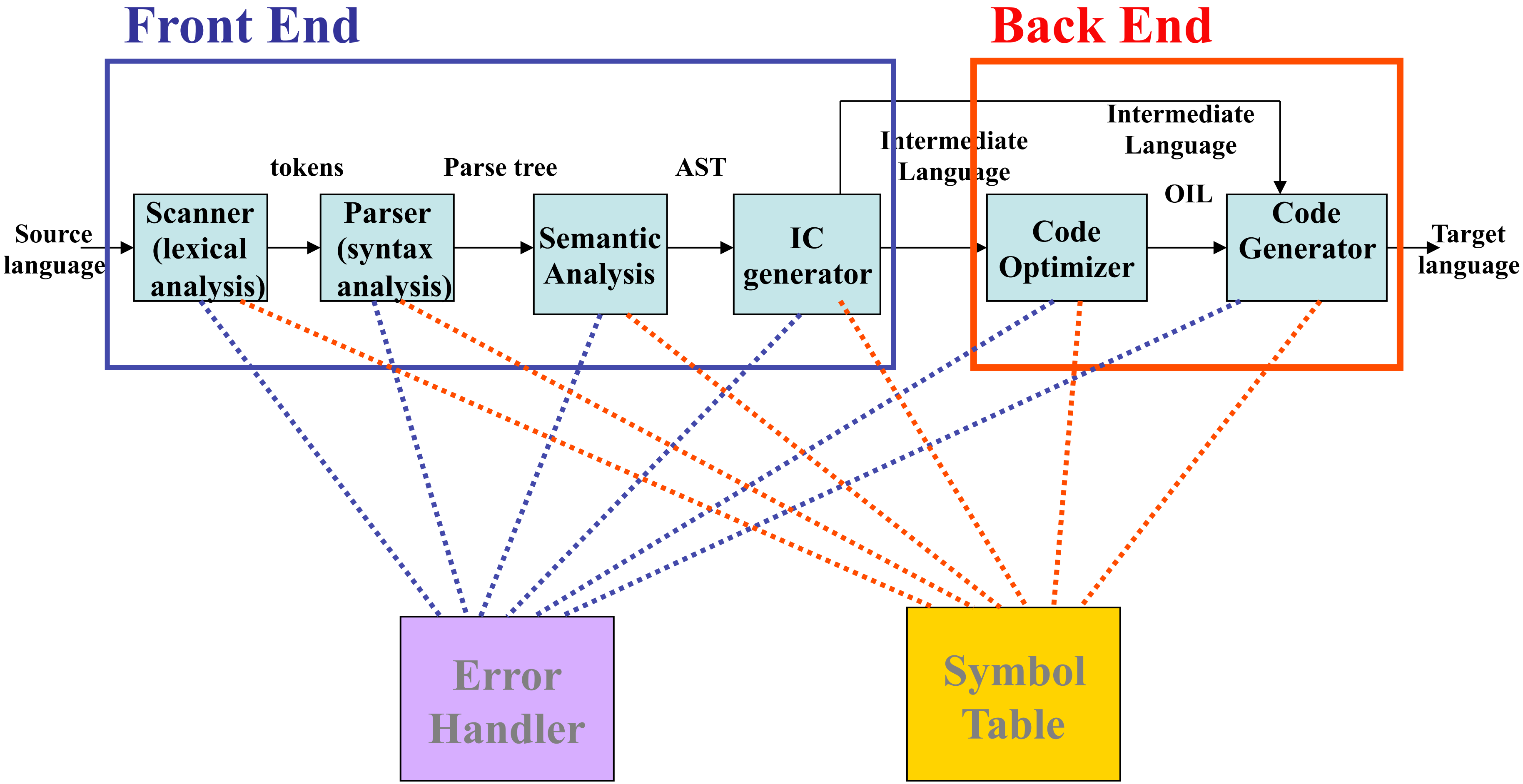
Language Processing Systems

Prof. Mohamed Hamada

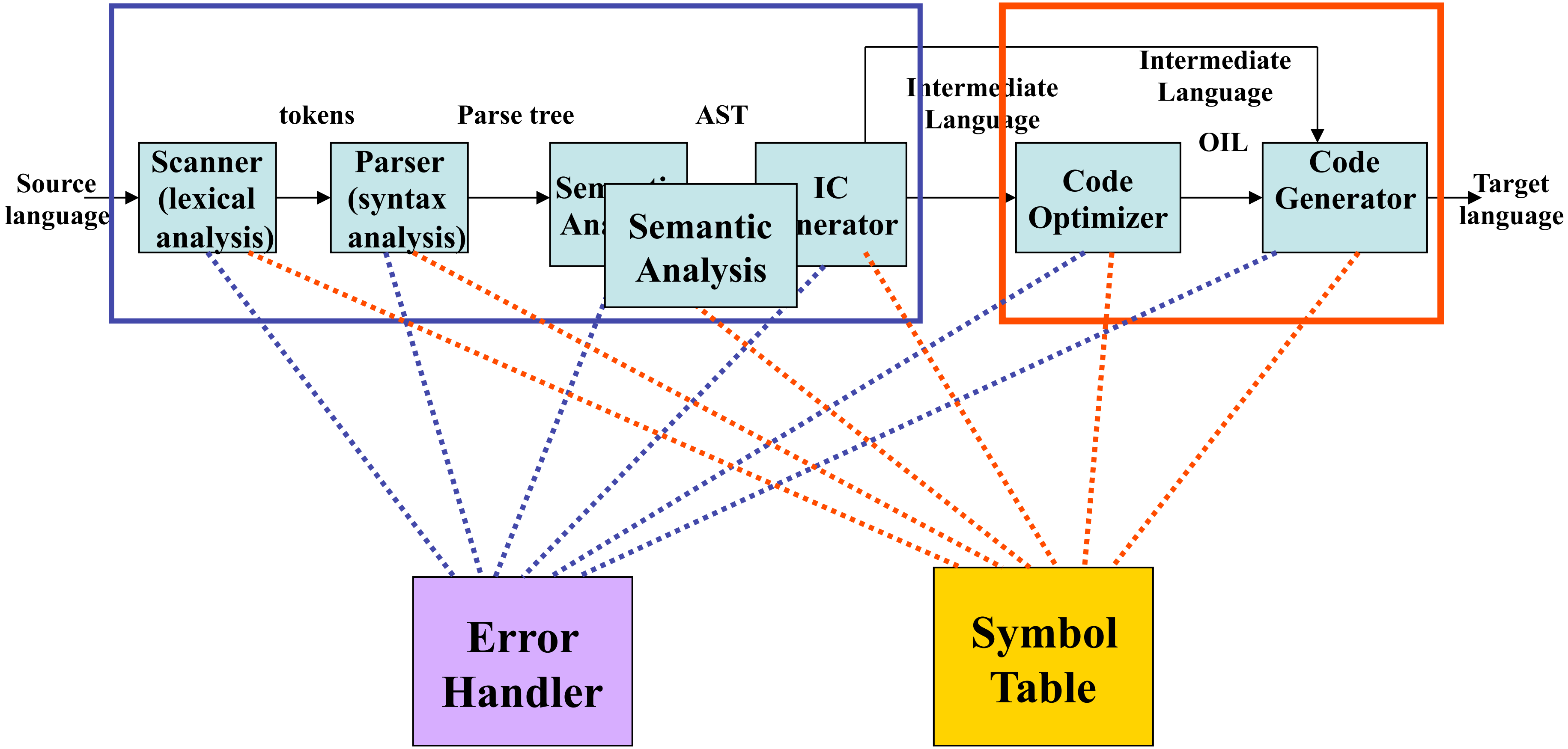
Software Engineering Lab.
The University of Aizu
Japan

