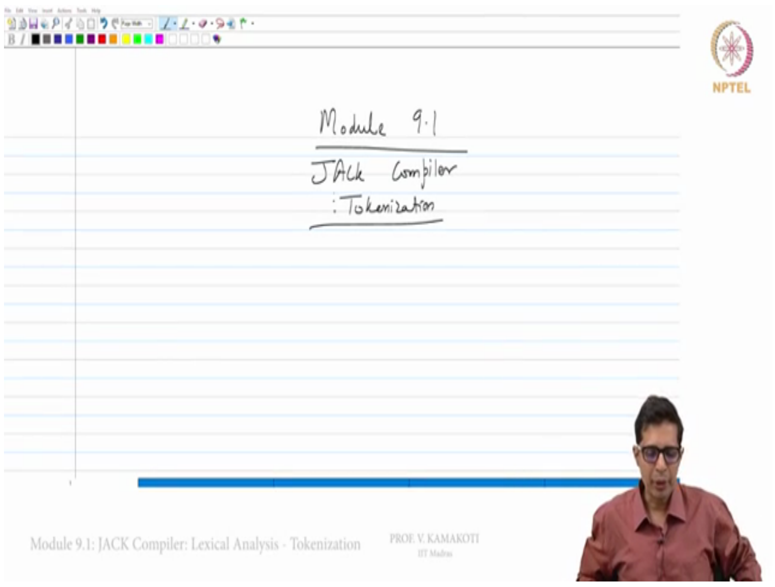
Foundations to Computer Systems Design
Professor V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology Madras
Module 9.1
JACK Compiler: Lexical Analysis – Tokenization

(Refer Slide Time: 0:16)



So welcome to module 9.1 and we are going to the penultimate stage of the entire course wherein we are looking at compiling the JACK language I hope you had done significant amount of exploration through project 9, where you got a reasonable understanding of the JACK as a language so some of the routines that were there I hope you executed those programs, you also made VM emulation, you converted into VM using the JACK compiler that is provided as part of your tolls and actually executed the VM to find out much more interesting facts about the programing language and also the interface of the programing language with the operating system.

So with that background assuming that background we have already got through. Now we will go into the most important aspect of writing a compiler. So writing a compiler is a very very interesting thing and there are very very few compiler researchers across the World, people who can actually hack into the compiler and basically you know come out with very good code for compilation.

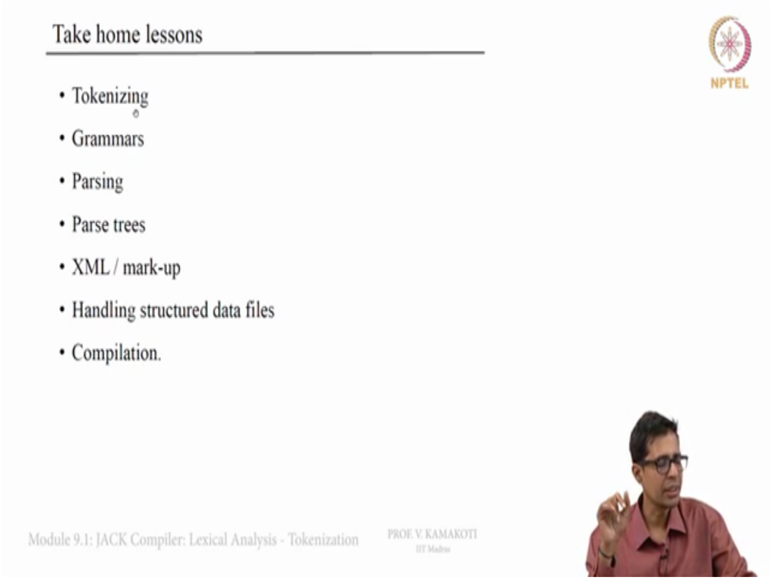
So understanding compiler and its full perspective is a very very important aspect of any computer science in engineering curriculum and I hope this module module 9.1 to 9.6 we will try and cover the what we call as the front end of the compiler, the front end of the compiler we will be explaining it in more detail but to just give you a nutshell the front end of the compiler will take the code and convert it into what you call as a parse tree, what is a parse tree we will be seeing.

