Machine Code Generation

Contents

- **Machine code generation – main issues**
- **Samples of generated code**
- **Two simple code generators**

- Optimal code generation
  - Sethi-Ullman algorithm
  - Dynamic programming based algorithm
  - Tree pattern matching based algorithm

Code Generation – Main Issues (1)

- **Transformation**:
  - **Intermediate code** -> m/c code (or **assembly**)
  - We assume code in IR (e.g., **quadruples** and **Control Flow Graphs)** 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

- Optimize for memory/ time ?

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
```

- if X < Y goto L

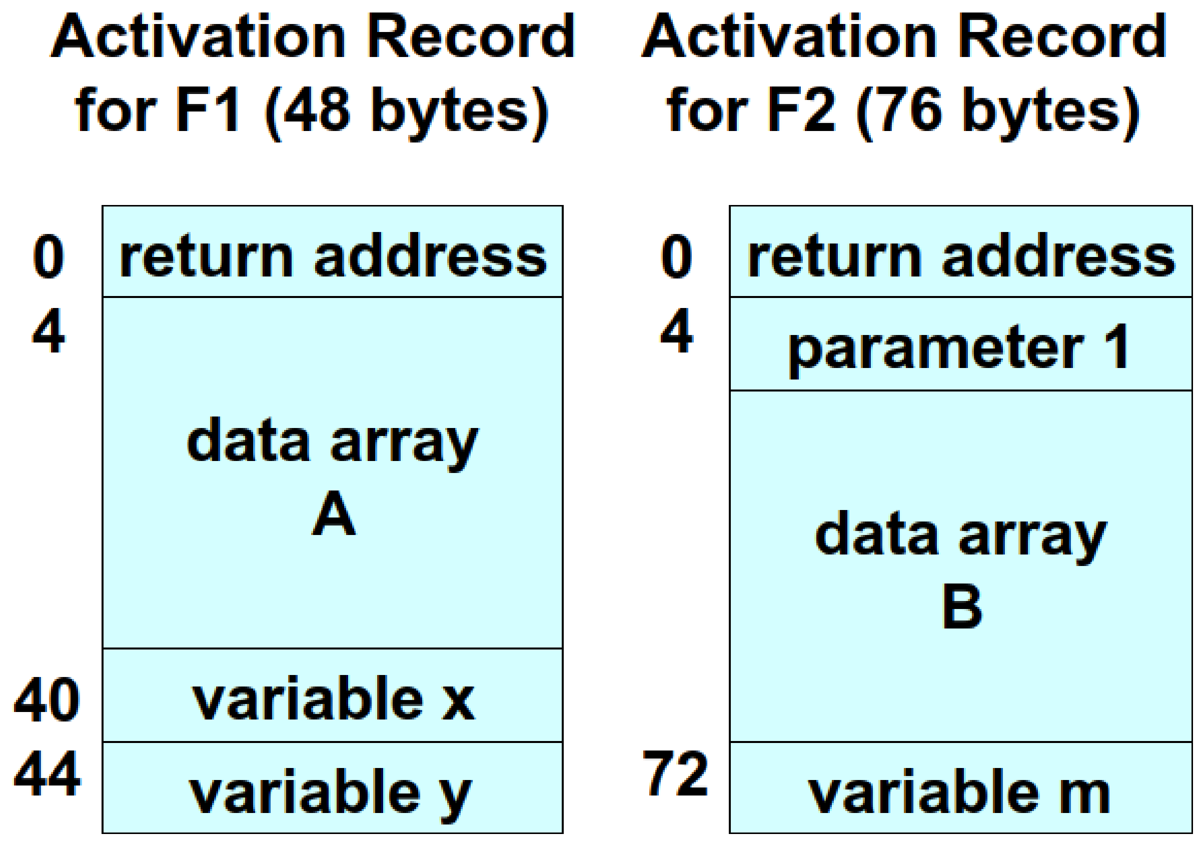
```
Load  X, R1
Load  Y, R2
Cmp   R1, R2
Bltz  L
```

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
```

