
Code Generation – Part 1

Y. N. Srikant

Computer Science and Automation

Indian Institute of Science

Bangalore 560 012

NPTEL Course on Compiler Design



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 4(b),4(c),5, and 6 will be covered in part 2 of the lecture

Code Generation – Main Issues (1)

- Transformation:
 - Intermediate code \rightarrow m/c code (binary or assembly)
 - We assume quadruples and CFG to be available
- Which instructions to generate?
 - For the quadruple $A = A+1$, we may generate
 - Inc A or
 - Load A, R1
Add #1, R1
Store R1, A
 - One sequence is faster than the other (cost implication)

Code Generation – Main Issues (2)

- In which order?
 - Some orders may use fewer registers and/or may be faster
- Which registers to use?
 - Optimal assignment of registers to variables is difficult to achieve
- Optimize for memory, time or power?
- Is the code generator easily retargetable to other machines?
 - Can the code generator be produced automatically from specifications of the machine?

Samples of Generated Code

■ $B = A[i]$

```
Load  i, R1 //  $R1 = i$   
Mult  R1, 4, R1 //  $R1 = R1 * 4$   
// each element of array  
// A is 4 bytes long  
Load  A(R1), R2 //  $R2 = (A + R1)$   
Store R2, B //  $B = R2$ 
```

■ $X[j] = Y$

```
Load  Y, R1 //  $R1 = Y$   
Load  j, R2 //  $R2 = j$   
Mult  R2, 4, R2 //  $R2 = R2 * 4$   
Store R1, X(R2) //  $X(R2) = R1$ 
```

■ $X = *p$

```
Load  p, R1  
Load  0(R1), R2  
Store R2, X
```

■ $*q = Y$

```
Load  Y, R1  
Load  q, R2  
Store R1, 0(R2)
```

■ if $X < Y$ goto L

```
Load  X, R1  
Load  Y, R2  
Cmp   R1, R2  
Bltz  L
```

Samples of Generated Code – Static Allocation (no JSR instruction)

Three Adress Code

```
// Code for function F1  
action code seg 1  
    call F2  
action code seg 2  
    Halt
```

```
// Code for function F2  
action code seg 3  
    return
```

Activation Record for F1 (48 bytes)

0	return address
4	
	data array A
40	
	variable x
44	
	variable y

Activation Record for F2 (76 bytes)

0	return address
4	
	parameter 1
	data array B
72	
	variable m

Samples of Generated Code – Static Allocation (no JSR instruction)

```
// Code for function F1
200:   Action code seg 1
// Now store return address
240:   Move #264, 648
252:   Move param1, 652
256:   Jump 400 // Call F2
264:   Action code seg 2
280:   Halt

...

// Code for function F2
400:   Action code seg 3
// Now return to F1
440:   Jump @648

...
```

```
//Activation record for F1
//from 600-647
600:   //return address
604:   //space for array A
640:   //space for variable x
644:   //space for variable y
//Activation record for F2
//from 648-723
648:   //return address
652:   // parameter 1
656:   //space for array B

...

720:   //space for variable m
```

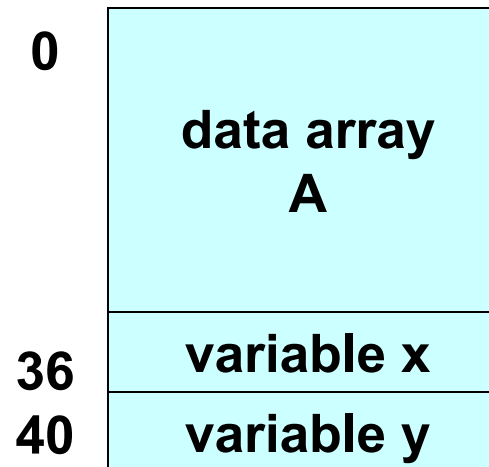
Samples of Generated Code – Static Allocation (with JSR instruction)

