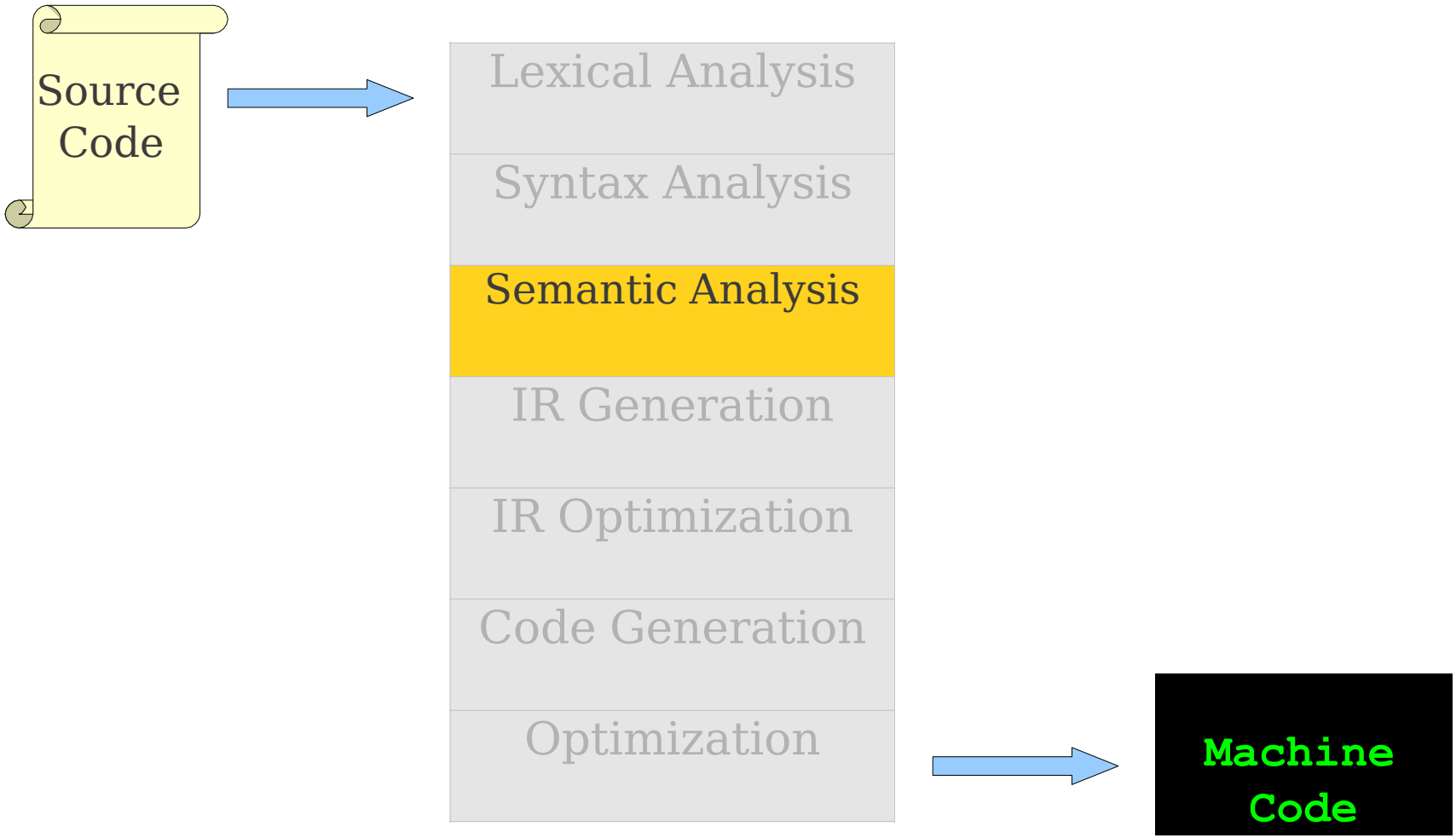
بسم الله الرحمن الرحيم

Semantic Analysis

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

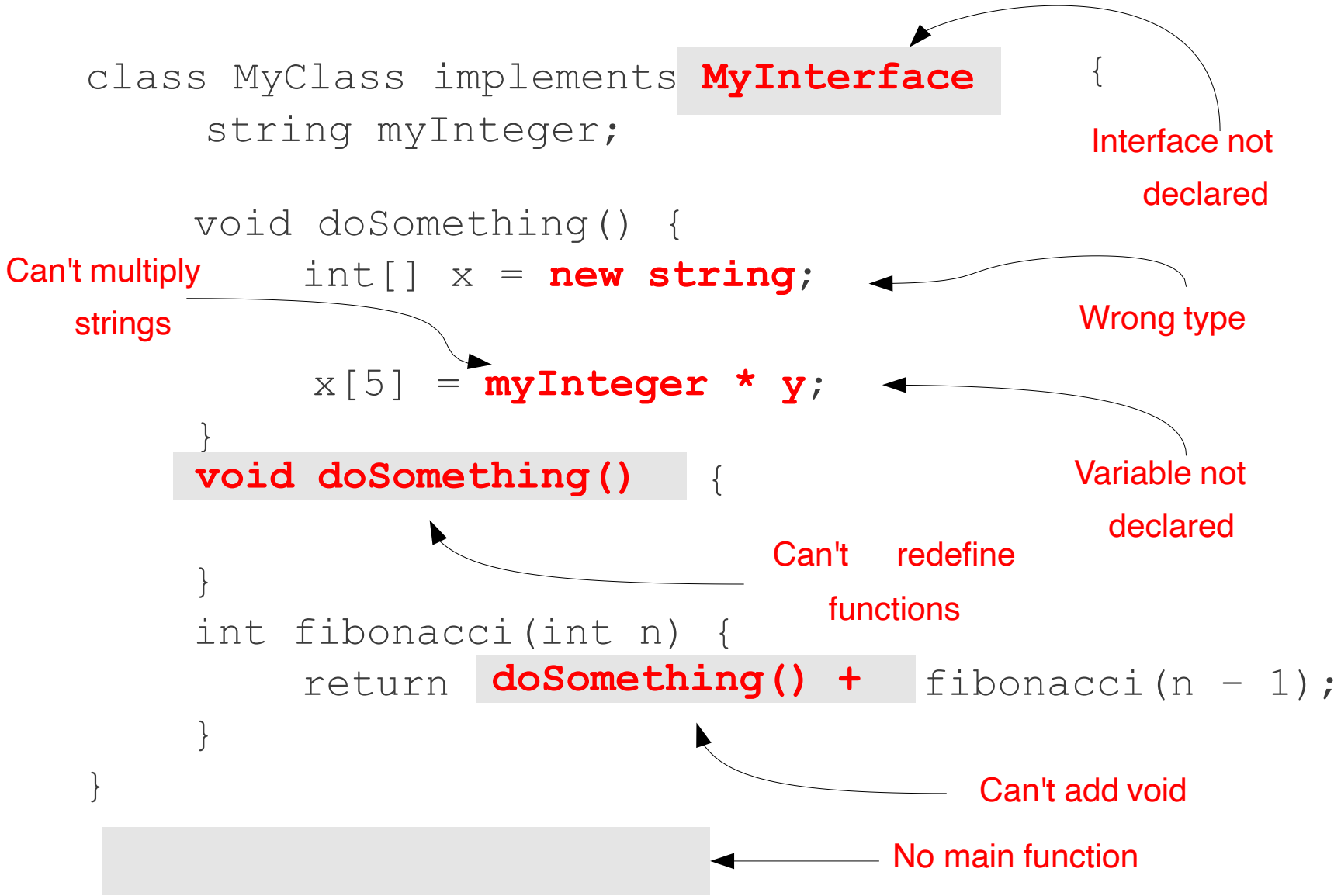
Where We Are

- Program is *lexically* well-formed:
  - Identifiers have valid names.
  - Strings are properly terminated.
  - No stray characters.
- Program is *syntactically* well-formed:
  - Class declarations have the correct structure.
  - Expressions are syntactically valid.
- Does this mean that the program is legal?

A Short Decaf Program

```
class MyClass implements MyInterface
     {  string myInteger;

     void doSomething()
          {  int[] x = new
          string;

           x[5] = myInteger * y;
     }
     void doSomething() {

     }
     int fibonacci(int n) {
           return doSomething() + fibonacci(n - 1);
     }
}
```

A Short Decaf Program

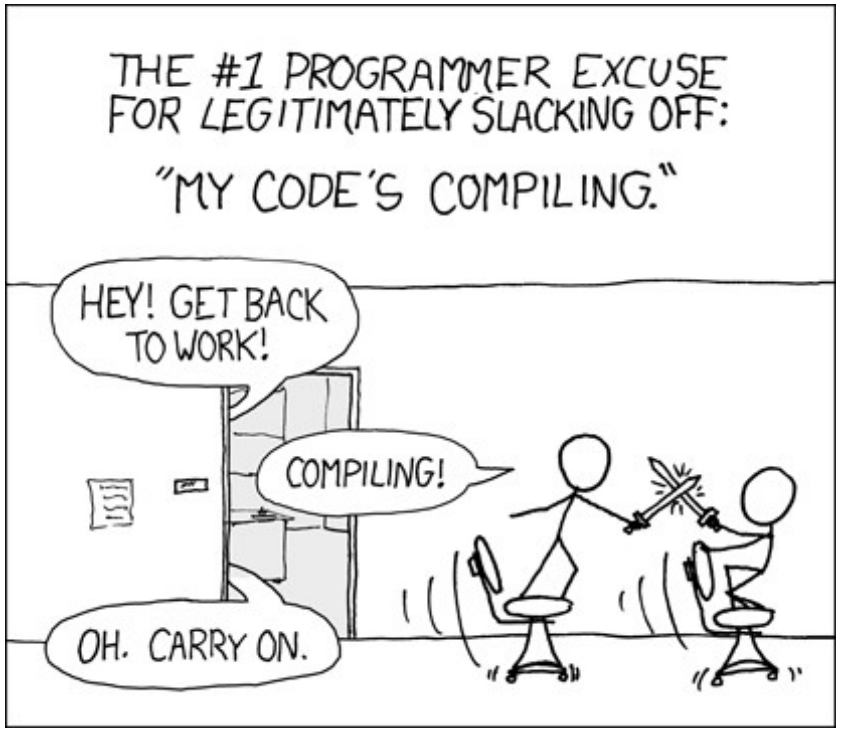


Semantic Analysis

- Ensure that the program has a well-defined **meaning**.
- Verify properties of the program that aren't caught during the earlier phases:
  - Variables are declared before they're used.
  - Expressions have the right types.
  - Arrays can only be instantiated with `NewArray`.
  - Classes don't inherit from nonexistent base classes
  - ...
- Once we finish semantic analysis, we know that the user's input program is legal.

Challenges in Semantic Analysis

- Reject the largest number of incorrect programs.
- Accept the largest number of correct programs.
- Do so quickly.



Why can't we just do this during parsing?

Limitations of CFGs

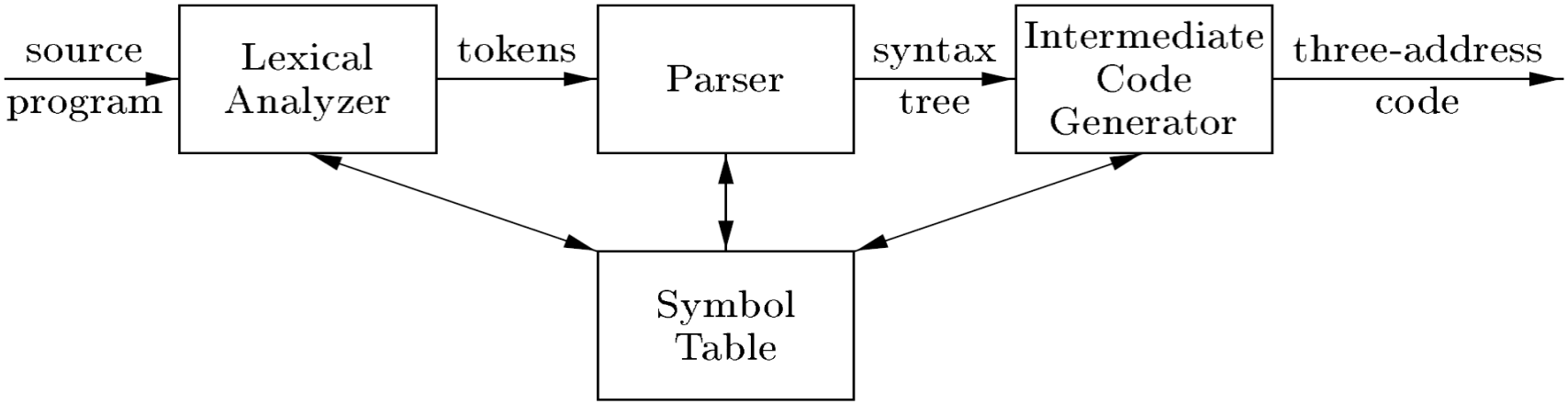
- Using CFGs:
  - How would you prevent duplicate class definitions?
  - How would you differentiate variables of one type from variables of another type?
  - How would you ensure classes implement all interface methods?
- For most programming languages, these are *provably impossible*.
  - Use the pumping lemma for context-free languages, or Ogden's lemma.

Implementing Semantic Analysis

- **Attribute Grammars**
  - Augment `bison` rules to do checking during parsing.
  - Approach suggested in the *Compilers* book.
  - Has its limitations; more on that later.
- **Recursive AST Walk**
  - Construct the AST, then use virtual functions and recursion to explore the tree.

A Remaining Question:

- How Does Parser and Semantic Analyzer interact?
- Recall the overall structure:



Communication to SA and CG

- Sometimes we can merge Semantic Analyzer and Code-Generating units.
- However we must design a way to communicate.
- i.e. What should we do when (e.g.) an addition is detected in program?

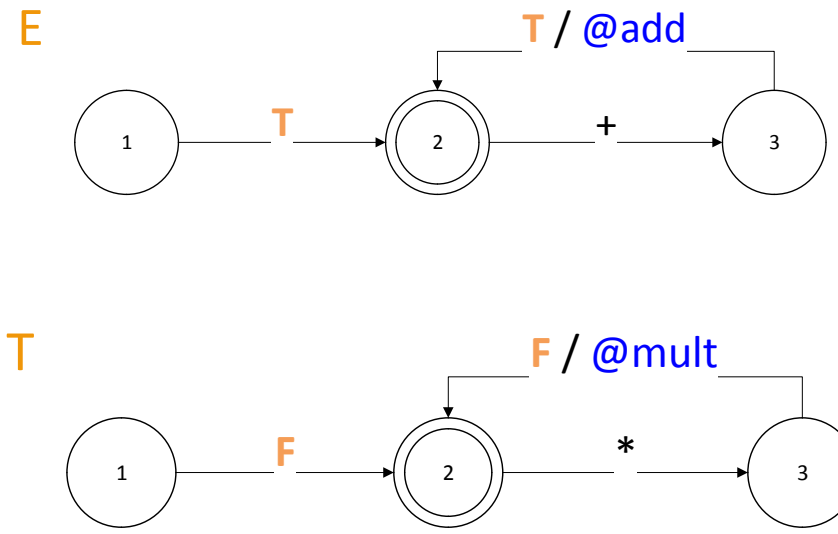
- Any idea?

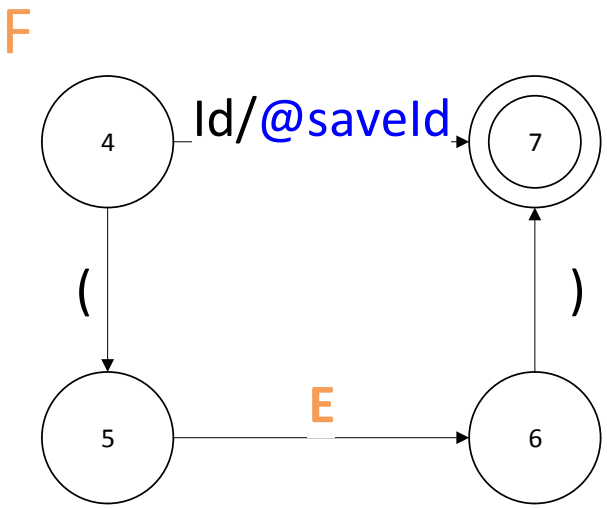
Syntax Graph

- On edges, we add semantic actions if necessary:

Syntax Graph

- On edges, we add semantic actions if necessary:
- Consider the graph of expression contains add.





LR-Parser

- Each production means a piece of semantic!
- So, By each reduction we need to communicate to Semantic Analyzer or perhaps code-generator.

1. **S** → **E**
2. **E** → **E** `+` **E**
3. **E** → **E** `*` **E**
4. **E** → `(` **E** `)`
5. **E** → `int`

LL(1) and RD Parsers

- We use semantic actions inside grammar rules, as pointers to routines:

1. E → T E’
2. E’ → + T E’
3. | ε
4. T → F T’
5. T’ → * F T’
6. | ε
7. F → id

LL(1) and RD Parsers

- We use semantic actions inside grammar rules, as pointers to routines:
- Obviously, in RD Parser you can call them when you need.

1. E → T E’
2. E’ → + T **@add** E’
3. | ε
4. T → F T’
5. T’ → * F @mult T’
6. | ε
7. F → @save id

id * id

1. E → T E’
2. E’ → + T @add E’
3. | ε
4. T → F T’
5. T’ → * F @mult T’
6. | ε
7. F → @save id

						*
			@save			F
		F	Id	Id		@mult
	T	T’	T’	T’	T’	T’
E	E’	E’	E’	E’	E’	E’
$	$	$	$	$	$	$

Syntax-Directed Translation

What is SDT?

- *Syntax-directed translation* refers to a method of compiler implementation where we use **parse tree** to direct **semantic analysis**.
- It can be a separated phase or do along parsing.

What is SDT?

- *Syntax-directed translation* refers to a method of compiler implementation where we use **parse tree** to direct **semantic analysis**.
- It can be a separated phase or do along parsing.
- This method can be used for IR-Generation, Type-checking and also implementing small languages (hope we can talk about it more later!).

What is SDT?

- *Syntax-directed translation* refers to a method of compiler implementation where we use **parse tree** to direct **semantic analysis**.
- It can be a separated phase or do along parsing.
- This method can be used for IR-Generation, Type-checking and also implementing small languages (hope we can talk about it more later!).
- So we need an **SDD (?)**.

SDT: example

- Consider the expr:

$$expr \rightarrow expr_1 + term$$

- The translation is:
  1. Translate $expr_1$
  2. Translate term
  3. Handle +

- So, in order to make an expr, we should translate first expr and then the term.
- Then we can handle the addition.

Syntax **D**irected **D**efinition

Syntax **D**irected **D**efinition

- We augment the grammar with information (rules and attributes), which helps us in semantic analysis.

Syntax **D**irected **D**efinition

- We augment the grammar with information (rules and attributes), which helps us in semantic analysis.
- This is done via non-terminals.

Syntax **D**irected **D**efinition

- We augment the grammar with information (rules and attributes), which helps us in semantic analysis.
- This is done via non-terminals.
- Each of program construct (**symbols**) is associated with some quantity we call, attributes.

Syntax **D**irected **D**efinition

- We augment the grammar with information (rules and attributes), which helps us in semantic analysis.
- This is done via non-terminals.
- Each of program construct (**symbols**) is associated with some quantity we call, attributes.
- They can have a name and value: a string, a number, a type, a memory location and etc.

Syntax **D**irected **D**efinition

- We augment the grammar with information (rules and attributes), which helps us in semantic analysis.
- This is done via non-terminals.
- Each of program construct (**symbols**) is associated with some quantity we call, attributes.
- They can have a name and value: a string, a number, a type, a memory location and etc.
- SDT is an SDD with explicitly specified the order of evaluation of semantic rules.

SDD: example 1

- Consider the expr:

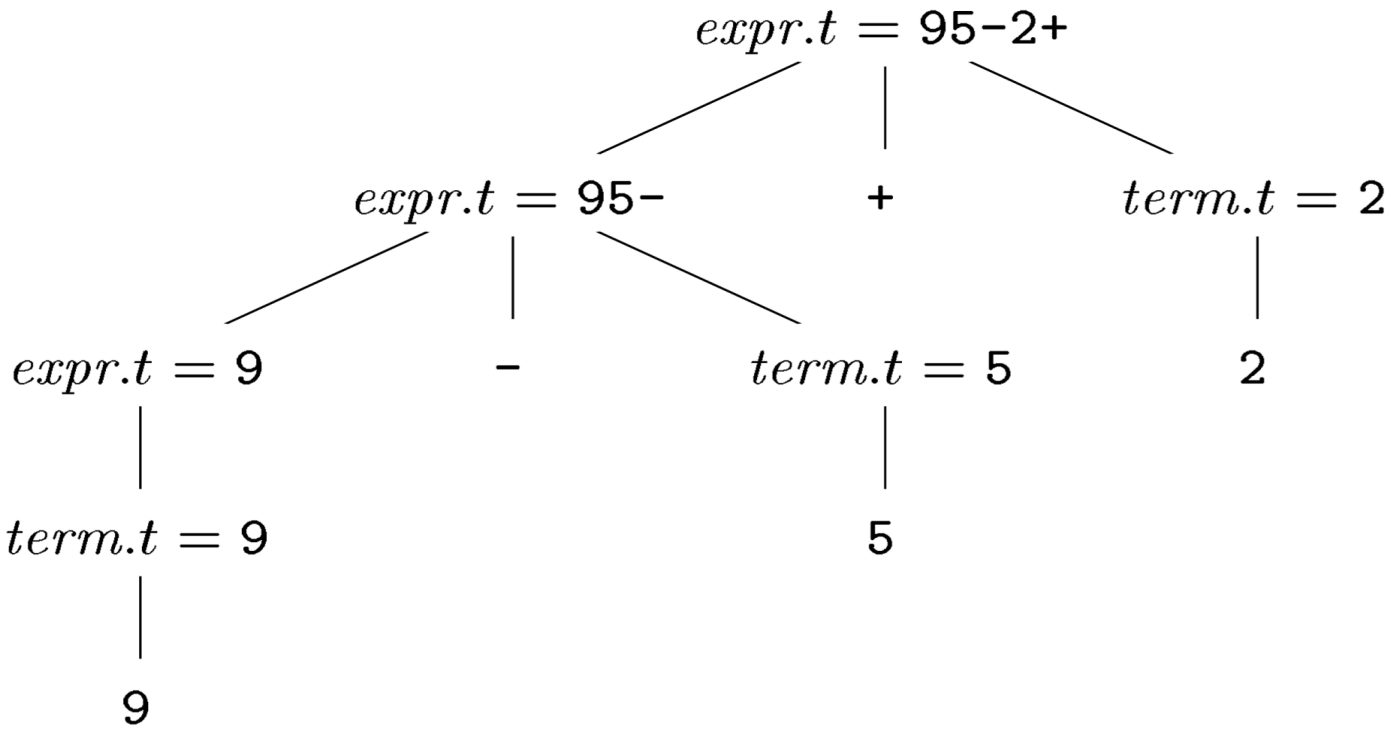
$$expr \rightarrow expr_1 + term$$

- The target is post-order so the translation is:
  1. Translate $expr_1$
  2. Translate term
  3. Handle +
- So we have the semantic rule:

$$expr\,.code = expr_1\,.code \parallel term\,.code \parallel +$$

SDD: example 1

- Consider single digit expr:



SDD: example 2

- Consider the following grammar:

$$INT \rightarrow INT\ DIGIT \mid DIGIT$$
$$DIGIT \rightarrow 0 \mid 1 \ldots \mid 9$$

SDD: example 2

- Consider the following grammar:

$$INT \rightarrow INT\ DIGIT \mid DIGIT$$
$$DIGIT \rightarrow 0 \mid 1 \ldots \mid 9$$

- We add attribute ‘value’ to store the correct number:

SDD: example 2

- Consider the following grammar:

$$INT \rightarrow INT\ DIGIT \mid DIGIT$$
$$DIGIT \rightarrow 0 \mid 1 \ldots \mid 9$$

- We add attribute 'value' to store the correct number:

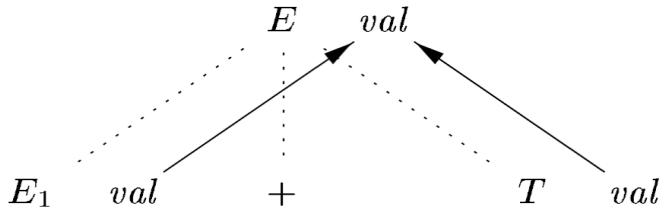
$$DIGIT \rightarrow 0\ \{DIGIT.value = 0\} \mid \ldots$$
$$INT \rightarrow DIGIT\ \{INT.value = DIGIT.value\}$$
$$INT \rightarrow \ldots \{INT.value = INT_1.value * 10 + DIGIT.value\}$$

SDD: example 2

- 435

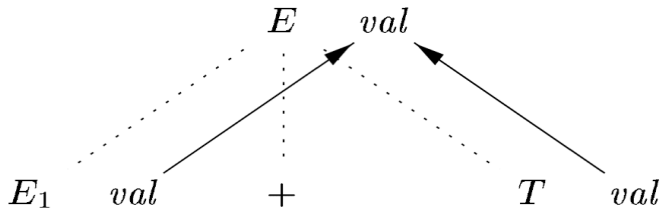
SDD vs SDT?

Dependency Graph



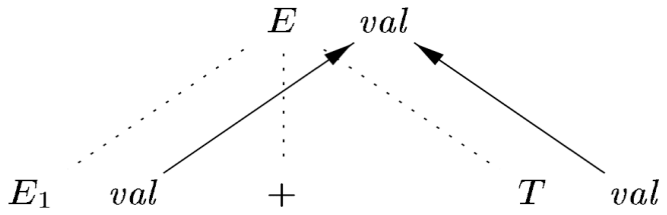
Dependency Graph

- Dependency Graph depict the flow of information among the attribute instances in a particular parse tree:



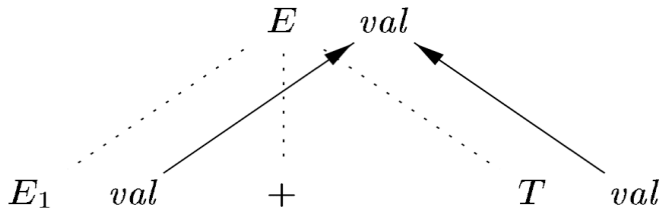
Dependency Graph

- Dependency Graph depict the flow of information among the attribute instances in a particular parse tree:
- Each node labeled with A in PT, DG has a node for each attribute associated with A.

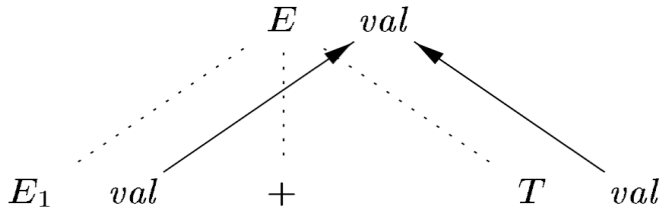


Dependency Graph

- Dependency Graph depict the flow of information among the attribute instances in a particular parse tree:
- Each node labeled with A in PT, DG has a node for each attribute associated with A.
- In a production p if $A \,.\, b = f(X \,.\, c)$ then we have a directed edge in DG from $X \,.\, c$ to $A \,.\, b$.

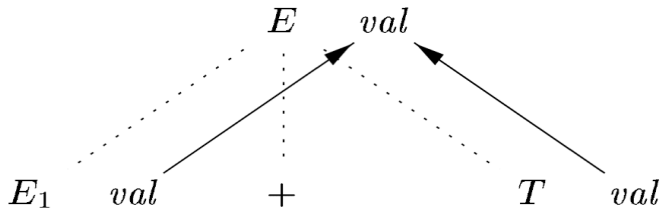


Dependency Graph



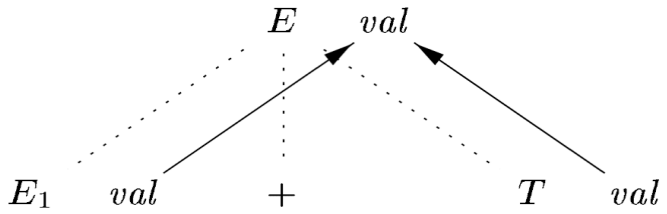
Dependency Graph

- Dependency Graph depict the flow of information among the attribute instances in a particular parse tree:



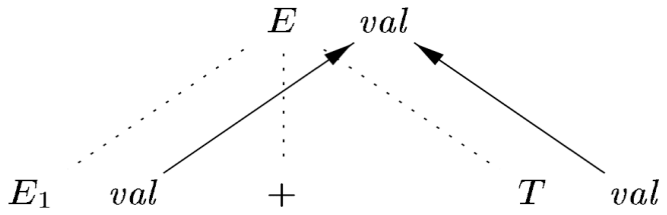
Dependency Graph

- Dependency Graph depict the flow of information among the attribute instances in a particular parse tree:
- Each node labeled with A in PT, DG has a node for each attribute associated with A.



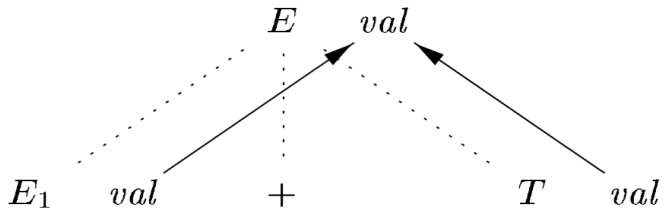
Dependency Graph

- Dependency Graph depict the flow of information among the attribute instances in a particular parse tree:
- Each node labeled with A in PT, DG has a node for each attribute associated with A.
- In a production p if $A \,.\, b = f(X \,.\, c)$ then we have a directed edge in DG from $X \,.\, c$ to $A \,.\, b$.



Dependency Graph

- Dependency Graph depict the flow of information among the attribute instances in a particular parse tree:
- Each node labeled with A in PT, DG has a node for each attribute associated with A.
- In a production p if $A.b = f(X.c)$ then we have a directed edge in DG from $X.c$ to $A.b$.
- In General topological sorting the DG give us an order for evaluation.

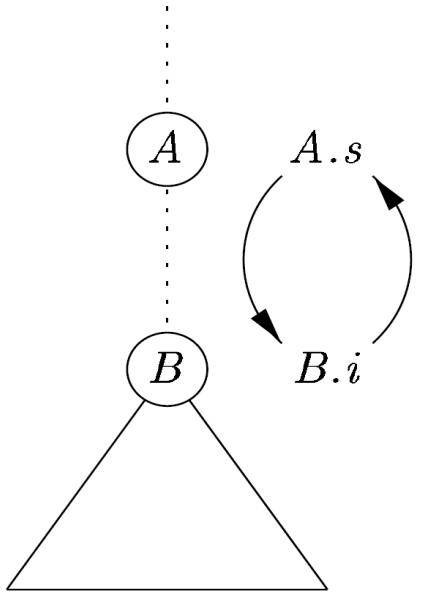


A Problem

- However, without any restriction on attribute’s code, some times it is not possible.

A Problem

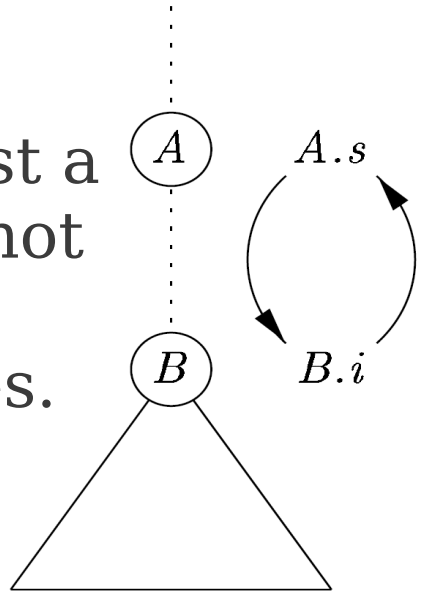
- However, without any restriction on attribute's code, some times it is not possible.



A Problem

- However, without any restriction on attribute's code, some times it is not possible.

- In fact, check whether there exist a parse tree which has a cycle or not is *hard*.
- So we classify useful SDD classes.



Classify Attributes

- A **synthesized attribute** for a non-terminal at a node in PT, is defined only in terms of attribute values of its **descendent** in PT.
  - i.e. at node N which contains $A \rightarrow X_1 \ldots X_n$, $A \,.\, s$ relies just on it's children or itself.
- An **inherited attribute** for a nonterminal B at a PT node is defined by a semantic rule associated with the production at its parent.
  - It must have B as a symbol in its body.
  - An inherited attribute at node N is defined only in terms of attribute values at N's **parent**, N **itself**, and N's **siblings**

Classify SDDs

- **S-Attributed**: An SDD is s-attributed every attribute is synthesized.
  - They can be evaluated in any bottom-up order.
  - Can be implemented during bottom-up parsing.

Classify SDDs

- **S-Attributed**: An SDD is s-attributed every attribute is synthesized.
  - They can be evaluated in any bottom-up order.
  - Can be implemented during bottom-up parsing.
- **L-Attributed**: each attribute is either:
  - Synthesized
  - Inherited but, in $A \rightarrow X_1 \ldots X_{i-1} X_i \ldots X_n$, each $X_i \,.\, inh$ may use:
    - Only inherited attributes associated with A
    - Inherited or synthesized attributes of X_j, $j < i$
    - X_i itself but without making any loop.

