Compiler Design

Prof. Y. N. Srikant

Department of Computer Science and Automation

Indian Institute of Science, Bangalore

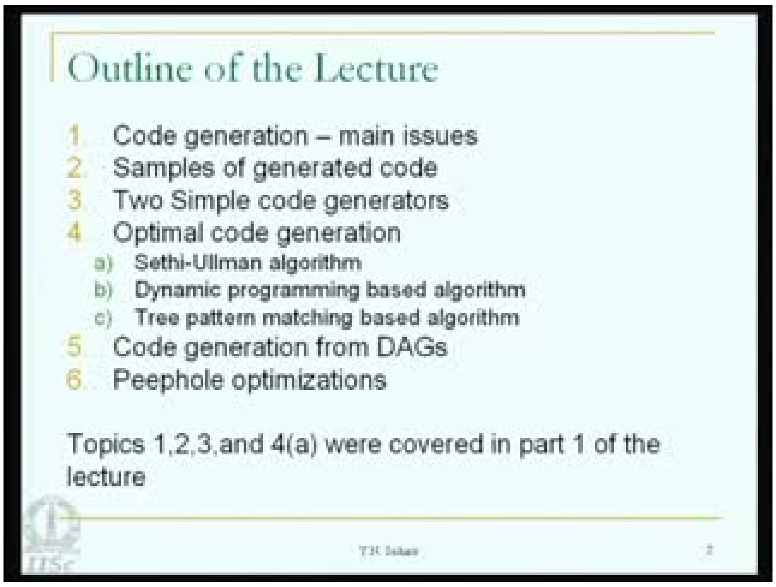
Module No. # 04

Lecture No. # 10

Code Generation Part – 2

Welcome to part II of the lecture on Code Generation.

(Refer Slide Time: 00:21)



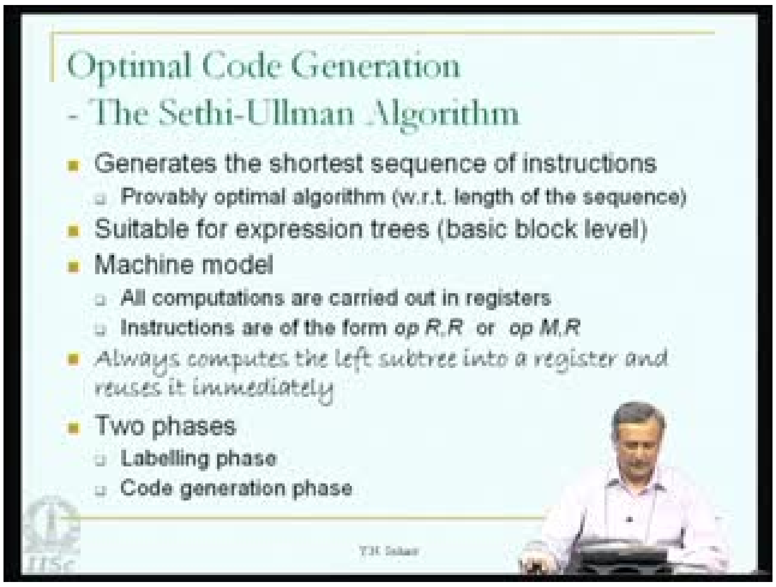
Outline of the Lecture

1. Code generation – main issues
2. Samples of generated code
3. Two Simple code generators
4. Optimal code generation
   a) Sethi-Ullman algorithm
   b) Dynamic programming based algorithm
   c) Tree pattern matching based algorithm
5. Code generation from DAGs
6. Peephole optimizations

Topics 1,2,3,and 4(a) were covered in part 1 of the lecture

In the last lecture, we learnt about the main issues of code generation and simple code generation algorithms. We also learnt about the Sethi-Ullman code generation algorithm, which produces optimal code. Today, let us review Sethi-Ullman algorithm, look at examples and then continue with the other optimal code generation algorithms.

(Refer Slide Time: 00:54)



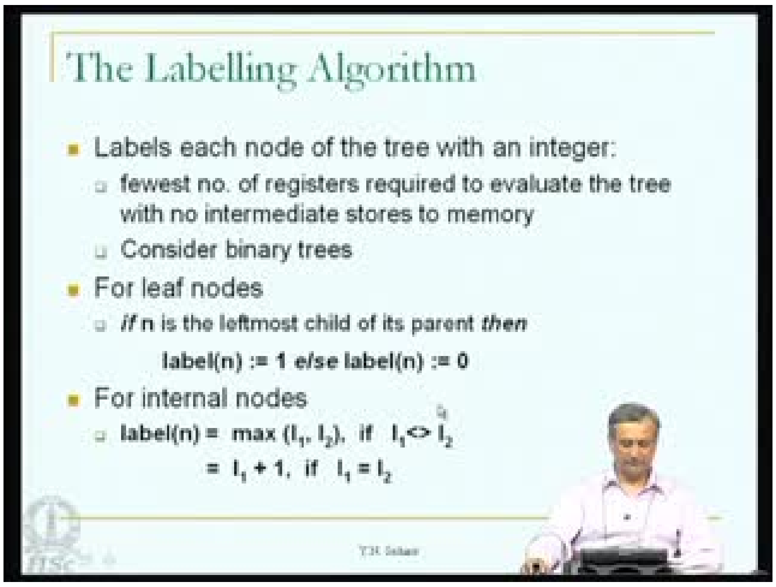
Optimal Code Generation - The Sethi-Ullman Algorithm

- Generates the shortest sequence of instructions
  - Provably optimal algorithm (w.r.t. length of the sequence)
- Suitable for expression trees (basic block level)
- Machine model
  - All computations are carried out in registers
  - Instructions are of the form *op R,R* or *op M,R*
- Always computes the left subtree into a register and reuses it immediately
- Two phases
  - Labelling phase
  - Code generation phase

Y.N. Srikant

Sethi-Ullman algorithm produces optimal code. It has two phases: the labelling phase and the code generation phase.

(Refer Slide Time: 01:03)



The Labelling Algorithm

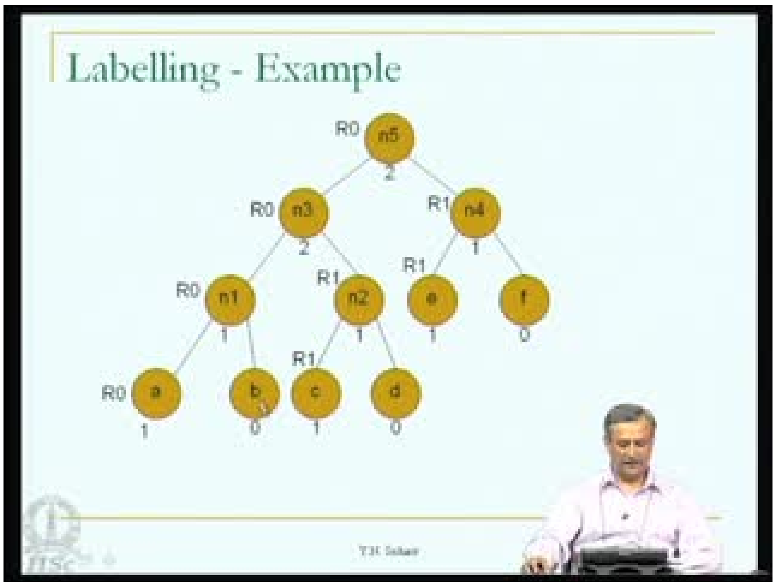
- Labels each node of the tree with an integer:
  - fewest no. of registers required to evaluate the tree with no intermediate stores to memory
  - Consider binary trees
- For leaf nodes
  - *if* n is the leftmost child of its parent *then*

    label(n) := 1 *else* label(n) := 0
- For internal nodes
  - label(n) = max (l_1, l_2), if $l_1 <> l_2$

    = $l_1 + 1$, if $l_1 = l_2$

Y.N. Srikant

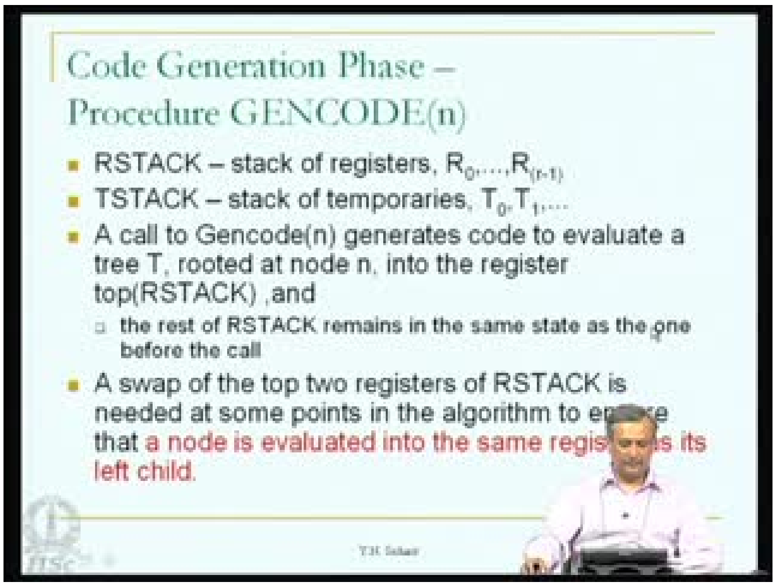
(Refer Slide Time: 01:05)



Labelling - Example

The labelling algorithm gives you the minimum number of registers required to evaluate the tree without any intermediate stores. For example, the leftmost child, which is a leaf is labeled 1 and all other children, which are leaves, are labeled 0. Here, this is 1 and 0. For internal nodes, if the 2 children have equal labels, then the node has plus 1 along with value of the label of the child plus 1; otherwise, if they are unequal, then the maximum of the 2 children values of the 2 children is taken. For example, here it is 1 and 0. So, this is max and this is 1. Here, it is 1 and 1. So, it is actually 1 plus 1, which is 2.

(Refer Slide Time: 01:52)



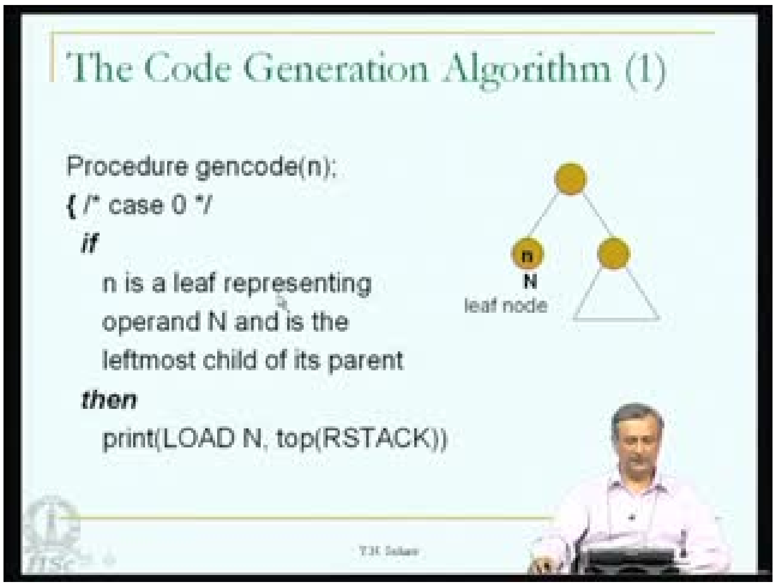
Code Generation Phase – Procedure GENCODE(n)

- RSTACK – stack of registers, $R_0, \ldots, R_{(r-1)}$
- TSTACK – stack of temporaries, $T_0, T_1, \ldots$
- A call to Gencode(n) generates code to evaluate a tree T, rooted at node n, into the register top(RSTACK) ,and
  - the rest of RSTACK remains in the same state as the one before the call
- A swap of the top two registers of RSTACK is needed at some points in the algorithm to ensure that a node is evaluated into the same register as its left child.

Y.N. Srikant

As I said in the last lecture, the algorithm uses two stacks: one is the register stack and the other is the temporary stack.

(Refer Slide Time: 02:03)



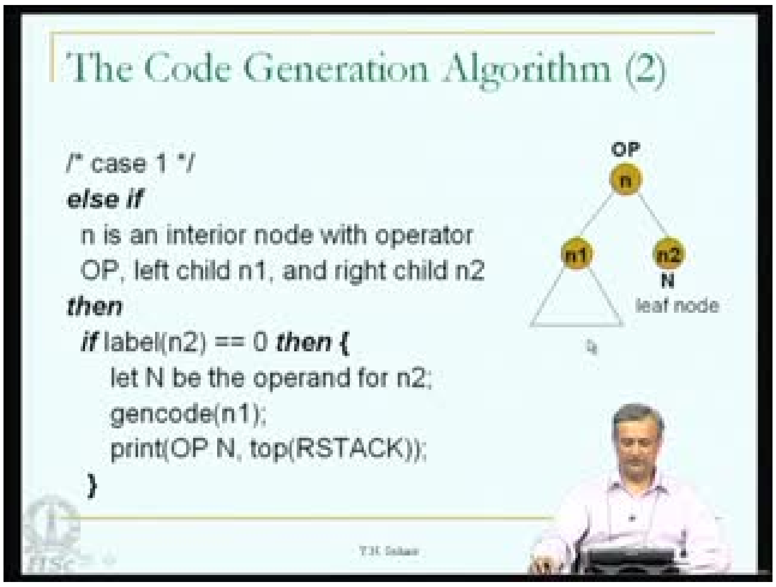
The Code Generation Algorithm (1)

```
Procedure gencode(n);
{ /* case 0 */
  if
    n is a leaf representing
    operand N and is the
    leftmost child of its parent
  then
    print(LOAD N, top(RSTACK))
```

n
N
leaf node
T.H. Sekar

There are 5 cases; rather 6 cases – 0, 1, 2, 3, 4, 5 in this algorithm. The first one is the left. We just have a leaf node and then for the leaf node, it generates a load statement to the register on the top of stack.

(Refer Slide Time: 02:23)



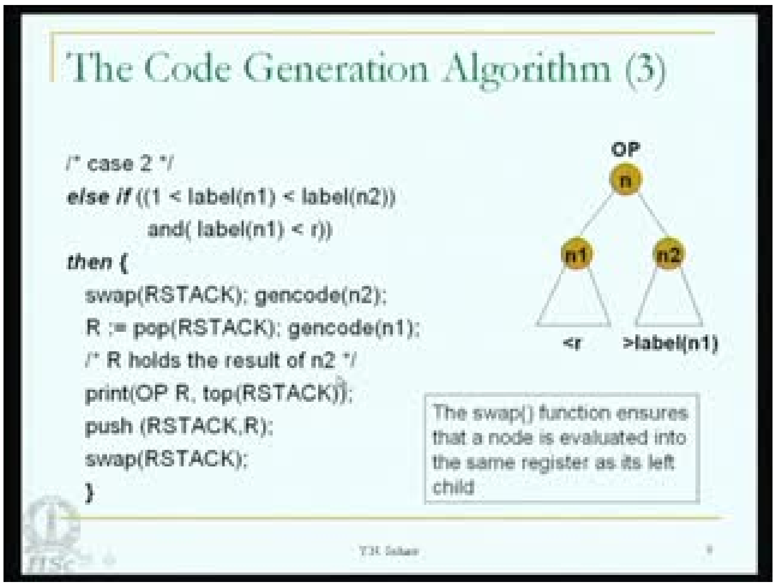
The Code Generation Algorithm (2)

```
/* case 1 */
else if
  n is an interior node with operator
  OP, left child n1, and right child n2
then
  if label(n2) == 0 then {
     let N be the operand for n2;
     gencode(n1);
     print(OP N, top(RSTACK));
  }
```

OP
n
n1
n2
N
leaf node
T.H. Sekar

Case 1: The right child of a node is a leaf node and the left child is a subtree. Then, we simply generate code for n1 by calling gencode and then insure an instruction OP N comma top of RSTACK to take care of this particular operation and this leaf node.

