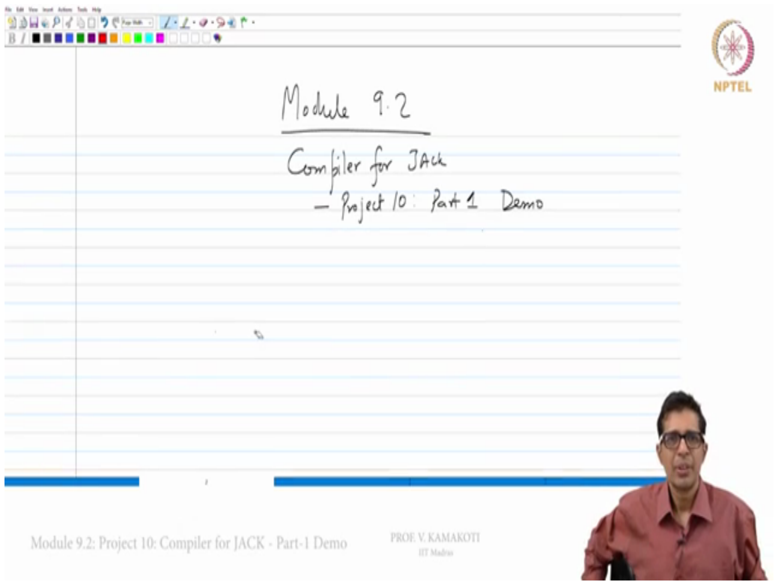
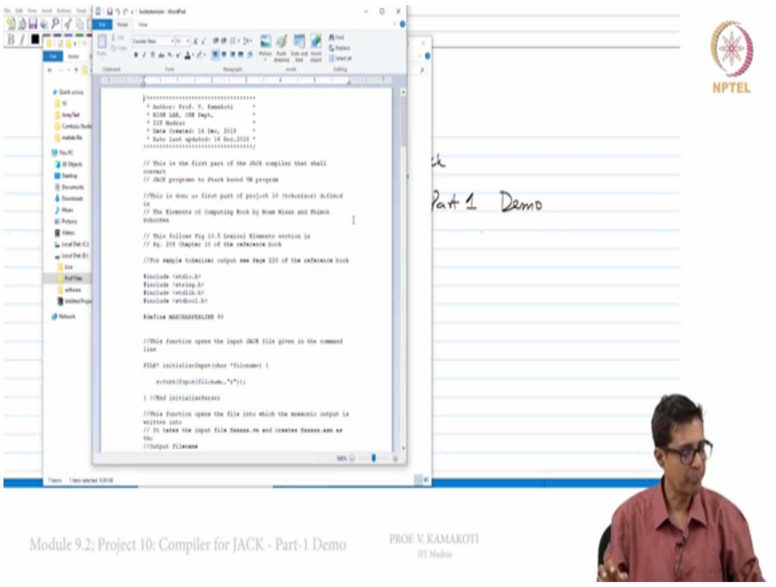
Foundations to Computer Systems Design
Professor V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology Madras
Module 9.2
Project 10: Compiler for JACK – Part-1 Demo

(Refer Slide Time: 0:16)


Module 9.2
Compiler for JACK
– Project 10: Part 1 Demo

So welcome to module 9.2 and this will be your first part of your project 10 namely the compiler project. So in project 10 as I mentioned earlier we will be doing the front end of the compiler which has two parts namely tokenization and then parsing. So in module 9 we have already seen what is tokenization but we will give you a demo of module 9.2 regarding tokenization, so how do we go about that?