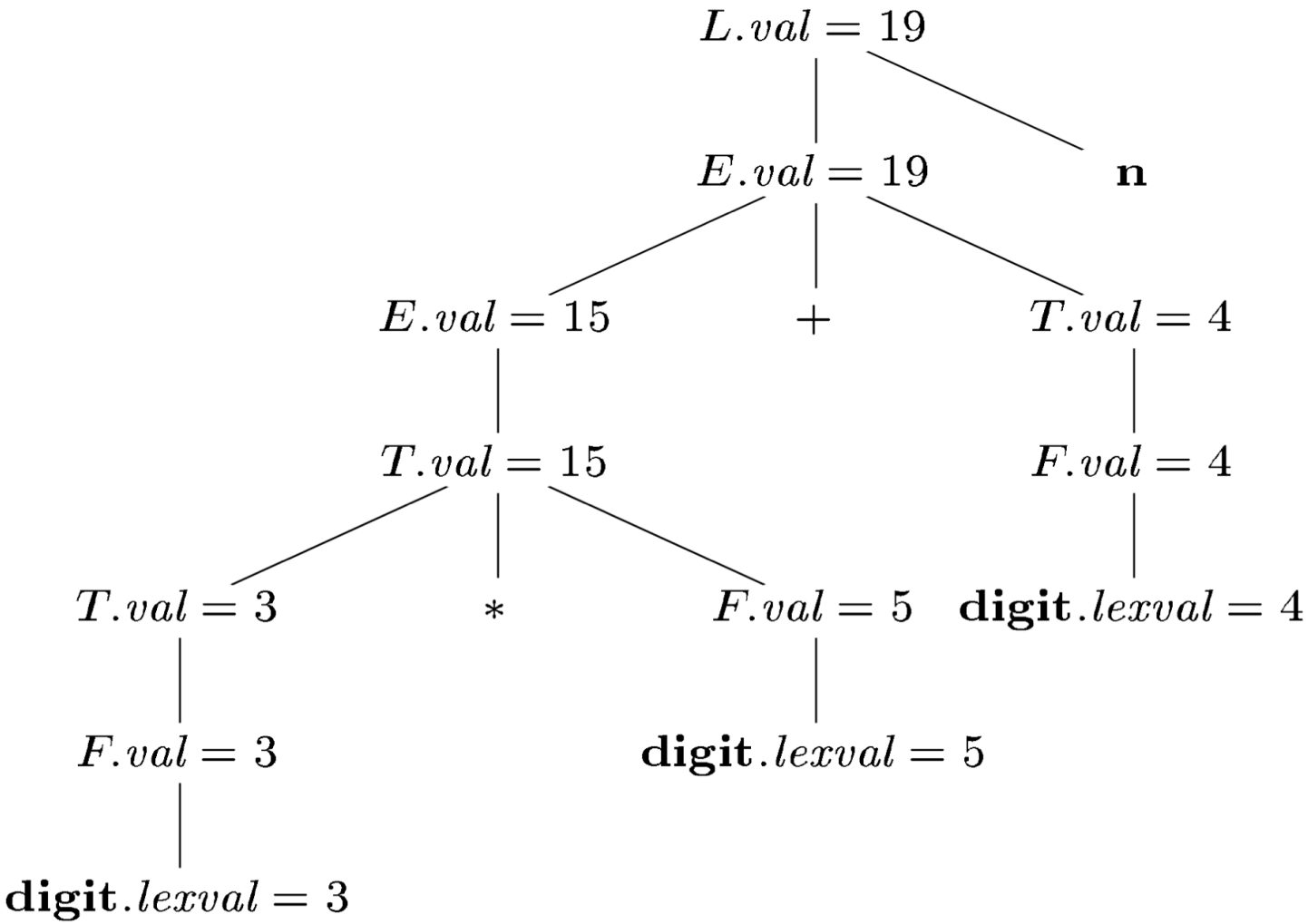
Example S-Attributed

- Consider the following SDD for calculation expr value:

	PRODUCTION	SEMANTIC RULES
1)	$L \to E$ **n**	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to$ **digit**	$F.val =$ **digit**.lexval

Annotated Pares Tree

3 * 5 + 4

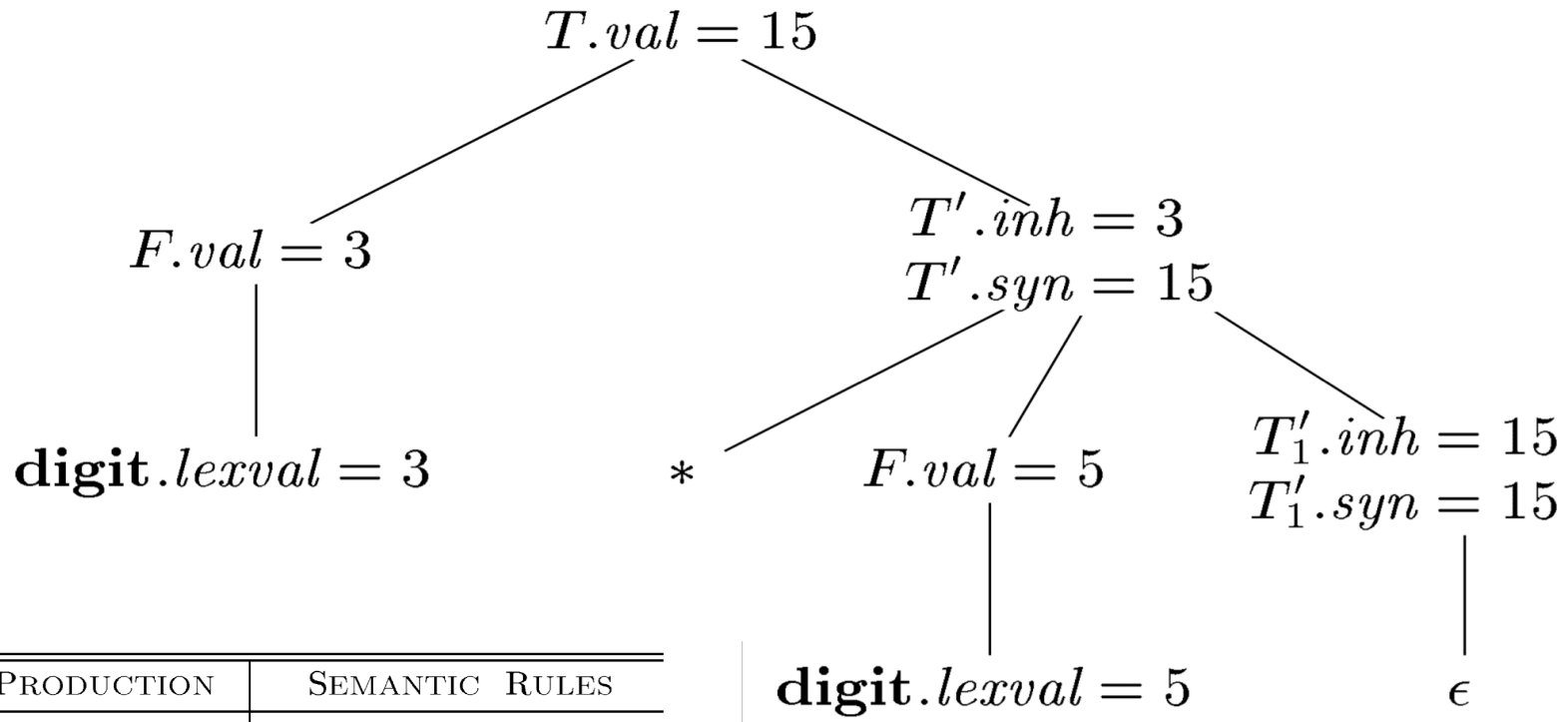


Example: L-Aattributed

- Again, expr.
- Note the difference:

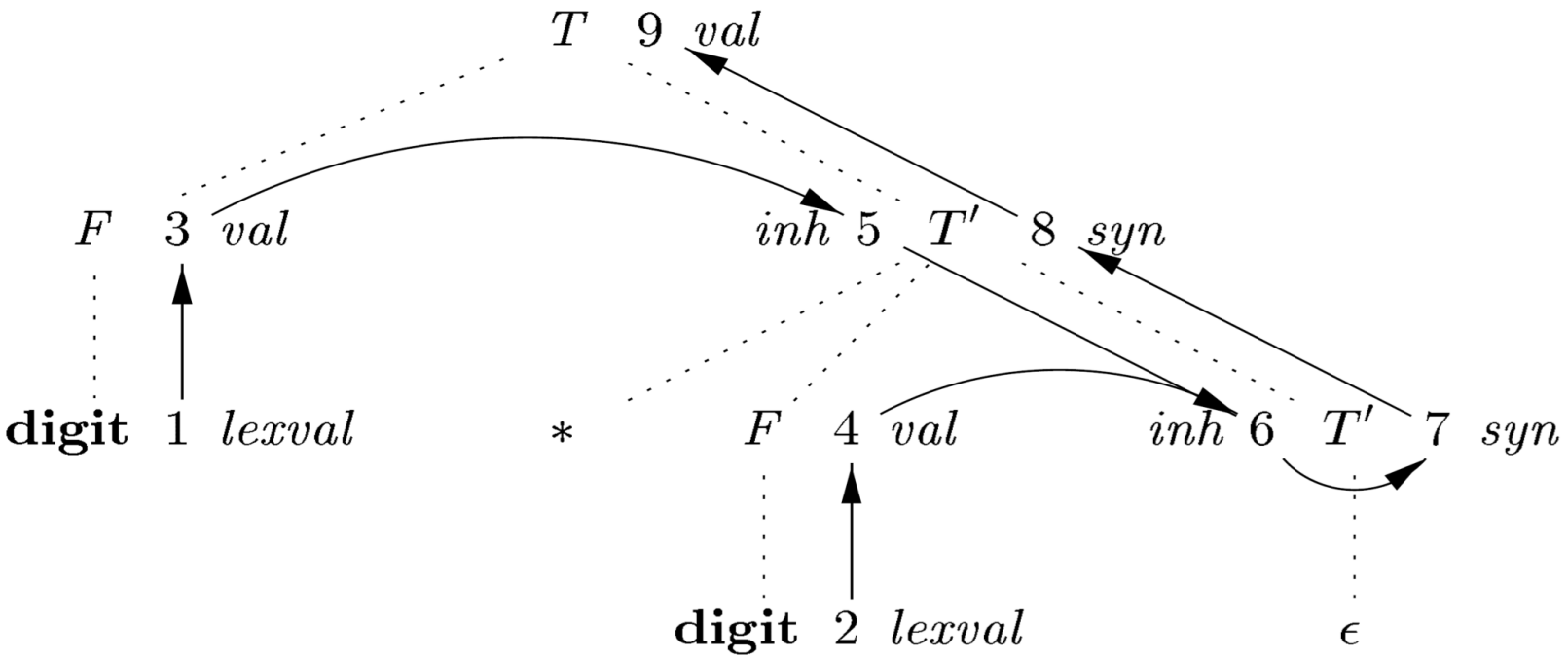
	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

Annotated Parse Tree



	Production	Semantic Rules
1)	$T \to F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \to *\ F\ T_1'$	$T_1'.inh = T'.inh \times F.val$ $T'.syn = T_1'.syn$
3)	$T' \to \epsilon$	$T'.syn = T'.inh$
4)	$F \to \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

Example: Dependency Graph

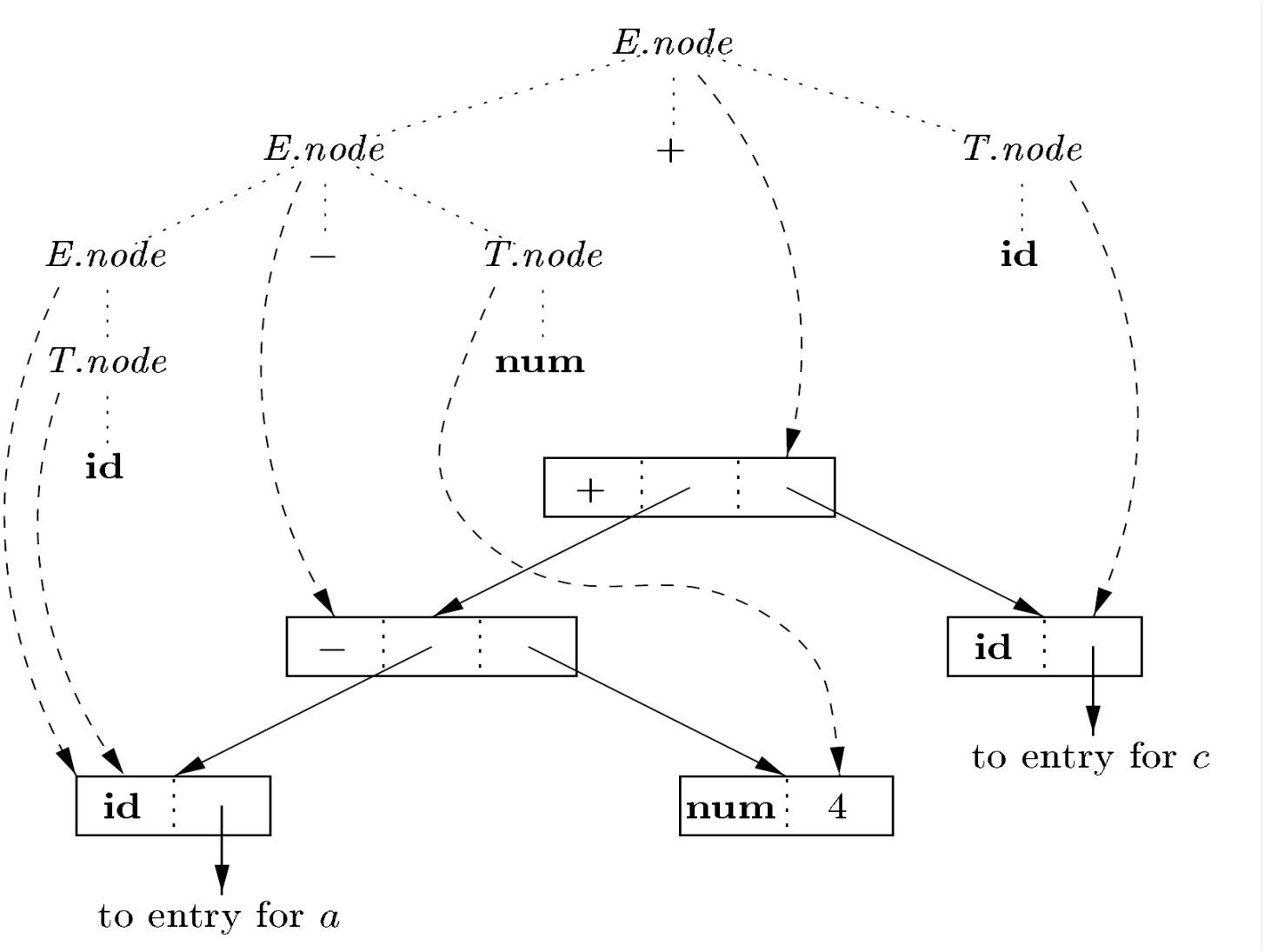


Application: Building Syntax Tree

- The following S-attributed definition construct syntax tree for simple expr grammars:

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (\ E\ )$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Syntax Tree


E.node
E.node
+
T.node
E.node
−
T.node
id
T.node
num
id
+
−
id
id
num
4
to entry for c
to entry for a

How to Implement SDD?

- We use Syntax-Directed Translation Schema to translate SDD.
- To do so:
  - Find parse tree without code fragments.
  - Then add code fragments.
  - At the end, by traverse the tree and run the code complete the translation.
- Typically, SDT's are implemented during parsing, without building a parse tree.

Syntax-Directed Translation Schemes

- A translation scheme is a notation for attaching *program fragments* to the productions of a grammar.

Syntax-Directed Translation Schemes

- A translation scheme is a notation for attaching *program fragments* to the productions of a grammar.
- The program fragments are **executed** when the production is used during syntax analysis.

Syntax-Directed Translation Schemes

- A translation scheme is a notation for attaching *program fragments* to the productions of a grammar.
- The program fragments are **executed** when the production is used during syntax analysis.
- These grammar embedded fragments calls *semantic actions*!

Syntax-Directed Translation Schemes

- A translation scheme is a notation for attaching *program fragments* to the productions of a grammar.
- The program fragments are **executed** when the production is used during syntax analysis.
- These grammar embedded fragments calls *semantic actions*!
- The combined result of all these fragment executions, produces the translation of the program.

Example: LR-Parser

- It can be implemented using a stack.
- The attribute(s) of each grammar symbol can be put on the stack in a so they can be found during the reduction.

$$
\begin{array}{lcll}
L & \rightarrow & E\ \mathbf{n} & \{\ \mathrm{print}(E.val);\ \} \\
E & \rightarrow & E_1 + T & \{\ E.val = E_1.val + T.val;\ \} \\
E & \rightarrow & T & \{\ E.val = T.val;\ \} \\
T & \rightarrow & T_1 * F & \{\ T.val = T_1.val \times F.val;\ \} \\
T & \rightarrow & F & \{\ T.val = F.val;\ \} \\
F & \rightarrow & (\ E\ ) & \{\ F.val = E.val;\ \} \\
F & \rightarrow & \mathbf{digit} & \{\ F.val = \mathbf{digit}.lexval;\ \}
\end{array}
$$

SDD with Action Inside Production

- Consider productions like:

$A \rightarrow X \{a\} Y$

SDD with Action Inside Production

- Consider productions like:

$A \rightarrow X \{a\} Y$

- In LR-parsers action a perform after find handle X or shift X.

SDD with Action Inside Production

- Consider productions like:

$A \rightarrow X \{a\} Y$

- In LR-parsers action a perform after find handle X or shift X.
- In Top-Down parsers, action a before expand Y or check Y on input.

SDD 2 SDT

- The semantic rules in an SDD can be converted into an SDT with actions that are executed at the right time.
- During parsing, an action in a production body is executed as soon as all the grammar symbols to the left of the action have been matched.
- What should we do with middle actions?
- For each middle action we add a *distinct* **marker nonterminal.** E.g. M_1. It has just one production $M_1 \to \epsilon$.

It can be problematic…

- Infix to prefix conversion.
- Using markers M_2 and M_4 for productions 2 and 4 respectively.
- A **digit** token face R/R conflict! (LR)
- Or print operator before they appear!

$$
\begin{array}{llll}
1) & L & \rightarrow & E\ \mathbf{n} \\
2) & E & \rightarrow & \{\ \mathrm{print}('+');\ \}\ \ E_1 + T \\
3) & E & \rightarrow & T \\
4) & T & \rightarrow & \{\ \mathrm{print}('*');\ \}\ \ T_1 * F \\
5) & T & \rightarrow & F \\
6) & F & \rightarrow & (\ E\ ) \\
7) & F & \rightarrow & \mathbf{digit}\ \ \{\ \mathrm{print}(\mathbf{digit}.\mathit{lexval});\ \}
\end{array}
$$

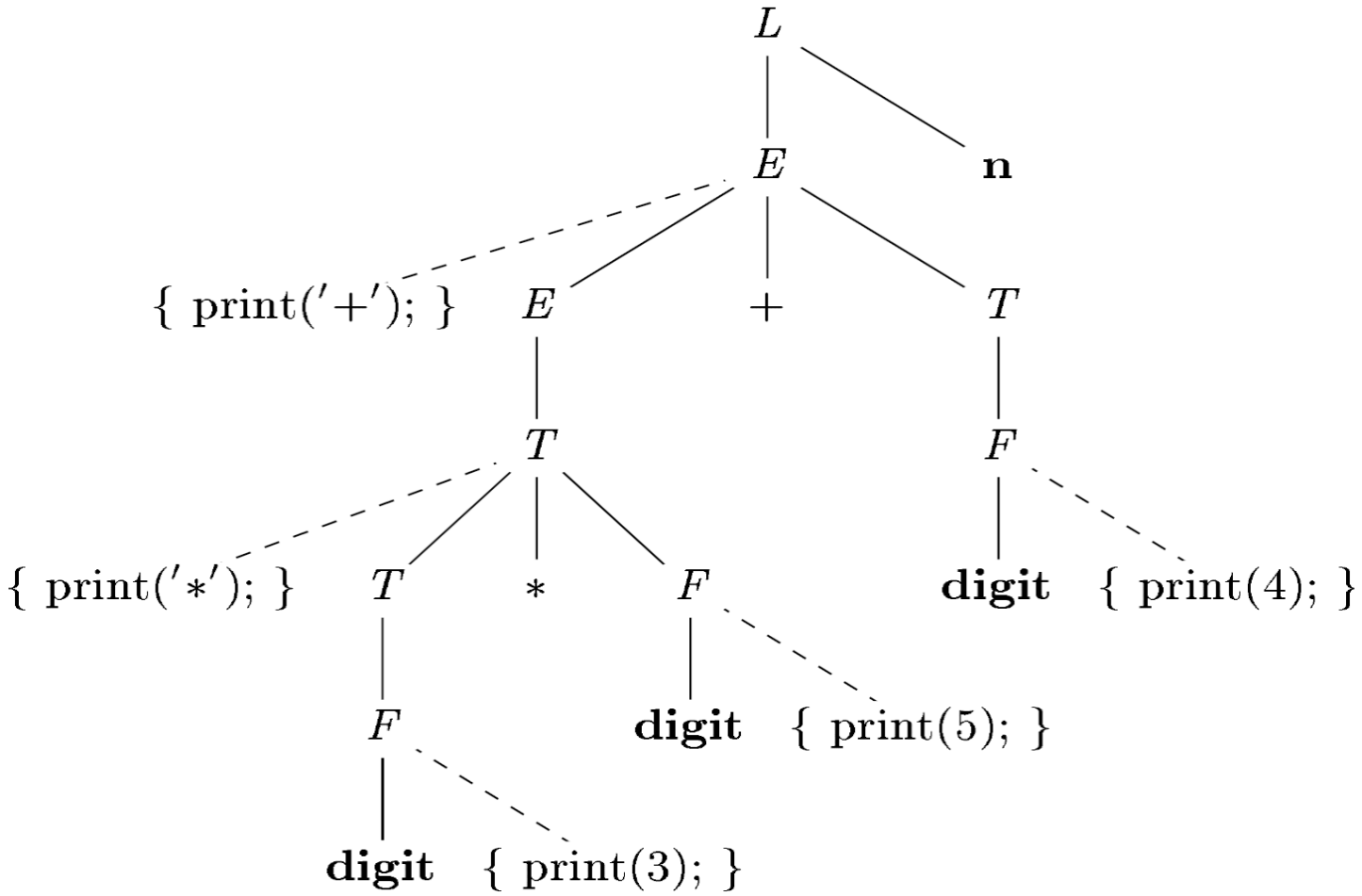
A General Solution

1. Ignoring the actions, parse the input and produce a parse tree as a result.

2. For each interior node N, production $A \rightarrow \alpha$: add additional children to N for the actions in α, so the children of N from left to right have exactly the symbols and actions of α.

3. Perform a **preorder** traversal, as soon as a node with action visited, perform the action.

Example


L
E
n
{ print('+'); }
E
+
T
T
F
{ print('*'); }
T
*
F
digit
{ print(4); }
F
digit
{ print(5); }
digit
{ print(3); }

SDT for L-Attributed

- The Rule is as follow:
- Embed the action that computes the **inherited attributes** for a nonterminal A immediately before that occurrence of A in the body of the production in order that those needed first are computed first.
- Place the actions that compute a synthesized attribute for the head of a production at the end of the body of that production

Example

- Loop:

$S \rightarrow \textbf{\textit{while}}\ (C)\ S_1$

- Here, S is a nonterminal generates all kinds of statements. C here is conditional statement.
- We use the following attributes:
- $S.next$: beginning of the code after S.
- $S.code$: code of loop body with jump at end.
- $C.true$: beginning of the code that must be executed if C is true.
- $C.false$: labels the beginning of the code that must be executed if C is false.
- $C.code$: the code of C with appropriate jumps.

Example: L-Attributed SDD

$S \rightarrow \textbf{while}\ (\ C\ )\ S_1$

$L1 = new();$
$L2 = new();$
$S_1.next = L1;$
$C.false = S.next;$
$C.true = L2;$
$S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code$

Example: L-Attributed SDD

$S \rightarrow \textbf{while}\ (\ C\ )\ S_1$

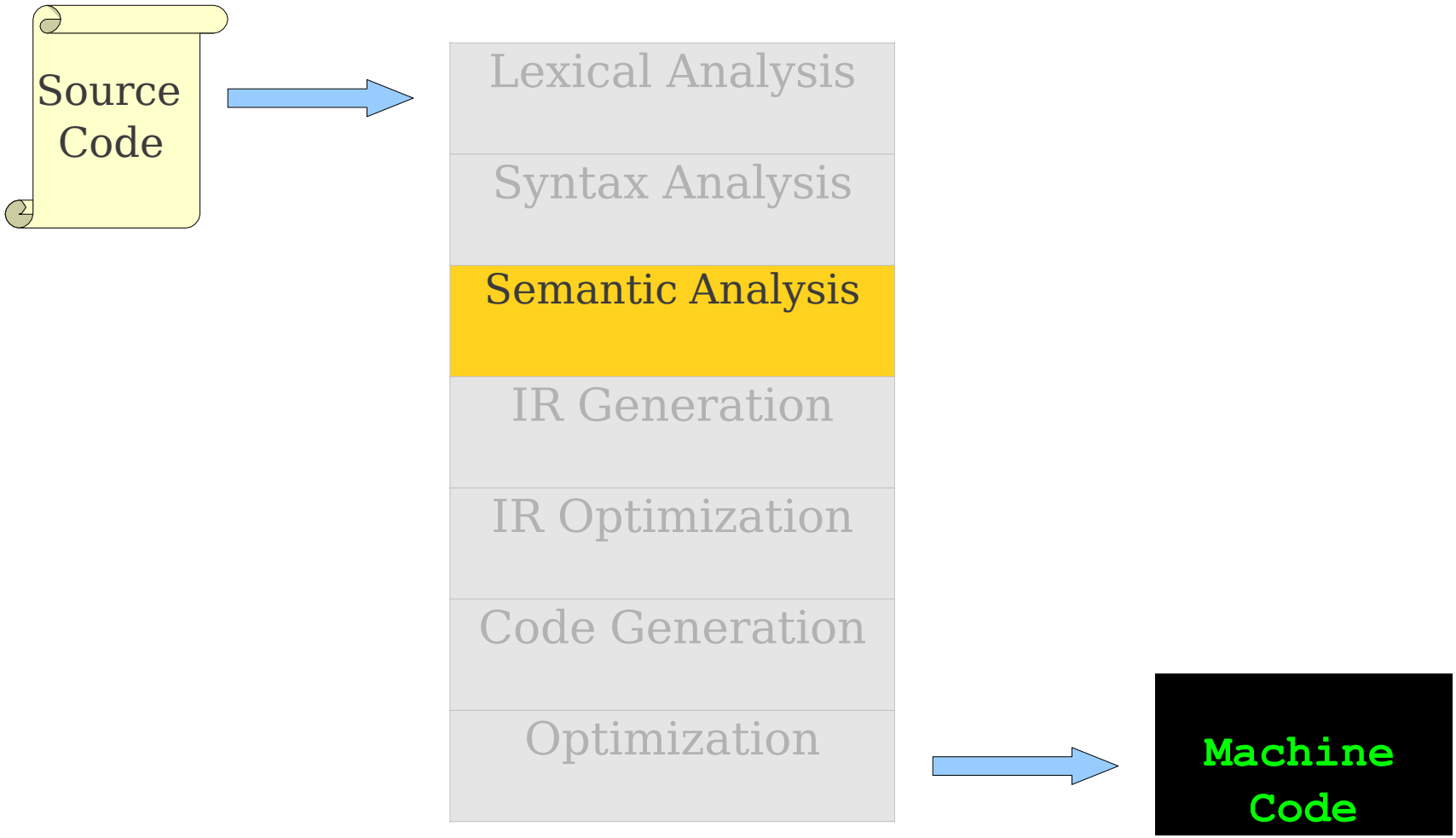
$L1 = new();$
$L2 = new();$
$S_1.next = L1;$
$C.false = S.next;$
$C.true = L2;$
$S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code$

$S \rightarrow \textbf{while}\ ($	$\{\ L1 = new();\ L2 = new();\ C.false = S.next;\ C.true = L2;\ \}$
$C\ )$	$\{\ S_1.next = L1;\ \}$
S_1	$\{\ S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code;\ \}$

بسم الله الرحمن الرحيم

Semantic Analysis, Scope

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

What's in a Name?

- The same name in a program may refer to fundamentally different things:
- This is perfectly legal Java code:

```
public class A {
     char A;
      A A(A A) {
           A.A = 'A';
           return A((A)
           A);
      }
}
```

What's in a Name?

- The same name in a program may refer to completely different objects:
- This is perfectly legal C++ code:

```
int Awful() {
    int x = 137;
    {
        string x = "Scope!"
        if (float x = 0)
            double x = x;
    }
    if (x == 137) cout << "Y";
}
```

What's in a Name?

- The same name in a program may refer to completely different objects:
- This is perfectly legal C++ code:

```
int Awful() {
    int x = 137;
    {
        string x = "Scope!"
        if (float x = 0)
            double x = x;
    }
    if (x == 137) cout << "Y";
}
```

Scope

- The **scope** of an entity is the set of locations in a program where that entity's name refers to that entity.
- The introduction of new variables into scope may hide older variables.
- How do we keep track of what's visible?

What else exist in Symbol Table?

- **Any information** about a name stores in Symbol Table data structure.
- Usually, We collect them during analysis phases.
- Its entries contain information about: identifiers (their lexemes), type, relative address.
- It also contains: Field description (for records), Number of arguments (for functions), bounds and dimensions (for arrays), Location of labels and pointed lines.

Symbol Tables

- A **symbol table** is a mapping from a name to the thing that name refers to.
- As we run our semantic analysis, continuously update the symbol table with information about what is in scope.
- Questions:
  - What does this look like in practice?
  - What operations need to be defined on it?
  - How do we implement it?

