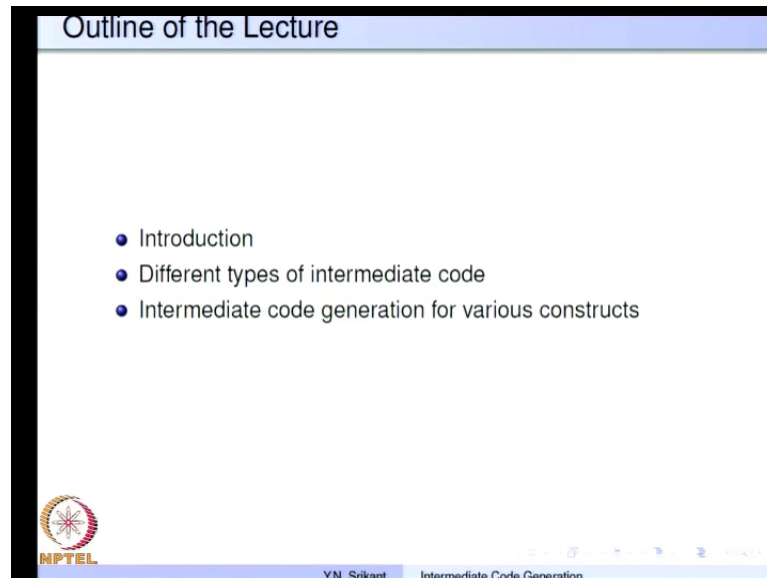


Principles of Compiler Design
Prof. Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

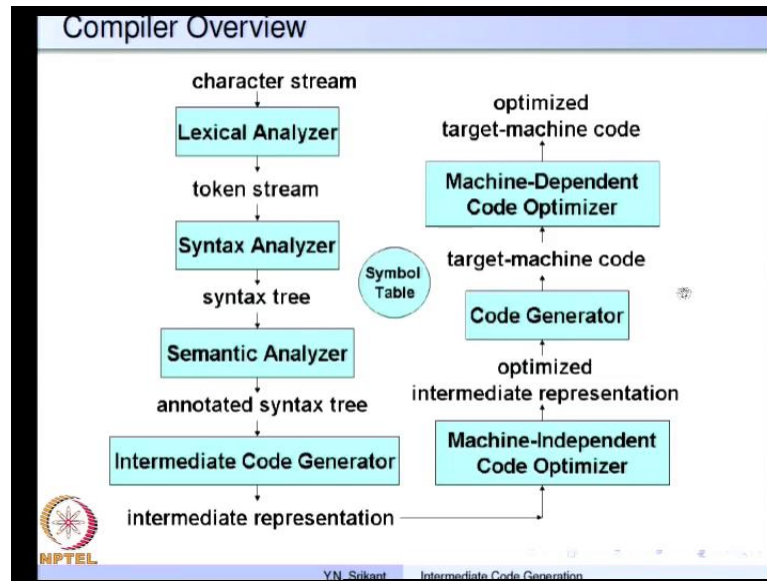
Lecture - 17
Intermediate Code Generation Part – 1

(Refer Slide Time: 00:20)



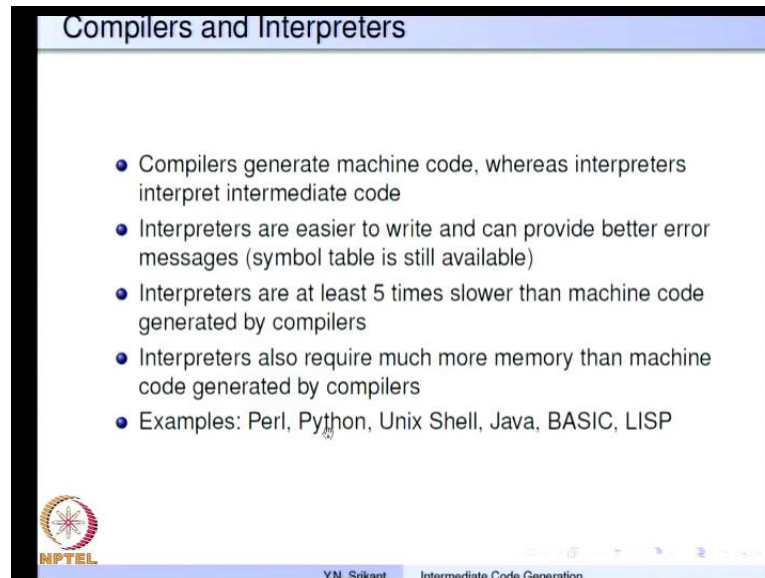
Welcome to the set of lectures on intermediate code generation. So, in this sequence of lectures, we are going to learn about different types of intermediate code; why intermediate codes are required? And we will also see how the attributed translation grammars can be used to generate intermediate codes for various constructs.

(Refer Slide Time: 00:40)



So, to begin with and to put the intermediate code generation phase in the right perspective, let us consider the compiler overview diagram that we have seen many times so far. So, we have... Once the character stream goes through the lexical analysis, syntax analysis and the semantic analysis stage, we get the annotated syntax tree over which intermediate code generation can be performed. So, the output of this will be sent to the machine-code optimizer. So, this is the perspective of intermediate code generation. So, we will look at the you know generation of the intermediate code using the SATG's, that is the synthesized attribute translation grammars. And we will also look at some aspects of code generation using LATG's, that is the L attributed translation grammars.

(Refer Slide Time: 01:38)



The slide is titled "Compilers and Interpreters" and contains a bulleted list of points. At the bottom left is the NPTEL logo, and at the bottom center is the text "Y.N. Srikant Intermediate Code Generation".

- Compilers generate machine code, whereas interpreters interpret intermediate code
- Interpreters are easier to write and can provide better error messages (symbol table is still available)
- Interpreters are at least 5 times slower than machine code generated by compilers
- Interpreters also require much more memory than machine code generated by compilers
- Examples: Perl, Python, Unix Shell, Java, BASIC, LISP

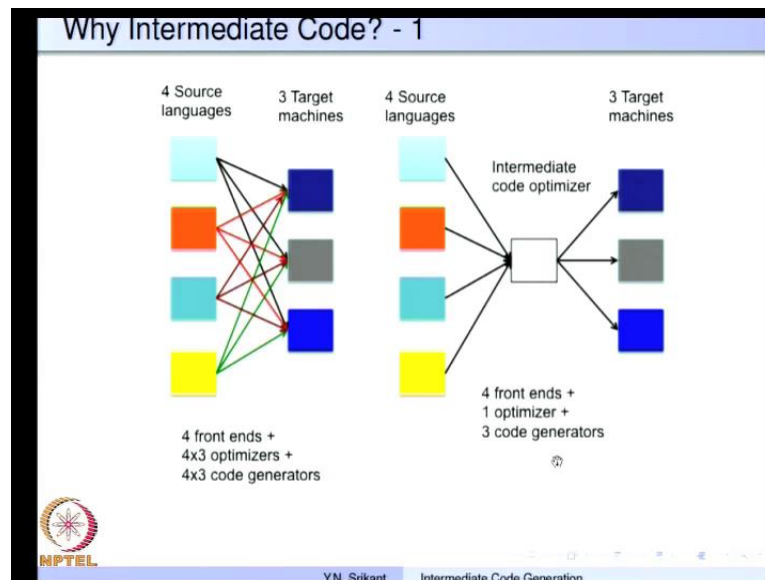
So, let us see the other users for intermediate code in the interpreters as such. So, what is the difference between compilers and interpreters? Compilers generate machine code and interpreters generate intermediate code; and then they continue with that process and interpret the intermediate code as well. So, when we say intermediate code is interpreted, the implication is the entire runtime environment that is required by the program to run is also provided by the interpreter itself.