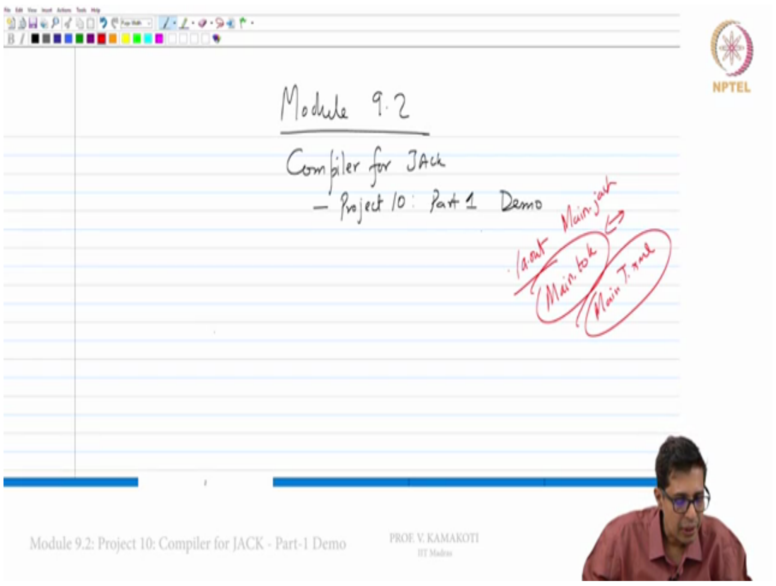
(Refer Slide Time: 0:49)



```
/*********************************
 * Author: Prof. V. Kamakoti     *
 * RISE LAB, CSE Dept,           *
 * IIT Madras                    *
 * Date Created: 16 Dec, 2018    *
 * Date last updated: 16 Dec,2018 *
*********************************/

// This is the first part of the JACK compiler that shall
convert
// JACK programs to Stack based VM program

//This is done as first part of project 10 (tokenizer) defined
in
// The Elements of Computing Book by Noam Nisan and Shimon
Schocken

// This follows Fig 10.5 Lexical Elements section in
// Pg. 208 Chapter 10 of the reference book

//For sample tokenizer output see Page 220 of the reference book

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

#define MAXCHARPERLINE 80


//This function opens the input JACK file given in the command
line

FILE* initializeInput(char *filename) {

    return(fopen(filename,"r"));

} //End initializeParser

//This function opens the file into which the mnemonic output is
written into
// It takes the input file Xxxxx.vm and creates Xxxxx.asm as
the
//Output filename
```

Module 9.2: Project 10: Compiler for JACK - Part-1 Demo

PROF. V. KAMAKOTI
IIT Madras



Module 9.2

Compiler for JACK

– Project 10: Part 1 Demo

Main.jack

a.out

Main.tok

MainT.xml

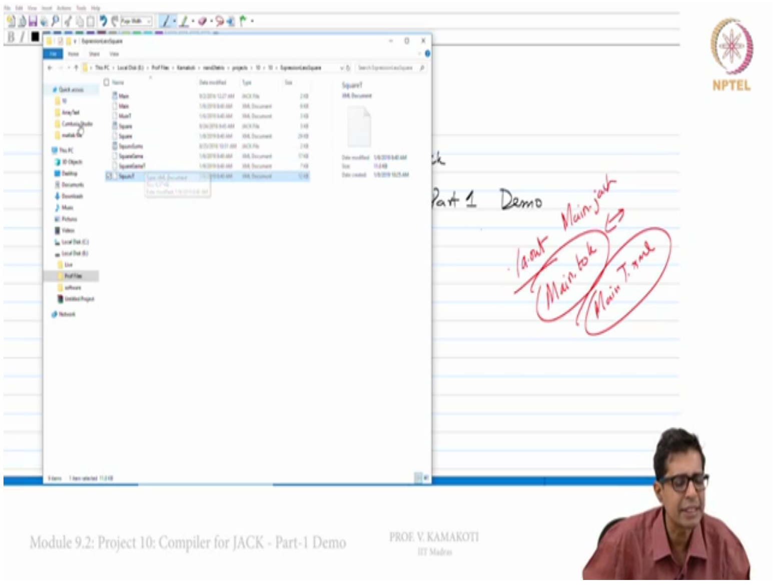
Module 9.2: Project 10: Compiler for JACK - Part-1 Demo

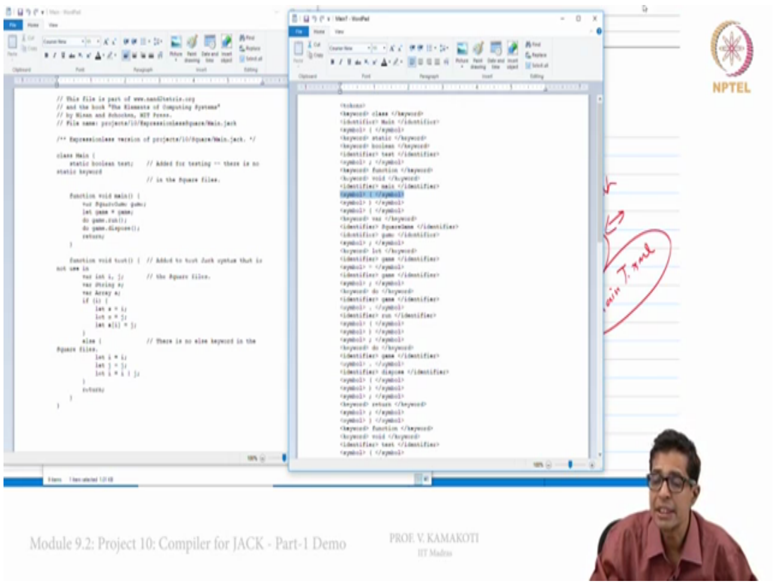
PROF. V. KAMAKOTI
IIT Madras

So I will just take you to the directory structure here so in a directory structure in project 10 you have three directories array test, expression less square and square and in each of this there will be a Jack file so in this case there is main dot Jack if you go and look at the second one namely your expression less square you have main dot Jack, then square dot Jack and square game dot Jack you actually have three Jack files and then if you go and look at the square actually you get again main square and square game, so there are three Jack files.