Symbol Tables: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x, y, z);
 4:   {
 5:     int x, z;
 6:     z = y;
 7:     x = z;
 8:     {
 9:       int y = x;
10:       {
11:         printf("%d,%d,%d\n", x, y, z);
12:       }
13:       printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```

Symbol Tables: The Intuition

Symbol Table

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x, y, z);
 4:   {
 5:     int x, z;
 6:     z = y;
 7:     x = z;
 8:     {
 9:       int y = x;
10:       {
11:         printf("%d,%d,%d\n", x, y, z);
12:       }
13:       printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```

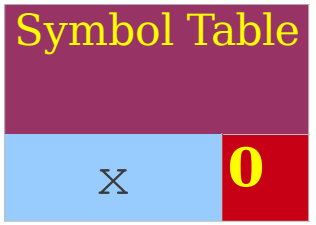
Symbol Tables: The Intuition

Symbol Table

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x, y, z);
 4:   {
 5:     int x, z;
 6:     z = y;
 7:     x = z;
 8:     {
 9:       int y = x;
10:       {
11:         printf("%d,%d,%d\n", x, y, z);
12:       }
13:       printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```

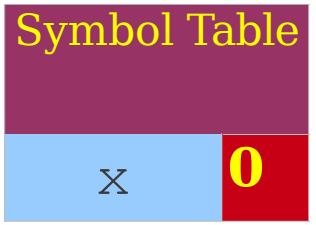
Symbol Tables: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x, y, z);
 4:   {
 5:     int x, z;
 6:     z = y;
 7:     x = z;
 8:     {
 9:       int y = x;
10:       {
11:         printf("%d,%d,%d\n", x, y, z);
12:       }
13:       printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```



Symbol Tables: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x, y, z);
 4:   {
 5:     int x, z;
 6:     z = y;
 7:     x = z;
 8:     {
 9:       int y = x;
10:       {
11:         printf("%d,%d,%d\n", x, y, z);
12:       }
13:       printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```



Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int     y)  {
 3:    printf("%d,%d,%d\n", x,     y, z);
 4:  {
 5:     int x, z;
 6:     z = y;
 7:     x = z;
 8:     {
 9:        int y = x;
10:        {
11:           printf("%d,%d,%d\n", x, y, z);
12:        }
13:           printf("%d,%d,%d\n", x, y, z);

14:        }
15:        printf("%d,%d,%d\n", x, y, z);
16:    }
17:  }
```

Symbol Table	
x	0
z	1

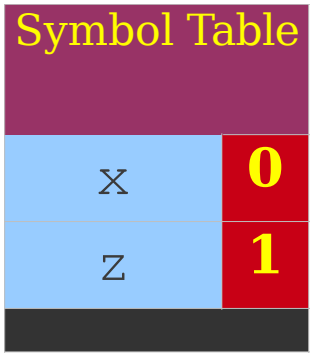
Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:    printf("%d,%d,%d\n", x, y, z);
 4:  {
 5:    int x, z;
 6:    z = y;
 7:    x = z;
 8:    {
 9:      int y = x;
10:      {
11:        printf("%d,%d,%d\n", x, y, z);
12:      }
13:         printf("%d,%d,%d\n", x, y, z);

14:      }
15:      printf("%d,%d,%d\n", x, y, z);
16:    }
17:  }
```

Symbol Table	
x	0
z	1

Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:    printf("%d,%d,%d\n", x, y, z);
 4: {
 5:    int x, z;
 6:    z = y;
 7:    x = z;
 8:    {
 9:      int y = x;
10:      {
11:        printf("%d,%d,%d\n", x, y, z);
12:      }
13:         printf("%d,%d,%d\n", x, y, z);

14:      }
15:      printf("%d,%d,%d\n", x, y, z);
16:    }
17:  }
```



Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:    printf("%d,%d,%d\n", x, y, z);
 4:  {
 5:     int x, z;
 6:     z = y;
 7:     x = z;
 8:     {
 9:        int y = x;
10:        {
11:           printf("%d,%d,%d\n", x, y, z);
12:        }
13:           printf("%d,%d,%d\n", x, y, z);

14:        }
15:        printf("%d,%d,%d\n", x, y, z);
16:    }
17: }
```

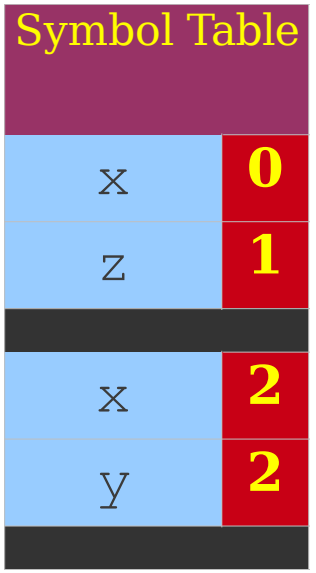
Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int      y)  {
 3:    printf("%d,%d,%d\n", x,      y, z);
 4:  {
 5:     int x, z;
 6:     z = y;
 7:     x = z;
 8:     {
 9:        int y = x;
10:        {
11:           printf("%d,%d,%d\n", x, y, z);
12:        }
13:           printf("%d,%d,%d\n", x, y, z);

14:        }
15:        printf("%d,%d,%d\n", x, y, z);
16:     }
17:  }
```

Symbol Table	
x	0
z	1
x	2
y	2

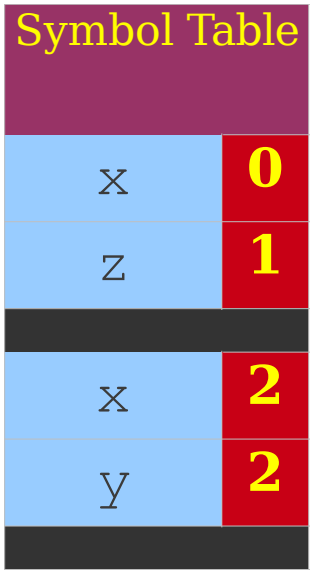
Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int      y)  {
 3:    printf("%d,%d,%d\n", x,      y, z);
 4:  {
 5:     int x, z;
 6:     z = y;
 7:     x = z;
 8:     {
 9:        int y = x;
10:        {
11:           printf("%d,%d,%d\n", x, y, z);
12:        }
13:           printf("%d,%d,%d\n", x, y, z);

14:        }
15:        printf("%d,%d,%d\n", x, y, z);
16:     }
17:  }
```



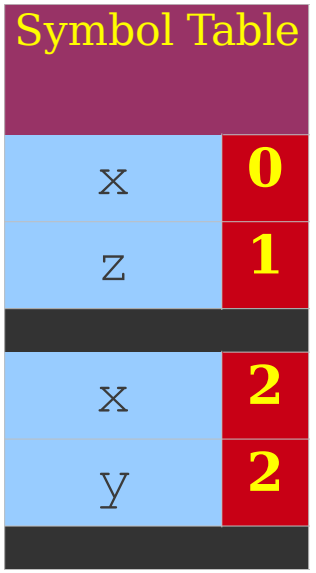
Symbol Tables: The Intuition

```
 0:   int x = 137;
 1:   int z = 42;
 2:   int MyFunction(int x, int y) {
 3:     printf("%d,%d,%d\n", x, y, z);
 4:   {
 5:       int x, z;
 6:       z = y;
 7:     { x = z;
 8:
 9:         int y = x;
10:         {
11:           printf("%d,%d,%d\n", x, y, z);
12:         }
13:          printf("%d,%d,%d\n", x, y, z);

14:       }
15:        printf("%d,%d,%d\n", x, y, z);
16:     }
17:  }
```



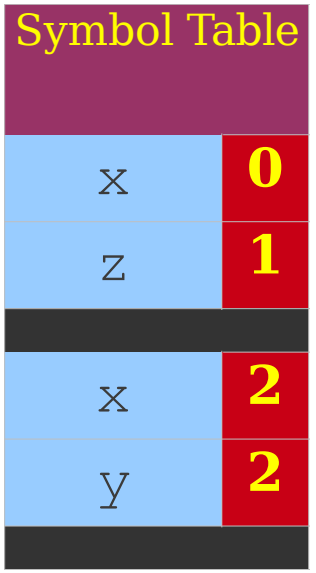
Symbol Tables: The Intuition

```
0:    int x = 137;
1:    int z = 42;
2:    int MyFunction(int x, int y) {
3:      printf("%d,%d,%d\n", x@2, y@2,  z@1);
4:    {
5:          int x, z;
6:          z = y;  x = z;
7:          {
8:            int y = x;
9:            {
10:             printf("%d,%d,%d\n", x, y, z);
11:           }
12:           printf("%d,%d,%d\n", x, y, z);
13:         }
14:         printf("%d,%d,%d\n", x, y, z);
15:     }
16: }
17:
```



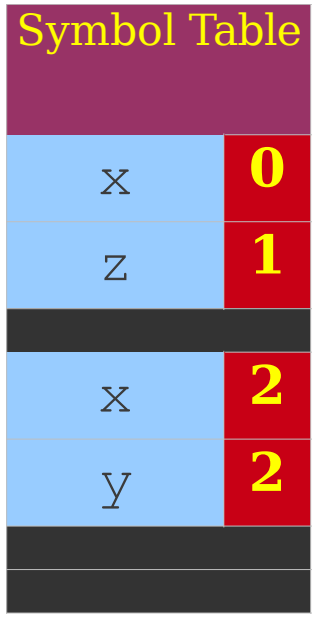
Symbol Tables: The Intuition

```
0:    int x = 137;
1:    int z = 42;
2:    int MyFunction(int x, int y) {
3:      printf("%d,%d,%d\n", x@2, y@2,  z@1);
4:    {
 5:         int x, z;
 6:         z = y;  x = z;
 7:         {
 8:           int y = x;
 9:           {
10:             printf("%d,%d,%d\n", x, y, z);
11:           }
12:           printf("%d,%d,%d\n", x, y, z);
13:         }
14:         printf("%d,%d,%d\n", x, y, z);
15:     }
16: }
17:
```



Symbol Tables: The Intuition

```
0:    int x = 137;
1:    int z = 42;
2:    int MyFunction(int x, int y) {
3:      printf("%d,%d,%d\n", x@2, y@2,  z@1);
4:    {
 5:         int x, z;
 6:         z = y;  x = z;
 7:         {
 8:           int y = x;
 9:           {
10:             printf("%d,%d,%d\n", x, y, z);
11:           }
12:           printf("%d,%d,%d\n", x, y, z);
13:         }
14:         printf("%d,%d,%d\n", x, y, z);
15:     }
16: }
17:
```



Symbol Tables: The Intuition

```
0:    int x = 137;
1:    int z = 42;
2:    int MyFunction(int x, int y) {
3:      printf("%d,%d,%d\n", x@2, y@2, z@1);
4:    {
5:        int x, z;
6:        z = y;  x = z;
7:        {
8:          int y = x;
9:          {
10:           printf("%d,%d,%d\n", x, y, z);
11:         }
12:         printf("%d,%d,%d\n", x, y, z);
13:       }
14:       printf("%d,%d,%d\n", x, y, z);
15:   }
16: }
17:
```

Symbol Tables: The Intuition

```
0:    int x = 137;
1:    int z = 42;
2:    int MyFunction(int x, int y) {
3:      printf("%d,%d,%d\n", x@2, y@2, z@1);
4:    {
5:        int x, z;
6:        z = y;  x = z;
7:        {
8:          int y = x;
9:          {
10:           printf("%d,%d,%d\n", x, y, z);
11:         }
12:         printf("%d,%d,%d\n", x, y, z);
13:       }
14:       printf("%d,%d,%d\n", x, y, z);
15:   }
16: }
17:
```

Symbol Tables: The Intuition

```
0:    int x = 137;
1:    int z = 42;
2:    int MyFunction(int x, int y) {
3:      printf("%d,%d,%d\n", x@2, y@2,  z@1);
4:    {
5:          int x, z;
6:          z = y;
7:          x = z;
8:          {
9:             int y = x;
10:            {
11:               printf("%d,%d,%d\n", x, y, z);
12:            }
13:            printf("%d,%d,%d\n", x, y, z);
14:         }
15:     }  printf("%d,%d,%d\n", x, y, z);
16:   }
17:
```

Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:    printf("%d,%d,%d\n", x@2, y@2,  z@1);
 4:    {
 5:      int x, z;
 6:      z@5 = y@2;
 7:      x = z;
 8:      {
 9:        int y = x;
10:        {
11:          printf("%d,%d,%d\n", x, y, z);
12:        }
13:         printf("%d,%d,%d\n", x, y, z);
14:       }
15:      printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```

Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:     printf("%d,%d,%d\n", x@2, y@2,          z@1
 4:     {
 5:        int x, z;
 6:        z@5 = y@2;
 7:        x = z;
 8:        {
 9:           int y = x;
10:           {
11:              printf("%d,%d,%d\n", x, y, z);
12:           }
13:            printf("%d,%d,%d\n", x, y, z);
14:         }
15:        printf("%d,%d,%d\n", x, y, z);
16:     }
17:  }
```

Symbol Tables: The Intuition

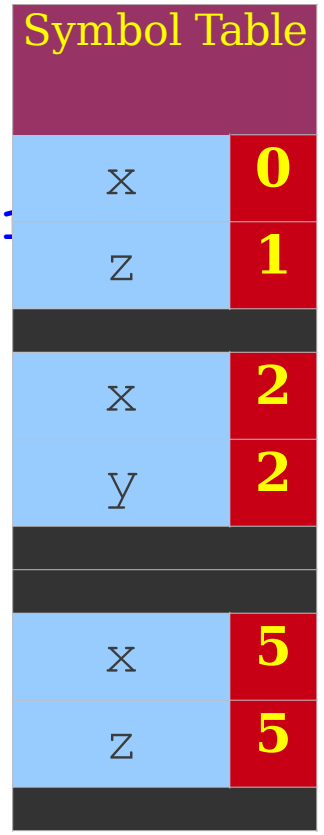
```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:    printf("%d,%d,%d\n", x@2, y@2,          z@1
 4:    {
 5:      int x, z;
 6:      z@5 = y@2;
 7:      x@5 = z@5;
 8:      {
 9:        int y = x;
10:        {
11:          printf("%d,%d,%d\n", x, y, z);
12:        }
13:         printf("%d,%d,%d\n", x, y, z);
14:       }
15:       printf("%d,%d,%d\n", x, y, z);
16:    }
17: }
```

Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:    printf("%d,%d,%d\n", x@2, y@2,          z@1
 4:  {
 5:      int x, z;
 6:      z@5 = y@2;
 7:      x@5 = z@5;
 8:      {
 9:        int y = x;
10:        {
11:          printf("%d,%d,%d\n", x, y, z);
12:        }
13:         printf("%d,%d,%d\n", x, y, z);
14:       }
15:      printf("%d,%d,%d\n", x, y, z);
16:    }
17:  }
```

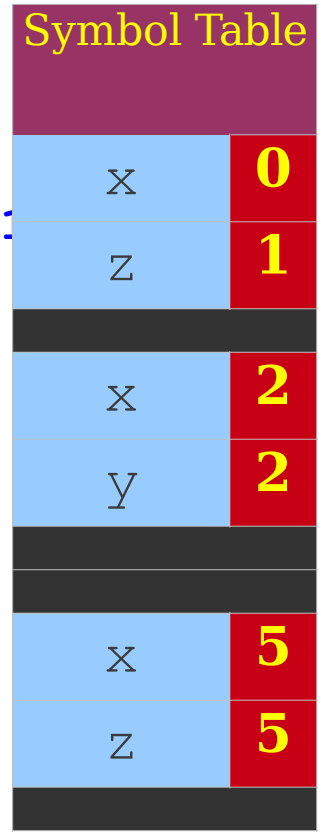
Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:    printf("%d,%d,%d\n", x@2, y@2,
 4:  {
 5:      int x, z;
 6:      z@5 = y@2;
 7:      x@5 = z@5;
 8:      {
 9:        int y = x;
10:        {
11:          printf("%d,%d,%d\n", x, y, z);
12:        }
13:         printf("%d,%d,%d\n", x, y, z);
14:       }
15:      printf("%d,%d,%d\n", x, y, z);
16:    }
17:  }
```



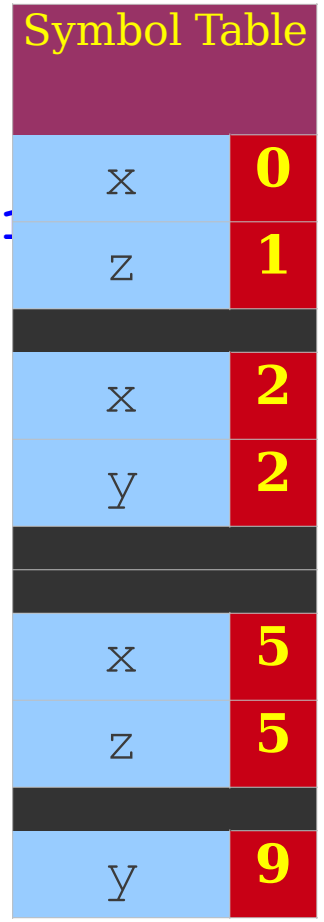
Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:    printf("%d,%d,%d\n", x@2, y@2,          z@1
 4:  {
 5:      int x, z;
 6:      z@5 = y@2;
 7:      x@5 = z@5;
 8: {
 9:        int y = x;
10: {

11:          printf("%d,%d,%d\n", x, y, z);

12: }

13:          printf("%d,%d,%d\n", x, y, z);
14:      }
```



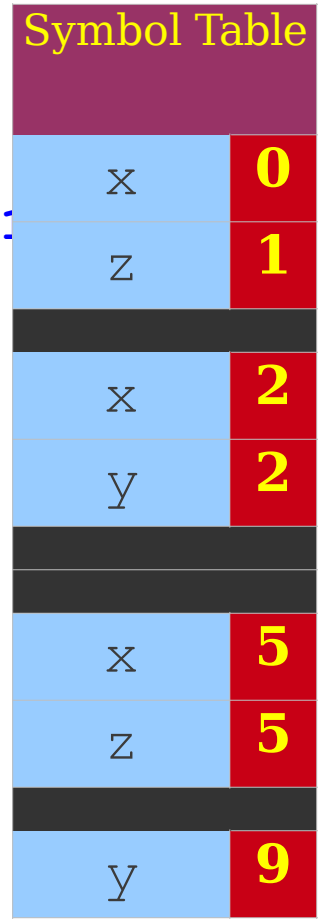
Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:    printf("%d,%d,%d\n", x@2, y@2,          z@1
 4:  {
 5:      int x, z;
 6:      z@5 = y@2;
 7:      x@5 = z@5;
 8: {
 9:        int y = x;
10: {

11:          printf("%d,%d,%d\n", x, y, z);

12: }

13:          printf("%d,%d,%d\n", x, y, z);
14:      }
```



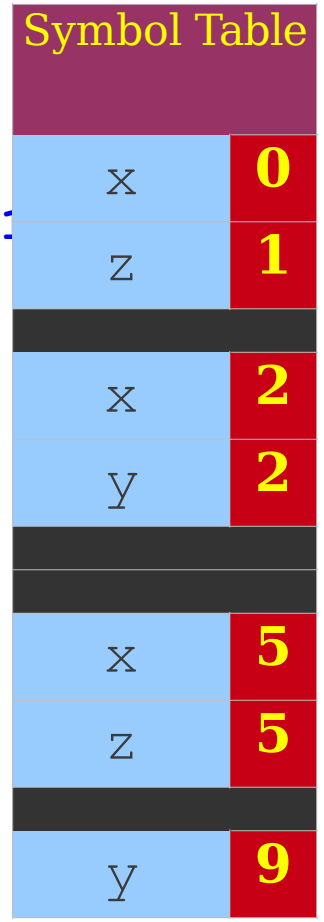
Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:    printf("%d,%d,%d\n", x@2, y@2,          z@1
 4:  {
 5:      int x, z;
 6:      z@5 = y@2;
 7:      x@5 = z@5;
 8: {
 9:        int y = x@5;
10: {

11:          printf("%d,%d,%d\n", x, y, z);

12: }

13:          printf("%d,%d,%d\n", x, y, z);
14:      }
```



Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:    printf("%d,%d,%d\n", x@2, y@2,
 4:  {
 5:      int x, z;
 6:      z@5 = y@2;
 7:      x@5 = z@5;
 8: {
 9:        int y = x@5;
10: {

11:          printf("%d,%d,%d\n", x, y, z);

12: }

13:          printf("%d,%d,%d\n", x, y, z);
14:      }
```



Symbol Tables: The Intuition

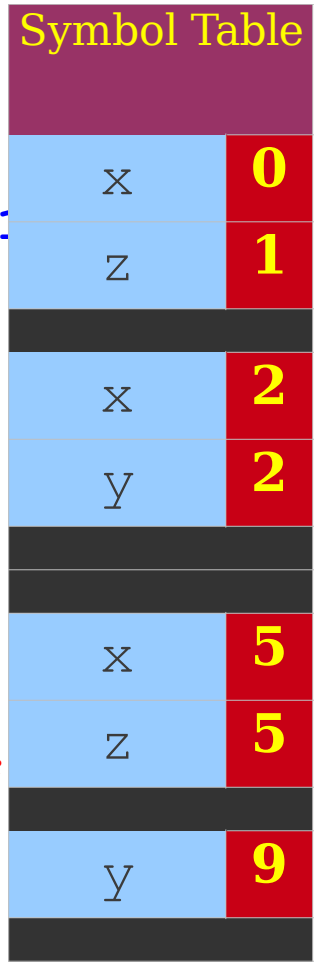
```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:    printf("%d,%d,%d\n", x@2, y@2,
 4:  {
 5:      int x, z;
 6:      z@5 = y@2;
 7:      x@5 = z@5;
 8: {
 9:        int y = x@5;
10: {

11:          printf("%d,%d,%d\n", x, y, z);

12: }

13:          printf("%d,%d,%d\n", x, y, z);
14:      }
```

Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:    printf("%d,%d,%d\n", x@2, y@2,
 4:  {
 5:      int x, z;
 6:      z@5 = y@2;
 7:      x@5 = z@5;
 8: {
 9:        int y = x@5;
10: {

11:          printf("%d,%d,%d\n", x, y, z);

12: }


13:          printf("%d,%d,%d\n", x, y, z);
14:       }
```

Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:    printf("%d,%d,%d\n", x@2, y@2,          z@1
 4:  {
 5:      int x, z;
 6:      z@5 = y@2;
 7:      x@5 = z@5;
 8: {
 9:        int y = x@5;
10: {

11:          printf("%d,%d,%d\n", x@5, y@9, z@5);

12: }

13:          printf("%d,%d,%d\n", x, y, z);
14:       }
```

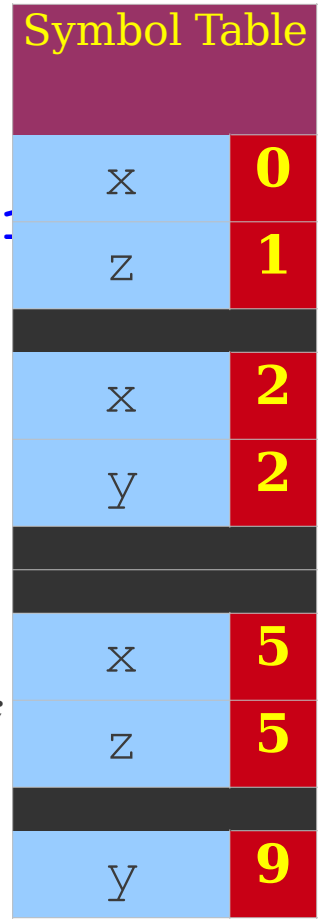


Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:    printf("%d,%d,%d\n", x@2, y@2,          z@1
 4:  {
 5:      int x, z;
 6:      z@5 = y@2;
 7:      x@5 = z@5;
 8: {
 9:        int y = x@5;
10: {

11:          printf("%d,%d,%d\n", x@5, y@9, z@5);

12: }


13:          printf("%d,%d,%d\n", x, y, z);
14:       }
```

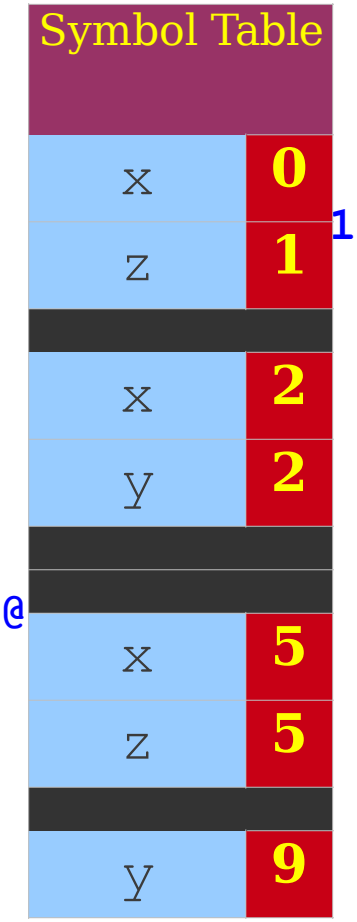
Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:    printf("%d,%d,%d\n", x@2, y@2,          z@1
 4:  {
 5:      int x, z;
 6:      z@5 = y@2;
 7:      x@5 = z@5;
 8: {
 9:        int y = x@5;
10: {
11:          printf("%d,%d,%d\n", x@5, y@9, z@5);
12: }
13:         printf("%d,%d,%d\n", x, y, z);
14:       }
15:      printf("%d,%d,%d\n", x, y, z);
16:    }
17:  }
```



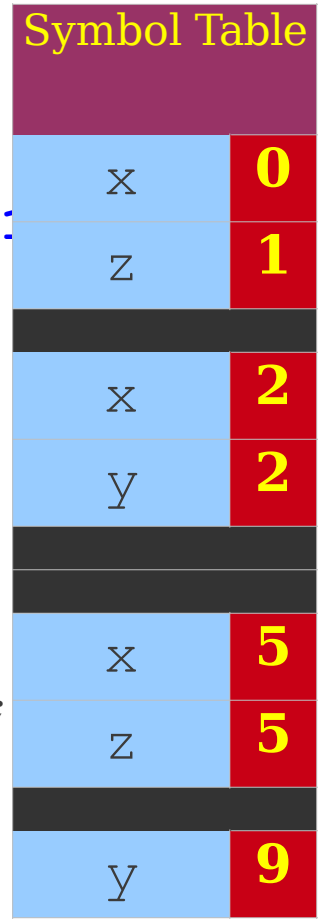
Symbol Tables: The Intuition

```
0:    int x = 137;
1:    int z = 42;
2:    int MyFunction(int x, int y) {
3:      printf("%d,%d,%d\n", x@2, y@2,
4:    {                                                    1);
5:          int x, z;
6:          z@5 = y@2;   x@5 = z@5;
7:          {
8:            int y = x@5;
9:            {
10:             printf("%d,%d,%d\n", x@5, y@9, z@
11:           }
12:           printf("%d,%d,%d\n", x, y, z);
13:         }
14:         printf("%d,%d,%d\n", x, y, z);
15:     }
16: }
17:
```



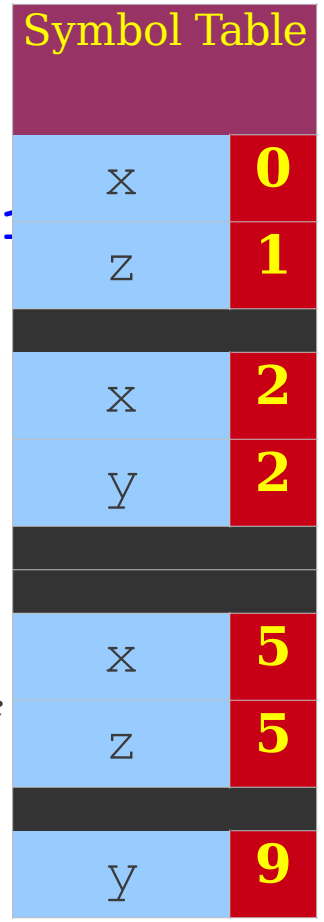
Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:    printf("%d,%d,%d\n", x@2, y@2,          z@1
 4:  {
 5:      int x, z;
 6:      z@5 = y@2;
 7:      x@5 = z@5;
 8: {
 9:        int y = x@5;
10: {
11:          printf("%d,%d,%d\n", x@5, y@9, z@5);
12: }
13:         printf("%d,%d,%d\n", x@5, y@9, z@5);
14:       }
15:       printf("%d,%d,%d\n", x, y, z);
16:    }
17:  }
```



Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:    printf("%d,%d,%d\n", x@2, y@2,          z@1
 4:   {
 5:       int x, z;
 6:       z@5 = y@2;
 7:       x@5 = z@5;
 8: {
 9:         int y = x@5;
10: {
11:           printf("%d,%d,%d\n", x@5, y@9, z@5);
12: }
13:          printf("%d,%d,%d\n", x@5, y@9, z@5);
14:        }
15:       printf("%d,%d,%d\n", x, y, z);
16:    }
17:  }
```



Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:    printf("%d,%d,%d\n", x@2, y@2,          z@1
 4:   {
 5:       int x, z;
 6:       z@5 = y@2;
 7:       x@5 = z@5;
 8: {
 9:         int y = x@5;
10: {
11:           printf("%d,%d,%d\n", x@5, y@9, z@5);
12: }
13:          printf("%d,%d,%d\n", x@5, y@9, z@5);
14:        }
15:       printf("%d,%d,%d\n", x, y, z);
16:    }
17:  }
```

Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:    printf("%d,%d,%d\n", x@2, y@2,          z@1
 4:  {
 5:      int x, z;
 6:      z@5 = y@2;
 7:      x@5 = z@5;
 8: {
 9:        int y = x@5;
10: {
11:          printf("%d,%d,%d\n", x@5, y@9, z@5);
12: }
13:         printf("%d,%d,%d\n", x@5, y@9, z@5);
14:       }
15:      printf("%d,%d,%d\n", x, y, z);
16:    }
17:  }
```

Symbol Tables: The Intuition

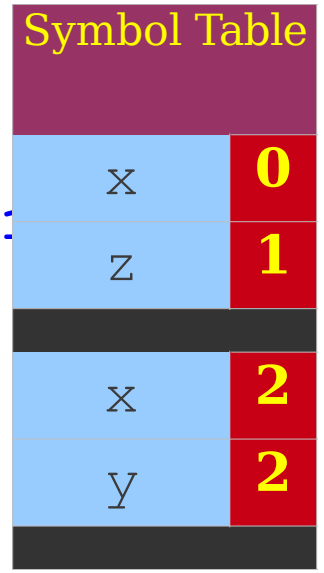
```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:    printf("%d,%d,%d\n", x@2, y@2,          z@1
 4:  {
 5:      int x, z;
 6:      z@5 = y@2;
 7:      x@5 = z@5;
 8: {
 9:        int y = x@5;
10: {
11:          printf("%d,%d,%d\n", x@5, y@9, z@5);
12: }
13:         printf("%d,%d,%d\n", x@5, y@9, z@5);
14:       }
15:      printf("%d,%d,%d\n", x@5, y@2, z@5);
16:    }
17:  }
```

Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:    printf("%d,%d,%d\n", x@2, y@2,          z@1
 4:   {
 5:       int x, z;
 6:       z@5 = y@2;
 7:       x@5 = z@5;
 8: {
 9:         int y = x@5;
10: {
11:           printf("%d,%d,%d\n", x@5, y@9, z@5);
12: }
13:          printf("%d,%d,%d\n", x@5, y@9, z@5);
14:        }
15:       printf("%d,%d,%d\n", x@5, y@2, z@5);
16:    }
17:  }
```

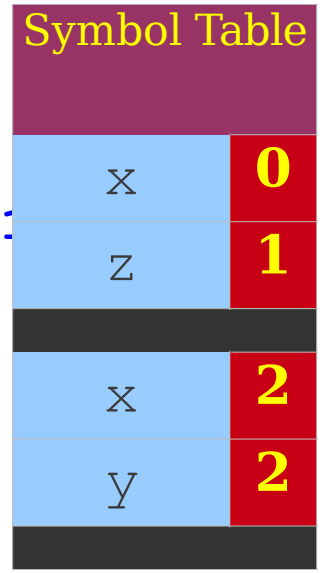
Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:    printf("%d,%d,%d\n", x@2, y@2,
 4:   {
 5:       int x, z;
 6:       z@5 = y@2;
 7:       x@5 = z@5;
 8: {
 9:         int y = x@5;
10: {
11:           printf("%d,%d,%d\n", x@5, y@9, z@5);
12: }
13:          printf("%d,%d,%d\n", x@5, y@9, z@5);
14:        }
15:       printf("%d,%d,%d\n", x@5, y@2, z@5);
16:    }
17:  }
```



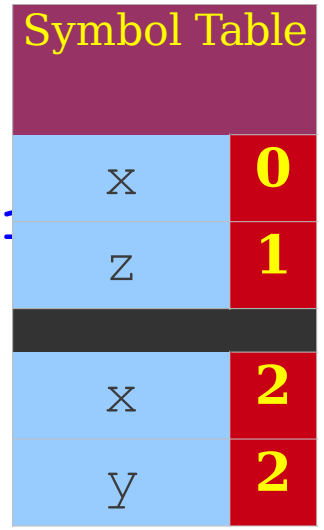
Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:    printf("%d,%d,%d\n", x@2, y@2,          z@1
 4:   {
 5:       int x, z;
 6:       z@5 = y@2;
 7:       x@5 = z@5;
 8: {
 9:         int y = x@5;
10: {
11:           printf("%d,%d,%d\n", x@5, y@9, z@5);
12: }
13:          printf("%d,%d,%d\n", x@5, y@9, z@5);
14:        }
15:       printf("%d,%d,%d\n", x@5, y@2, z@5);
16:    }
17: }
```



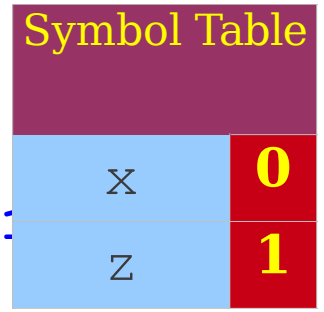
Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:    printf("%d,%d,%d\n", x@2, y@2,          z@1
 4:   {
 5:       int x, z;
 6:       z@5 = y@2;
 7:       x@5 = z@5;
 8: {
 9:         int y = x@5;
10: {
11:           printf("%d,%d,%d\n", x@5, y@9, z@5);
12: }
13:          printf("%d,%d,%d\n", x@5, y@9, z@5);
14:        }
15:       printf("%d,%d,%d\n", x@5, y@2, z@5);
16:    }
17: }
```



Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:    printf("%d,%d,%d\n", x@2, y@2,          z@1
 4:   {
 5:       int x, z;
 6:       z@5 = y@2;
 7:       x@5 = z@5;
 8: {
 9:         int y = x@5;
10: {
11:           printf("%d,%d,%d\n", x@5, y@9, z@5);
12: }
13:          printf("%d,%d,%d\n", x@5, y@9, z@5);
14:        }
15:       printf("%d,%d,%d\n", x@5, y@2, z@5);
16:    }
17: }
```



Symbol Tables: The Intuition

```
 0:  int x = 137;
 1:  int z = 42;
 2:  int MyFunction(int x, int y) {
 3:    printf("%d,%d,%d\n", x@2, y@2, z@1);
 4:  {
 5:      int x, z;
 6:      z@5 = y@2;
 7:      x@5 = z@5;
 8: {
 9:        int y = x@5;
10: {
11:          printf("%d,%d,%d\n", x@5, y@9, z@5);
12: }
13:         printf("%d,%d,%d\n", x@5, y@9, z@5);
14:       }
15:       printf("%d,%d,%d\n", x@5, y@2, z@5);
16:    }
17:  }
```

Symbol Table	
x	0
z	1

Symbol Table Operations

- Typically implemented as a **stack of maps**. Each map corresponds to a particular scope.
- Stack allows for easy “enter” and “exit” operations.
- Symbol table operations are
  - **Push scope**: Enter a new scope.
  - **Pop scope**: Leave a scope, discarding all declarations in it.
  - **Insert symbol**: Add a new entry to the current scope.
  - **Lookup symbol**: Find what a name corresponds to.

Using a Symbol Table

- To process a portion of the program that creates a scope (block statements, function calls, classes, etc.)
  - Enter a new scope.
  - Add all variable declarations to the symbol table.
  - Process the body of the block/function/class.
  - Exit the scope.
- Much of semantic analysis is defined in terms of recursive AST traversals like this.

Another view: Spaghetti Stacks

- Treat the symbol table as a linked structure of scopes.
- Each scope stores a pointer to its parents, but not vice-versa.
- From any point in the program, symbol table appears to be a stack.
- This is called a **spaghetti stack**.