In cases such as Java, the intermediate code is actually produced by the compiler, and then there is a separate interpretation phase; whereas, in other languages such as Perl, Python or even Unix Shell, BASIC, LISP, the compilation process to produce intermediate code and the interpretation process are in the same program. Obviously, interpreters are much easier to write and can provide better error messages than a compiler, because the optimization and machine code generation phase is absent in an interpreter. The symbol table is still available to an interpreter. And therefore, error messages are easier to provide; and better error messages can also be provided. But, the catch is interpreters are very slow; at least five times slower than the machine code generated by compilers.

To offset this problem or the deficiency, the Java runtime system and the interpretation system produce – actually provides what is known as a just-in-time compilation. So, in JIT compilers, the interpreter code is actually compiled into machine code and then run.

This is very useful if the code is going to be run again and again. So, in such cases, JIT compilers are probably very close in execution speed to the compiler code. Interpreters also require much more memory than the machine code generated by compilers, because interpreters all said and done also have the symbol table and other data structures. And they really need to simulate the entire machine environment in which the code is supposed to run. So, all these require much more memory than that required by the machine code, which is generated by compilers. So, I already said that, Perl, Python, Unix Shell, Java, BASIC, LISP are examples of interpreted code; whereas, the compile code we all know C, C plus plus, Pascal and many other languages. The compilers for these languages produce machine code.

(Refer Slide Time: 04:50)



So, now, the big question that needs to be answered properly. Why do we require intermediate code at all? The other option is you have source languages and you have target machines; I just write a compiler for the source language A and the target machine X. So, why cannot this be done? So, let us look at the implications of this process. So, let us take an example. There are four source languages and there are three target machines; and we want to implement all the four source languages on all the three machines. So, to begin with, we obviously require 4 front ends, which do lexical analysis, parsing, semantic analysis and intermediate code generation. And if the intermediate code is immediately converted to machine code within the compiler or it is also possible that the intermediate code is not produced at all. So, intermediate code could be at a very high

level such as an abstract syntax tree in these cases. And it will not be at lower levels such as quadruples that we are going to use in our intermediate language study.

So, for all practical purposes, we can say that, the source language is directly compiled into machine code. So, 4 front ends, which actually do the first part of compilation. Then we require 4 into 3 – 12 optimizers, which will optimize the code. And we also require 4 into 3 – 12 machine code generators. So, really speaking, this order is kind of interleaved because we produce machine code and we also optimize the machine code itself; we do not have any intermediate code here. So, these two actually are mixed with each other. Some of the optimizations are done on the basic blocks in the machine code; whereas, some of the optimizations are done on loops, etcetera. So, this is a fairly heavy investment. For each of these languages, we require an optimizer and also a code generator.

Let us see what happens when we have an intermediate language. So, definitely we require the 4 front ends which do go up to the semantic analysis; and they produce intermediate code instead of producing machine code. In such a case, we require 4 front ends. And then the intermediate code optimizer – just one of them is enough, because all the four source languages compile into the same intermediate language. And the intermediate code can be compiled into the machine codes. So, we require three different machine code generators as well. So, the extra in the first case is quite a bit; we require a large number of optimizers and machine code generators; whereas, here we are able to reuse the machine code generators. And of course, you may argue that, this front end and this front end are not the same, because in this front end we do not do any intermediate code generation; whereas, in this front end, we do some intermediate code generation. But, producing intermediate code is very simple as we are going to see and it definitely is not as difficult as writing too many optimizers and code generators.

(Refer Slide Time: 08:37)

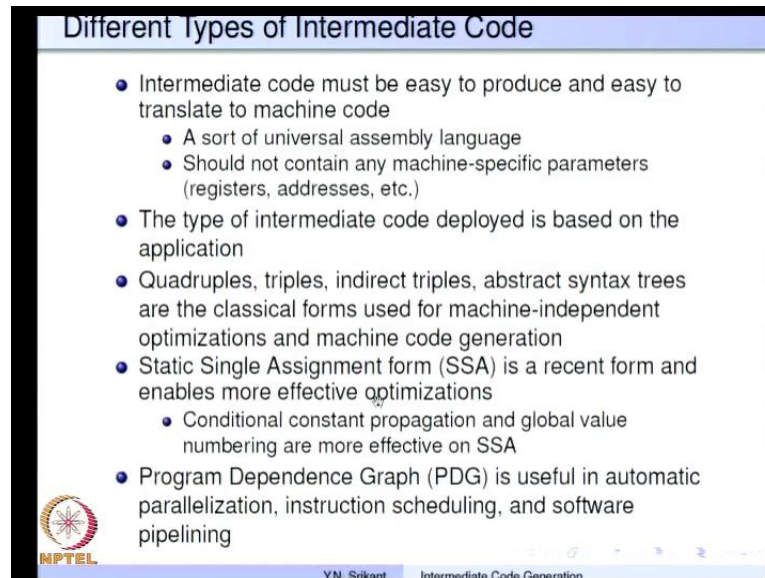
The slide is titled "Why Intermediate Code? - 2" and contains the following text:

- While generating machine code directly from source code is possible, it entails two problems
 - With m languages and n target machines, we need to write m front ends, $m \times n$ optimizers, and $m \times n$ code generators
 - The code optimizer which is one of the largest and very-difficult-to-write components of a compiler, cannot be reused
- By converting source code to an intermediate code, a machine-independent code optimizer may be written
- This means just m front ends, n code generators and 1 optimizer

The slide also features the NPTEL logo in the bottom left corner and the text "Y.N. Srikant Intermediate Code Generation" in the bottom right corner.

So, this is one of the problems. So, too much code to write, too much code to debug. Now, the problem is we are not able to reuse the code that we have written so far. So, the code optimizer is one of the largest and extremely difficult to write components of a compiler. And since in this case, we have a machine code optimizer and not an intermediate code optimizer, which is independent of the machine language, we really cannot reuse the optimizer written for this language in this particular code generation system or code optimizer system. Each one of them will have to be rewritten. Whereas, if you produce intermediate code, the machine independent code optimizer is just a single piece of code; it can be reused with all the compilers. So, this is a very efficient solution to the problem of producing many compilers for many source languages and machines.

(Refer Slide Time: 09:44)



Different Types of Intermediate Code

- Intermediate code must be easy to produce and easy to translate to machine code
 - A sort of universal assembly language
 - Should not contain any machine-specific parameters (registers, addresses, etc.)
- The type of intermediate code deployed is based on the application
- Quadruples, triples, indirect triples, abstract syntax trees are the classical forms used for machine-independent optimizations and machine code generation
- Static Single Assignment form (SSA) is a recent form and enables more effective optimizations
 - Conditional constant propagation and global value numbering are more effective on SSA
- Program Dependence Graph (PDG) is useful in automatic parallelization, instruction scheduling, and software pipelining

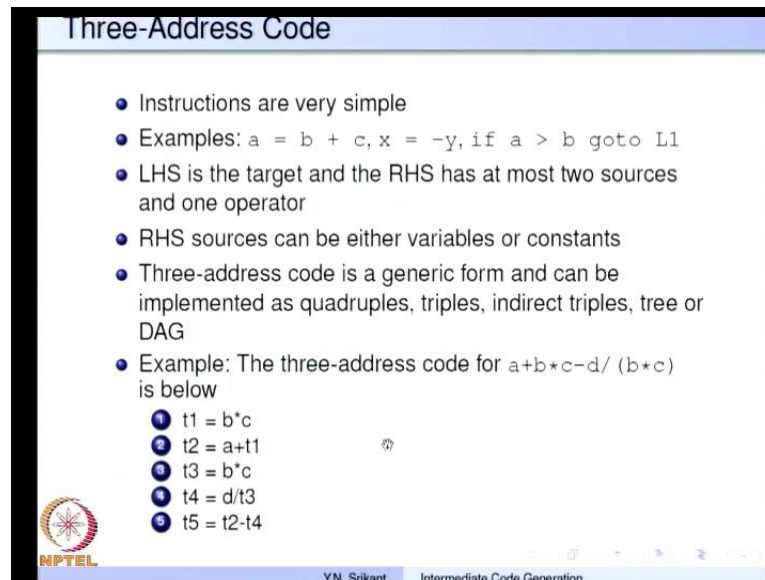
NPTEL YN. Srikant Intermediate Code Generation

What are the various types of intermediate code that we have available in literature? So, to do that, we must first of all understand what is the level at which the intermediate code is positioned. So, first of all, intermediate code must be very easy to produce and it must be easy to translate to machine code. This is something in between the source language in the machine code. So, you can call it as a sort of universal assembly language. And obviously, because this is supposed to be independent of any machine, it should not contain any machine-specific parameters such as registers, addresses, etcetera. The type of intermediate code deployed is based on the application. So, there are many of them. For example, we have quadruples, we have triples, we have indirect triples, and we have abstract syntax trees. These are the classical forms of intermediate code. And these are used for machine-independent optimization and machine code generation. So, this is the traditional use of these intermediate codes.