Now associated with each of these Jack files so let us take here main dot Jack you main a main t dot Jack, this main t dot Jack is nothing but the tokenized file. So what you need to do is you need to write a program as I was shown here called Jack tokenizer dot c, this Jack tokenizer dot c you compile it you get dot slash a dot out, then you give this main dot Jack as an input to this file, this will give you say some main dot tok, this main dot tok should actually agree with your main dot main t dot xml both should be same then your tokenizer has worked.

(Refer Slide Time: 2:20)





```
// 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/10/ExpressionLessSquare/Main.jack

/** Expressionless version of projects/10/Square/Main.jack. */

class Main {
    static boolean test;    // Added for testing -- there is no
static keyword
                            // in the Square files.

    function void main() {
        var SquareGame game;
        let game = game;
        do game.run();
        do game.dispose();
        return;
    }

    function void test() {  // Added to test Jack syntax that is
not use in
        var int i, j;       // the Square files.
        var String s;
        var Array a;
        if (i) {
            let s = i;
            let s = j;
            let a[i] = j;
        }
        else {              // There is no else keyword in the
Square files.
            let i = i;
            let j = j;
            let i = i | j;
        }
        return;
    }
}
```

```
<tokens>
<keyword> class </keyword>
<identifier> Main </identifier>
<symbol> { </symbol>
<keyword> static </keyword>
<keyword> boolean </keyword>
<identifier> test </identifier>
<symbol> ; </symbol>
<keyword> function </keyword>
<keyword> void </keyword>
<identifier> main </identifier>
<symbol> ( </symbol>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> var </keyword>
<identifier> SquareGame </identifier>
<identifier> game </identifier>
<symbol> ; </symbol>
<keyword> let </keyword>
<identifier> game </identifier>
<symbol> = </symbol>
<identifier> game </identifier>
<symbol> ; </symbol>
<keyword> do </keyword>
<identifier> game </identifier>
<symbol> . </symbol>
<identifier> run </identifier>
<symbol> ( </symbol>
<symbol> ) </symbol>
<symbol> ; </symbol>
<keyword> do </keyword>
<identifier> game </identifier>
<symbol> . </symbol>
<identifier> dispose </identifier>
<symbol> ( </symbol>
<symbol> ) </symbol>
<symbol> ; </symbol>
<keyword> return </keyword>
<symbol> ; </symbol>
<symbol> } </symbol>
<keyword> function </keyword>
<keyword> void </keyword>
<identifier> test </identifier>
<symbol> ( </symbol>
```

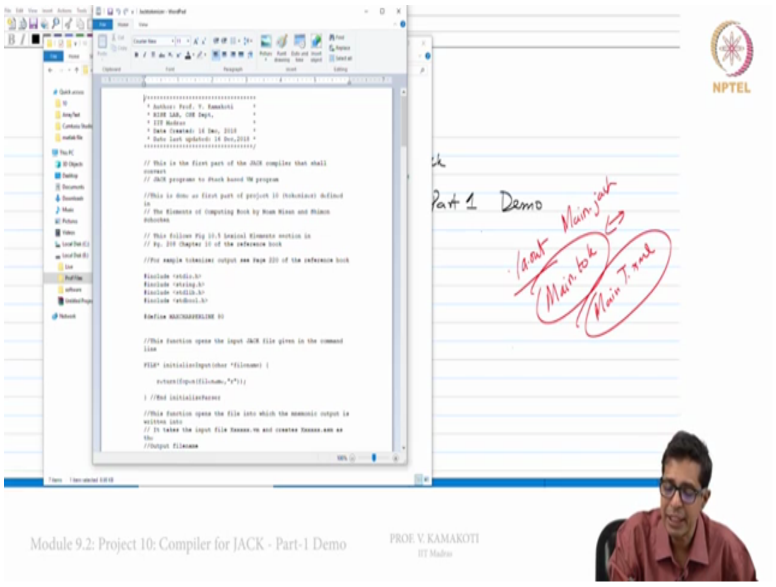
Similarly, if you go to the next directory here as you see here if you go to the let us go to the expression less square, so you compile main it will give you some main dot tok, you compare square it will give you square dot tok, you compare square game it will give you square game dot tok that main dot tok you compare it with main t dot xml square dot tok with square dot xml and (square game dot tok with) square dot tok with square t dot xml and square game dot tok with square game t dot xml. So your tokenizer basically has to take this file and generate that equivalent of that xml file and how will the xml file look like?