Now in the module 10, we will go into what we call as the backend of the compiler probably the last module 9.6 or module 10 somewhere there we will look at the backend of the compiler, wherein the actual code for the VM is the actually VM code is generated. So to basically get into that thing we are now starting the attempt of taking a JACK program and converting into a VM program stack based VM program.

Once we can convert it to that then the remaining part we have already done, we know how to covert VM to mnemonics and we know how to convert mnemonics to binary. So this is our next step and associated with this module is what you call as the project 10 which basically does the front end of the compiler so it is called compiler 1 which is the front end and project 11 will do the backend of the compiler and we will be ending up the course with project 12 which will be teaching you more about the operating system interface.

Of all the modules this module is going to be pretty easy provided you actually make a good attempt to understand what we are trying to tell. So as the first part of this module 9 which is 9.1 we will be talking about something called tokenization, so what is token we will see.

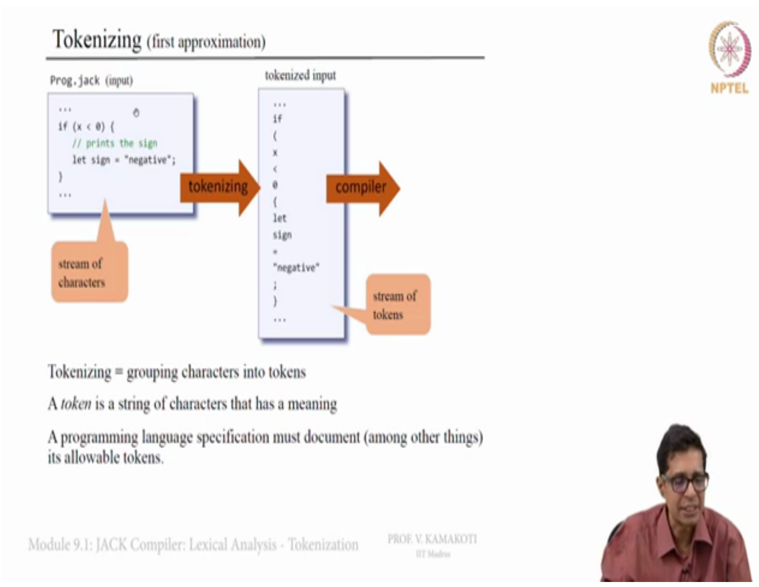
(Refer Slide Time: 3:44)


Take home lessons

- Tokenizing
- Grammars
- Parsing
- Parse trees
- XML / mark-up
- Handling structured data files
- Compilation.

Module 9.1: JACK Compiler: Lexical Analysis - Tokenization

PROF. V. KAMAKOTI
IIT Madras



Tokenizing (first approximation)

Prog.jack (input)

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

tokenized input

```
...
if
(
x
<
0
{
let
sign
=
"negative"
;
}
...
```

Tokenizing = grouping characters into tokens

A *token* is a string of characters that has a meaning

A programming language specification must document (among other things) its allowable tokens.

Module 9.1: JACK Compiler: Lexical Analysis - Tokenization PROF. V. KAMAKOTI IIT Madras

Now I am basically using the slides that are available in (())(3:49) under projects division, so you can download the same slides and keep it for your reference, there are slides for every project that you have done and you can continue to see that there, okay so let me just. So we will be doing tokenization as the first part of this entire exercise what is tokenization? If I have a program I would love to understand what are all the different elements of the programing language that the program actually contains?

