Machine Code Generation - 4

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

- Mach. code generation – main issues (in part 1)
- Samples of generated code (in part 2)
- Two Simple code generators (in part 2)
- Optimal code generation
  - Sethi-Ullman algorithm (in part 3)
  - Dynamic programming based algorithm (in part 3)
  - Tree pattern matching based algorithm
- Code generation from DAGs
- Peephole optimizations

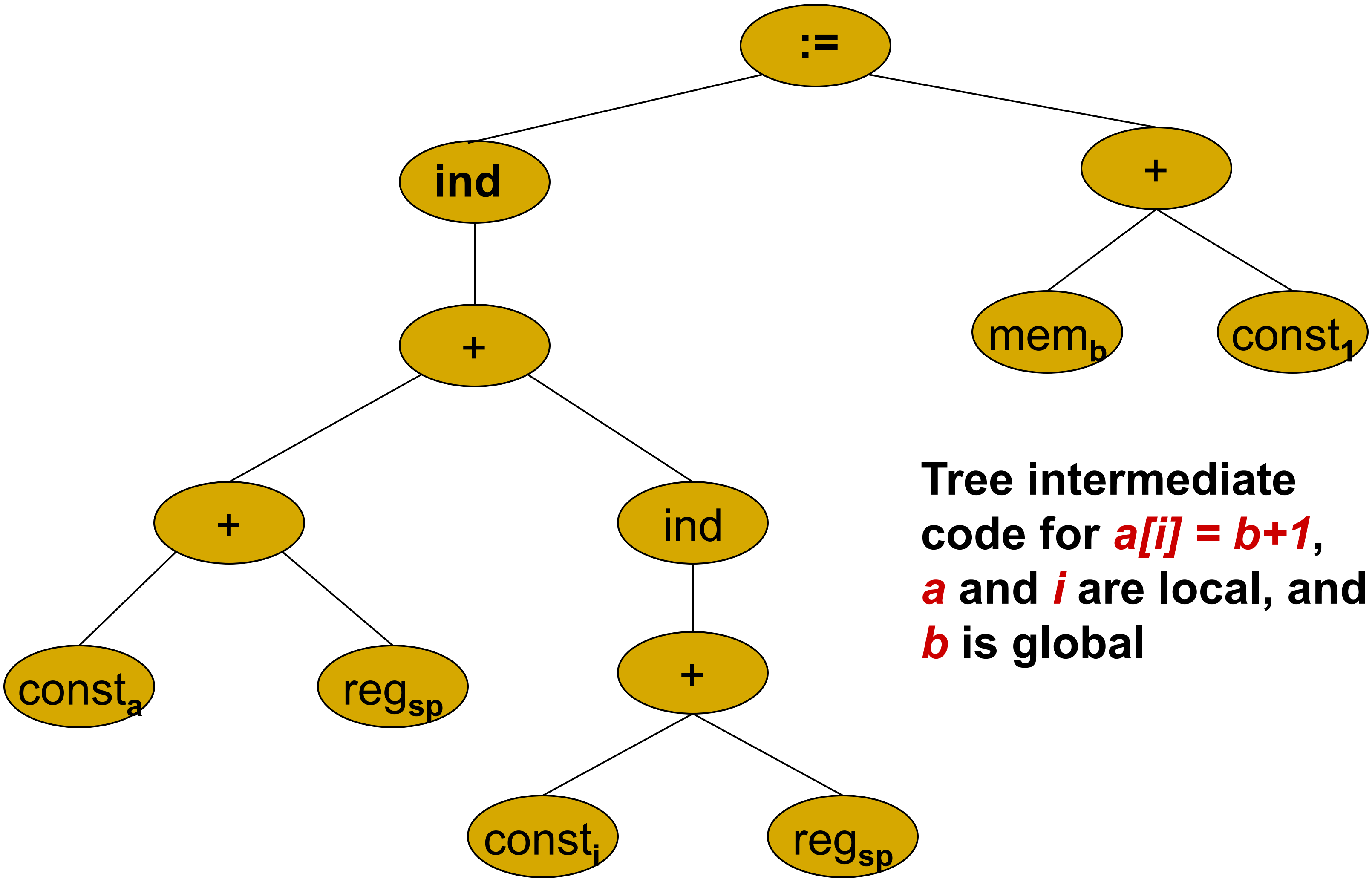
Code Generation based on Dynamic Programming - Limitations

- Several instructions require even-odd register pairs – (R_0,R_1), (R_2,R_3), etc.
  - example: multiplication in x86
  - may require non-contiguous evaluation to ensure optimality
  - cannot be handled by DP

