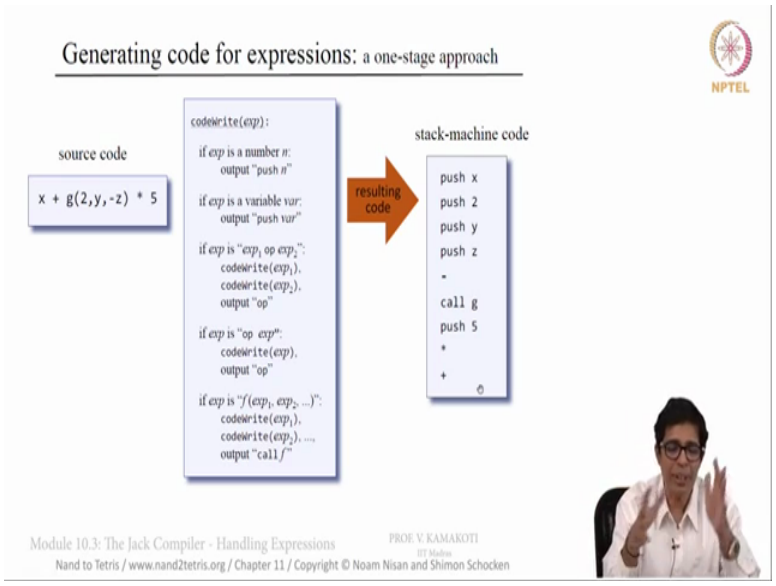
```
push x
push 2
push y
push z
-
call g
push 5
*
+
```

When executed, the generated code ends up leaving the value of the expression at the top of the stack.

So essentially if I have x plus g of 2 into 2, y, minus z minus 5, g is a function call x and 5 variables so this is how we generate this parse tree, we will just have this x plus g of 2, y, is at minus, so x plus g of 2, y, minus z minus 5 right, so when I want to evaluate this I put push x then I say push 2, push y, push z then negate z minus then called g, so when I am calling a function already I assume that all the arguments for that function would be already in the stack which is anyway in the stack now (())(09:40) and I say call g and then what we say is once call g completes, the result of this g of 2, y, minus z will be on the stack.

So when I say call g, at the end of this call g, I know that this 2 y z are all going to be removed and you know and the result 2 y z which are, 2 y minus z which are part of the top of the stack are removed and only X will be on the stack and at after call g finishes the result of g of 2, y, minus z will be on the stack.

(Refer Slide Time: 10:20)



Generating code for expressions: a one-stage approach

source code

```
x + g(2,y,-z) * 5
```

```
codeWrite(exp):
  if exp is a number n:
      output "push n"
  if exp is a variable var:
      output "push var"
  if exp is "exp1 op exp2":
      codeWrite(exp1),
      codeWrite(exp2),
      output "op"
  if exp is "op exp":
      codeWrite(exp),
      output "op"
  if exp is "f(exp1, exp2, ...)":
      codeWrite(exp1),
      codeWrite(exp2), ...,
      output "call f"
```

resulting code

stack-machine code

```
push x
push 2
push y
push z
-
call g
push 5
*
+
```

Module 10.3: The Jack Compiler - Handling Expressions
PROF. V. KAMAKOTI
IIT Madras
Nand to Tetris / www.nand2tetris.org / Chapter 11 / Copyright © Noam Nisan and Shimon Schocken