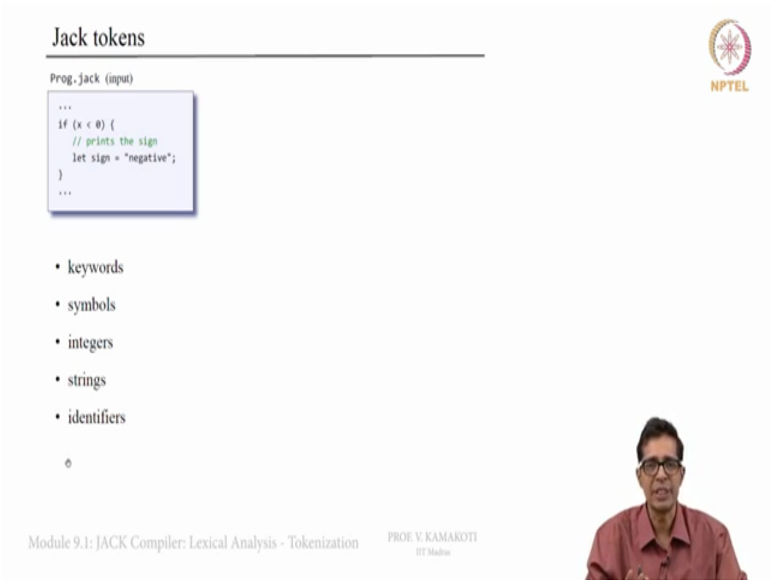
For example look at this slide, I have on the left hand side you have if x greater than 0 then let sign equal to negative so this is a simple statement there are comments also. So what is relevant for the compiler? First and foremost the comment is not relevant for the compiler so the comment is removed. Now what will tokenization do? It will split every small small entity of the left hand side into separate lines one entity per line. Each entity is actually called a lexical element, lexical is basically associated with the syntax or grammar of your language.

So what does your language basically possess here? your language possess what you see on the right hand side say if it is a separate lexical element let us say keyword bracket parentheses open left parentheses is a separate element x is an identifier, it is an element of your language less than sign is an element of your language, 0 is again a constant element of your and then we have missed the closed parentheses there should be a closed parentheses also.

Then after the 0 I am talking about the right hand side and I am moving the mouse there, so then the braces is one element let then the comment is completely not necessary for us. Let is a statement let is a keyword, sign is an identifier, equal to is a symbol and negative this is a string constant and then what do you see? Semi colon is a separator again it is a symbol, then this closing parentheses is also a symbol.

So your program what you call as prog dot jack that you see on the left hand side is a stream of characters and what essentially happens after tokenization is that every element here every programmable entity here get split into different lines one program will entity or what you call as a lexical element is put in every line and you essentially each lexical element is called a token. So a stream of characters that form your jack program essentially gets converted into a stream of tokens and this is what we call as tokenization.

(Refer Slide Time: 7:10)



Jack tokens

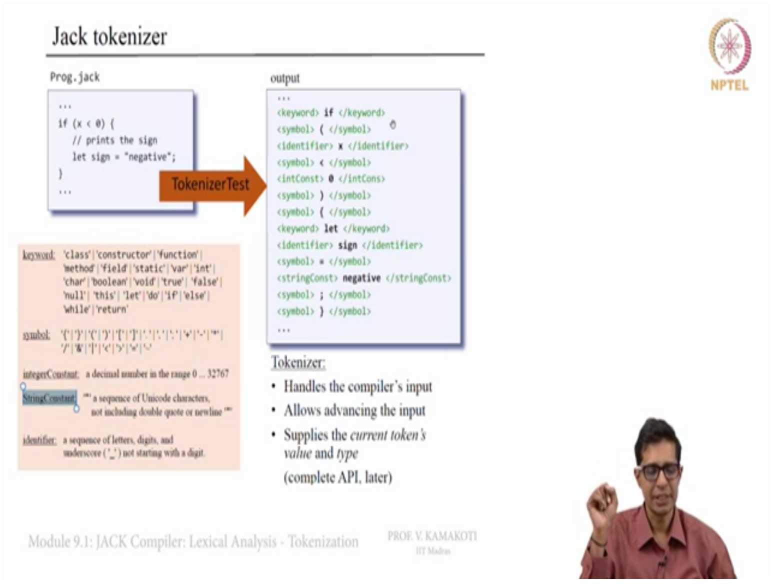
Prog.jack (input)