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 is possible

Example



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

Some Tree Rewriting Rules and Associated Actions

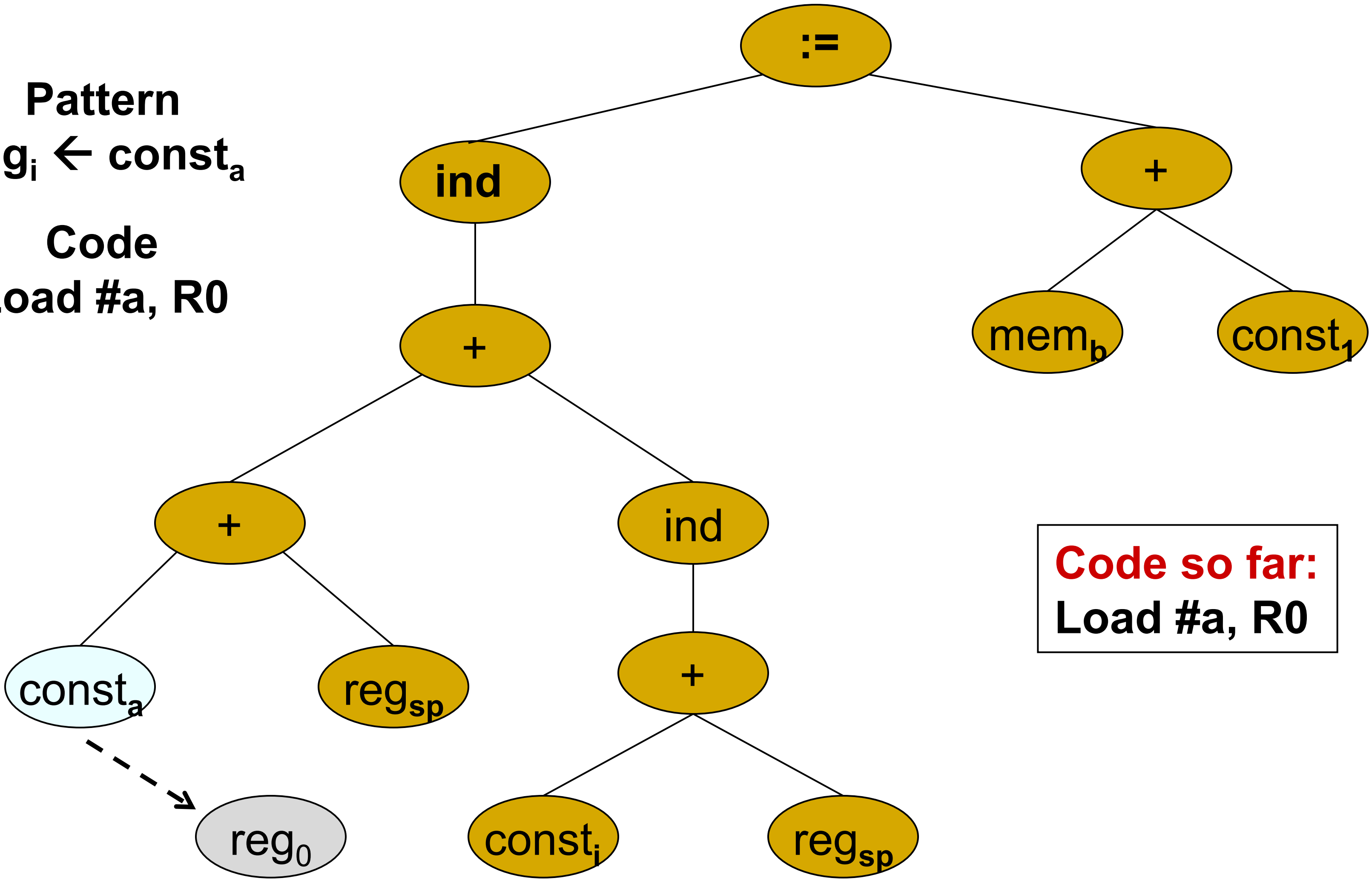
1. $reg_i \leftarrow const_a$ { Load #a, reg_i }
2. $reg_i \leftarrow +(reg_i , reg_j)$ { Add reg_i , reg_j }
3. $reg_i \leftarrow ind\ (+(const_c , reg_j))$ { Load #c(reg_j), reg_i }
4. $reg_i \leftarrow +(reg_i , ind\ (+(const_c , reg_j)))$
   { Add #c(reg_j), reg_i }
5. $reg_i \leftarrow mem_a$ { Load b, reg_i }

6. $reg_i \leftarrow +(reg_i , const_1)$ { Inc reg_i }
7. $mem \leftarrow :=(ind\ (reg_i) , reg_j)$ { Load reg_j , *reg_i }

