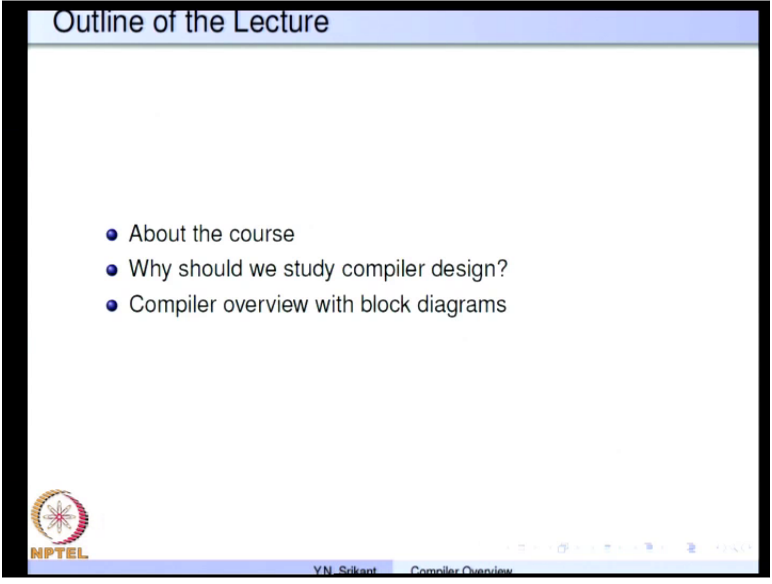
Principles of Compiler Design
Prof. Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

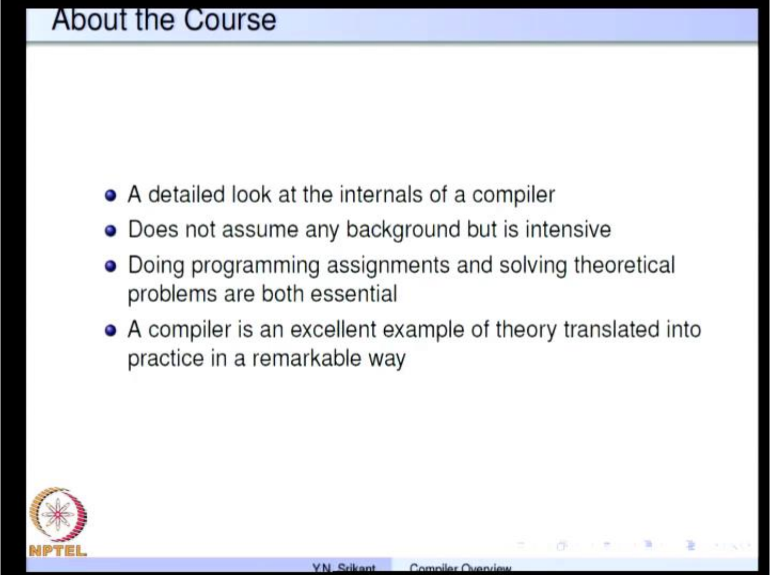
Lecture - 1
An Overview of a Compiler

(Refer Slide Time: 00:27)



Welcome to this new course on principles of compiler design. So, in this lecture, I will give you an overview of a compiler, but before that we will also see how exactly the course is organized. And then the motivation for studying compiler design and of course then go on to the details of a compiler with block diagrams.

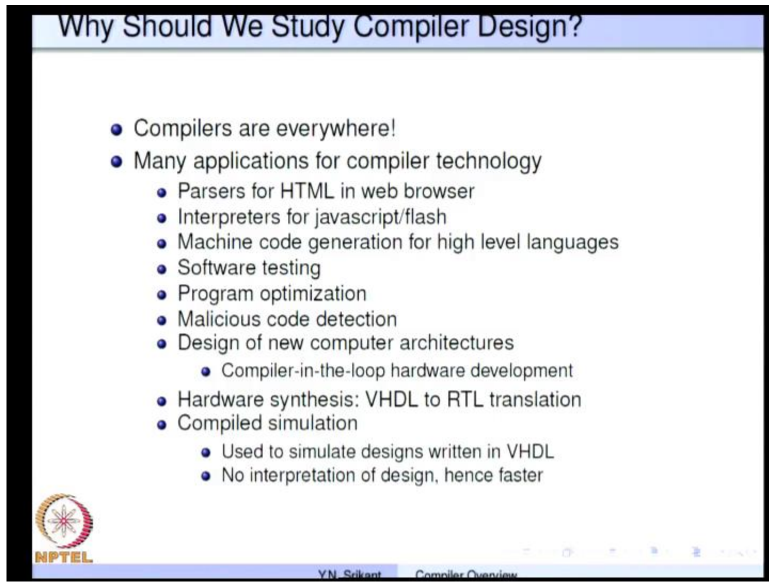
(Refer Slide Time: 00:43)



So, the course is actually a first level course. In other words this takes a detail look at the internals of a compiler, and I do not assume any background for this particular course, but it is a very intensive course. So, the phase of the course is going to be you know not very slow, but at the same time it is not going to be racy either, but the students who are actually going to take this course seriously are requested to do the programming assignments they are also supposed to solve theoretical problems, which I am going to suggest, otherwise this course will not be understood you know properly.

The reason is a compiler is an excellent example of the theory being translated into practice and this is a wonderful example of that. So, let us see the motivation for studying a compiler.

(Refer Slide Time: 01:52)



So, compilers are really everywhere. So, if you look at the applications of modern compiler technology pickup the browser open it, and then immediately you know there would be HTML files, which are displayed on the homepage and that you are going to visit and so on. So, the HTML parsers are based on compiler technology and then behind the screen there is java script there is flash etcetera are running inside the browser.

And the interpreters for this java you know script and flash etcetera are also based on the modern compiler technology. Then of course, for the compiler itself we require machine code generation and for high level languages, whenever we need code generation we you know, we need to use compiler technology anyway, but then apart from that it has uses in software engineering as well. For example, software testing and then program optimization then in the security domain malicious code detection design of new computer architectures.

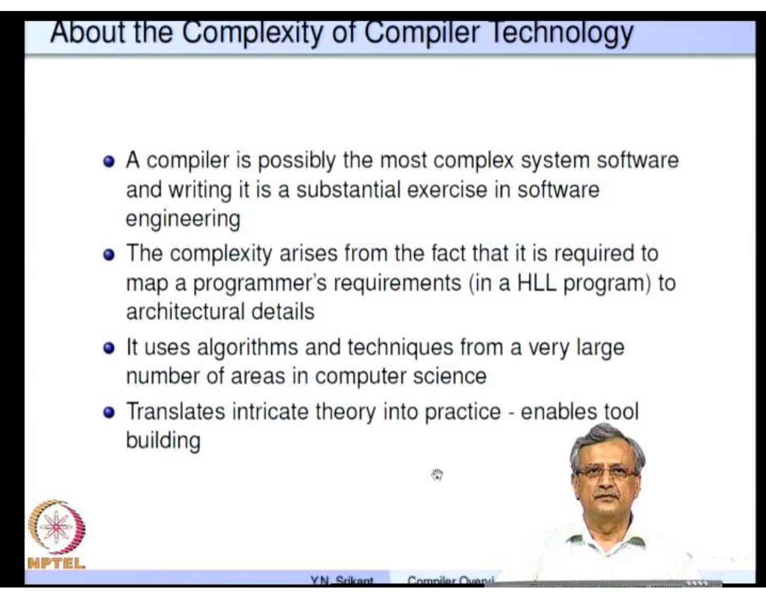
So, why are these important? For example, if you look at the development of a new processor nobody builds a processor you know right away even if the design is hundred percent accurate and all that the performance etcetera will all be know only after the hardware is built. Therefore, there is a simulator, which is built for a new CPU and then people also build compiler for that particular CPU. So, once the compiler is built you can compile right programs in c or c plus plus or any other language. Compile those programs and then run them on the simulator see what kind of performance it has is

giving, if necessary make changes in the hardware. So, that is called a compiler in the loop hardware development and its very useful perhaps a very widely used by chip designers in various companies.

