Lecture 06

Semantic Analysis:

Symbol Table & Scope Checking

Outline

- Semantic Analysis
  - Overview of Semantic Analysis
  - Attributes and Attribute Grammars
  - Dependency Graphs and Algorithms for Attribute Computation
  - Symbol Table and Scope Checking
  - Type Checking for Semantic Analysis of a Program

Symbol Table

- The data structure that is created and maintained by the compilers for information storing regarding the occurrence of various entities like names of variables, functions, objects, classes

Symbol Table

- A symbol table may serve the following purposes depending upon the language in hand
  - To store the names of all entities in a structured form at one place
  - To verify if a variable has been declared
  - Type checking: verifying assignments and expressions in the source code are semantically correct
  - To determine the scope of a name (scope resolution)
  - Code Generation: generating call instructions based on the storage location of the function.

Symbol Table

- The following possible information about identifiers are stored in symbol table
  - The name (as a string)
  - Attribute: Variable name, Procedure name, Constant name
  - The data type
  - The block level
  - Its scope (global, local, or parameter)
  - Location in memory

Symbol Table

- Implementation
  - Symbol table can be mostly implemented as Hash table
  - The source code symbol itself is treated as a key for the hash function
  - the return value is the information about the symbol.
- Entry format
  - A symbol table maintains an entry for each name, usually in the following format:

    <symbol name, type, attribute>

Example :Declaration part of a program

```
CONST
    A : Integer = 35;
    B : Real = 49.0;
VAR
    C : char;
PROCEDURE Add;
    VAR G : Integer
```

Name	Kind	Data Type	value	scope	location	size
A	Constant	Integer	35	Lev	DX+offset1	4
B	Constant	Real	49.0	Lev	DX+offset2	8
C	Variable	char	--	Lev	DX+offset3	1
Add	PROCEDURE		--	Lev	BP	4
G	Variable	Integer	--	Lev+1	BP+offset4	4

The entries in the symbol table might look like this:

```
struct SymbolEntry {
    std::string name;
    std::string type;
    std::string scope;
    std::string segment;
    int offset;
    int size;                   };
```

```
SymbolEntry entryName = {
    .name = “A",
    .type = "int",
    .scope = "GLOBAL",
    .segment = ".data",
    .offset = 0x1000,
    .size = 4
};
```

Or like this:

```
ConstantEntry consEntry = {
        .name = “A",
        .type = "int",
        .value = 35,
        .scope = GLOBAL,
       .storageLocation = { .segment = "data", .offset = 0x1000 }
};

FunctionEntry funEntry = {
        .name = "Add",
        .returnType = "int",
        .parameters = {       { .name = "a", .type = "int" },
                              { .name = "b", .type = "int" }    },
        .scope = GLOBAL,
        .address = { .segment = "code", .offset = 0x1234 }
};
```

Symbol Table

- Typical operations
  - insert
    - used to store the information provided by name declarations when processing these declarations
    - more frequently used during analysis phase where tokens are identified
    - takes the symbol and its attributes as arguments
  - delete
    - remove the information provided by a declaration when that declaration no longer applies

Symbol Table

- Typical operations
  - search(lookup)
    - used to search a name in the symbol table to determine:
      - if the symbol exists in the table.
      - if it is declared before it is being used.
      - if the name is used in the scope.
      - if the symbol is initialized.
      - if the symbol declared multiple times.

Symbol Table

- The timing of when identifiers are recorded in the symbol table
  - Identifiers are entered into the symbol table, typically not during the Lexical Analysis phase.
  - During the Syntax Analysis and Semantic Analysis phases, construct the Abstract Syntax Tree (AST), check semantic correctness, and insert identifiers along with their associated information into the symbol table.

Scope Management

- In programming languages, **scope** defines the visibility and lifetime of identifiers.
- How do we keep track of what's visible?