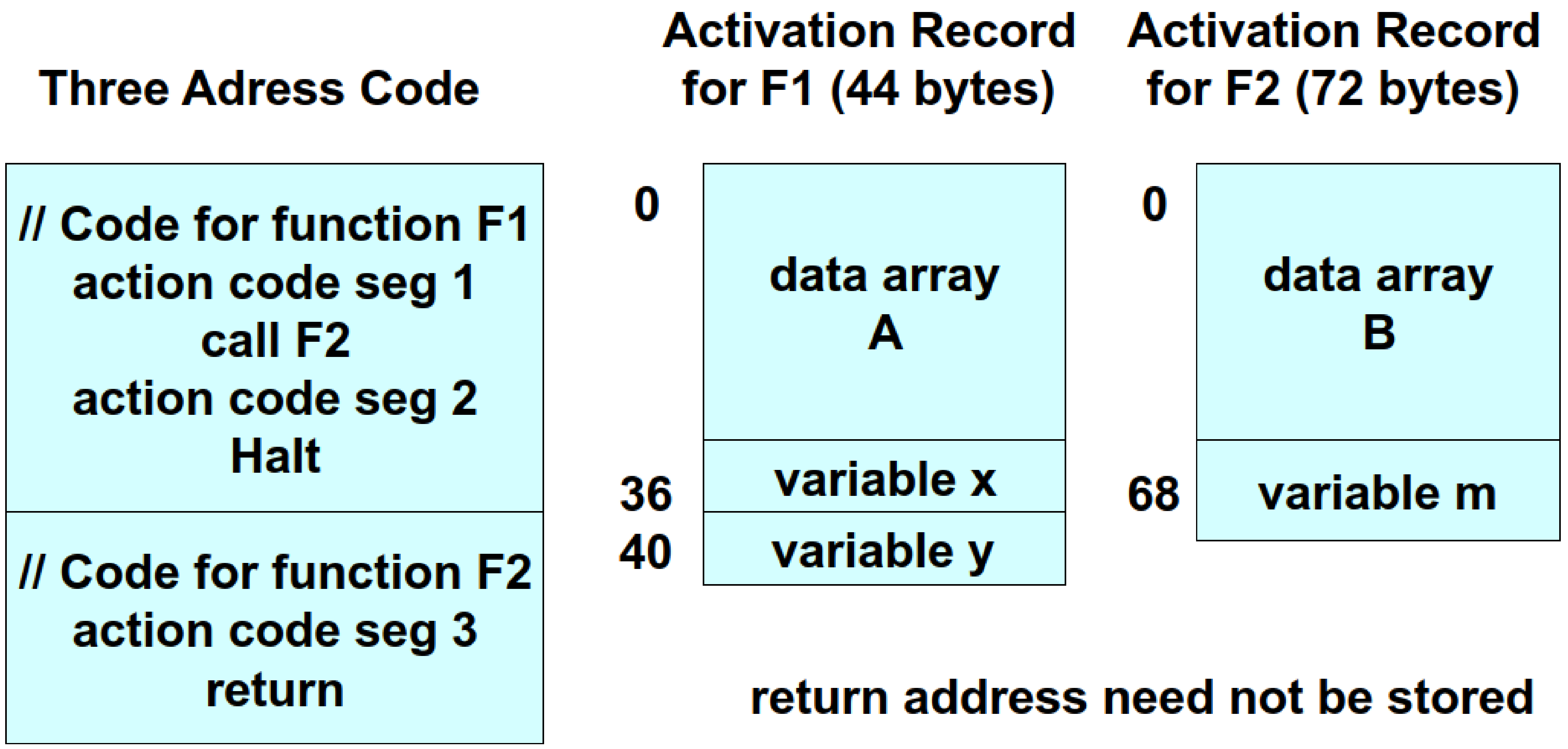


Samples of Generated Code – Static Allocation (no JSR instruction)

```
// Code for function F1
200:    Action code seg 1
// Now store return address
240:    Move #264, 648
252:    Move val1, 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)



return address need not be stored

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
```

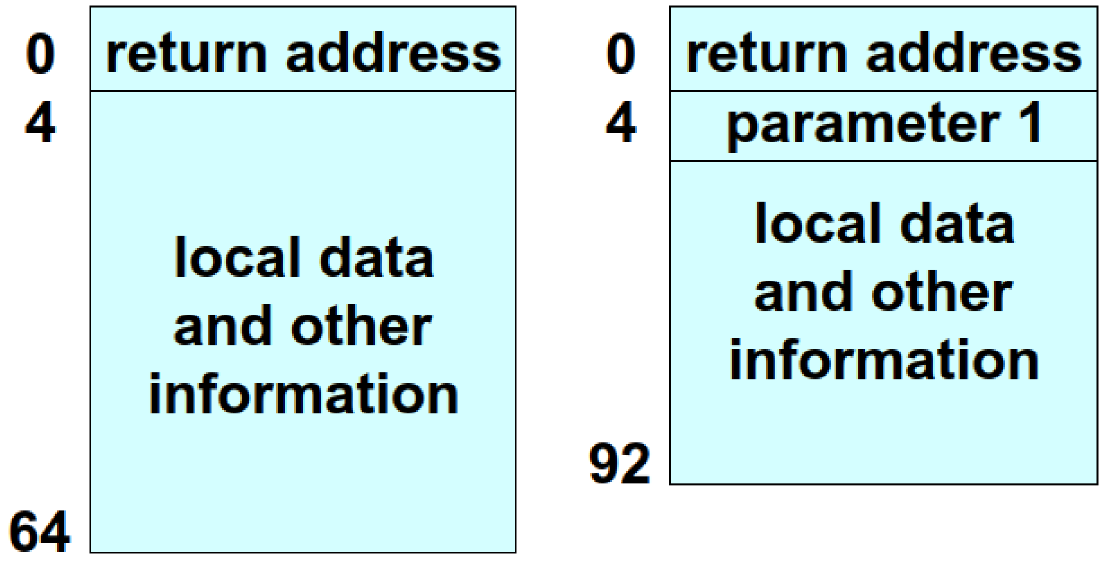
Samples of Generated Code – **Dynamic Allocation** (**no** JSR instruction)

Three Adress 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)**



0
return address
4
local data and other information
64
0
return address
4
parameter 1
local data and other information
92