Recently – when I recently, it is still about 10-15 years ago that, the static single assignment form was invented. So, this a form, which is very effective for certain types of optimizations. So, for example, there is an optimization called conditional constant propagation and another optimization called the global value numbering. These are far more effective on the static single assignment form rather than on the traditional intermediate codes in the form of quadruples and triples. Finally, the program dependence graph or the PDG has been in use for many decades in the automatic parallelization of code. And they are also useful in instruction scheduling and software

pipelining phases of the machine-dependent optimizer. So, these are the various forms of intermediate code starting with the classical forms, then the SSA and the PDG. So, we are going to really study all forms of these intermediate codes in the coming lectures.


(Refer Slide Time: 12:18)



Three-Address Code

- Instructions are very simple
- Examples: `a = b + c`, `x = -y`, `if a > b goto L1`
- LHS is the target and the RHS has at most two sources and one operator
- RHS sources can be either variables or constants
- Three-address code is a generic form and can be implemented as quadruples, triples, indirect triples, tree or DAG
- Example: The three-address code for $a+b*c-d / (b*c)$ is below

- 1 `t1 = b*c`
- 2 `t2 = a+t1`
- 3 `t3 = b*c`
- 4 `t4 = d/t3`
- 5 `t5 = t2-t4`

 YN. Subant Intermediate Code Generation

So, let us look at a conceptual intermediate code called the three address code. So, let me emphasize that, the three-address code is really a generic form of intermediate code; and it can be implemented as quadruples, triples, indirect triples, trees or DAG. I will give you some examples very soon. In the three-address code, the instructions are extremely simple. There are three examples of instructions here; `a equal to b plus c`; `x equal to minus 5`; `if a greater than b, goto L1`. So, these are three examples of intermediate code. We will see many more as we go on. In the assignment statements of this kind, either `a equal to b plus c` or `x equal to minus y`, the LHS is the target and the RHS has at most two sources and one operator. So, this is the operator; and the `b` and `c` are the sources.

Why did we say at most two sources? In the case of such simple unary of instructions, we have just one source; so maximum of two and minimum of one. If you consider the branch statement; even here we can say this `L1` is the target and these are the sources and this is the operator. So, RHS sources can be either variables or constants. So, we can say `a equal to b plus 1`; we can say `a greater than 2`; but, we cannot definitely say `2 equal to b plus c`. So, the left-hand side must always be an address. So, let us take a simple expression `a plus b star c minus d slash b star c`. The interpretation would be subject to

the usual understanding of the operators. So, the multiplication takes precedence over plus; plus and minus are at the same level; and slash and star are also at the same level. So, the first one is the first intermediate instruction would be a equal to b star c, because we cannot do a plus b first; we will have to do b star c first.

Then, the second instruction is t 2 equal to a plus t 1. So, we have evaluated b star c; then we say a plus t 1. The third one is again t 3 equal to b star c; this particular thing, because we cannot do division before we evaluate this. And since division has more precedence than minus, we will have to do the division first. To do division first, we will have to do multiplication even earlier. Then we do t 4 equal to d slash t 3; and finally, t 5 equal to t 2 minus t 4. So, a few points have to be emphasized here; of course, the form of the intermediate code – it is that of the three-address code here.

So, we have one binary operator and two operands in each of these instructions. More important – the left-hand sides are all temporary variables, which are generated during the intermediate code generation phase. So, it is very important to remember that, the intermediate code employs a large number of temporaries; and these temporaries will be generated as and when we require them. There is usually no reuse of temporaries after their work is over; we just generate new temporaries and go on using them. The machine code generation and the optimization phase will take care of eliminating the redundant temporaries.

(Refer Slide Time: 16:17)

Implementations of 3-Address Code

3-address code

```

1 t1 = b*c
2 t2 = a+t1
3 t3 = b*c
4 t4 = d/t3
5 t5 = t2-t4
        
```

Quadruples

op	arg ₁	arg ₂	result
*	b	c	t1
+	a	t1	t2
*	b	c	t3
/	d	t3	t4
-	t2	t4	t5

Triples

op	arg ₁	arg ₂
*	b	c
+	a	(0)
*	b	c
/	d	(2)
-	(1)	(3)

Syntax tree

DAG

YN_Srikant Intermediate Code Generation

Here is an implementation of the 3-address code, rather many implementations of the 3-address code. So, we have 3-address code, then the quadruples, then triples, then syntax tree and DAG. So, traditionally, this has been used as the textual form; and the other four are used as data structures inside the machine or inside the compiler. So, the quadruple gets its name because there are four fields in each instruction: op, arg 1, arg 2 and result. So, it is possible to in fact show even jumps using the same format, because as I said, the result is the jump target; then arg 1, arg 2 are the arguments of the expression, and op is the relational operator. So, this is just a listing of the 3-address code here. So, there is nothing very special here. So, we can ... This is self explanatory.

Triples are slightly different. We really do not show the temporaries explicitly in the case of triples. So, let us go through them. The first instruction is star b c; and we have not shown any temporary. So, when we want to do t 2 equal to a plus t1, a is depicted here; and instead of t1, we provide the index of the instruction, which computes that particular operand. So, in this case, this is the instruction 0; star b c is the instruction, which is executing. So, next, we again do star b c; then we have slash d and 2. So, t4 equal to d slash t3. So, t3 is this particular instruction.

So, we provide the index of that instruction here as 2. Finally, for minus, we say t2 minus t4. So, t2 is number 1 – this particular instruction; and t4 is number 3, that is, this particular instruction. So, really speaking, this is nothing, but a straightforward encoding of the tree in this array form. So, if you look at the tree, this is easy. So, we have star b c here and then we have a plus b star c. Then we have b star c here and then d slash b star c, and then finally, a minus. So, this is nothing but an array encoding of this tree; that is it.

What is a directed acyclic graph representation of this 3-address code? It is very similar to that of the tree with the difference that, whenever there is some expression, which is already available, we do not recompute it, but we simply make the operand pointer point to it. So, in this case, b star c has already been computed. And therefore, the tree for b star c is right here. We just point the right operand of this slash to this particular subtree. And that is why this is a directed acyclic graph and not a tree representation. The important difference between DAG and all other forms of intermediate code that we have here is that, these catch what are known as common sub-expressions. So, there is no expression, which is recomputed unnecessarily. It is all reused again and again whenever


necessary and of course, if possible. Why did I say if possible? Suppose you assume that, either b or c has been assigned a value before $b * c$. In that case, this particular $b * c$ and the prior occurrence of $b * c$ are obviously very different; and in such cases, there is no question of reuse; we recompute $b * c$. So, this is how the 3-address code is actually implemented in practice. So, in our discussion, we will use 3-address code of the textual form in this form and we will say that, the machine implementation can use any one of these.