Then again in the area of hardware synthesis nobody really writes you know the low level assembly type of core for generating VLSI designs and so on and so forth. That is called RTL register transfer logic. People really write it very high description you know high level description languages called VHDL or VDL and so on. And then the compilers really generate the RTL from VHDL again compiler technology is involved here. And then a novel application of compiler technology is compiled simulation. Suppose you write a program is VHDL, how do you really find out the performance of the chip or whatever is designed using that VHDL. So, typically the program a compiler is used to generate a simulator.

And the simulator is actually a computer program, which is generated for that particular program, which is being simulated. So, this is called simulation of the design and it is an example of compiled simulation. There is no interpretation of the design here and hence such simulations are much faster than interpretations.

(Refer Slide Time: 05:23)



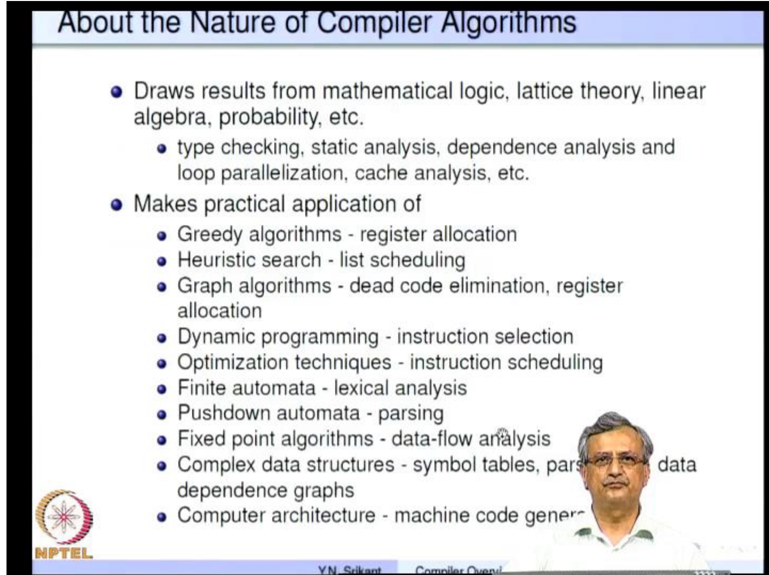
So, about the complexity of compiler technology, it similarly also necessary to say a few words about this aspect, if you look at the compiler, it is possibly the most complex

system software and writing it is a substantial exercise in software engineering. So in fact, in our institutes whenever somebody teaches courses on compiler design.

The associated assignments are really small compilers themselves and writing these is a fairly large software engineering exercise. The complexity of a compiler really arises from the fact that it is required to map a program as requirements, which are written in high level languages to architectural details. So, we are really talking about a c program and then it is translated into a machine level program. So, in between you know the compiler has to actually travel a long distance it is not as if it is a very simple operation. So, there is a huge amount of complexity here. So, we are going to discuss this particular complexity and the complex operation and its details in our course.

A compiler uses you know algorithms and techniques from a very larger number of areas in computer science we are going to see some examples very soon. So, it is not in that compiler technology is subject on its own, it has to borrow a huge number of techniques and algorithms from other areas in computer science. And compiler translates I already mentioned this, intricate theory into practice. So, you take, for example, a context free grammar specification, which is very formal, very precise and absolutely essential for writing a compile of describing a language. It can be translated into a parser with minimal effort there are tools such as YACC, which do this. So, this enables tool building. So, tools take very abstract specifications and generate programs from these specifications.

(Refer Slide Time: 07:42)



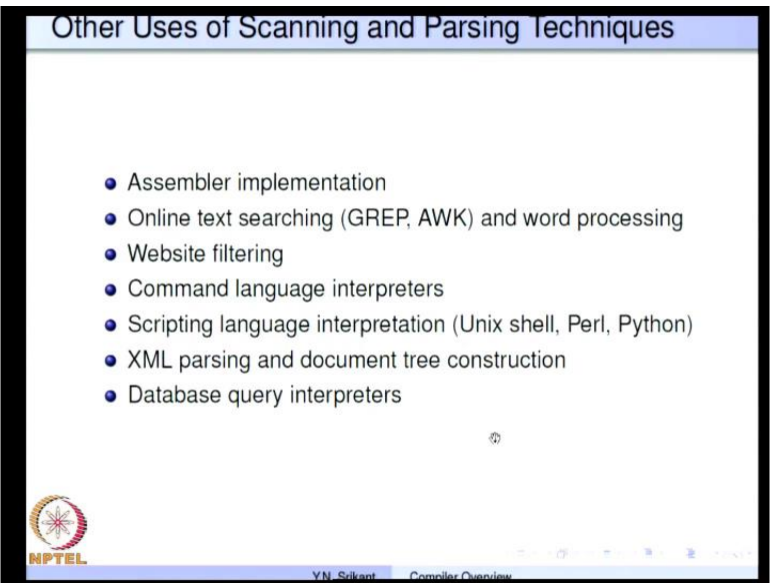
So now, let us look at the type of algorithms which are used inside a compiler and see and let me show you how we require many types of algorithms here. So, we require results from mathematical logic, lattice theory, linear algebra, probability, etcetera, etcetera. So, where are these used? For example, mathematical logic is used to you know in developing type checking theory then the lattice theory is used in developing static analysis, dependence analysis is heavily based on linear algebra and loop parallelization is based on dependence analysis.

So, cache analysis uses both static analysis and probability theory. So, in other words a very deep mathematical background is required to develop new compiler algorithms. In our course we are going to study the existing compiler algorithms, but we will also look at some of the, you know the bases which actually makeup this particular topic. Then there are practical applications of many algorithms. For example, greedy algorithms are used in register location. So, heuristic search is used in list scheduling, which is a part of a code generator, graph algorithms are used in dead code eliminations register location etcetera, etcetera.

Dynamic programming is used in instruction selection that is how to generate machine code. Optimization techniques are used in instruction scheduling, finite automata play a part in lexical analysis, pushdown automata very helpful for parsing, fixed point algorithms are used for data analysis, very complex data structures such as symbol tables

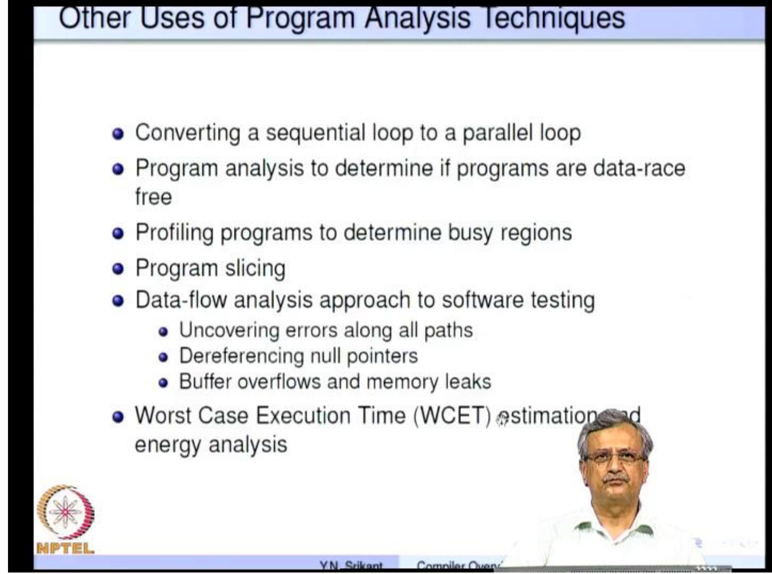
parse trees data dependence graphs are all going to be built. So, we use you know trees, balanced trees, graphs, etcetera for such applications. Computer architecture itself uses machine these used in the knowledge of computer architecture is used in machine code generation.

(Refer Slide Time: 09:58)



And then what are the other uses of some parts of compiler technology, some aspects of compiler technology. For example, scanning and parsing techniques are of course, used in compilers, but they are useful for assembler implementation there useful for online text searching for example, GREP and AWK, which are available in Unix are based on you know, the scanning techniques. Word processing, website filtering, command language interpreters that is for Unix for example, shell you know. So, here command language cum scripting language. So, interpreters for such languages are useful scripting language interpretation again Perl, Python, Unix shell, XML parsing document tree construction, database interpreters the list is very big. So, I have mentioned only few of them which are very important.