Another View of Symbol Tables

Another View of Symbol Tables

```
 0: int x;
 1: int y;
 2: int MyFunction(int x, int y)
 3: {
 4:   int w, z;
 5:   {
 6:     int y;
 7:   }
 8:   {
 9:     int w;
10:   }
11: }
```

Another View of Symbol Tables

Root Scope

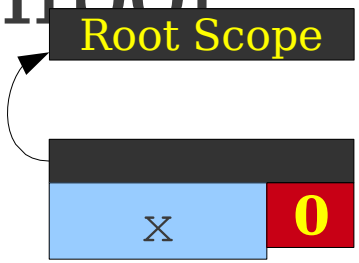
```
 0: int x;
 1: int y;
 2: int MyFunction(int x, int y)
 3: {
 4:   int w, z;
 5:   {
 6:     int y;
 7:   }
 8:   {
 9:     int w;
10:   }
11: }
```

Another View of Symbol Tables

Root Scope

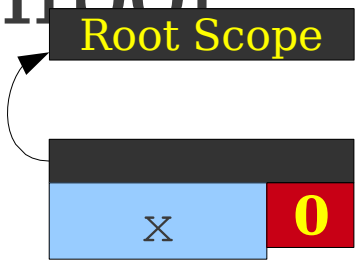
```
 0: int x;
 1: int y;
 2: int MyFunction(int x, int y)
 3: {
 4:   int w, z;
 5:   {
 6:     int y;
 7:   }
 8:   {
 9:     int w;
10:   }
11: }
```

Another View of Symbol Tables




```
0: int x;
1: int y;
2: int MyFunction(int x, int y)
3: {
 4:       int w, z;
 5:       {

  6:             int y;
  7:     }
  8:     {
  9:             int w;
 10:     }

 11: }
```

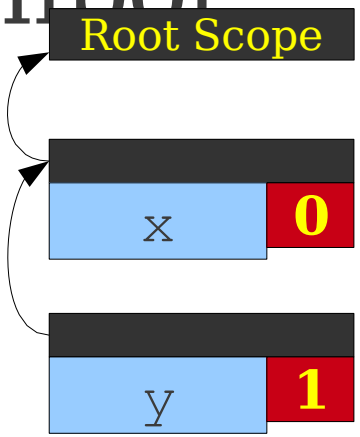
Another View of Symbol Tables




```
0: int x;
1: int y;
2: int MyFunction(int x, int y)
3: {
 4:       int w, z;
 5:       {

  6:             int y;
  7:     }
  8:     {
  9:             int w;
 10:     }

 11: }
```

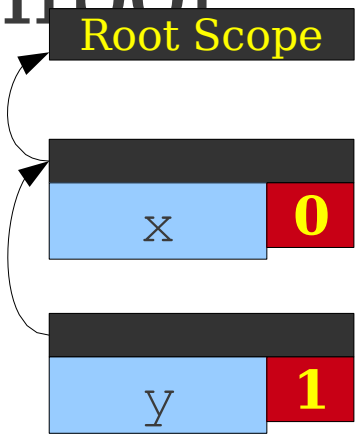
Another View of Symbol Tables

```
 0: int x;
 1: int y;
 2: int MyFunction(int x, int y)
 3: {
 4:   int w, z;
 5:   {
 6:     int y;
 7:   }
 8:   {
 9:     int w;
10:   }
11: }
```



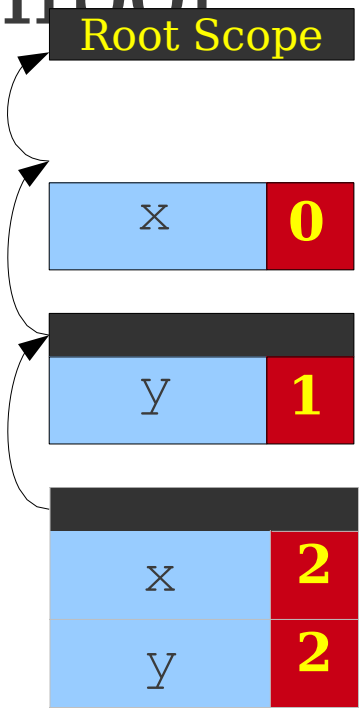
Another View of Symbol Tables

```
 0:   int x;
 1:   int y;
 2:   int MyFunction(int x, int y)
 3:   {
 4:      int w, z;
 5:      {
 6:         int y;
 7:      }
 8:      {
 9:         int w;
10:      }
11:   }
```



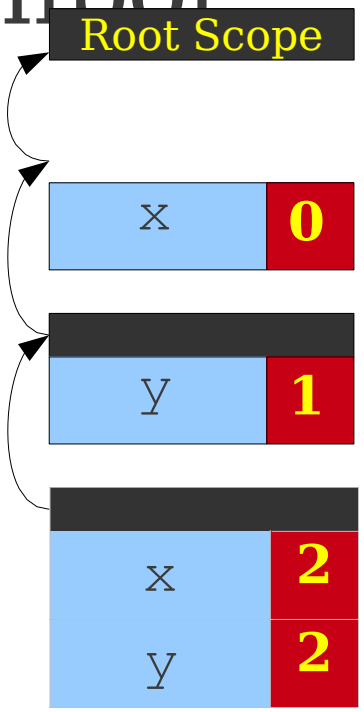
Another View of Symbol Tables

```
 0: int x;
 1: int y;
 2: int MyFunction(int x, int y)
 3: {
 4:   int w, z;
 5:   {
 6:     int y;
 7:   }
 8:   {
 9:     int w;
10:   }
11: }
```



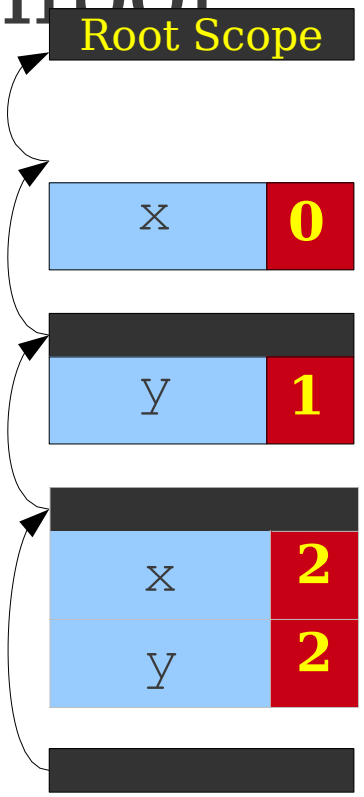
Another View of Symbol Tables

```
 0: int x;
 1: int y;
 2: int MyFunction(int x, int y)
 3: {
 4:   int w, z;
 5:   {
 6:     int y;
 7:   }
 8:   {
 9:     int w;
10:   }
11: }
```



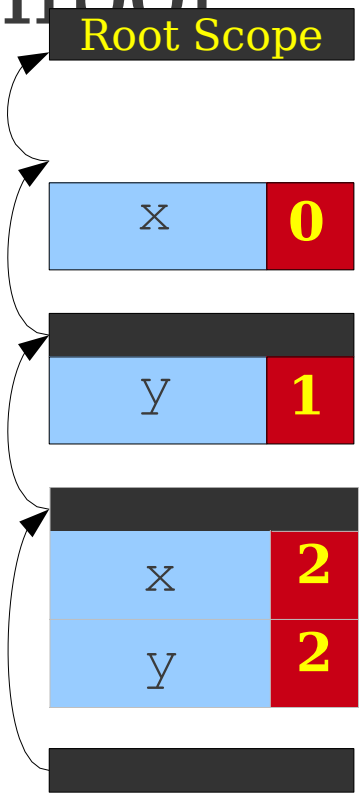
Another View of Symbol Tables

```
 0: int x;
 1: int y;
 2: int MyFunction(int x, int y)
 3: {
 4:   int w, z;
 5:   {
 6:     int y;
 7:   }
 8:   {
 9:     int w;
10:   }
11: }
```



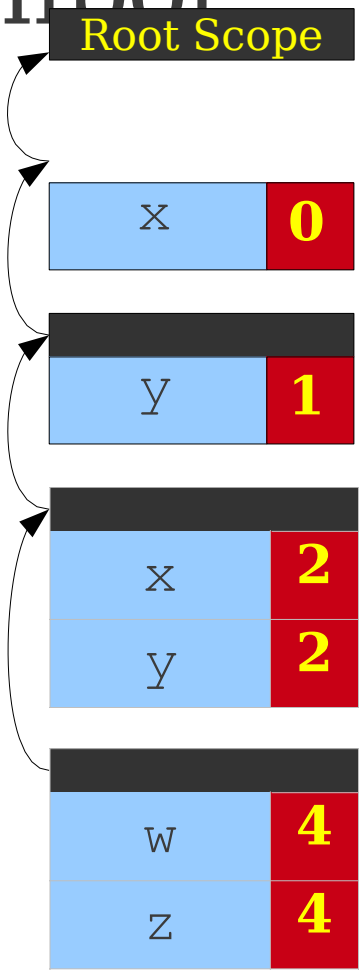
Another View of Symbol Tables

```
 0: int x;
 1: int y;
 2: int MyFunction(int x, int y)
 3: {
 4:   int w, z;
 5:   {
 6:     int y;
 7:   }
 8:   {
 9:     int w;
10:   }
11: }
```



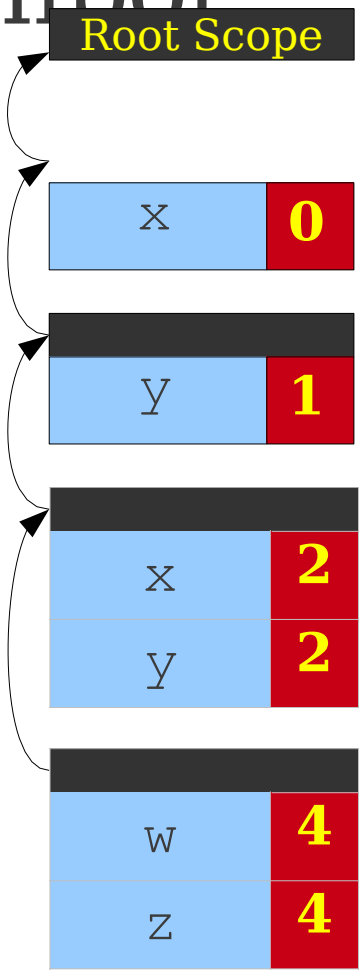
Another View of Symbol Tables

```
 0: int x;
 1: int y;
 2: int MyFunction(int x, int y)
 3: {
 4:   int w, z;
 5:   {
 6:     int y;
 7:   }
 8:   {
 9:     int w;
10:   }
11: }
```



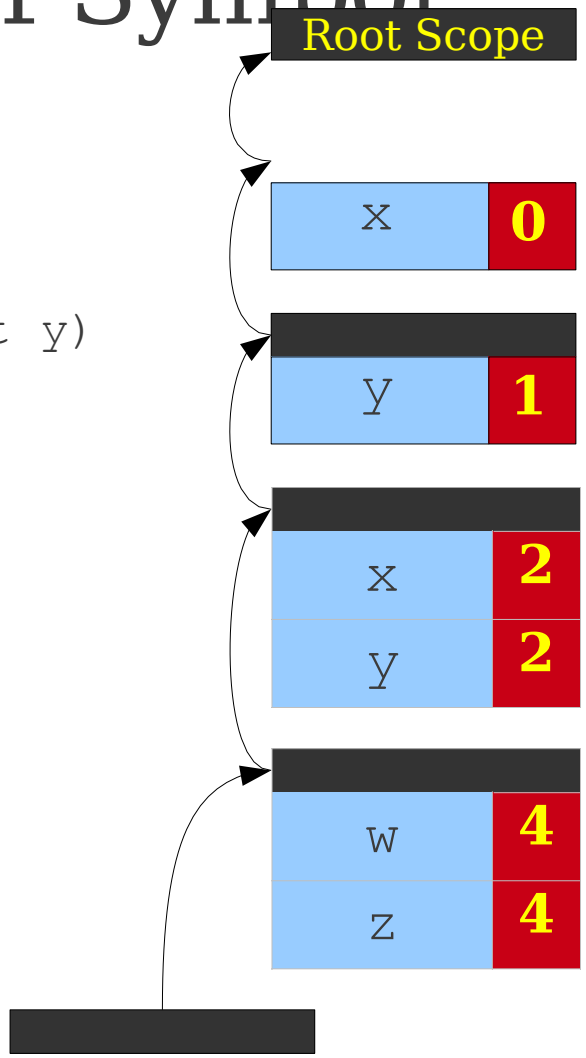
Another View of Symbol Tables

```
 0: int x;
 1: int y;
 2: int MyFunction(int x, int y)
 3: {
 4:   int w, z;
 5:   {
 6:     int y;
 7:   }
 8:   {
 9:     int w;
10:   }
11: }
```



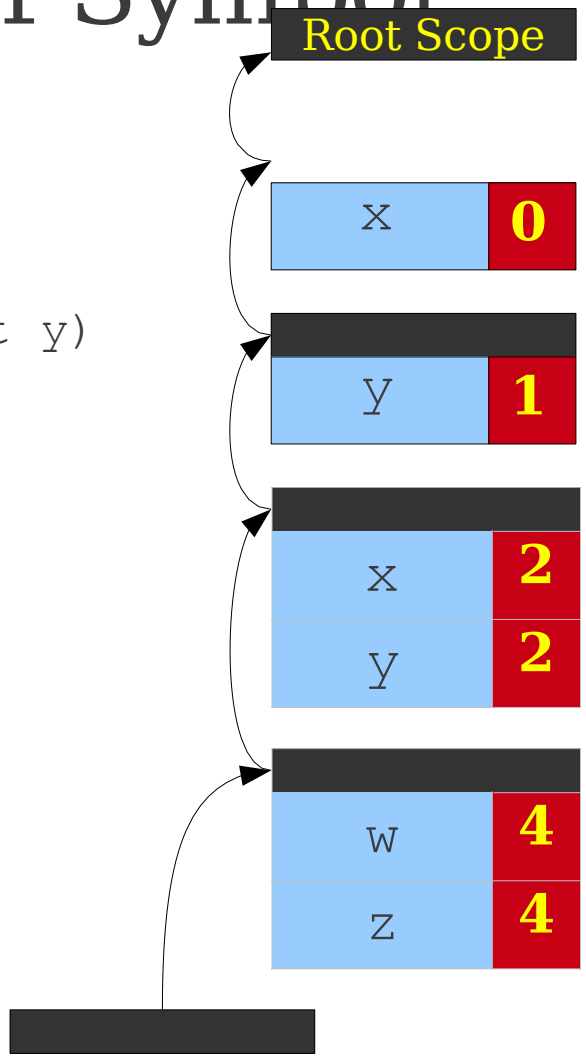
Another View of Symbol Tables

```
 0: int x;
 1: int y;
 2: int MyFunction(int x, int y)
 3: {
 4:   int w, z;
 5:   {
 6:     int y;
 7:   }
 8:   {
 9:     int w;
10:   }
11: }
```



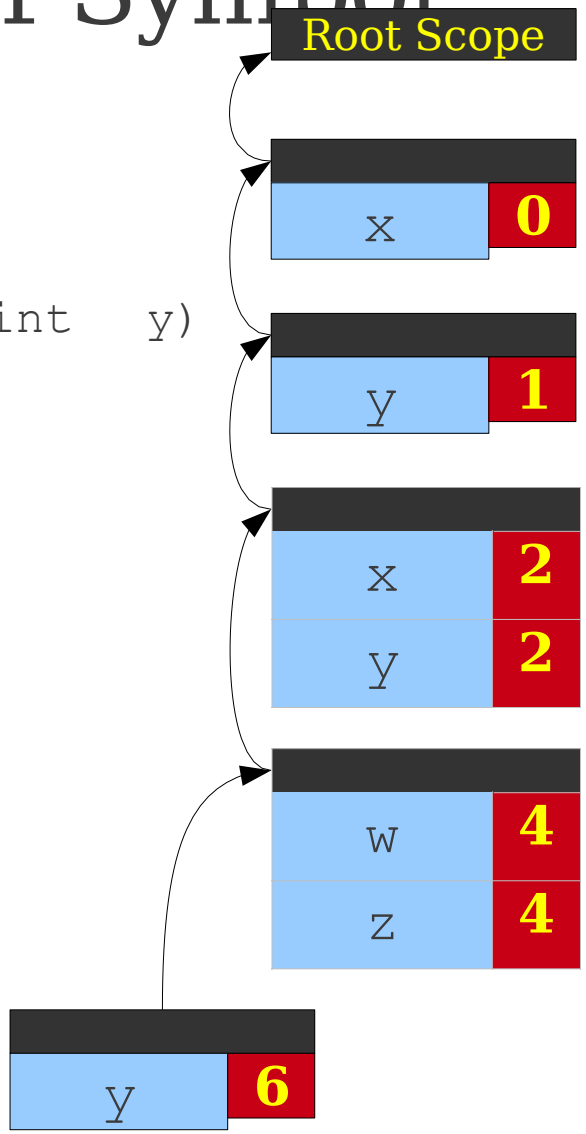
Another View of Symbol Tables

```
 0: int x;
 1: int y;
 2: int MyFunction(int x, int y)
 3: {
 4:   int w, z;
 5:   {
 6:     int y;
 7:   }
 8:   {
 9:     int w;
10:   }
11: }
```



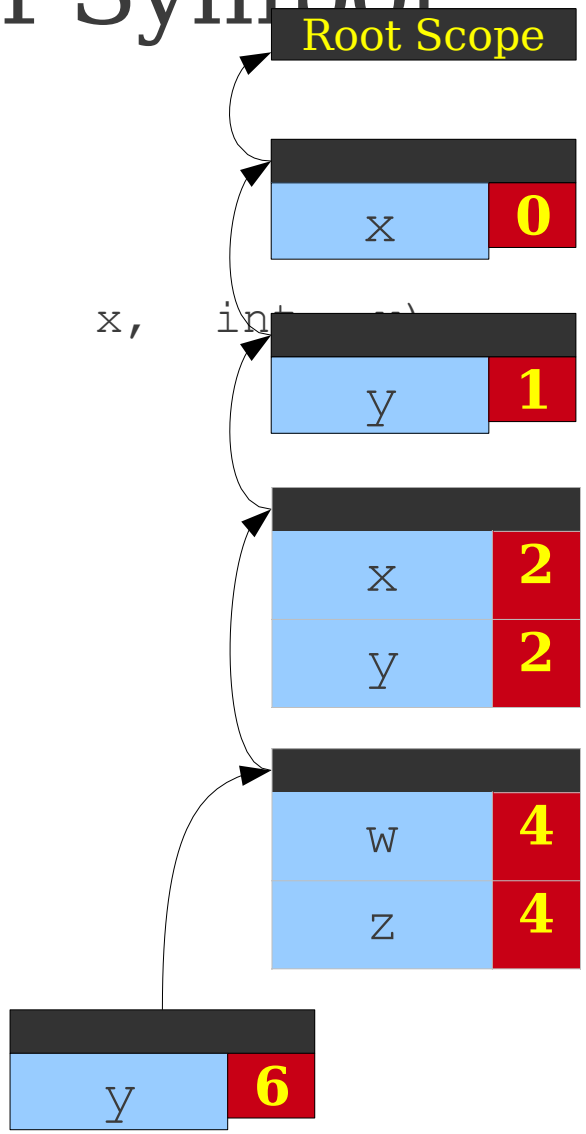
Another View of Symbol Tables

```
 0:   int x;
 1:   int y;
 2:   int MyFunction(int x, int y)
3:    {
 4:       int w, z;
 5:       {
 6:          int y;
 7:       }
 8:       {
 9:          int w;
10:       }
11:   }
```



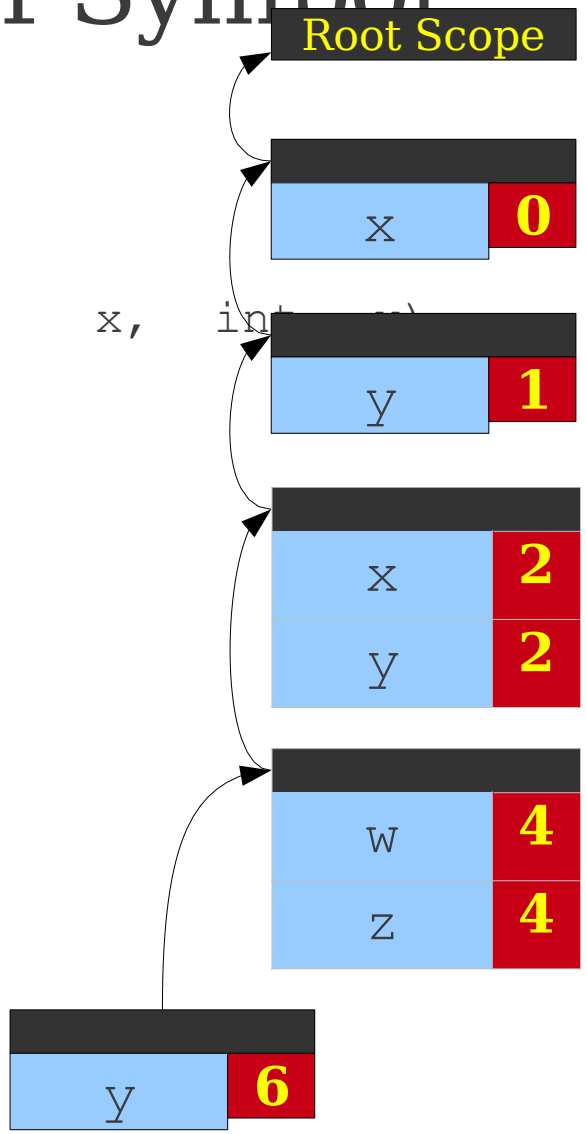
Another View of Symbol Tables

```
 0:   int x;
 1:   int y;
 2:   int MyFunction(int x, int y)
 3:   {
 4:      int w, z;
 5:      {
 6:         int y;
 7:      }
 8:      {
 9:         int w;
10:      }
11:   }
```



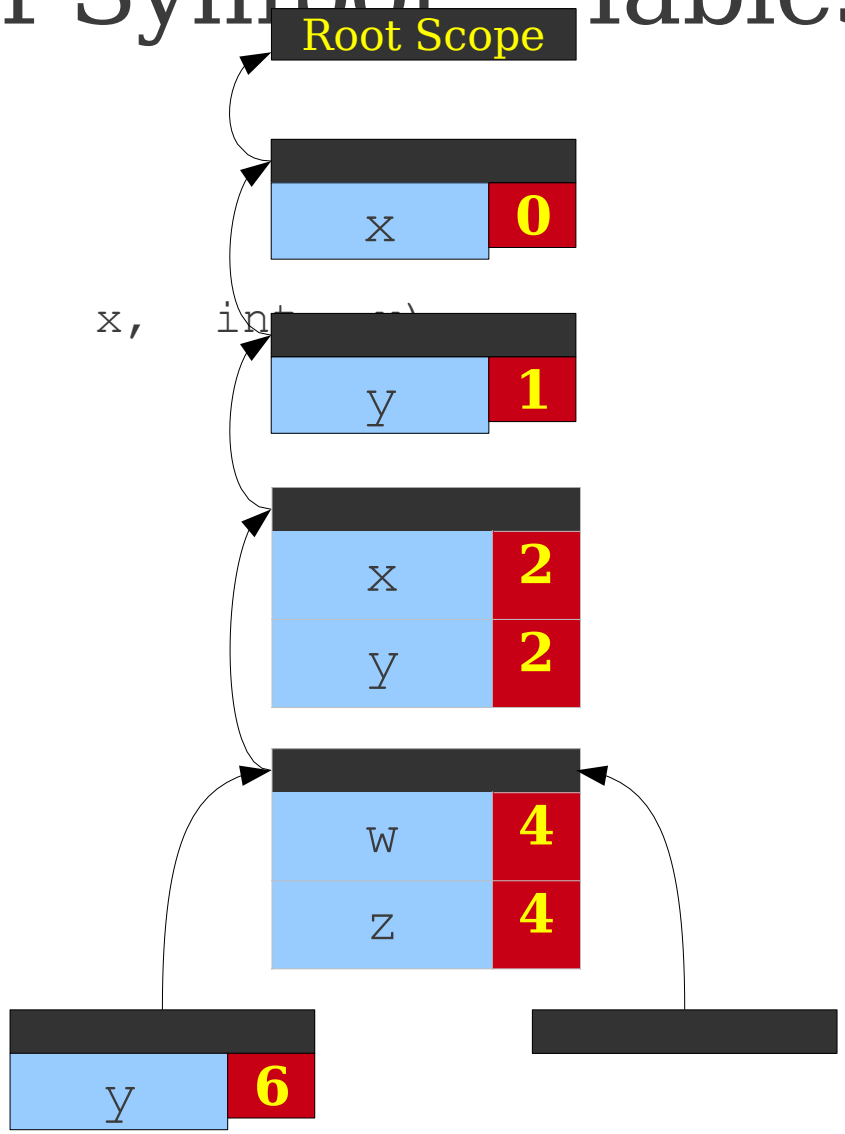
Another View of Symbol Tables

```
 0:   int x;
 1:   int y;
 2:   int MyFunction(int x, int y)
 3:   {
 4:      int w, z;
 5:      {
 6:         int y;
 7:      }
 8:      {
 9:         int w;
10:      }
11:   }
```



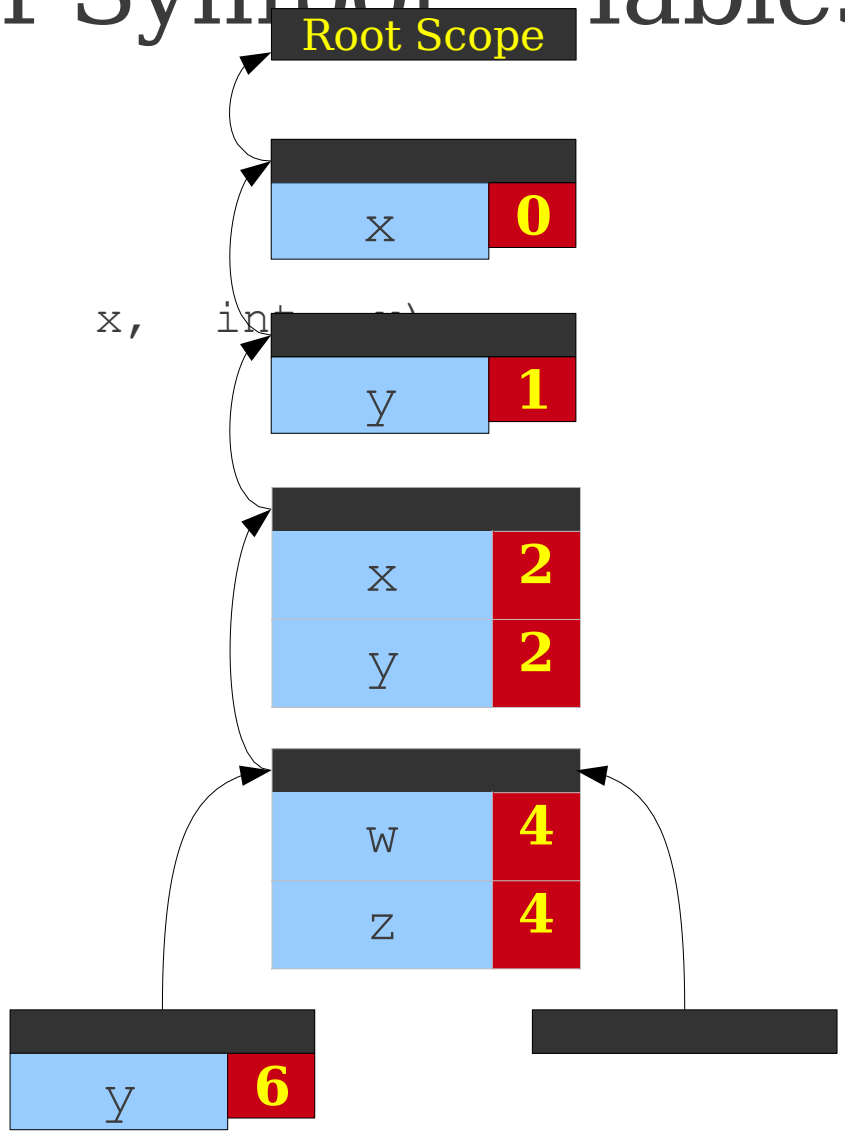
Another View of Symbol Tables

```
 0:   int x;
 1:   int y;
 2:   int MyFunction(int x, int y)
 3:   {
 4:      int w, z;
 5:      {
 6:         int y;
 7:      }
 8:      {
 9:         int w;
10:      }
11:   }
```



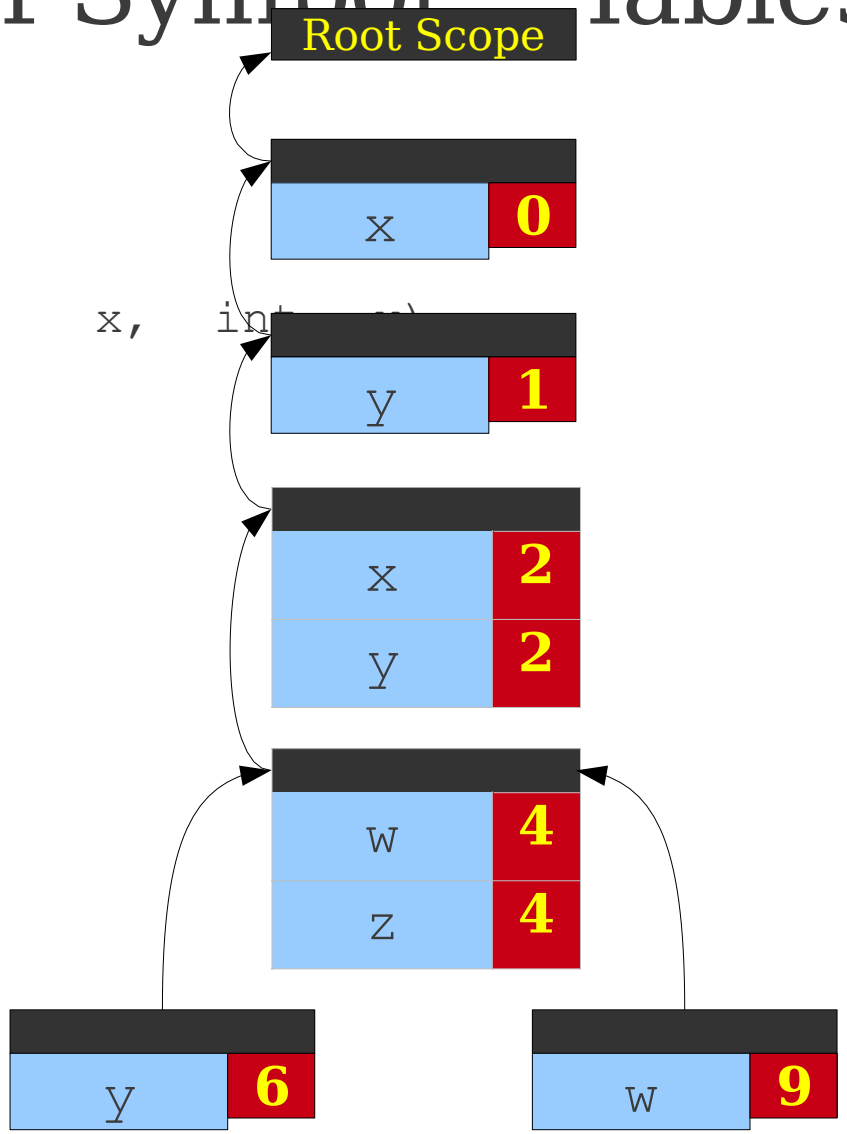
Another View of Symbol Tables

```
 0:   int x;
 1:   int y;
 2:   int MyFunction(int x, int y)
 3:   {
 4:      int w, z;
 5:      {
 6:         int y;
 7:      }
 8:      {
 9:         int w;
10:      }
11:   }
```



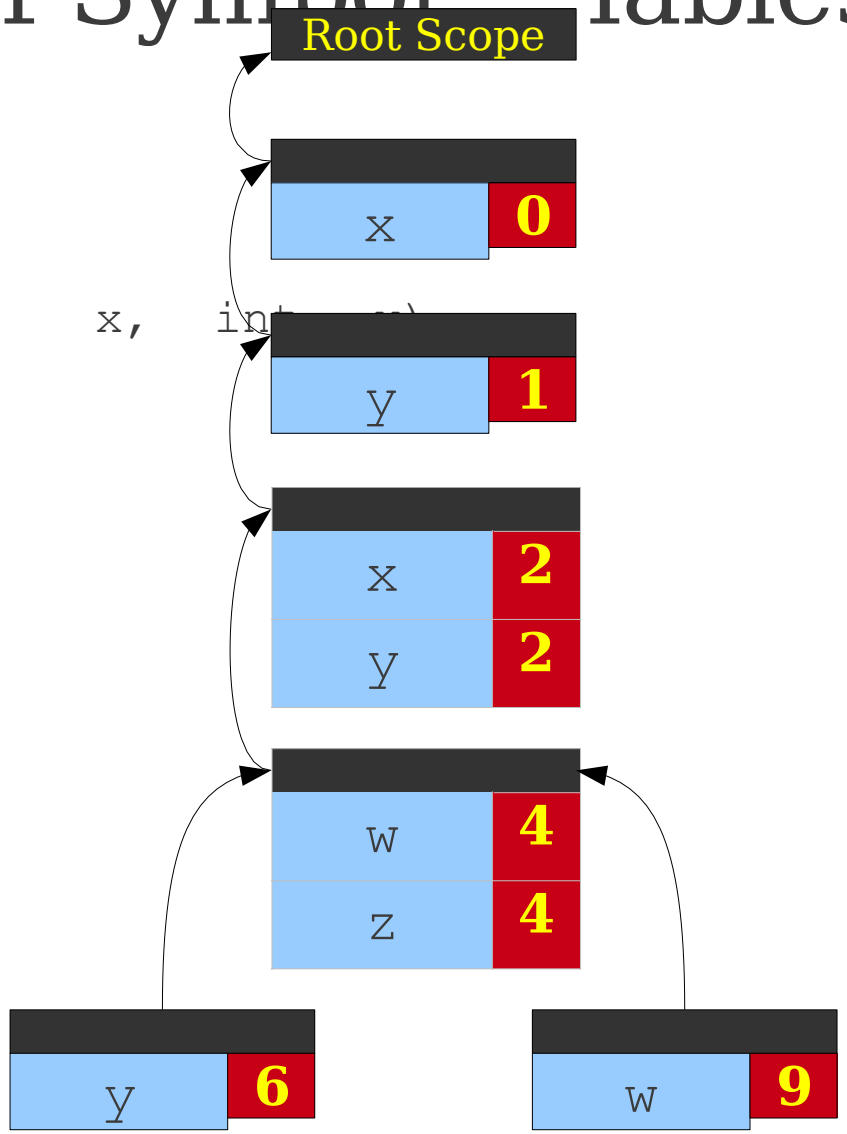
Another View of Symbol Tables

```
 0:   int x;
 1:   int y;
 2:   int MyFunction(int x, int y)
 3:   {
 4:      int w, z;
 5:      {
 6:         int y;
 7:      }
 8:      {
 9:         int w;
10:      }
11:   }
```