(Refer Slide Time: 20:49)

Instructions in Three-Address Code - 1

- 1 **Assignment instructions:**
 $a = b \text{ biop } c, a = \text{uop } b,$ and $a = b$ (copy), where
 - *biop* is any binary arithmetic, logical, or relational operator
 - *uop* is any unary arithmetic ($-$, shift, conversion) or logical operator (\sim)
 - Conversion operators are useful for converting integers to floating point numbers, etc.

- 2 **Jump instructions:**
 $\text{goto } L$ (unconditional jump to L),
 $\text{if } t \text{ goto } L$ (if t is *true* then jump to L),
 $\text{if } a \text{ relop } b \text{ goto } L$ (jump to L if $a \text{ relop } b$ is *true*),
 where
 - L is the label of the next three-address instruction to be executed
 - t is a boolean variable
 - a and b are either variables or constants



Y.N. Srikant Intermediate Code Generation

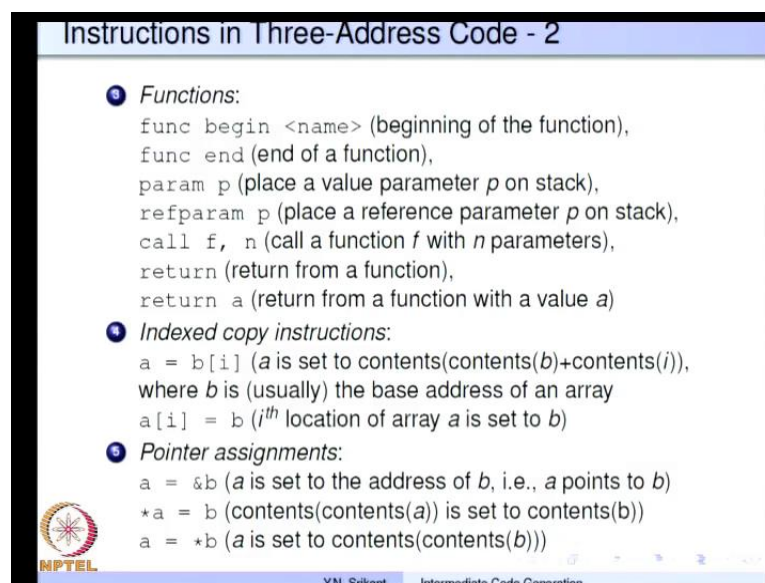
So, what are the various forms of 3-address code? So, I gave you a very few examples. Now, let us look at the exhaustive list. There are many types of assignment instructions. So, $a = b \text{ biop } c, a = \text{uop } b,$ and $a = b.$ *biop* is a binary operator; it can be arithmetic operator, logical operator or relational operator. *uop* is a unary operator; it is either an arithmetic operator or a shift operator or a conversion operator or logical operator; minus also is included, I missed it.

So, minus shift and conversion are all arithmetic type of operators; and logical operator is the compliment operator. So, what exactly is special about conversion? Minus and shift we understand already. Conversion is useful in converting integers into floating point numbers and floating point numbers into integers, characters into integers, and so on. So, we saw in semantic analysis that, we look at the coercibility of various types. So, if the coercibility is defined by the programming language, then we need to convert these

operands into suitable types before we emit the intermediate instruction corresponding to it. So, we are going to look at this also in the intermediate code generation phase.

Then, we have several types of jump instructions. So, there is an unconditional goto L. So, L is the label of the instruction to either target instruction. If t goto L; so if t is true, then jump to L. If a relop b goto L. So, if a relop b is true, then jump to L; otherwise, continue. So, here t is a boolean variable. So, either take 0 or 1. a and b are either variables or constants.

(Refer Slide Time: 23:16)



Instructions in Three-Address Code - 2

- **Functions:**
func begin <name> (beginning of the function),
func end (end of a function),
param p (place a value parameter p on stack),
refparam p (place a reference parameter p on stack),
call f, n (call a function f with n parameters),
return (return from a function),
return a (return from a function with a value a)
- **Indexed copy instructions:**
a = b[i] (a is set to contents(contents(b)+contents(i)),
where b is (usually) the base address of an array
a[i] = b (ith location of array a is set to b)
- **Pointer assignments:**
a = &b (a is set to the address of b, i.e., a points to b)
*a = b (contents(contents(a)) is set to contents(b))
a = *b (a is set to contents(contents(b)))

NPTEL
Y.N. Srikant Intermediate Code Generation

Then, we have many types of instructions to take care of function declaration and function call. For the function declaration, we require a function begin and name of the function. A function end instruction to end a function. Then to pass a parameter and place it on a stack, we require param p instruction; and this is a value parameter. There is a refparam p, which is required for a reference parameter. So, different types of parameter schemes will be learnt a little later. But, now, I should tell you that, value parameters actually evaluate the expression, which is passed as a parameter in the high level language and then place that value as the parameter; whereas, in the case of a reference parameter, the expression is evaluated and the address of that particular value is placed as a reference parameter. Then there is a call f comma n, which is an instruction to call a function f with n parameters. There is a return instruction without any value; and there is a return a instruction in which we return a value from the function.

Then, we have a indexed copy instructions. So, $a = b[i]$. So, $b[i]$ looks like it is an array; obviously, b is an array; i is the index into that array. The only difference is even though this appears as a single dimensional array, we are really going to convert multidimensional array accesses to such signal sequence single dimensional array accesses. So, that is why this is intermediate code; we are breaking down higher level statements into lower level statements. a is set to the contents of contents of b plus contents of i . So, usually, if it is a simple array, then b is the base address of the array and i is the offset into that array. So, you take the base address, add the contents of i ; then you get the place, where we actually want the value. So, access the value of that particular place and put it into a . This is the semantics of $a = b[i]$.

Similarly, $a[i] = b$ implies i -th location of array a is set to b . So, again as I said, this could be a translation of the multidimensional array into a single dimensional array. This may be the result of that. Then we have... So, you must also observe that, we do not have any instruction of the form $a[i] = b[i]$. This is because $a[i]$ is already... – it has an indexing operator. So, here for example, if you say $a = b[i]$ just like $a = b * c$ and $*$ being an operator, here we have b and i as source operands; the indexing is the operator; and this is the target of the assignment. Similarly, here as well, i is an operand; b is another operand, because they are not modified, and is an assignment. And then of course, indexed assignment is the operator. So, this is usually indicated as $a[b[i]] = c$ and this is indicated as $a[b[i]] = c$.

Pointer assignment – we have $a = b$, which sets a to the address of b ; that is, a points to b . $*a = b$; so we take b ; then evaluate the address as $*a$. So, take the contents of a ; treat it as an address; go to that address; and that is where we are going to put b . So, contents of contents of a is set to contents of b . The effective address is obtained by looking at the contents of a and go to that particular place. So, it is not a single level addressing here; there is indirect addressing mechanism as well. $a = *b$ is similar. So, a is set to the contents of contents of b . So, contents of b would be an address. So, we have to take the contents again. So, here also, the contents of a would be the address; where, b is placed.

(Refer Slide Time: 28:05)


Intermediate Code - Example 1

C-Program

```
int a[10], b[10], dot_prod, i;
dot_prod = 0;
for (i=0; i<10; i++) dot_prod += a[i]*b[i];
```

Intermediate code

dot_prod = 0;		T6 = T4[T5]
i = 0;		T7 = T3*T6
L1: if(i >= 10)goto L2		T8 = dot_prod+T7
T1 = addr(a)		dot_prod = T8
T2 = i*4		T9 = i+1
T3 = T1[T2]		i = T9
T4 = addr(b)		goto L1
T5 = i*4		L2:



Y.N. Srikant Intermediate Code Generation

Now, we are going to look at a series of programs and the intermediate code that is produced by a typical compiler for such programs. The C program has int a 10, b 10, dot product and i. They are all integers; a and b are arrays of size 10. dot prod is assigned 0 to begin with; initialized to 0. There is a loop, which starts from 0 goes up to, but not inclusive of 10; and it is incremented once with an increment of 1 every time. dot product equal to dot product plus a i star b i; that is the meaning of this. So, we compute the dot product and the translation is quite straightforward. The declaration does not have any translation; obviously, there is no code produced for declarations. We start with dot prod equal to 0; this is already in a very simple form. So, there is nothing more to do. Then we have i equal to 0; this is a translation of the loop. So, we check whether i greater than or equal to 10; if so goto L 2; that is the exit of the loop; otherwise, the body of the loop.