Three Adress Code

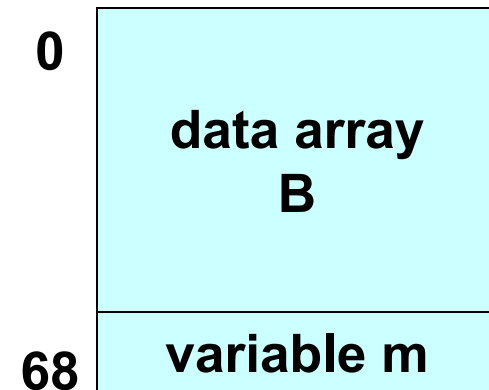
```
// Code for function F1  
action code seg 1  
call F2  
action code seg 2  
Halt
```

```
// Code for function F2  
action code seg 3  
return
```

Activation Record for F1 (44 bytes)



Activation Record for F2 (72 bytes)



Samples of Generated Code – Static Allocation (with JSR instruction)

```
// Code for function F1
200:  Action code seg 1
// Now jump to F2, return addr
// is stored on hardware stack
240:  JSR 400 // Call F2
248:  Action code seg 2
268:  Halt
...
// Code for function F2
400:  Action code seg 3
// Now return to F1 (addr 248)
440:  return
...
```

```
//Activation record for F1
//from 600-643
600:  //space for array A
636:  //space for variable x
640:  //space for variable y
//Activation record for F2
//from 644-715
644:  //space for array B
...
712:  //space for variable m
```

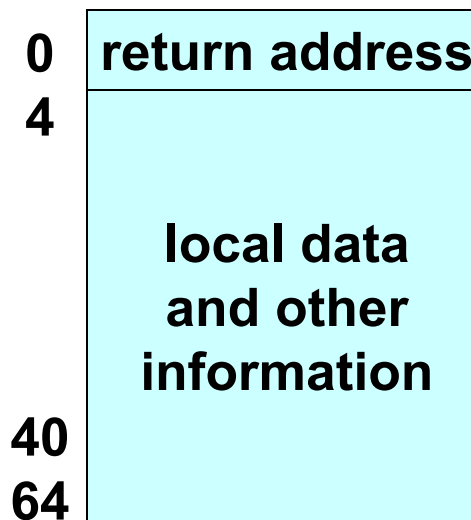
Samples of Generated Code – Dynamic Allocation (no JSR instruction)

Three Address Code

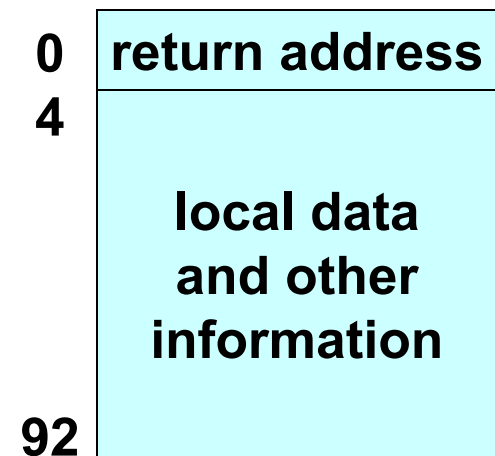
```
// Code for function F1
action code seg 1
  call F2
action code seg 2
  return
```

```
// Code for function F2
action code seg 3
  call F1
action code seg 4
  call F2
action code seg 5
  return
```

Activation Record for F1 (68 bytes)



Activation Record for F2 (96 bytes)



Samples of Generated Code – Dynamic Allocation (no JSR instruction)

```
//Initialization
```

```
100:  Move #800, SP
```

```
...
```

```
//Code for F1
```

```
200:  Action code seg 1
```

```
230:  Add #96, SP
```

```
238:  Move #254, @SP
```

```
246:  Jump 300
```

```
254:  Sub #96, SP
```

```
262:  Action code seg 2
```

```
296:  Jump @SP
```

```
//Code for F2
```

```
300:  Action code seg 3
```

```
340:  Add #68, SP
```

```
348:  Move #364, @SP
```

```
356:  Jump 200
```

```
364:  Sub #68, SP
```

```
372:  Action code seg 4
```

```
400:  Add #96, SP
```

```
408:  Move #424, @SP
```

```
416:  Jump 300
```

```
424:  Sub #96, SP
```

```
432:  Action code seg 5
```

```
480:  Jump @SP
```