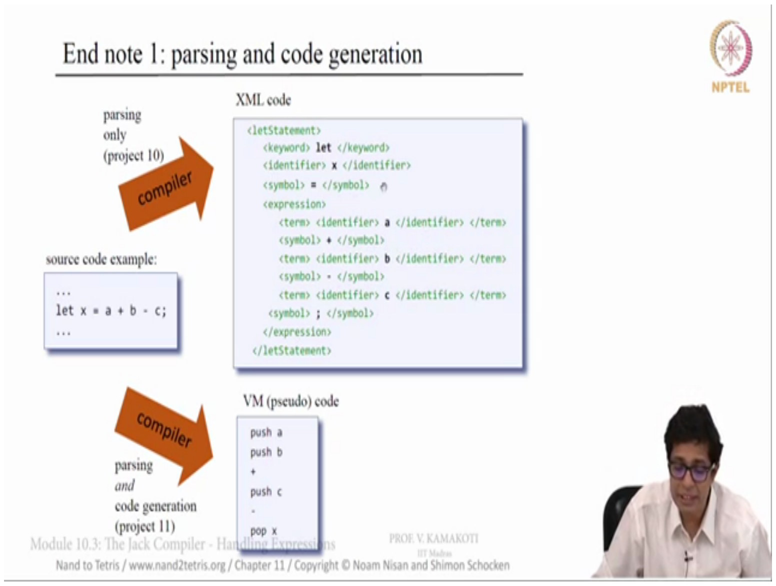
Okay if this is the source code x plus g of 2, y, minus z, that star 5 then this is how this will be push x, push 2, push y, push z, negate, it will become minus z, then call g, then the answer for that will be available now I push 5 and do star, so whatever the result of this g 2, y, z gets multiplied with 5 and that (())(10:49) will be on the top, now I do plus that gets into x plus this so this is how we actually start generating code for expressions.

So whatever I have described here is you want, we want an infix so this is how the code writing we need to do if the expression is a number n we need to push n, if the expression is a variable pushed at variable whatever whether it is a field or static or local or variable, if the expression is of the form exp one op exp two, first you finish writing the code for exp one then for exp 2 and output op, so this basically takes care of your post fix notation and if the expression is of the form op x exp then actually code right exp and then output op okay.

So this will, so code write exp will make the answer of that expression on the top of the stack and if I say op that whatever unary operation will happen on the top, is exp is function of exp 1 to exp 2 to etc. we write code rate of exp 1 we do code write of exp 2 and then at the output we just say call if right so what we essentially mean is that push all the arguments of the function f and then do a call f okay.

So this is how we start writing code and the most important thing that we need to keep up here is about this which will take care of here postfix notation and why do we need for fixed notation because then only I can implement it directly on the stack, so when I am writing a code for expression evaluation that should, since the VM is a stack based virtual machine everything would be mapped down to stack.

(Refer Slide Time: 12:58)



End note 1: parsing and code generation

parsing only (project 10)

compiler

source code example:

```
...
let x = a + b - c;
...
```

XML code

```
<letStatement>
   <keyword> let </keyword>
   <identifier> x </identifier>
   <symbol> = </symbol>
   <expression>
      <term> <identifier> a </identifier> </term>
      <symbol> + </symbol>
      <term> <identifier> b </identifier> </term>
      <symbol> - </symbol>
      <term> <identifier> c </identifier> </term>
    <symbol> ; </symbol>
   </expression>
</letStatement>
```

compiler

parsing *and* code generation (project 11)

VM (pseudo) code

```
push a
push b
+
push c
-
pop x
```

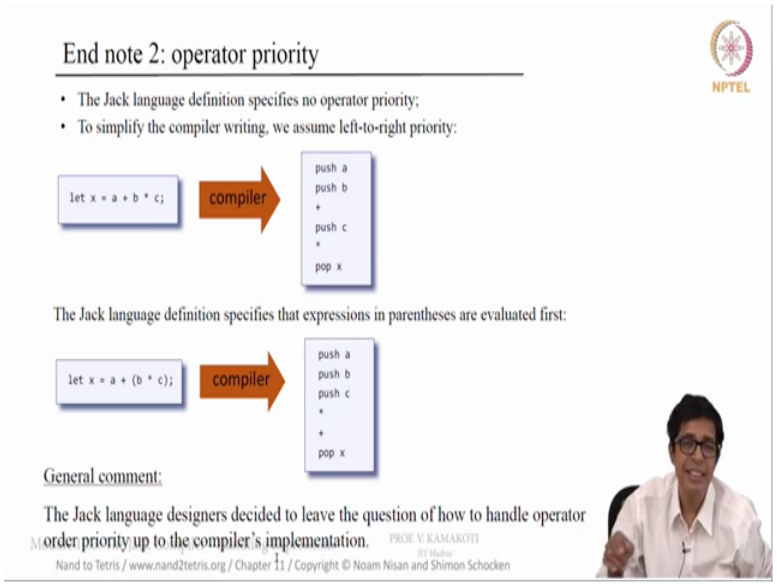
Module 10.3: The Jack Compiler - Handling Expressions