(Refer Slide Time: 10:57)



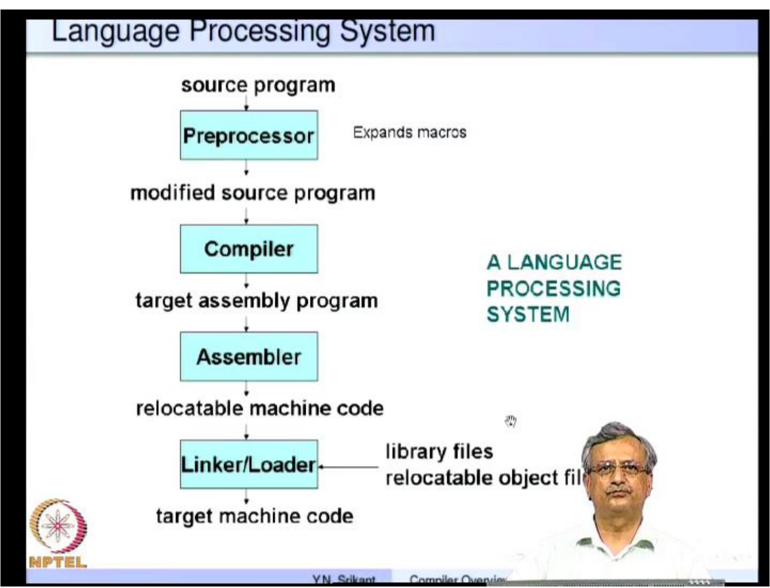
What about program analysis. So, one part of a compiler is scanning and you know, perform scamming and parsing and another part of compiler performs program analysis. So, program analysis techniques are useful in converting sequential programs to parallel programs. And very important it can be used to determine if programs are data race free. In other words are they two a parts of the program two threads actually accessing the same locations etcetera, etcetera can all be decided using program analysis. Profiling programs to determine busy regions of the core well, if this is done then we can possibly make that particular region of the core much more efficient.

Programs slicing techniques are used in debugging. So, a slice of a program is one small part of a program. So, data flow analysis approach to software testing is again based on program analysis for example, uncovering errors along all paths dereferencing null pointers, buffer overflows, memory leaks these are all common errors which, actually occur in programs. And to some extent taking detecting such problems using software testing requires a data flow analysis approach. Then worst case execution time estimation and energy analysis, if you look at a program it is very difficult to say what is the worst case time that it requires because time of execution for a particular program is based on its input, in depends on the input.

So, finding out the worst case input is a very hard problem and worst case execution time estimation is also equally hard and uses, it uses program analysis techniques. Energy

analysis implies the, rather computation of the amount of energy that a program takes. This is as difficult or more difficult than time analysis.

(Refer Slide Time: 13:08)



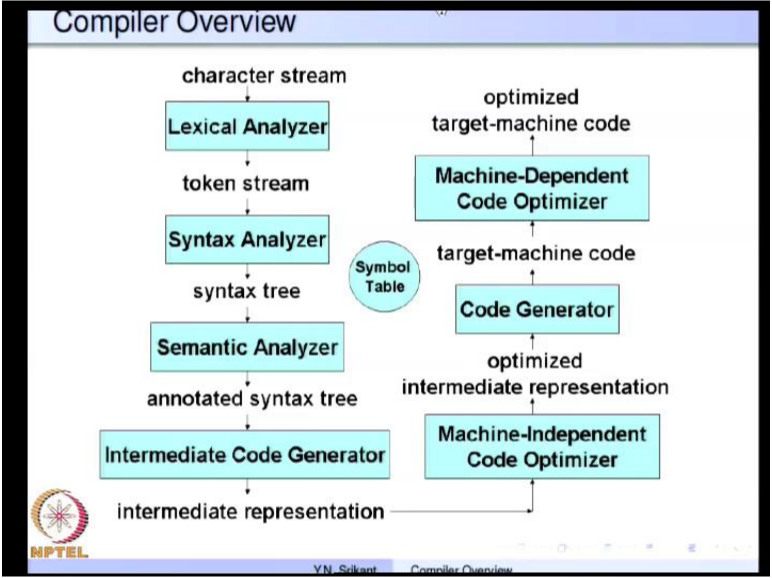
So, program analysis techniques are used in energy analysis as well. So, that is about you know, the motivation to study this subject called compiler design. So, let us begin with this block diagram which talks about a general language processing system. So, to begin with we have for example, here a preprocessor which takes a source program as in input and then outputs modified source program.

So, even in a c compiler we have such preprocessor. So, for example, you write hash define macros or hash include you know directives inside a program then such macros are expanded by the preprocessor. And the preprocessor expands and provides the source program as input to the compiler. A compiler itself takes a clean you know expanded source program in a high level language such as c or c plus plus or any other language and outputs is assembly program for the particular machine. So, we will see the details of a compiler in a short while. The target assembly program is input to the assembler, which takes the mnemonics in the assembly language and translate them to actual binary machine code.

Finally the assembler output is called as a relocatable machine code, which is combined with library files and relocatable you know object files of from other sources by the

linker and loader. To provide the target machine code which can run on a particular machine.

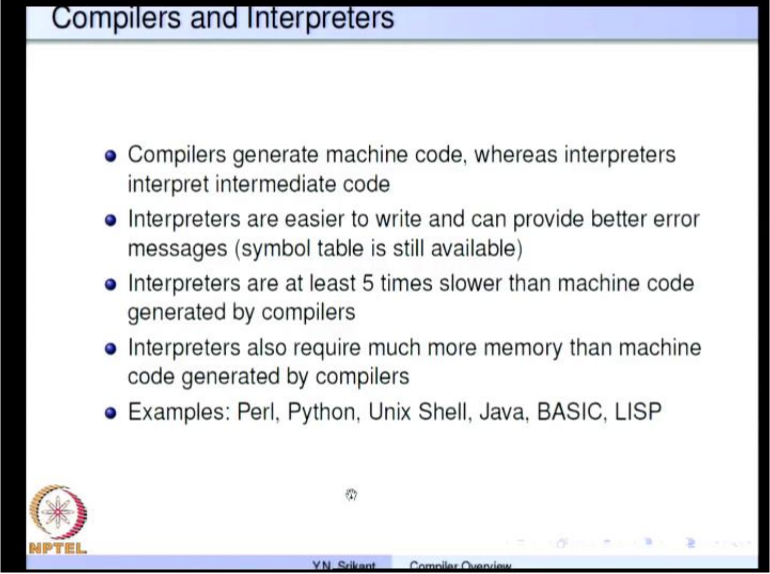
(Refer Slide Time: 14:53)



So now, the zero in on the compiler itself, so compiler consists of many blocks. They are all listed here, lexical analyzer is the first one and the output of that goes to a syntax analyzer. The output of which is again fed to a semantic analyzer following that is the intermediate code generator. And then comes the machine independent code optimizer, the machine code generator and finally, the machine dependent code optimizer.

And all these parts of a compiler use a data structure called as a symbol table, which is actually somewhere in the middle. So, we know have for example, let us look at lexical analyzer and see what it does. A lexical analyzer takes as input a character stream. So, we are now going to take each of these blocks and study them in detail. So, let us begin with this lexical analyzer, but before we begin with lexical analyzer, I must hasten to add that there is a difference between what are known as compilers and interpreters. So, if we look at the previous slide, the compiler consists of this entire you know seven blocks along with the symbol table. Whereas, an interpreter stops after the intermediate code generation stage and the output of the intermediate code generator is fed to an interpreter.

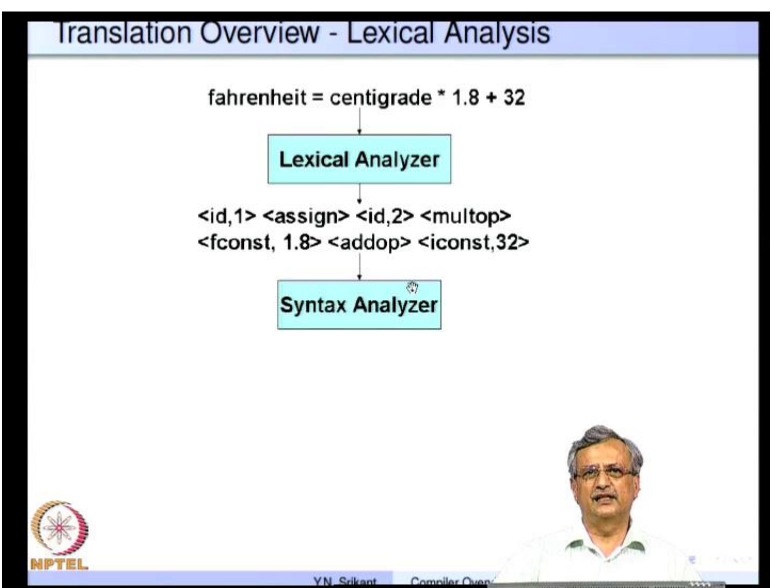
(Refer Slide Time: 16:37)



So, let us see the difference between these two. Compilers generate machine code whereas; interpreters you know interpret intermediate code. Of course, interpreters are only fifty percent of a compiler so they are easier to write and can provide better error messages because we have, we still have access to the symbol table and so on. But the catch is interpreters are five times lower or more actually more than five times lower that machine code generated by compilers. So, running machine code is much faster, but interpreters are easier to build. So, people tend to write you know interpreters for certain types of languages.