Match #1

Pattern
$reg_i \leftarrow const_a$

Code
Load #a, R0

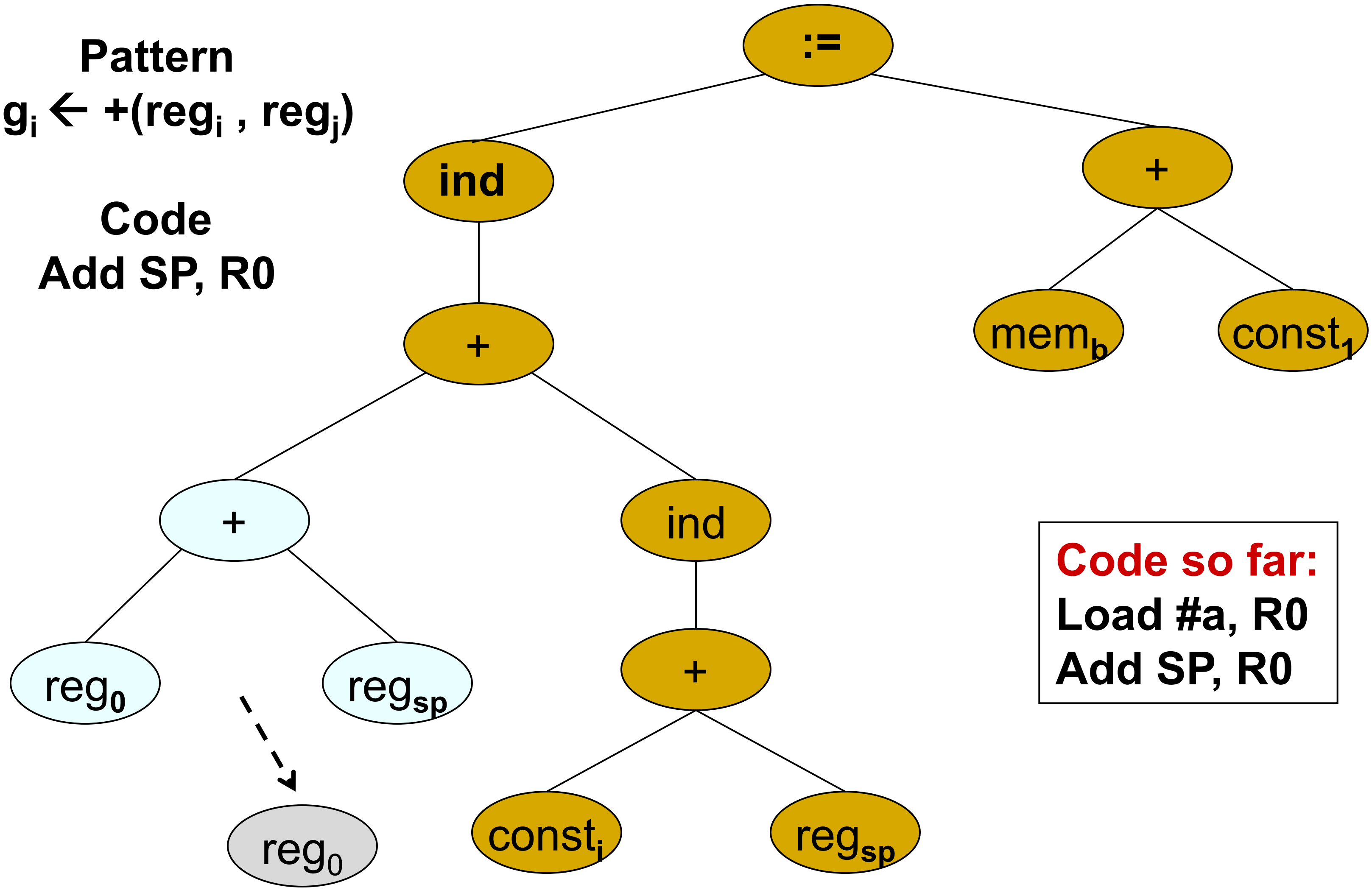


Code so far:
Load #a, R0

Match #2

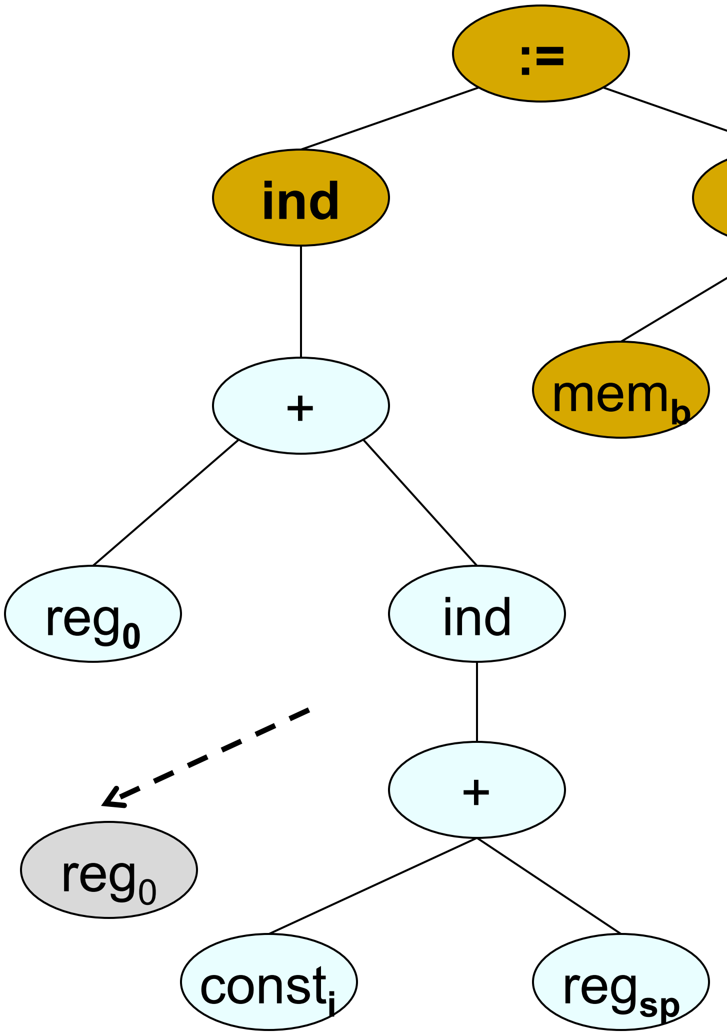
Pattern
$reg_i \leftarrow +(reg_i\ ,\ reg_j)$