So, now, take the address of the a. In fact, the address of a could be the stack pointer value pointing to the place, where a is placed. Then we have second instruction T2 equal to i star 4. So, we are now translating a of i. Then T3 equal to T1 of T2. So, essentially, we are doing a of i with these three instructions. So, you can easily see that, a single instruction a of i, rather single access a of i translates to three instructions in the intermediate code. Then we translate b of i; which is T4 equal to address b; T5 equal to i star 4; and T6 equal to T4 of T5. So, this is effectively b of i. Now, we do the multiplication. So, T7 equal to T3 star T6. Then we add that to dot product. So, T8 equal

to dot product plus T7. Then we must assign it back to dot product. So, dot product equal to T8.

Now, we do the second part; the increment here for the loop. T9 equal to i plus 1; and i equal to T9. So, you should also observe that, the intermediate code generation produces really dumb intermediate code. It is easy to see that, this is nothing but i equal to i plus 1, but, we do not do that, and even this. It is nothing but dot prod equal to dot prod plus T7; but, we do not do that. The intermediate code generation that is why is a simple-minded program. And an optimizer is anyways necessary to improve the program. Finally, there is goto L1, which repeats the loop. So, this is the intermediate code produced; it is just like assembly code for this particular program. So, let us look at a second example; the same dot product program. But, let us say we use a pointer to run through the arrays instead of using indexing as we have done here; a i plus star b i. So, a i star b i.

(Refer Slide Time: 31:54)


Intermediate Code - Example 2

C-Program

```
int a[10], b[10], dot_prod, i; int* a1; int* b1;
dot_prod = 0; a1 = a; b1 = b;
for (i=0; i<10; i++) dot_prod += *a1++ * *b1++;
```

Intermediate code

<pre>dot_prod = 0; a1 = &a b1 = &b i = 0 L1: if(i>=10)goto L2 T3 = *a1 T4 = a1+1 a1 = T4 T5 = *b1 T6 = b1+1</pre>	<pre>b1 = T6 T7 = T3*T5 T8 = dot_prod+T7 dot_prod = T8 T9 = i+1 i = T9 goto L1 L2:</pre>
------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------


YN. Subant Intermediate Code Generation

Instead of that, let us run through the arrays using pointers. So, we have a 10; we have b 10, dot prod and i as integers. Then we have pointers to integers – int star a1 and int star b1. So, we start with dot product equal to 0; a1 equal to a. So, the pointer a1 is pointing to a; pointer b1 is pointing to b. The loop in the body is different, but the loop header is the same. So, we write dot prod plus equal to star a1 plus plus star star b1 plus plus. So, what is the meaning of this assignment? We do star a1 first. So, that gets you the contents of the array in a way similar to a of i. Then we must go to the next location in

the array. In this case, we actually... In this case, we did a of i; and then this i plus plus took care of progressing to the next element in the array. Since we are not using the i to index into the array, we must alter the pointer itself. So, after star a1, we do a1 plus plus. So, that automatically takes you to the next element in the array. Similarly, star b1 gets you the contents of that location and b1 plus plus will take you to the next location. Multiplication of these two will produce the product and then we add it to dot prod. So, that is really the same dot product that we had seen earlier. Here the loop variable I is not used in the computation, but it is used only for the termination of the loop.

So, let us see what the code corresponding to this b. So, this is easy – dot prod equal to 0. Then the pointer assignment a1 equal to ampersand a. So, that is the address of a. Similarly, b1 equal to ampersand b – address of b. Then the initialization of i; i equal to 0. Now, the loop. So, this part is the same. If i greater than equal to 10, goto L2. The body of the loop – first, we do T3 equal to star a1. We have an intermediate code instruction for that. Then we do T4 equal to a1 plus 1. That is the a1 plus plus part. Then we do a1 equal to T4. So, these two together do the auto increment on a1. Then we have star T5 equal to star b1; T6 equal to b1 plus 1, and b1 equal to T6. So, that is the star b1 plus plus. We do the multiplication T3 star T5. Then add it to the dot product in these two as before; then the loop control here. So, this shows an example with the pointer to the array instead of indexing.

(Refer Slide Time: 34:56)


Intermediate Code - Example 3

```

C-Program (function)
int dot_prod(int x[], int y[]){
    int d, i; d = 0;
    for (i=0; i<10; i++) d += x[i]*y[i];
    return d;
}

Intermediate code
func begin dot_prod |      T6 = T4[T5]
d = 0;               |      T7 = T3*T6
i = 0;               |      T8 = d+T7
L1: if(i >= 10)goto L2 |      d = T8
T1 = addr(x)         |      T9 = i+1
T2 = i*4              |      i = T9
T3 = T1[T2]           |      goto L1
T4 = addr(y)          |L2: return d
T5 = i*4              |      func end

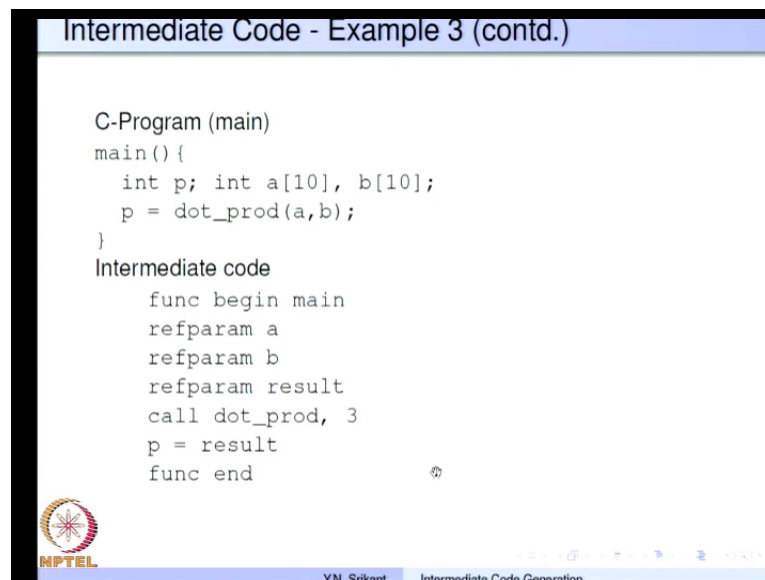
```


Y.N. Subramanian Intermediate Code Generation

The third program shows you a function for the dot product. So, these are all different variants of the same computation. So, the function is `int dot_prod`; it takes two arrays as parameters: `int x` and `int y`. We have `d, i` as integer variables inside the function. `d` is initialized to 0. Then the loop runs exactly the way it used to. And we have `d plus equal to x i star y i` exactly the way it was in the main program before. So, `return d` returns the value of the dot product. So, `func begin dot_prod`; obviously, beginning of the function requires this intermediate instruction. Then `d equal to 0` and `i equal to 0` as before. So, the loop control is also as before. So, nothing to expand. But, here after the loop terminates, we need to return the value of the dot product and then go back to the program. So, `return d` combines the tasks of value return and return to the main pro...call e. `Func end` of course, ends the function.

In the body of the program, the code is not very different. So, I am not going to expand it all over again, explain it all over again. So, we have address `x i star 4, T1 T2`, address `y, i star 4, T4 T5`, `T3 star T6`, `d plus T7`, `d equal to T8`, etcetera. Now, we should also see how the function is called. So, what is special about this? This shows you how the function is written and how the values are returned by the function.