Scope Management

- Static Scope and Dynamic Scope
  - Static scope (lexical scope, will be discussed in our class
    - The scope of variables being determined at compile time.
    - The visibility of variables is based on their position in the source code.
    - The nesting levels of scopes are clearly defined at compile time.
  - Dynamic Scope
    - the scope of variables being determined at runtime
    - The visibility of variables is based on the current state of the call stack.
    - The nesting levels of scopes are dynamically determined at runtime.

```
// A C program to demonstrate static scoping.
#include<stdio.h>
int x = 10;

// Called by g()
int f(){
   return x;
}

// g() has its own variable  named as x and calls f()
int g(){
   int x = 20;
   return f();
}

int main(){
  printf( "%d" , g());   //x=10
  printf("\n");
  return 0;
}
```

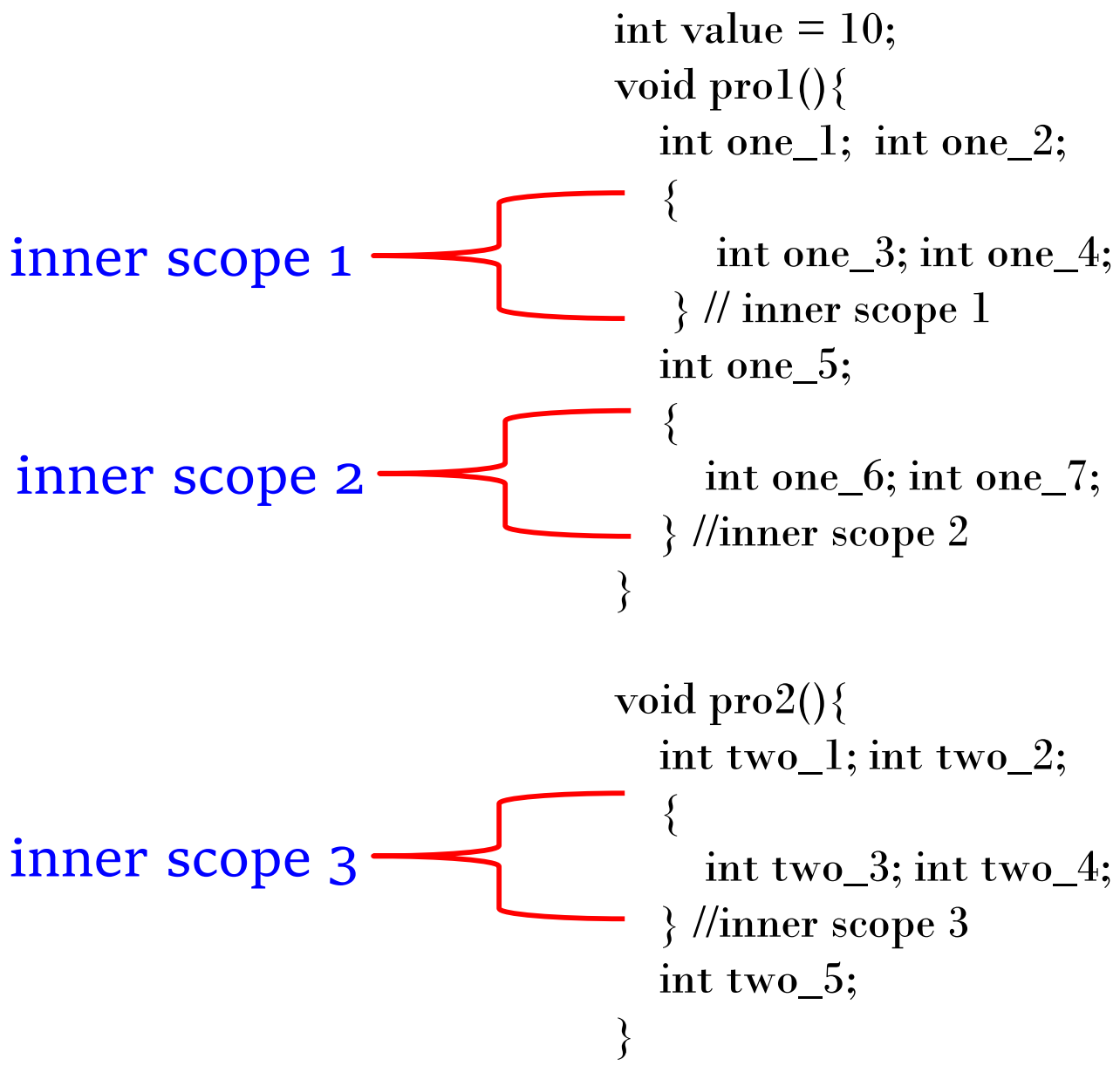
Scopes

- Scope rules in programming languages vary widely.
- Two common rules
  - declaration before use
  - Nested scope

Nested Scope

- Block structure
  - A block is any construct that can contains declarations
  - A language is block structured
    - If the nesting blocks inside other blocks is permitted
    - If the scope of declarations in a block is limited to that block and other blocks nested in that block
- **Most-closely nested rule for blocks**
  - Given several different declarations for the same name, the declaration that applies to a reference is the one in the most closely nested block to the reference.