Code
Add SP, R0



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

Match #3

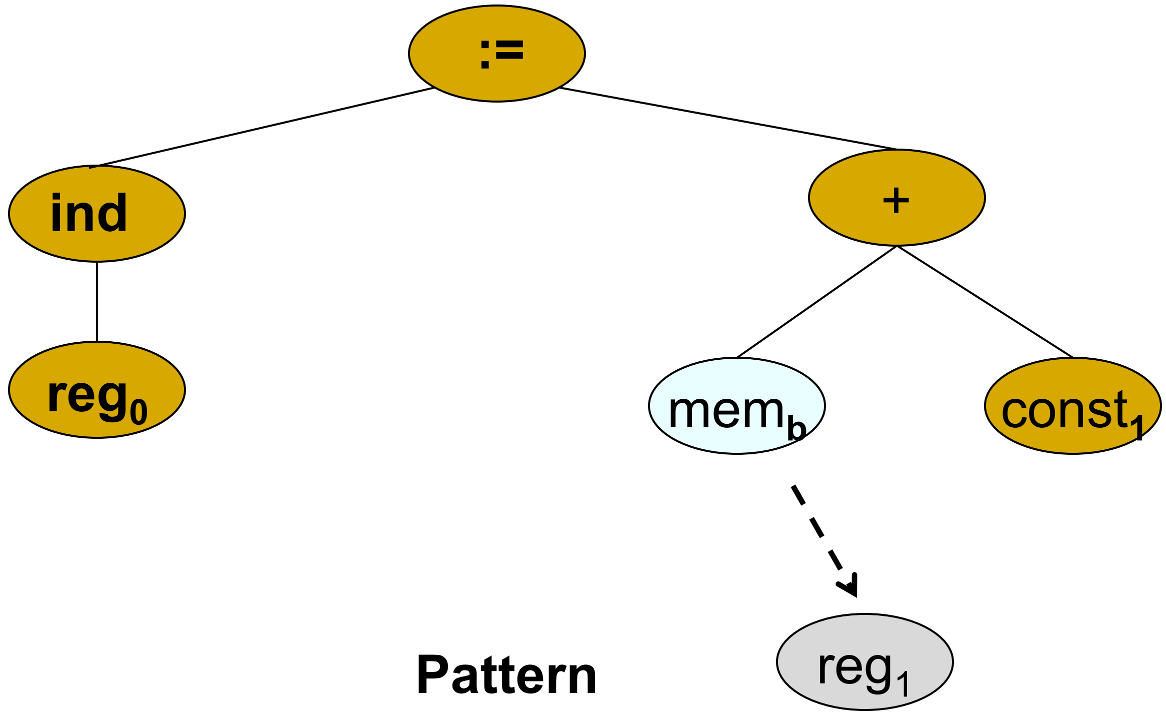


Code so far:
Load #a, R0
Add SP, R0
Add #i(SP), R0

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

Match #4

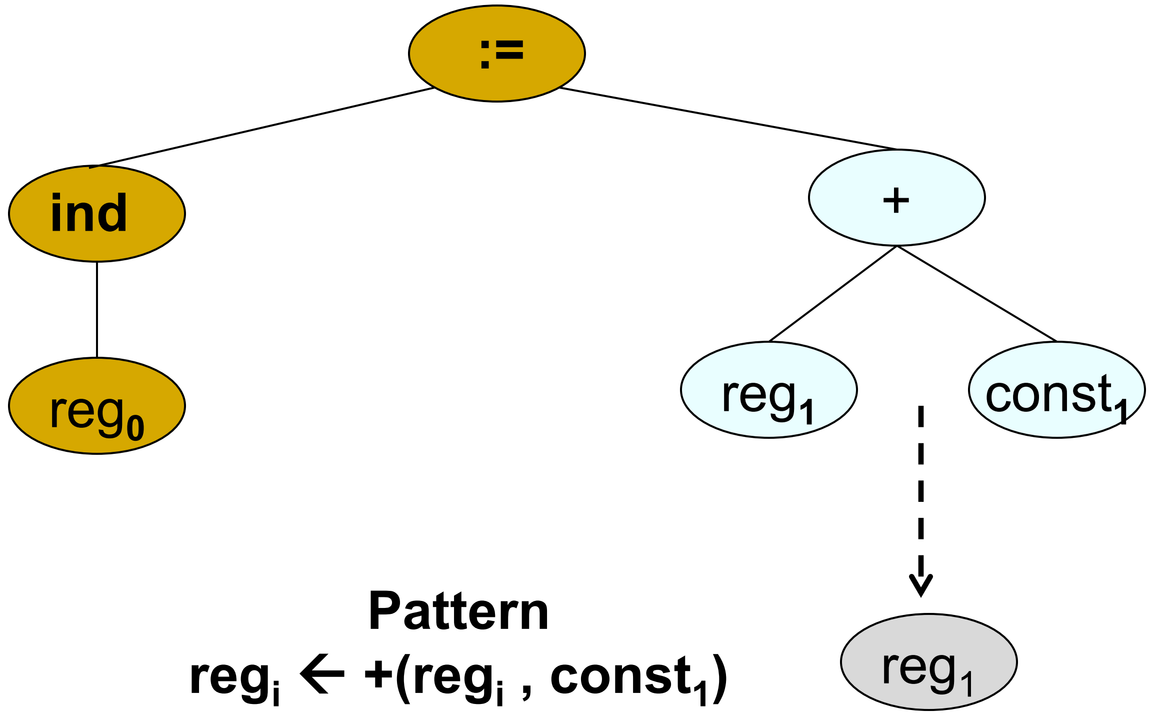


Code so far:
Load #a, R0
Add SP, R0
Add #i(SP), R0