Semantic Analysis

Compiler Architecture



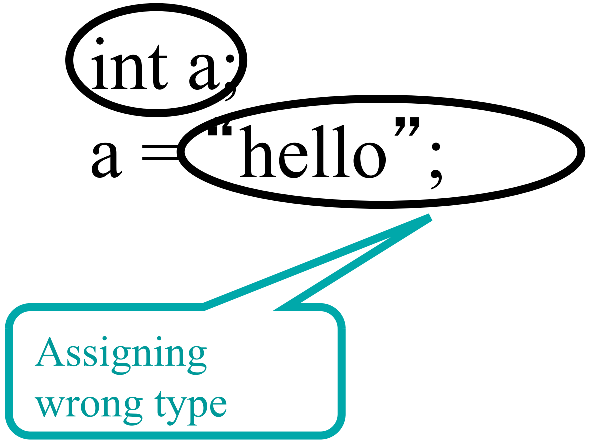
Semantic Analysis



- “Meaning”
- Type/Error Checking
- Intermediate Code Generation – abstract machine

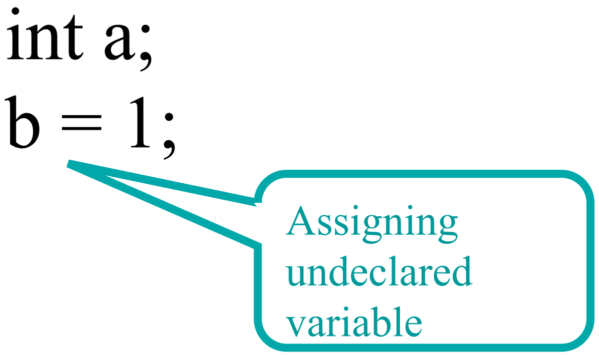
Semantic analysis motivation

- Syntactically correct programs may still contain errors
  - Lexical analysis does not distinguish between different variable names (same ID token)
  - Syntax analysis does not correlate variable declaration with variable use, does not keep track of types



```
int a;
a = "hello";
```

Assigning wrong type



```
int a;
b = 1;
```

Assigning undeclared variable

Goals of semantic analysis

- Check “correct” use of programming constructs
- Provide information for subsequent phases
- Context-sensitive – beyond context free grammars
  - Lexical analysis and syntax analysis provide relatively shallow checks of program structure
  - Semantic analysis goes deeper
- Correctness specified by semantic rules
  - Scope rules
  - Type-checking rules
  - Specific rules
- Note: semantic analysis ensures only partial correctness of programs
  - Runtime checks (pointer dereferencing, array access)

Example of semantic rules

- A variable must be declared before used
- A variable should not be declared multiple times
- A variable should be initialized before used
- Non-void method should contain return statement along all execution paths
- `break`/`continue` statements allowed only in loops
- `this` keyword cannot be used in static method
- `main` method should have specific signature
- …
- Type rules are important class of semantic rules
  - In an assignment statement, the variable and assigned expression must have the same type
  - In a condition test expression must have boolean type

Semantic Analysis

- Compilers examine code to find semantic problems.
  - Easy: undeclared variables, tag matching
  - Difficult: preventing execution errors
- Essential Issues:
    - Abstract Syntax Trees (AST)
    - Scope
    - Symbol tables
    - Type checking

Semantic Analysis

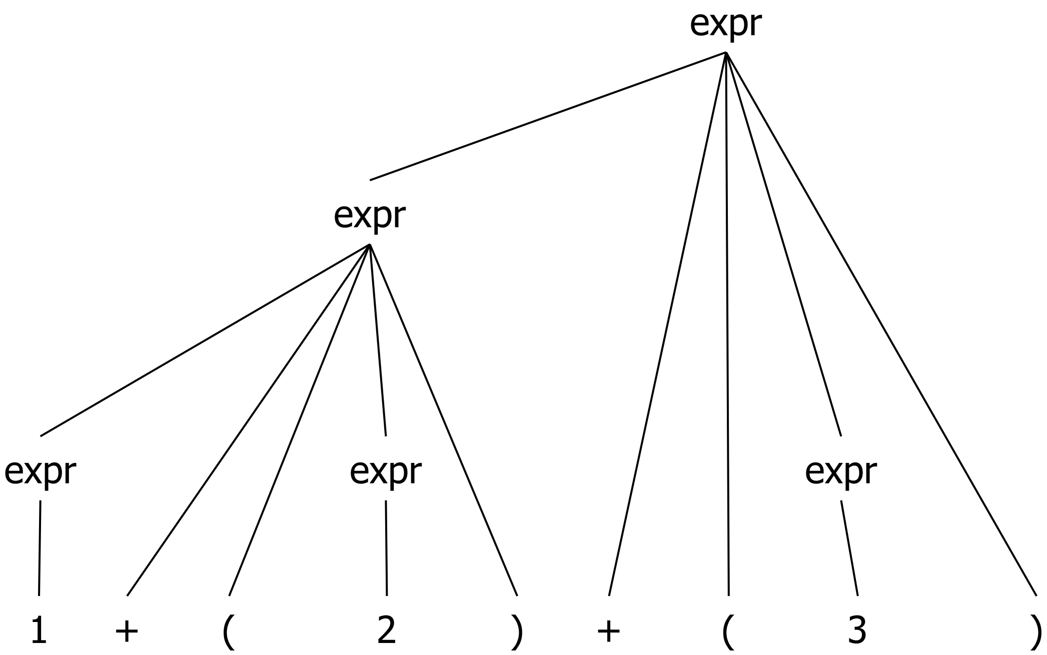
- Essential Issues:
  - Abstract Syntax Trees (AST)
  - Scope
  - Symbol tables
  - Type checking

AST

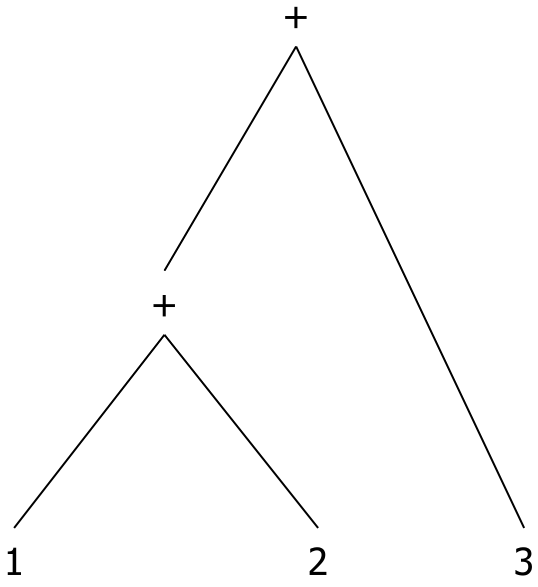
- Abstract Syntax Tree

Parse Tree vs. AST

Consider the expression: 1+(2)+(3)



Parse Tree



Abstract Syntax Tree

Why AST ?