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 #258, @SP
246:    Move val1, @SP+4
250:    Jump 300
258:    Sub #96, SP
266:    Action code seg 2
296:    Jump   @SP
```

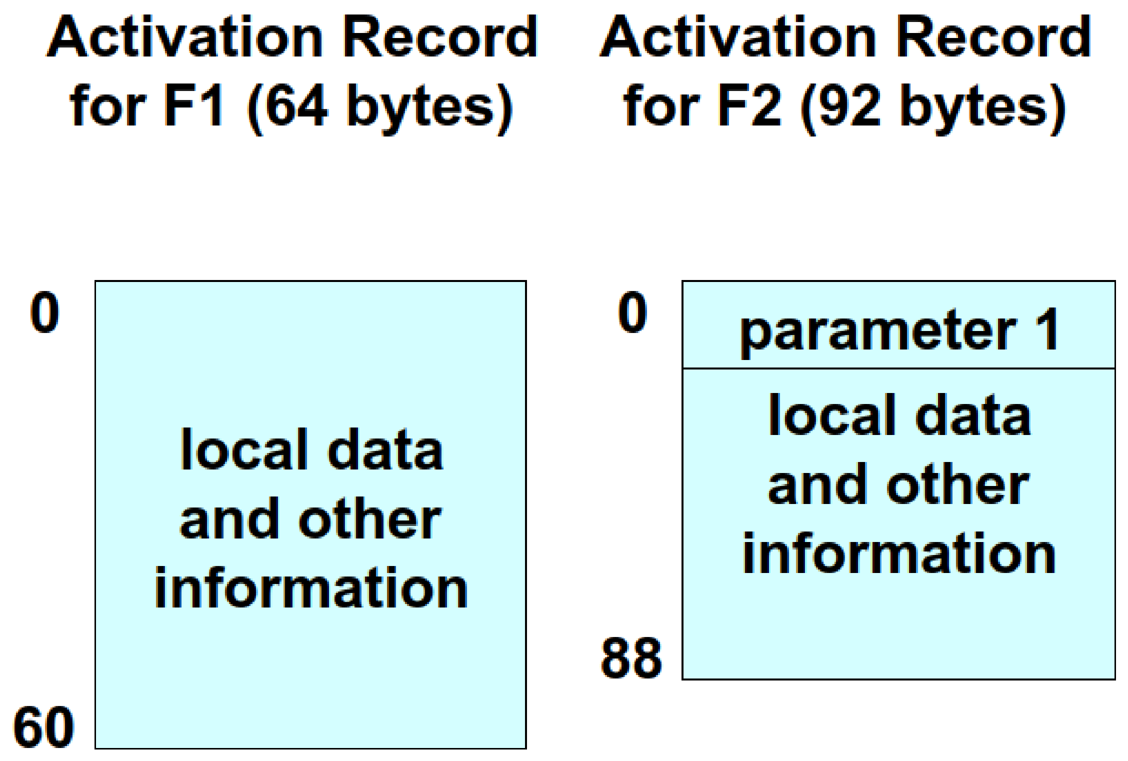
```
//Code for F2
300:    Action code seg 3
340:    Add #68, SP
348:    Move #364, @SP
356:    Jump 200
364:    Sub #68, SP
372:    Action code seg 4
400:    Add #96, SP
408:    Move #424, @SP
416:    Move val2, @SP+4
420:    Jump 300
428:    Sub #96, SP
436:    Action code seg 5
480:    Jump @SP
```

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

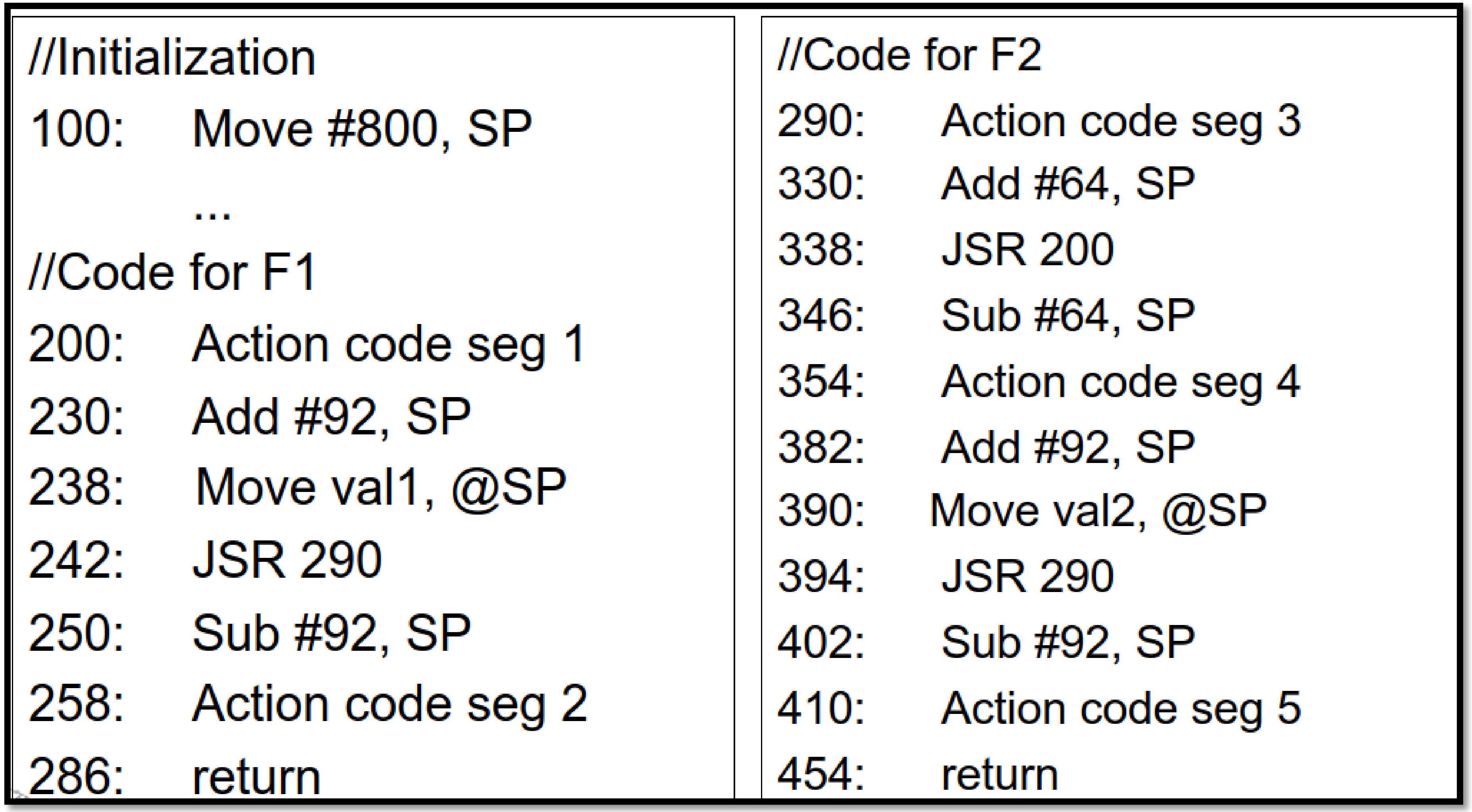
Three Adress Code

```
// 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)
0
local data and other information
60
Activation Record for F2 (92 bytes)
0
parameter 1
local data and other information
88

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 val1, @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:    Move val2, @SP
394:    JSR 290
402:    Sub #92, SP
410:    Action code seg 5
454:    return
```

Simple Code Generator – Scheme A

- Treat each quadruple as a 'macro'
  - Example: The quad *A := B + C* will result in

```
Load B, R1
Load C, R2
Add R2, R1
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
  - **Address descriptors**
    - Tracks <variable name, location> pairs
    - A single name may have its value in multiple locations, such as, memory, register, ..

A Simple Code Generator – Scheme B

- Leave computed result in a register *as long as possible*
- Store to *home* location only at **the end of a basic block** or **when that register is needed for another computation**
  - A variable is **live** at a point, if it is used later, possibly in other blocks – obtained by dataflow analysis
  - 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 = ... (assignment to A)

  $\Downarrow$ (control flows from *i* to *j* with no assignments to A)

  Quad *j* : = A 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

Example- 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)

Code generation scheme

- We deal with one basic block at a time
- For each quad A := B op C do the following
- Find a location ***L*** to perform B op C
    - Usually a ***register*** returned by GETREG() (could be a **mem loc**)
- Where is **B**?
    - found using address descriptor for **B**
    - Prefer register if it is available in memory and register
    - Generate Load B' , L (if ***B*** is not in ***L***)
- Where is **C**?
    - found using address descriptor for **C**
    - Generate op C' , L
- Update descriptors for **L, A, B, C**

Function GETREG()

- Finds ***L*** for computing A := B op C
  1. If **B** is in a register (say **R**), 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** op **C** needs a register
       - 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.
       - generate an instruction: **MOV R, 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
     1. Return a memory location

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
	Store R0, W		W in memory

Optimal Code Generation-
The Sethi-Ullman algorithm

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
  - Target machine model is simple
    - a **load** instruction,
    - a **store** instruction, and
    - binary operations involving either a register and a memory, or two registers.
  - Instructions are of the form
    - **op R,R**
    - **op M,R**
- Approach computes the left sub-tree into a register and reuses it immediately