Load b, R1

Pattern
$reg_i \leftarrow mem_a$

Code
Load b, R1

Match #5



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

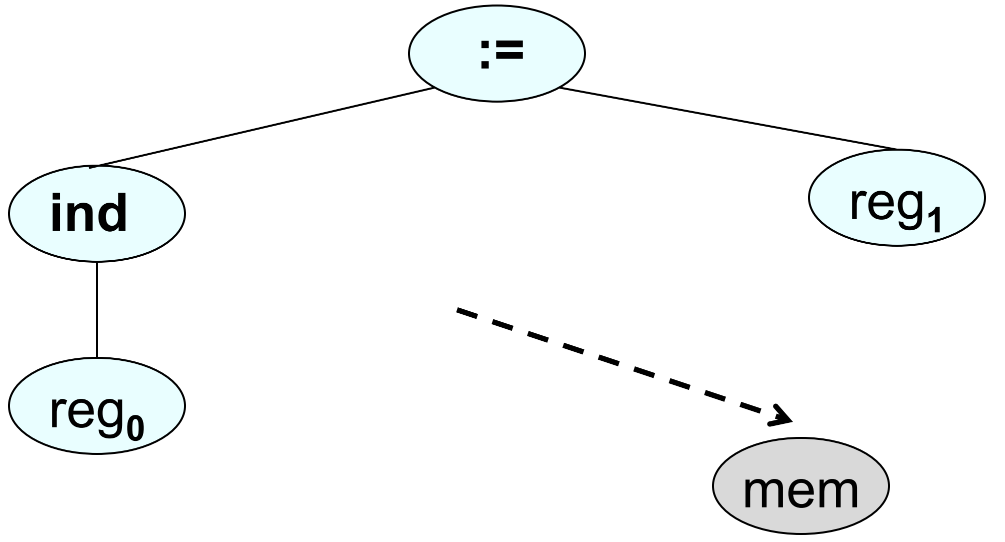
Pattern

$\mathbf{reg_i \leftarrow +(reg_i\ ,\ const_1)}$

Code

Inc R1

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 ← :=(ind (reg_i) , reg_j)