- A more useful representation of the syntax tree
  - Actual level of details depends on your design
- Evaluate expression by AST traversal
- Basis for semantic analysis
- Later – annotate AST
  - Type information
  - Computed values

AST Construction

- AST Nodes constructed during parsing
- Bottom-up parser
  - Grammar rules annotated with actions for AST construction
  - When node is constructed all children available (already constructed)
- Top-down parser
  - More complicated

AST Construction

1 + (2) + (3)

expr + (2) + (3)

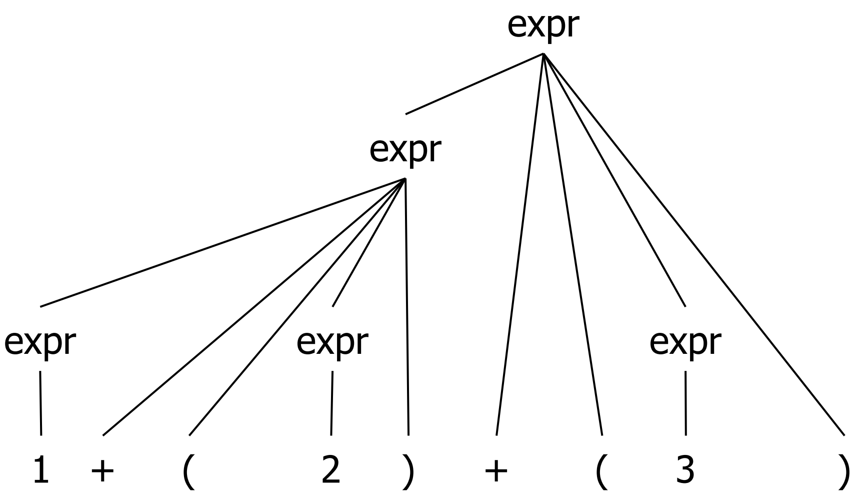
expr + (expr) + (3)

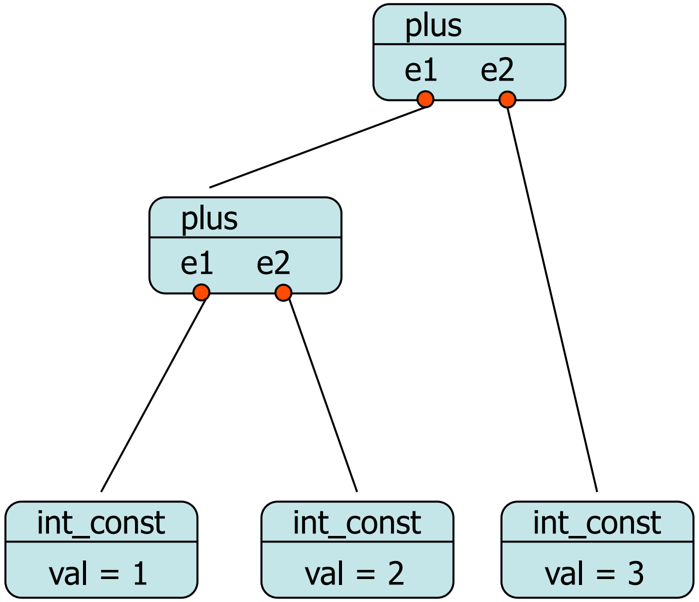
expr + (3)

expr + (expr)

expr

```
expr : expr '+' expr
          { $$ = plus($1,$3); }
     | '(' expr ')'
          { $$ = $2; }
     | INT_CONST
          { $$ = int_const($1); }
```

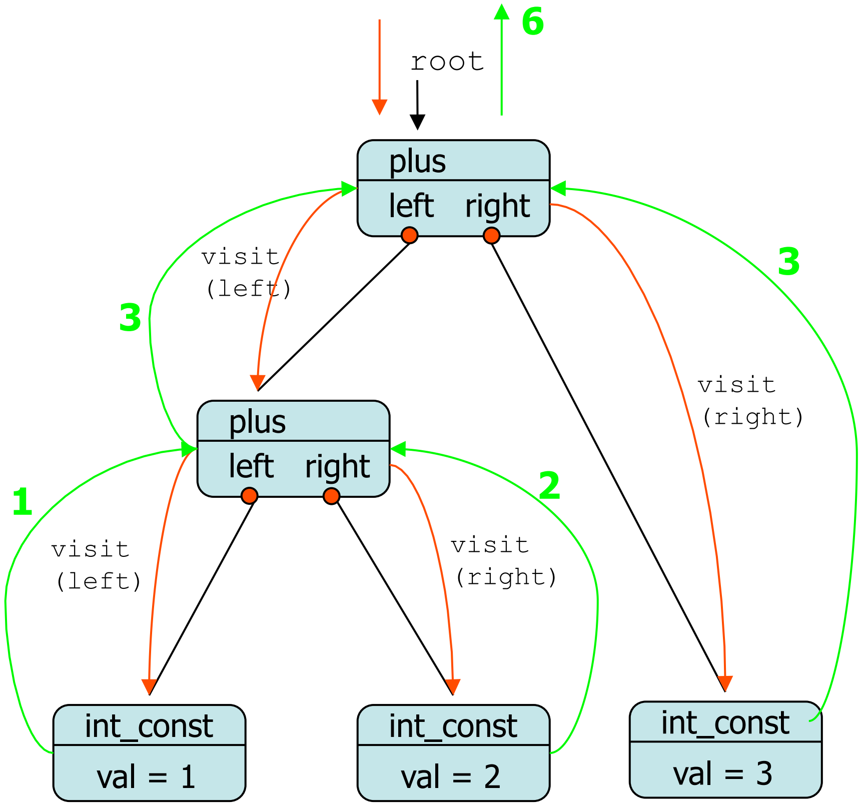




AST Traversal – Visitor Pattern

- Separate operations on objects of a data structure from object representation
- Each operation may be implemented as separate visitor
- Example
  - AST traversal for type-checking
  - …

AST Traversal



```
ExprEvalVisitor ev = new ExprEvalVisitor();
Integer result = (Integer)root.accept(ev);
```

```
class plus : public Expression {
public:
    Object accept(Visitor v) {
     return v.visit(this);
  }
  Expression left, right;
}
class int_const : public Expression {
public:
    Object accept(Visitor v) {
     return v.visit(this);
  }
  int val;
}
class ExprEvalVisitor : Visitor {
public:
   Object visit(plus e) {
    int leftVal=
     ((int)e.left.accept(this))).intValue();
    int rightVal=
     ((Integer)e.right.accept(this)).intValue();
    return leftVal + rightVal;
  }
  Object visit(int_const e) {
    return e.val;
  }
  ...
}
```

Semantic Analysis

- Essential Issues:
  - Abstract Syntax Trees (AST)
  - Scope
  - Symbol tables
  - Type checking

Scope

Scope and visibility

- Scope (visibility) of identifier = portion of program where identifier can be referred to
- Lexical scope = textual region in the program
  - Statement block
  - Method body
  - Class body
  - Module / package / file
  - Whole program (multiple modules)

Scope

Scope rules：

- identifiers are defined
  - no multiple definition of same identifier
  - local variables are defined before used
  - program conforms to scope rules