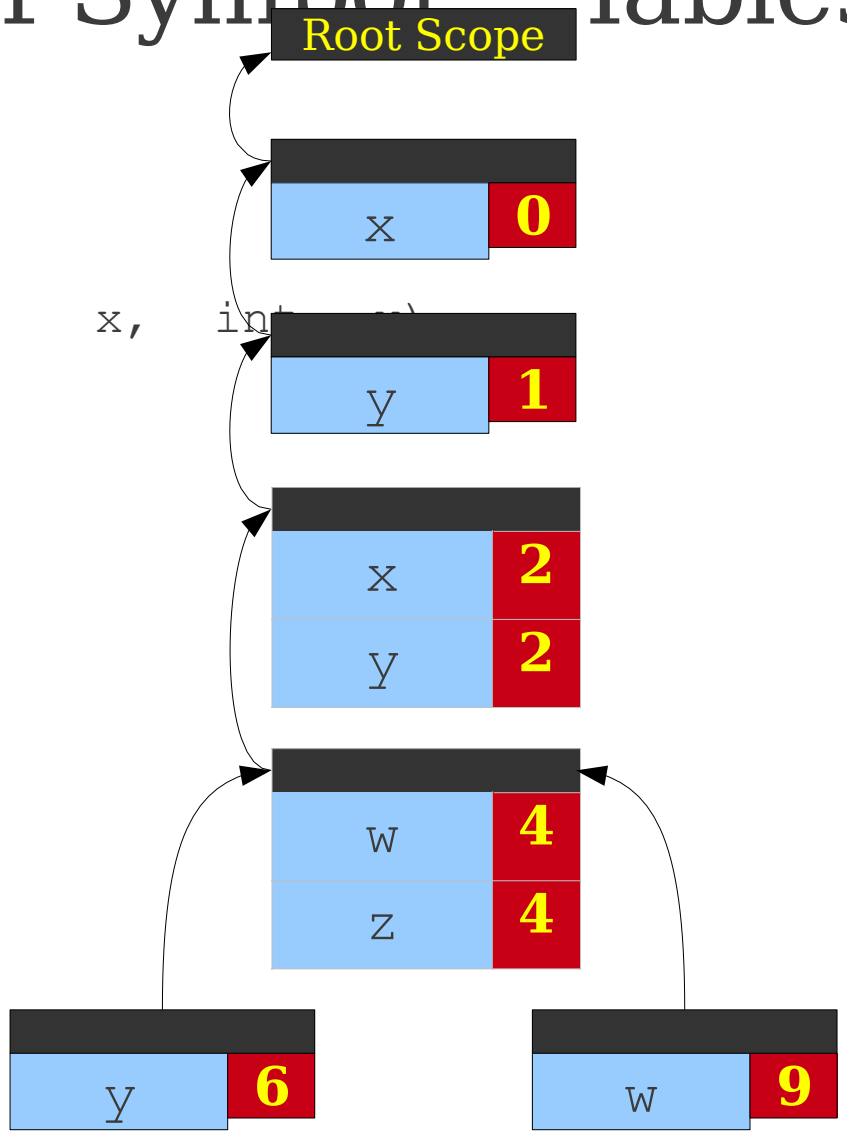
Another View of Symbol Tables

```
 0:   int x;
 1:   int y;
 2:   int MyFunction(int x, int y)
 3:   {
 4:      int w, z;
 5:      {
 6:         int y;
 7:      }
 8:      {
 9:         int w;
10:      }
11:   }
```



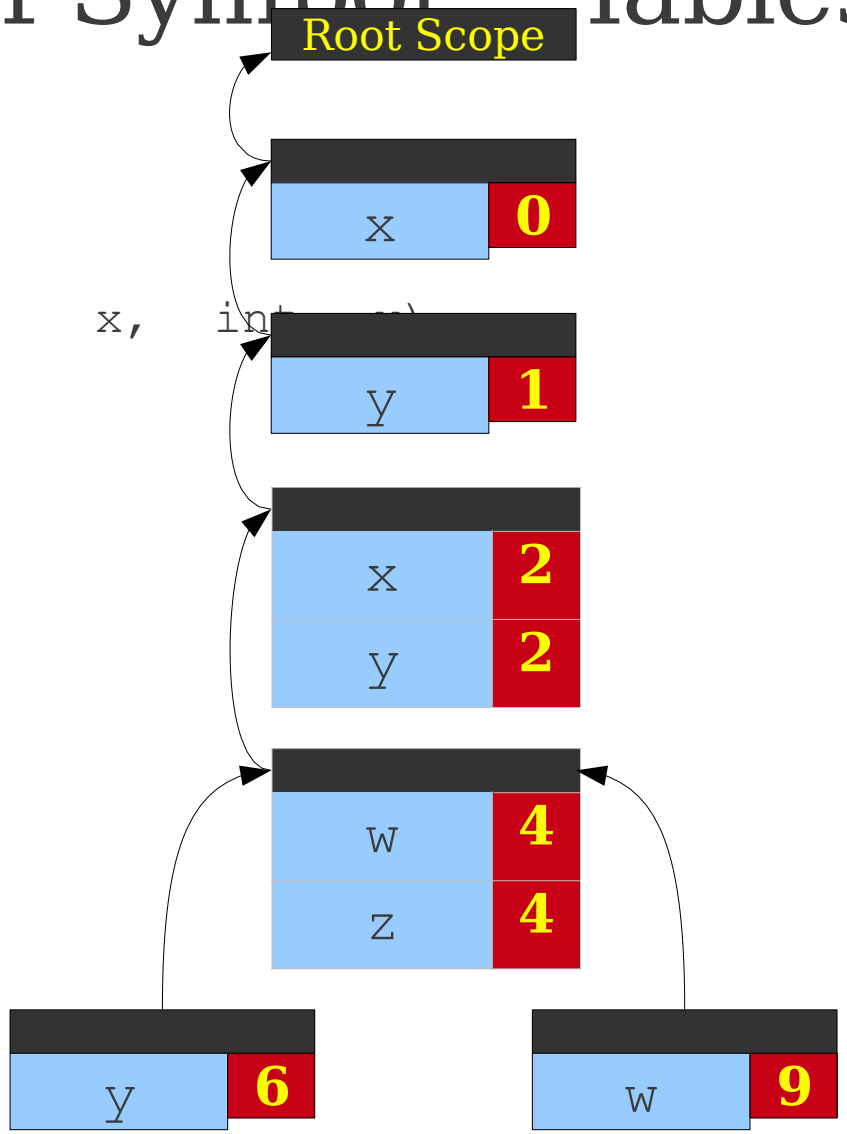
Another View of Symbol Tables

```
 0:   int x;
 1:   int y;
 2:   int MyFunction(int x, int y)
 3:   {
 4:      int w, z;
 5:      {
 6:         int y;
 7:      }
 8:      {
 9:         int w;
10:      }
11:  }
```



Another View of Symbol Tables

```
 0:   int x;
 1:   int y;
 2:   int MyFunction(int x, int y)
 3:   {
 4:      int w, z;
 5:      {
 6:         int y;
 7:      }
 8:      {
 9:         int w;
10:      }
11:   }
```



Why Two Interpretations?

- Spaghetti stack more accurately captures the scoping structure.
- Spaghetti stack is a *static* structure; explicit stack is a *dynamic* structure.
- Explicit stack is an optimization of a spaghetti stack; more on that later.

Scoping in Object-Oriented Languages

Scoping with Inheritance

```
public class Base {
     public int publicBaseInt = 1;
     protected int baseInt = 2;
}
```

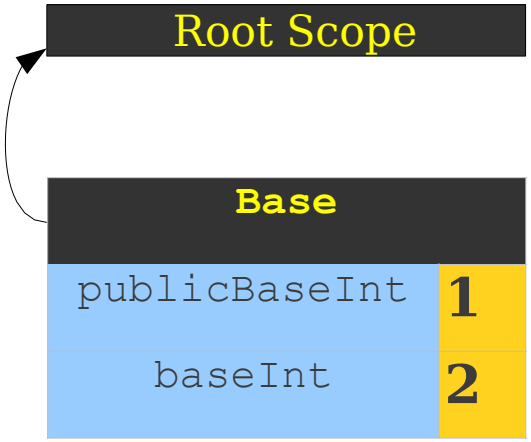
Scoping with Inheritance

Root Scope

```
public class Base {
     public int publicBaseInt = 1;
     protected int baseInt = 2;
}
```

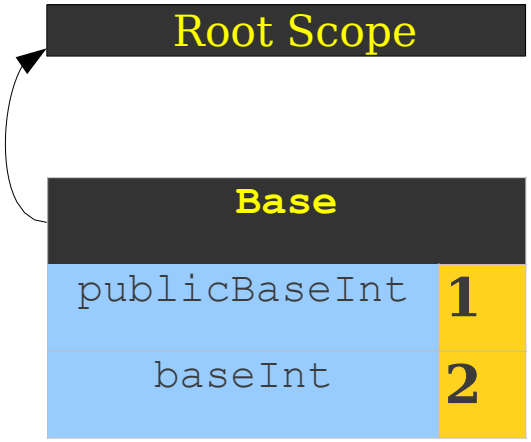
Scoping with Inheritance

```
public class Base {
     public int publicBaseInt = 1;
     protected int baseInt = 2;
}
```



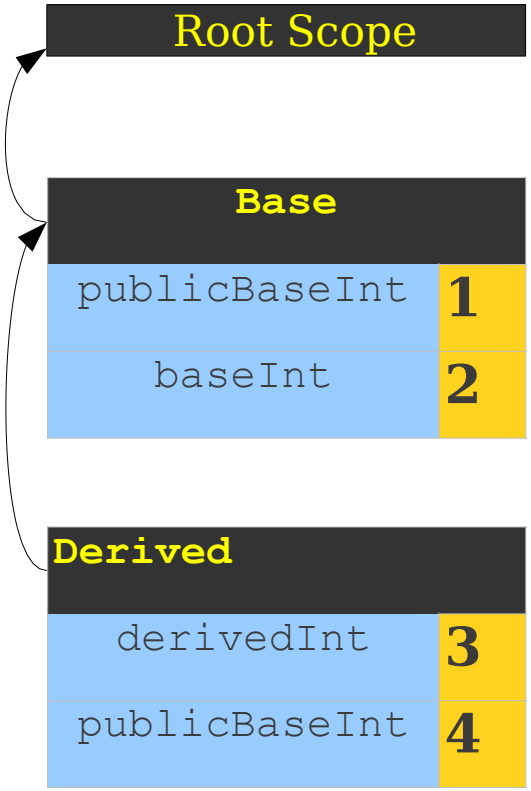
Scoping with Inheritance

```
public class Base {
     public int publicBaseInt = 1;
     protected int baseInt = 2;
}

public class Derived extends Base {
     public int derivedInt = 3;
     public int publicBaseInt = 4;

     public void doSomething()
          {  System.out.println(publicBaseIn
          t);  System.out.println(baseInt);
          System.out.println(derivedInt);

           int publicBaseInt = 6;
           System.out.println(publicBaseInt);
     }
}
```



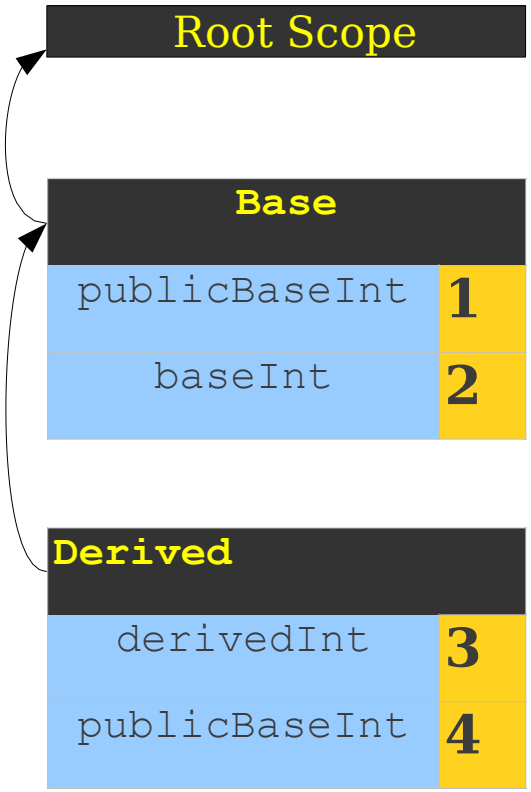
Scoping with Inheritance

```
public class Base {
     public int publicBaseInt = 1;
     protected int baseInt = 2;
}

public class Derived extends Base {
     public int derivedInt = 3;
     public int publicBaseInt = 4;

     public void doSomething()
          {  System.out.println(publicBaseIn
          t);  System.out.println(baseInt);
          System.out.println(derivedInt);

           int publicBaseInt = 6;
           System.out.println(publicBaseInt);
     }
}
```



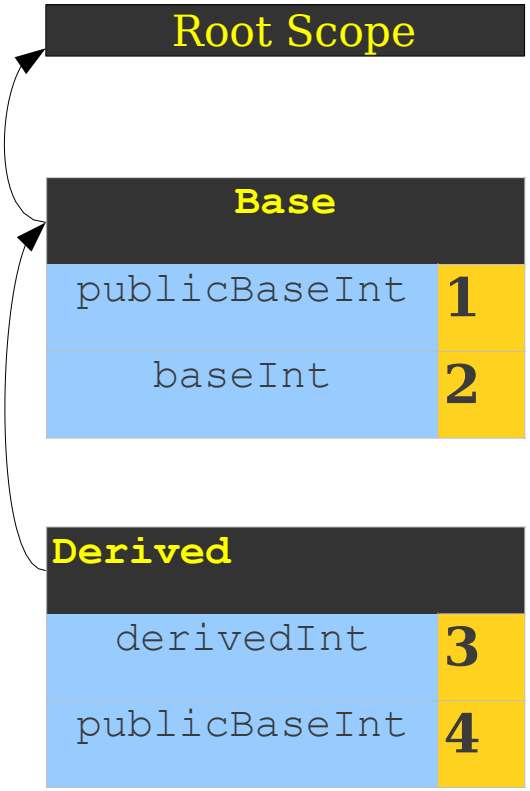
Scoping with Inheritance

```
public class Base {
     public int publicBaseInt = 1;
     protected int baseInt = 2;
}

public class Derived extends Base {
     public int derivedInt = 3;
     public int publicBaseInt = 4;

     public void doSomething()
          {  System.out.println(publicBaseIn
          t);  System.out.println(baseInt);
          System.out.println(derivedInt);

           int publicBaseInt = 6;
           System.out.println(publicBaseInt);

     }
}
```



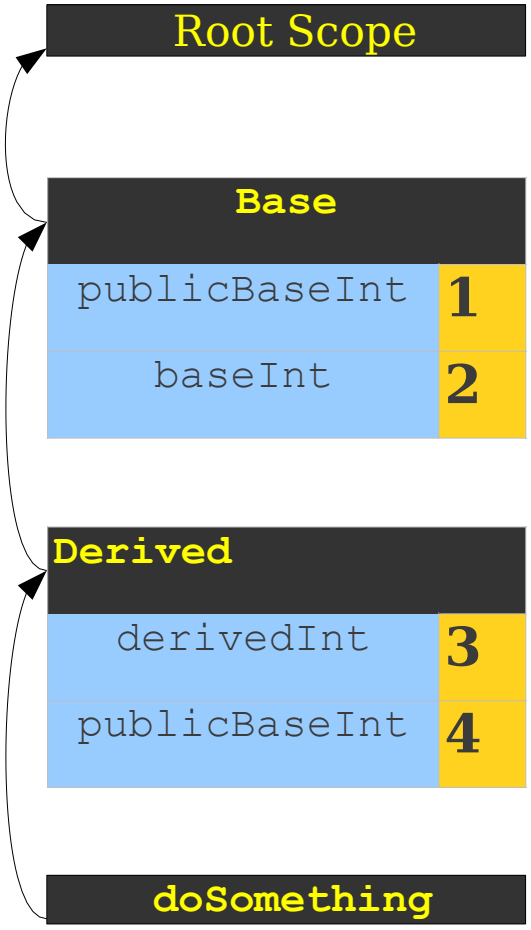
Scoping with Inheritance

```
public class Base {
     public int publicBaseInt = 1;
     protected int baseInt = 2;
}

public class Derived extends Base {
     public int derivedInt = 3;
     public int publicBaseInt = 4;

     public void doSomething()
          {  System.out.println(publicBaseIn
          t);  System.out.println(baseInt);
          System.out.println(derivedInt);

           int publicBaseInt = 6;
           System.out.println(publicBaseInt);

     }
}
```



Scoping with Inheritance

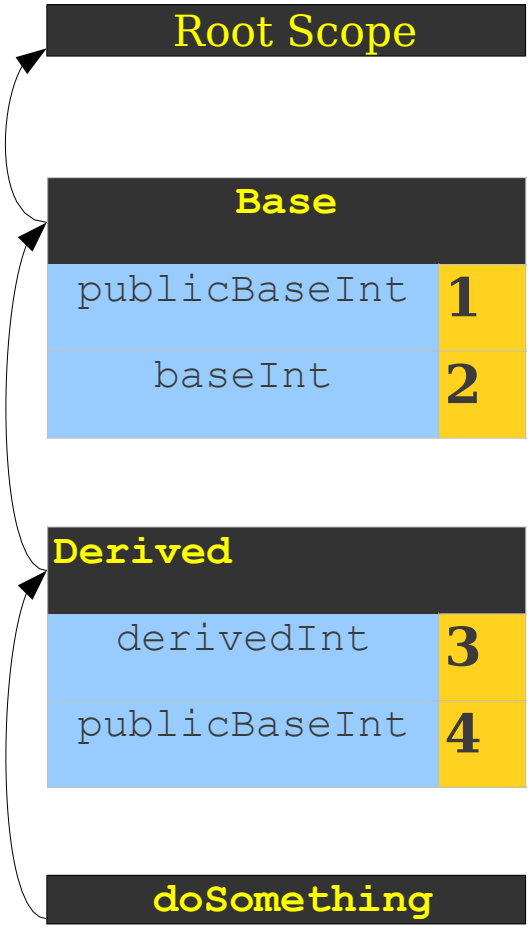
```
public class Base {
     public int publicBaseInt = 1;
     protected int baseInt = 2;
}

public class Derived extends Base {
     public int derivedInt = 3;
     public int publicBaseInt = 4;

     public void doSomething()
          {  System.out.println(publicBaseIn
          t);  System.out.println(baseInt);
          System.out.println(derivedInt);

           int publicBaseInt = 6;
           System.out.println(publicBaseInt);

     }
}
```



Scoping with Inheritance

```
public class Base {
     public int publicBaseInt = 1;
     protected int baseInt = 2;
}

public class Derived extends Base {
     public int derivedInt = 3;
     public int publicBaseInt = 4;

     public void doSomething()
          {  System.out.println(publicBaseIn
          t);  System.out.println(baseInt);
          System.out.println(derivedInt);

           int publicBaseInt = 6;
           System.out.println(publicBaseInt);

     }
}
```

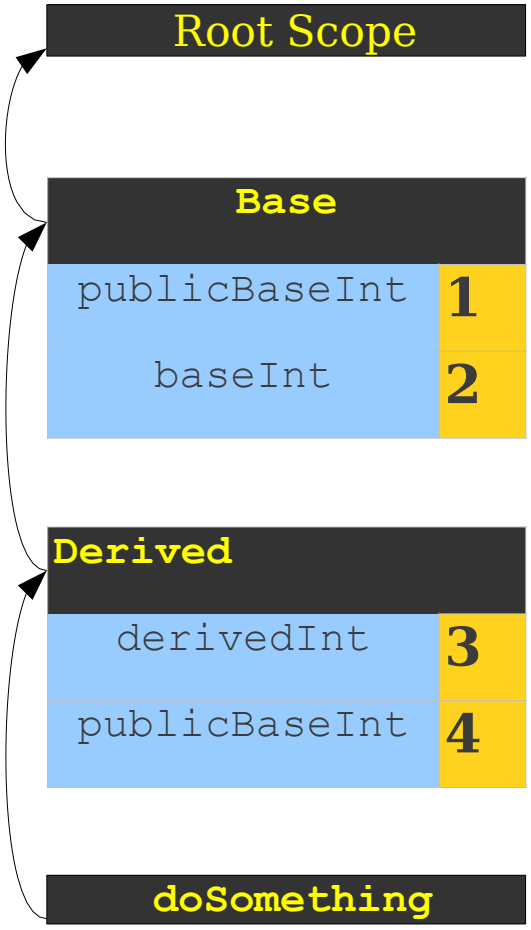
```
>
```



Scoping with Inheritance

```
public class Base {
     public int publicBaseInt = 1;
     protected int baseInt = 2;
}

public class Derived extends Base {
     public int derivedInt = 3;
     public int publicBaseInt = 4;

     public void doSomething()
          {  System.out.println(publicBaseIn
          t);  System.out.println(baseInt);
          System.out.println(derivedInt);

           int publicBaseInt = 6;
           System.out.println(publicBaseInt);

     }
}
```

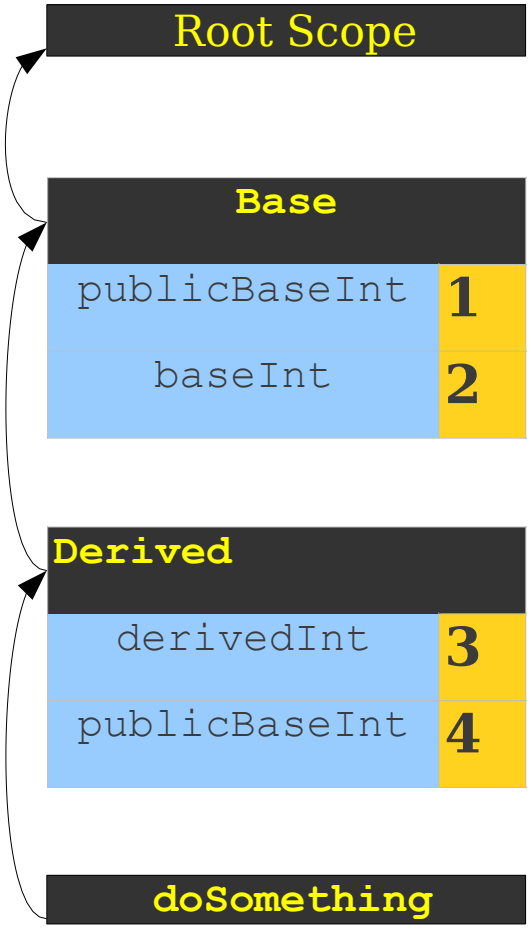
```
> 4
```



Scoping with Inheritance

```
public class Base {
     public int publicBaseInt = 1;
     protected int baseInt = 2;
}

public class Derived extends Base {
     public int derivedInt = 3;
     public int publicBaseInt = 4;

     public void doSomething()
          {  System.out.println(publicBaseIn
          t);  System.out.println(baseInt);
          System.out.println(derivedInt);

           int publicBaseInt = 6;
           System.out.println(publicBaseInt);

     }
}
```

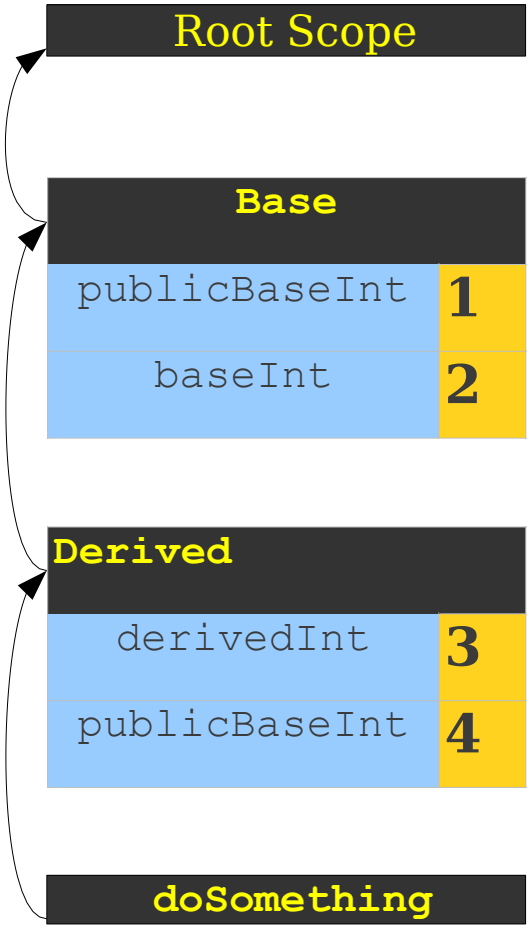
```
> 4
```



Scoping with Inheritance

```
public class Base {
     public int publicBaseInt = 1;
     protected int baseInt = 2;
}

public class Derived extends Base {
     public int derivedInt = 3;
     public int publicBaseInt = 4;

     public void doSomething()
          {  System.out.println(publicBaseIn
          t);  System.out.println(baseInt);
          System.out.println(derivedInt);

           int publicBaseInt = 6;
           System.out.println(publicBaseInt);

     }
}
```

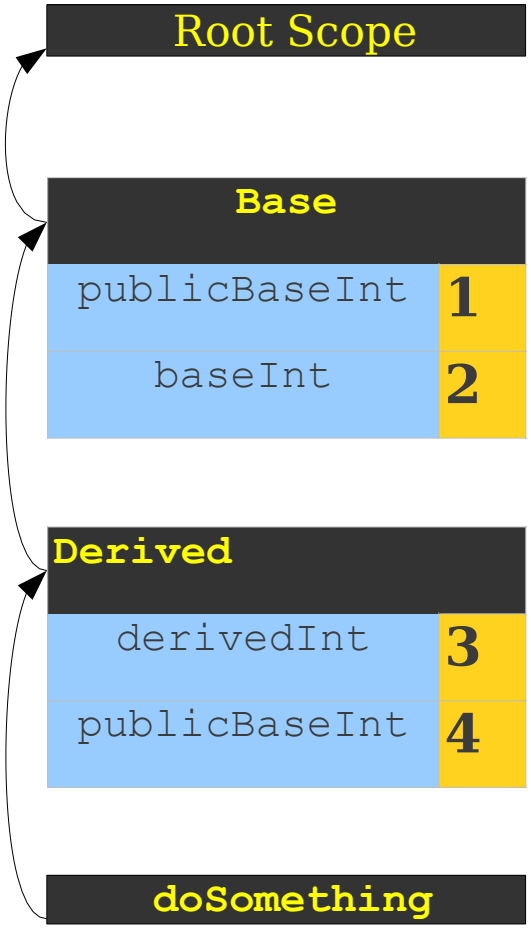
```
>4
 2
```



Scoping with Inheritance

```
public class Base {
     public int publicBaseInt = 1;
     protected int baseInt = 2;
}

public class Derived extends Base {
     public int derivedInt = 3;
     public int publicBaseInt = 4;

     public void doSomething()
          {  System.out.println(publicBaseIn
          t);  System.out.println(baseInt);
          System.out.println(derivedInt);

          int publicBaseInt = 6;
          System.out.println(publicBaseInt);

     }
}
```

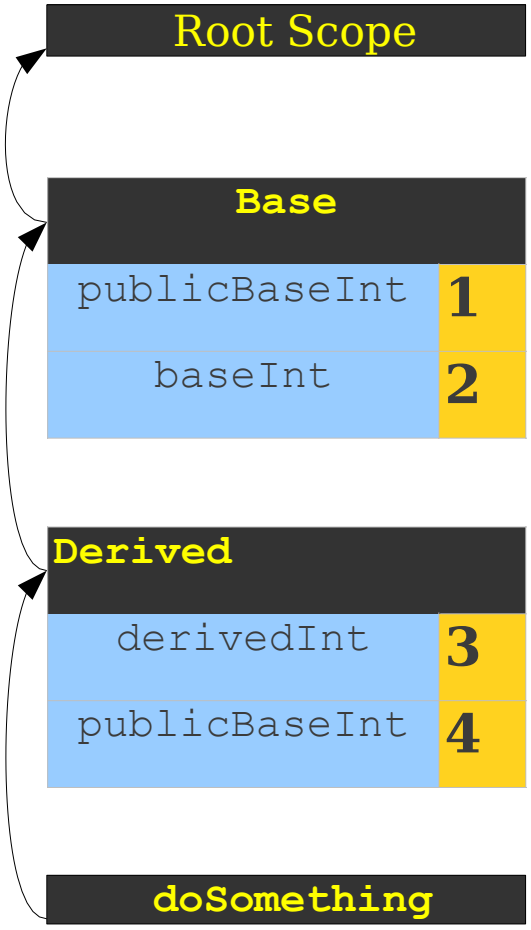
```
>4
 2
```



Scoping with Inheritance

```
public class Base {
     public int publicBaseInt = 1;
     protected int baseInt = 2;
}

public class Derived extends Base {
     public int derivedInt = 3;
     public int publicBaseInt = 4;

     public void doSomething()
          {  System.out.println(publicBaseIn
          t);  System.out.println(baseInt);
          System.out.println(derivedInt);

          int publicBaseInt = 6;
          System.out.println(publicBaseInt);

     }
}
```

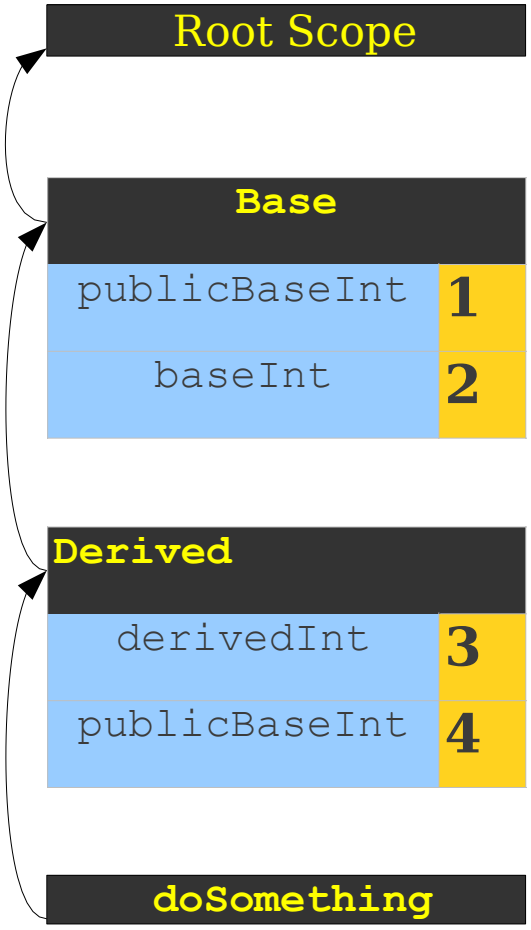
```
> 4
  2
  3
```



Scoping with Inheritance

```
public class Base {
     public int publicBaseInt = 1;
     protected int baseInt = 2;
}

public class Derived extends Base {
     public int derivedInt = 3;
     public int publicBaseInt = 4;

     public void doSomething()
          {  System.out.println(publicBaseIn
          t);  System.out.println(baseInt);
          System.out.println(derivedInt);

          int publicBaseInt = 6;
          System.out.println(publicBaseInt);

     }
}
```

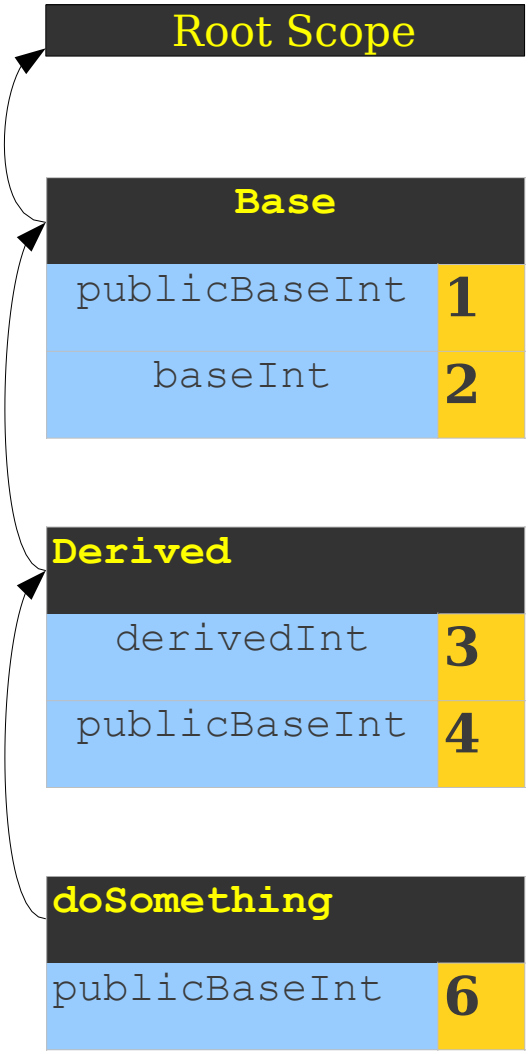
```
> 4
  2
  3
```



Scoping with Inheritance

```
public class Base {
     public int publicBaseInt = 1;
     protected int baseInt = 2;
}

public class Derived extends Base {
     public int derivedInt = 3;
     public int publicBaseInt = 4;

     public void doSomething()
          {  System.out.println(publicBaseIn
          t);  System.out.println(baseInt);
          System.out.println(derivedInt);

          int publicBaseInt = 6;
          System.out.println(publicBaseInt);

     }
}
```

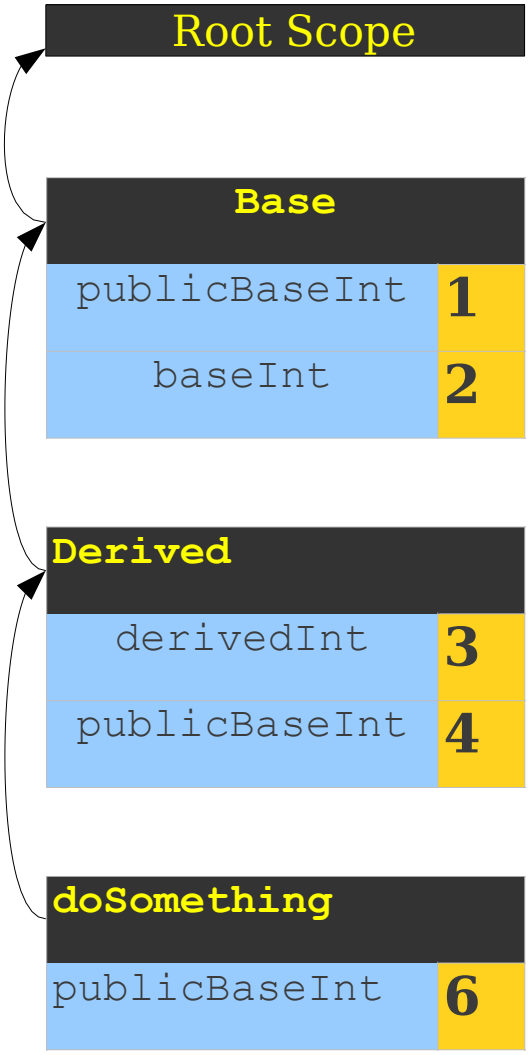
```
>4
 2
 3
```



Scoping with Inheritance

```
public class Base {
     public int publicBaseInt = 1;
     protected int baseInt = 2;
}

public class Derived extends Base {
     public int derivedInt = 3;
     public int publicBaseInt = 4;

     public void doSomething()
          {  System.out.println(publicBaseIn
          t);  System.out.println(baseInt);
          System.out.println(derivedInt);

          int publicBaseInt = 6;
          System.out.println(publicBaseInt);

     }
}
```