Code

Load R1, *R0

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
- Twig, BURG, and IBURG are such CGGs

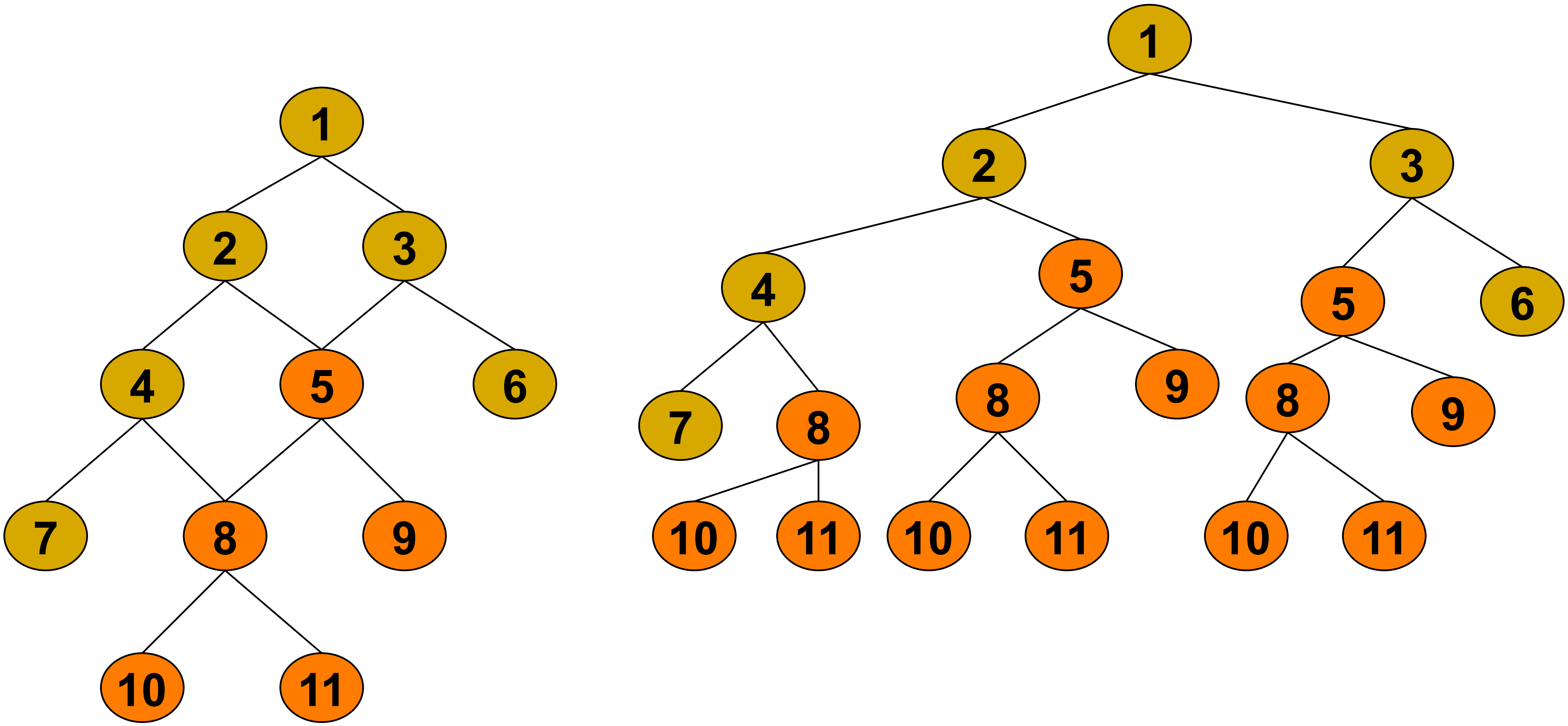
Code Generator Generators (2)