Optimal Code Generation- The Sethi-Ullman algorithm

Two phases

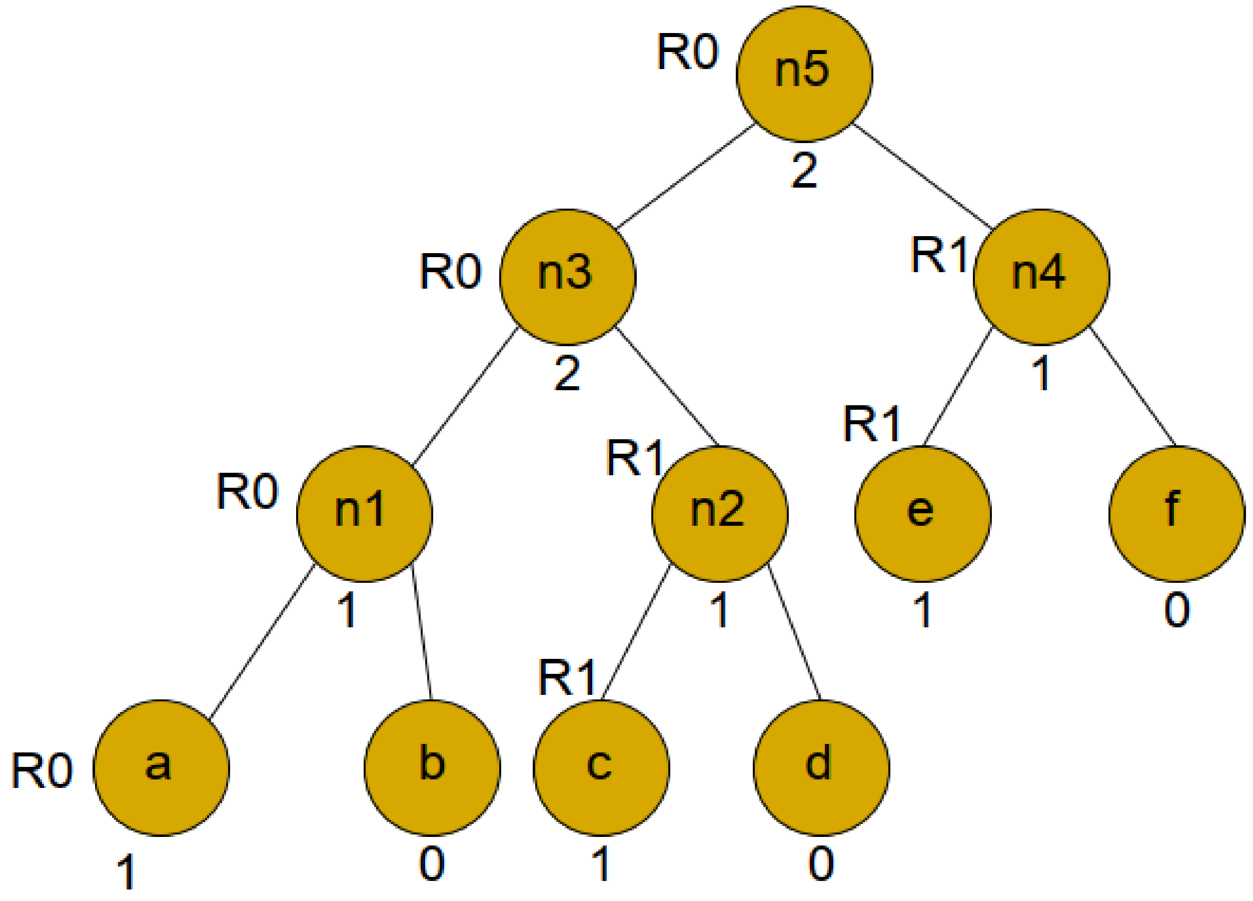
- Labelling phase
- Code generation phase

Optimal Code Generation- The Sethi-Ullman : 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)** := **if** l1<> l2 ➔ max (l1, l2),
    if l1 == l2 ➔ l1 + 1,

Optimal Code Generation- The Sethi-Ullman : Labelling Algorithm (Example)



Optimal Code Generation- The Sethi-Ullman : Code generation phase- GENCODE(n) procedure

- Stack of registers- **RSTACK**: $R_0,\ldots,R_{(r-1)}$
- Stack of temporaries- **TSTACK**: $T_0,T_1,\ldots$

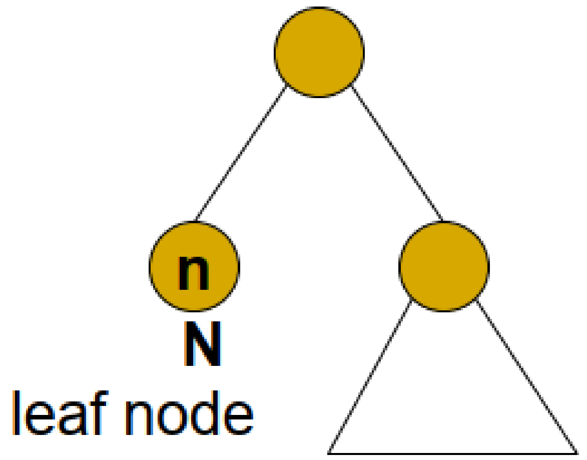
- A call to **GENCODE(n)** generates code to evaluate a tree **T**, rooted at node **n**, into the register *top*(**RSTACK**)
    - the rest of RSTACK remains in the same state as the one before the call

- A ***swap*** of the top two registers of **RSTACK** may be needed at some points in the algorithm to ensure that a node is evaluated into the same register as its left child.

