Machine Code Generation - 1

Y. N. Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- 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

Code Generation – Main Issues (1)

- Transformation:
  - Intermediate code → 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
```

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, location> 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
  - 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)

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

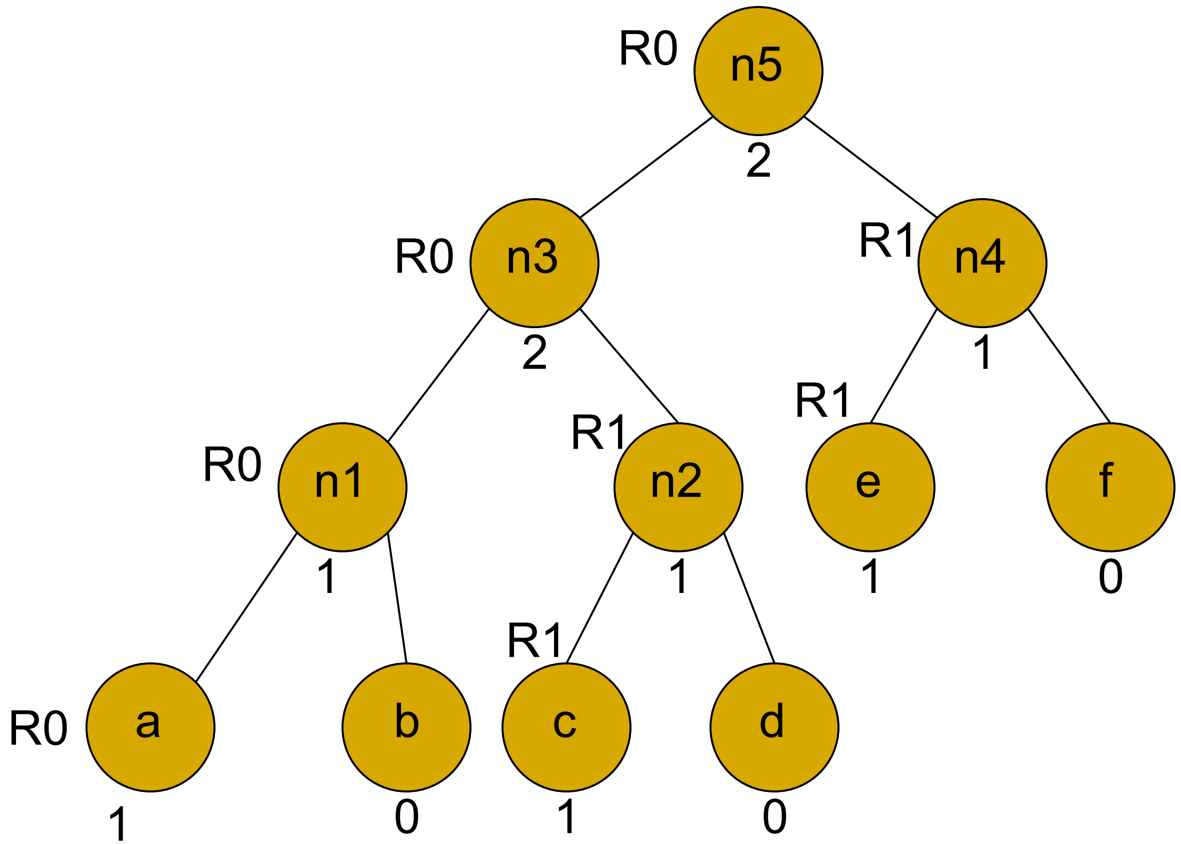
The Labelling Algorithm

- 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 (l_1, l_2), if l_1<> l_2**

    = l_1 + 1, if l_1 = l_2

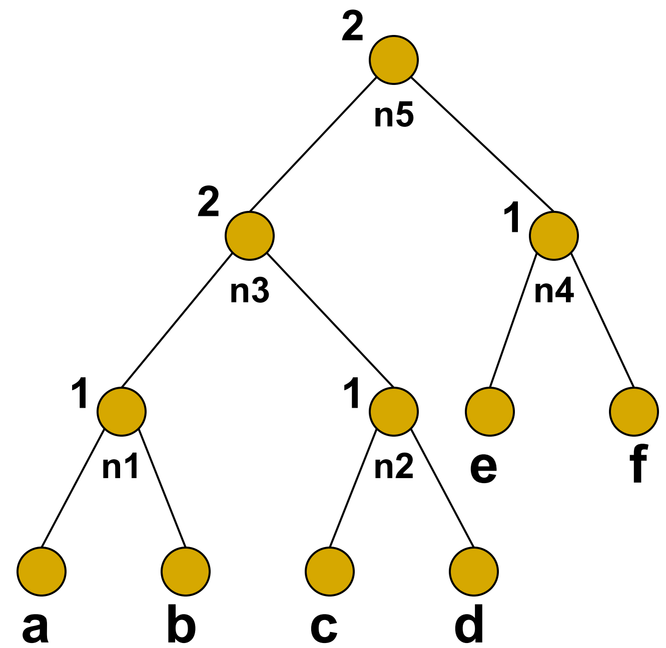
Labelling - Example



Code Generation Phase – Example 1

No. of registers = r = 2

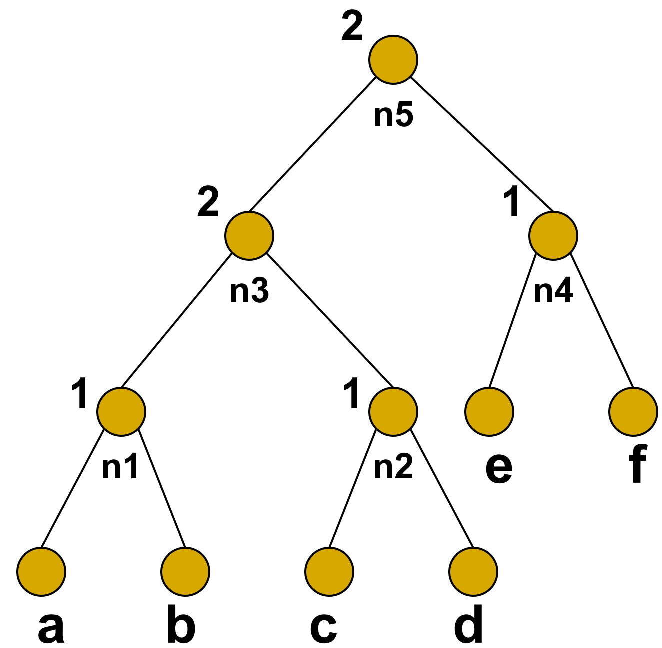
n5 → n3 → n1 → a → Load a, R0
→ op_{n1} b, R0
→ n2 → c → Load c, R1