So I will just open it up in WordPad and show you how it will be, so you start off with tokens and end with again end with tokens slash tokens that is one thing that you need to give and in each line you have basically so this is main there so let us take main so in each line you are going to put one entity of that for example all the comments are taken off here as you see here so there are single line comments then there are interface level comments this is another comment and then there are comments which start in the middle of the line and go to the end, so these are the things.

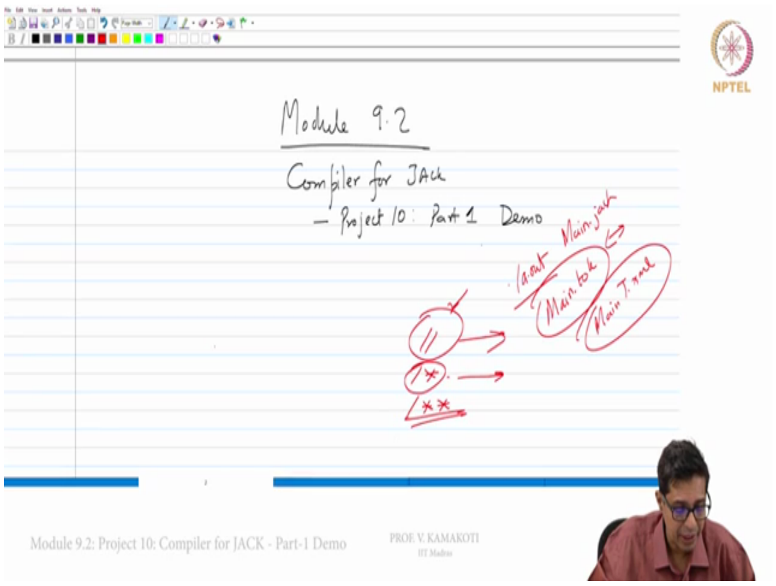
So let us see so all the comments are ignored as I told you earlier so the first one would be keyword class there is a class this is the first lexical element and you know that it is a keyword so you get this keyword class keyword slash. The next one is main, main is an identifier and so you get it, then there is a symbol which is this one, then static is a keyword, Boolean is a keyword, test is identifier, semicolon is a symbol, then again static is a keyword sorry then again function is a keyword, void is a keyword, main is an identifier, then this small parentheses close parentheses are two symbols as you see here, symbol and symbol and then there is a start symbol, this is this open braces, then another symbol keyword var, and then square game is an identifier, game is a identifier, okay. So like this every every lexical entity here gets converted and then you get this value.

So your tokenizer is expected to do this conversion one after one and then you compare it with so if you take main dot Jack you will get main dot tok or something and then you compare it main t dot xml to see both are same, if they are same then you can ensure that your tokenizer is going to be straight forward is going to be correct.

(Refer Slide Time: 5:58)



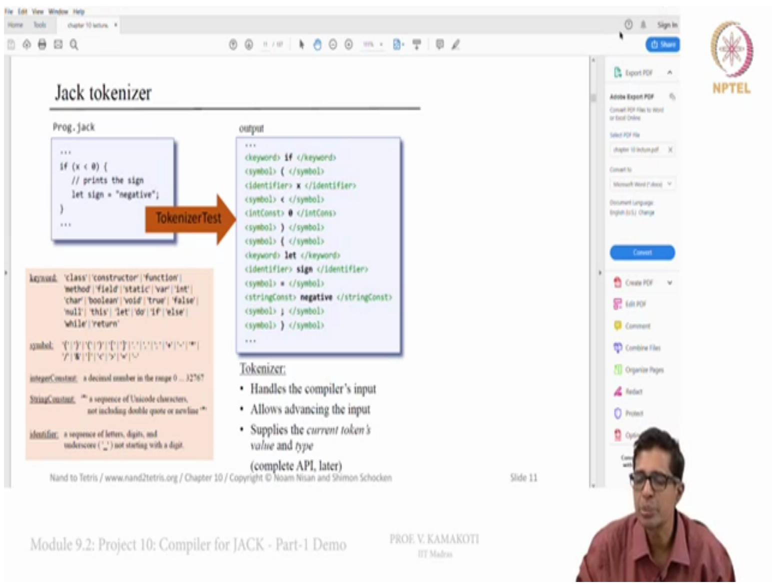
```
/*********************************
 * Author: Prof. V. Kamakoti      *
 * RISE LAB, CSE Dept,            *
 * IIT Madras                     *
 * Date Created: 16 Dec, 2018     *
 * Date last updated: 16 Dec,2018 *
 *********************************/

// This is the first part of the JACK compiler that shall
convert
// JACK programs to Stack based VM program

//This is done as first part of project 10 (tokenizer) defined
in
// The Elements of Computing Book by Noam Nisan and Shimon
Schocken

// This follows Fig 10.5 Lexical Elements section in
// Pg. 208 Chapter 10 of the reference book

//For sample tokenizer output see Page 220 of the reference book

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

#define MAXCHARPERLINE 80


//This function opens the input JACK file given in the command
line

FILE* initializeInput(char *filename) {

   return(fopen(filename,"r"));

} //End initializeParser

//This function opens the file into which the mnemonic output is
written into
// It takes the input file Xxxxxx.vm and creates Xxxxxx.asm as
the
//Output filename
```