(Refer Slide Time: 02:46)



The Code Generation Algorithm (3)

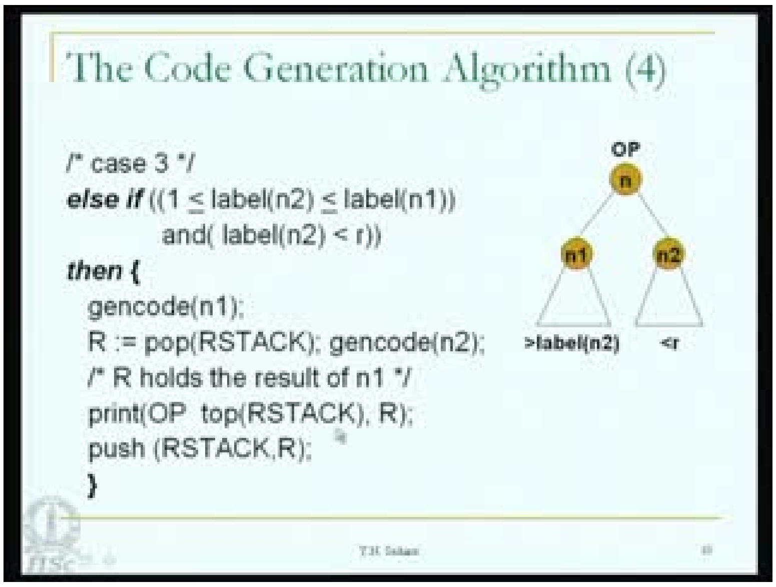
```
/* case 2 */
else if ((1 < label(n1) < label(n2))
        and( label(n1) < r))
then {
  swap(RSTACK); gencode(n2);
  R := pop(RSTACK); gencode(n1);
  /* R holds the result of n2 */
  print(OP R, top(RSTACK));
  push (RSTACK,R);
  swap(RSTACK);
  }
```

The swap() function ensures that a node is evaluated into the same register as its left child

OP
n
n1
n2
<r
>label(n1)

Case 2: Here, the left subtree requires less than r registers and right subtree requires greater than the number required by the node n1. So, this is a slightly special case because it requires a swap of the two registers on the top of the register stack. This is required to make sure that a node is evaluated into the same register as its left child. So, here we swap, then we generate code for the subtree headed by n2. This is because, we always generate code for the subtree, which requires more number of registers and then we generate code for the left subtree. Finally, emit the code as required and push the register on to the stack and swap it again.

(Refer Slide Time: 03:40)



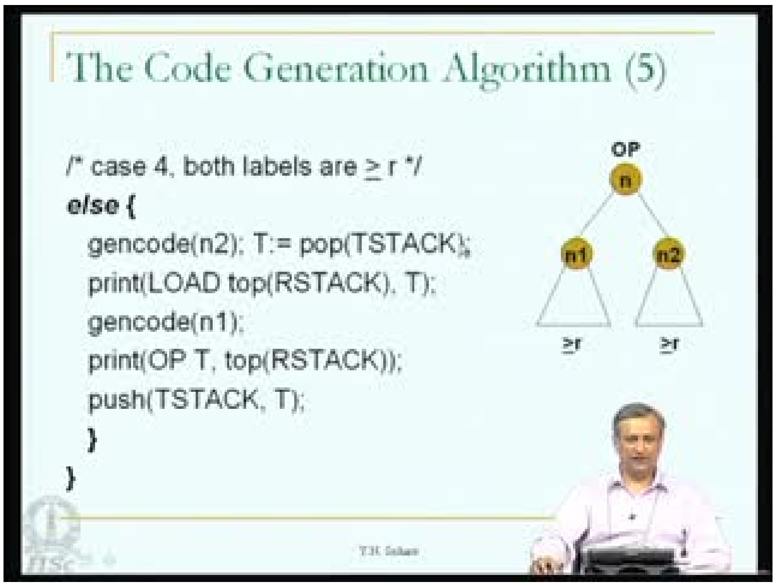
The Code Generation Algorithm (4)

```
/* case 3 */
else if ((1 ≤ label(n2) ≤ label(n1))
          and( label(n2) < r))
then {
  gencode(n1);
  R := pop(RSTACK); gencode(n2);
  /* R holds the result of n1 */
  print(OP  top(RSTACK), R);
  push (RSTACK,R);
  }
```

OP
n
n1
n2
>label(n2)
<r

Then, Case 3: Here, it is the other way. The right subtree requires less than r registers and left subtree requires greater than what n2 requires. So, here, there is no swap needed; we just generate code for n1 followed by generate code for n2, and then finally, the instruction for n itself.

(Refer Slide Time: 04:01)

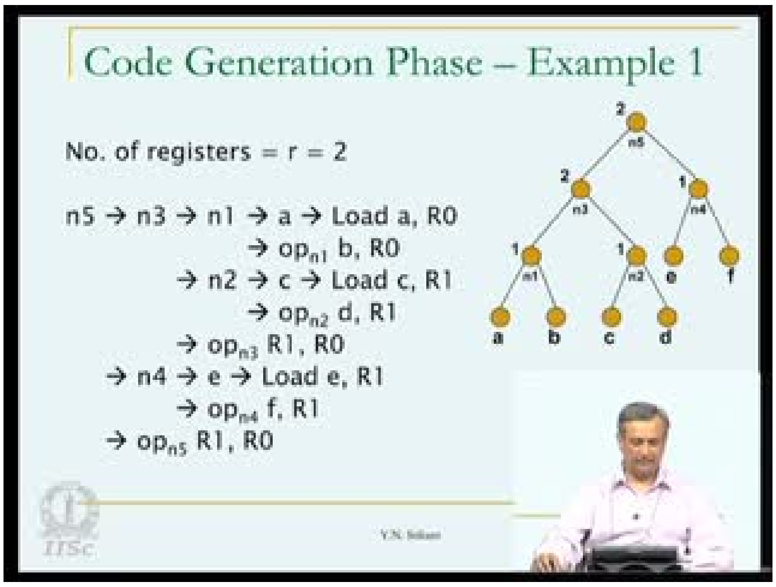


The Code Generation Algorithm (5)

```
/* case 4, both labels are ≥ r */
else {
  gencode(n2); T:= pop(TSTACK);
  print(LOAD top(RSTACK), T);
  gencode(n1);
  print(OP T, top(RSTACK));
  push(TSTACK, T);
  }
}
```

The last case: Both the subtrees require greater than or equal to r number of registers. Here, we require a temporary to store the result because we do not have as many registers as necessary. So, we generate the code for n2, then store the result of n2 into a temporary, generate code for n1, and finally, generate the instruction for the node n. So, this is the summary of how the algorithm works.

(Refer Slide Time: 04:31)



Code Generation Phase – Example 1

No. of registers = r = 2

```
n5 → n3 → n1 → a → Load a, R0
                 → $op_{n1}$ b, R0
          → n2 → c → Load c, R1
                 → $op_{n2}$ d, R1
          → $op_{n3}$ R1, R0
     → n4 → e → Load e, R1
          → $op_{n4}$ f, R1
     → $op_{n5}$ R1, R0
```

Y.N. Srikant

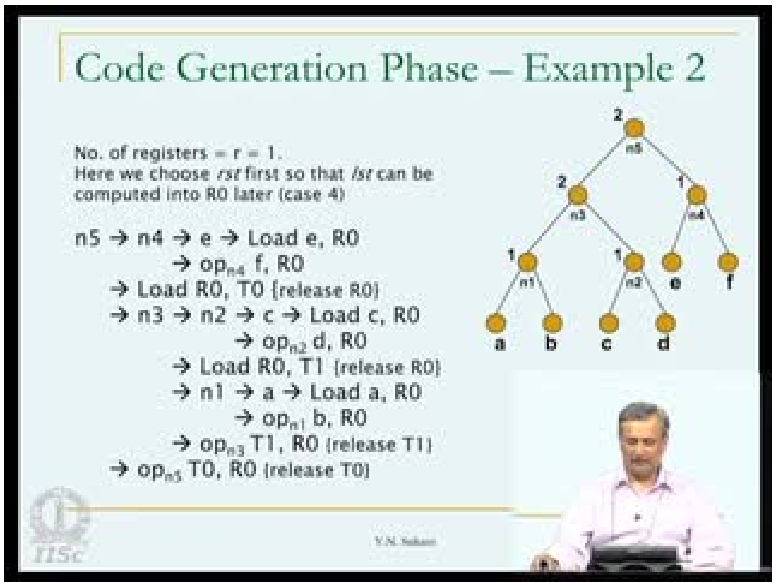
Let us look at an example. Here, we have the same tree for which we computed the min reg values. All are listed here: 1 1 2 3. The number of registers is 2. This requires three registers and then we look at the 2 subtrees: n3 and n4; n3 requires two registers and n4 requires only one register. So, obviously, code for this subtree headed by n3 is generated first followed by the code for n4.

Here, as what it is indicated, start with n5, then you go to n3 Once you go to n3, these two are equal. So, you just go to the left; that is, n1, then it goes to a, and finally, a load is generated for this particular leaf. So, followed by an op instruction for this; op n1 b comma R0.

After this, it goes to this particular subtree n2 (Refer Slide Time: 05:42) because this code cannot be generated without generating code for n2. So, it goes to n2, goes to c, loads c into R1, and then generates code for this op n2 d comma R1. Finally, when it goes to n3, it generates code for n3 op n3 R1 comma R0 there by the result of this entire subtree n3 is held in the register R0, which is the register used by the left subtree of n3.

Later, it goes to n4, generates code for loading e into R1 because now R1 is free and only R0 holds this value. It generates code for op n4 and then finally, op n5 is generated. So, this is a very simple strategy – we have two registers and the subtrees require only 2. So, everything is fine as far as this is concerned.

(Refer Slide Time: 06:46)



Code Generation Phase – Example 2

No. of registers = r = 1.
Here we choose *rst* first so that *lst* can be computed into R0 later (case 4)

```
n5 → n4 → e → Load e, R0
            → op_n4 f, R0
   → Load R0, T0 (release R0)
   → n3 → n2 → c → Load c, R0
                 → op_n2 d, R0
        → Load R0, T1 (release R0)
        → n1 → a → Load a, R0
                 → op_n1 b, R0
        → op_n3 T1, R0 (release T1)
   → op_n5 T0, R0 (release T0)
```

2
n5
2
n3
1
n4
1
n1
1
n2
e
f
a
b
c
d

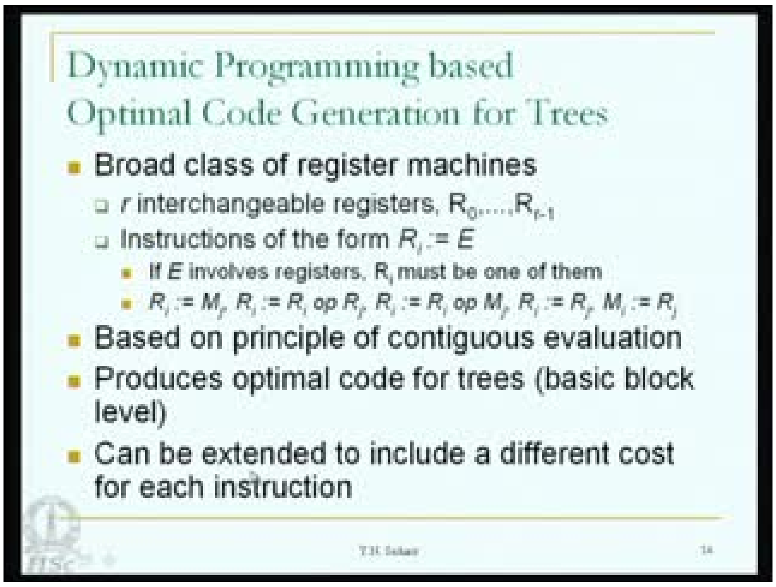
Y.N. Srikant

Now, the number of registers is only 1. So, now we have a small problem. We need a temporary. So, we choose the right subtree because LST has to be computed into R0 later on; the subtree has to be computed into R0. So, we go to this. So, n5, n4, then e, and finally, load e into R0 and op n4 f comma R0. So, this subtree generates code and gets evaluated into R0.

Now, release the register R0, come here: n3, then this subtree is traversed, n2, and then load c into R0; op n2. So, this node is also taken care of. Now, load this register into T1. So, this is also released. Then, you go to n1 load a comma R0; op n1 b comma R0. So, the code for this subtree is now generated.

Now, the code for this entire thing can be generated. Op n3 T1 comma R0. So, R0 contains the result of this entre thing and finally, op n5 T0 comma R0 generates code and evaluates this entire tree. Here, you see that the number of registers is only 1. So, we require temporaries to evaluate the subtrees and this is how it is performed.

(Refer Slide Time: 08:20)



Dynamic Programming based Optimal Code Generation for Trees

- Broad class of register machines
  - r interchangeable registers, $R_0, \ldots, R_{r-1}$
  - Instructions of the form $R_i := E$
    - If E involves registers, R_i must be one of them
    - $R_i := M_j$, $R_i := R_i \text{ op } R_j$, $R_i := R_i \text{ op } M_j$, $R_i := R_j$, $M_i := R_j$
- Based on principle of contiguous evaluation
- Produces optimal code for trees (basic block level)
- Can be extended to include a different cost for each instruction

Y.N. Srikant 14

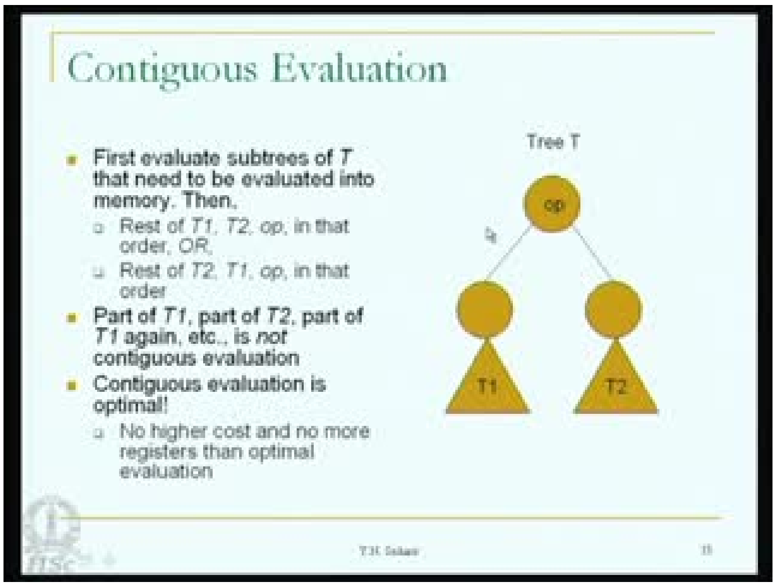
So far we saw code generation examples based on Sethi-Ullman algorithm, which requires a restricted form of machine. If we go to the next type of code generation algorithm, which uses dynamic programming, the restrictions on the machines actually can be removed to some extent.