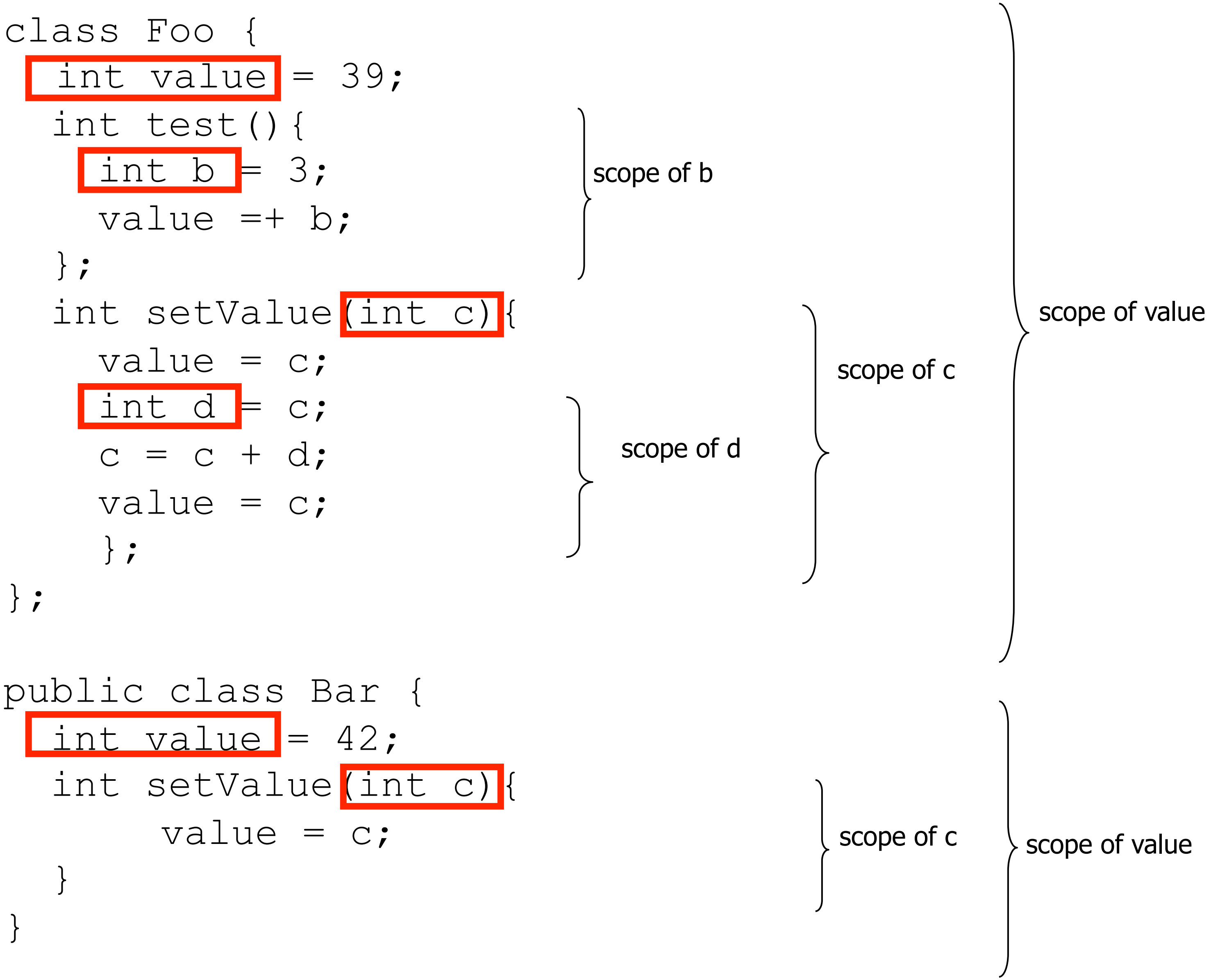
Scope

Each **scope** maps a set of variables to a set of meanings.

The **scope of a variable declaration** is the part of the program where that variable is visible.

Scopes Example

Consider the following example:



```
class Foo {
  int value = 39;
  int test(){
    int b = 3;
    value =+ b;
  };
  int setValue(int c){
    value = c;
    int d = c;
    c = c + d;
    value = c;
    };
};

public class Bar {
  int value = 42;
  int setValue(int c){
       value = c;
  }
}
```

scope of b

scope of d

scope of c

scope of value

scope of c

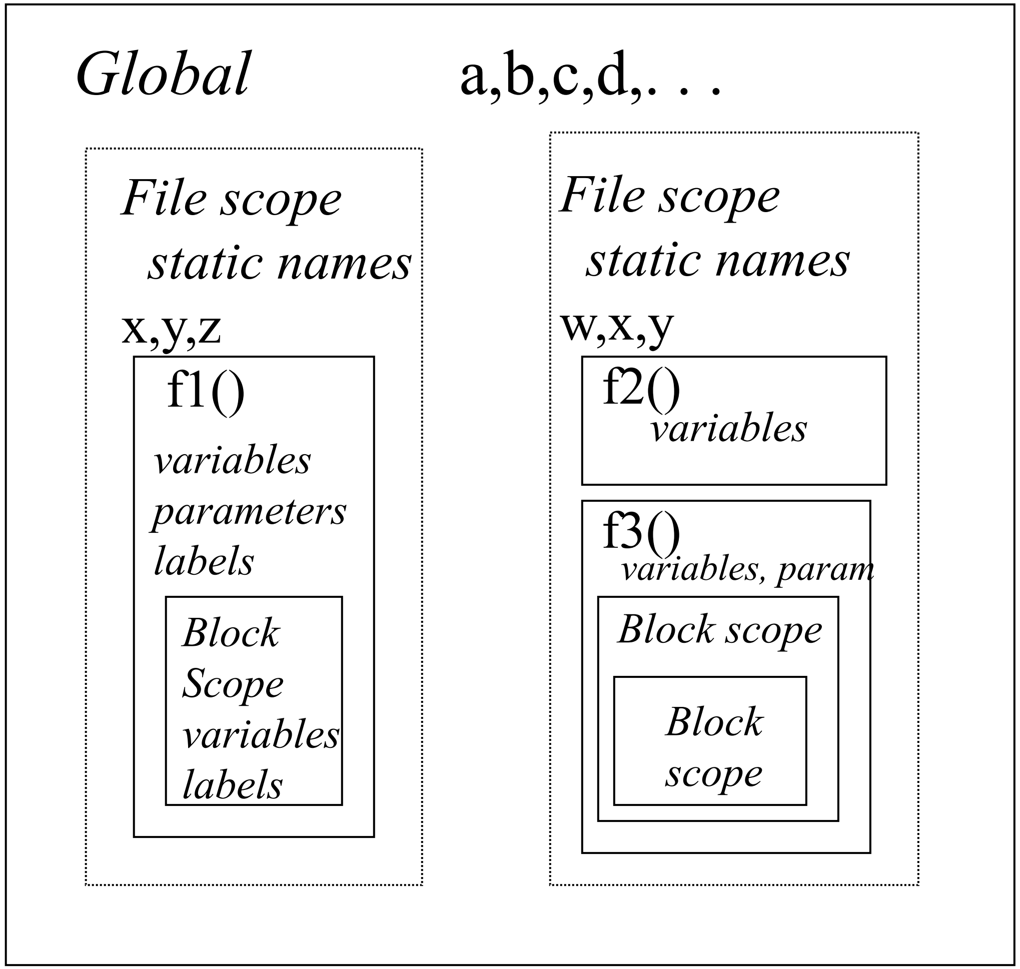
scope of value

Scope

In most languages, a complete program will contain several different **scopes**.