Whereas they try to write compilers for languages, which are used to write professional programs, interpreters also require much more memory. So, and then you know even the compilation lexical analysis parsing etcetera is all the time required by the interpreter itself its added to the time required by the interpreter. So, they require much more memory, much more time than machine code generated by compilers and there are very famous examples, Perl, Python, Unix shell, java, basic, lisp, etcetera are all you know interpreter based languages.

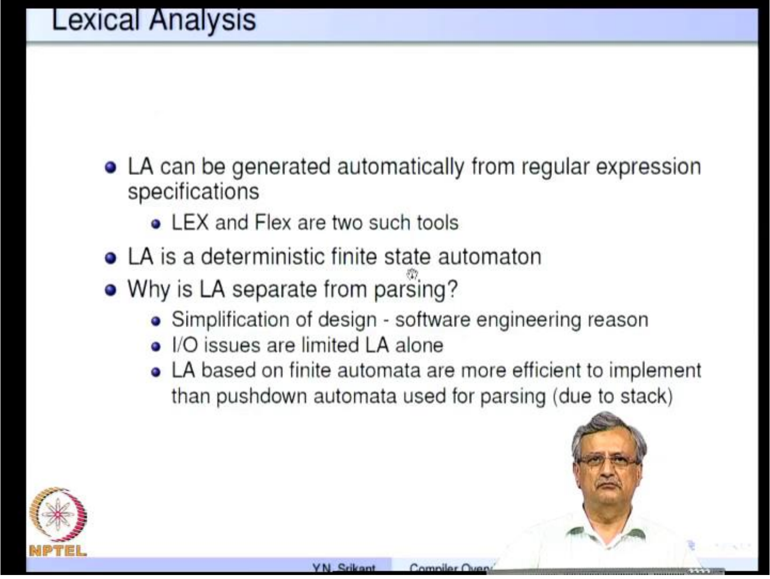
(Refer Slide Time: 17:55)



Now, let us get back to the block diagram and lets look the lexical analyzer in some detail. A lexical analyzer takes source program, for example, here there is a line Fahrenheit is equal to centigrade star 1 point 8 plus 32. This is an assignment statement in any particular language you know there is no need to talk about a particular c or c plus plus such assignments are available in every language. So, now, the lexical analyzer takes as input such sentences from a source program and then it generates what is known as a token stream. So, in this particular case the two names Fahrenheit and centigrade are all are both called identifiers. So, Fahrenheit is coded as identifier one and centigrade is coded as identifier two. The equal to sign which corresponds to assignment is actually made an assign is made into a token of kind of sign.

Similarly, multiply operator is multop and then plus is made into and addop, the constant 1.8 is a floating point constant, iconst you know thirty two corresponds to the number 32. So, in other words, we now have a stream of these tokens, the first part of the token i d assign, i d multop, fconst, addop, iconst these are typically integers, they are numbers. Whereas, the second part whenever it is present is actually gives you hints about what type of you know the token it is. For example, if it is i d then that 1 and 2 may point to a table within this is 1 and 2 containing this string corresponding to the identifier. The iconst, fconst you know the second part will tell you the value of that particular number and so on and so forth. This is the input to syntax analyzer.

(Refer Slide Time: 20:12)



So, now let us look at the reasons why a lexical analysis is required very briefly. So, lexical analyzers can be generated automatically from regular expression specifications. So, for example, LEX and flex are two tools available in Unix and if we feed a regular expression specification, we are going to study these specifications later in the course outcomes a program, which works as lexical analyzer. The lexical analyzers are actually is a deterministic finite state automation each one them is a finite state machine and we will learn what these are in the coming lectures.

But now, let us answer the question why is lexical analysis separate from parsing? It is not a theoretical limitation, in other words the next phase of a compiler called parser can actually incorporate the lexical analysis also. There is not much difficulty as far as theory is concerned, but practically if you look at the design a compiler is an extremely large piece of software millions of lines of code. And simplifying the design making it modular is the only way its complexity can be controlled. So, simplification of the design by making lexical analyzer a separate module is a reason because of software engineering purposes.

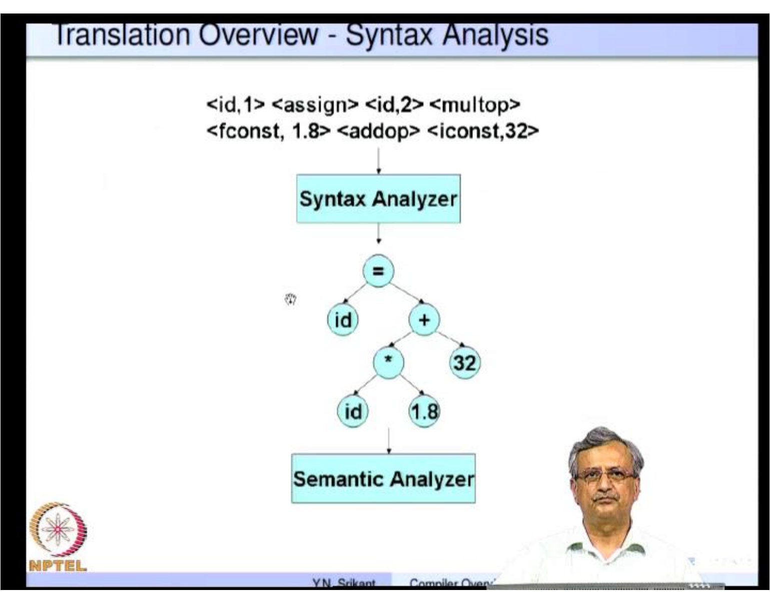
The second reason, input, output issues are very limited you know very serious and it is not a good idea to distribute such input output you know issues allover a compiler. They are best handled in one module and in this case in a compiler lexical analyzer handles all the input output issues. So for example, it reads programs you know the from the source

file and then any errors etcetera are all actually listed by the lexical analyzer after collecting them from various parts then and so on. So, lexical analyzers based on finite automata are efficient to implement than pushdown automata, which are used for parsing. This is a very deep reason.

So, as I already mentioned a lexical analyzer is nothing but a deterministic finite state automaton. And parser as we will see later corresponds to a pushdown automaton. A pushdown automaton uses a stack for its operation. So, if we actually try, I mean doing lexical analysis with a pushdown automaton then we will end up actually pushing a lot of symbols on to the stack then popping them of the stack and so on, which are very inefficient operations. And therefore, using incorporating lexical analyzer into a parser makes it very inefficient compared to the making a finite automaton based lexical analyzer. So, these are the reasons why lexical analyzers are separate.

And of course, if you look at the previous slide, as I said each one of these tokens. The first part of the token is an integer so storing these integers is much more efficient than storing the characters corresponding to the source program. So, it is actually a very you know succinct and compact way of may you know, giving the program to the syntax analyzer.