So the entire effort of developing this tokenizer is square rest upon de-commenting so we have to so this is the basic tokenizer code so we spend quite a bit of time de-commenting. So there can be three types of comments which is slash slash, slash star and slash star star while this is single line comment after that nothing else is needed but the slash star need not be a single line comment it can span across multiple lines and so is slash star star.

So whenever I see a slash star we keep on in every line we inspect if we get a star slash if you get a star slash that means your assembly part is over, if you do not get a star slash then we proceed to the next line and every line I go and check if I am part of a comment or not. so that is one of the most important thing. In addition your double slash or these things can start with the middle of the line they need not start at the beginning of the line, so that also makes the whole thing little more challenge in de-commenting the code once you de-comment that code then the whole thing becomes easy.

(Refer Slide Time: 7:16)



Jack tokenizer

Prog.jack

```
...
if (x < 0) {
   // prints the sign
   let sign = "negative";
}
...
```

output

```
...
<keyword> if </keyword>
<symbol> ( </symbol>
<identifier> x </identifier>
<symbol> < </symbol>
<intConst> 0 </intCons>
<symbol> ) </symbol>
<symbol> { </symbol>
<keyword> let </keyword>
<identifier> sign </identifier>
<symbol> = </symbol>
<stringConst> negative </stringConst>
<symbol> ; </symbol>
<symbol> } </symbol>
...
```

keyword: 'class' | 'constructor' | 'function' | 'method' | 'field' | 'static' | 'var' | 'int' | 'char' | 'boolean' | 'void' | 'true' | 'false' | 'null' | 'this' | 'let' | 'do' | 'if' | 'else' | 'while' | 'return'

symbol: '{' | '}' | '(' | ')' | '[' | ']' | '.' | ',' | ';' | '+' | '-' | '*' | '/' | '&' | '|' | '<' | '>' | '=' | '~'

integerConstant: a decimal number in the range 0 ... 32767

StringConstant: '"' a sequence of Unicode characters, not including double quote or newline '"'

identifier: a sequence of letters, digits, and underscore ('_') not starting with a digit.

Tokenizer:

- Handles the compiler's input
- Allows advancing the input
- Supplies the *current token's value* and *type*

(complete API, later)

Nand to Tetris / www.nand2tetris.org / Chapter 10 / Copyright © Noam Nisan and Shimon Schocken   Slide 11

Module 9.2: Project 10: Compiler for JACK - Part-1 Demo   PROF. V. KAMAKOTI IIT Madras

Now we have to so let us see the how the thing goes so every time so what happens is that every time I see a character or an underscore, if I see a character or underscore I basically into go to either the keyword or identifier, if I see an underscore I need not go to a keyword, no keyword starts with an underscore but identifiers can start with underscore, if I see anything other than that I need to go and confirm whether it is an identifier, or a keyword the moment I so this is a string so I the moment we decide on the based on the first character like if I start with i, i is a character it is an alphabet.