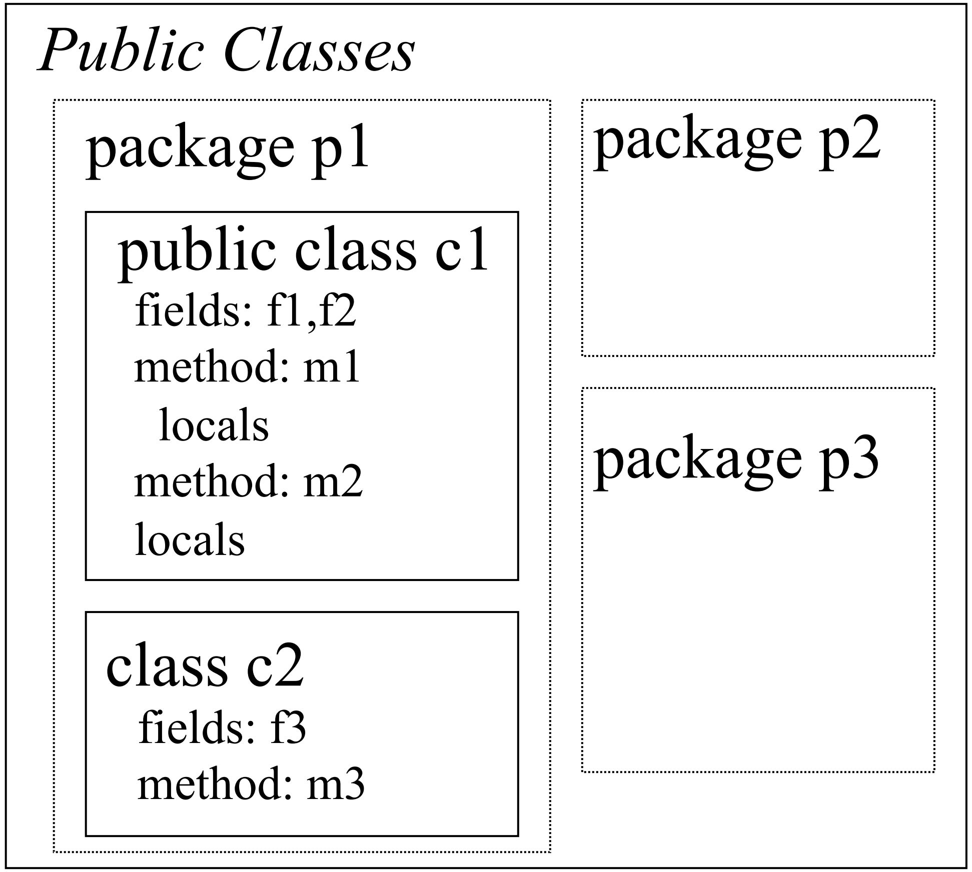
Different languages have different rules for **scope** definition

Example 3: C Scopes



- **Global scope holds variables and functions**
- **No function nesting**
- **Block level scope introduces variables and labels**
- **File level scope with static variables that are not visible outside the file (global otherwise)**

Example 4: Java Scopes



- **Limited global name space with only public classes**
- **Fields and methods in a public class can be public ➔ visible to classes in other packages**
- **Fields and methods in a class are visible to all classes in the same package unless declared private**
- **Class variables visible to all objects of the same class.**

Scopes: Referencing Environment

The **referencing environment** at a particular location in source code is the set of variables that are visible at that point.

- A variable is **local** to a procedure if the declaration occurs in that procedure.
- A variable is **non-local** to a procedure if it is visible inside the procedure but is not declared inside that procedure.
- A variable is **global** if it occurs in the outermost scope (special case of non-local).

Types of Scoping

- Static – scope of a variable determined from the source code.
  - "Most Closely Nested"
  - Used by most languages
- Dynamic – current call tree determines the relevant declaration of a variable use.

Static Scope: Most Closely Nested Rule

The scope of a particular declaration is given by the most closely nested rule

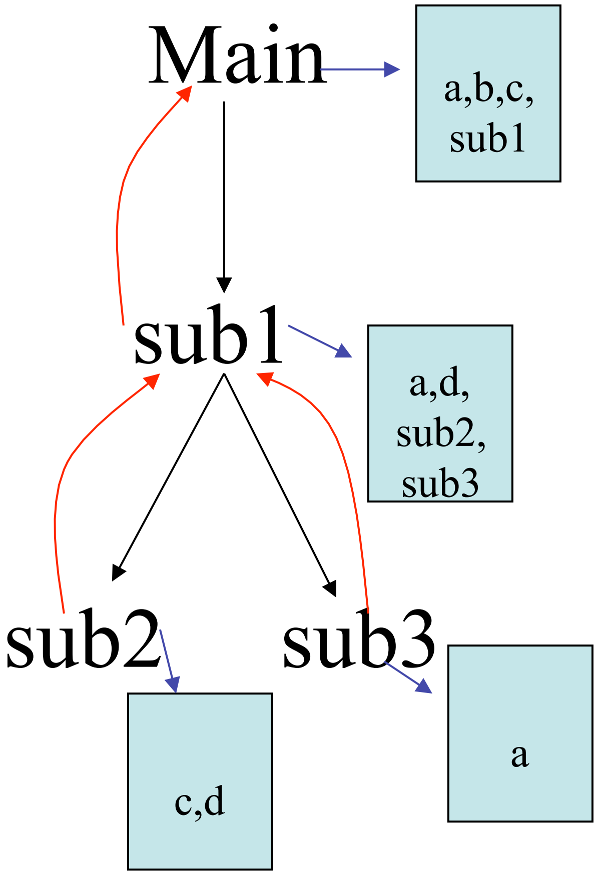
- The scope of a variable declared in block B, includes B.
- If x is not declared in block B, then an occurrence of x in B is in the scope of a declaration of x in some enclosing block A, such that A has a declaration of x and A is more closely nested around B than any other block with a declaration of x.