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

- keywords
- symbols
- integers
- strings
- identifiers

Module 9.1: JACK Compiler: Lexical Analysis - Tokenization

PROF. V. KAMAKOTI
IIT Madras



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 '"'

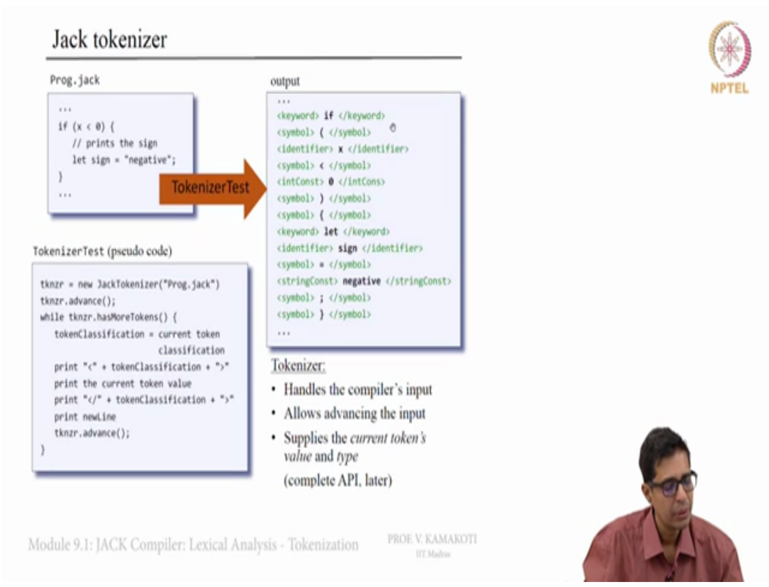
identifier: a sequence of letters, digits, and underscore ('_') not starting with a digit.

Tokenizer:

- Handles the compiler's input
- Allows advancing the input
- Supplies the *current token's value* and *type* (complete API, later)

Module 9.1: JACK Compiler: Lexical Analysis - Tokenization

PROF. V. KAMAKOTI
IIT Madras



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>
...
```

TokenizerTest (pseudo code)