Broader class of register machines can be accommodated here. So, for example, they would have r interchangeable registers, R0 to R r minus 1. Then, instructions are of the form R i equal to E and if E involves registers, R i must be 1 of them. That is all. That is the only restriction that we have. Form of instructions are these – R i equal to M j R i equal to R i op R j R i equal to R i op M j. So, now you see R i equal to R j and M i equal to R j. So, all types of instructions copy, then store into memory, and then register can be… You can add more varieties also. For example, you can add R i equal to M j op R i. We have considered only these instructions to make the examples simpler; that is all.

You can literally add any type of instruction that you have. Each of these will translate to actual machine code as this would be for example, load M j comma R i; this would be moved R i op of R j comma R i and so on and so forth. So, the exact syntax can be different for every machine, but this is the essence of the instruction. There is a very important principle known as the principle of contiguous evaluation, which we are going to deal with very soon. This entire dynamic programming based code generation strategy is based on this principle of contiguous evaluation.

Dynamic programming is an optimization technique. So, definitely, when this is used, it produces optimal code for trees at the basic block level again. So, it can be extended to include a different cost for each instruction. Here, all the instructions have a single cost, but extension can be made to introduce a different cost for each one of the instructions.

(Refer Slide Time: 11:10)



Contiguous Evaluation

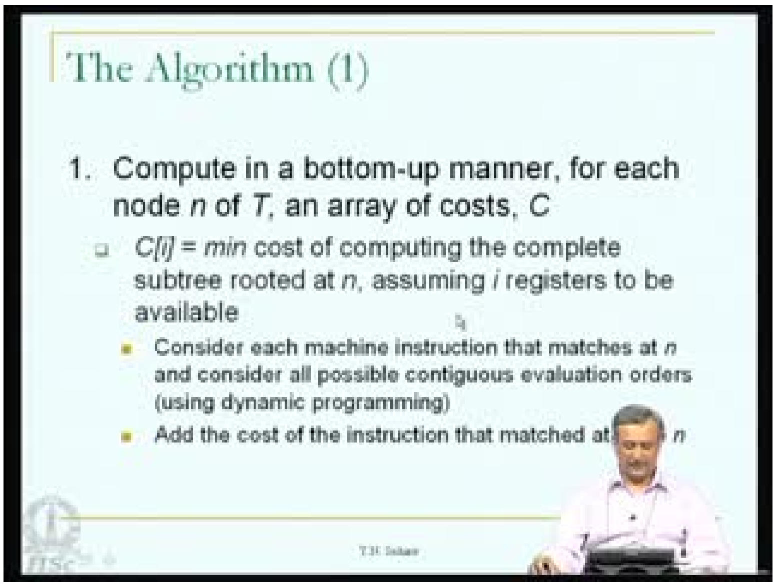
- First evaluate subtrees of *T* that need to be evaluated into memory. Then,
  - Rest of *T1*, *T2*, *op*, in that order, *OR*,
  - Rest of *T2*, *T1*, *op*, in that order
- Part of *T1*, part of *T2*, part of *T1* again, etc., is *not* contiguous evaluation
- Contiguous evaluation is optimal!
  - No higher cost and no more registers than optimal evaluation

Let us see what contiguous evaluation is. You have a tree here – Tree T; that is an op here. Then, there is a subtree T1 and another subtree T2. You could actually evaluate this entire subtree T1, then evaluate the entire subtree T2, and finally, evaluate the op. Or, you could do the entire T2 first, then the T1, and then the op. So, both these are examples of contiguous evaluation.

What is not contiguous evaluation? You generate code to evaluate a part of T1, then the code for evaluating a part of T2 appears, then you go back to evaluation of T1, and then return to the evaluation of T2, so on and so forth. Finally, once these two are completed, you do the evaluation of op. So, this hopping between T1 and T2 doing partial evaluations is not contiguous evaluation.

What is the advantage of contiguous evaluation? Contiguous evaluation is optimal. In other words, no higher cost and no more registers than any optimal evaluation. So, contiguous evaluation is guaranteed to give you optimal cost compared to any other evaluation algorithm. So, if we follow contiguous evaluation and then say – do entire T1 followed by T2 op, or entire T2 followed by T1 op, that is sufficient to yield optimal code. There is no need to jump between these two and then getting into trouble, but of course, there are machines for which contiguous evaluation does not yield optimal code. So, examples of this are – when two registers or two consecutive registers are required for a floating point or integer multiplication, etcetera, for such machines, it is still possible to extend this dynamic programming based algorithm and use it.

(Refer Slide Time: 13:32)



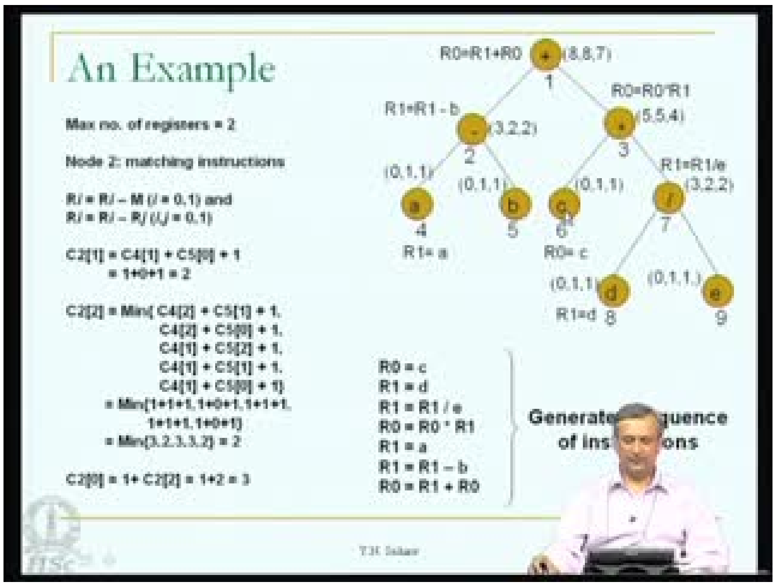
The Algorithm (1)

1. Compute in a bottom-up manner, for each node *n* of *T*, an array of costs, *C*
   - *C[i]* = *min* cost of computing the complete subtree rooted at *n*, assuming *i* registers to be available
     - Consider each machine instruction that matches at *n* and consider all possible contiguous evaluation orders (using dynamic programming)
     - Add the cost of the instruction that matched at *n*

What is algorithm? First step is to compute in a bottom-up manner, for each node n of T, an array of costs, C. So, how many elements this array have? You are really going to look at the cost of computing a node using 0 number of registers: one register, two registers, three registers, etcetera. The number of elements is as many as the number of registers available for the evaluation.

C i is the minimum cost of computing the complete subtree rooted at n, assuming i registers to be available. So, the length of the vector or the array is as much as the number of registers available to us and of course, plus 1 for 0 number of registers. How do we do this? Let us first see how exactly a tree is decorated using such a vector and then come back to this algorithm.

(Refer Slide Time: 14:41)



An Example

Max no. of registers = 2

Node 2: matching instructions

R*i* = R*i* – M (*i* = 0,1) and
R*i* = R*i* – R*j* (*i*,*j* = 0,1)

C2[1] = C4[1] + C5[0] + 1
= 1+0+1 = 2

C2[2] = Min{ C4[2] + C5[1] + 1,
C4[2] + C5[0] + 1,
C4[1] + C5[2] + 1,
C4[1] + C5[1] + 1,
C4[1] + C5[0] + 1}
= Min{1+1+1, 1+0+1, 1+1+1,
1+1+1, 1+0+1}
= Min{3,2,3,3,2} = 2

C2[0] = 1+ C2[2] = 1+2 = 3

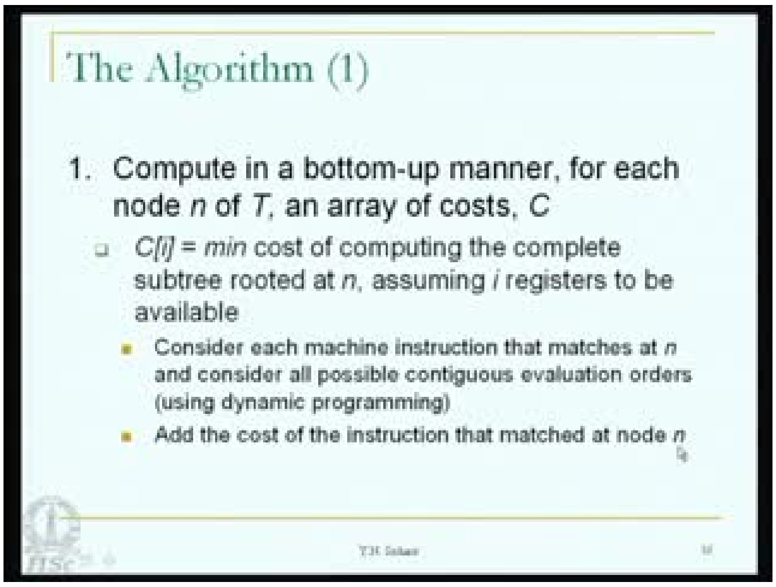
R0 = c
R1 = d
R1 = R1 / e
R0 = R0 * R1
R1 = a
R1 = R1 – b
R0 = R1 + R0

Generate[illegible]quence of ins[illegible]ons

Y.N. Srikant

For example, here is a tree and here are our cost vectors. So, 8 8 7 says the cost of evaluating this entire subtree with zero registers is 8, with one register it is still 8, and with two registers it is 7. Similarly, if you consider a leaf node here, the cost of evaluating this particular leaf node with zero registers is 0 simply because it is already in memory. So, there is no need to compute anything, whereas the cost of computing it with one register or two registers is still 1 because you need to generate an instruction, which moves this particular element from memory to that register.

(Refer Slide Time: 15:31)



The Algorithm (1)

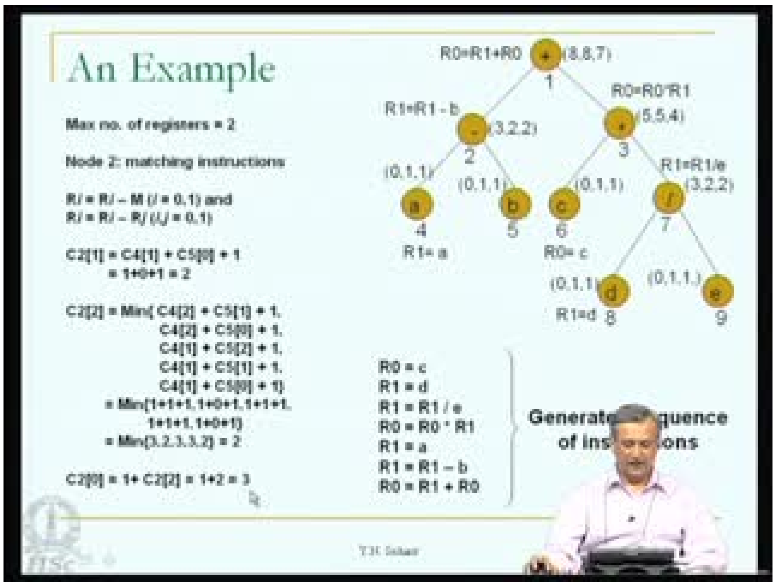
1. Compute in a bottom-up manner, for each node *n* of *T*, an array of costs, *C*
    - *C[i]* = *min* cost of computing the complete subtree rooted at *n*, assuming *i* registers to be available
        - Consider each machine instruction that matches at *n* and consider all possible contiguous evaluation orders (using dynamic programming)
        - Add the cost of the instruction that matched at node *n*

Y.N. Srikant

How do we compute this vector? We consider each machine instruction that matches at n and then consider all possible contiguous evaluation orders using dynamic programming. So, there may be many instructions and there may be many contiguous evaluation orders. We must consider all these, use dynamic programming, and then compute the vector.

Finally, add the cost of the instruction to that matched at node n. The instruction that matches at node n also has to be taken care of. So, what we are trying to do here is – using dynamic programming, we are going to retain the cost at the lower levels of the tree and then use them as we go on to the higher levels of the tree. We will see in the example how this is done.

(Refer Slide Time: 16:28)



An Example

Max no. of registers = 2

Node 2: matching instructions

$Ri = Ri - M$ (i = 0,1) and
$Ri = Ri - Rj$ (i,j = 0,1)

$C2[1] = C4[1] + C5[0] + 1$
$= 1+0+1 = 2$

$C2[2] = \text{Min}\{C4[2] + C5[1] + 1,$
$C4[2] + C5[0] + 1,$
$C4[1] + C5[2] + 1,$
$C4[1] + C5[1] + 1,$
$C4[1] + C5[0] + 1\}$
$= \text{Min}\{1+1+1, 1+0+1, 1+1+1, 1+1+1, 1+0+1\}$
$= \text{Min}\{3,2,3,3,2\} = 2$

$C2[0] = 1 + C2[2] = 1+2 = 3$

R0 = c
R1 = d
R1 = R1 / e
R0 = R0 * R1
R1 = a
R1 = R1 – b
R0 = R1 + R0

Generate[illegible]quence of ins[illegible]ons

T.K. Srikant

Let us look at that example and come back. The maximum number of registers available to us is 2. Node 2 is what we are going to consider. So, we are going to consider node 2 and see how to compute this factor. For the leaves, it is always going to be 0 1 1. Look at all the leaves: a, b, c, d and e. The cost vector is always 0 1 1 because there are only two registers. The cost of computing something into memory is 0 because it is already in memory, whereas moving something into a register is always 1. That is why, these are all 0 1 1.

What are the instructions that match at this particular node; that is, node number 2? With our instruction set, we have instructions of the form Ri equal to Ri minus M; we do not have any instructions of the form Ri equal to M minus Ri. So, we are forced to put the left subtree into a register and possibly let the right subtree in memory. This holds for i equal to 0 comma 1. In other words, we have R0 equal to R0 minus M and R1 equal to R1 minus M. So, these are the 2 instructions, which match. Then, we have the pattern Ri equal to Ri minus Rj. So, both can be in registers. Again, we have R0 equal to R0 minus R1 and then we have R1equal to R1 minus R0. So, all these possibilities exist.