Example Program: Static Scope

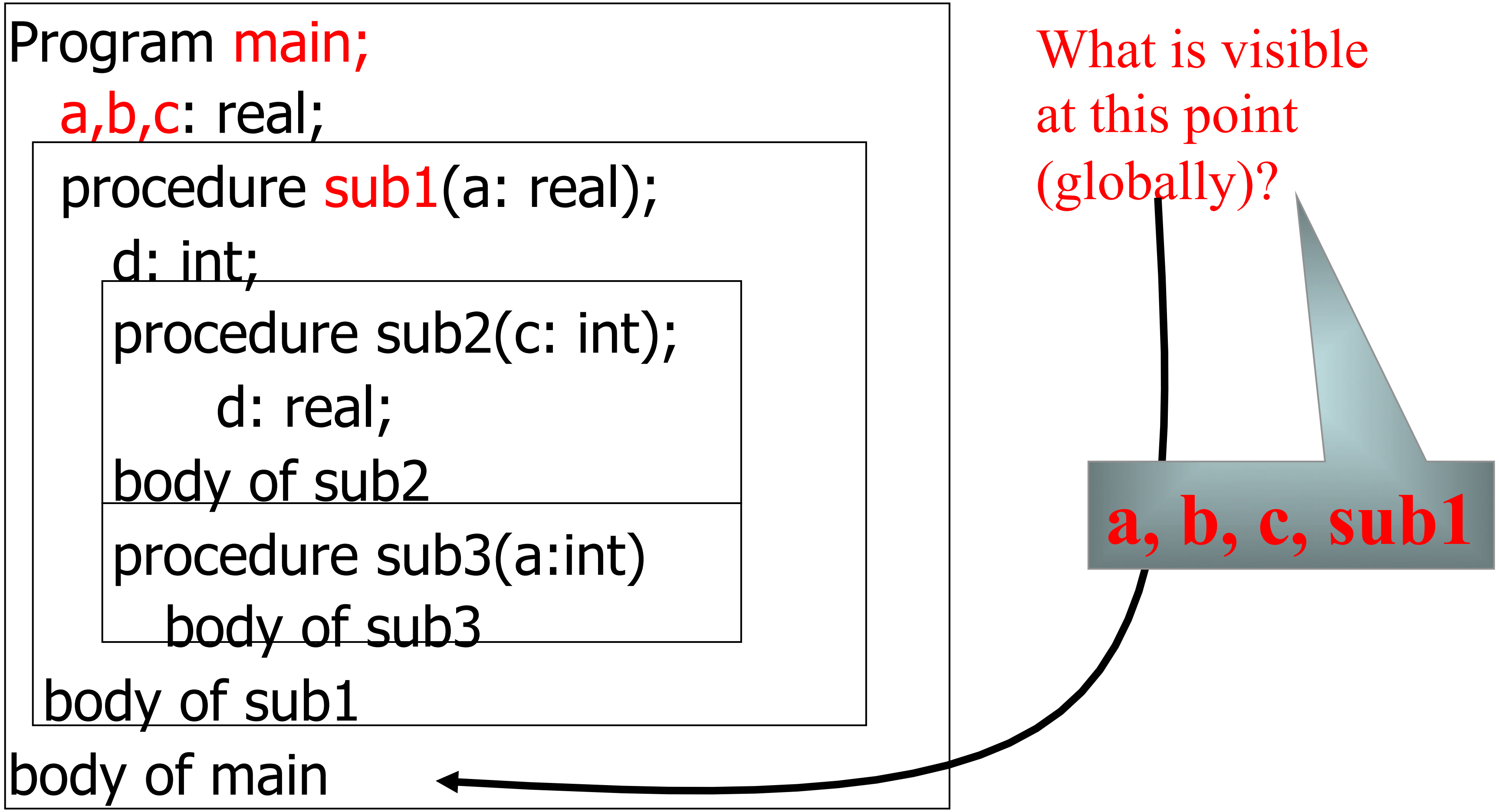
```
Program main;
  a,b,c: real;
  procedure sub1(a: real);
    d: int;
    procedure sub2(c: int);
         d: real;
    body of sub2
    procedure sub3(a:int)
      body of sub3

  body of sub1

body of main
```



Example Program: Static Scope

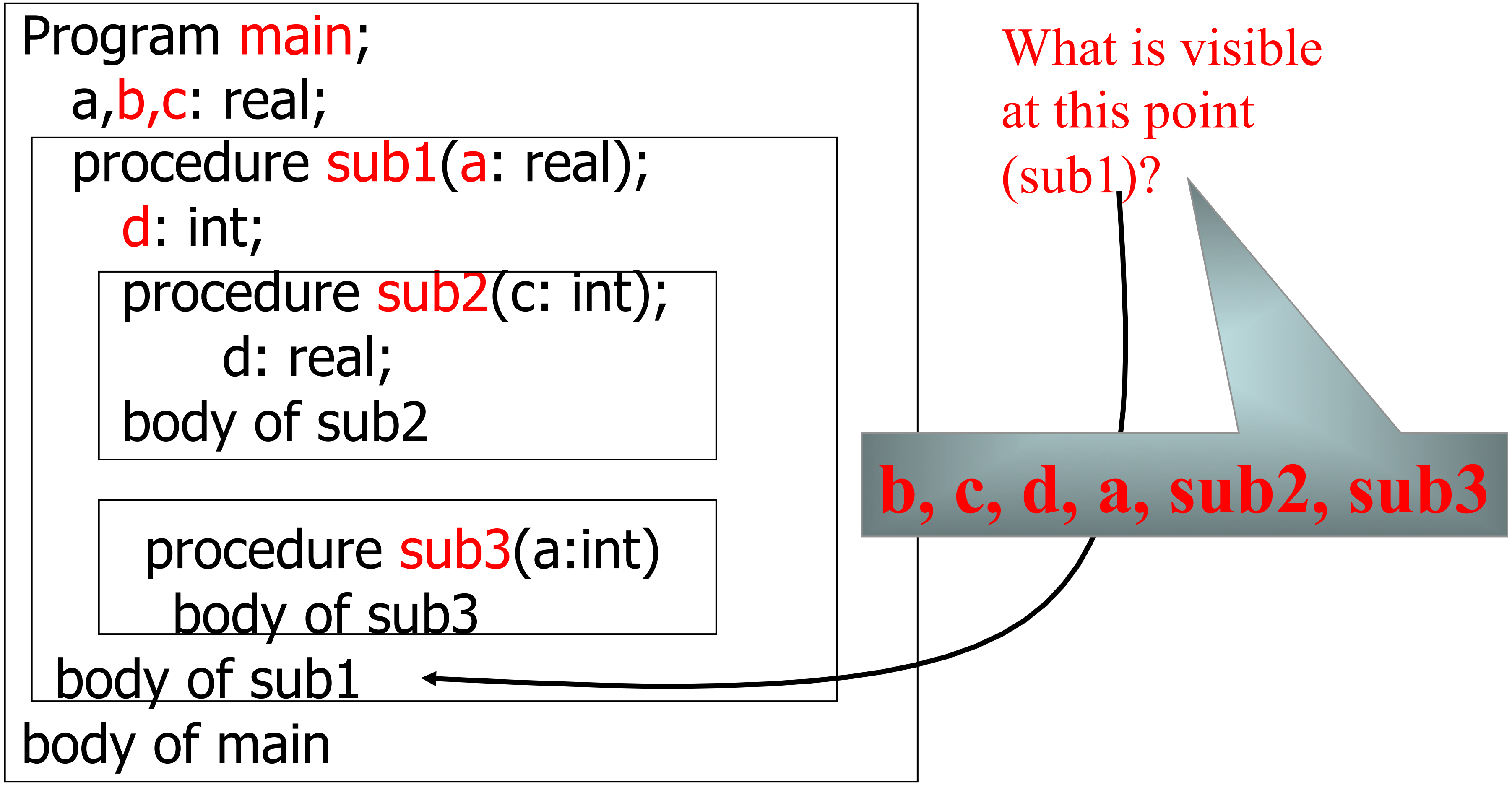


```
Program main;
  a,b,c: real;
  procedure sub1(a: real);
     d: int;
     procedure sub2(c: int);
           d: real;
     body of sub2
     procedure sub3(a:int)
        body of sub3
 body of sub1
body of main
```

What is visible at this point (globally)?