(Refer Slide Time: 24:00)

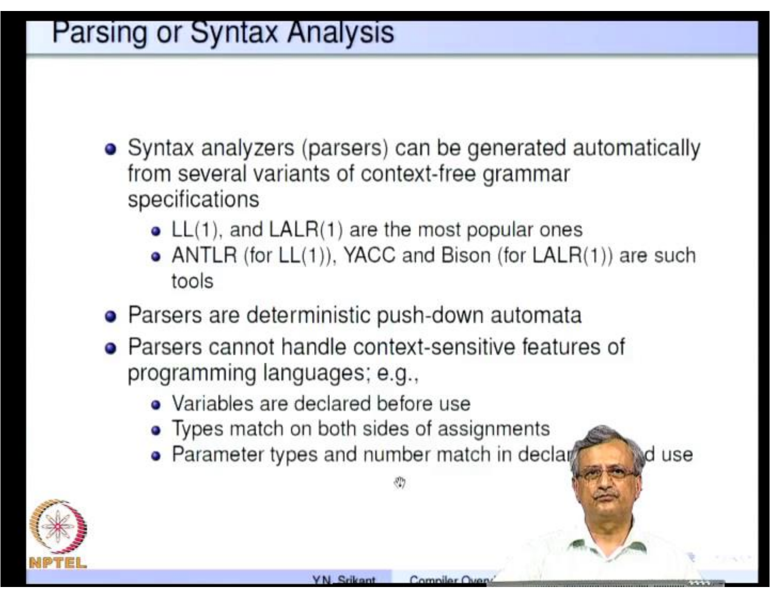


So, then once we understand lexical analysis we must see how it actually helps syntax analysis the output of the lexical analyzer is fed to a syntax analyzer. So, we have these

tokens coming into a syntax analyzer. The syntax analyzer looks at the tokens and finds out whether the syntax is according to a grammar specification.

In other words the assignment statement must have a variable on the left side an expression on the right side and so on. So, does this have you know the assignment operator is it correct or is it there is there a mistake and whether the plus star etcetera are all properly inserted into the expression these are all the syntax checks that syntax analyzer can perform based on grammar specification which is given to it. The output of a syntax analyzer is a tree, this small tree is called as a syntax tree or abstract syntax tree. So, for example, here it shows the structure of the assignment statement above, this was Fahrenheit equal to you know, let us look at it Fahrenheit equal to centigrade into 1 point 8 plus 32. So, here this i d corresponds to Fahrenheit, this i d corresponds to the next identifier and then we have the assignment symbol plus star 1 point 8 and 32. So, everything is working out well. So, this structure is shown in the form of a syntax tree, which is the input to a next phase of a compiler called the semantic analyzer.

(Refer Slide Time: 25:44)



So, syntax analyzers can be generated automatically from context free grammars. So, we will learn about these specifications a little later, but right now, it suffices to say that the you know there are tools to do this. For example, the YACC and ANTLR are two such tools.

So, they handle what are known as you know L A L R 1 grammars that is YACC and bison and ANTLR handles what are known as L L 1 grammars they generate c programs or c plus plus program which correspond to these parsers you know and they can be used by the compiler. So, as I already said, the parsers are based on pushdown automata. So, they are actually what are known as the deterministic pushdown automata and there is a reason, why we need the next phase of analysis called semantic analysis. The reason is parses cannot handle context sensitive features of programming languages. Let me give you few examples.

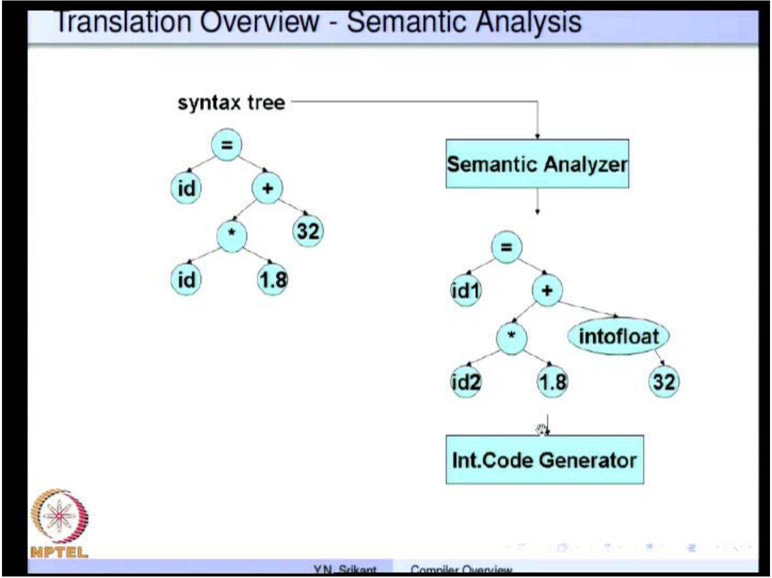
So, if you are looking at the syntax alone that is whether the assignment statement has an identifier on the variable or identifier on the left hand side a properly formed expression on the right hand side this forms a syntax. But if you say, can I check whether an integer variable is being assigned a floating point expression that is type matching on both sides of assignment; this cannot be handled by a parser. So, there are theoretical limitations here, context free grammars cannot express such features of programming languages. Another feature which is here is variables are declared before use. So, we all know that, we declare variables int a you know float b etcetera, etcetera.

And then at the time of using a and b, that is suppose a is used in an assignment statement or in expression. The compiler makes sure that a was declared before and it also makes sure that the type corresponding to the usage of a is the same as the type corresponding to its declaration. So, this feature variables declared and then checked after use or even declared before use cannot be captured using context free grammars. They require higher form of grammars called as context sensitive grammars and this is a reason why we need another phase of analysis called semantic analysis.

Another very important feature of programming languages, which cannot be captured in a context free grammar and hence cannot be caught mistakes of this kind, cannot be caught by a parser is here. You know parameters types and number match in declaration and use. So, we declare a large number of parameters in program functions, which we write and each one of these parameters has a type attached to it. So, when we actually call that function or procedure, we have to make sure that the actual parameters that we supply are of the same type and the numbers of parameters, we supply are exactly the same as the one in the declaration. So, parameter type and number match in declaration and use this property cannot be caught by the parser. Is the misuse of this property

cannot be mistakes of this type cannot be caught by a parser. And therefore, semantic analysis is supposed to rake care of it.

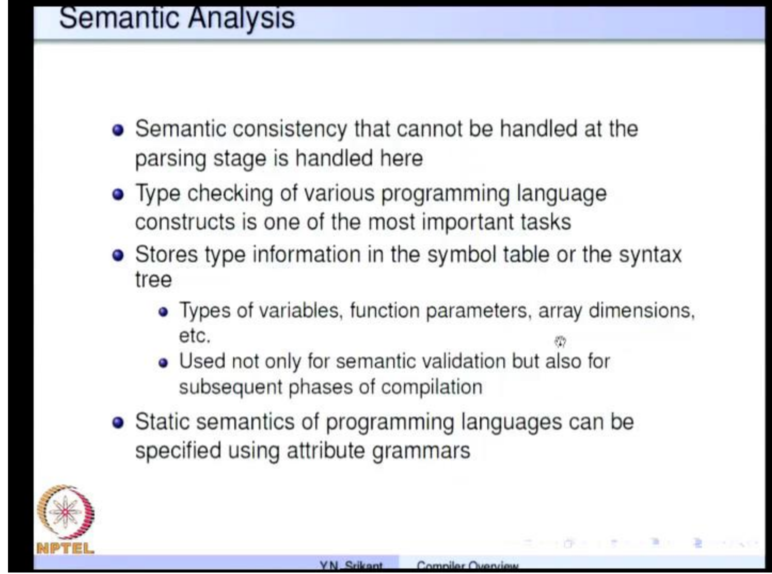
(Refer Slide Time: 29:50)



So, the next phase is the semantic analysis phase, the input to the semantic analysis phase is a syntax tree. So, we already know that syntax tree is produced by a syntax analyzer, it goes into a semantic analyzer and the output is also a syntax tree, but it is actually modified syntax tree. In other words, there are changes made to this particular syntax tree. So, that the types of operants are all taken care of. For example this expression i d star 1 point 8 corresponds to a floating point type, you know both the i d 2 and 1 point 8 are floating point types, but then we are adding an integer called 32. So, if there is some violation now, you cannot really add floating point numbers and integers directly because the representation of these numbers is different inside a compiler.

So, what does the compiler do inside a machine? So, the representation is different inside a machine. So, what does the compiler do? It converts the number 32 into a floating point number and then proceeds to generate you know machine code for this particular statement. So, and this is recorded faithfully in the syntax tree, which is called as the annotated syntax tree. So, that the code generator need not worry too much it just goes ahead with code generation looking at what is available in the annotated syntax tree.