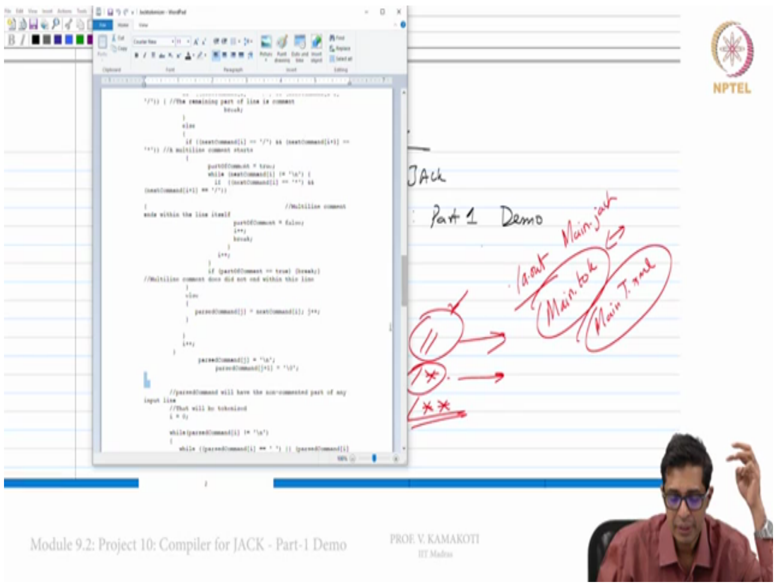
So now what is going to be after i will be a keyword or an identifier, it cannot be the other three elements or symbol, it cannot be integer constant, it cannot be a string constant because you did not see a double code, so we keep on proceeding (())(8:10) then till that terminator comes like terminator is in this case white space as you see on top of this, then you see a white space so there is where we stop.

Now we take this to i and f which is if and then compare if there is a keyword, yes if it is a keyword so I put keyword if. Then the next character I see is a nither an integer nor a double code nor a underscore, nor it is a integer. So I see something which is not a character, not integer, not a double code so obviously I will go and check for the symbol and I see less than as a symbol so immediately I put symbol here.

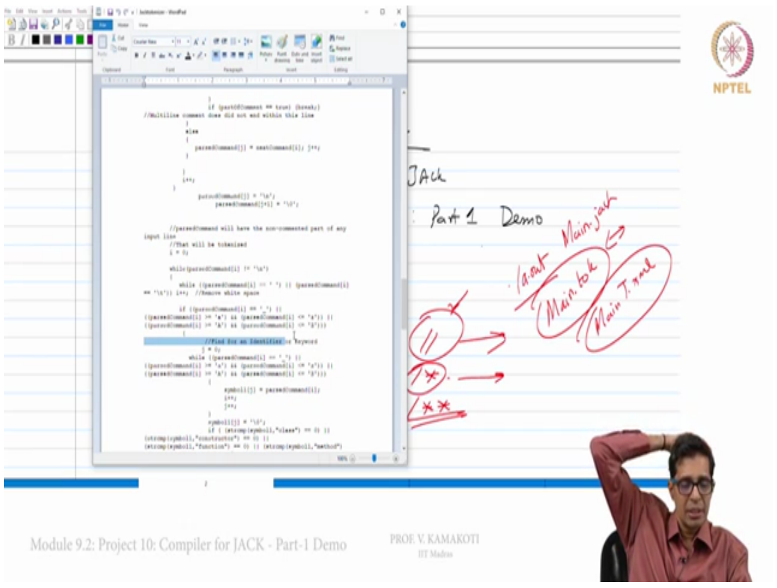
The next I see is a single digit integer 0, after 0 I scan I immediately get the next thing so the 0 stops here so that is why 0 it becomes an integer constant here. Now similarly next we see the closing parentheses and that is a symbol and stops, next your thing will stop in the starting of the braces which is again a symbol, and then of course the entire comment has to be taken off so then again we see let it becomes a keyword and so on.

So this is the way we can quickly do the project by screening one character after another the stream of characters examining one character after another and making a decision of whether what is that what is the lexical element so I have to get the next lexical element and then classify whether it is a keyword symbol, integer constant, string constant, or identifier, and this is essentially what we have done in this project.

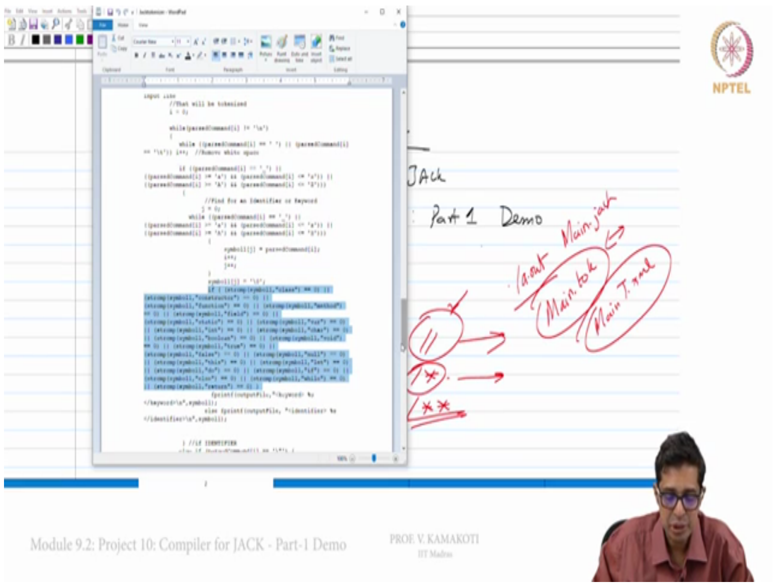
(Refer Slide Time: 10:13)