a, b, c, sub1

Example Program: Static Scope



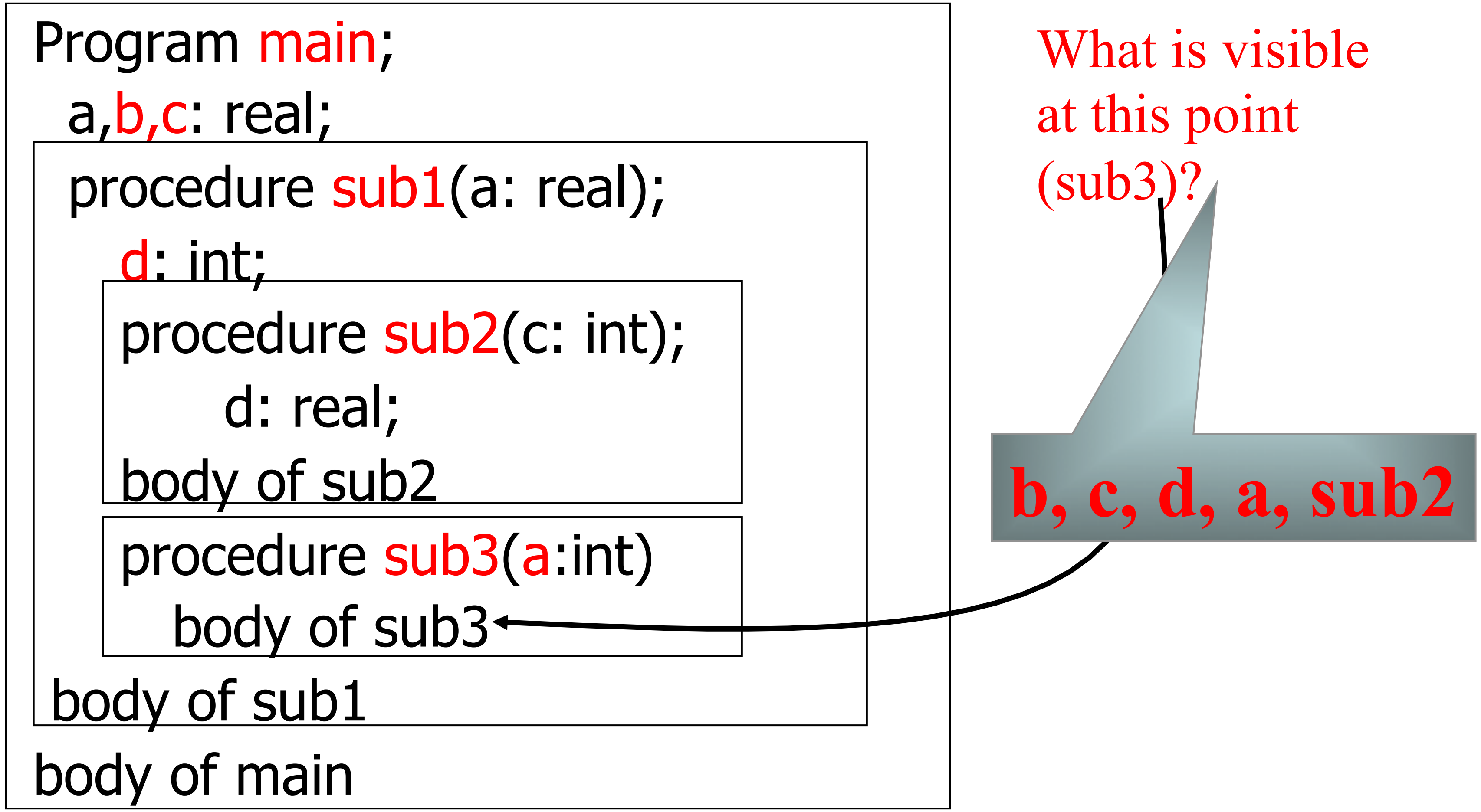
```
Program main;
   a,b,c: real;
   procedure sub1(a: real);
      d: int;
      procedure sub2(c: int);
            d: real;
      body of sub2

       procedure sub3(a:int)
         body of sub3
  body of sub1
body of main
```

What is visible at this point (sub1)?

b, c, d, a, sub2, sub3

Example Program: Static Scope

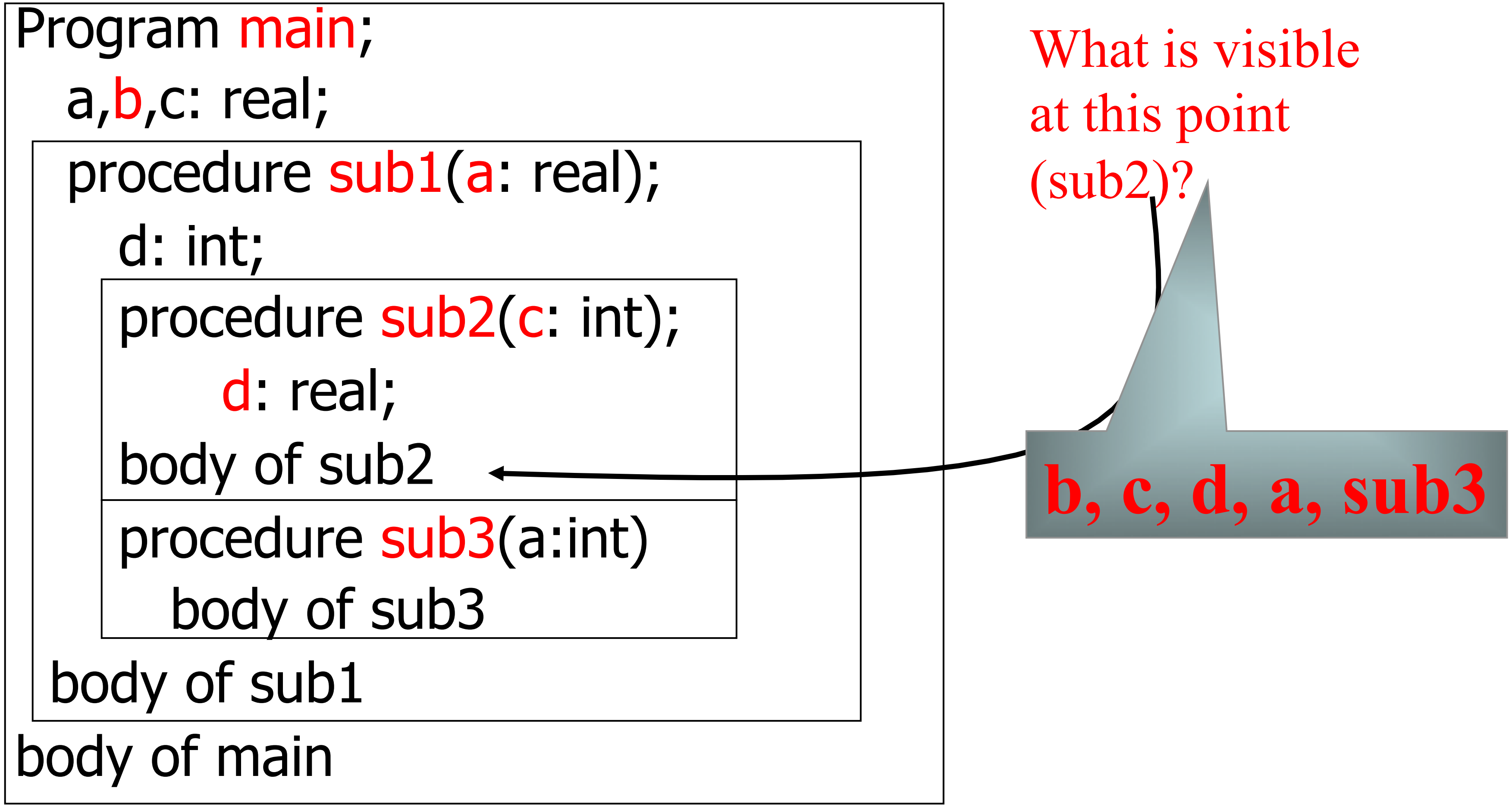


```
Program main;
  a,b,c: real;
  procedure sub1(a: real);
    d: int;
    procedure sub2(c: int);
          d: real;
    body of sub2
    procedure sub3(a:int)
       body of sub3
  body of sub1
body of main
```

