Foundations to Computer Systems Design
Professor V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology, Madras
Module 9.6
Project 10 Part 3 Testing the Correctness

(Refer Slide Time: 00:16)

So welcome to module 9 point 6 and then this, we will see how we go about the final project 10 testing.

(Refer Slide Time: 00:25)

So you would have generated 2 files, 1 is the jack tokenizer and the jack analyzer, so to this so there are three directories in your project 10, one is the this array test square and expressionless square, you will see three directories as a part of your project 10 array test expressionless square and square as you see here. So if you go into that array test you will see if you will actually see each one you will see a main file main dot jack, main T dot XML and name main dot XML.

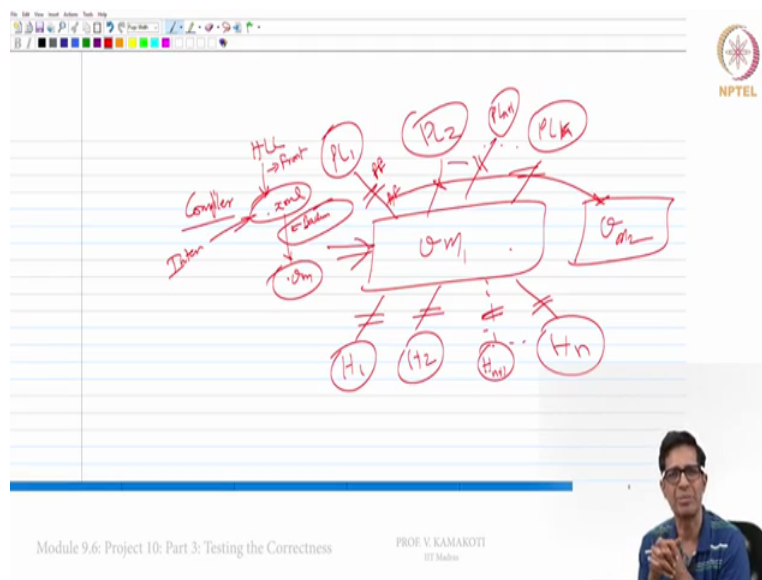
So you take this jack tokenizer to this jack tokenizer you give as a input your main dot jack and this will give me say let my let me call it main dot tok, this and this main T dot XML should be the same. Now to this jack analyzer you give this main dot tok as input and this would give me let us say it is going to give you some main dot ansL or whatever this and this should be same.

So if you are using Linux you can use this dift command windows error, windows you can find (sub) similar command but in Linux you can use dift command and all this files are basically docs file, so use this (())(02:08) call docs to Unix in Linux to convert these file, so it will remove all the docs character so that when you compare rematch, right. So in the case of square you will actually see a main dot jack again main T dot XML, main dot XML then you will also see a square dot jack dot XML, T dot square, T dot XML and square, square dot XML, you will also see a square game dot XML, square game T, square game dot jack, square game T dot XML and square game dot XML, ok so three files, it is the same thing.

So you have to compare compile each of this file so with your jack tokenizer and then analyzer and then compare these two as you see and similarly you have to do for the other thing and a similarly you can do it for the expressionless square. So all these three if you test and if they the dift matches that is there is no difference then you of got your tokenizer and analyzer in a proper shape, so this is what is expected out of you as a part of project 10 and if you are any doubts please do put on the discussion forum and we will discuss that in greater depth there.

So this is the part 1 or what we call as the front end of your compiler where you have taken the jack representation and you have a got an XML representation which could be which can be use for generating the code, this XML representation is also called as an intermediate code right, now this intermediate code is now going to be taken to generate code for your virtual machine, right.

(Refer Slide Time: 04:18)



So the interesting part now is one of the thing that we need to realize is that so we introduce the notion of a virtual machine to supports several hardware architectures right and so several programming languages can be there, so this is what was our earlier notion of virtual machine if you go back to the earlier modules some of the introduced. So I could have k different programming languages on n different hardware I have a common virtual machine, so I write a compiler for this and then an interpreter for this and it is done.

So whenever I introduce a new hardware say H dash n or H n plus 1 I need to write a compiler for this, I need to just write an interpreter for this and I introduce then if I write a interpreter all these program languages can start working on this. In similarly if you have if i introduce one PL n plus 1 I need to write a compiler for this and all the hardware here can basically execute there is a notion of this virtual machine.

Now suppose if I have two virtual machines vm 1 and vm 2 right, now this compiler what it does it takes the high level language and it has got me and dot XML based intermediator presentation from there I am going to get a dot vm file, you just the now this twos, so this is called the front end as a I mentioned earlier this is called the back end. So if I am introducing two virtual machines when I want this PL 1 to execute on vm 2 or vm 1 the only thing I need to change is only the back end, the front end can remain the same.

So there is a front end, there is a back end now, so let a say let us put a another virtual machine here vm 2, now this program should run on the vm 2 I need to take this back end alone and change for every one of this programming language the front end can remain the

same only the back end I need to change for this vm 2, the that is a major effort. So now we have got two projects right, project 10 and 11 for doing this for every time I introduce a new virtual machine only the project 11 has to be whatever you are going to discuss next is going need to be done, the other part may not be done.

So the (comp) so there is the compiler now itself has two parts one is the virtual machine independent part which is the front end and the virtual machine dependent part, so by doing this separation of virtual machine independent and dependent for me to port to compiler across different virtual machines become extremely easy. So that is the good thing about compilers, so I hope you have if you actually (comp) finish of this part what normally is there in many of the curriculum across different universities of compiler which basically stops with intermediate code generation that you have done within this two weeks.

Now we have understood quite a bit of that, so try and put that effort, so that you are very clear of how compiler indeed works, there are lot to none in compilers but nevertheless this is a very good starting point, it will I hope it will serve as a very good starting point for you, right. So we will now start looking at the core generation in the next module 10, thank you.