(Refer Slide Time: 31:34)



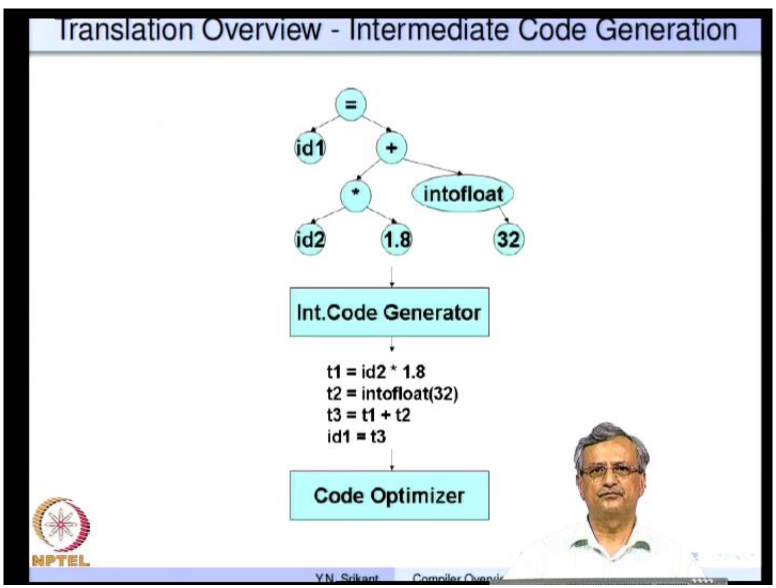
Semantic consistency that cannot be handled at the parsing stage is handled here. So, I already give you examples of this. So, I am in the same thing is repeated here. Type checking of various programming language constructs is one of the most important task. A semantic analyzer also stores information in the symbol table or the syntax tree itself.

So, each node of the syntax tree could store information corresponding to that particular node. What is the type of information that is stored, what are the types of variables? Is it int, float, is it a stag, is it an array, etcetera. What are the types of function parameters or what are the dimensions of an array etcetera, etcetera. So, these are the informations that are stored in a symbol table by the semantic analyzer. This information is used not only for caching errors semantic validation as we know it.

But it is also used for subsequent phases of the compilation process itself, for example, the code optimizer will also require information about the types of operants in order to perform certain types of optimization. And then the machine code generator needs to know the types of variables in order to generate the appropriate types of instructions. So, both these phases require access to the symbol table. So, this is the semantic analysis phase builds the symbol table and that is the data base, which is used by the, you know the phases later in compilation. Static semantics of programming languages can be specified using what are known as a attribute grammars.

So, we are going to study attribute grammars also in our course a little later of course, and attribute grammars actually are an extension of context free grammars. They are useful for specifying the semantics what are known as static semantics of programming languages and that is possible to generate the semantic analyzers semi automatically from such specifications of attribute grammars.

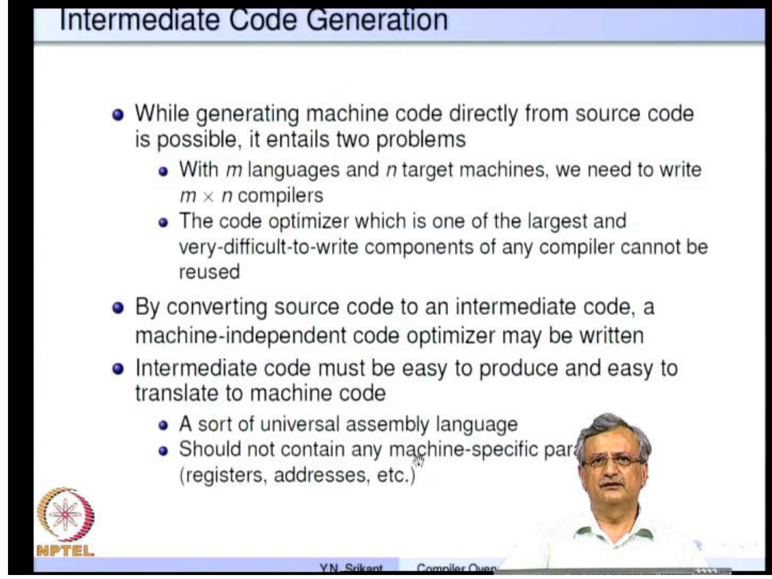
(Refer Slide Time: 33:55)



The next phase of a compilation is the intermediate code generation phase. So, the annotated syntax tree, which is output from a semantic analyzer is the input to an intermediate code generator and the output of the intermediate code generator goes to a machine independent code optimizer. So, let us see what this intermediate code generator has done on our example.

So, here is a small tree corresponding to an assignment statement. So, it is very obvious that we need to do these this multiplication first then the intofloat and then the plus and finally, the assignment. So, and that is the order in which the intermediate code has been generated. So, I will tell you why intermediate code after a few minutes, but let us understand this code to see what the intermediate code generator has done. It has generated t 1 equal to i d 2 into 1 point 8 corresponding to this expression, it has generated t 2 equal to intofloat 32 corresponding to this expression, t 3 equal to t 1 plus t 2 corresponding to this small tree. And finally i d 1 equal to t 3 corresponding to that assignment operator.

(Refer Slide Time: 35:19)



So, an intermediate code program has the same semantics as the original source level program, but it is at much lower level compared the source level program, but I must you know mention that it is not a machine language. So, let us see why we require such intermediate code and what exactly we do with it. So, when generating machine code from directly from source code is definitely possible there is no theoretical or practical limitation, there are two problems associated with this approach. The problem is you need to write too many compilers, suppose we want to write compilers for m languages and let us says you have n target machines for which you require compilers.

So, if we directly write you know generate machine code without generating any other form of intermediate code we need to write m into n number of compilers. Now, inside a compiler the code optimizer is perhaps one of the largest and the most difficult to write component. And it so happens that if we write, you know compilers which generate machine code directly, we will not be able to reuse any part of this optimizer. To give you the, you know to some inkling of what is involved about fifty percent of the compiler source code is for you know the front end that is the lexical analyzer the parser and as semantic analyzer and let us assume that there is an intermediate code generator.

So, all this four components together form about 50 percent of the source code of a compiler. The other 50 percent is for the code optimizer and the machine code generator. So, out of these about 30, 35 percent is meant for just this code optimizer and the other

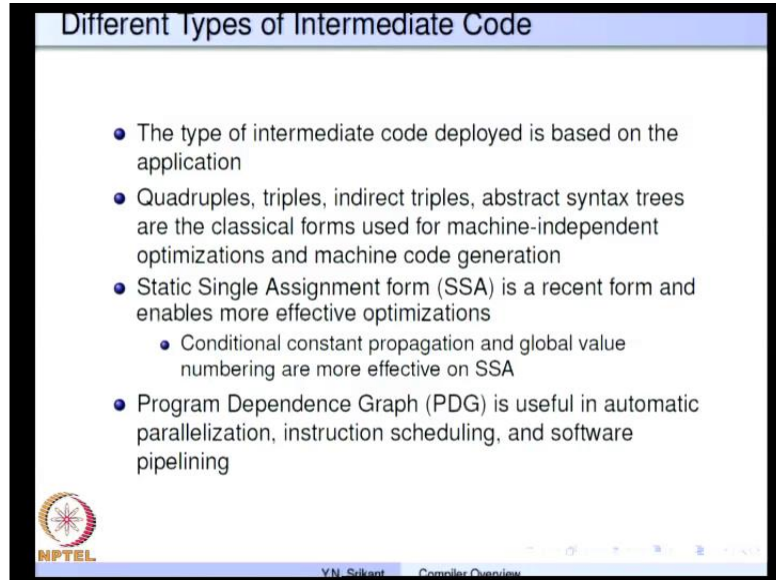
20 percent 25 percent is for machine code generation and machine code optimization. So, a very large part of compiler 30 to 40 percent, if it has to be rewritten again and again you know for every language and ever machine it is a waste of effort. What we try to do is to generate intermediate code from the source code and then this intermediate code will be the same for many languages and many types of target machines. So, whether it is c or c plus plus or fortrune or java the intermediate code will be very similar.

So, in fact, for G C C the same type of intermediate code is used by the entire family of G C C, G N U compilers really, G C C is one of them. We have G N U compilers for fortrune java and c plus as well. So, all these compilers use the same form of intermediate code. Once we have the same intermediate code for many languages, we can write a single machine independent code optimizer. So, in other words that 35 percent component is going to be used for different languages and it is a common module, which will be used or different compilers as well.