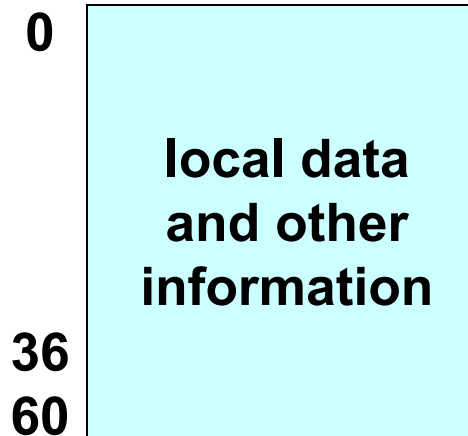
Samples of Generated Code – Dynamic Allocation (with JSR instruction)

Three Address Code

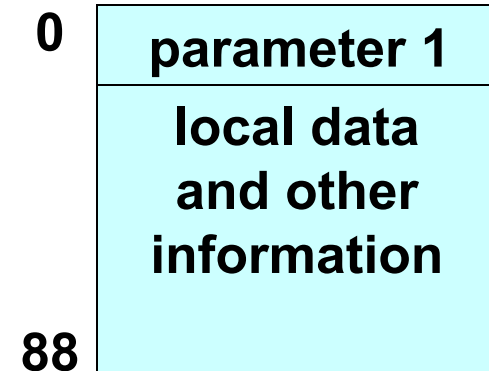
```
// Code for function F1  
action code seg 1  
  call F2  
action code seg 2  
  return
```

```
// Code for function F2  
action code seg 3  
  call F1  
action code seg 4  
  call F2  
action code seg 5  
  return
```

Activation Record for F1 (64 bytes)



Activation Record for F2 (92 bytes)



Samples of Generated Code – Dynamic Allocation (with JSR instruction)

```
//Initialization
```

```
100:  Move #800, SP
```

```
...
```

```
//Code for F1
```

```
200:  Action code seg 1
```

```
230:  Add #92, SP
```

```
238:  Move param1, @SP
```

```
242:  JSR 290
```

```
250:  Sub #92, SP
```

```
258:  Action code seg 2
```

```
286:  return
```

```
//Code for F2
```

```
290:  Action code seg 3
```

```
330:  Add #64, SP
```

```
338:  JSR 200
```

```
346:  Sub #64, SP
```

```
354:  Action code seg 4
```

```
382:  Add #92, SP
```

```
390:  JSR 290
```

```
398:  Sub #92, SP
```

```
406:  Action code seg 5
```

```
454:  return
```

A Simple Code Generator – Scheme A

- Treat each quadruple as a ‘macro’
 - Example: The quad $A := B + C$ will result in
 - Load B, R1 OR Load B, R1**
 - Load C, R2**
 - Add R2, R1 Add C, R1**
 - Store R1, A Store R1, A**
 - Results in inefficient code
 - Repeated load/store of registers
 - Very simple to implement

A Simple Code Generator – Scheme B

- Track values in registers and reuse them
 - If any operand is already in a register, take advantage of it
 - Register descriptors
 - Tracks <register, variable name> pairs
 - A single register can contain values of multiple names, if they are all copies
 - Address descriptors
 - Tracks <variable name, locations> pairs
 - A single name may have its value in multiple locations, such as, memory, register, and stack



A Simple Code Generator – Scheme B

- Leave computed result in a register as long as possible
- Store only at the end of a basic block or when that register is needed for another computation
 - On exit from a basic block, store only **live variables** which are not in their memory locations already (use address descriptors to determine the latter)
 - If liveness information is not known, assume that all variables are live at all times