→ op_{n2} d, R1
→ op_{n3} R1, R0
→ n4 → e → Load e, R1
→ op_{n4} f, R1
→ op_{n5} R1, R0



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)

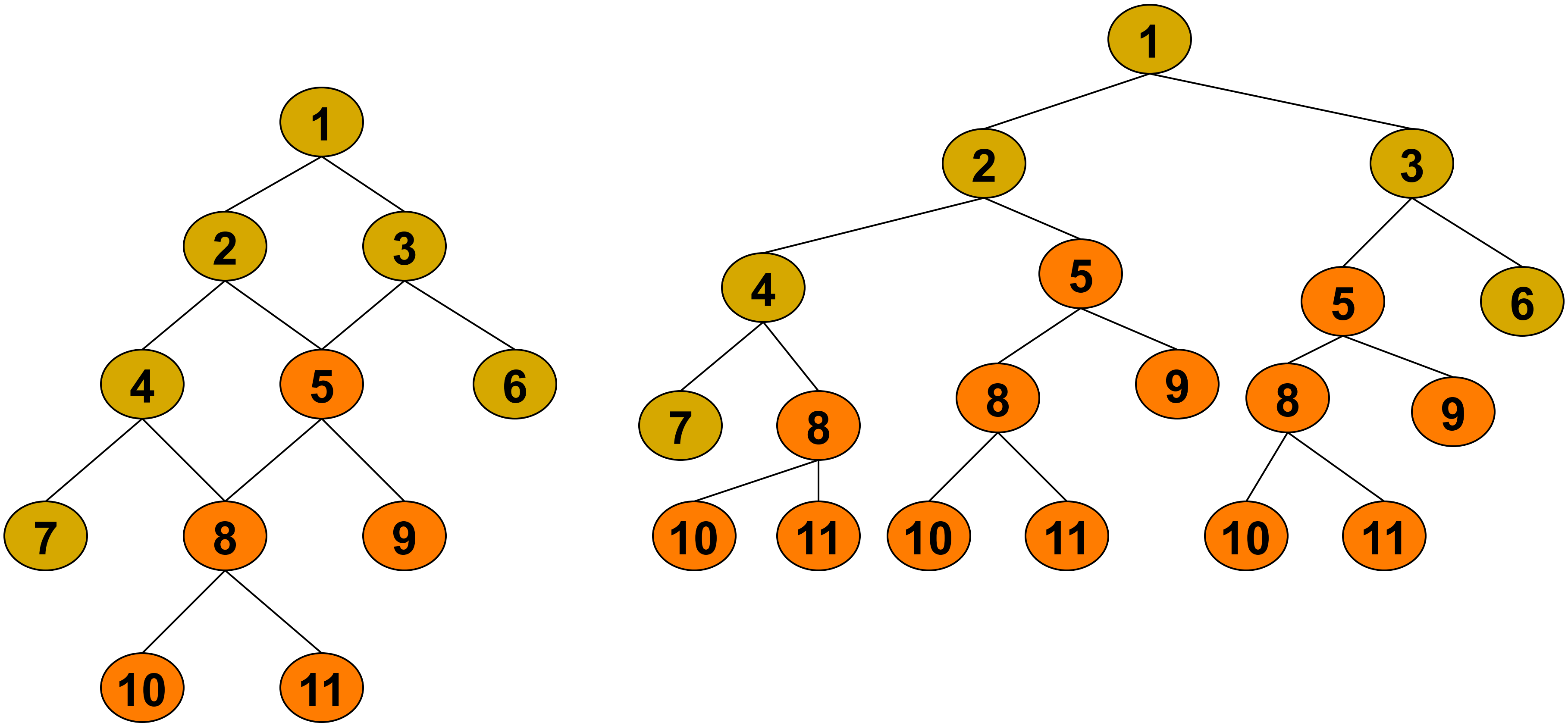
n5 → n4 → e → Load e, R0
→ op_{n4} f, R0
→ Load R0, T0 {release R0}
→ n3 → n2 → c → Load c, R0
→ op_{n2} d, R0
→ Load R0, T1 {release R0}
→ n1 → a → Load a, R0
→ op_{n1} b, R0
→ op_{n3} T1, R0 {release T1}
→ op_{n5} T0, R0 {release T0}



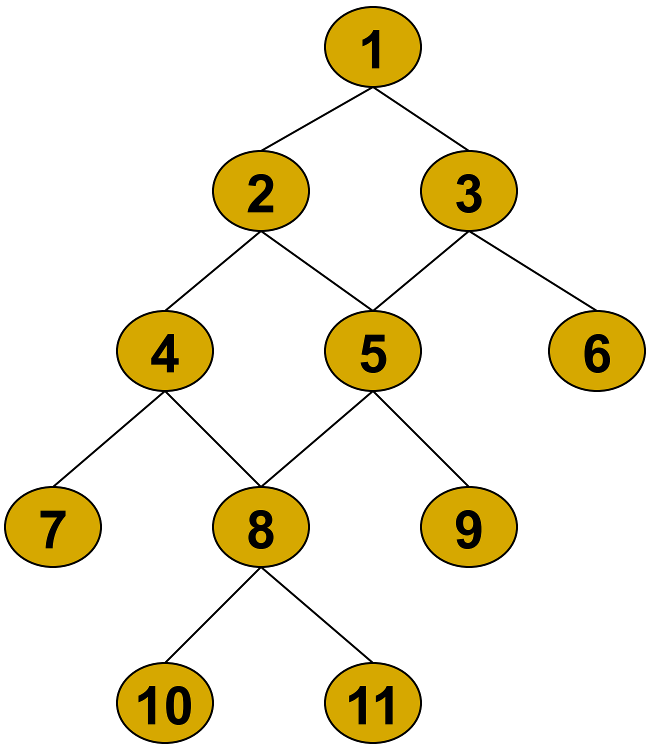
Code Generation from DAGs

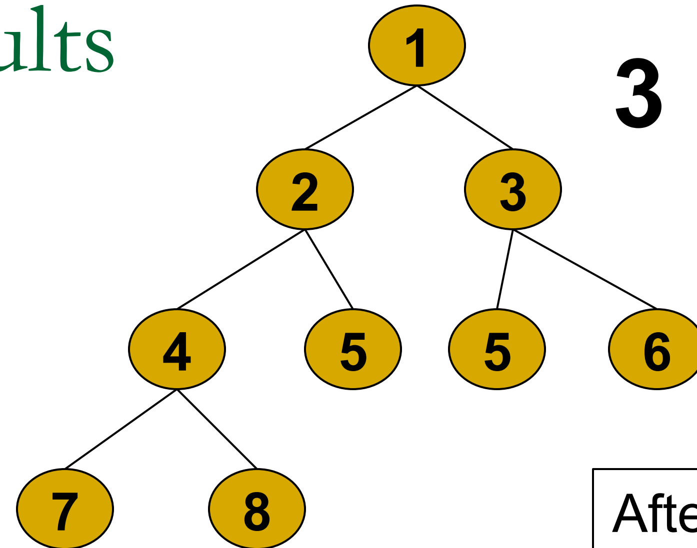
- Optimal code generation from DAGs is NP-Complete
- DAGs are divided into trees and then processed
- We may replicate shared trees
  - Code size increases drastically
- We may store result of a tree (root) into memory and use it in all places where the tree is used
  - May result in sub-optimal code

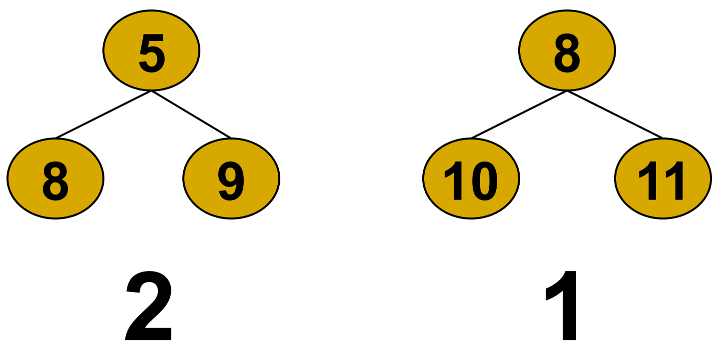
DAG example: Duplicate shared trees