Nested Scope



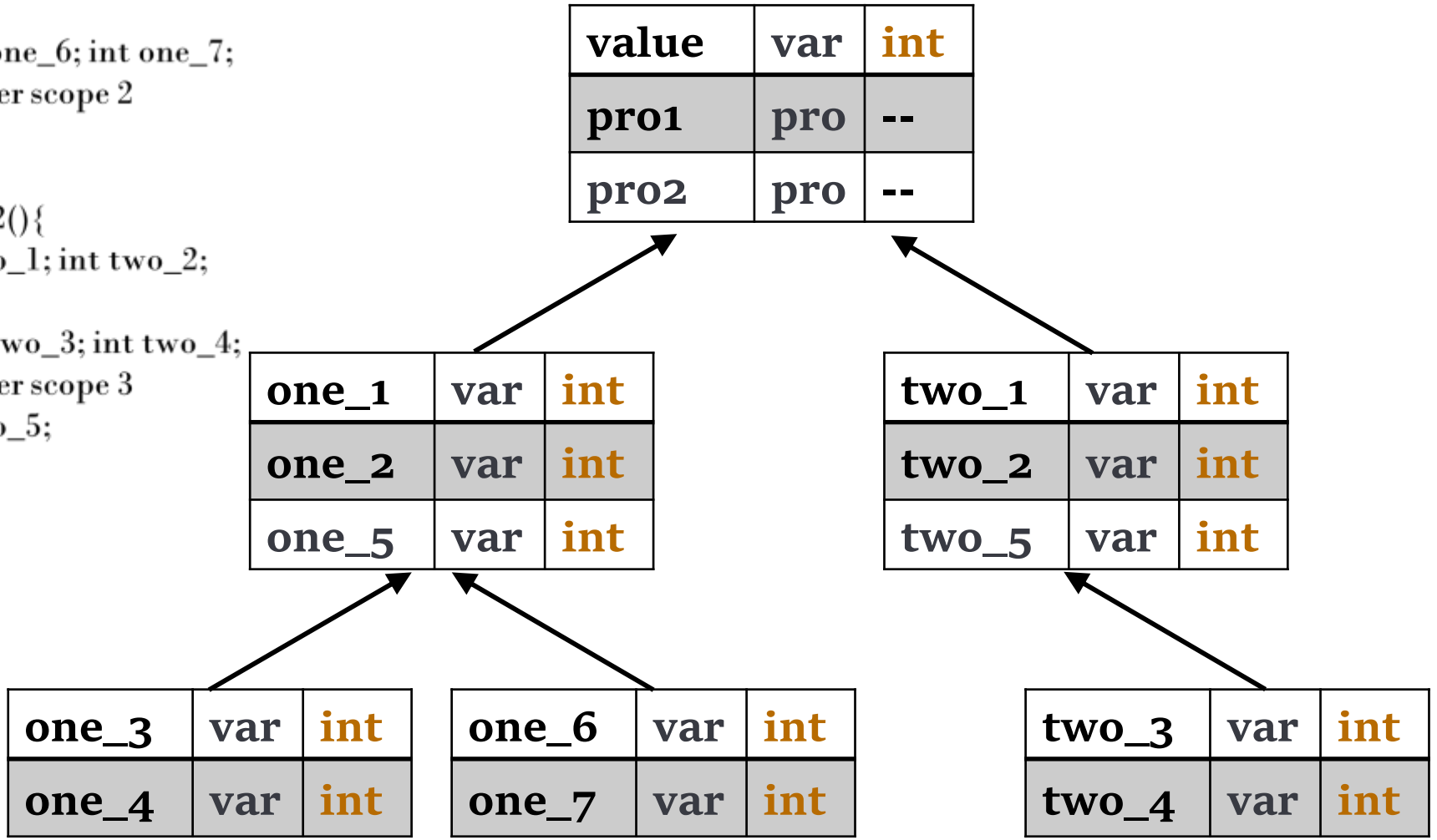
Nested Scope

- **Most-closely nested rule for blocks**
  - Given several different declarations for the same name, the declaration that applies to a reference is the one in the most closely nested block to the reference.
- Most-closely nested rule for blocks can be implemented by chaining symbol tables.
  - Chaining symbol tables is a static structure
  - 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.

```
int value = 10;
void pro1(){
   int one_1;  int one_2;
   {
       int one_3; int one_4;
    } // inner scope 1
   int one_5;
   {
      int one_6; int one_7;
   } //inner scope 2
}

void pro2(){
   int two_1; int two_2;
   {
      int two_3; int two_4;
   } //inner scope 3
   int two_5;
}
```