Example

- $A := B + C$
 - If B and C are in registers R1 and R2, then generate
 - $ADD\ R2, R1$ (cost = 1, result in R1)
 - legal only if B is *not live* after the statement
 - If R1 contains B, but C is in memory
 - $ADD\ C, R1$ (cost = 2, result in R1) **or**
 - $LOAD\ C, R2$
 $ADD\ R2, R1$ (cost = 3, result in R1)
 - legal only if B is *not live* after the statement
 - attractive if the value of C is subsequently used (it can be taken from R2)

Next Use Information

- Next use info is used in code generation and register allocation
- Next use of A in quad i is j if

Quad i : $A = \dots$ (assignment to A)

↳ (control flows from i to j with no assignments to A)

Quad j : $\quad = A \text{ op } B$ (usage of A)

- In computing next use, we assume that on exit from the basic block
 - All temporaries are considered non-live
 - All programmer defined variables (and non-temps) are live
- Each procedure/function call is assumed to start a basic block
- Next use is computed on a backward scan on the quads in a basic block, starting from the end
- Next use information is stored in the symbol table



Example of computing Next Use

3	T1 := 4 * I	T1 – (nlv, lu 0, nu 5), I – (lv, lu 3, nu 10)
4	T2 := addr(A) – 4	T2 – (nlv, lu 0, nu 5)
5	T3 := T2[T1]	T3 – (nlv, lu 0, nu 8), T2 – (nlv, lu 5, nnu), T1 – (nlv, lu 5, nu 7)
6	T4 := addr(B) – 4	T4 – (nlv, lu 0, nu 7)
7	T5 := T4[T1]	T5 – (nlv, lu 0, nu 8), T4 – (nlv, lu 7, nnu), T1 – (nlv, lu 7, nnu)
8	T6 := T3 * T5	T6 – (nlv, lu 0, nu 9), T3 – (nlv, lu 8, nnu), T5 – (nlv, lu 8, nnu)
9	PROD := PROD + T6	PROD – (lv, lu 9, nnu), T6 – (nlv, lu 9, nnu)
10	I := I + 1	I – (lv, lu 10, nu 11)
11	if I ≤ 20 goto 3	I – (lv, lu 11, nnu)

Scheme B – The algorithm

- We deal with one basic block at a time
- We assume that there is no global register allocation
- For each quad $A := B \text{ op } C$ do the following
 - Find a location L to perform $B \text{ op } C$
 - Usually a register returned by $GETREG()$ (could be a mem loc)
 - Where is B ?
 - B' , found using address descriptor for B
 - Prefer register for B' , if it is available in memory and register
 - Generate $Load\ B', L$ (if B' is not in L)
 - Where is C ?
 - C' , found using address descriptor for C
 - Generate $op\ C', L$
 - Update descriptors for L and A
 - If B/C have no next uses, update descriptors to reflect this information

Function *GETREG()*

Finds L for computing $A := B \text{ op } C$

1. If B is in a register (say R), R holds no other names, and
 - **B has no next use, and B is not live after the block, then return R**
2. Failing (1), return an empty register, if available
3. Failing (2)
 - If A has a next use in the block, OR if $B \text{ op } C$ needs a register (e.g., op is an indexing operator)
 - **Use a *heuristic* to find an occupied register**
 - a register whose contents are referenced farthest in future, or
 - the number of next uses is smallest etc.
 - **Spill it by generating an instruction, $\text{MOV } R, \text{mem}$**
 - mem is the memory location for the variable in R
 - That variable is not already in mem
 - **Update *Register* and *Address* descriptors**
4. If A is not used in the block, or no suitable register can be found
 - **Return a memory location for L**

Example

T,U, and V are temporaries - **not live** at the end of the block
W is a non-temporary - **live** at the end of the block, **2 registers**

Statements	Code Generated	Register Descriptor	Address Descriptor
T := A * B	Load A, R0 Mult B, R0	R0 contains T	T in R0
U := A + C	Load A, R1 Add C, R1	R0 contains T R1 contains U	T in R0 U in R1
V := T - U	Sub R1, R0	R0 contains V R1 contains U	U in R1 V in R0
W := V * U	Mult R1, R0	R0 contains W	W in R0
	Load R0, W		W in memory (restored)