Optimal Code Generation- The Sethi-Ullman : Code generation phase- GENCODE(n) procedure

- Case 1:

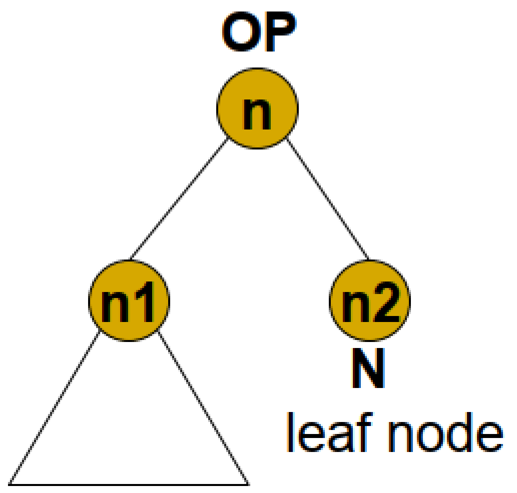
```
Procedure gencode(n);
{ /* case 1 */
  if
    n is a leaf representing
    operand N and is the
    leftmost child of its parent
  then
    print(LOAD N, top(RSTACK))
```



Optimal Code Generation- The Sethi-Ullman : Code generation phase- GENCODE(n) procedure

- Case 2:

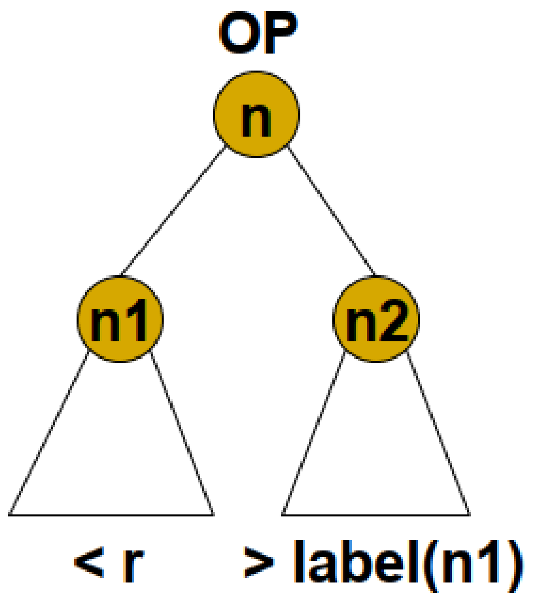
```
/* case 2 */
else if
  n is an interior node with operator
  OP, left child n1, and right child n2
then
  if  label(n2) == 0 then {
      let N be the operand for n2;
      gencode(n1);
      print(OP N, top(RSTACK));
   }
```



Optimal Code Generation- The Sethi-Ullman : Code generation phase- GENCODE(n) procedure

- Case 3:

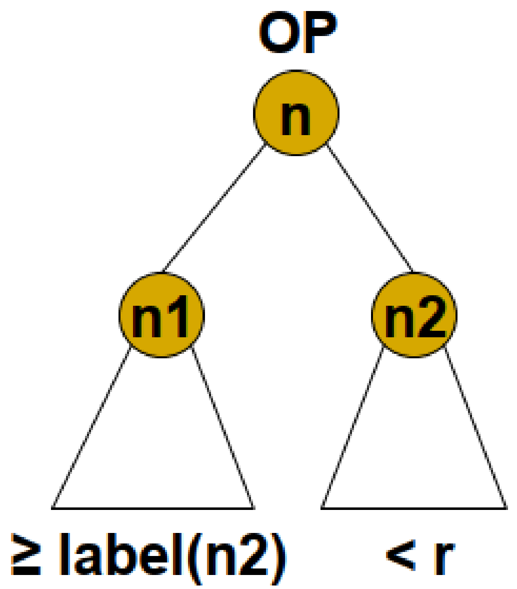
```
/* case 3 */
else if ((1 ≤ label(n1) < label(n2))
           and( label(n1) < r))
then {
  swap(RSTACK); gencode(n2);
  R := pop(RSTACK); gencode(n1);
  /* R holds the result of n2 */
  print(OP R, top(RSTACK));
  push (RSTACK,R);
  swap(RSTACK);
  }
```



Optimal Code Generation- The Sethi-Ullman : Code generation phase- GENCODE(n) procedure

- Case 4:

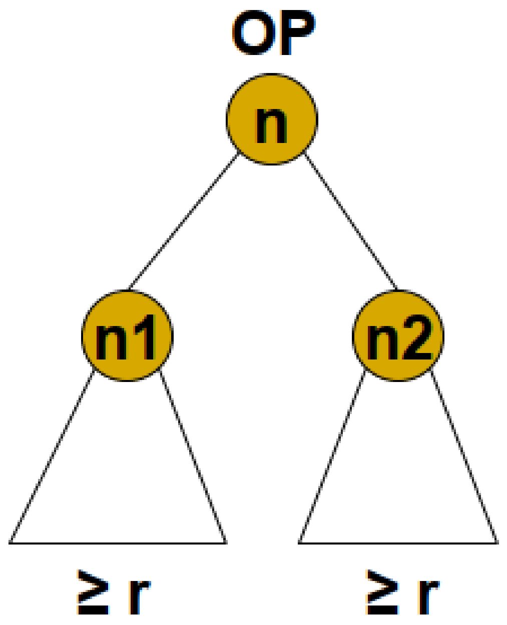
```
/* case 4 */
else if ((1 < label(n2) < label(n1))
          and( label(n2) < r))
then {
  gencode(n1);
  R := pop(RSTACK); gencode(n2);
  /* R holds the result of n1 */
  print(OP  top(RSTACK), R);
  push (RSTACK,R);
  }
```



Optimal Code Generation- The Sethi-Ullman : Code generation phase- GENCODE(n) procedure

- Case 5:

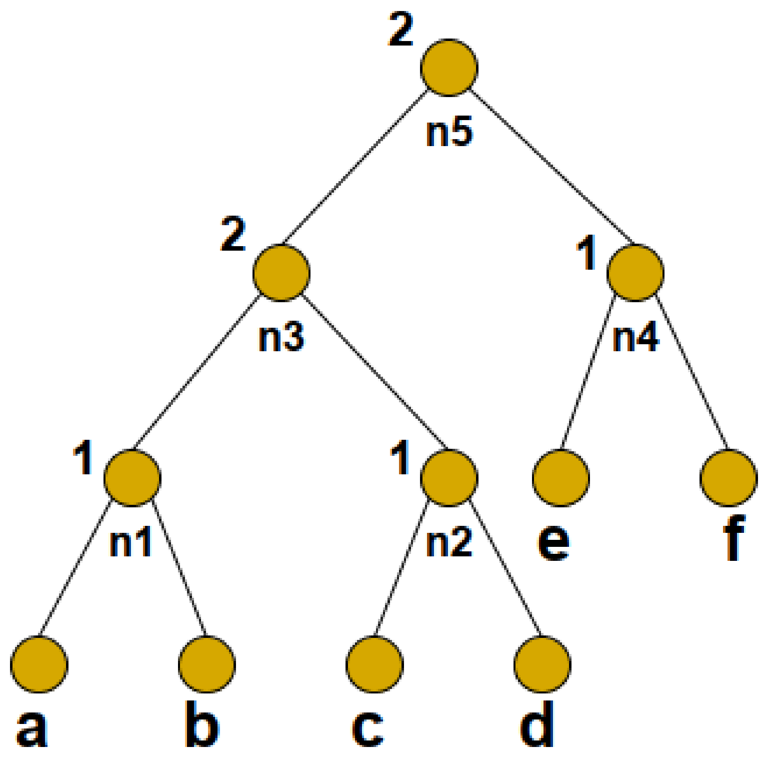
```
/* case 5, both labels are > r */
else {
   gencode(n2); T:= pop(TSTACK);
   print(LOAD top(RSTACK), T);
   gencode(n1);
   print(OP T, top(RSTACK));
   push(TSTACK, T);
   }
}
```