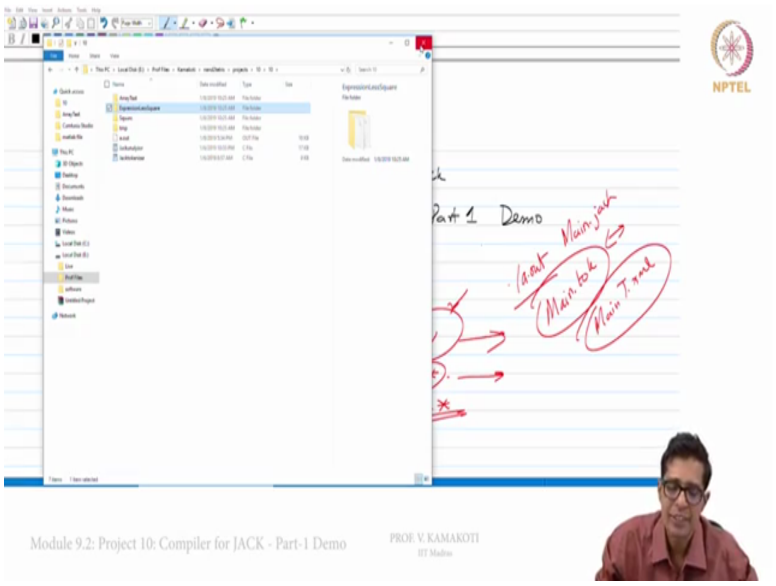
```
'/')) { //The remaining part of line is comment
                        break;
                }
                else
                {
                 if ((nextCommand[i] == '/') && (nextCommand[i+1] ==
'*')) //A multiline comment starts
                 {
                        partOfComment = true;
                        while (nextCommand[i] != '\n') {
                          if  ((nextCommand[i] == '*') &&
(nextCommand[i+1] == '/'))

{                                               //Multiline comment
ends within the line itself
                                partOfComment = false;
                                i++;
                                break;
                              }
                           i++;
                        }
                        if (partOfComment == true) {break;}
//Multiline comment does did not end within this line
                 }
                 else
                 {
                   parsedCommand[j] = nextCommand[i]; j++;
                 }

                }
                i++;
              }
                 parsedCommand[j] = '\n';
                      parsedCommand[j+1] = '\0';

          //parsedCommand will have the non-commented part of any
input line
          //That will be tokenized
          i = 0;

          while(parsedCommand[i] != '\n')
          {
            while ((parsedCommand[i] == ' ') || (parsedCommand[i]
```



```
                 }
                        if (partOfComment == true) {break;}
//Multiline comment does did not end within this line
                 }
                 else
                 {
                   parsedCommand[j] = nextCommand[i]; j++;
                 }

                }
                i++;
              }
                 parsedCommand[j] = '\n';
                      parsedCommand[j+1] = '\0';

          //parsedCommand will have the non-commented part of any
input line
          //That will be tokenized
          i = 0;

          while(parsedCommand[i] != '\n')
          {
            while ((parsedCommand[i] == ' ') || (parsedCommand[i]
== '\t')) i++;  //Remove white space

            if ((parsedCommand[i] == '_') ||
((parsedCommand[i] >= 'a') && (parsedCommand[i] <= 'z')) ||
((parsedCommand[i] >= 'A') && (parsedCommand[i] <= 'Z')))
              {
                        //Find for an Identifier or Keyword
                      j = 0;
                    while ((parsedCommand[i] == '_') ||
((parsedCommand[i] >= 'a') && (parsedCommand[i] <= 'z')) ||
((parsedCommand[i] >= 'A') && (parsedCommand[i] <= 'Z')))
                        {
                            symbol1[j] = parsedCommand[i];
                            i++;
                            j++;
                        }
                        symbol1[j] = '\0';
                        if ( (strcmp(symbol1,"class") == 0) ||
(strcmp(symbol1,"constructor") == 0) ||
(strcmp(symbol1,"function") == 0) || (strcmp(symbol1,"method")
```



```
input line
        //That will be tokenized
        i = 0;

        while(parsedCommand[i] != '\n')
        {
          while ((parsedCommand[i] == ' ') || (parsedCommand[i]
== '\t')) i++;  //Remove white space

          if ((parsedCommand[i] == '_') ||
((parsedCommand[i] >= 'a') && (parsedCommand[i] <= 'z')) ||
((parsedCommand[i] >= 'A') && (parsedCommand[i] <= 'Z')))
            {
                //Find for an Identifier or Keyword
               j = 0;
             while ((parsedCommand[i] == '_') ||
((parsedCommand[i] >= 'a') && (parsedCommand[i] <= 'z')) ||
((parsedCommand[i] >= 'A') && (parsedCommand[i] <= 'Z')))
                {
                    symbol1[j] = parsedCommand[i];
                    i++;
                    j++;
                }
                symbol1[j] = '\0';
                if ( (strcmp(symbol1,"class") == 0) ||
(strcmp(symbol1,"constructor") == 0) ||
(strcmp(symbol1,"function") == 0) || (strcmp(symbol1,"method")
== 0) || (strcmp(symbol1,"field") == 0) ||
(strcmp(symbol1,"static") == 0) || (strcmp(symbol1,"var") == 0)
|| (strcmp(symbol1,"int") == 0) || (strcmp(symbol1,"char") == 0)
|| (strcmp(symbol1,"boolean") == 0) || (strcmp(symbol1,"void")
== 0) || (strcmp(symbol1,"true") == 0) ||
(strcmp(symbol1,"false") == 0) || (strcmp(symbol1,"null") == 0)
|| (strcmp(symbol1,"this") == 0) || (strcmp(symbol1,"let") == 0)
|| (strcmp(symbol1,"do") == 0) || (strcmp(symbol1,"if") == 0) ||
(strcmp(symbol1,"else") == 0) || (strcmp(symbol1,"while") == 0)
|| (strcmp(symbol1,"return") == 0) )
                 fprintf(outputFile,"<keyword> %s
</keyword>\n",symbol1);
                else fprintf(outputFile, "<identifier> %s
</identifier>\n",symbol1);


            } //if IDENTIFIER
```