Optimal Code Generation

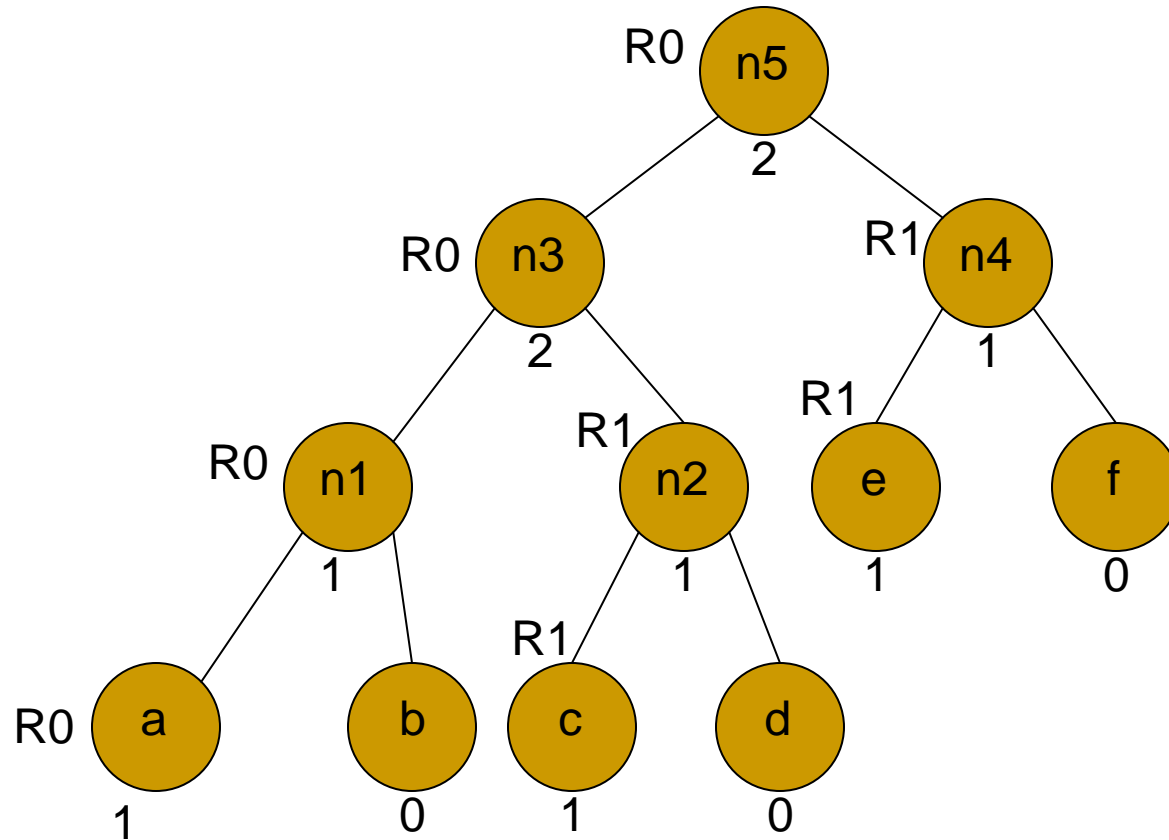
- The Sethi-Ullman Algorithm

- 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

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 (l₁, l₂), if l₁ <> l₂**
= l₁ + 1, if l₁ = l₂

Labelling - Example

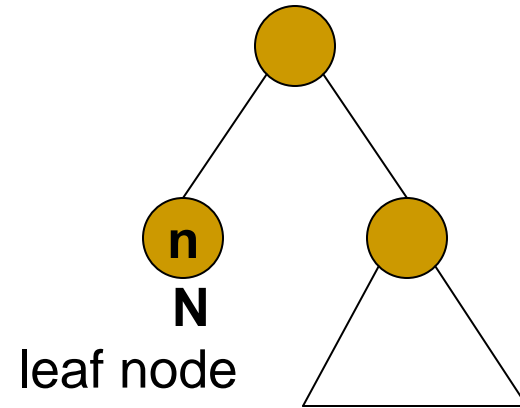


Code Generation Phase – Procedure GENCODE(n)