Optimal Code Generation- The Sethi-Ullman : Code generation phase- Example (1)

No. of registers (r): 2

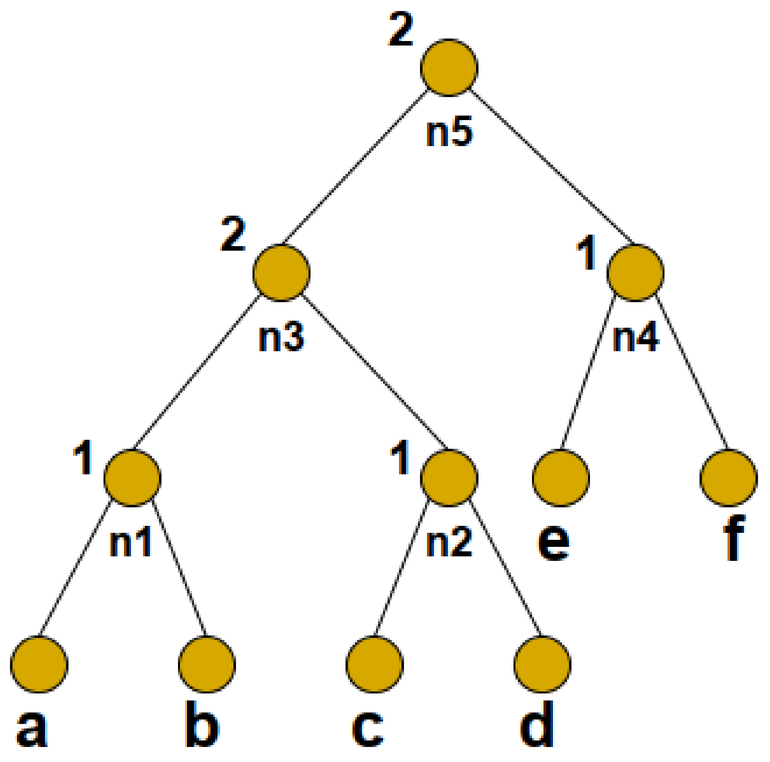
Code generated using the algorithm?



Optimal Code Generation- The Sethi-Ullman : Code generation phase- Example (1)

No. of registers (r): 2

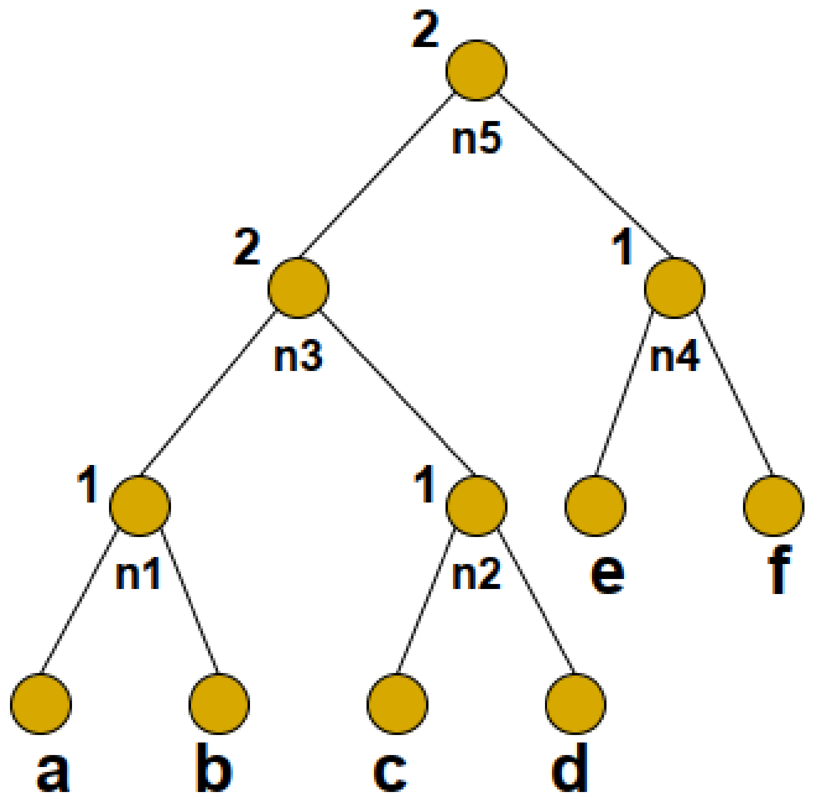
n5 → n3 → n1 → a → Load a, R0
→ op_{n1} b, R0
→ n2 → c → Load c, R1
→ op_{n2} d, R1
→ op_{n3} R1, R0
→ n4 → e → Load e, R1
→ op_{n4} f, R1
→ op_{n5} R1, R0



Optimal Code Generation- The Sethi-Ullman : Code generation phase- Example (2)

No. of registers (r): 1

Code generated?

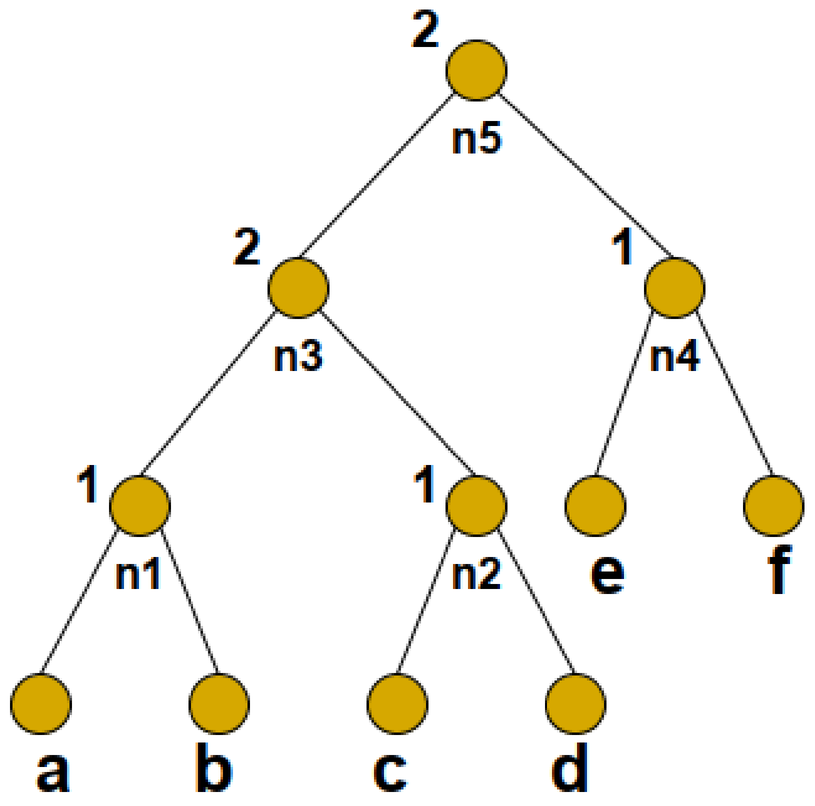


Select **r-st** first; **l-st** can computed into R0 later (**case 5**)