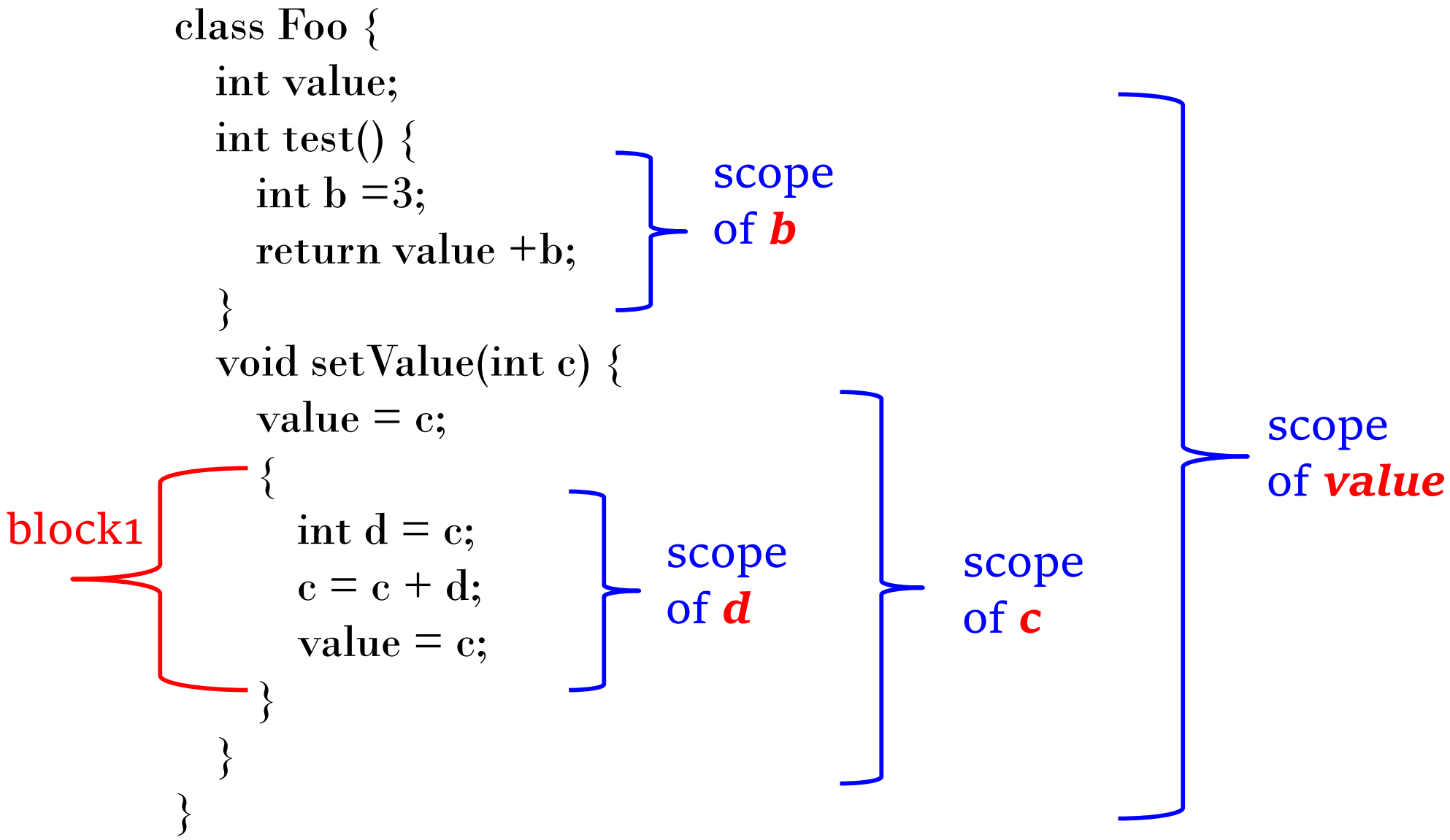
Nested Scope



Nested Scope

- Typical operations for chaining symbol tables.
  - create a symbol table with its parent symbol table as parameter.
    - Enter a new scope.
    - Add all variable declarations to the symbol table.
    - Process the body of the block/function/class.
    - Exit the scope.
  - lookup a symbol.

Checking Scope rules



```
class Foo {
  int value;
  int test() {
    int b =3;
    return value +b;
  }
  void setValue(int c) {
    value = c;
    {
      int d = c;
      c = c + d;
      value = c;
    }
  }
}
```

(Foo)

symbol	kind	type	attributes
value	field	int	...
test	method	->int	...
setValue	method	int->void	...