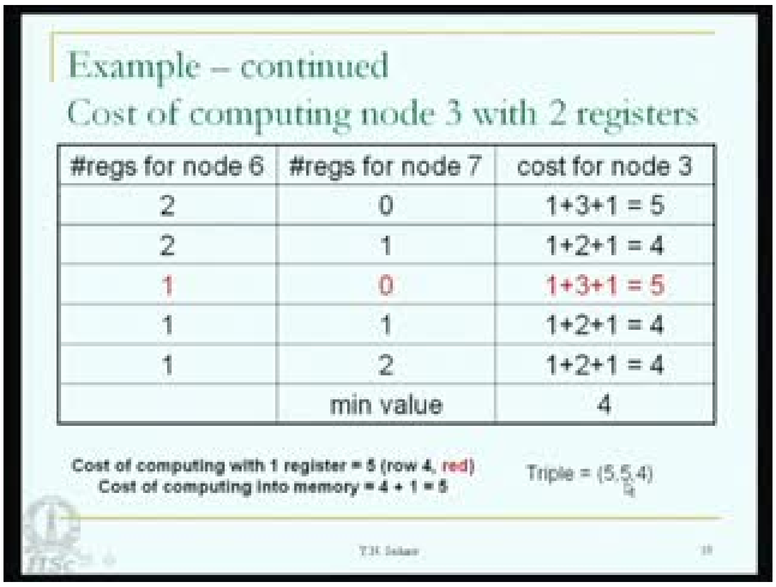
Let us see what happens. Let us compute the cost of this particular node with just one register. So, this C2 indicates cost of node 2 with one register. If you use one register, then the cost of computing C4, the child with one register; then no more registers are left; C5 with zero registers and then the cost of this node itself; that is, plus 1. So, from the vector here, we easily see that C4 of 1 is 1; this is 0 and this is 1. Then, C5 of 0 is 0 and then plus 1 that (( )) 2.

What about two registers? Now, there are many possibilities. So, we take the minimum of these; (Refer Slide Time: 19:01) C4 with two registers, then one of the registers retains the result, and other is released. So, C5 with one register and then plus 1 for this node; C4 with two registers, C5 with zero registers plus 1. So, this can be in memory. C4 with one register, C5 with two registers; exactly the other way plus 1. C4 with one register, C5 with one register plus 1, and then C4 with one register, C5 with zero registers plus 1. Just because we are given two registers, it does not mean we must use both of them. We may possibly use only one.

If you look at these, using these vectors here (Refer Slide Time: 19:49), we get 1 plus 1 plus 1 comma 1 plus 0 plus 1 etcetera for all these values. Then, the minimum of all these is 2. So, the cost of computing node 2 with two registers; the minimum value is 2. Now, the cost of computing node 2 with zero registers is always 1 plus the cost of computing it with two registers because use all the registers, then store the value computed into memory, and then release all the registers. That is our principle. So, 1 plus C2 of 2 will be 1 plus 2 equal to 3.

Our vector is really going to be – the first component is 3 with zero registers, second component is 2 with one register, and third component is 2 again with two registers. So, we can compute the vectors in a similar manner.

(Refer Slide Time: 20:49)



Example – continued

Cost of computing node 3 with 2 registers

#regs for node 6	#regs for node 7	cost for node 3
2	0	1+3+1 = 5
2	1	1+2+1 = 4
1	0	1+3+1 = 5
1	1	1+2+1 = 4
1	2	1+2+1 = 4
	min value	4

Cost of computing with 1 register = 5 (row 4, red)
Cost of computing into memory = 4 + 1 = 5

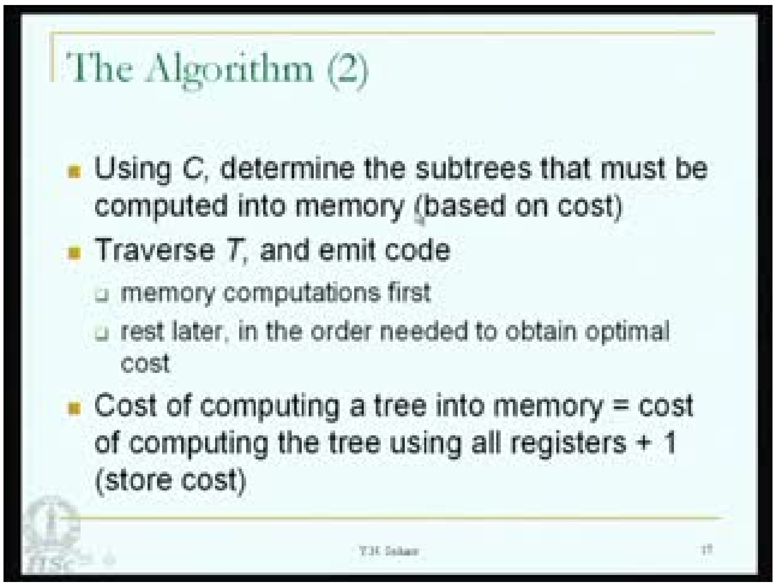
Triple = (5,5,4)

Y.N. Srikant

Let me show you one more example; very simple. For node number 3; that is, (Refer Slide Time: 20:54) this particular node, we consider many possibilities again; 6 – the left child with two registers, right child with 0, left child with 2, right child with 1, then left child with 1, right child with 0, left child with 1, right child with 1, and then left child with 1 and right child with 2. These are the various possibilities that we have. So, minimum of all these is 4.

Cost of computing with one register is this particular row (Refer Slide Time: 21:23), which is marked in red. That is because we have 1 for the left child and 0 for the right child. So, we are using only one register here. So, that would be 5. Cost of computing into memory is this particular cost (Refer Slide Time: 21:37) with two registers plus 1; that is 5. So, triple is 5 for zero registers, 5 for one register, and 4 for two registers.

(Refer Slide Time: 21:49)



The Algorithm (2)

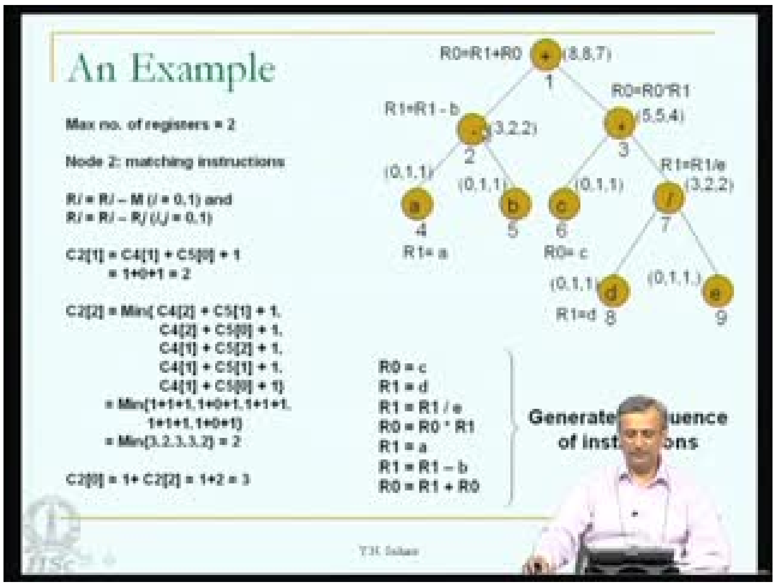
- Using *C*, determine the subtrees that must be computed into memory (based on cost)
- Traverse *T*, and emit code
  - memory computations first
  - rest later, in the order needed to obtain optimal cost
- Cost of computing a tree into memory = cost of computing the tree using all registers + 1 (store cost)

Y.N. Srikant 17

Using the cost vector, we determine the subtrees that must be computed into memory. So, that is based on the cost. Just look at the vector of cost and find out which is the cheapest. So, those particular nodes where computation into memory is cheapest are segregated.

Now, traverse T and emit code. Here, you must actually emit code for the memory computations first. So, for those nodes, which will be computed into memory, we must emit code first. So, those subtrees must be evaluated first and then the others. So, in the order needed to obtain the optimal cost. The algorithm is simple – you look at the tree, label it with these vectors, then consider all those subtrees, which have low cost when computed into memory, generate the code for such subtrees, and finally, generate code for other parts of the tree.

(Refer Slide Time: 23:11)



An Example

Max no. of registers = 2

Node 2: matching instructions

Ri = Ri – M (i = 0,1) and
Ri = Ri – Rj (i,j = 0,1)

C2[1] = C4[1] + C5[0] + 1
= 1+0+1 = 2

C2[2] = Min{ C4[2] + C5[1] + 1,
C4[2] + C5[0] + 1,
C4[1] + C5[2] + 1,
C4[1] + C5[1] + 1,
C4[1] + C5[0] + 1}
= Min{1+1+1, 1+0+1, 1+1+1,
1+1+1, 1+0+1}
= Min{3,2,3,3,2} = 2

C2[0] = 1+ C2[2] = 1+2 = 3

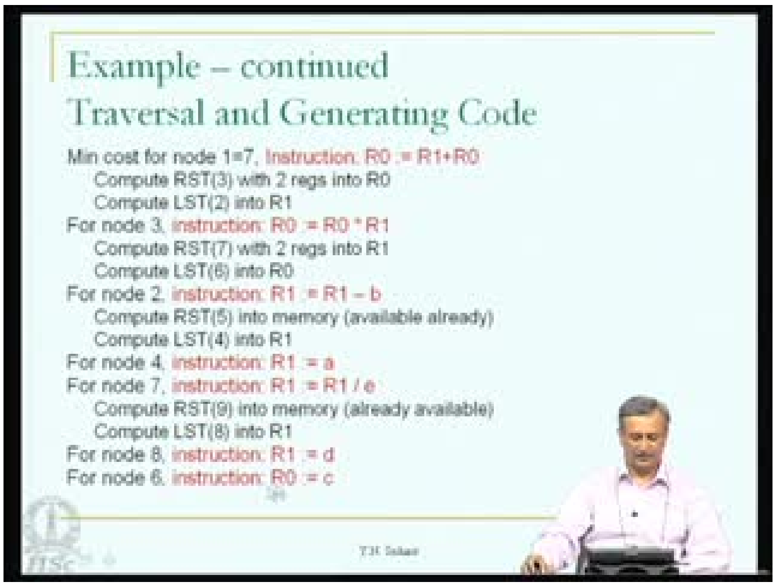
R0 = c
R1 = d
R1 = R1 / e
R0 = R0 * R1
R1 = a
R1 = R1 – b
R0 = R1 + R0

Generate[illegible]uence of inst[illegible]ons

Y.N. Srikant

Let us look at a simple example. Before that, maybe we can look at the picture itself. Look at this – we have decorated all these with vectors. Now, the cost of computing the root with two registers is the minimum; that is, 7. We also need… As it is quite obvious that in order to generate code, we also need to keep track of the patterns or the instructions, which were used at these 2 children, and finally, the instruction, which is used at this particular node in order to yield this cost.

(Refer Slide Time: 23:54)



Example – continued
Traversal and Generating Code

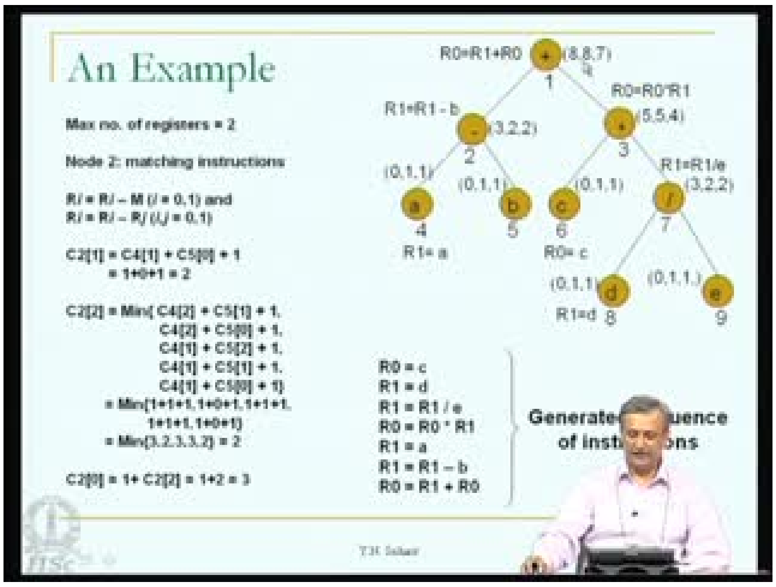
Min cost for node 1=7, Instruction: R0 := R1+R0
Compute RST(3) with 2 regs into R0
Compute LST(2) into R1
For node 3, instruction: R0 := R0 * R1
Compute RST(7) with 2 regs into R1
Compute LST(6) into R0
For node 2, instruction: R1 := R1 – b
Compute RST(5) into memory (available already)
Compute LST(4) into R1
For node 4, instruction: R1 := a
For node 7, instruction: R1 := R1 / e
Compute RST(9) into memory (already available)
Compute LST(8) into R1
For node 8, instruction: R1 := d
For node 6, instruction: R0 := c

Y.N. Srikant

At every node, we go down from here (Refer Slide Time: 23:47). This gives me 7. So, look at the possibilities, which yielded to give this 7. That is what I was showing here. Minimum cost for node 1 is 7. The instruction, which yielded this cost was R0 equal to R1 plus R0. If you trace it back, then it says compute the right subtree; node 3 with two registers into R0 and compute the left subtree; that is, node 2 into R1. So, this actually yielded the cost as 7. We must go down further, look at node number 3 and see what happens.

The instruction, which matched at that point and gave us the right cost was R0 equal to R0 star R1. Now, 3 has 2 children 7 and 6. So, 7 must be computed with two registers into R1 and 6 must be computed into R0. So, go down for node number 2. The instruction, which gave us this cost was R1 equal to R1 minus b; we keep going further. This RST was computed into memory; that was the optimal cost, but it is already in memory. This is a leaf node; this b. There is nothing to do and we must compute the left subtree into R1. If we keep 4, the computation, which makes 4 get into R1 is R1 equal to a. So, this instruction is emitted. For node 7, the instruction is R1 equal to R1 slash e. So, this can be broken down into 2 more steps: 9 into memory and 8 into R1. So, for 8, the instruction is R1 equal to d and for 6, the instruction is R0 equal to c.