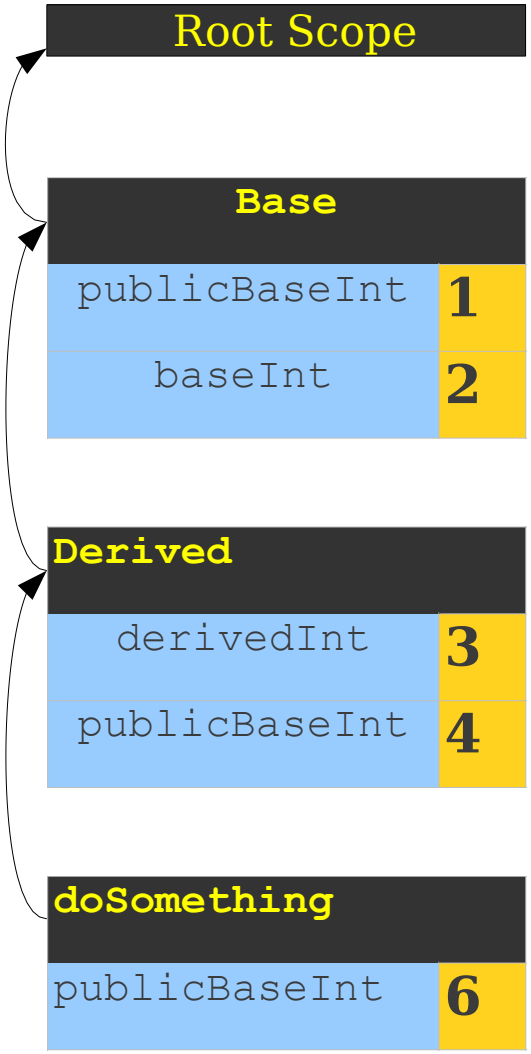
```
>4
 2
 3
```



Scoping with Inheritance

```
public class Base {
     public int publicBaseInt = 1;
     protected int baseInt = 2;
}

public class Derived extends Base {
     public int derivedInt = 3;
     public int publicBaseInt = 4;

     public void doSomething()
          {  System.out.println(publicBaseIn
          t);  System.out.println(baseInt);
          System.out.println(derivedInt);

          int publicBaseInt = 6;
          System.out.println(publicBaseInt);

     }
}
```

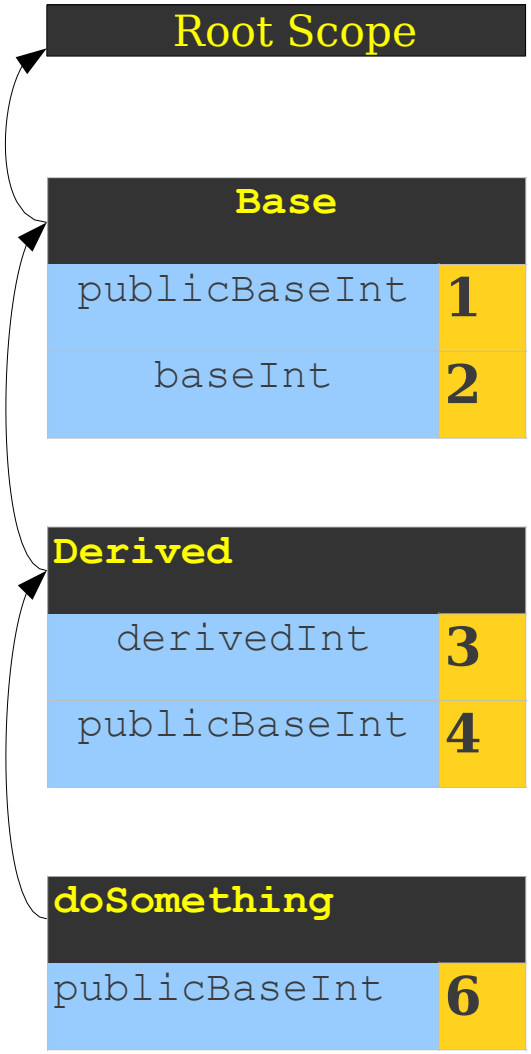
```
> 4
  2
  3
  6
```



Scoping with Inheritance

```
public class Base {
     public int publicBaseInt = 1;
     protected int baseInt = 2;
}

public class Derived extends Base {
     public int derivedInt = 3;
     public int publicBaseInt = 4;

     public void doSomething()
          {  System.out.println(publicBaseIn
          t);  System.out.println(baseInt);
          System.out.println(derivedInt);

           int publicBaseInt = 6;
           System.out.println(publicBaseInt);

     }
}
```

```
>4
 2
 3
 6
```

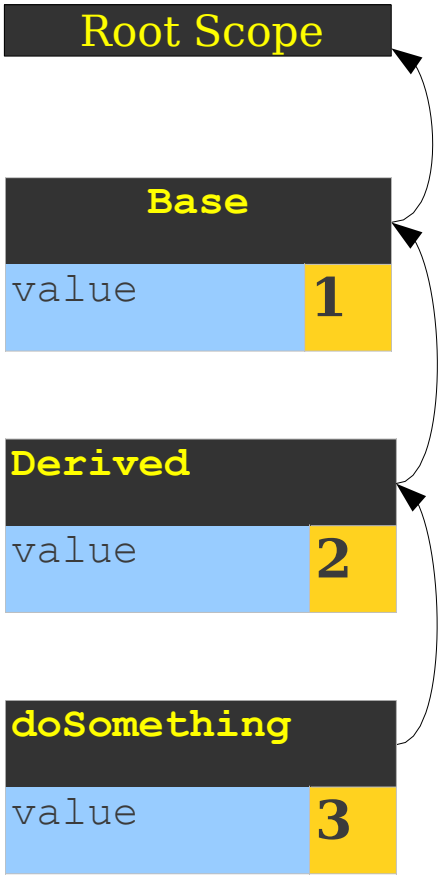


Inheritance and Scoping

- Typically, the scope for a derived class will store a link to the scope of its base class.
- Looking up a field of a class traverses the scope chain until that field is found or a semantic error is found.

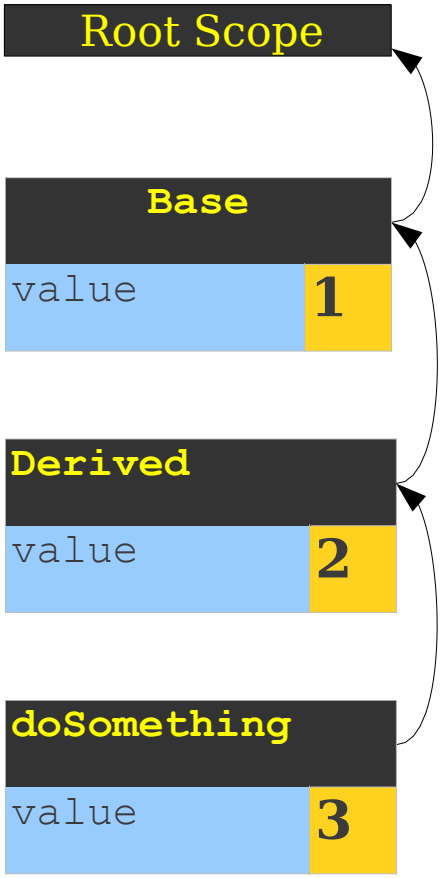
Explicit Disambiguation

```
public class Base
     {  public int value =
     1;
}

public class Derived extends Base {
     public int value = 2;

      public void doSomething()
           {  int value = 3;
           System.out.println(value);
            System.out.println(this.value);
            System.out.println(super.value)
            ;
      }
}
```



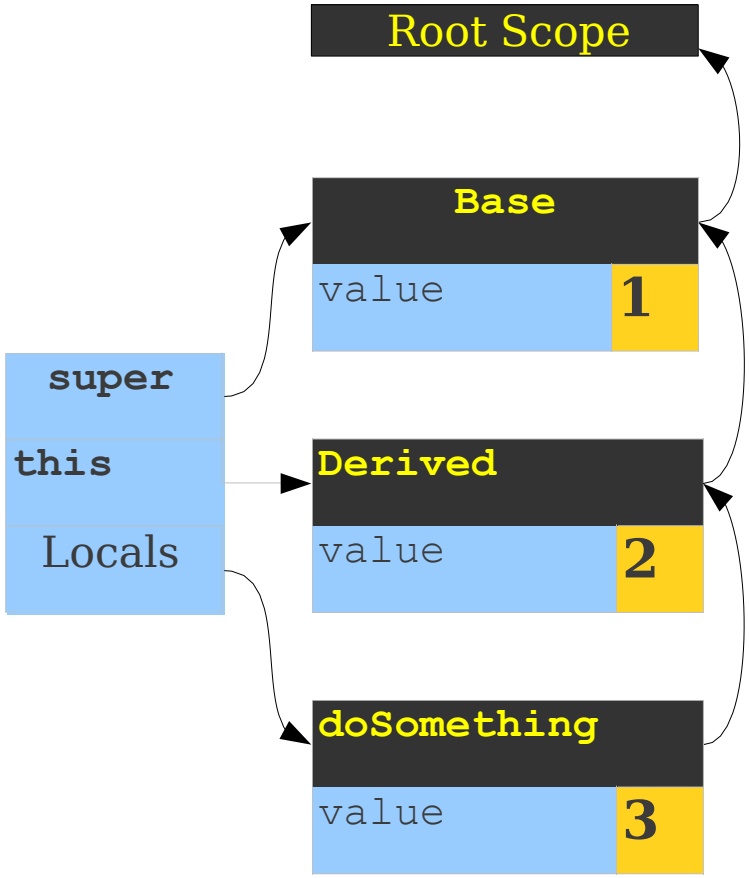
Explicit Disambiguation

```
public class Base
     {  public int value =
     1;
}

public class Derived extends Base {
     public int value = 2;

      public void doSomething()
           {  int value = 3;
           System.out.println(value);
            System.out.println(this.value);
            System.out.println(super.value)
            ;
      }
}
```



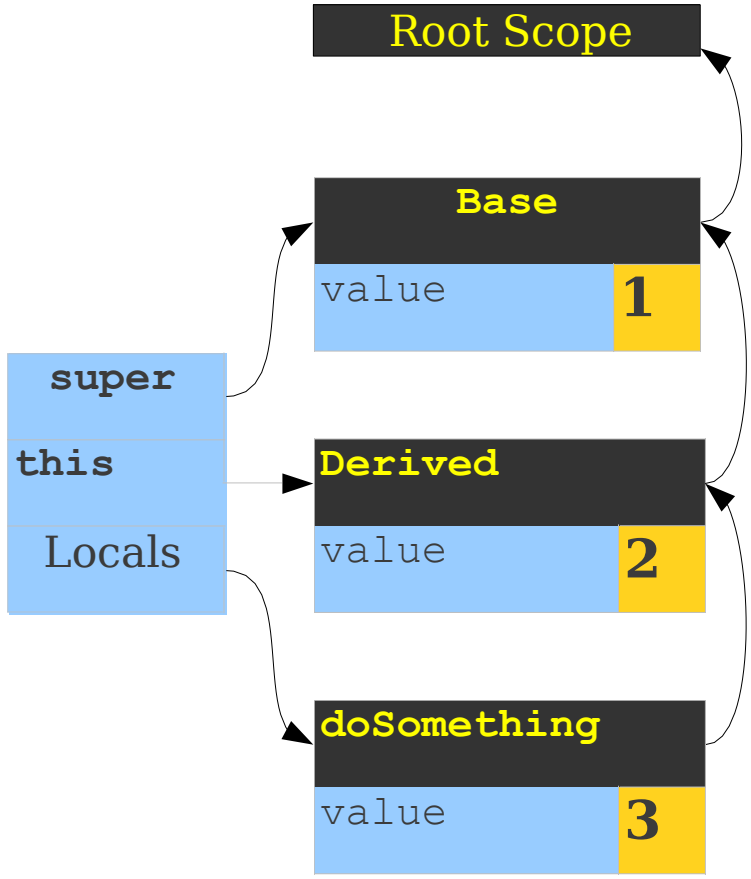
Explicit Disambiguation

```
public class Base
     {  public int value =
     1;
}

public class Derived extends Base {
     public int value = 2;

      public void doSomething()
           {  int value = 3;
           System.out.println(value);
            System.out.println(this.value);
            System.out.println(super.value)
            ;
      }
}
```



Explicit Disambiguation

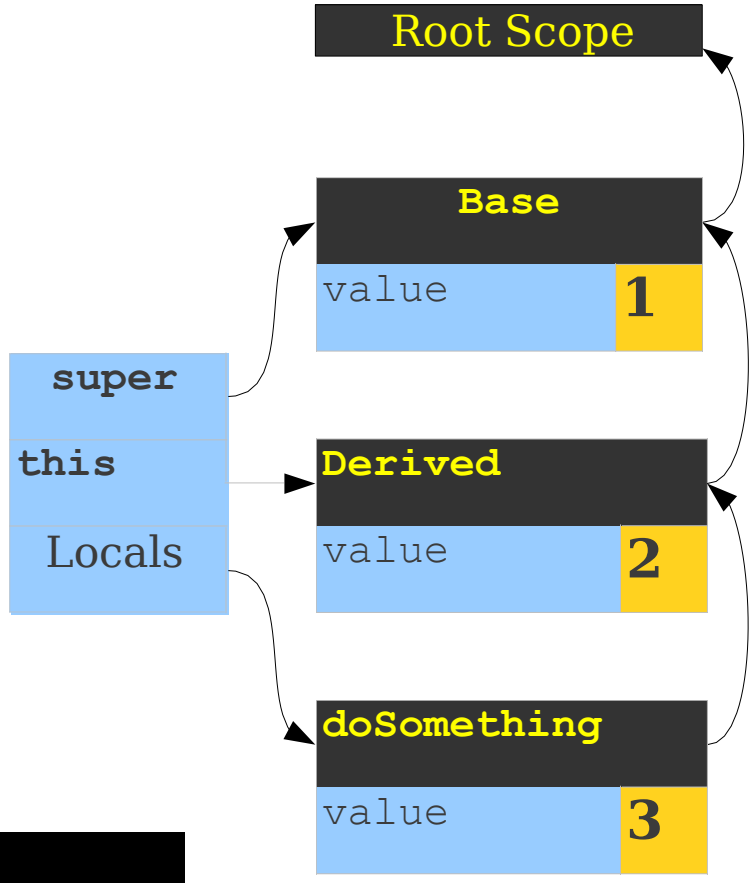
```
public class Base
     {  public int value =
     1;
}

public class Derived extends Base {
     public int value = 2;

      public void doSomething()
           {  int value = 3;
           System.out.println(value);
            System.out.println(this.value);
            System.out.println(super.value)
            ;
      }
}
```



Explicit Disambiguation

```
public class Base
     {  public int value =
     1;
}

public class Derived extends Base {
     public int value = 2;

      public void doSomething()
           {  int value = 3;
           System.out.println(value);
            System.out.println(this.value);
            System.out.println(super.value)
            ;
      }
}
```

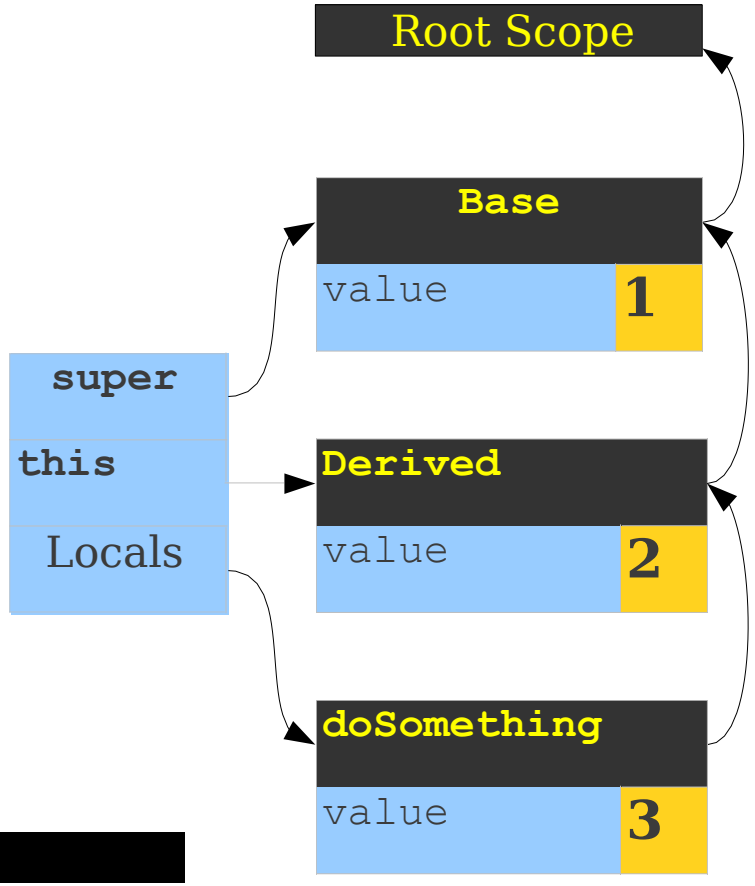
```
>
```



Explicit Disambiguation

```
public class Base
     {  public int value =
     1;
}

public class Derived extends Base {
     public int value = 2;

      public void doSomething()
           {  int value = 3;
           System.out.println(value);
            System.out.println(this.value);
            System.out.println(super.value)
            ;
      }
}
```

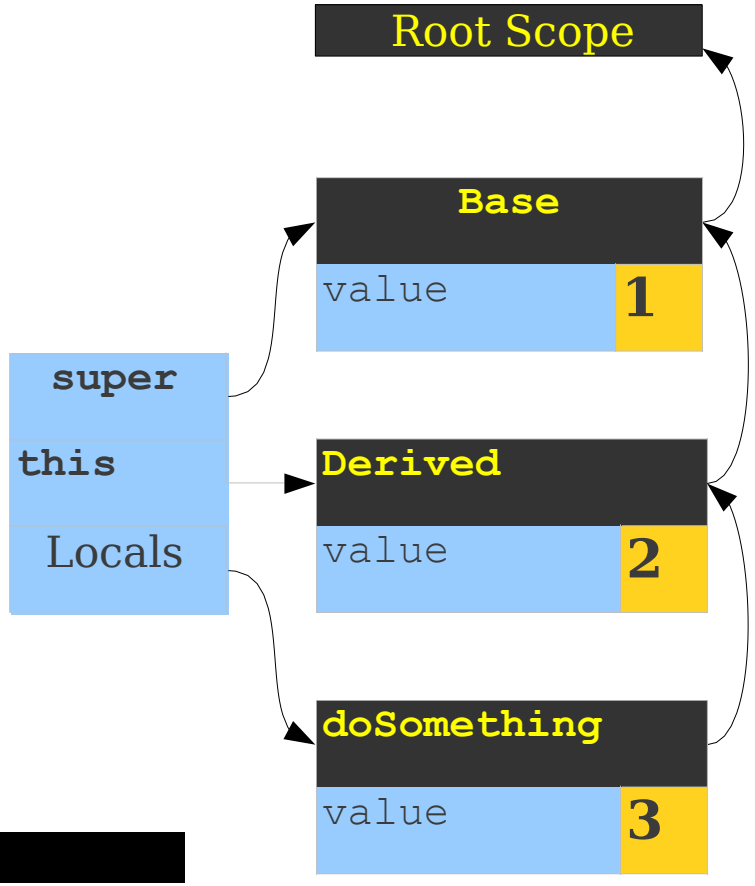
```
>
```



Explicit Disambiguation

```
public class Base
     {  public int value =
     1;
}

public class Derived extends Base {
     public int value = 2;

      public void doSomething()
           {  int value = 3;
           System.out.println(value);
            System.out.println(this.value);
            System.out.println(super.value)
            ;
      }
}
```

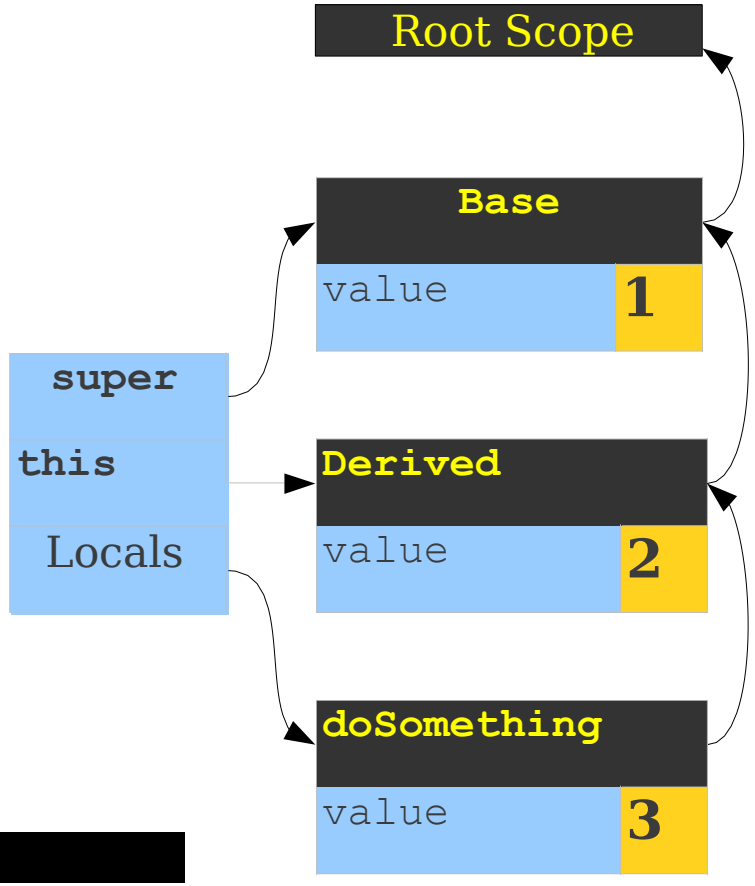
```
> 3
```



Explicit Disambiguation

```
public class Base
     {  public int value =
     1;
}

public class Derived extends Base {
     public int value = 2;

      public void doSomething()
           {  int value = 3;
           System.out.println(value);
           System.out.println(this.value);
           System.out.println(super.value);
      }
}
```

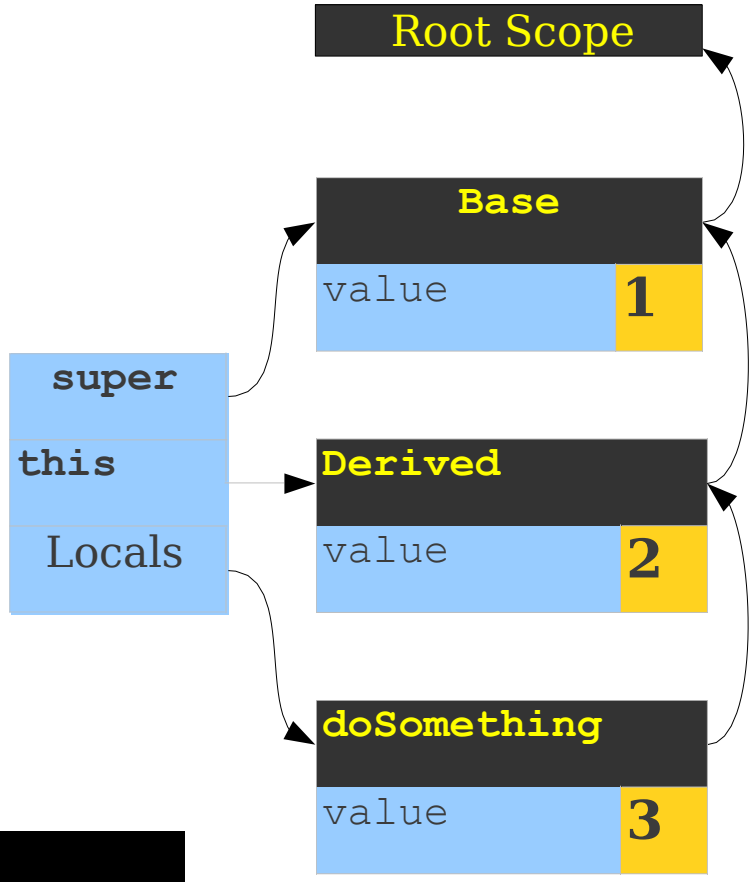
```
> 3
```



Explicit Disambiguation

```
public class Base
     {  public int value =
     1;
}

public class Derived extends Base {
     public int value = 2;

      public void doSomething()
           {  int value = 3;
           System.out.println(value);
           System.out.println(this.value);
           System.out.println(super.value);
      }
}
```

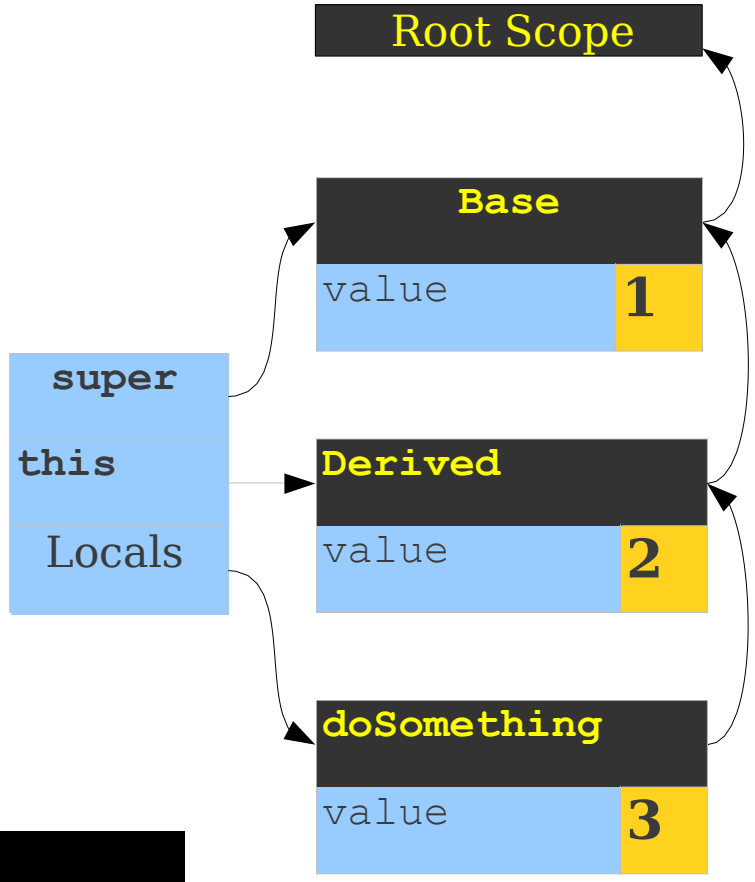
```
>3
 2
```



Explicit Disambiguation

```
public class Base
     {  public int value =
     1;
}

public class Derived extends Base {
     public int value = 2;

      public void doSomething()
           {  int value = 3;
           System.out.println(value);
            System.out.println(this.value);
            System.out.println(super.value);
      }
}
```

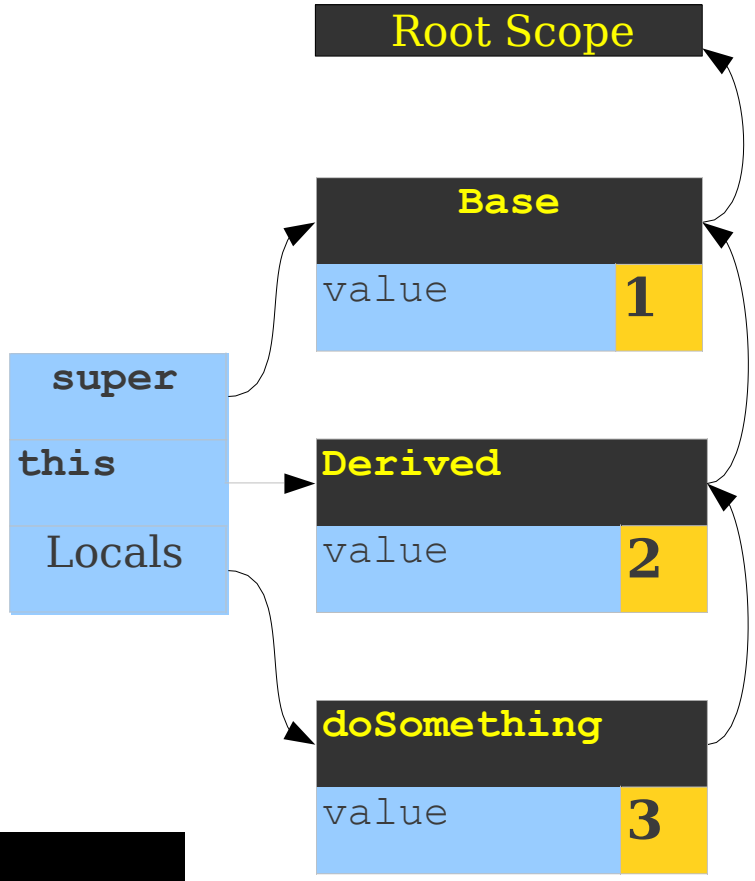
```
>3
 2
```



Explicit Disambiguation

```
public class Base
     {  public int value =
     1;
}

public class Derived extends Base {
     public int value = 2;

      public void doSomething()
           {  int value = 3;
           System.out.println(value);
            System.out.println(this.value);
            System.out.println(super.value);
      }
}
```

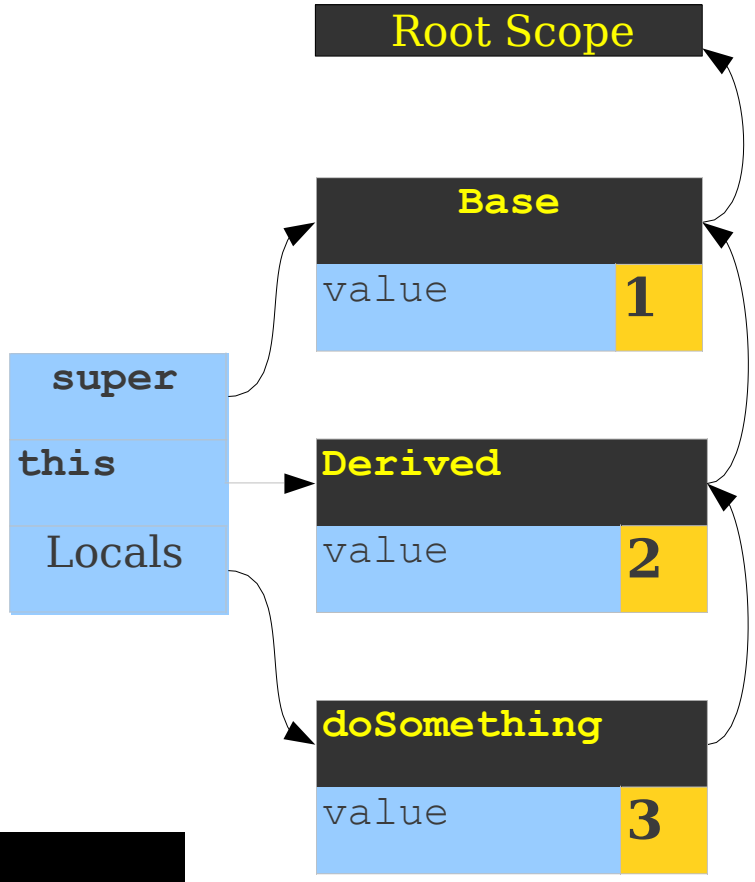
```
>3
 2
 1
```



Explicit Disambiguation

```
public class Base
     {  public int value =
     1;
}

public class Derived extends Base {
     public int value = 2;

      public void doSomething()
           {  int value = 3;
           System.out.println(value);
            System.out.println(this.value);
            System.out.println(super.value)
            ;
      }
}
```

```
>3
 2
 1
```

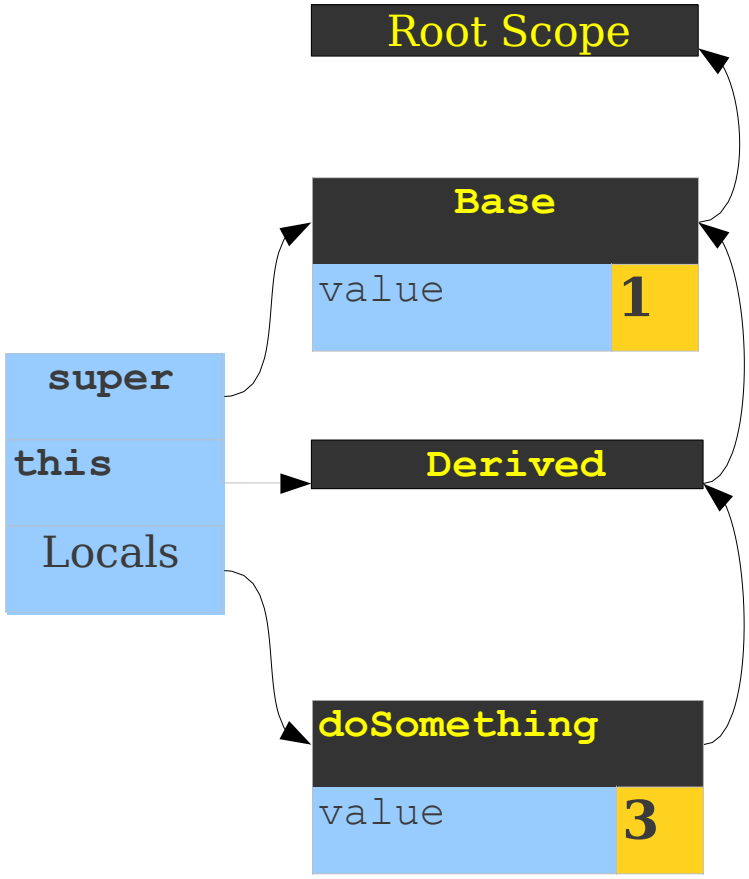


Disambiguating Scopes

- Maintain a second table of pointers into the scope stack.
- When looking up a value in a specific scope, begin the search from that scope.
- Some languages allow you to jump up to any arbitrary base class (for example, C++).