(Refer Slide Time: 36:58)



```
Intermediate Code - Example 3 (contd.)

C-Program (main)
main(){
  int p; int a[10], b[10];
  p = dot_prod(a,b);
}
Intermediate code
func begin main
refparam a
refparam b
refparam result
call dot_prod, 3
p = result
func end
```

The slide displays the C-Program (main) and its corresponding intermediate code. The C-Program (main) is shown as a function `main()` that declares `int p`, `int a[10]`, and `int b[10]`, and then calls `dot_prod(a,b)` to assign the result to `p`. The intermediate code shows the function `func begin main`, followed by `refparam a`, `refparam b`, and `refparam result`. It then shows a `call dot_prod, 3` instruction, followed by `p = result` and `func end`. The slide also features the NPTEL logo and the text 'Y.N. Srikant Intermediate Code Generation' at the bottom.

In the main program, we have `int p`, `int a 10`, `b 10`; and then `p equal to dot prod a comma b`. I have skipped the part, where we read values into `a` and `b`. So, `func begin main`. So, `main` is also a function in C. And then the first parameter – array is always passed by

reference. So, we have refparam a. And the second parameter b is again an array. So, it is refparam b. The base address of the arrays are passed in these places. Then we also need a place for the result. So, refparam result. So, you must keep in mind that, this location result is actually in this main program; it is not a part of the function. Therefore, the code generator must be able to produce the appropriate code for this return instruction. So, we will see that later anyway. Then there is a call to dot product and the number of parameters is 3 including the result. So, a, b and the result. Then they come out equal to result; the result would have been assigned a value by the function; and then we have func end. So, these are a couple of examples to show the various constructs in the intermediate code.

(Refer Slide Time: 38:23)


Intermediate Code - Example 4

C-Program (function)

```
int fact(int n){
    if (n==0) return 1;
    else return (n*fact(n-1));
}
```

Intermediate code

func begin fact		T3 = n*result
if (n==0) goto L1		return T3
T1 = n-1		L1: return 1
param T1		func end
refparam result		
call fact, 2		


YN. Subant Intermediate Code Generation

And, one final example will show you how recursion is handled in the intermediate code. So, we have the famous factorial function here – int fact n; if n is 0, return 1; otherwise, return n star fact n minus 1. So, it is quite straightforward; nothing very special here. func begin fact; if n equal to 0 goto L1. So, in L1, we have return 1. So, that is this part. Then we compute n minus 1. Push that parameter using the param T1. Then the result refparam result. Then call fact with two parameters: first is T1, the second is a result. Then T3 accumulates the value n star result and we return T3. So, as I said, since return combines two functions: one is sending a value back to the caller and second is to return to the caller. So, there is no question of the control flow going to return 1 after the return T3 instruction. So, nothing to worry here.

(Refer Slide Time: 39:37)

Code Templates for If-Then-Else Statement

Assumption: No short-circuit evaluation for E (i.e., no jumps within the intermediate code for E)

If (E) S1 else S2

```
code for E (result in T)
if T ≤ 0 goto L1 /* if T is false, jump to else part */
code for S1 /* all exits from within S1 also jump to L2 */
goto L2 /* jump to exit */
L1: code for S2 /* all exits from within S2 also jump to L2 */
L2: /* exit */
```

If (E) S

```
code for E (result in T)
if T ≤ 0 goto L1 /* if T is false, jump to exit */
code for S /* all exits from within S also jump to L1 */
L1: /* exit */
```

NPTEL
YN. Srikant Intermediate Code Generation

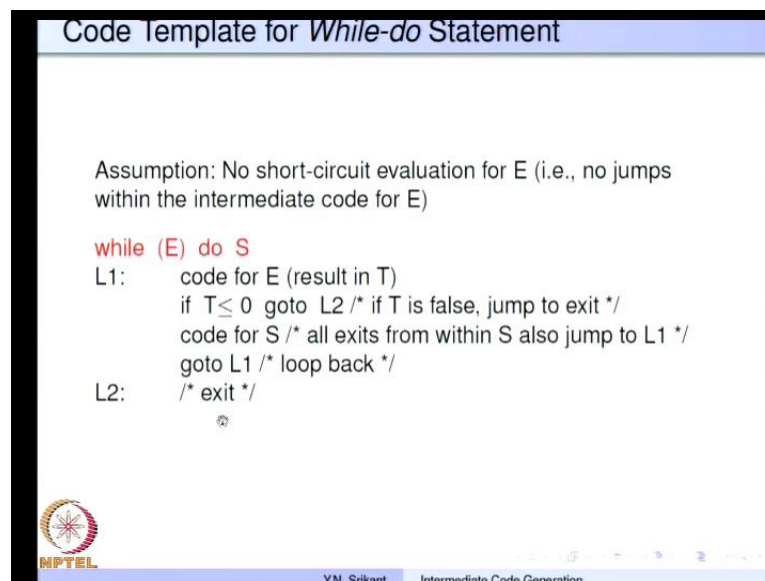
So, that is about the examples of various types of intermediate code, how they are produced, rather what intermediate code is produced for programs, and so on. So, now, let us delve into the details of producing such intermediate code for various constructs in the language. So, let us look at code templates for if then else statement. So, the form of the if then else statement we already know very well. If E S1 else S2. The other way is if ES. So, the assumption is we do not have what is known as a short circuit evaluation for E. So, we will see a little later that, if... Since E is a boolean expression, we can actually have jumps out of the expression E if we produce what is known as a control flow code for the boolean expression. So, this is known as short circuit evaluation for E.

So, let us assume that, there is no short circuit evaluation. In other words, the expression E is evaluated completely with no jumps and then the decision of whether it is true or false is made. So, obviously, the code that must be produced for this is quite intuitive. So, first of all, we must produce the code for E. Then let us assume that, the result of this is in the temporary T. Then here we must check whether T is true or false. So, if T is false, the else part has to be executed. So, goto L1. So, that is why the jump. If that T variable contains a true value, then we execute S1.

So, code for S1. Now, after S1, we actually have come to this point; we should not fall through and execute S2; we should actually jump to outside of S2. So, that is why go to L2, which is the exit. But, there are also cases where there are jumps from within in S1.

We will see examples of this very soon to understand it. Similarly, there will be jumps from within S2 as well. So, all exits from within S1 and S2 also jump to L2. So, this is something we must be careful about. And I will show you examples of how this can happen. If E S is only a subset of what we have discussed so far; code for E; then the branch statement to check whether T is true or false; and then if it is false, then we go out; otherwise, we execute code S and then go out. So, all exits from S also jump to L1.

(Refer Slide Time: 42:39)



The slide is titled "Code template for *While-do* Statement". It contains the following text:

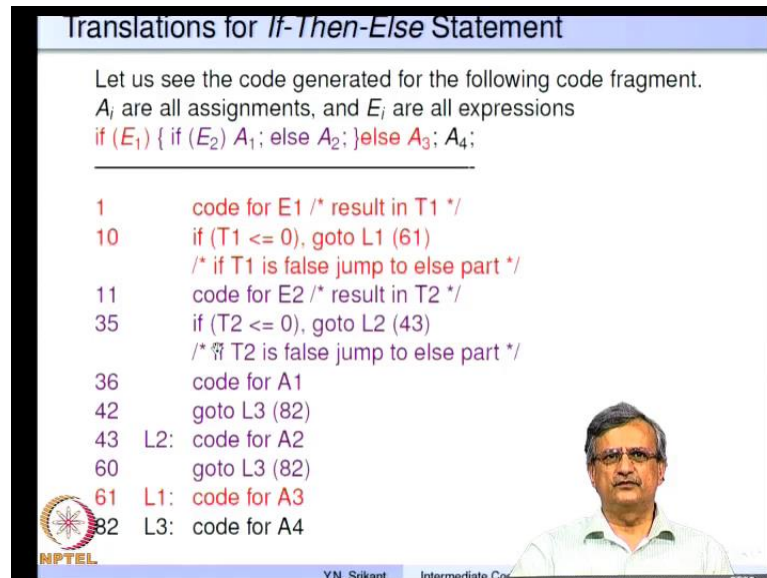
Assumption: No short-circuit evaluation for E (i.e., no jumps within the intermediate code for E)

```
while (E) do S
L1:   code for E (result in T)
      if  $T \leq 0$  goto L2 /* if T is false, jump to exit */
      code for S /* all exits from within S also jump to L1 */
      goto L1 /* loop back */
L2:   /* exit */
```

The slide also features the NPTEL logo in the bottom left corner and the text "YN. Srikant Intermediate Code Generation" in the bottom right corner.