- IBURG
  - Uses 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

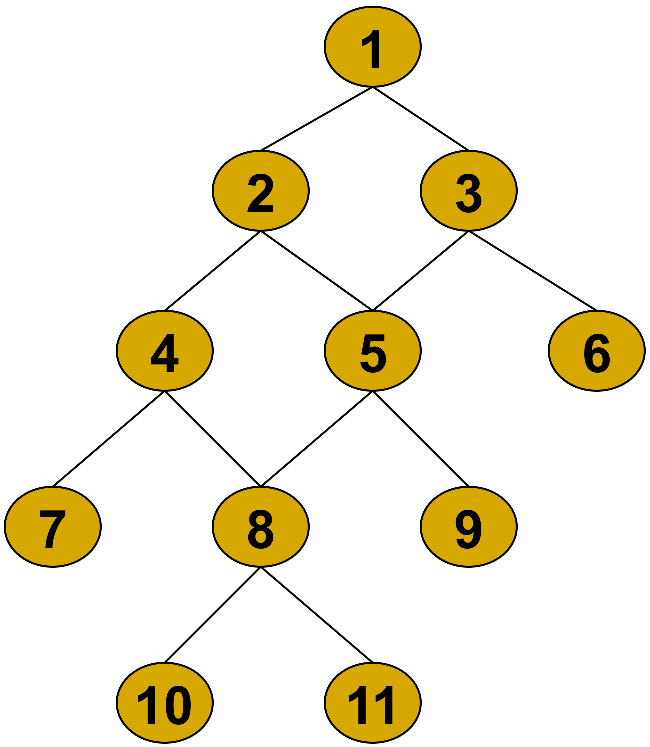
Code Generation from DAGs

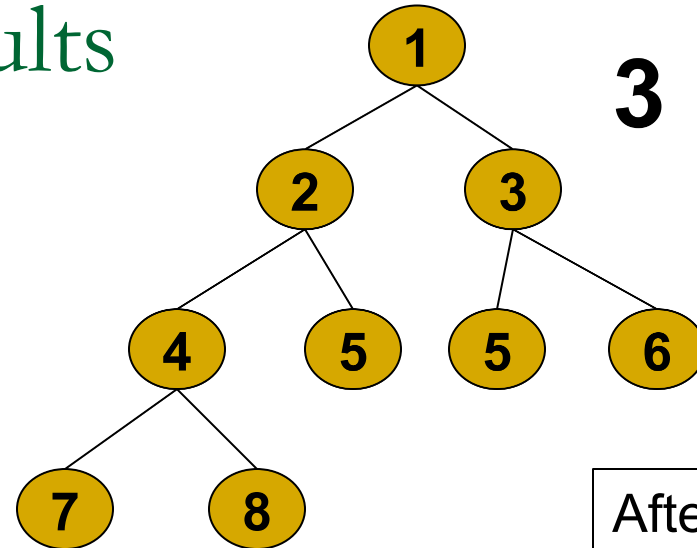
- Optimal code generation from DAGs is NP-Complete
- DAGs are divided into trees and then processed
- We may replicate shared trees
  - Code size increases drastically
- We may store result of a tree (root) into memory and use it in all places where the tree is used
  - May result in sub-optimal code

DAG example: Duplicate shared trees



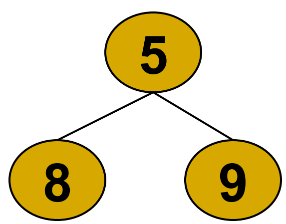
DAG example: Compute shared trees once and share results



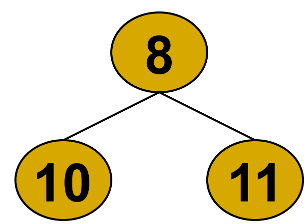


3

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



2



1

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)

Reduction in Strength and Use of Machine Idioms

- Floating point division by a constant can be approximated as multiplication by a constant
- Auto-increment and auto-decrement addressing modes can be used wherever possible
  - Subsume INCREMENT and DECREMENT operations (respectively)
- Multiply and add is a more complicated pattern to detect