I will just show you the source of this project the tokenizer so we spent quite a bit of so this is the main function spent quite a bit of time and energy to basically find out if it is a comment. For example a multi-line comment can pass across like this. So every time I read a line I need to find out whether it is part of that comment, if it is part of the comment then I can ignore because that is irrelevant for the compiler, if it is not part of the comment we need to basically start processing.

Okay, so the entire effort till this as you see I am slowly going through the code till this part is going to be comments. So you read a line and whatever is the comment you remove the remaining things you should put it into what we call as the past comment, okay so this is put as the part of the past comment. Now what is it that we need to do? So this past comment is every line we take and remove off the comments and whatever remaining is actually put as a part of this past comment and what will the past comment do? It will check this is exactly what it is doing, if your current thing that I am reading if it is an underscore or a or a capital A or (())(11:58) then it is an identifier or keyword, right.

Now if it is not this then you go and find if it is a number then it becomes integer constant, if it is a double code it becomes string constant and the other element, okay so this is where or it becomes an identifier, it is a identifier or keyword again we have decided that so once it is an identifier or keyword now this particular big one go and compare and find sort if it is agreeing with any of the keyword then you say it is a keyword, otherwise it continues going identifier, else it is a string constant or integer constant or it can be a symbol (())(12:47) so each element can be.

So this is how you build a tokenizer, now you compile that tokenizer you get a dot out apply on the Jack file you will get something dot tok or whatever that you compare with the main t dot xml that (())(13:07) and you should not get an error and that you repeat it for all the three things for example if you take the expression less square there are three Jack files main, square and square game so apply this dot slash a dot out on main you get main dot tok, square dot tok, square game dot tok, now we can basically compare what you have generate this talk with token main dot tok with main t dot tok main t dot xml, square dot tok with square t dot xml and square game dot tok with square game t dot xml. So whatever you have generated you can compare with is given already here and if there is no difference and there is no difference in all the three things array, expression less square and square there is no difference then your tokenizer project is successfully completed.

So all the best this is a very very simple exercise you can do it in Python, C plus plus, C, C sharp whatever is interesting to you, but nevertheless this is the very very simple (())(14:18) and exercise, we will again meet in module 9.3, thank you.