PROF. V. KAMAKOTI IIT Madras

Nand to Tetris / www.nand2tetris.org / Chapter 11 / Copyright © Noam Nisan and Shimon Schocken

So X is equal to a plus b minus c this was the source and this is the XML code you wanted and from this XML code we need to get this particular pseudo code okay so this is the, so what happened in project 10 was this x is equal to a plus b plus c got converted to this and in the case of project, this current project we need to pass this XML code and get this (())(13:29)

(Refer Slide Time: 13:33)



End note 2: operator priority

- The Jack language definition specifies no operator priority;
- To simplify the compiler writing, we assume left-to-right priority:

```
let x = a + b * c;
```

```
push a
push b
+
push c
*
pop x
```

The Jack language definition specifies that expressions in parentheses are evaluated first:

```
let x = a + (b * c);
```

```
push a
push b
push c
*
+
pop x
```

General comment:

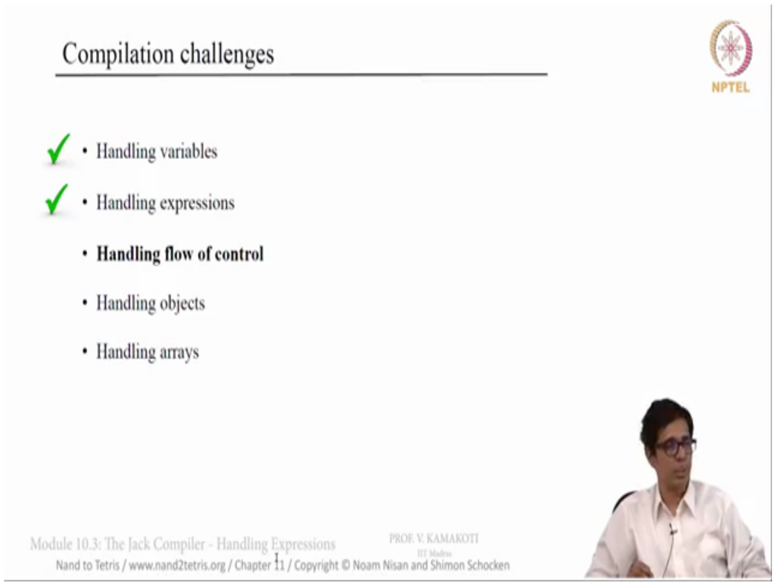
The Jack language designers decided to leave the question of how to handle operator order priority up to the compiler's implementation.

PROF. V. KAMAKOTI
IIT Madras
Nand to Tetris / www.nand2tetris.org / Chapter 11 / Copyright © Noam Nisan and Shimon Schocken

And in many languages lot of time we are given this operator priority suppose I say a plus b into c, b and c are multiplied first and then a right so but in the case of our Jack language we give a left-to-right priority so the Jack language definitions (())(13:57) that the expression in parentheses are evaluated first however right, so the main thing is that so when we are trying to do this when I say a plus b into c, we will be doing a plus b followed the whole thing into c that is what we mean by left-to-right priority but suppose I want to do a plus b into c then put b into c in a bracket and that will be useful for us right.

If you do not put it in the bracket then it will be treated as I say a plus b into c, if I want a plus b into c then put b into c in the bracket okay.

(Refer Slide Time: 14:39)



Compilation challenges

✓ • Handling variables

✓ • Handling expressions

• **Handling flow of control**

• Handling objects

• Handling arrays

Module 10.3: The Jack Compiler - Handling Expressions

PROF. V. KAMAKOTI IIT Madras

Nand to Tetris / www.nand2tetris.org / Chapter 11 / Copyright © Noam Nisan and Shimon Schocken

So this is all about handling expressions and when we actually construct the compiler we will get lot more insight of how do we go about handling the VM code for expressions right, now we will in the next module we will start looking at handling the flow of control, thank you.