What is visible at this point (sub3)?

b, c, d, a, sub2

Example Program: Static Scope



```
Program main;
   a,b,c: real;
   procedure sub1(a: real);
      d: int;
      procedure sub2(c: int);
            d: real;
      body of sub2
      procedure sub3(a:int)
         body of sub3
  body of sub1
body of main
```

What is visible at this point (sub2)?

b, c, d, a, sub3

Dynamic Scope

- Based on calling sequences of program units, not their textual layout (temporal versus spatial)
- References to variables are connected to declarations by searching the chain of subprogram calls (runtime stack) that forced execution to this point

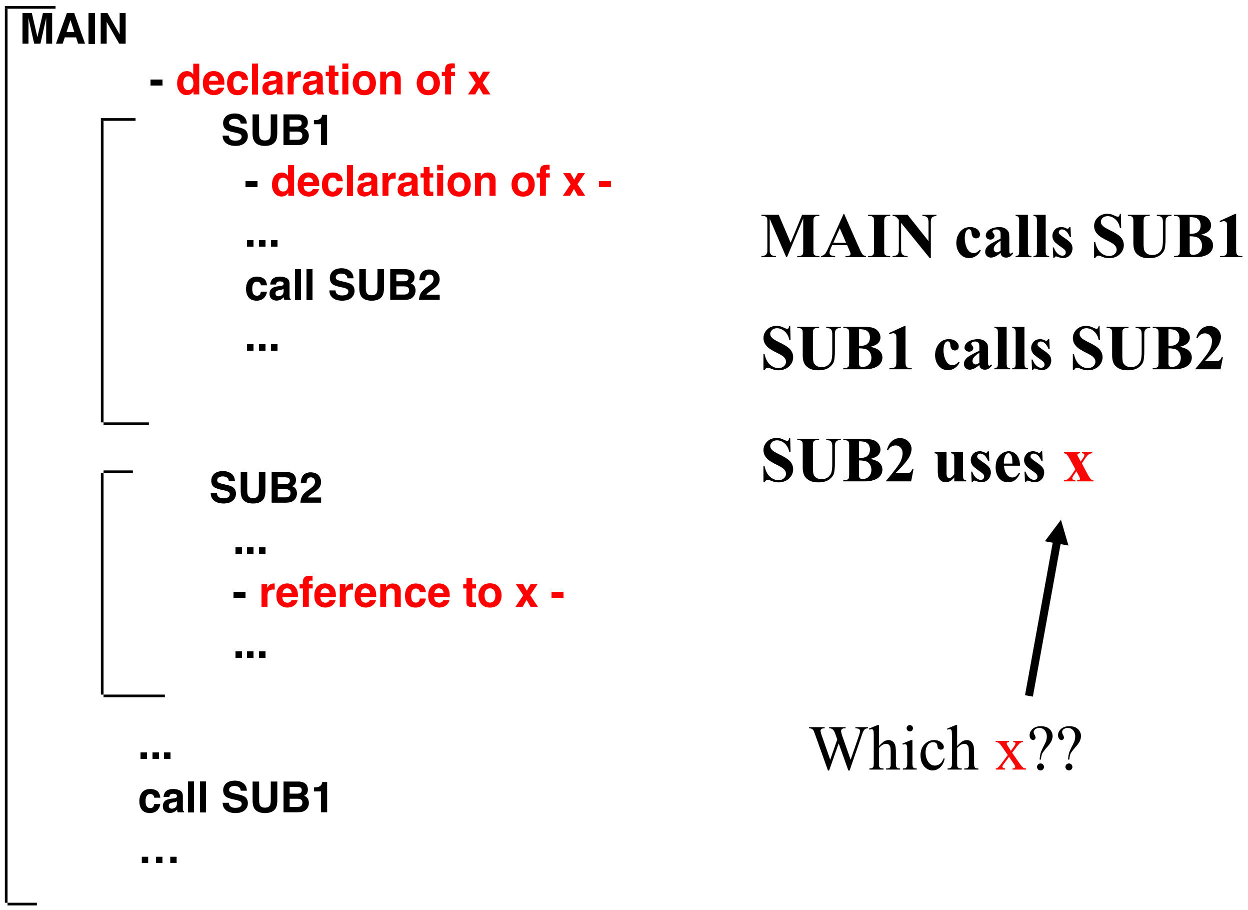
Dynamic Scope

- In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms.
- A subprogram is **active** if its execution has begun but has not yet terminated.

Dynamic Scope

- Evaluation of Dynamic Scoping:
  - **Advantage**: convenience (easy to implement)
  - **Disadvantage**: poor readability, unbounded search time

Scope Example



```
MAIN
          - declaration of x
     SUB1
       - declaration of x -
       ...
       call SUB2
       ...

     SUB2
       ...
       - reference to x -
       ...

     ...
     call SUB1
     ...
```

MAIN calls SUB1

SUB1 calls SUB2

SUB2 uses x

Which x??

Scope Example

```
MAIN
      - declaration of x
        SUB1
          - declaration of x -
          ...
          call SUB2
          ...

        SUB2
          ...
          - reference to x -
          ...

      ...
      call SUB1
      ...
```

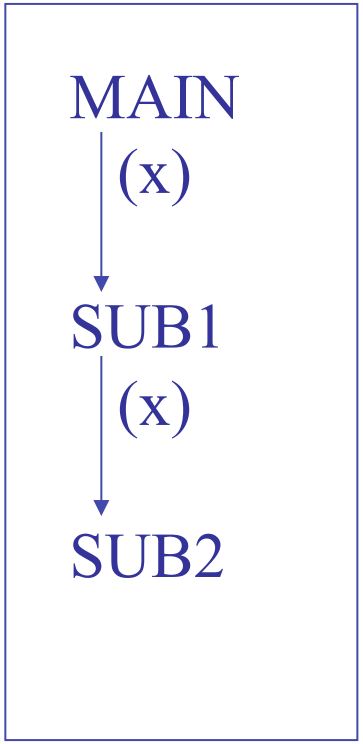
MAIN calls SUB1

SUB1 calls SUB2

SUB2 uses x

For static scoping,
it is main’s **x**

Scope Example


MAIN

(x)

SUB1

(x)

SUB2

```
MAIN
          - declaration of x
            SUB1
              - declaration of x -
              ...
              call SUB2
              ...

            SUB2
             ...
             - reference to x -
             ...

          ...
          call SUB1
          ...
```

MAIN calls SUB1

SUB1 calls SUB2

SUB2 uses x

For dynamic scoping,
it is sub1's **x**