(Refer Slide Time: 25:34)



An Example

Max no. of registers = 2

Node 2: matching instructions

Ri = Ri – M (i = 0,1) and
Ri = Ri – Rj (i,j = 0,1)

C2[1] = C4[1] + C5[0] + 1
= 1+0+1 = 2

C2[2] = Min{ C4[2] + C5[1] + 1,
C4[2] + C5[0] + 1,
C4[1] + C5[2] + 1,
C4[1] + C5[1] + 1,
C4[1] + C5[0] + 1}
= Min{1+1+1, 1+0+1, 1+1+1,
1+1+1, 1+0+1}
= Min{3,2,3,3,2} = 2

C2[0] = 1+ C2[2] = 1+2 = 3

R0 = c
R1 = d
R1 = R1 / e
R0 = R0 * R1
R1 = a
R1 = R1 – b
R0 = R1 + R0

Generated sequence of instructions

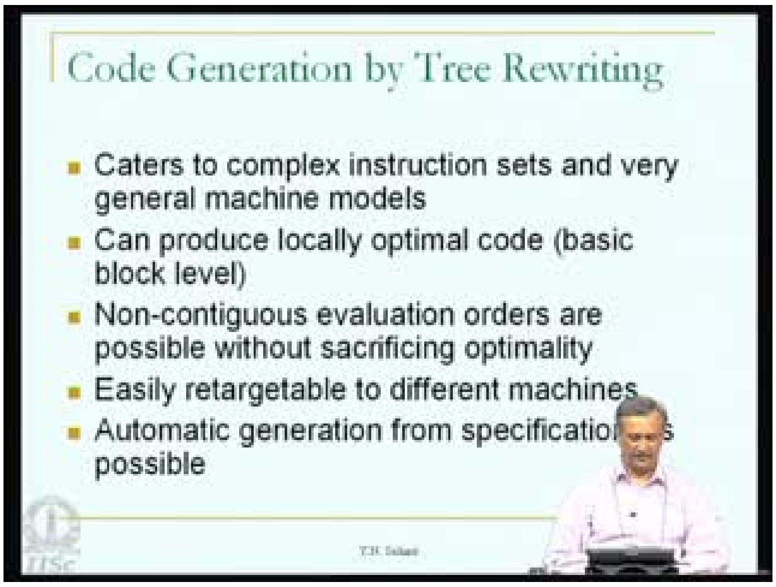
T.H. Sahar

In other words, what we really did was – take this 7, find out who is responsible for giving this 7, and similarly go down further. Finally, you would have decorated all the nodes in the tree with appropriate instructions. Now, if you simply emit this code in a particular order, it will give you the code itself.

For example, R0 equal to c, R1 equal to d, and then R1 equal to R1 slash e. So, you cannot really do a preorder traversal and come back, but we have to actually generate the code in the order dictated by this particular cost (Refer Slide Time: 26:14). So, this first, then this, and so on; that is what it says. We have to do RST 3 first and then 2. If you do not do this (Refer Slide Time: 26:23), the code cannot be generated in an optimal fashion.

That is about code generation by dynamic programming. The dynamic programming based algorithms is very good, but it still has some restrictions on the type of instructions and so on and so forth.

(Refer Slide Time: 26:27)



Code Generation by Tree Rewriting

- Caters to complex instruction sets and very general machine models
- Can produce locally optimal code (basic block level)
- Non-contiguous evaluation orders are possible without sacrificing optimality
- Easily retargetable to different machines
- Automatic generation from specifications possible

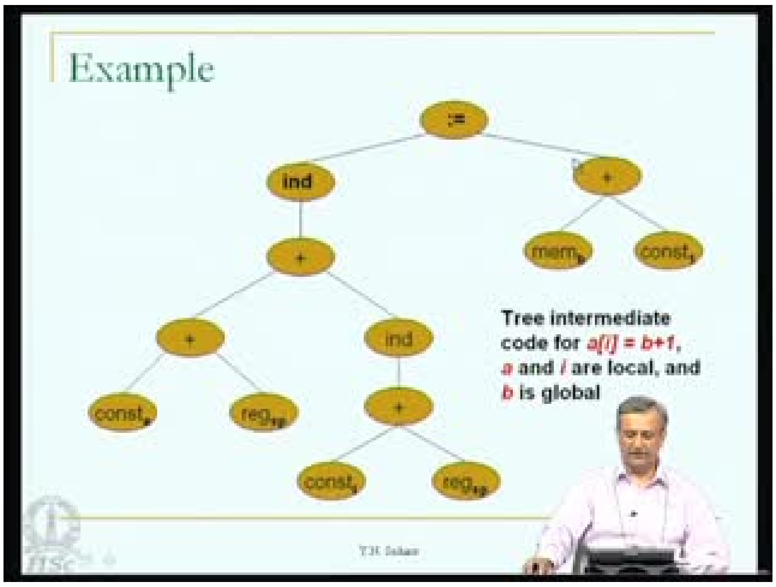
Y.N. Srikant

The next strategy is to use tree rewriting in order to do code generation. Using tree rewriting, it caters to complex instruction sets and very general machine models. There is no problem about any restriction on the type of instructions and so on and so forth.

It can definitely produce locally optimal code like the others because they are still using dynamic programming here as well. This is a combination of dynamic programming and tree rewriting. Non-contiguous evaluation orders are possible without sacrificing optimality. Here, there is no problem about evaluation order. Tree pattern matching will give us the order as we will see very soon. It is easily retargetable to different machines.

So, we can use a tree grammar in order to specify the machine and then use a generator for tree grammars to give us the code generator. Automatic generation from specifications is possible.

(Refer Slide Time: 27:54)



Example

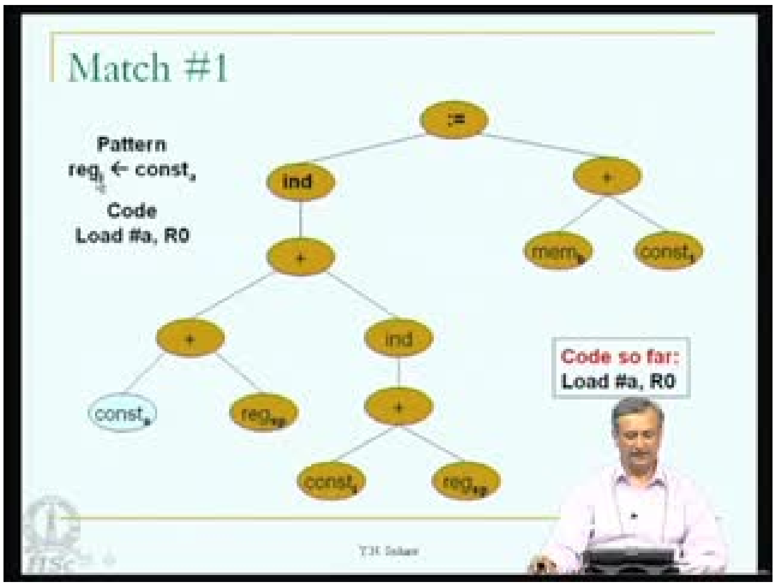
Tree intermediate code for *a[i] = b+1*, *a* and *i* are local, and *b* is global

Let us look at an example of how tree pattern matching is used or tree rewriting is used to generate code. As is obvious to perform tree rewriting, we need a tree intermediate code. Let us take the example of the tree intermediate code for the high level statement a i equal to b plus 1. Here, the variables a and i are local to the procedure and b is a global variable. So, the root node, of course is the assignment operator, the right subtree of the assignment corresponds to b plus 1.

There is a plus, memb is the offset of the variable b; it is a global variable; so, it has a static allocation, and then const 1 shows that it is a constant 1. The left subtree corresponds to a i, rather the address of a i. Let us look at it. First of all, we need i, then we need to dig into the array a, look at the ith location and take the address of that. These are local variables. So, they are all stored on the activation record. There is a base address for a, rather the offset of the array a in the activation record is some constant a; const a. Then, there is a stack pointer, which is in the stack pointer register.

If we add these two (Refer Slide Time: 29:26), we get to the beginning of the array a in the activation record. So, that is what this plus does. Then, take the offset of the variable i, add s p, and then you get to the location of i inside the activation record. Then, one more level of ind gives you the content of that particular location i. Now, value of i is obtained. This is added (Refer Slide Time: 29:56) to this particular a and this ind, which is always on the left side of assignment will yield us the address. This ind end and this ind end are different. When it is on the left side, this ind (Refer Slide Time: 30:09) will give us the address. So, what we get is the address of a i. Address of a i here (Refer Slide Time: 30:08), the b plus 1 here, and this assignment operator shows that this value is loaded into this particular address (Refer Slide Time: 30:26). So, this is the tree intermediate code for our example.

(Refer Slide Time: 30:32)



Match #1

Pattern
$reg_i \leftarrow const_a$

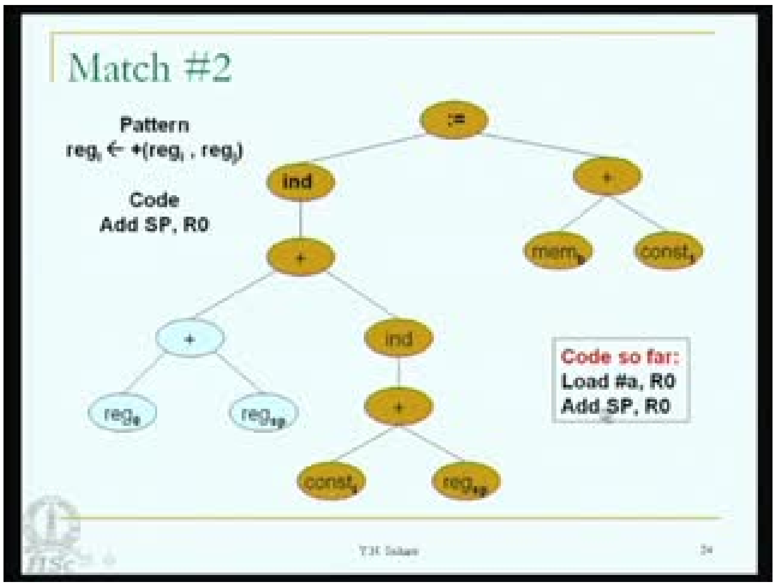
Code
Load #a, R0

Code so far:
Load #a, R0

Let us see how tree pattern matching happens here. The pattern – tree pattern is indicated as similar to a context free grammar production. It is just that the arrow is in the reverse direction. This pattern says – whenever there is a constant, it can be replaced by or rewritten by a register and when this is done it is necessary to generate the code, load the constant value a into R0. This is quite logical.

Here, we have const. Let us see what happens if we use this particular pattern at this point. So, this will be (Refer Slide Time: 31:15) replaced by a node called register.

(Refer Slide Time: 31:20)



Match #2

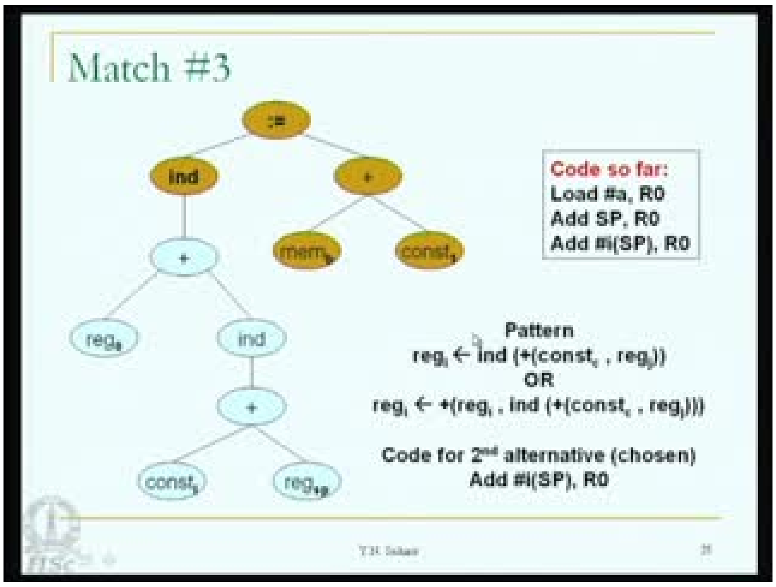
Pattern
$reg_i \leftarrow +(reg_i , reg_j)$

Code
Add SP, R0

Code so far:
Load #a, R0
Add SP, R0

Here, we have register. Now, the rest of it - the tree has been rewritten at this point and the pattern considered is this particular one. So, the pattern is plus, left child is reg i and right child is reg j; reg i corresponds to register 0 and reg j corresponds to register s p; stack pointer. When we match this particular pattern, then the code add SP to R0 must be generated; that means, this is addition. So far, if you match this pattern and the previous one, the code that would have been generated is load constant a into R0, add SP to R0.

(Refer Slide Time: 32:08)



Match #3

Code so far:
Load #a, R0
Add SP, R0
Add #i(SP), R0

Pattern
$reg_i \leftarrow ind\ (+(const_c , reg_j))$
OR
$reg_i \leftarrow +(reg_i , ind\ (+(const_c , reg_j)))$

Code for 2nd alternative (chosen)
Add #i(SP), R0

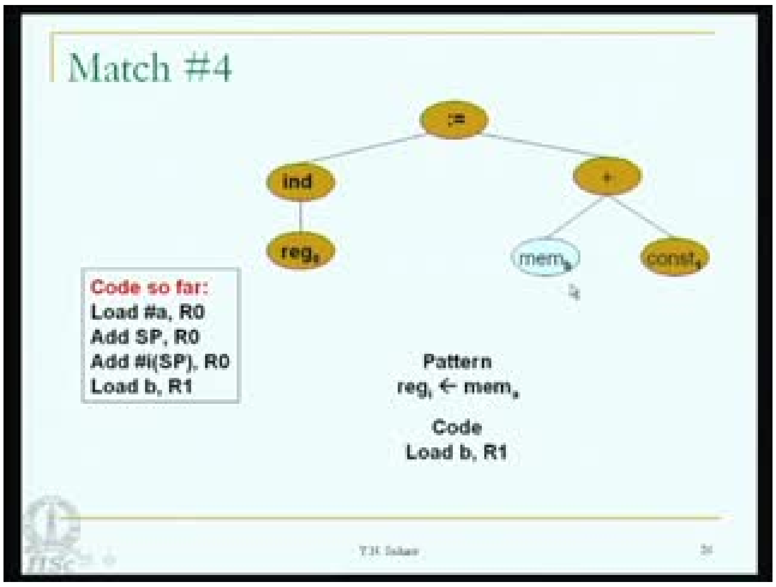
Let us see what happens now. At this point, we have two possibilities; 2 patterns match –

first pattern is regi is ind of plus const c comma regj; that is, this particular subtree (Refer Slide Time: 32:26); ind plus const reg. Or, it is the bigger tree. So, we have plus, reg, then ind, plus, const, and reg. Whether we match the smaller pattern or the larger pattern, correct code can still be generated.

Normally, we use a bigger subtree, which is matched because shorter code sequence can be generated. If we assume that the second pattern matches, then the code that is generated for the second pattern - when this (Refer Slide Time: 33:05) matches, it is add constant i SP comma R0. This is indexed mode of addressing. So, i is added to SP, then the contents of SP are now treated as an address. So, you go to that address, fetch its value, and then that is added to R0. So, that computes this entire thing and that gives you the address of a of i.

The code generated so far is this (Refer Slide Time: 33:37). Now, remember - this has been computed (Refer Slide Time: 33:45), this is already in R0. That is why, all these are possible. So, these three instructions have been generated so far.