What about the while construct? Again we have no short circuit evaluation for E; that is the assumption. We will consider short circuit evaluation a little later. So, we produce the code for E. The result is in T. Then as usual, we must check whether E is true or false. So, if T is less than or equal to 0, that is false; we jump out; that is L2. If E is true, then we continue and execute S. So, the code for S must be produced. After code for S, we must go back to the code for E, evaluate it and continue with the loop. So, there is a goto for L1. The other special case here is if there are any jumps out of S, all these must actually jump to L1. So, that must be taken care of.

(Refer Slide Time: 43:36)



Translations for If-Then-Else Statement

Let us see the code generated for the following code fragment.
 A_i are all assignments, and E_i are all expressions
`if (E_1) { if (E_2) A_1 ; else A_2 ; }else A_3 ; A_4 ;`

```
1      code for E1 /* result in T1 */
10     if (T1 <= 0), goto L1 (61)
      /* if T1 is false jump to else part */
11     code for E2 /* result in T2 */
35     if (T2 <= 0), goto L2 (43)
      /* if T2 is false jump to else part */
36     code for A1
42     goto L3 (82)
43 L2: code for A2
60     goto L3 (82)
61 L1: code for A3
82 L3: code for A4
```

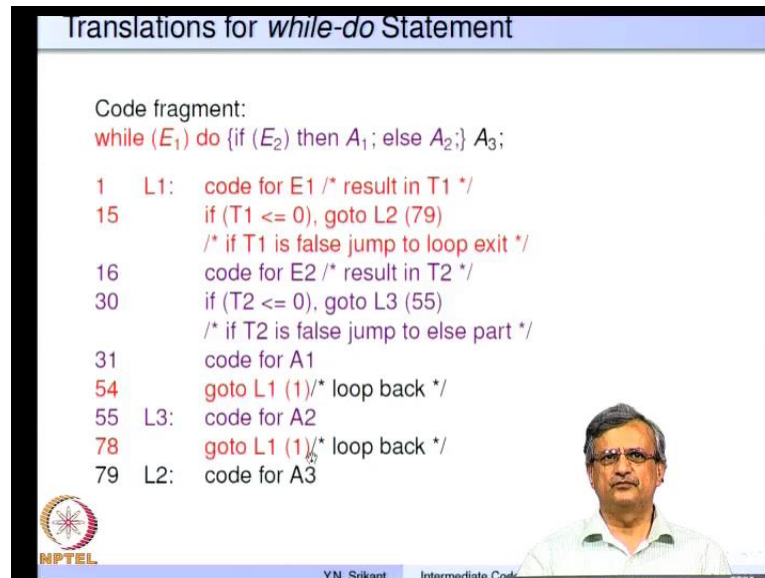
NPTEL
Y.N. Srikant Intermediate Co

So, let us look at an elaborate example to show how the jumps from statements within can also arise. So, let A in this example be assignments and E_i be expressions. So, the code is `if E_1` . And in the then part, we have a complete if then else again. And in the else part, we have `else A_3` . And after this entire statement of the outer part, we have the next statement A_4 . So, the intuitive understanding is we check whether E_1 is true. If E_1 is true, we execute the second if then else; if it is false, we execute A_3 and then go on to A_4 . So, if it is true, we come inside; we again check whether E_2 is true. If it is true, then we must execute A_1 and then jump to A_4 directly. If it is false, we must execute A_2 and then jump directly to A_4 . We should never execute A_3 after any one of these. So, this is how jumps from within a statement can arise. So, there is a jump from here and also a jump from here, which should actually take you to A_4 and not to A_3 . So, let us see the code for this.

There is code for E_1 . Then the temporary for E_1 is tested. T_1 less than equal to 0 goto L1. So, that would be the code for A_3 – the else part. So, in red, we show the code for the outer part; and in violet, we show code for the inner part. Then the code for E_2 ; if that is false, the E_2 expression is false; goto L2. So, that is else part of this second expression – second if then else. We execute that and then jump to L3, that is, the code for A_4 . So, observe that, this is the jump out of the inner statement in this if then else. Then we have code for A_1 and we jump to again L3. So, as I was saying, it is this jump.

Then we have code for A 2 jump to this thing; and finally, code for A 3 and fall through to A 4. So, that is how the jumps out of inner statements can arise.

(Refer Slide Time: 46:13)



The slide is titled "Translations for *while-do* Statement". It contains the following text:

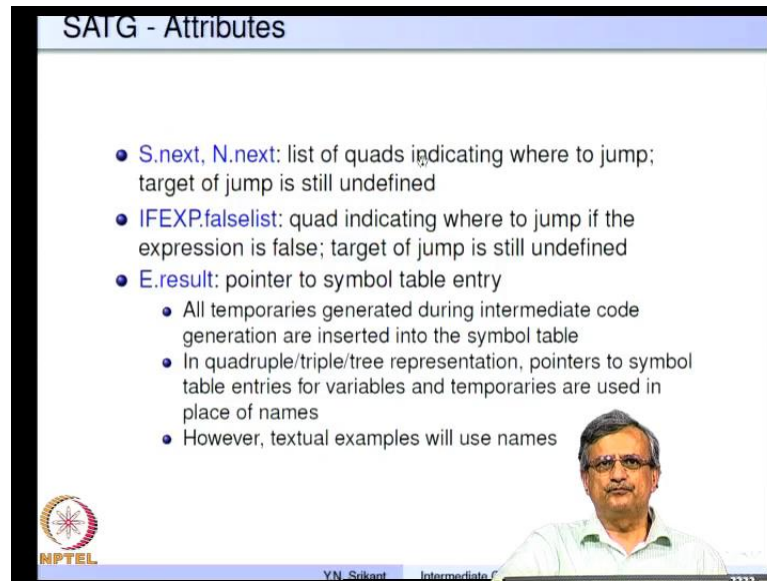
```
Code fragment:
while (E1) do {if (E2) then A1; else A2;} A3;

1  L1:  code for E1 /* result in T1 */
15     if (T1 <= 0), goto L2 (79)
      /* if T1 is false jump to loop exit */
16     code for E2 /* result in T2 */
30     if (T2 <= 0), goto L3 (55)
      /* if T2 is false jump to else part */
31     code for A1
54     goto L1 (1)/* loop back */
55  L3:  code for A2
78     goto L1 (1)/* loop back */
79  L2:  code for A3
```

The slide also features the NPTEL logo in the bottom left corner, a portrait of a man in the bottom right corner, and a footer with the text "Y.N. Srikant Intermediate Comp" and "5555".