Optimal Code Generation- The Sethi-Ullman : Code generation phase- Example (2)

No. of registers (r): 1

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}



Here we choose **r-st** first; **l-st** can computed into R0 later (**case 5**)

Summary- Code generation

- **Machine code generation – main issues**
- **Samples of generated code**
- **Two Simple code generators**
- **Optimal code generation**
    - **Sethi-Ullman algorithm**
    - Dynamic programming based algorithm
    - Tree pattern matching based algorithm
- Code generation from DAGs

Peephole Optimizations

Peephole Optimizations

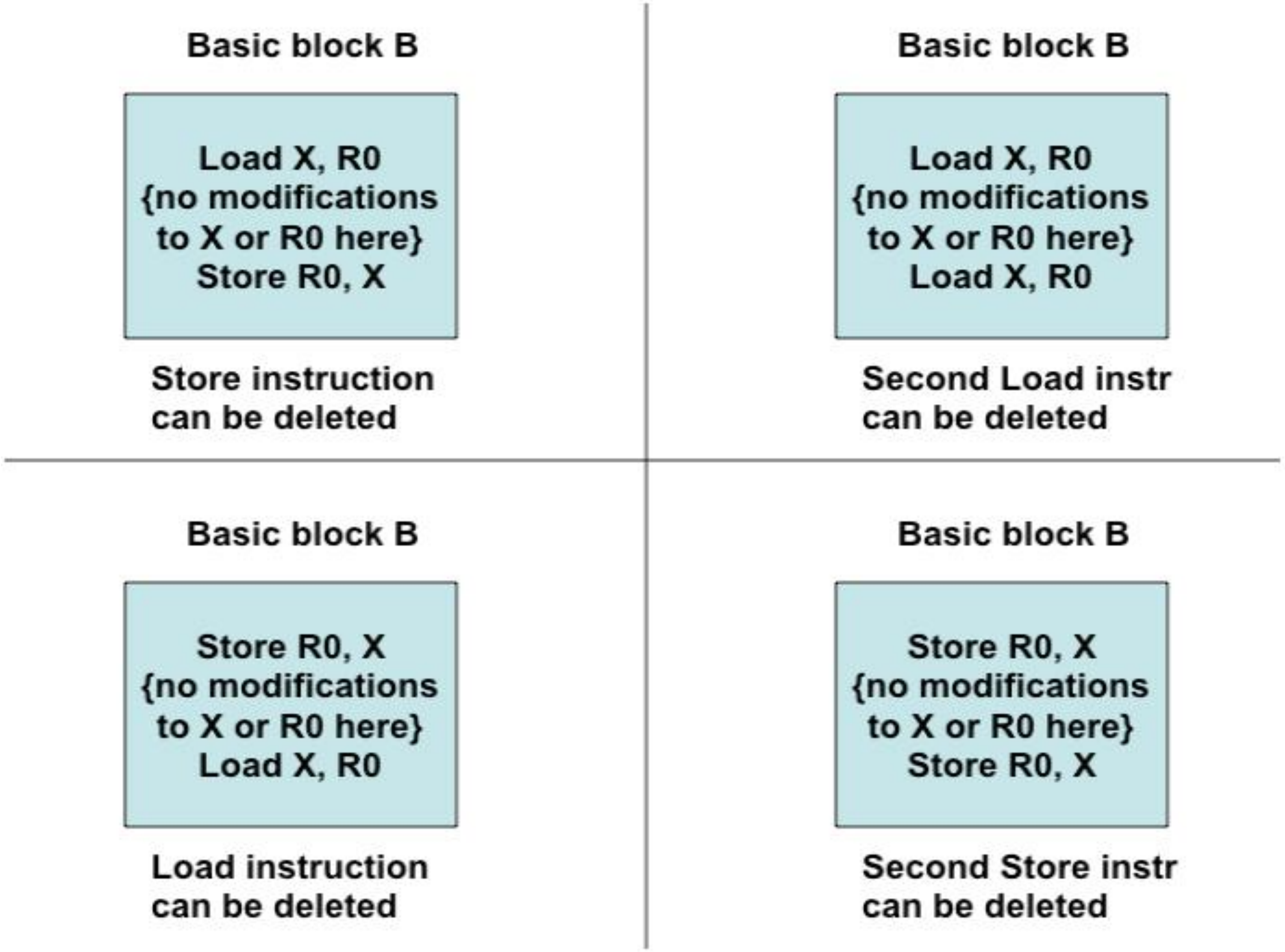
- Simple but effective local optimization
- Usually carried out on machine code
- Examines a ***sliding window*** of code (***peephole***), and replaces it by a **shorter** or **faster** sequence, if possible
- Each improvement provides opportunities for additional improvements
  - May require repeated passes over code

Peephole Optimizations

Well known peephole optimizations

- eliminating redundant instructions
- eliminating unreachable code
- eliminating jumps over jumps
- algebraic simplifications
- strength reduction
- use of machine idioms

Redundant Load and Store Elimination



Basic block B

```
Load X, R0
{no modifications
to X or R0 here}
Store R0, X
```

Store instruction can be deleted

Basic block B

```
Load X, R0
{no modifications
to X or R0 here}
Load X, R0
```

Second Load instr can be deleted

Basic block B

```
Store R0, X
{no modifications
to X or R0 here}
Load X, R0
```

Load instruction can be deleted

Basic block B

```
Store R0, X
{no modifications
to X or R0 here}
Store R0, X
```