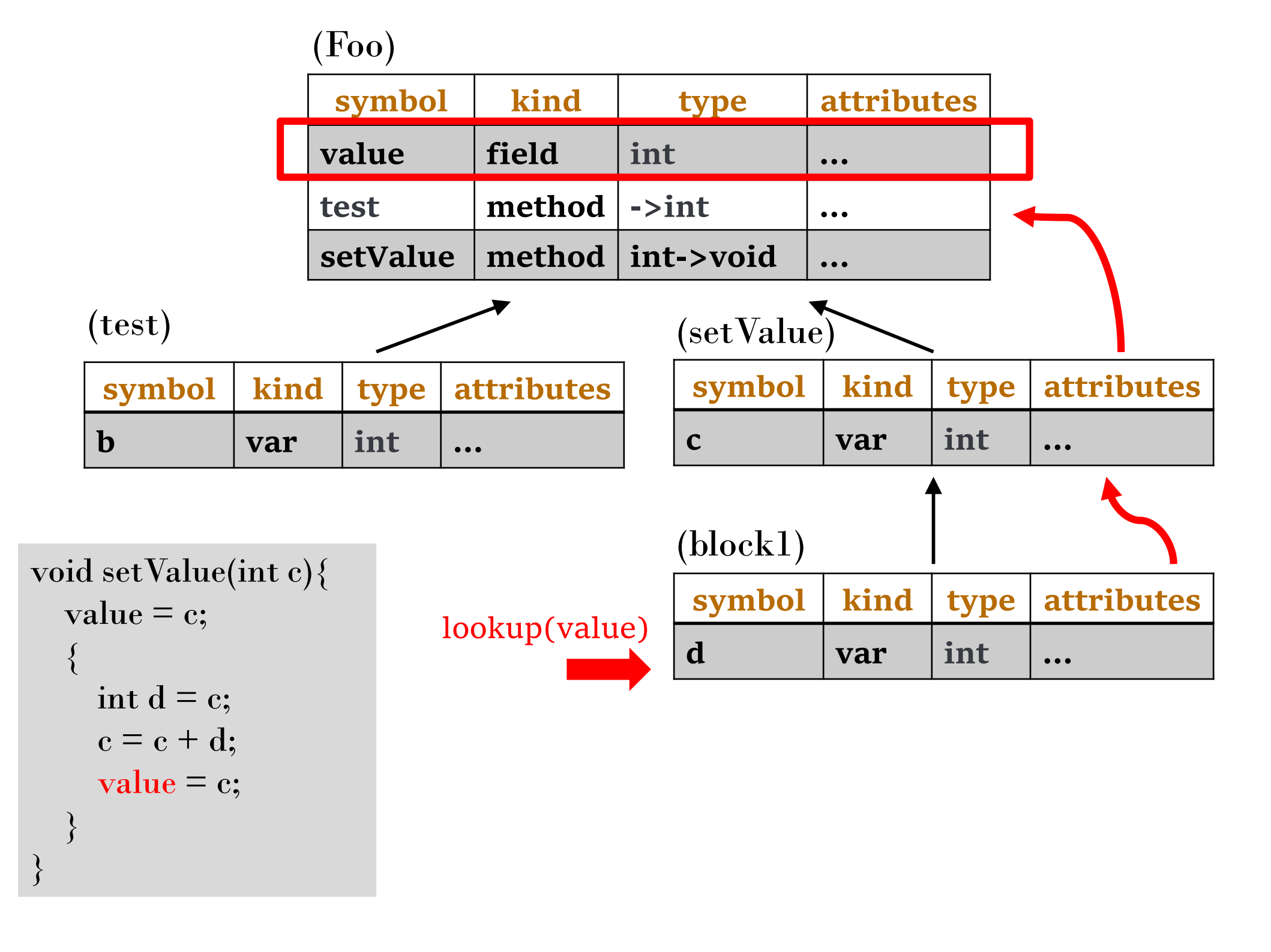
(test)

symbol	kind	type	attributes
b	var	int	...

(setValue)

symbol	kind	type	attributes
c	var	int	...

symbol	kind	type	attributes
d	var	int	...


(Foo)
symbol kind type attributes
value field int ...
test method ->int ...
setValue method int->void ...
(test)
symbol kind type attributes
b var int ...
(setValue)
symbol kind type attributes
c var int ...
(block1)
symbol kind type attributes
d var int ...
lookup(value)
void setValue(int c){
value = c;
{
int d = c;
c = c + d;
value = c;
}
}

(Foo)

symbol	kind	type	attributes
value	field	int	...
test	method	->int	...
setValue	method	int->void	...

Error!

(test)

symbol	kind	type	attributes
b	var	int	...

(setValue)

symbol	kind	type	attributes
c	var	int	...

(block1)

symbol	kind	type	attributes
d	var	int	...

lookup(myValue)

```
void setValue(int c){
    value = c;
    {
        int d = c;
        c = c + d;
        myValue = c;
    }
}
```