(Refer Slide Time: 33:53)


Match #4

Code so far:
```
Load #a, R0
Add SP, R0
Add #i(SP), R0
Load b, R1
```

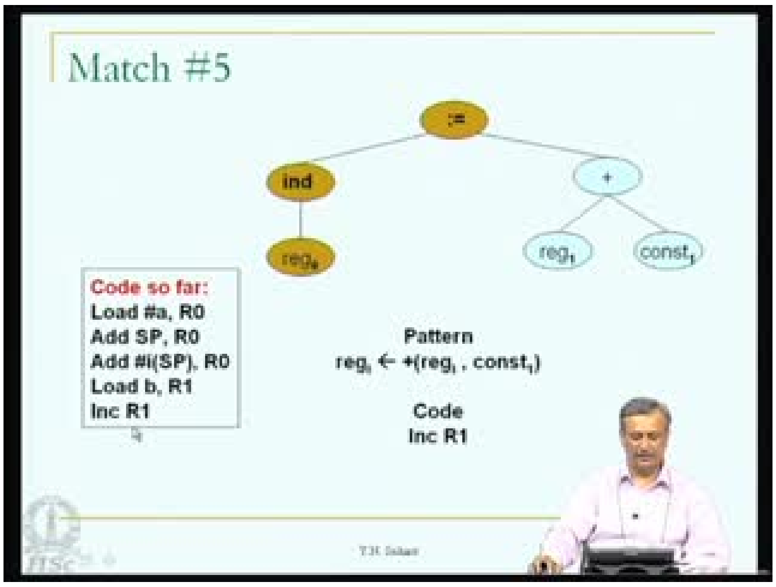
Pattern
$reg_1 \leftarrow mem_b$

Code
Load b, R1

Now, we cannot do much with this particular part until we finish this. So, we come here. This is non-contiguous evaluation. Already we did one part here, we came here, and then we possibly go back. So, memb indicates that it is a memory location and there is a pattern possibly matching memb and changing it to a register.

If we do that, then the code that must be generated is load b comma R1 thereby transferring b into the register R1. So, then this will change to reg node with 1. So, the code so far is load constant a to R0, add SP to R0, add constant i SP to R0, and then load b to R1.

(Refer Slide Time: 34:47)



Match #5

Code so far:
Load #a, R0
Add SP, R0
Add #i(SP), R0
Load b, R1
Inc R1

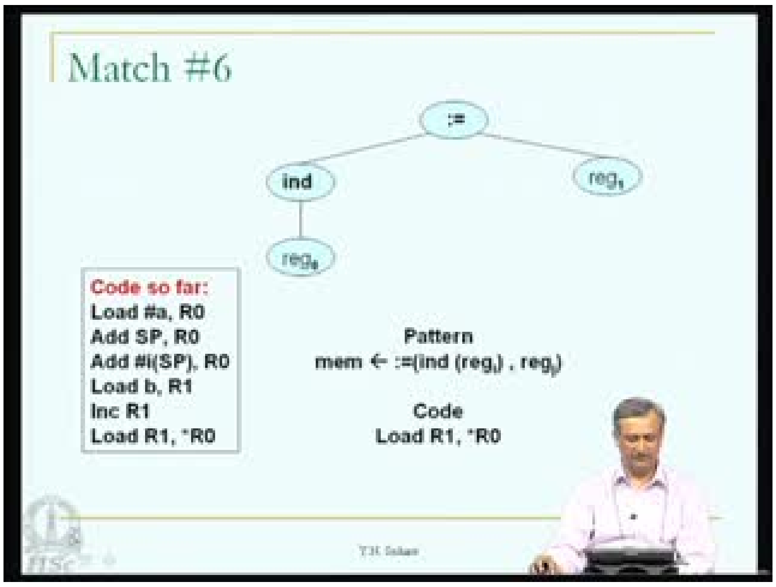
Pattern
$reg_1 \leftarrow +(reg_1, const_1)$

Code
Inc R1

T.H. Srikant

Finally, this particular entire thing can be taken care of. The pattern, which matches is plus reg and const and the code generated is increment R1 because this constant is just 1. Now, we add this increment R1 to this particular code.

(Refer Slide Time: 35:04)



Match #6

Code so far:
Load #a, R0
Add SP, R0
Add #i(SP), R0
Load b, R1
Inc R1
Load R1, *R0

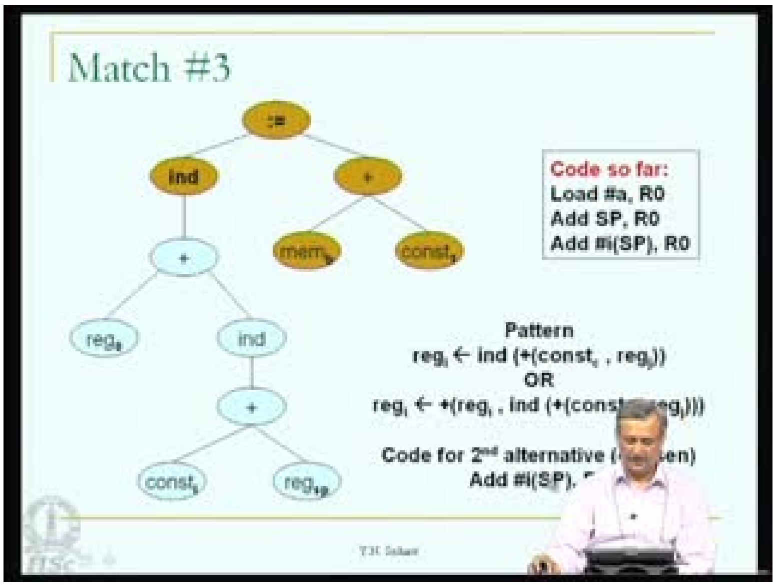
Pattern
$mem \leftarrow :=(ind\ (reg_0), reg_1)$

Code
Load R1, *R0

T.H. Srikant

Finally, this entire thing is matched by the pattern – assign ind reg comma reg. So, this is the address; whatever is given in reg_0 is taken as an address and the code corresponding to it is load R1 to star of R0. So, whatever is here is transferred to this particular address. That is the last instruction that is generated. As you can see, the tree pattern matching process is definitely something that takes a part of the left side, then the right side, then the left side, and so on. So, it is non-contiguous, but it gives you more flexibility.

(Refer Slide Time: 36:04)



Match #3

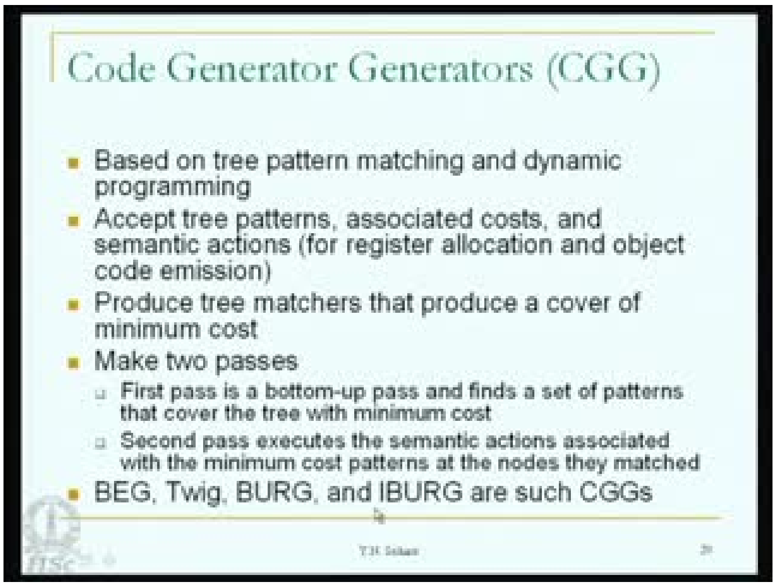
Code so far:
Load #a, R0
Add SP, R0
Add #i(SP), R0

Pattern
$reg_i \leftarrow$ ind (+(const$_c$, reg_j))
OR
$reg_i \leftarrow$ +(reg_i , ind (+(cons[illegible]reg_j)))

Code for 2nd alternative ([illegible]en)
Add #i(SP), [illegible]

Now, the challenge is how to use this along with dynamic programming and generate the optimal code. The reason is – we saw in just one example here that there are two alternatives for pattern matching. With large intermediate code, there may be many alternatives and there are const implications of each alternative. So, how to choose the best? This is carried out by using dynamic programming.

(Refer Slide Time: 36:24)



Code Generator Generators (CGG)

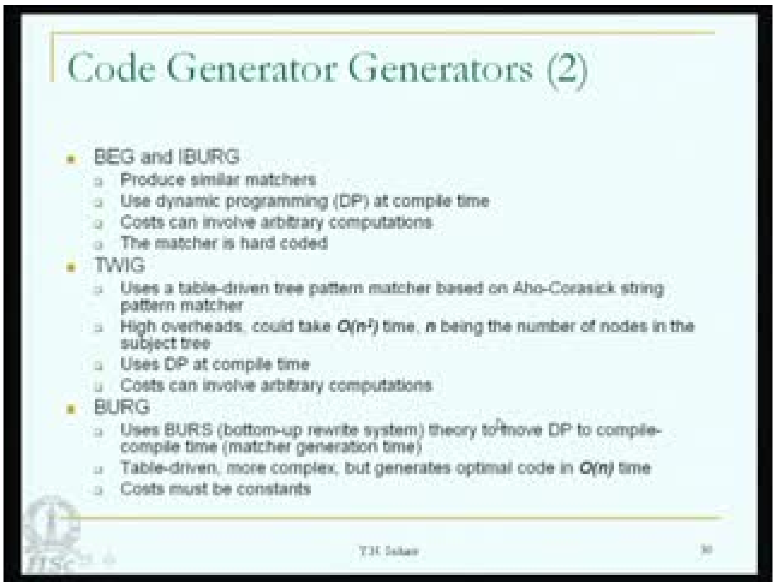
- Based on tree pattern matching and dynamic programming
- Accept tree patterns, associated costs, and semantic actions (for register allocation and object code emission)
- Produce tree matchers that produce a cover of minimum cost
- Make two passes
  - First pass is a bottom-up pass and finds a set of patterns that cover the tree with minimum cost
  - Second pass executes the semantic actions associated with the minimum cost patterns at the nodes they matched
- BEG, Twig, BURG, and IBURG are such CGGs

Y.N. Srikant

Now, let us see Code Generator Generators, which are based on tree pattern matching and dynamic programming. So, these are based on tree pattern matching and dynamic programming. They accept tree patterns, associated costs, and semantic actions just like yak for register allocation and object code emission. So, we already saw object code emission and semantic action; for each pattern, there was an action. A similar action is possible for register allocation also. These are all written in yak style. These CGGs produce tree pattern matchers that produce a cover of minimum cost; cover is covering the tree with instructions and finally, generating code. So, cover is the cover of instructions and patterns.

The code generator makes two passes. First pass is a bottom-up pass and finds a set of patterns that cover the tree with minimum cost. So, this is very obvious. We must find the alternatives, which give us the best cost and then at the root, there must be some pattern, which matches the minimum cost. That minimum cost leads to other costs downwards, patterns are matched, and thereby you get a cover of minimum cost. Second pass executes the semantic actions associated with the minimum cost patterns at the nodes they matched. So, these actions will do register allocation and they also do code generation. There are many tools of this kind: BEG, Twig, BURG, and IBURG are such Code Generator Generators. In our examples, we will use IBURG.

(Refer Slide Time: 38:12)



Code Generator Generators (2)

- BEG and IBURG
  - Produce similar matchers
  - Use dynamic programming (DP) at compile time
  - Costs can involve arbitrary computations
  - The matcher is hard coded
- TWIG
  - Uses a table-driven tree pattern matcher based on Aho-Corasick string pattern matcher
  - High overheads, could take *$O(n^2)$* time, *n* being the number of nodes in the subject tree
  - Uses DP at compile time
  - Costs can involve arbitrary computations
- BURG
  - Uses BURS (bottom-up rewrite system) theory to move DP to compile-compile time (matcher generation time)
  - Table-driven, more complex, but generates optimal code in *$O(n)$* time
  - Costs must be constants

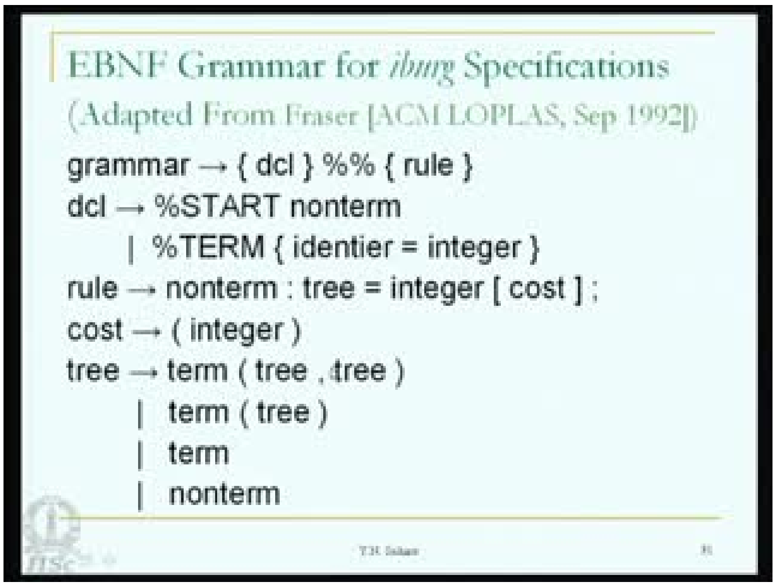
Y.N. Srikant 30

Let us look at some specialties of each of these CGGs. BEG and IBURG both produce similar matchers. They are actually hard coded matchers. In other words, the generators produce actual C code; it is not table-driven, but it is actually hard coded matcher. They use dynamic programming at compile time. As I said, to choose various alternatives, it uses dynamic programming. Costs can involve arbitrary computations because we permit ordinary C code to be associated with the rule numbers.

TWIG is a top-down table-driven tree pattern matching machine. It uses the Aho-Corasick string pattern matcher. String pattern matching is used to do tree pattern matching here. So, this is a fairly nice elegant example, but we are not going to cover it here. This has very high overheads and it could take up to O n square amount of time; n being the number of nodes in the subject tree. It uses dynamic programming at compile time like IBURG, but costs can involve arbitrary computations like here (Refer Slide Time: 39:42), but it is not hard coded, it is table driven.

BURG is special. This is (Refer Slide Time: 39:48) top-down tree pattern matching, whereas BURG uses bottom-up tree rewriting system. It uses dynamic programming at compile-compile time; that is, when the automatic generation of the pattern matcher happens, at that time itself you compute all the costs. This is also table-driven, much more complex than IBURG, but generates optimal code in just O n time. So, these require (Refer Slide Time: 40:17) more time, whereas this requires much lesser time. So, IBURG can also take O n square time. Costs must be constants. This is a problem here. You cannot have arbitrary computations, you cannot call routines to do register allocation, cost computation on the fly, etcetera. So, the flexibility is little less, but it is more efficient time-wise.