Explicit Disambiguation

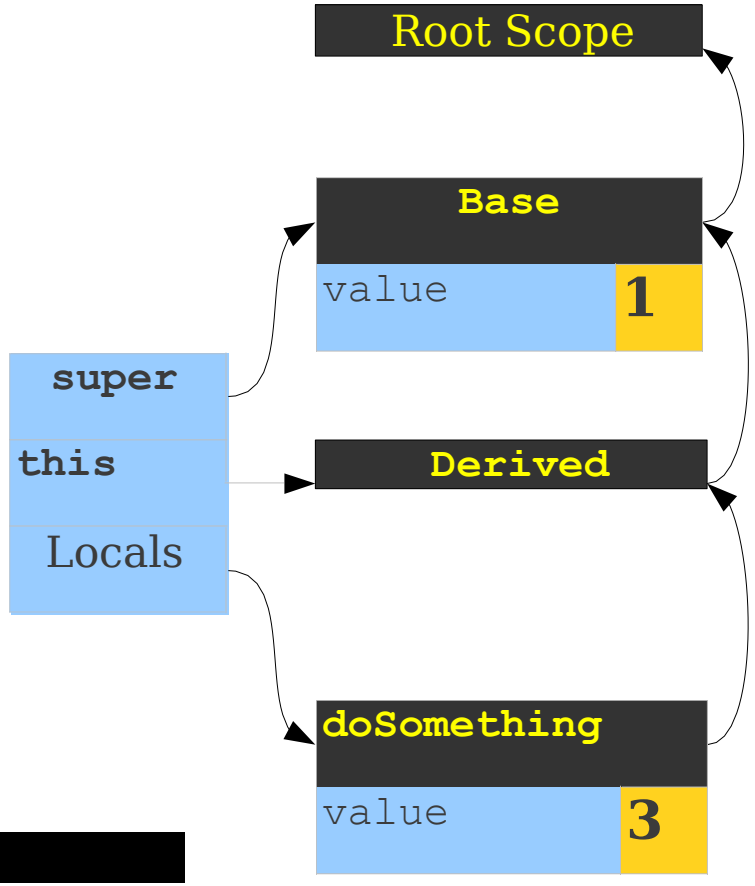
```
public class Base
     {  public int value =
     1;
}

public class Derived extends Base {

     public void doSomething()
          {  int value = 3;
          System.out.println(value);
           System.out.println(this.value);
           System.out.println(super.value);
     }
}
```



Explicit Disambiguation

```
public class Base
     {  public int value =
     1;
}

public class Derived extends Base {

     public void doSomething()
          {  int value = 3;
          System.out.println(value);
           System.out.println(this.value);
           System.out.println(super.value);
     }
}
```

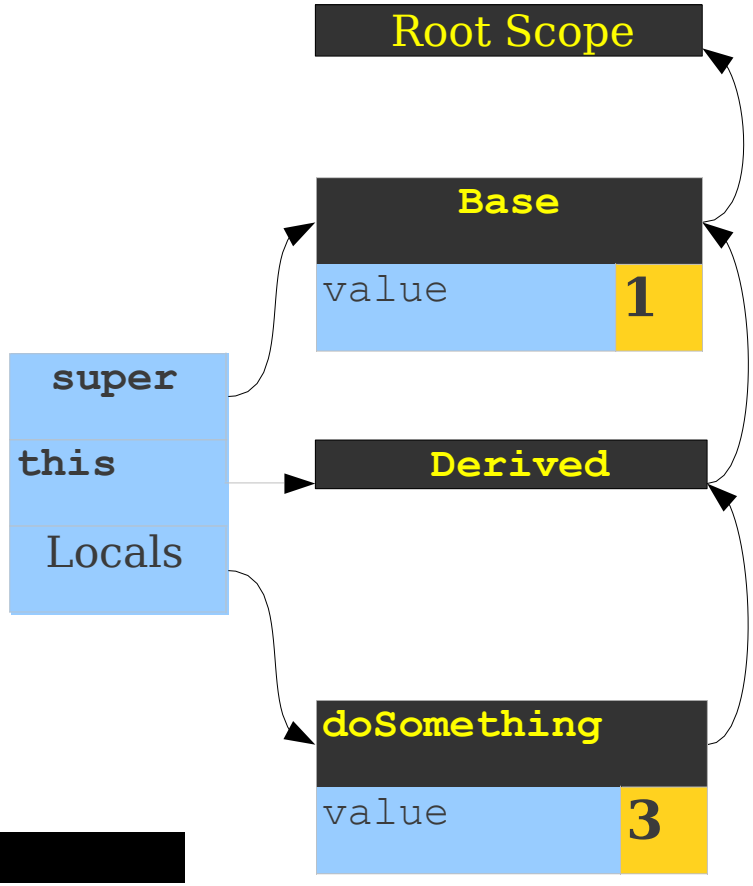
```
>
```



Explicit Disambiguation

```
public class Base
     {  public int value =
     1;
}

public class Derived extends Base {

     public void doSomething()
          {  int value = 3;
          System.out.println(value);
           System.out.println(this.value);
           System.out.println(super.value);
     }
}
```

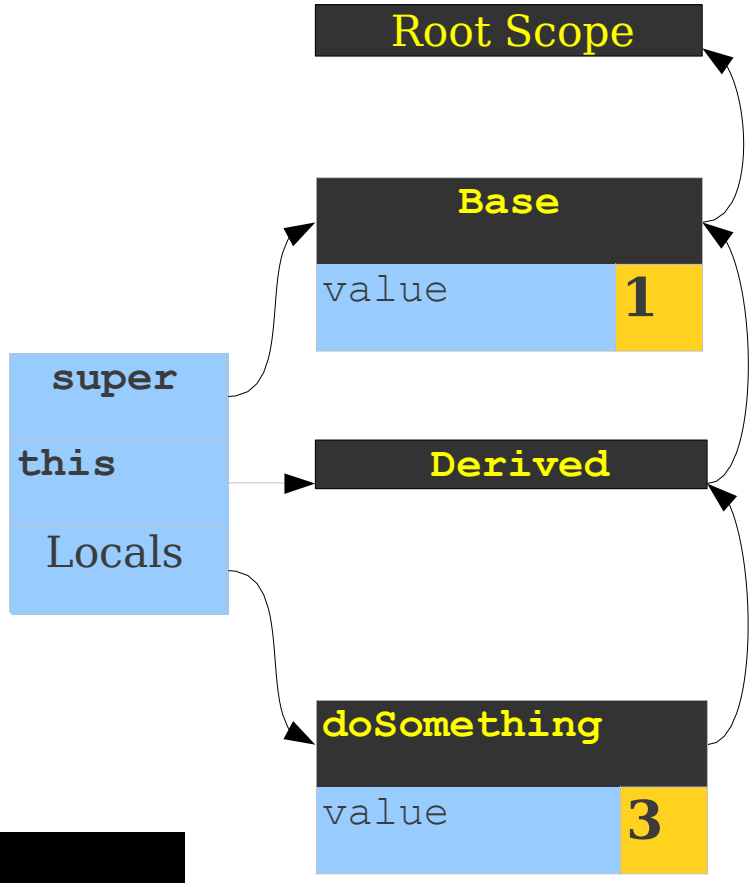
```
>
```



Explicit Disambiguation

```
public class Base
     {  public int value =
     1;
}

public class Derived extends Base {

     public void doSomething()
          {  int value = 3;
          System.out.println(value);
           System.out.println(this.value);
           System.out.println(super.value);
     }
}
```

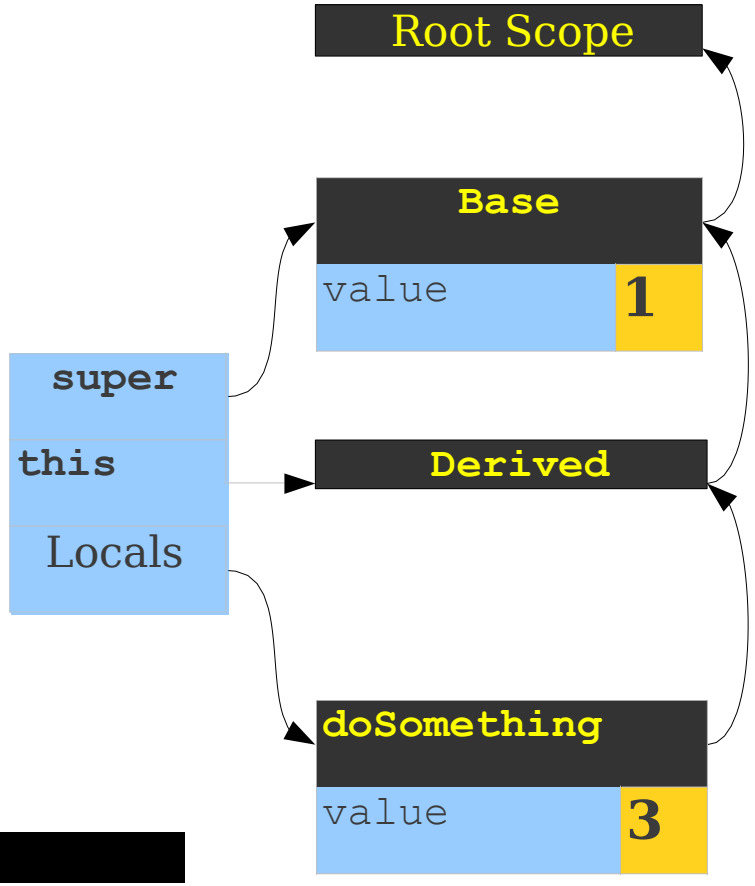
```
> 3
```



Explicit Disambiguation

```
public class Base
     {  public int value =
     1;
}

public class Derived extends Base {

     public void doSomething()
          {  int value = 3;
          System.out.println(value);
          System.out.println(this.value);
          System.out.println(super.value);
     }
}
```

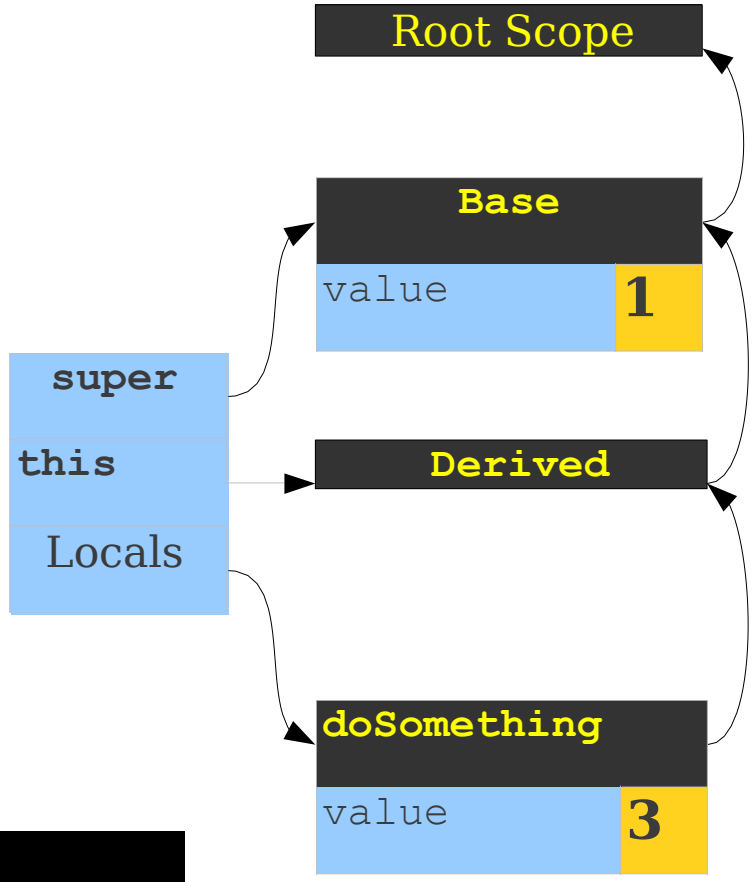
```
> 3
```



Explicit Disambiguation

```
public class Base
     {  public int value =
     1;
}

public class Derived extends Base {

     public void doSomething()
          {  int value = 3;
          System.out.println(value);
          System.out.println(this.value);
          System.out.println(super.value);
     }
}
```

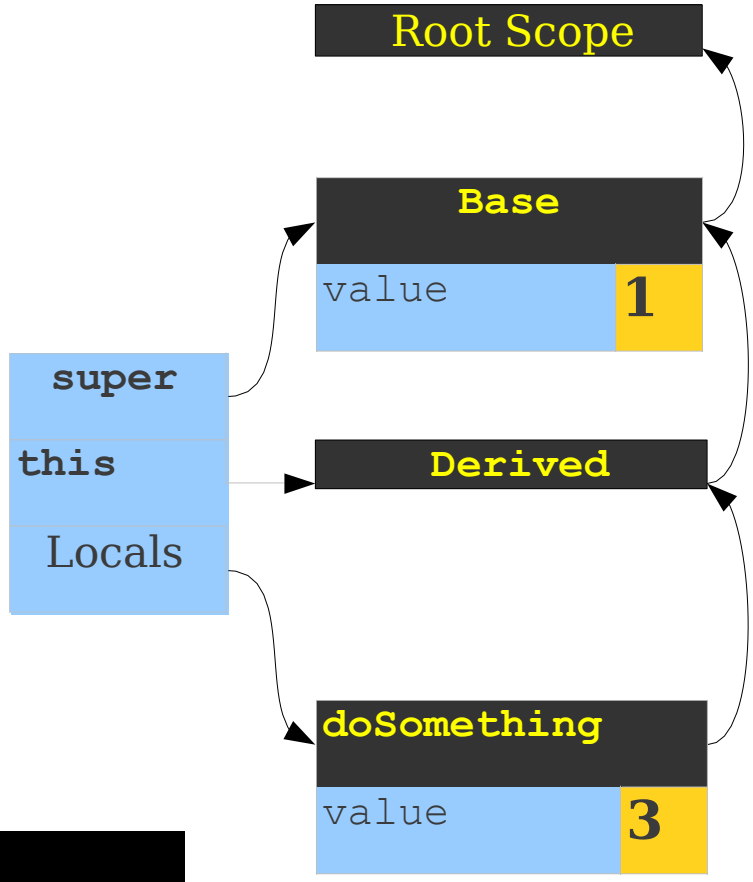
```
>3
 1
```



Explicit Disambiguation

```
public class Base
     {  public int value =
     1;
}

public class Derived extends Base {

     public void doSomething()
          {  int value = 3;
          System.out.println(value);
           System.out.println(this.value);
           System.out.println(super.value);
     }
}
```

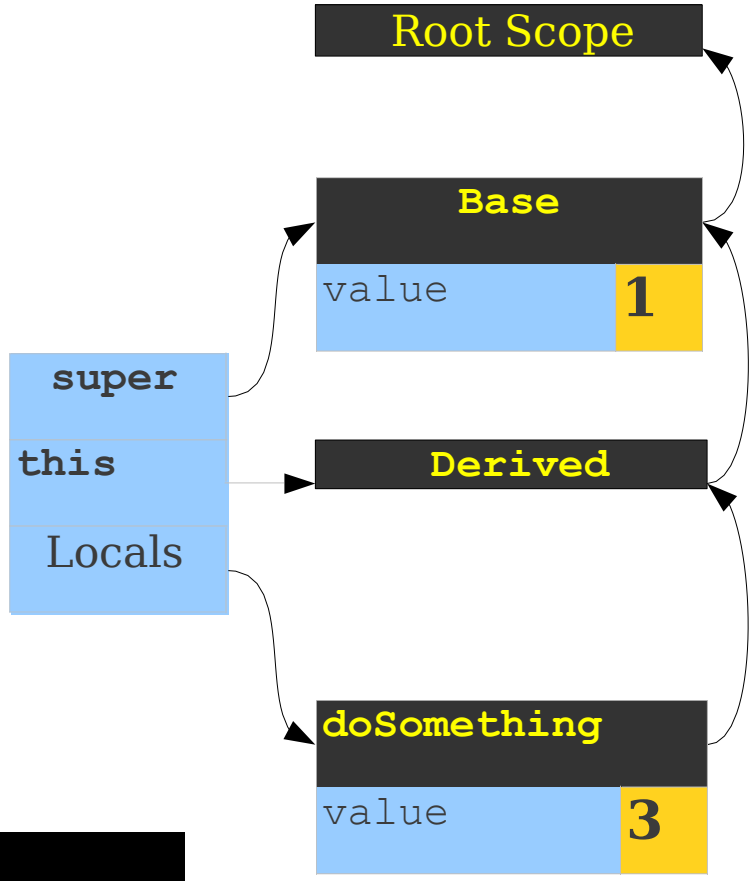
```
>3
 1
```



Explicit Disambiguation

```
public class Base
     {  public int value =
     1;
}

public class Derived extends Base {

     public void doSomething()
          {  int value = 3;
          System.out.println(value);
           System.out.println(this.value);
           System.out.println(super.value);
     }
}
```

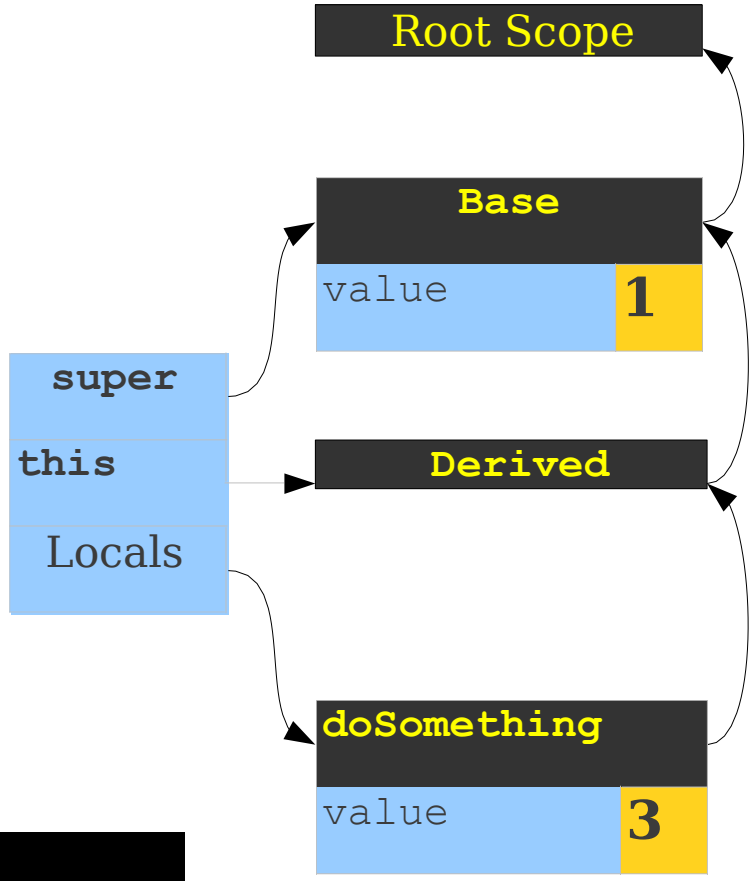
```
>3
 1
 1
```



Explicit Disambiguation

```
public class Base
     {  public int value =
     1;
}

public class Derived extends Base {

     public void doSomething()
          {  int value = 3;
          System.out.println(value);
           System.out.println(this.value);
           System.out.println(super.value);
     }
}
```

```
>3
 1
 1
```



Scoping in Practice

Scoping in C++ and Java

```
class A {
public:
     /* … */

private:
     B* myB
};

class B {
public:
     /* … */

private:
     A* myA;
};
```

```
class A {
     private B myB;
};

class B {
     private A myA;
};
```

Scoping in C++ and Java

```
class A {
public:
     /* … */

private:
     B* myB
};

class B {
public:
     /* … */

private:
     A* myA;
};
```

Error: B not declared

```
class A {
     private B myB;
};

class B {
     private A myA;
};
```

Perfectly fine!

Single- and Multi-Pass Compilers

- Our predictive parsing methods always scan the input from left-to-right.
  - **L**L(1), **L**R(0), LA**L**R(1), etc.
- Since we only need one token of lookahead, we can do scanning and parsing simultaneously in one pass over the file.
- Some compilers can combine scanning, parsing, semantic analysis, and code generation into the same pass.
  - These are called **single-pass compilers**.
- Other compilers rescan the input multiple times.
  - These are called **multi-pass compilers**.

Single- and Multi-Pass Compilers

- Some languages are designed to support single-pass compilers.
  - e.g. C, C++.
- Some languages *require* multiple passes.
  - e.g. Java, **Decaf**.
- Most modern compilers use a huge number of passes over the input.

Scoping in Multi-Pass Compilers

- Completely parse the input file into an abstract syntax tree (first pass).
- Walk the AST, gathering information about classes (second pass).
- Walk the AST checking other properties (third pass).
- Could combine some of these, though they are logically distinct.

Scoping with Multiple Inheritance

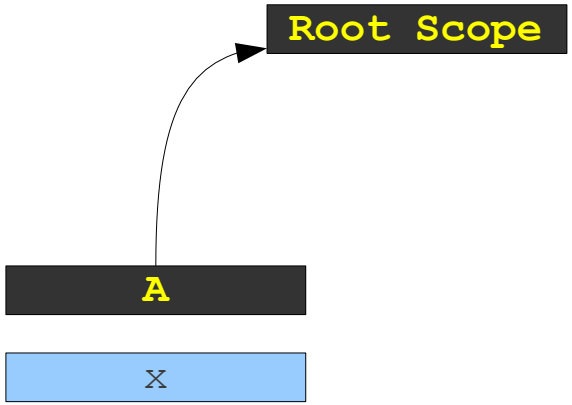
```
class A {
public:
    int x;
};

class B {


};

class C: public A, public B {
public:
    void doSomething() {
        cout << x <<
        endl;
    }
}
```

Scoping with Multiple Inheritance

Root Scope

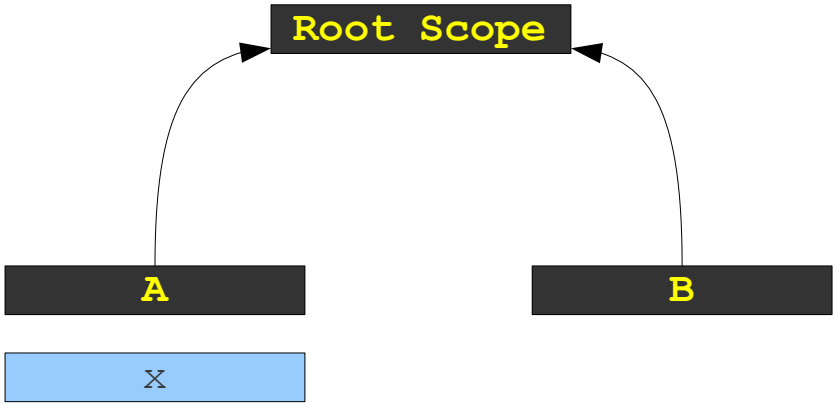
```
class A {
public:
     int x;
};

class B {


};

class C: public A, public B {
public:
     void doSomething() {
          cout << x <<
          endl;
     }
}
```

Scoping with Multiple Inheritance



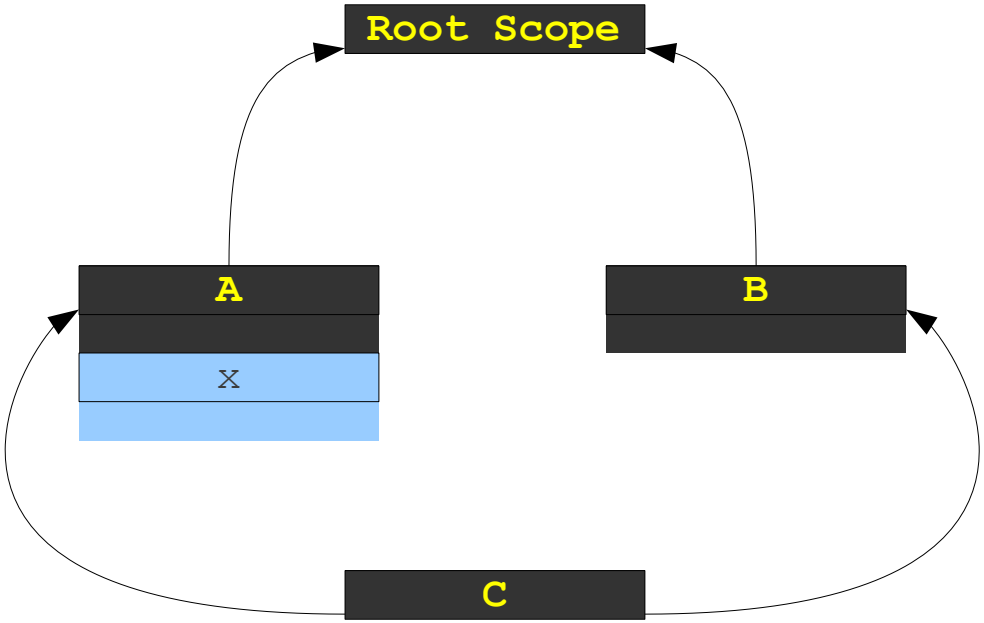

```
class A {
public:
     int
     x;
};
class B {


};

class C: public A, public B {
public:
     void doSomething() {
          cout << x <<
          endl;
     }
}
```

Scoping with Multiple Inheritance




```
class A {
public:
     int
     x;
};
class B {


};

class C: public A, public B {
public:
     void doSomething() {
          cout << x <<
          endl;
     }
}
```

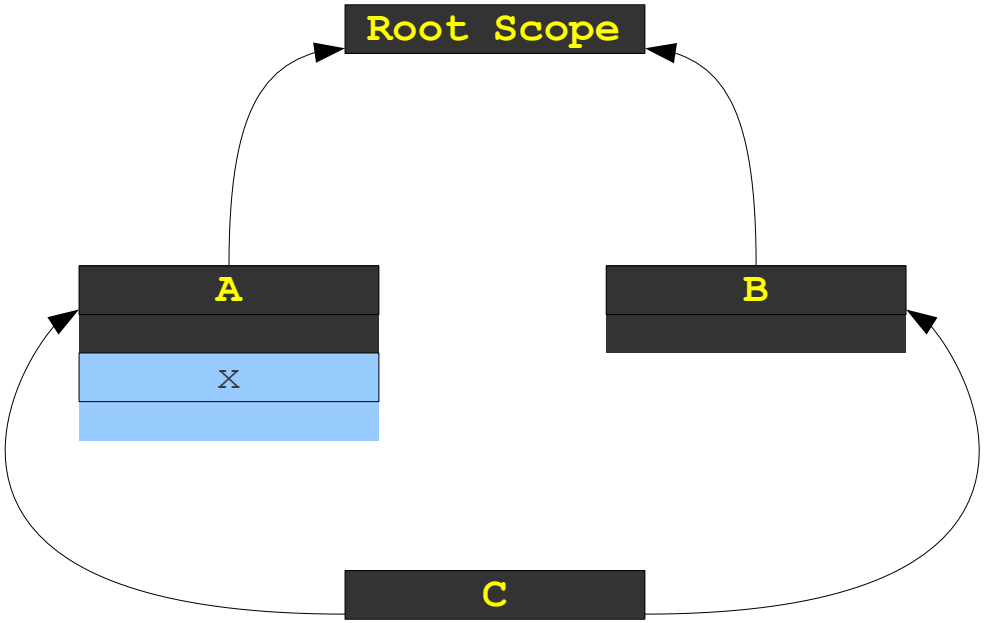
Scoping with Multiple Inheritance




```
class A {
public:
     int
     x;
};
class B {


};

class C: public A, public B {
public:
     void doSomething() {
          cout << x <<
          endl;
     }
}
```

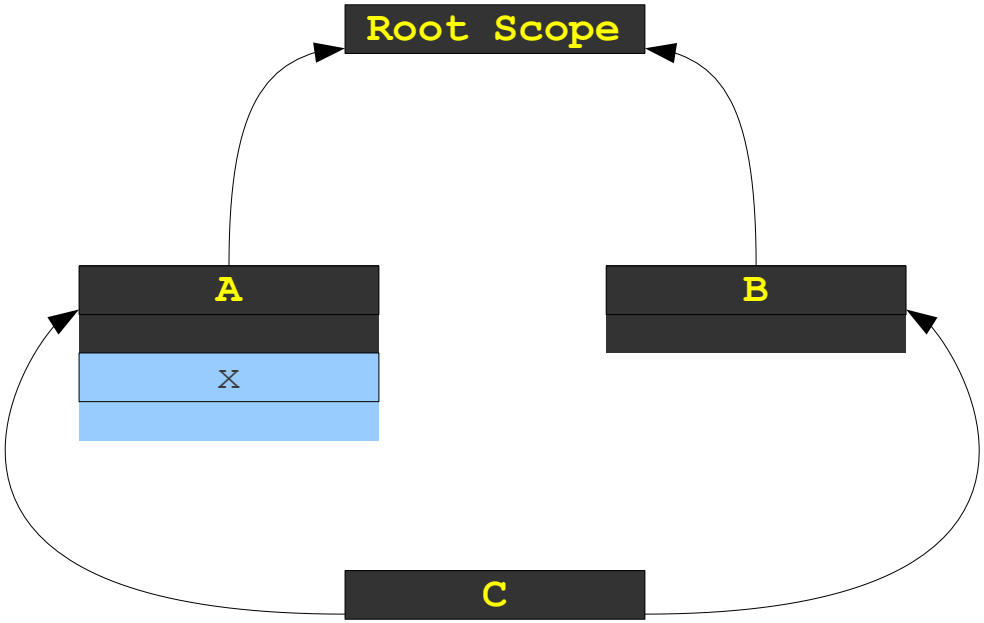
Scoping with Multiple Inheritance

```
class A {
public:
     int
     x;
};
class B {


};

class C: public A, public B {
public:
     void doSomething() {
          cout << x <<
          endl;
     }
}
```

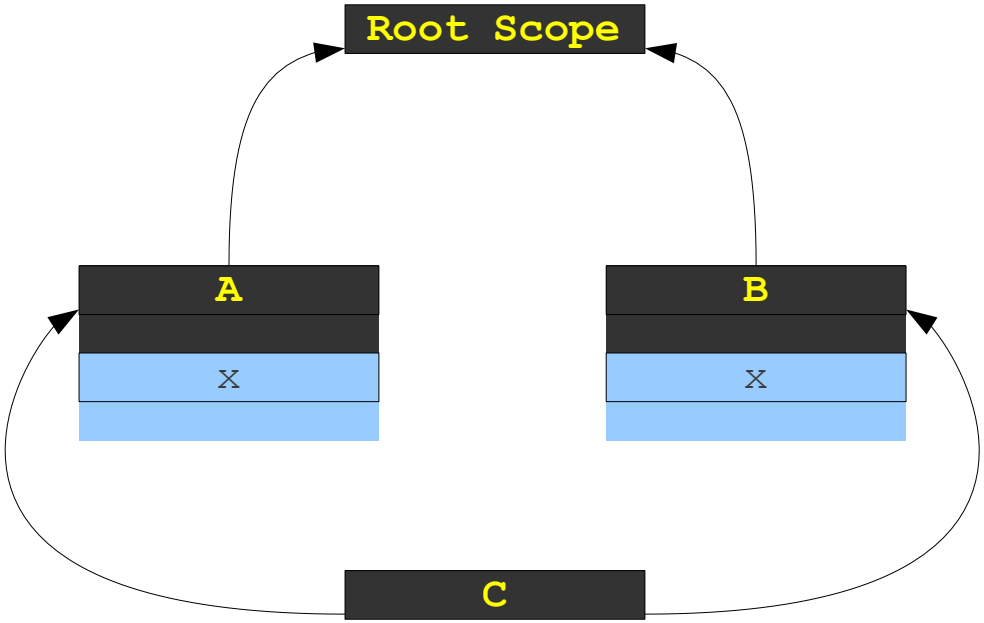


Scoping with Multiple Inheritance

```
class A {
public:
     int
     x;
};
class B {
public:
     int
     x;
};
class C: public A, public B {
public:
     void doSomething() {
          cout << x <<
          endl;
     }
}
```

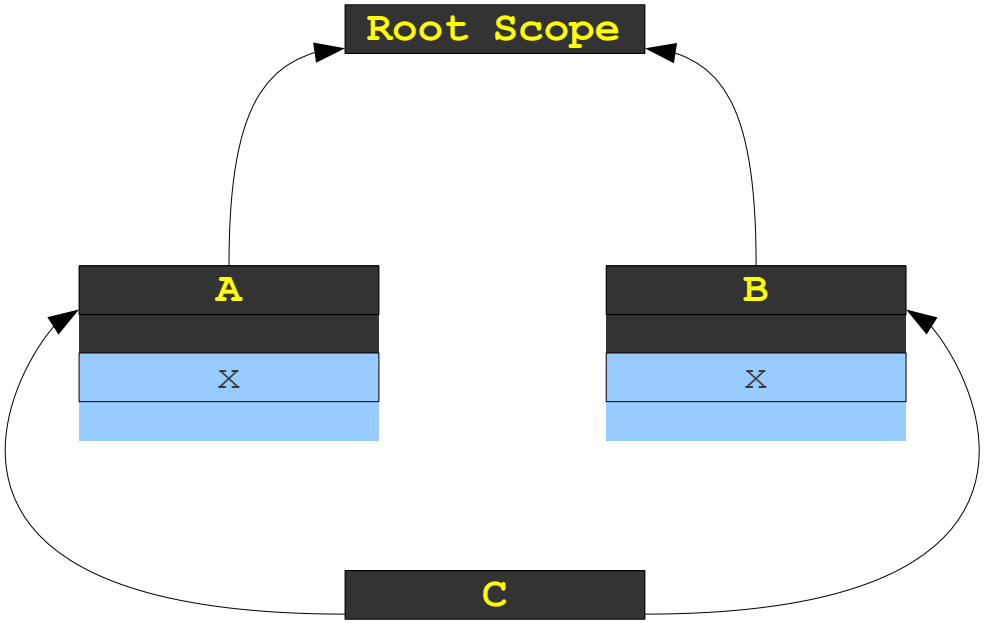


Scoping with Multiple Inheritance




```
class A {
public:
     int
     x;
};
class B {
public:
     int
     x;
};
class C: public A, public B {
public:
     void doSomething() {
          cout << x <<
          endl;
     }
}
```