```
tknzr = new JackTokenizer("Prog.jack")
tknzr.advance();
while tknzr.hasMoreTokens() {
   tokenClassification = current token
                         classification
   print "<" + tokenClassification + ">"
   print the current token value
   print "</" + tokenClassification + ">"
   print newLine
   tknzr.advance();
}
```

Tokenizer:

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

(complete API, later)

Module 9.1: JACK Compiler: Lexical Analysis - Tokenization   PROF. V. KAMAKOTI IIT Madras

Now how do we approach this tokenization? Pretty simple so what are the things that are part of your what are the different tokens that you have? You could have keywords, you could have symbols which are basically identifiers, you could have integer constants, you could have string constants and you could have identifiers, you could have symbols sorry symbols are equal to, greater than, less than, braces, etc. Then you could have integer constants, string constants and identifiers so there are five different lexical elements in your entire language of jack.

So whatever is relevant to the compiler in a program comments are not relevant, everything other than comments are relevant to the compiler and every entity in that program can be mapped on to one of these five different heads that we have listed here. So this is what so this is so what are the all the possible keywords? You have these time class, constructor, function, method, field, static, var, int like that there are totally you know 16, 18, 21 different keywords.

So anywhere I am seeing for example in this case as I am moving the mouse there on the top if you see let, let is a keyword I need to classify it as a keyword. Similarly there are so many number of symbols, open braces, close braces, parentheses, square, brackets, plus, minus, star, tilde, etc okay there are so many of them. And then there could be integer constants, there could be string constants, how will the string constant look like? It will start with a double code and end with a double code and then I could have identifiers which are variables which can be a sequence of letters, digits and underscore but not starting with a digit.

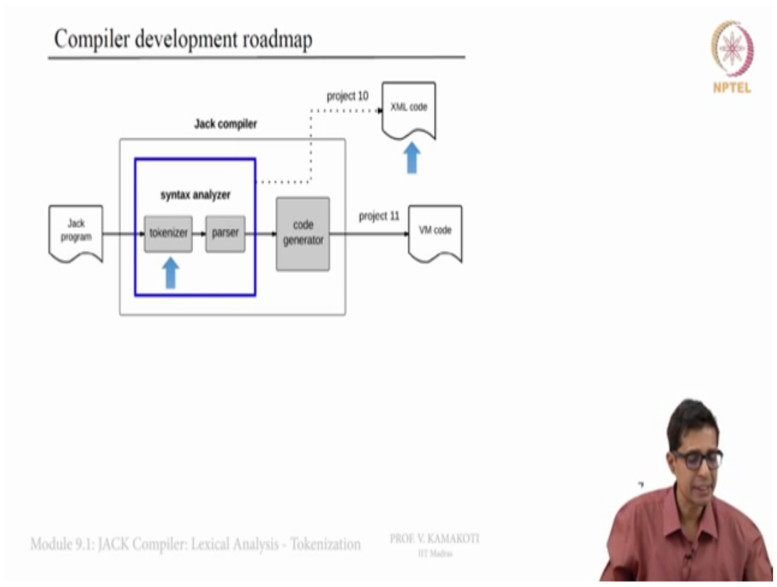
For example x is an identifier, sign is an identifier on top, they are all a sequence of letters, or digits and underscore and that sequence should not start with a number like digit. So I could not have variables which are starting like 2x or something like, so I cannot start a variable with a digit, I can always start a variable with an underscore array alphabet and can be forward.