- RSTACK – stack of registers, $R_0, \dots, R_{(r-1)}$
- TSTACK – stack of temporaries, T_0, T_1, \dots
- A call to Gencode(n) generates code to evaluate a tree T , rooted at node n , into the register $\text{top}(\text{RSTACK})$, 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.**

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))
```



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
```

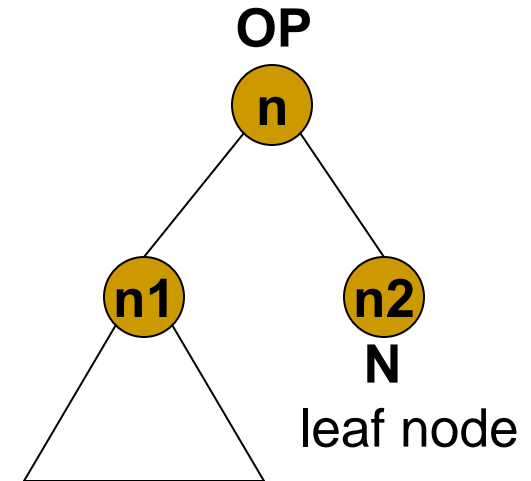
```
if label(n2) == 0 then {
```

```
    let N be the operand for n2;
```

```
    gencode(n1);
```

```
    print(OP N, top(RSTACK));
```

```
}
```



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);
```

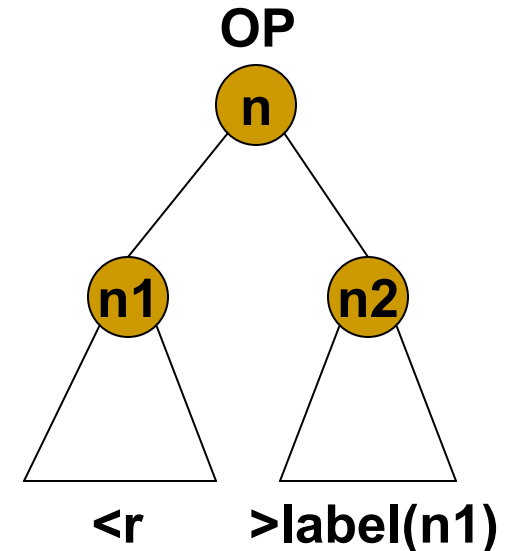
```
  /* R holds the result of n2 */
```

```
  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

The Code Generation Algorithm (4)

```
/* case 3 */
```

```
else if ((1 ≤ label(n2) ≤ label(n1))  
         and( label(n2) < r))
```

```
then {
```

```
  gencode(n1);
```

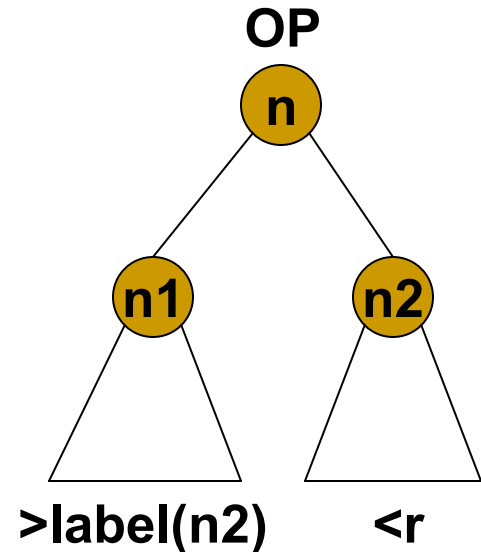
```
  R := pop(RSTACK); gencode(n2);
```

```
  /* R holds the result of n1 */
```

```
  print(OP top(RSTACK), R);
```

```
  push (RSTACK,R);
```

```
}
```



The Code Generation Algorithm (5)

/ case 4, both labels are $\geq r$ */*

else {

gencode(n2); T:= pop(TSTACK);

print(LOAD top(RSTACK), T);

gencode(n1);

print(OP T, top(RSTACK));

push(TSTACK, T);

}

}