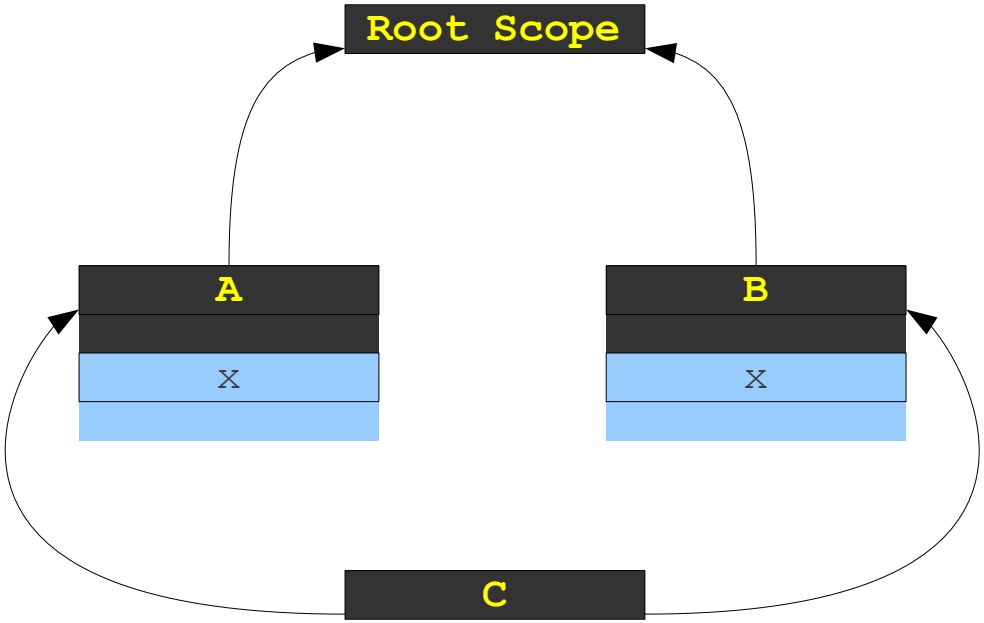
Scoping with Multiple Inheritance




```
class A {
public:
     int
     x;
};
class B {
public:
     int
     x;
};
class C: public A, public B {
public:
     void doSomething() {
          cout << x <<
          endl;
     }
}
```

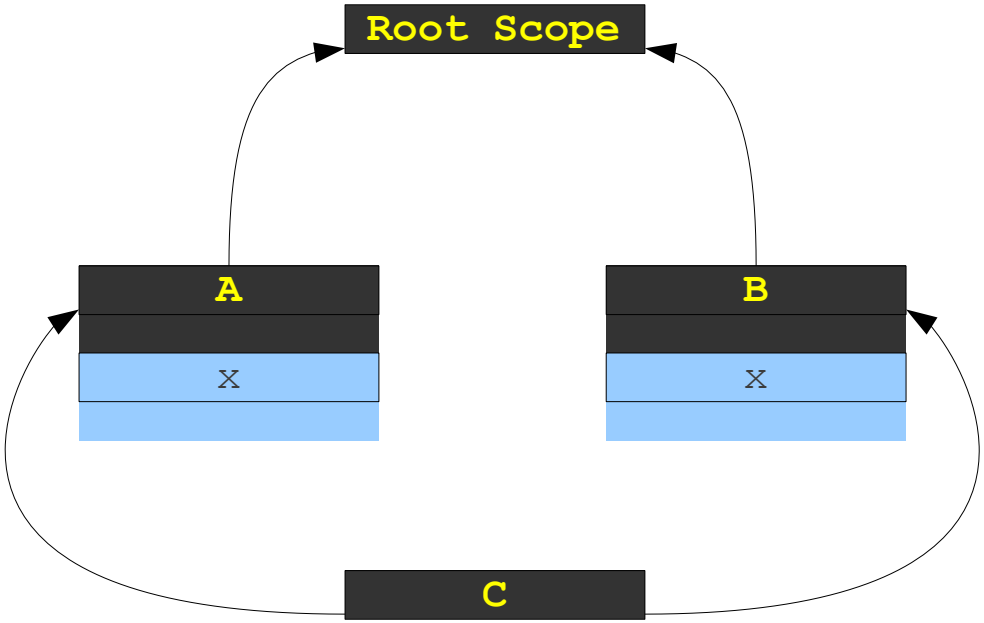
Scoping with Multiple Inheritance

```
class A {
public:
     int
     x;
};
class B {
public:
     int
     x;
};
class C: public A, public B {
public:
     void doSomething() {
          cout << x << endl;
     }
}
```



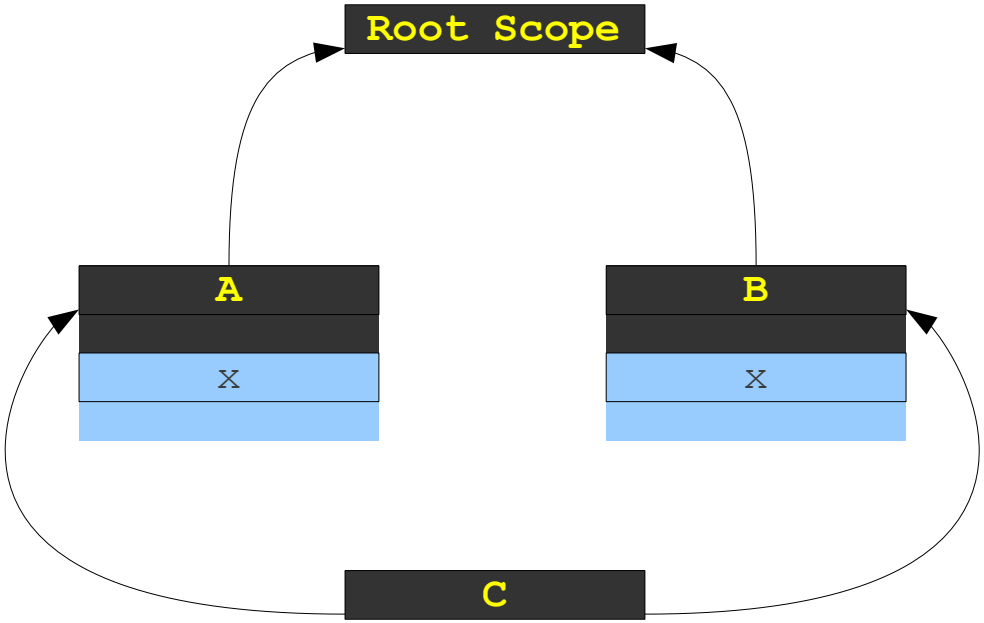
Ambiguous – which x?

Scoping with Multiple Inheritance




```
class A {
public:
     int
     x;
};
class B {
public:
     int
     x;
};
class C: public A, public B {
public:
     void doSomething() {
          cout << A::x << endl;
     }
}
```

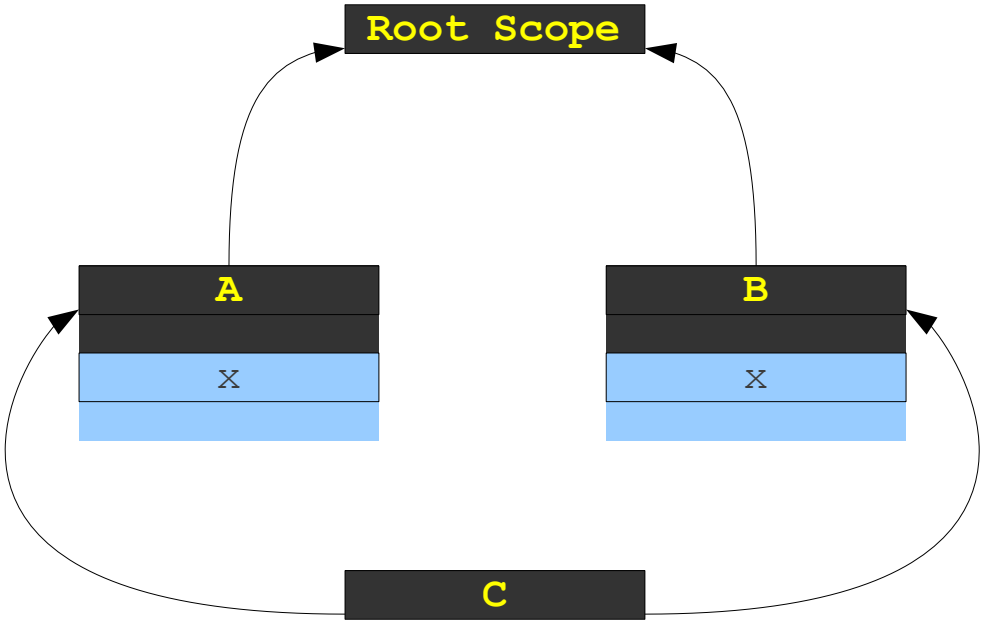
Scoping with Multiple Inheritance



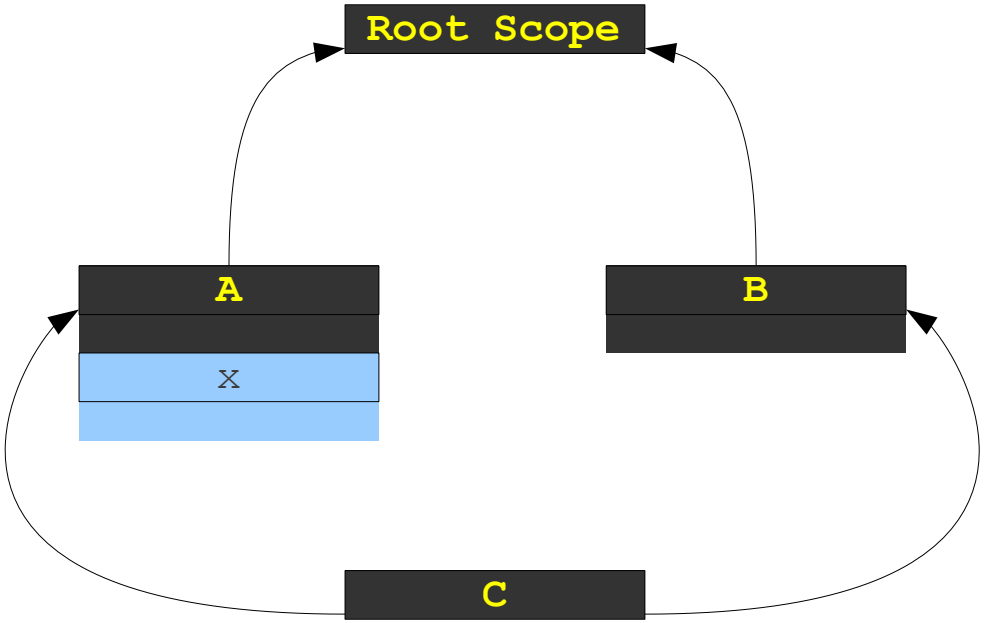

```
class A {
public:
     int
     x;
};
class B {
public:
     int
     x;
};
class C: public A, public B {
public:
     void doSomething() {
          cout << x <<
          endl;
     }
}
```

Scoping with Multiple Inheritance

```
class A {
public:
     int
     x;
};
class B {


};

class C: public A, public B {
public:
     void doSomething() {
          cout << x <<
          endl;
     }
}
```



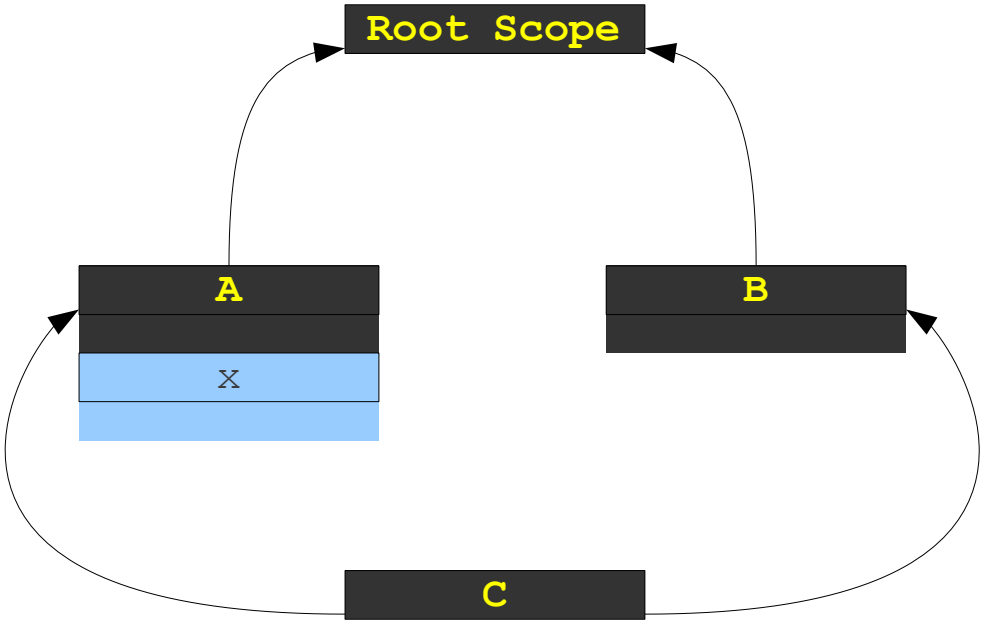
Scoping with Multiple Inheritance




```
class A {
public:
     int
     x;
};
class B {


};

class C: public A, public B {
public:
     void doSomething() {
          cout << x <<
          endl;
     }
}
```

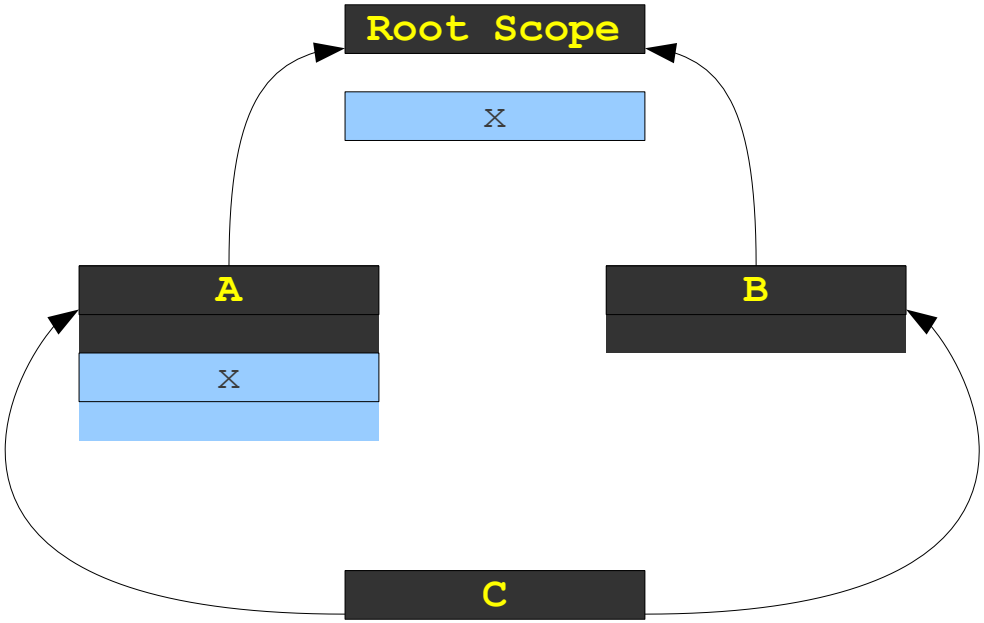
Scoping with Multiple Inheritance

```
int x;

class A {
public:
     int
     x;
};
class B {


};

class C: public A, public B {
public:
     void doSomething() {
          cout << x <<
          endl;
     }
}
```



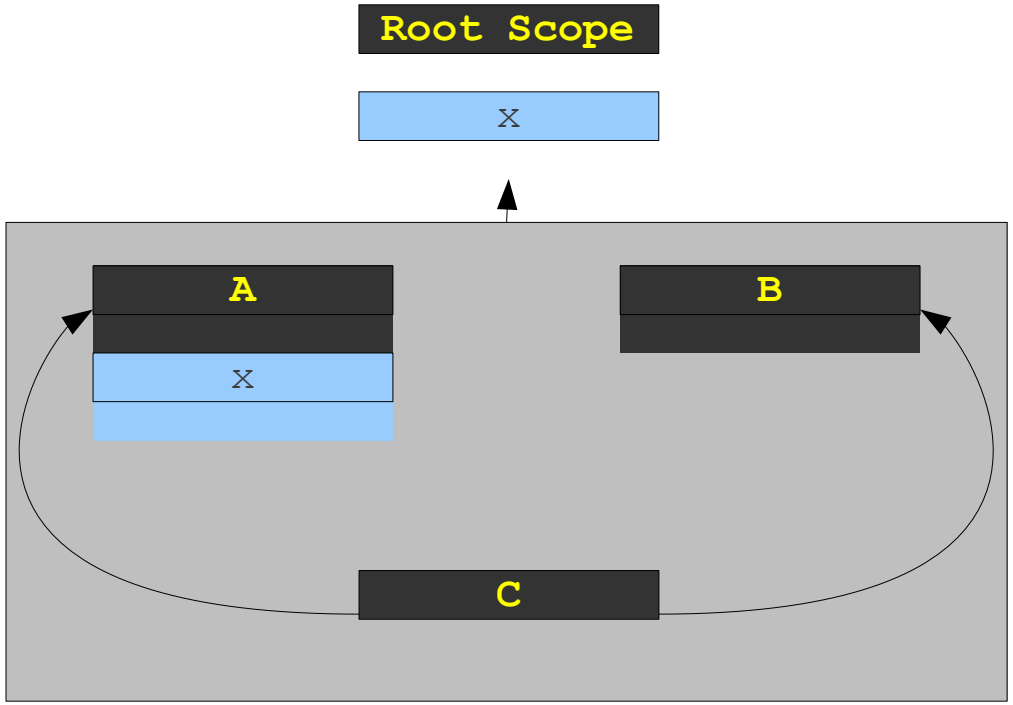
Scoping with Multiple Inheritance

```
int x;

class A {
public:
     int
     x;
};
class B {


};

class C: public A, public B {
public:
     void doSomething() {
          cout << x <<
          endl;
     }
}
```



Scoping with Multiple Inheritance

```
int x;

class A {
public:
     int
     x;
};
class B {


};

class C: public A, public B {
public:
     void doSomething() {
          cout << x <<
          endl;
     }
}
```



(Simplified) C++ Scoping Rules

- Inside of a class, search the entire class hierarchy to see the set of names that can be found.
  - This uses the standard scoping lookup.
- If only one name is found, the lookup succeeds unambiguously.
- If more than one name is found, the lookup is ambiguous and requires disambiguation.
- Otherwise, restart the search from outside the class.

Dynamic Scoping

Static and Dynamic Scoping

- The scoping we've seen so far is called **static scoping** and is done at compile-time.
  - Names refer to lexically related variables.
- Some languages use **dynamic scoping**, which is done at runtime.
  - Names refer to the variable with that name that is *most closely nested at runtime*.

Dynamic Scoping

```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```

Dynamic Scoping

```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	**137**
y	**42**

```
>
```

Dynamic Scoping

```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	**137**
y	**42**

```
>
```

Dynamic Scoping

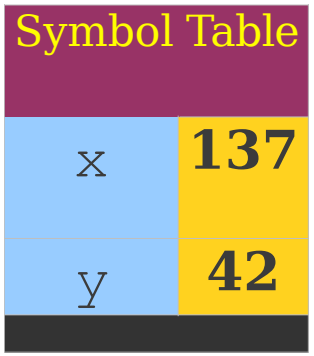
```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	**137**
y	**42**

```
>
```

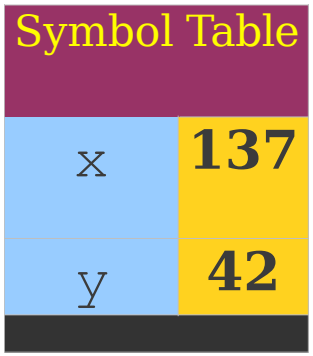
Dynamic Scoping

```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```



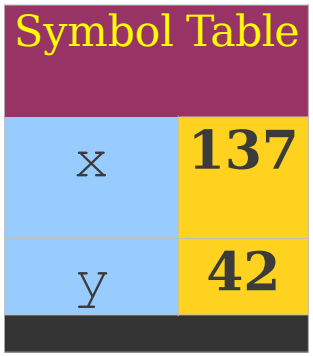
Dynamic Scoping

```
int  x  = 137;
int  y  = 42;
void Function1() {
     Print(x + y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```



Dynamic Scoping

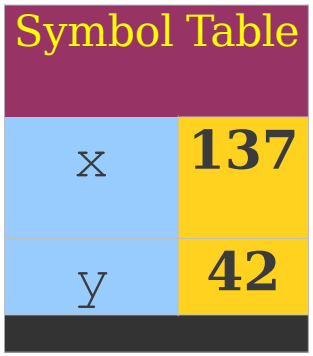
```
int  x  = 137;
int  y  = 42;
void Function1() {
     Print(x + y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```




```
> 179
>
```

Dynamic Scoping

```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```




```
> 179
>
```

Dynamic Scoping

```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	**137**
y	**42**

```
> 179
>
```

Dynamic Scoping

```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	**137**
y	**42**

```
> 179
>
```

Dynamic Scoping

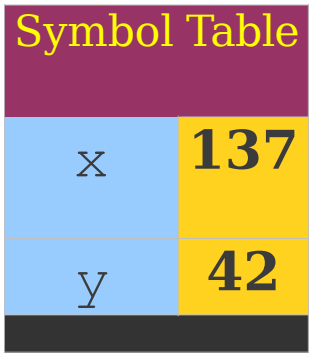
```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	**137**
y	**42**

```
> 179
>
```

Dynamic Scoping

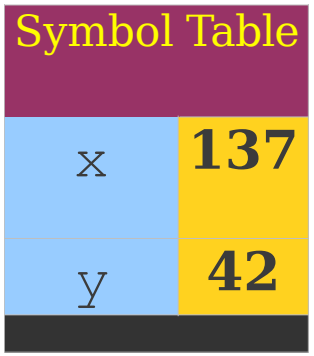
```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```




```
> 179
>
```

Dynamic Scoping

```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```




```
> 179
>
```

Dynamic Scoping

```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	137
y	42
x	0

```
> 179
>
```

Dynamic Scoping

```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	137
y	42
x	0

```
> 179
>
```

Dynamic Scoping

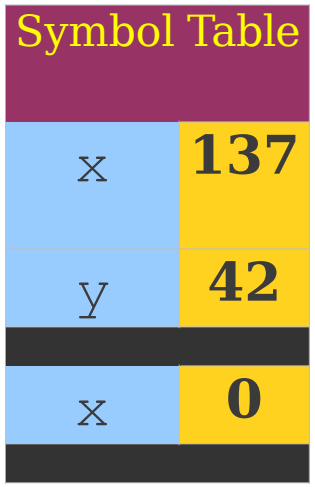
```
int  x  = 137;
int  y  = 42;
void Function1() {
     Print(x + y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	137
y	42
x	0

```
> 179
>
```

Dynamic Scoping

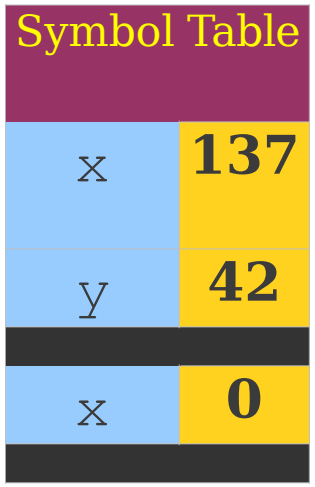
```
int  x  = 137;
int  y  = 42;
void Function1() {
     Print(x + y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```




```
> 179
>
```

Dynamic Scoping

```
int  x  = 137;
int  y  = 42;
void Function1() {
     Print(x + y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```




```
> 179
>
```

Dynamic Scoping

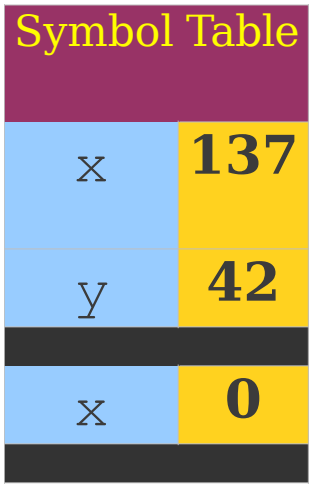
```
int  x  = 137;
int  y  = 42;
void Function1() {
     Print(x + y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	137
y	42
x	0

```
> 179
> 42
>
```

Dynamic Scoping

```
 int  x  = 137;
 int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```




```
> 179
> 42
>
```

Dynamic Scoping

```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	137
y	42
x	0

```
> 179
> 42
>
```

Dynamic Scoping

```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	137
y	42
x	0

```
> 179
> 42
>
```

Dynamic Scoping

```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	**137**
y	**42**

```
> 179
> 42
>
```

Dynamic Scoping

```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	**137**
y	**42**

```
> 179
> 42
>
```

Dynamic Scoping

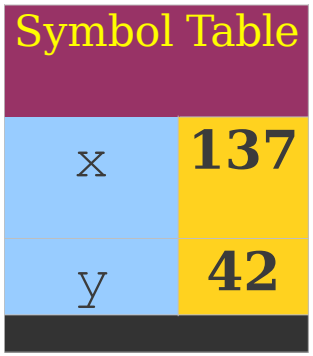
```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	**137**
y	**42**

```
> 179
> 42
>
```

Dynamic Scoping

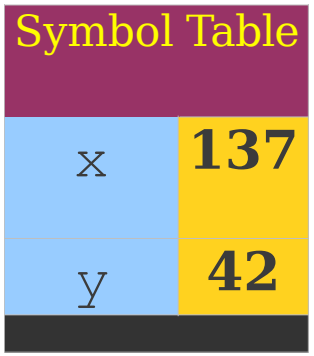
```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```




```
> 179
> 42
>
```

Dynamic Scoping

```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```




```
> 179
> 42
>
```

Dynamic Scoping

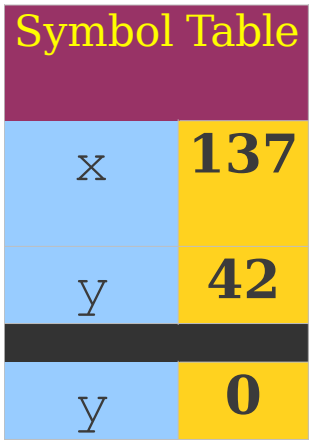
```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	137
y	42
y	0

```
> 179
> 42
>
```

Dynamic Scoping

```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```




```
> 179
> 42
>
```

Dynamic Scoping

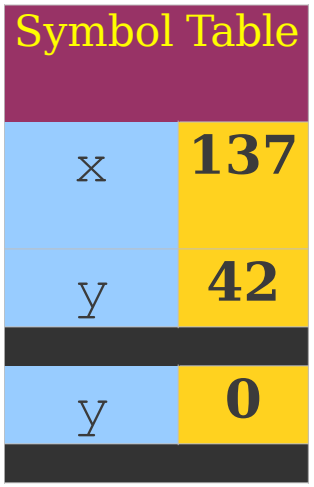
```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	137
y	42
y	0

```
> 179
> 42
>
```

Dynamic Scoping

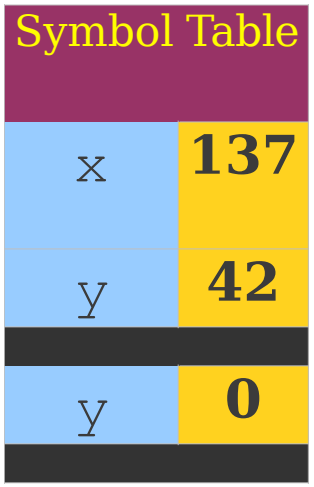
```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```




```
> 179
> 42
>
```

Dynamic Scoping

```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```




```
> 179
> 42
>
```

Dynamic Scoping

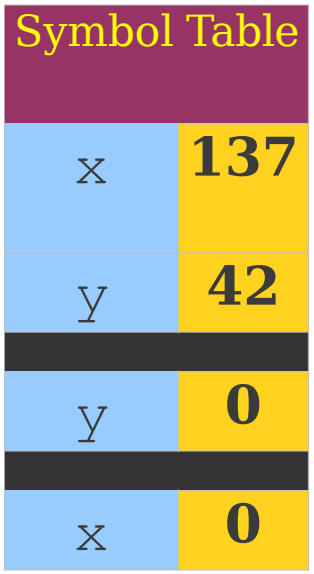
```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	137
y	42
y	0
x	0

```
> 179
> 42
>
```

Dynamic Scoping

```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```




```
> 179
> 42
>
```

Dynamic Scoping

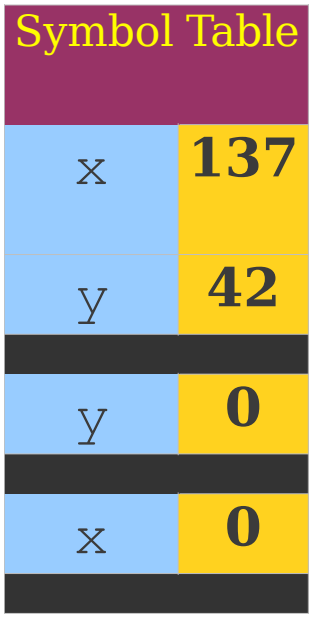
```
int  x  = 137;
int  y  = 42;
void Function1() {
     Print(x + y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	137
y	42
y	0
x	0

```
> 179
> 42
>
```

Dynamic Scoping

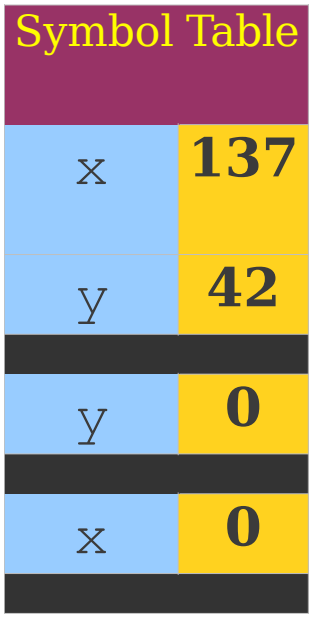
```
int  x  = 137;
int  y  = 42;
void Function1() {
     Print(x + y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```




```
> 179
> 42
>
```

Dynamic Scoping

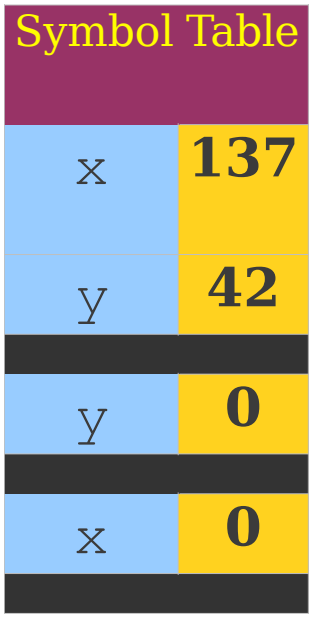
```
int  x  = 137;
int  y  = 42;
void Function1() {
     Print(x + y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```




```
> 179
> 42
>
```

Dynamic Scoping

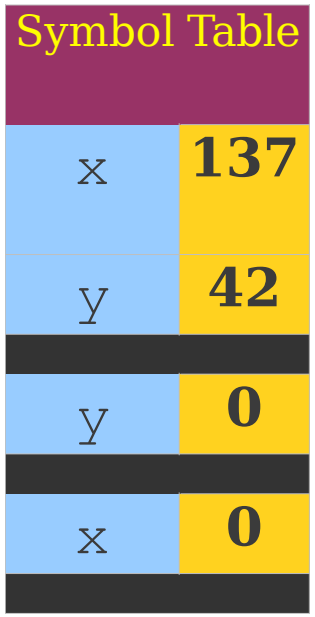
```
int  x  = 137;
int  y  = 42;
void Function1() {
     Print(x + y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```




```
> 179
> 42
> 0
>
```

Dynamic Scoping

```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```




```
> 179
> 42
> 0
>
```

Dynamic Scoping

```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	137
y	42
y	0
x	0

```
> 179
> 42
> 0
>
```

Dynamic Scoping

```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	137
y	42
y	0
x	0

```
> 179
> 42
> 0
>
```

Dynamic Scoping

```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	137
y	42
y	0

```
> 179
> 42
> 0
>
```

Dynamic Scoping

```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	137
y	42
y	0

```
> 179
> 42
> 0
>
```

Dynamic Scoping

```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	**137**
y	**42**

```
> 179
> 42
> 0
>
```

Dynamic Scoping

```
int  x  = 137;
int  y  = 42;
void Function1()
     {  Print(x +
     y);
}
void Function2()
     {  int x = 0;
     Function1();
}
void Function3()
     {  int y = 0;
     Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	**137**
y	**42**

```
> 179
> 42
> 0
>
```

Dynamic Scoping in Practice

- Examples: Perl, Common LISP.
- Often implemented by preserving symbol table at runtime.
- Often less efficient than static scoping.
  - Compiler cannot “hardcode” locations of variables.
  - Names must be resolved at runtime.

Summary

- **Semantic analysis** verifies that a syntactically valid program is correctly-formed and computes additional information about the meaning of the program.
- **Scope checking** determines what objects or classes are referred to by each name in the program.
- Scope checking is usually done with a **symbol table** implemented either as a stack or **spaghetti stack**.
- In object-oriented programs, the scope for a derived class is often placed inside of the scope of a base class.
- Some semantic analyzers operate in multiple passes in order to gain more information about the program.
- In dynamic scoping, the actual execution of a program determines what each name refers to.
- With multiple inheritance, a name may need to be searched for along multiple paths.

Next Time

Type Checking

- Types as a proof system.
- Static and dynamic types.
- Types as a partial order.