And of course, so once we do that, we do not require m into n compilers, but we will really require m plus n compilers. So, for m different languages we require different front ends that is lexical analyzer parser etcetera, etcetera. And we also require n numbers of code generators, which are specific to the various target machines, but the intermediate code optimizer is going to be common between these. So, its strictly speaking you really requires m plus n plus 1, number of components for then compilers. Intermediate code must be easy to produce it should not be as complicated as machine code, otherwise the effort spent in writing a machine a machine code generator and a machine independent or intermediate code generator will be similar.

So, we do not want that to happen, we want the intermediate code to be very simple and very easy to produce. This is some type of a universal language which can be mapped to any, you know machine code and it should not contain any machine specific parameters, no registers, no addressers, etcetera, etcetera.

(Refer Slide Time: 40:30)



There are different types of intermediate code as well. So, the type of intermediate code that is deployed actually is based on the application. So, quadruples, triples, indirect triples, abstract syntax trees, these are all classical forms of intermediate code. They have been in existence for decades and they have been used in commercial compilers machine, independent optimization, machine code generation in various types of compilers.

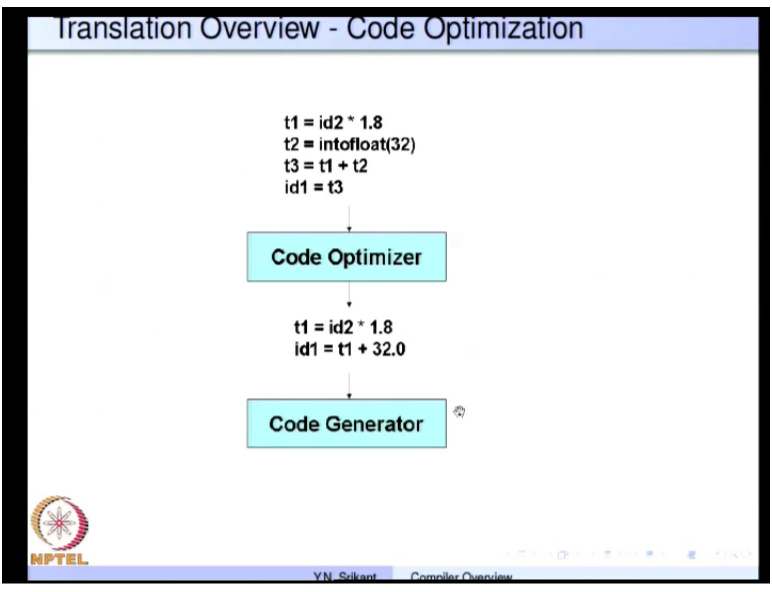
What something we are going to study later in our course. Then there is a form of a intermediate code called the static single assignment form, which is a recent one when I say recent it is a or the past seven eight years, it is been deployed in the G C C compiler. And using this type of intermediate code makes some of the optimizations more effective. For example, conditional constant propagation global value numbering these are two very important optimizations, which are carrying out by a good compiler not necessarily simple compilers, but gold quality compilers and these optimizations are more effective on an intermediate code such as SSA rather than the quadruples or triples. So, modern compilers nowadays invariably use SSA form as one of their intermediate forms.

So, in other words, we may end up using two or more types of intermediate code in our compiler to begin with it could be quadruples or abstract syntax trees it may be translated to static single assignment form for better optimization and again translate to another

type of intermediate code for better machine instruction machine code generation. Finally, program dependence type of graph is another type of intermediate code, which is useful in automatic parallelization, instruction scheduling, software pipelining, etcetera, etcetera. This is the, I know, intermediate code, which shows the dependence between various types of statements in the program.

For example, if there are two assignment statements in the program, one of them produces the variable a, the other one uses a variable a then there is a dependence between these two statements. So, this is a nutshell what a program dependence graph shows. So, this type of dependence is useful for automatic parallelization and other operations, which I have mentioned here.

(Refer Slide Time: 43:21)

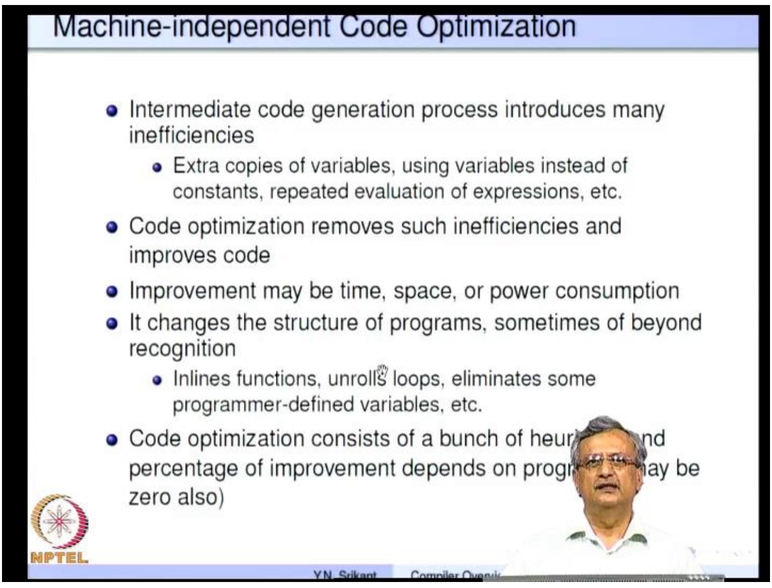


Now, the code optimizer is the next phase, which takes as input the intermediate code and generates, you know produces very efficient code optimizers, code, intermediate code and inputs into the machine code generator. So, I have already told you what code optimizer is it improves code.

So, lets see how it operates here. So, here the first statement t 1 equal to i d 2 star 1 point 8, remains as it is there is not much we can do in that, but it is not necessary to retain the second statement, which is t 2 equal to intofloat 32. So, we might as well create a floating point constant 32.0 and generate a new quadruple i d 1 equal to t 1 plus 32.0 instead of the three quadruples, which are stated here. So, we have actually reduced the

number of quadruples from 4 to 2 you know, it a really good achievement because for the short program it implies 50 percent improvement.

(Refer Slide Time: 44:35)

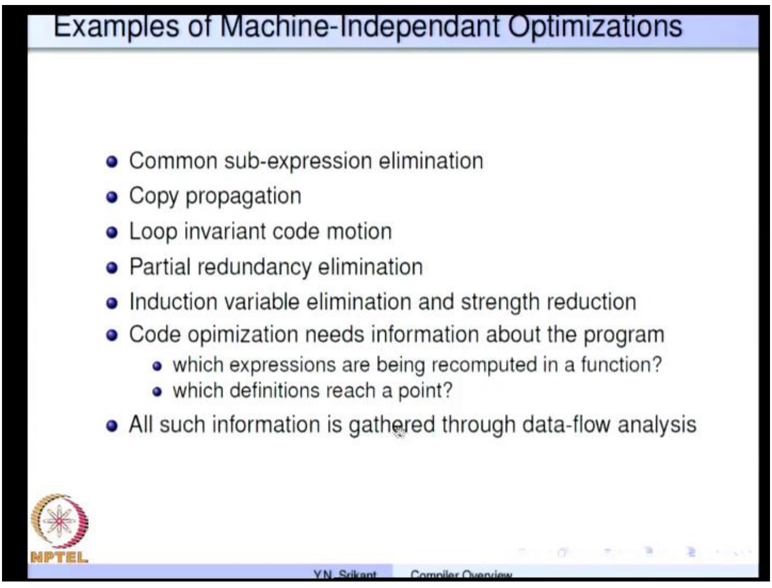


The machine independent code optimization actually becomes necessary because intermediate code as I said is a very simple type of code and the intermediate code generation process introduces many inefficiencies. So, there are extra copies of variables then instead of constants we actually put them into variables and then use that variable and then some expressions are evaluated again and again. So, these are all inefficiencies which result from intermediate code generation. So, code optimization removes such inefficiencies and improves code and the improvement may be either time or space or power consumption. So, depending on what you require. So, for example, for very efficient servers you require times and memory optimization whereas, for embedded systems it could be power consumption which needs to be minimized. Code optimizers also change the structure of programs and sometimes they change it beyond recognition.