(Refer Slide Time: 40:40)



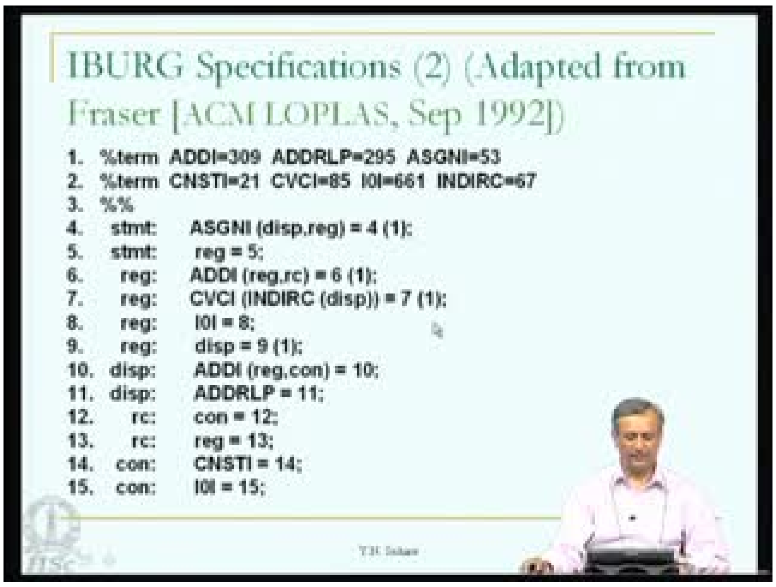
EBNF Grammar for *iburg* Specifications (Adapted From Fraser [ACM LOPLAS, Sep 1992])

```
grammar → { dcl } %% { rule }
dcl → %START nonterm
    | %TERM { identier = integer }
rule → nonterm : tree = integer [ cost ] ;
cost → ( integer )
tree → term ( tree , tree )
     | term ( tree )
     | term
     | nonterm
```

Y.N. Srikant

Let us look at the grammar for specifying IBURG specifications. This is the specification of the tree grammar itself; how are tree grammar specified. Maybe we can look at an example and come back to this.

(Refer Slide Time: 40:57)



IBURG Specifications (2) (Adapted from Fraser [ACM LOPLAS, Sep 1992])

```
1.  %term  ADDI=309  ADDRLP=295  ASGNI=53
2.  %term  CNSTI=21  CVCI=85  I0I=661  INDIRC=67
3.  %%
4.    stmt:    ASGNI (disp,reg) = 4 (1);
5.    stmt:     reg = 5;
6.     reg:    ADDI (reg,rc) = 6 (1);
7.     reg:    CVCI (INDIRC (disp)) = 7 (1);
8.     reg:     I0I = 8;
9.     reg:     disp = 9 (1);
10.   disp:     ADDI (reg,con) = 10;
11.   disp:     ADDRLP = 11;
12.     rc:     con = 12;
13.     rc:     reg = 13;
14.    con:     CNSTI = 14;
15.    con:     I0I = 15;
```

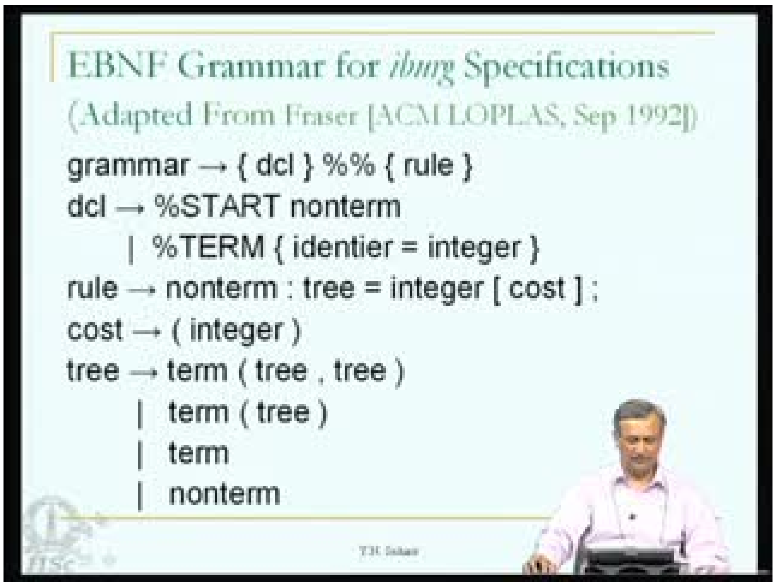
Y.N. Srikant

Here is a specification, which you can see for a very simple set of instructions. There are terminal symbols: ADDI, ADDRLP, ASGNI, CNSTI, CVCI, IOI, and INDIRC. Each of these has been given a code just like the tokens. These all are terminal symbols; tokens are given code in yak. Then, here are the rules; are the tree patterns.

First one says statement is ASGNI disp comma reg. So, here is an ASGNI node and then 2: left subtree is disp and the right subtree is reg. So, that will be the tree corresponding to statement and this has given a rule number of 4. So, this 4 and this 4 (Refer Slide Time: 41:54) are identical, but this is only coincidental. You could give any number here; it could be 104 also. The reason for this number is – this is used in order to call the appropriate semantic action once the matching process is over. The cost when this particular rule matches is indicated in parenthesis here (Refer Slide Time: 42:16). So, the cost is 1 here.

A statement going to reg, rather reg going to statement; the number is 5 and the cost is 0. No cost is mentioned. So, it is 0. reg colon ADDI reg comma rc has a cost 1 and its number is 6. Similarly, all these (Refer Slide Time: 42:39). So, there may be many of these nonterminals; these are all terminal symbols and these lower cases are all nonterminal symbols. There may be many productions corresponding to each nonterminal statement as to: reg has 4, disp has 2, rc has 2, con has 2. Some of these have costs attached (Refer Slide Time: 43:02) to them like this and some others do not have.

(Refer Slide Time: 43:09)



EBNF Grammar for *iburg* Specifications

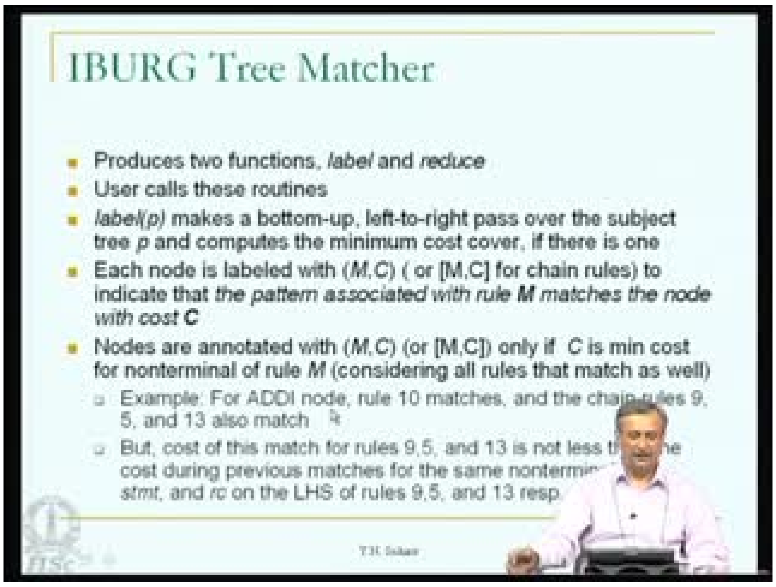
(Adapted From Fraser [ACM LOPLAS, Sep 1992])

```
grammar → { dcl } %% { rule }
dcl → %START nonterm
    | %TERM { identier = integer }
rule → nonterm : tree = integer [ cost ] ;
cost → ( integer )
tree → term ( tree , tree )
     | term ( tree )
     | term
     | nonterm
```

T.H. Srikant

Let us see this particular grammar; I have already explained it - every declaration has a START nonterminal and declares a set of terminals, which we already saw. Each rule has nonterminal colon tree, then there is a rule number, and then a cost. So, that is what it is (Refer Slide Time: 43:26).

(Refer Slide Time: 43:28)



IBURG Tree Matcher

- Produces two functions, *label* and *reduce*
- User calls these routines
- *label(p)* makes a bottom-up, left-to-right pass over the subject tree *p* and computes the minimum cost cover, if there is one
- Each node is labeled with (*M*,*C*) (or [M,C] for chain rules) to indicate that *the pattern associated with rule **M** matches the node with cost* **C**
- Nodes are annotated with (*M*,*C*) (or [M,C]) only if *C* is min cost for nonterminal of rule *M* (considering all rules that match as well)
  - Example: For ADDI node, rule 10 matches, and the chain rules 9, 5, and 13 also match
  - But, cost of this match for rules 9,5, and 13 is not less t[illegible]e cost during previous matches for the same nontermi[illegible] *stmt*, and *rc* on the LHS of rules 9,5, and 13 resp.

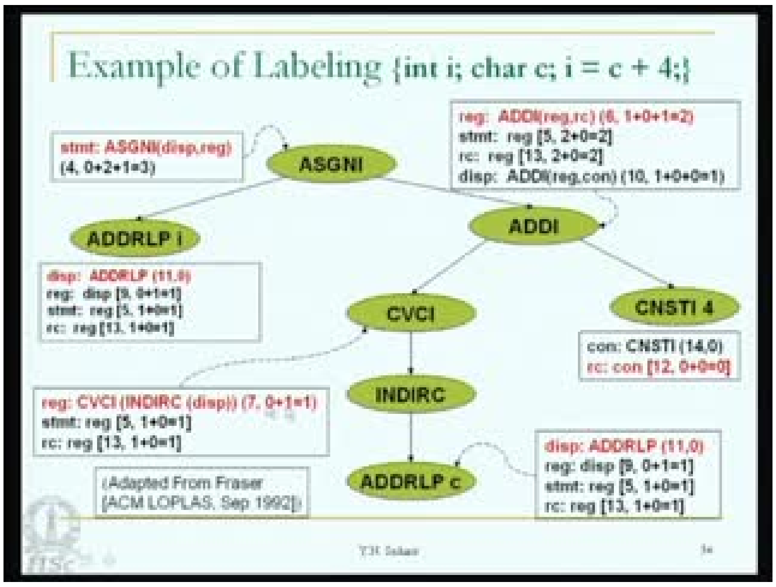
T.H. Srikant

How does the IBURG tree matcher work? It produces two functions called label and reduce. The user is responsible for calling these routines. What does label do? label p takes p, which is the root of that particular tree. It makes a bottom-up, left-to-right pass over the subject tree headed by p, and then computes the minimum cost cover, if there is one. We will see how this is done with an example.

Each node is labeled with (M,C) or [M,C] for chain rules; chain rules are unit productions, for example (Refer Slide Time: 44:16), statement colon reg, statement colon IOI, reg colon disp; these are all unit productions or chain rules. So, each node is labeled with a cost pair (M,C) to indicate that tree pattern associated with rule M matches the node with cost C. Here (Refer Slide Time: 44:38) is the rule number and here is the cost; that is appropriate for this particular node.

The nodes are annotated with (M,C) or [M,C] only if C is minimum cost for nonterminal of rule M. So, you must consider all the rules that match, take the nonterminals on the left side and whenever the nonterminals are the same, take the rule with the minimum cost that matches here and annotate that here (Refer Slide Time: 45:11).

(Refer Slide Time: 45:16)



I will give you an example. Let us see how this works. Here is a small tree. The labeling starts in a bottom-up fashion. Let us take this particular node it says ADDRLP I; it says the rule, which matches is disp going to ADDRLP. So, let us make sure that such a thing exists here (Refer Slide Time: 45:40); disp going to ADDRLP is here; that is, rule number 11; it has a cost 0. So, rule number 11 and its cost is 0. It is not a unit production or chain rule. A chain rule involves to nonterminals only, but once disp colon ADDRLP matches, all the rules with only disp on the right side will match. So, the only rule that is applicable here is reg colon disp; this is rule number 9. So, that must be added to this particular set of matches. This is rule number 9, the cost of matching disp colon ADDRLP is 0, and the cost of matching reg colon disp is 1. Let us make sure of that first. reg colon disp is 1; see (Refer Slide Time: 46:37). The cost is here; it is 1. So, 0 plus 1; that is, 1.

Now, you will look at all those productions with the reg on the right hand side. There is one with statement colon reg; the rule number is 5; its cost is 1 plus the cost for this particular (Refer Slide Time: 47:00) statement colon reg, which is 0. Let us make sure of that; that is, statement colon reg (Refer Slide Time: 47:14). This has a cost of 0. Now, look at another alternative with reg on the right side; this is rc colon reg. Similarly, this rule is number 13, the cost of this disp (Refer Slide Time: 47:32) would be 1; that is, this cost is 1 here and then rc colon reg itself has a cost of 0. So, the total cost is 1. If we do one of these reductions (Refer Slide Time: 47:50), we may be able to do this reduction and then either this reduction or this reduction; one of these two. So, these are all the matches, which happen at this particular ADDRLP.

Similarly, let us look at this now (Refer Slide Time: 48:05); CVCI, or rather let us look at this (Refer Slide Time: 48:08) constant and then come back to this, and finally, this. Const – there only there is only one. This rc; con is either CNSTI and then con on the right side; that is, rc colon con. con const is 14 comma 0; cost is 0 and rule number is 14. We are using a chain production with con on the right side rc colon con; that is, rule number is 12 and costs are both 0. So, the total cost is 0.

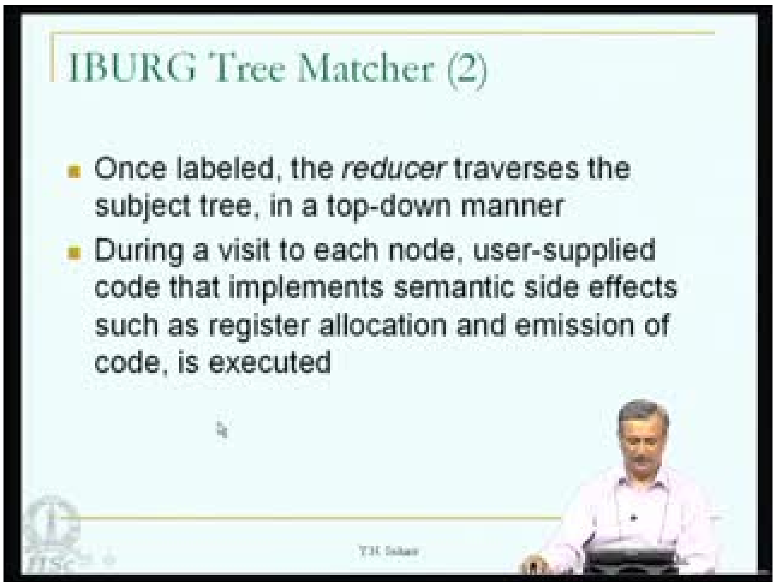
When you look at ADDRLP c (Refer Slide Time: 48:39) here, this is exactly the same as this. So, there is no difference at all. So, let us skip discussion of this. Look at this – CVCI (Refer Slide Time: 48:46). reg colon CVCI INDIRC disp; this is the entire match. There is no match for this; there is no rule with INDIRC on the right hand side as a single nonterminal, but there is one with CVCI INDIRC and disp. That means, ADDRLP has to be in disp. If we do that, this is rule number 7 (Refer Slide Time: 49:07), the cost of this disp is 0, and then the cost of this reg would be 1. So, this entire cost is 1. Then, there are similar unit productions or chain rules, which happen and we have noted them here (Refer Slide Time: 49:22).