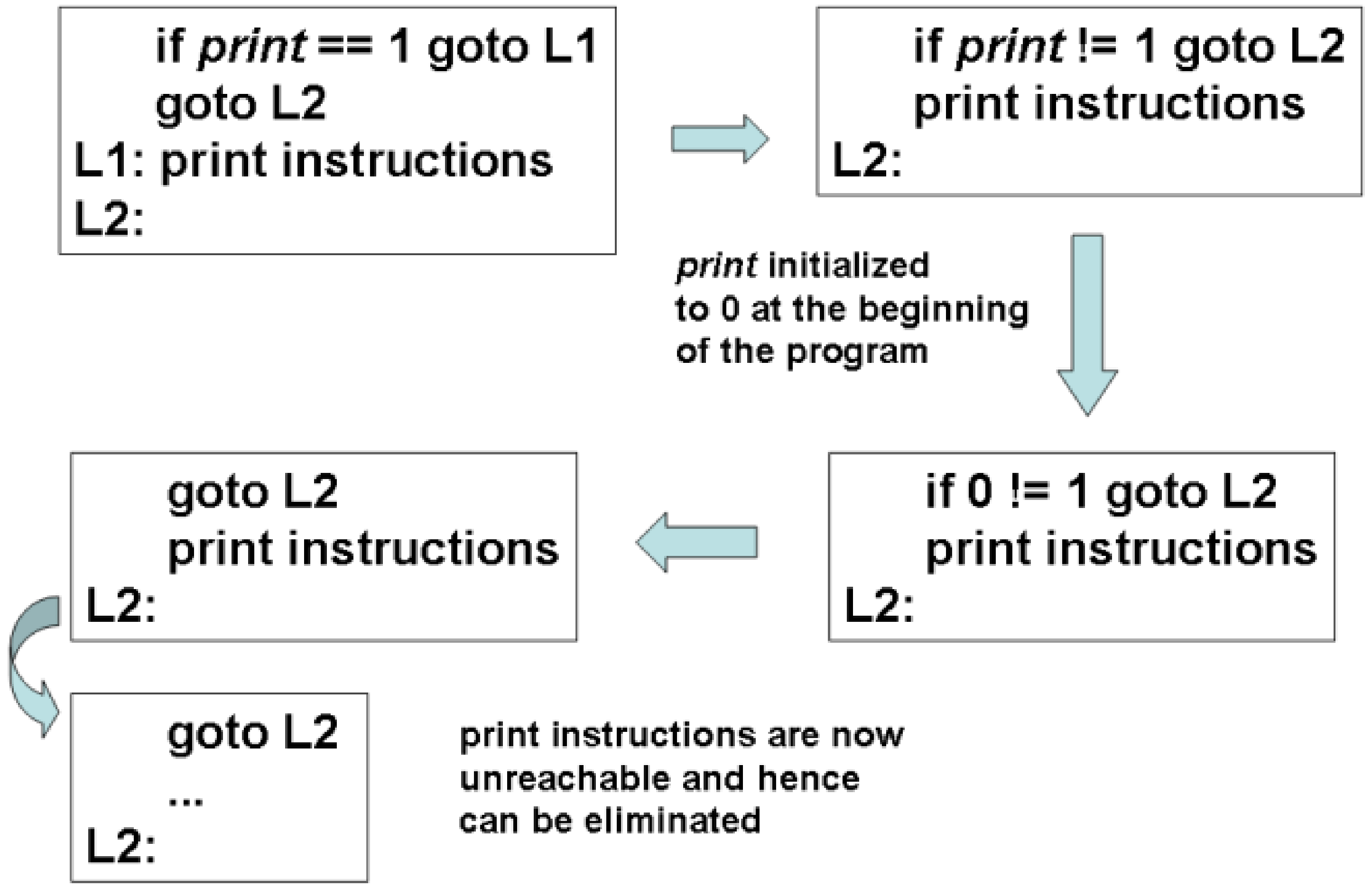
Second Store instr can be deleted

Eliminating Unreachable Code

- **Example-** An **unlabeled** instruction **immediately following an unconditional jump** can be removed

    - e.g., a print statement may have been added during debugging

Eliminating Unreachable Code



```
     if print == 1 goto L1
     goto L2
L1: print instructions
L2:
```

```
     if print != 1 goto L2
     print instructions
L2:
```

print initialized
to 0 at the beginning
of the program

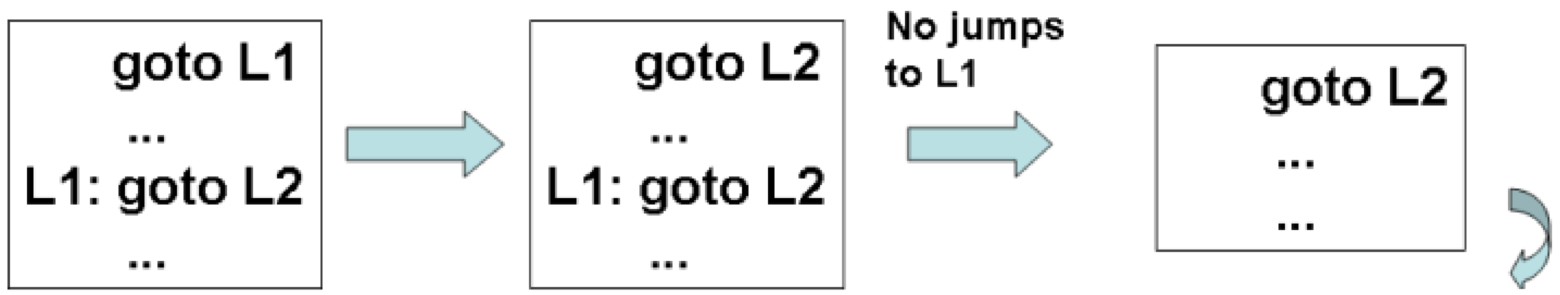
```
     if 0 != 1 goto L2
     print instructions
L2:
```

```
     goto L2
     print instructions
L2:
```

```
     goto L2
     ...
L2:
```

print instructions are now
unreachable and hence
can be eliminated

Optimizations: Flow-of-control



```
     goto L1
     ...
L1: goto L2
     ...
```

```
     goto L2
     ...
L1: goto L2
     ...
```

```
     goto L2
     ...
     ...
```

Can we always remove
Statement L1:... ??

Strength Reduction

- $\mathbf{x^2}$ is cheaper to implement as **x*x**, than as a call to an exponentiation routine
- For integers, $\mathbf{x*2^3}$ is cheaper to implement as **x << 3** (**x** left-shifted by **3** bits)
- For integers, $\mathbf{x/2^2}$ is cheaper to implement as **x >> 2** (x right-shifted by 2 bits)

- ……………………

All the best !!