So this program that you see essentially has these elements. First there is a keyword which is if so we need to generate this file, what is the entire aspect of tokenization is to generate this file. So we sake keyword if slash keyword, symbol, then one blank then one parentheses another blank and symbol note that the parentheses is a symbol, then x is an identifier so I have identifier slash identifier, then I have symbol, again less than is a symbol, then again 0 is a int constant actually you should say integer constant as you see here in the program that you are going to write you should get integer constant, not int constant as you see here, integer constant (())(10:35) 0 integer constant, then symbol, then one blank, then actually that symbol close parentheses and then blank and slash symbol, so that is all.

So the first 1, 2, 3, 4, 5, 6 entities essentially gets tokenized as this, then the entire comment has to be neglected it should be removed. So then we have no we start with this symbol less than with braces and then the entire comment has to be removed, then we have identifier, then we have a keyword let, then we have a identifier sign, then we have a symbol equal to, then we have a constant from this const it should be string constant as you see here string constant and so that is negative slash string constant, then you have a symbol, then you have another symbol.

So every entity every lexical element that you see in this program gets or put into a separate line each and then identify it whether it is a keyword or a symbol or an integer constant or a string constant, or identifier? So the jack basically has these five entities. So this is basically your tokenization.

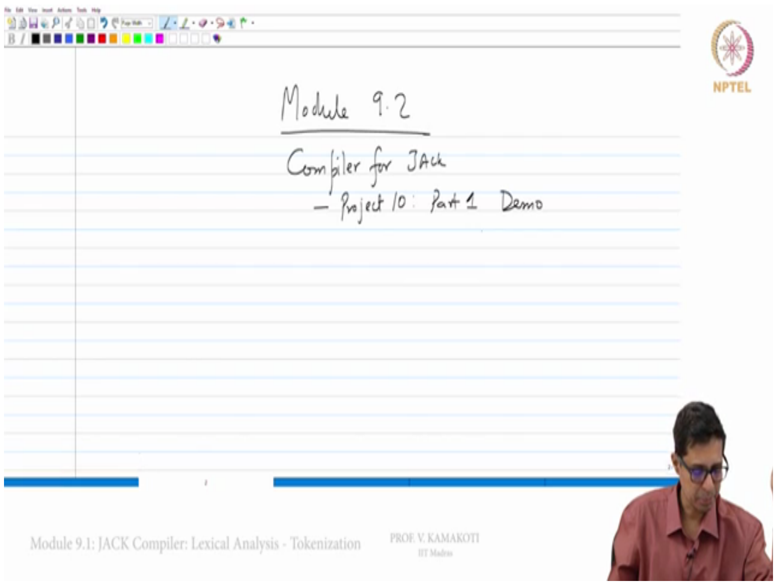
(Refer Slide Time: 12:08)



Compiler development roadmap

Module 9.1: JACK Compiler: Lexical Analysis - Tokenization

PROF. V. KAMAKOTI
IIT Madras

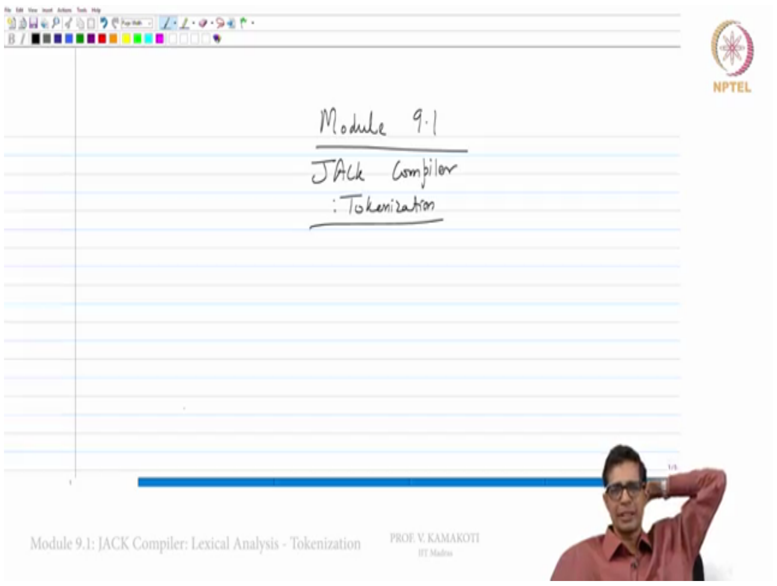


Module 9.2

Compiler for JAck
– Project 10 : Part 1 Demo

Module 9.1: JACK Compiler: Lexical Analysis - Tokenization

PROF. V. KAMAKOTI
IIT Madras



Module 9.1

JAck Compiler
: Tokenization

Module 9.1: JACK Compiler: Lexical Analysis - Tokenization

PROF. V. KAMAKOTI
IIT Madras

Now so now let us see this so the first part of your project 10 is to do this tokenization, so the first part of your project 10 is to basically do this tokenization and what you will see in the next module is a demo of this tokenization, before we go into the demo part what we need to do as a part of writing that program for the tokenization that also we will basically see in module 9.2.

So to sum-up module 9.1 we now take the language identify every lexical element and put it in separate lines and classify this element as whether it is a keyword, or it is a identifier, or it is symbol, or it is a integer constant, or a string constant one of these five. So that is what is what we mean by tokenization. By tokenization we have understood the complete every lexical element of your program is basically understood and in addition we also understand in which sequence they are arranged one after another.

So the sequence in which these tokens are coming and what are those tokens and identifying of every lexical element is done as a part of the first part of your compiler which is called tokenization, thank you.