Implementing Object-Oriented Languages

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

NPTEL Course on Principles of Compiler Design

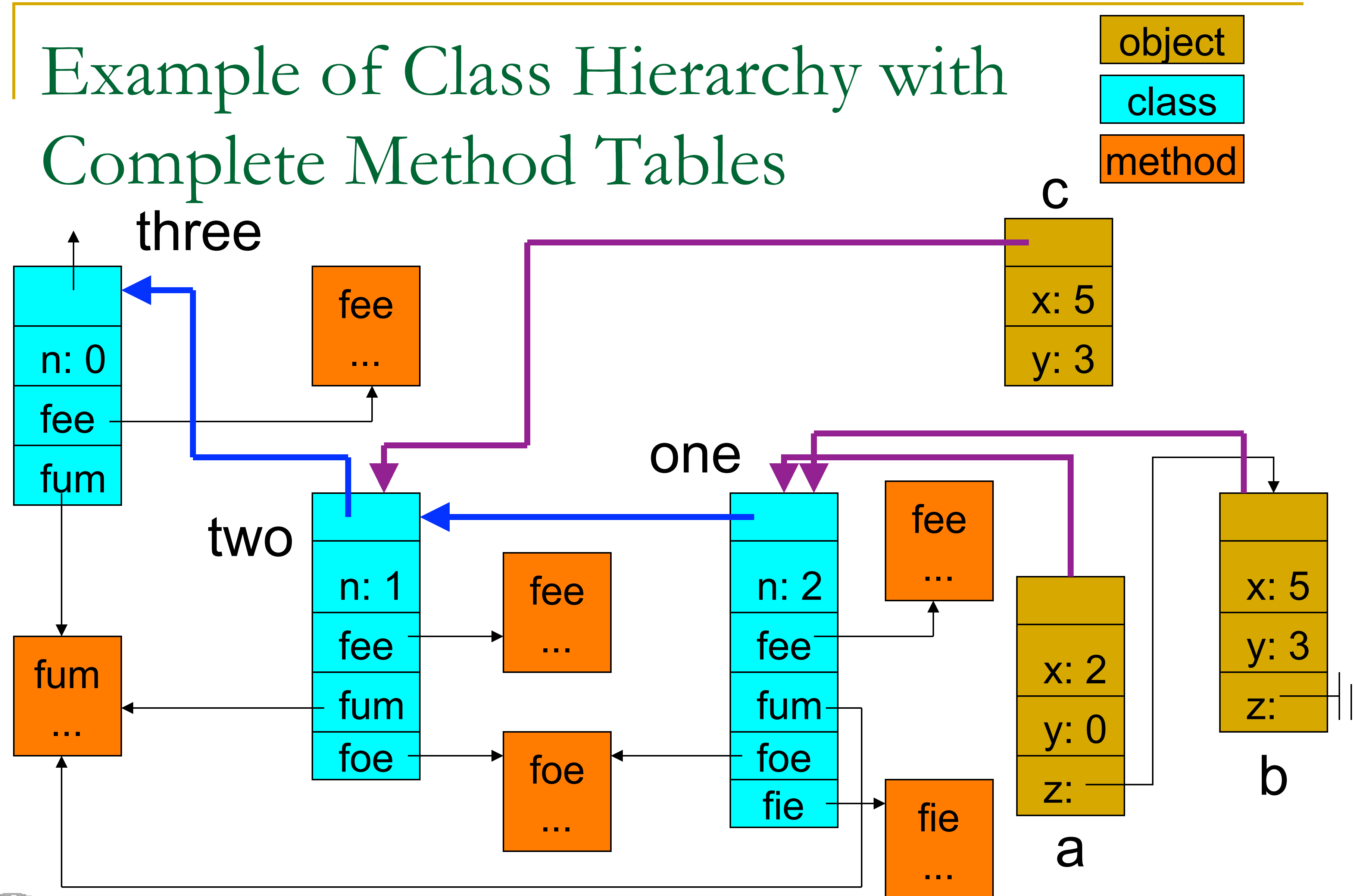
Outline of the Lecture

- Language requirements
- Mapping names to methods
- Variable name visibility
- Code generation for methods
- Simple optimizations
- **Parts of this lecture are based on the book, "Engineering a Compiler", by Keith Cooper and Linda Torczon, Morgan Kaufmann, 2004, sections 6.3.3 and 7.10.**

Language Requirements

- Objects and Classes
- Inheritance, subclasses and superclasses
- Inheritance requires that a subclass have all the instance variables specified by its superclass
  - Necessary for superclass methods to work with subclass objects
- If A is B's superclass, then some or all of A's methods/instance variables may be redefined in B

Example of Class Hierarchy with Complete Method Tables



Mapping Names to Methods

- Method invocations are not always static calls
- *a.fee*() invokes *one.fee*(), *a.foe*() invokes *two.foe*(), and *a.fum*() invokes *three.fum*()
- Conceptually, method lookup behaves as if it performs a search for each procedure call
  - These are called virtual calls
  - Search for the method in the receiver's class; if it fails, move up to the receiver's superclass, and further
  - To make this search efficient, an implementation places a complete method table in each class
  - Or, a pointer to the method table is included (virtual tbl ptr)

Mapping Names to Methods

- If the class structure can be determined wholly at compile time, then the method tables can be statically built for each class
- If classes can be created at run-time or loaded dynamically (class definition can change too)
  - full lookup in the class hierarchy can be performed at run-time or
  - use complete method tables as before, and include a mechanism to update them when needed