Let us look at an example of the while statement as well. So, here is the while part; and inside, we have an if then else; and finally, we have... The body of this while loop is an if then else. And finally, we have another assignment statement. So, while the expression E 1 is true, we go on executing this; and then finally, we jump to A 3 when the expression becomes false. So, again after A 1, there is a jump out of the if then else; and after A 2, again there is another jump out of the if then else. Both of them will take us to the beginning of E 1. So, code for E 1; and then if T 1 less than equal to 0, goto L2; that is the exit; code for A 3 directly; otherwise, we execute the code for E 2, then test it. If it false, we go to the else part, that is, the L3. And then we go back to the beginning of the code. So, otherwise, we execute the code for L1 and go back to the beginning of while loop. So, this is how jumps can arise from within while loops as well.

(Refer Slide Time: 47:24)



The slide is titled "SATG - Attributes" and contains a bulleted list of attributes. The first three items are: "S.next, N.next: list of quads indicating where to jump; target of jump is still undefined", "IFEXP.falselist: quad indicating where to jump if the expression is false; target of jump is still undefined", and "E.result: pointer to symbol table entry". The "E.result" item has three sub-bullets: "All temporaries generated during intermediate code generation are inserted into the symbol table", "In quadruple/triple/tree representation, pointers to symbol table entries for variables and temporaries are used in place of names", and "However, textual examples will use names". In the bottom right corner of the slide, there is a portrait of a man with glasses and a light green shirt. The NPTEL logo is in the bottom left corner, and the text "YN. Srikant Intermediate" is at the bottom center.

- S.next, N.next: list of quads indicating where to jump; target of jump is still undefined
- IFEXP.falselist: quad indicating where to jump if the expression is false; target of jump is still undefined
- E.result: pointer to symbol table entry
 - All temporaries generated during intermediate code generation are inserted into the symbol table
 - In quadruple/triple/tree representation, pointers to symbol table entries for variables and temporaries are used in place of names
 - However, textual examples will use names

So, it is... Now, we move on and start looking at the – now, SATG – attribute translation grammar with the synthesized attributes to produce intermediate code for various types of constructs. So, there are many attributes. So, let us look at some of them; the others will be clear as we along. Most important we have what is known as S dot next and N dot next; S and N are two non-terminals that we are going to use in the grammar. These are lists of quadruples indicating where to jump. So, the target of the jump would still be undefined when it is on this list. Then there is if expression dot falselist. So, these are all synthesized attributes. So, we are not going to put any arrows corresponding to it. If expression dot falselist indicates that it is a quadruple. So, we want to jump; where to jump if the expression is false. So, when we generate the jump statement for the expression – if expression is false, then jump; the target is still undefined, and the target will be defined a little later.

E dot result in general is a pointer into the symbol table entry, which is... It is a temporary storing the result of evaluating the expression. So, as I already told you, all temporaries generated during intermediate code generation stage are inserted into the symbol table. And in quadruple, triple and tree representations, that is, the implementation, pointers to symbol table entries for variables and temporaries are used in place of names; whereas, in our textual examples of 3-address code, we will continue to use the names. But, wherever we have used a name, its meaning is that, there is a

pointer to the entry for that particular entry in the symbol table. So, the SATG will make these attributes very clear.

(Refer Slide Time: 49:38)

The slide is titled "SATG - Auxiliary functions/variables" and contains a bulleted list of functions. The functions are: **nextquad**: global variable containing the number of the next quadruple to be generated; **backpatch(list, quad_number)**: patches target of all 'goto' quads on the 'list' to 'quad_number'; **merge(list-1, list-2, ..., list-n)**: merges all the lists supplied as parameters; **gen('quadruple')**: generates 'quadruple' at position 'nextquad' and increments 'nextquad'. A sub-bullet for **gen** states: "In quadruple/triple/tree representation, pointers to symbol table entries for variables and temporaries are used in place of names. However, textual examples will use names"; **newtemp(temp-type)**: generates a temporary name of type *temp-type*, inserts it into the symbol table, and returns the pointer to that entry in the symbol table. The slide also features the NPTEL logo in the bottom left corner and the text "YN. Srikant Intermediate Code Generation" in the bottom right corner.

Then, we have a global variable called `nextquad`, which actually contains a number; the number indicating next quadruple to be generated. So, whenever we increment the `nextquad`, that means, we have generated one instruction and we want to go to the next hole, where an instruction can be placed. Then there is a `backpatch` instruction, which takes two parameters: first one is the list; the other one is the quadruple number. The list contains a large number of, a number of rather branch instructions, whose targets are unfilled as of now. Those targets will be filled with quad number by this `backpatch` instruction. So, this is a compiler instruction and not an intermediate code instruction. Merge is a compiler function; it takes many lists and merges them into one.

The core of the SATG would always be this `gen` quadruple, which outputs a quadruple. So, where is the quadruple going to be placed? It is going to be placed at position `nextquad`; and `nextquad` counter will be incremented as well. So, again just to stress this point, the temporaries of variables, which are inserted into the symbol table; and we use names in the textual representation. But, in actual implementation, the pointers to symbol table will be used. We have used a large number of temporaries. So, we require a temporary generator. `newtemp` is a temporary generator and its parameter is the type of the temporary that is needed whether it is `int` or `real` or `char` or etcetera. This generates a

temporary name of the temp-type; inserts the name, which is generated into the symbol table and returns the pointer to that entry in the symbol table. So, that is how T 3, T 4, etcetera are dealt with.

(Refer Slide Time: 51:57)

SAIG for *If-Then-Else* Statement

- $IFEXP \rightarrow \text{if } E$


```
{ IFEXP.falselist := makelist(nextquad);
  gen('if E.result ≤ 0 goto ___'); }
```
- $S \rightarrow IFEXP S_1; N \text{ else } M S_2$

```
{ backpatch(IFEXP.falselist, M.quad);
  S.next := merge(S1.next, S2.next, N.next); }
```
- $S \rightarrow IFEXP S_1;$

```
{ S.next := merge(S1.next, IFEXP.falselist); }
```
- $N \rightarrow \epsilon$

```
{ N.next := makelist(nextquad);
  gen('goto ___'); }
```
- $M \rightarrow \epsilon$

```
{ M.quad := nextquad; }
```



YN. Subant Intermediate Code Generation

So, as we had done in the case of semantic analysis, we need to break the production for if then else into two parts. So, this is the production S going to if expression, then S_1 else S_2 . So, we have broken the if E part into another production; $IFEXP$ going to if E . In the case of semantic analysis, the purpose was to provide an appropriate error message very early; whereas, in the case of the intermediate code generation, this becomes necessary, because we must first produce the test for E immediately after parsing E before we produce the code for S_1 and S_2 . If the code for S_1 and S_2 is produced early without the jump, then it is incorrect code.

So, what is happening here? So, we have parsed E ; we have come up to this stage; the reduction has not happened in the parser. So, here if expression dot falselist – that is the attribute of the left-hand side. We will now store the next instruction in that list. So, makelist nextquad. So, next quad is the quadruple number of this particular instruction at this time. The instruction generated is if E dot result less than or equal to 0 goto dash. So, this blank or dash is unfilled at this time; and if possible, we will fill it; otherwise, we are going to fill it in some other production. So far we have tested the expression and produced code here.

In the second one, S going to if expression S 1; then we have a marker non-terminal N else and another marker non-terminal M; and finally, S 2. So, here if we need to patch this instruction, which implies that the E dot result is false and address to which it is patched is the beginning of the code for S 2. That is stored in M dot quad. So, M going to epsilon will store the quadruple number as M dot quad. That is the address of the code corresponding to S 2. So, backpatch if expression dot falselist comma M dot quad.

And then on the S dot next, we still do not know where jumps out of S 1 will be going to. So, all the S 1 dot next list will be placed inside this merge statement. We do not know where exactly this jump for N will go to. So, N going to epsilon will produce a goto statement after the then part. And we do not know where the jumps out of S 2 will go to. So, all these will be merged into S dot next. Here this is the if then expression. So, IFEXP S 1. We just do a merger of S 1 dot next and IFEXP dot falselist and put it on S dot next. So, we will stop here and continue with rest of the translation in the next lecture.

Thank you.