DAG example: Compute shared trees once and share results







After computing tree 1, the computation of subtree 4-7-8 of tree 3 can be done before or after tree 2

Peephole Optimizations

- Simple but effective local optimization
- Usually carried out on machine code, but intermediate code can also benefit from it
- 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
- Therefore, repeated passes over code are needed

Peephole Optimizations

- Some well known peephole optimizations
  - eliminating redundant instructions
  - eliminating unreachable code
  - eliminating jumps over jumps
  - algebraic simplifications
  - strength reduction
  - use of machine idioms

Elimination of Redundant Loads and Stores

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

- An unlabeled instruction immediately following an unconditional jump may be removed
  - May be produced due to debugging code introduced during development
  - Or due to updates to programs (changes for fixing bugs) without considering the whole program segment

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

Flow-of-Control Optimizations

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

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

No jumps to L1

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

Statement L1: ... can be removed only if it is preceded by an unconditional jump

```
if a<b goto L1
       ...
L1: goto L2
       ...
```

```
if a<b goto L2
       ...
L1: goto L2
       ...
```

always executes “goto L1”

```
      goto L1
       ...
L1: if a<b goto L2
L3:
       ...
```

Only one jump to L1, L1 is preceded by an unconditional goto

sometimes skips “goto L3”

```
      if a<b goto L2
      goto L3
       ...
L3:
       ...
```

Reduction in Strength and Use of Machine Idioms

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