Finally, we come to ADDI and then ASGNI. In the case of ADDI, there is an ADDI on the right hand side with a reg and rc. So, this must be (Refer Slide Time: 49:34) reg and this must be in rc. The corresponding cost is here; 1 and the corresponding cost here; that is, 0. You have rule number 6 for this match, then the 1 plus 0 plus its own cost; this cost is 1; that is 2. Then, there are other alternatives, which follow here; reg on the right side (Refer Slide Time: 49:57), reg on the right side, and then the other alternative with ADDI is reg comma con; the rule number is 10. That means, this left side (Refer Slide Time: 50:09) must be in reg and the right side must be in con. So, the cost of that would be – this is 1, this rule matches here; that is 0, and finally, this cost is also 0. So, the total cost is 1.

When you look at ASGNI, the only rule, which matches is ASGNI disp comma reg. That forces this entire thing (Refer Slide Time: 50:31) to be in disp and this entire thing to be in reg. So, whatever is indicated in red are the final matched rules, which give you a cover for this entire tree. So, this here (Refer Slide Time: 50:45), this here, this here, this here, this here, and this here. With that, the total cost would be 0 for this (Refer Slide Time: 50:55), then 2 for this, and finally, 1 for this itself. So, total cost would be 3. This is how the cost is computed for a small tree.

Once we know this particular (Refer Slide Time: 51:14); this is the cover for this entire tree, it is easy to… Once we know the rule number, we can execute the action corresponding to each of these rule numbers, which have matched and then we actually generate code. So, rule for here and so on and so forth. So, execute semantic actions and generate code.

(Refer Slide Time: 51:37)



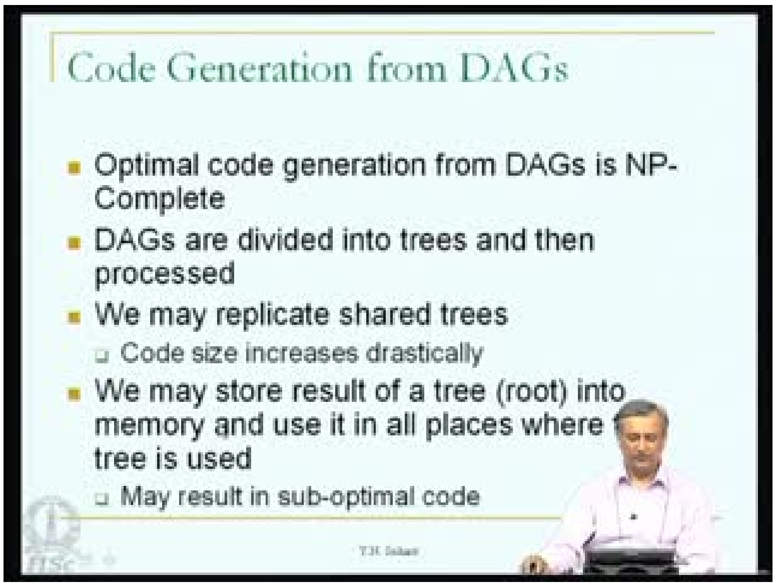
IBURG Tree Matcher (2)

- Once labeled, the *reducer* traverses the subject tree, in a top-down manner
- During a visit to each node, user-supplied code that implements semantic side effects such as register allocation and emission of code, is executed

Y.N. Srikant

This is the summery of how the IBURG tree pattern matcher works. Once labeled, the reducer traverses the subject tree, in a top-down manner. During a visit to each node, user-supplied code that implements semantic side effects such as register allocation and emission of code, is executed and finally, the machine code gets generated.

(Refer Slide Time: 52:00)



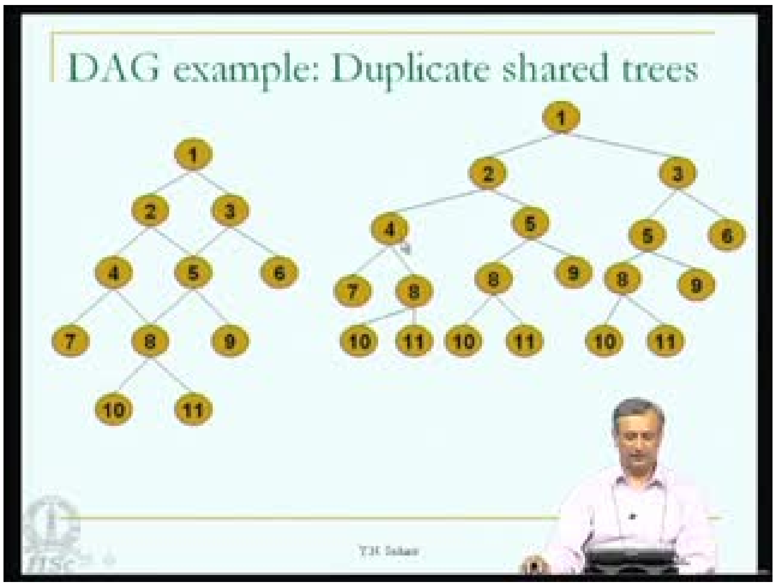
Code Generation from DAGs

- Optimal code generation from DAGs is NP-Complete
- DAGs are divided into trees and then processed
- We may replicate shared trees
  - Code size increases drastically
- We may store result of a tree (root) into memory and use it in all places where the tree is used
  - May result in sub-optimal code

Y.N. Srikant

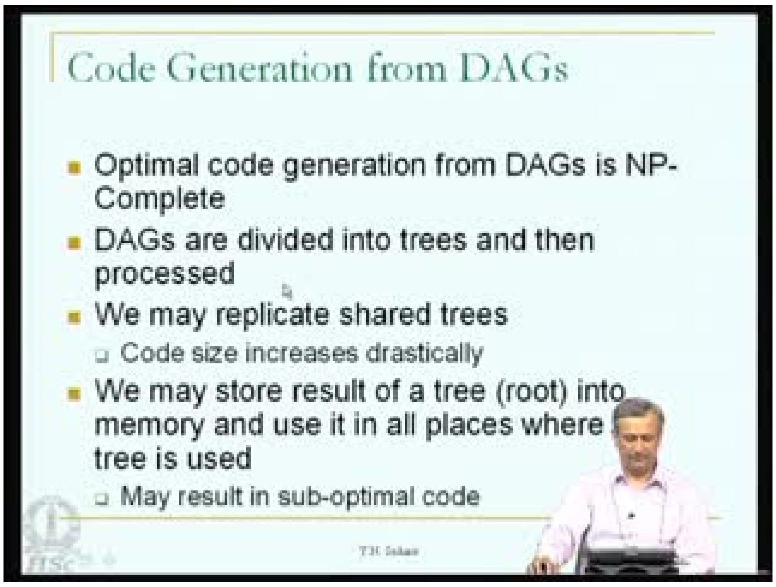
That was a brief discussion of how code is generated using tree pattern matching. Now, let us do a little bit of discussion on how to generate code from Directed Acyclic Graphs.

(Refer Slide Time: 52:21)



We have; let us say DAGs. DAGs look like this. These are all DAGs. 8 is shared by 4 and 5, whereas in a tree, such sharing is not permitted. 5 is shared between 2 and 3 and so on and so forth.

(Refer Slide Time: 52:32)

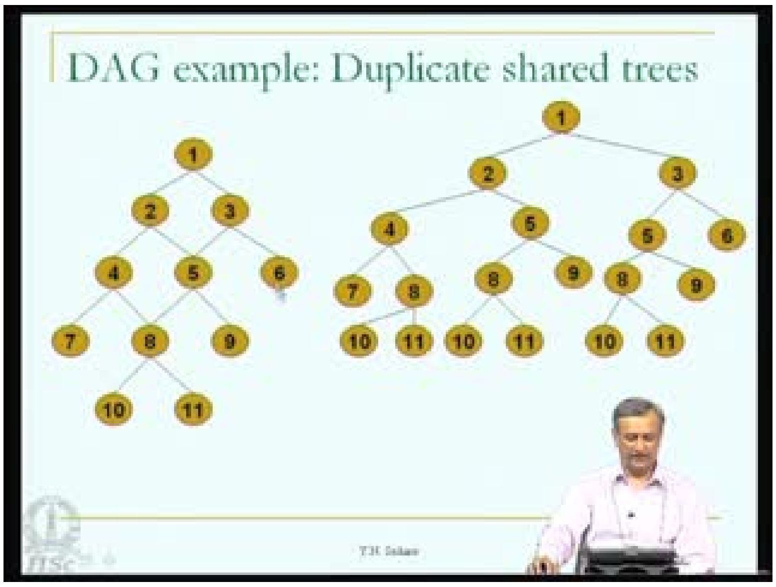


Code Generation from DAGs

- Optimal code generation from DAGs is NP-Complete
- DAGs are divided into trees and then processed
- We may replicate shared trees
  - Code size increases drastically
- We may store result of a tree (root) into memory and use it in all places where tree is used
  - May result in sub-optimal code

This general problem of code generation from DAGs is an NP-Complete problem; not much can be done about it. How does it become NP-complete? This will become clearer as we go on. So, we will see little later. Let us assume that it is NP-complete. In other words, when this code generation problem is NP-complete, the implication is – unless you look at all possible code generation orders on this DAG, which is made up of several trees, you really cannot say which takes the minimum amount of time. So, heuristics are the only way to handle this situation. There are many heuristics. Let us look at two of them – DAGs are divided into trees and then processed.

(Refer Slide Time: 53:29)

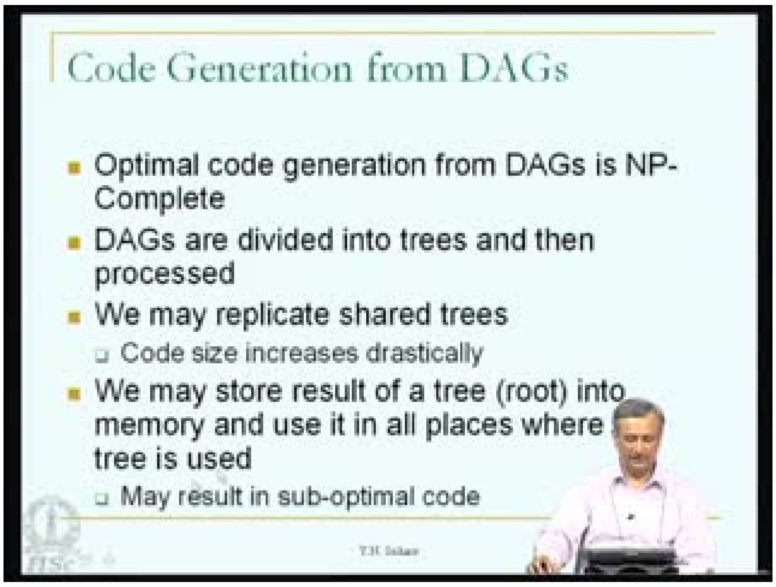


DAG example: Duplicate shared trees

Here for example, this is a DAG, which has many shared trees. What we do is – whenever a node is shared between the parents, we create copies/duplicate. For example, 4 and 5 share 8. So, 4 has its own copy of 8 followed by its subtree; 5 has its own copy of 8 followed by its subtree.

5 itself is shared between 2 and 3 (Refer Slide Time: 54:06. So, 2 would have its own copy of 5 and its associated subtrees; 3 would have its own copy of 5 and the associated subtrees, but we already replicated 8 between 4 and 5. So, there were two copies: one for 4, one for 5. Now, 5 itself is being made into a copy. So, this entire tree is also copied. This 2 (Refer Slide Time: 54:32) has a copy of 5 and 8, 10, and 11. This 3 (Refer Slide Time: 54:36) also have a copy of 5 and 8, 10, and 11. So, this small subtree 8, 10, 11 has been replicated three times. Probably now, you realize the difficulty.

(Refer Slide Time: 54:48)



Code Generation from DAGs

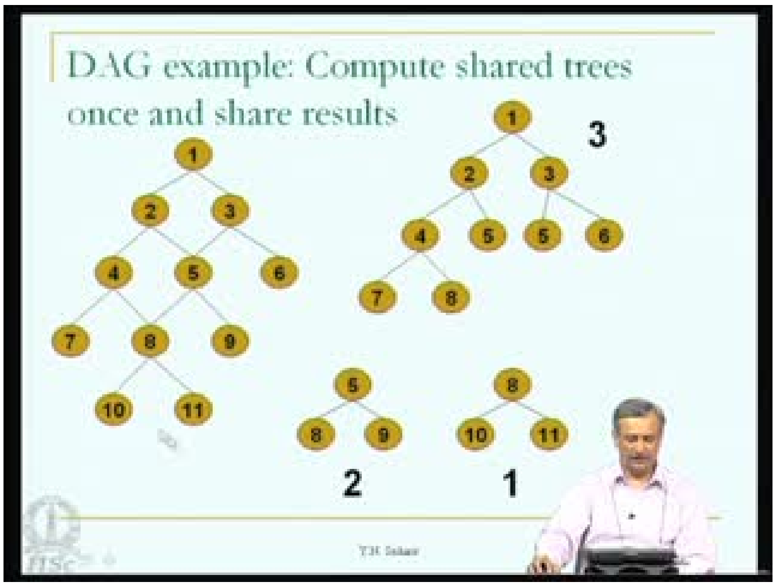
- Optimal code generation from DAGs is NP-Complete
- DAGs are divided into trees and then processed
- We may replicate shared trees
  - Code size increases drastically
- We may store result of a tree (root) into memory and use it in all places where tree is used
  - May result in sub-optimal code

Y.N. Srikant

We may replicate shared trees, but then the code size increases drastically. So, once we have this tree (Refer Slide Time: 54:58), you can apply dynamic programming, or Sethi-Ullman algorithm, or tree pattern matching in order to generate very good code on this particular tree.

What is the other strategy? One may store the result of a tree; that is, the root into memory and then use it in all places where the tree is used. So, shared trees are cut off, prune from the mother tree, and then it is stored into memory and finally, that result is used. Let us see how it works.

(Refer Slide Time: 55:31)



DAG example: Compute shared trees once and share results

This is our original tree. 8 10 11 is a shared tree. So, 8 10 11 is separated into a small tree. Then, 5, 8, and 9 is another shared tree. So, 5, 8 and 9 is separated into another small tree. Remember that 8 10 11 has already been removed. So, what we would have had is just 5, 8, and 9. Now, the rest of it would be just a tree because 1 2 3, then you have 4 and 5. 5 8 9 is separate tree. So, there is nothing here. Then 8 10 11 is separate tree; so, 4 7 8 is here.

Now, what we really do is – evaluate this particular tree (Refer Slide Time: 56:19) and generate code for it, generate an instruction to store this entire result into memory. So, that is what this 8 would be; a memory location. The same is done for 5 8 9. We have done this first. So, 8 is now a leaf node, which is actually a memory location. Then, 5 8 9 code generation is performed. An instruction is generated to store 5 into memory. Then, this would also become a (Refer Slide Time: 56:50) memory location. Then, we can generate code for this particular tree because these are now just memory locations, there is no problem, we can load them whenever we require, and evaluate the tree. Code generation for this subtree happens.

The problem is the order in which all these (Refer Slide Time: 57:11) must be done; there are many orders and unless we test each one of these orders, we cannot say that the code is optimal. This is what makes the code generation problem from DAGs NP-complete. So, we will stop at this point and in the next lecture, we will look at Peephole

Optimizations.

Thank you