So, they may inline functions, they may unroll loops. So, what does inlining of functions mean, when there is a function called instead of making a subroutine call for that particular function. The code of that particular function is embedded into the program and that is called inlining. Unrolling loops of course, is easy and fairly well known, we do not execute a loop 100 times instead of that we may execute it only ten times, but the body of the loop is made ten times bigger, ten iterations are actually inlined inside the

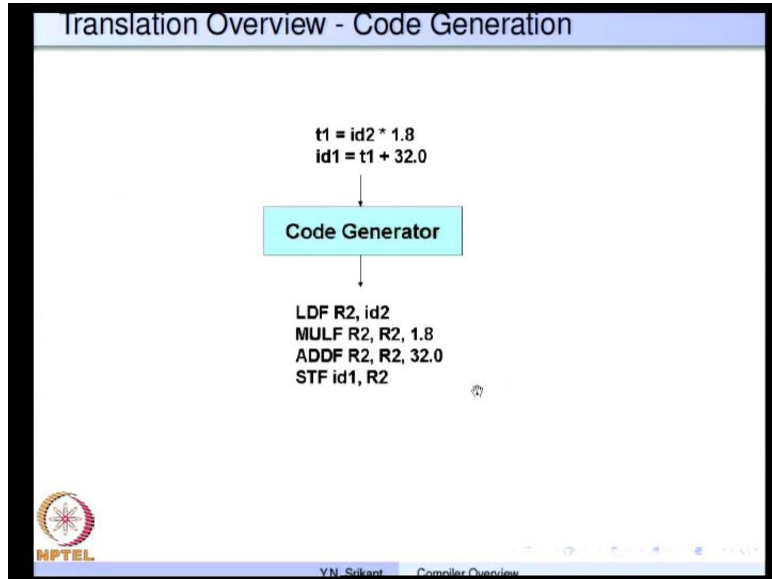
loop and that is called unrolling of a loop. and eliminating some program or defined variables. So, if there is a counter called i and there is another variable j, which is dependent on i in such a case it may be possible to remove i itself, eliminate i. So, this is called induction variable elimination. Code optimization is actually a bunch of heuristics and the improvement may actually be just 0. You never know whether there would be improvement or not, but some programs yield improvement some other programs may not any yield any improvement.

(Refer Slide Time: 46:57)



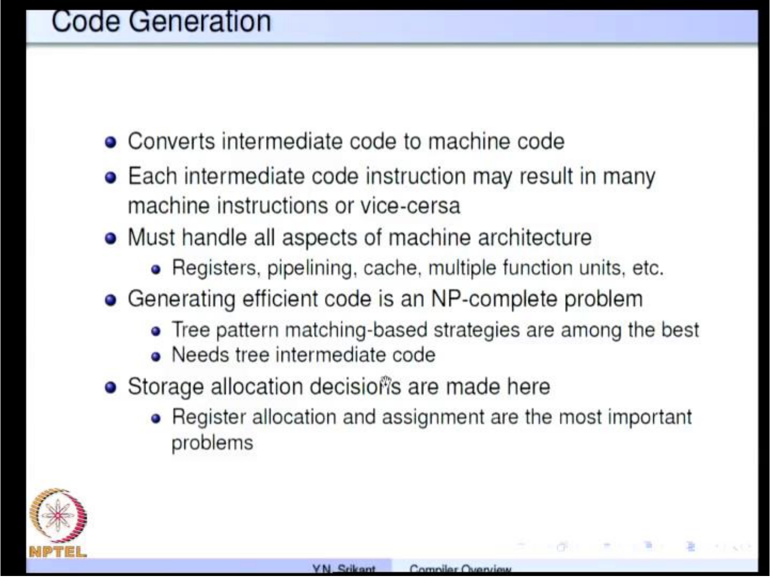
So, there are different types of machine dependent optimizations for example, common sub expression elimination, copy propagation, loop invariant code motion, partial redundancy elimination, induction variable elimination and strength reduction, code and to perform these optimizations we require information about the program. What type of information which expressions are being recomputed in the function which definitions reach a particular point. So, analysis of the program to determine such information and storing it in particular way is called data flow analysis and we are going to study this part of a compiler towards the end of the course.

(Refer Slide Time: 47:41)



Finally, the machine code generation so it takes intermediate code as input and outputs a particular type of machine code. In this case you can say load floating point and then multiply floating point, add floating point, store floating point corresponding to these two instructions are being generated here.

(Refer Slide Time: 48:03)

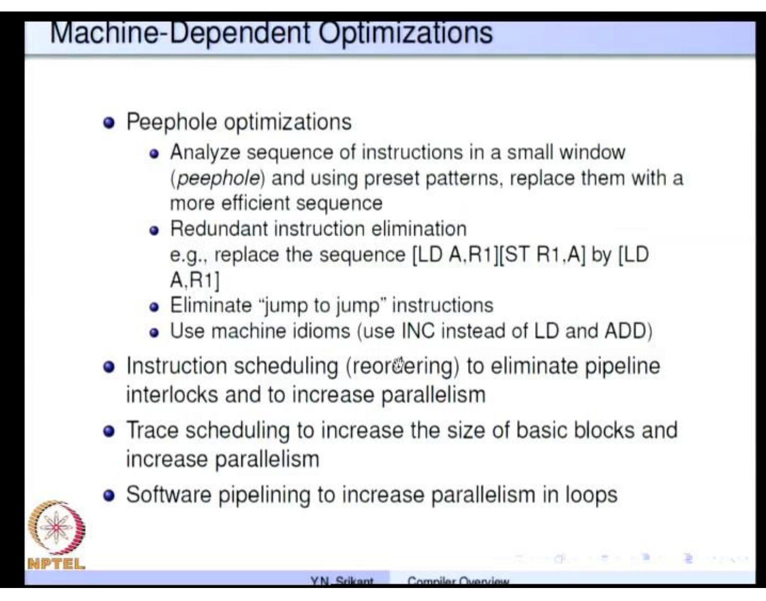


So, it converts intermediate code into machine code and each intermediate code instruction may actually result in many instructions. Otherwise it is possible that many intermediate code instructions actually give rise to only one single machine instruction

depends on the complexity of the machine. It must also handle all aspects of machine architecture registers, pipelining, cache, multiple function units, multiple course whatever all these aspects must be handled by the machine code generator.

Generating efficient code is a very difficult problem is usually NP complete and there only for a very simple type of machines is can be done optimally. And generally tree pattern matching based strategies are among the best that are available. So, of course, we require tree intermediate code for this type of tree pattern matching based generation. Storage allocation decisions are also made here. So, you know register location which registers are used, which operant should go into, which register etcetera, etcetera are all solved in the machine code generation phase of the compiler.

(Refer Slide Time: 49:17)



There are also after machine code generation even the code that results is not very efficient. It is possible to improve it little more, for example, there are, what are known as machine dependent optimizations ok.

If you are listed here, there are, what are known as peephole optimizations. So, you analyze a sequence of instructions say 10, 15 in what is known as a small window and this window is called as a peephole. And using preset patterns you replace them with more efficient instructions. So, for example, load A comma R 1, store R 1 comma A well you know we are loading and then storing immediately. So, this is not necessary. So, you could just say load A comma R 1 get rid of the store instruction. sometimes there is no

need to if you have a some core where there jump instruction and the target is another jump. Then this jump to jump can be eliminated and replaced by a single jump. It is possible to use say increment instead of load and add.

So, these are called machine idioms and these form part of peephole optimizations. Instruction scheduling that is reordering of instructions to eliminate pipeline interlocks and increase parallelism. So, usually we should not make the pipeline het stuck there should be a free flow into the pipeline and out of the pipeline. So, reordering instructions to make this happen is called instruction scheduling and that is one of the machine dependent optimizations. And if the base is our programs are what are known as basic blocks which are single entry, single exit pieces of code. So, if they are very small there is no way you can increase the parallelism in the program.

So, we must make them bigger and this technique called trace scheduling is used to increase the parallelism available in the program by increasing the size of basic blocks. And finally, the software pipelining which is too complex to explain orally is a very sophisticated optimization, which is used to increase parallelism in loops. So, that brings us to the end of the overview and in the next lecture we are going to look at the details of lexical analyses, parsing etcetera, etcetera.

Thank you very much.