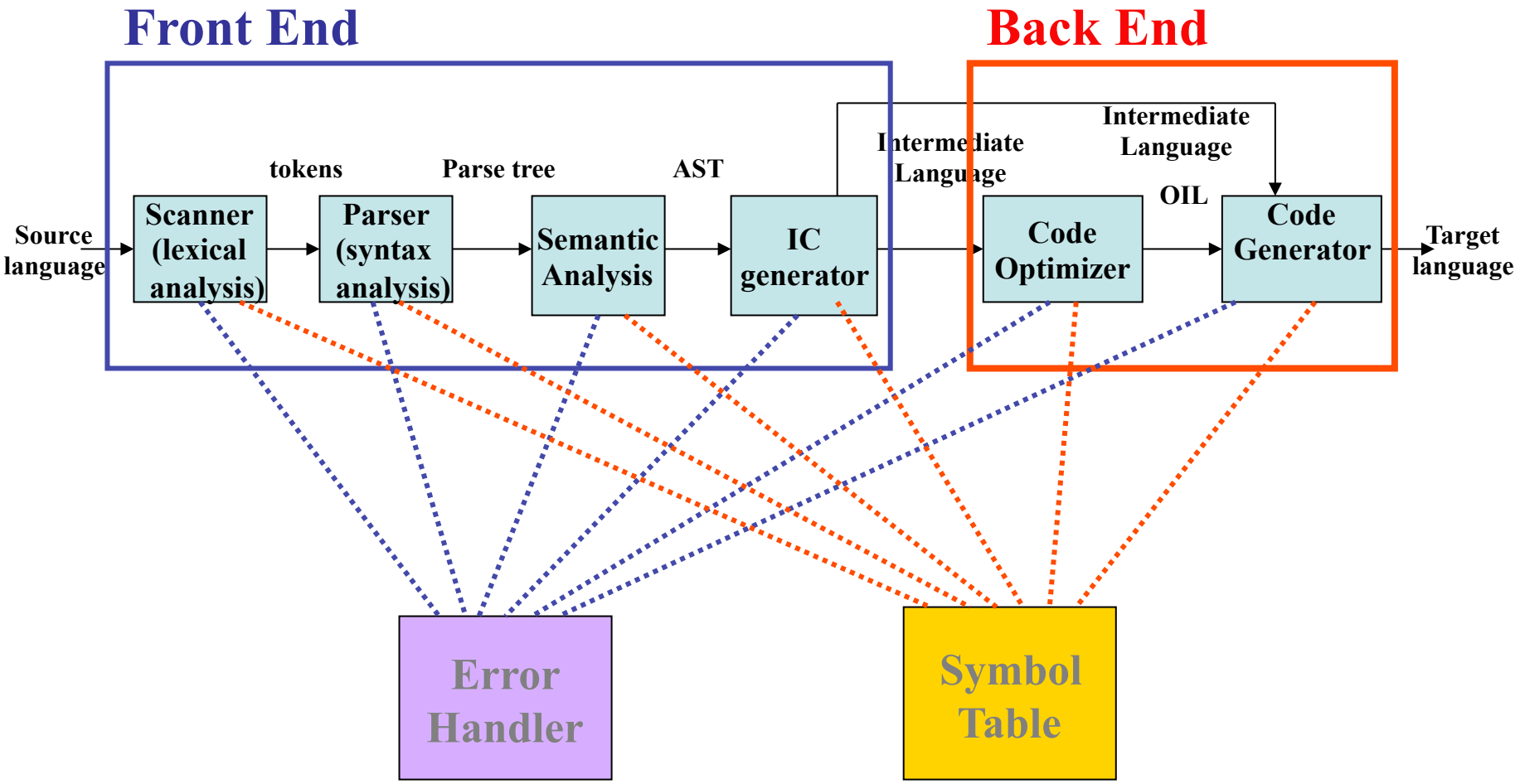
Language Processing Systems

Prof. Mohamed Hamada

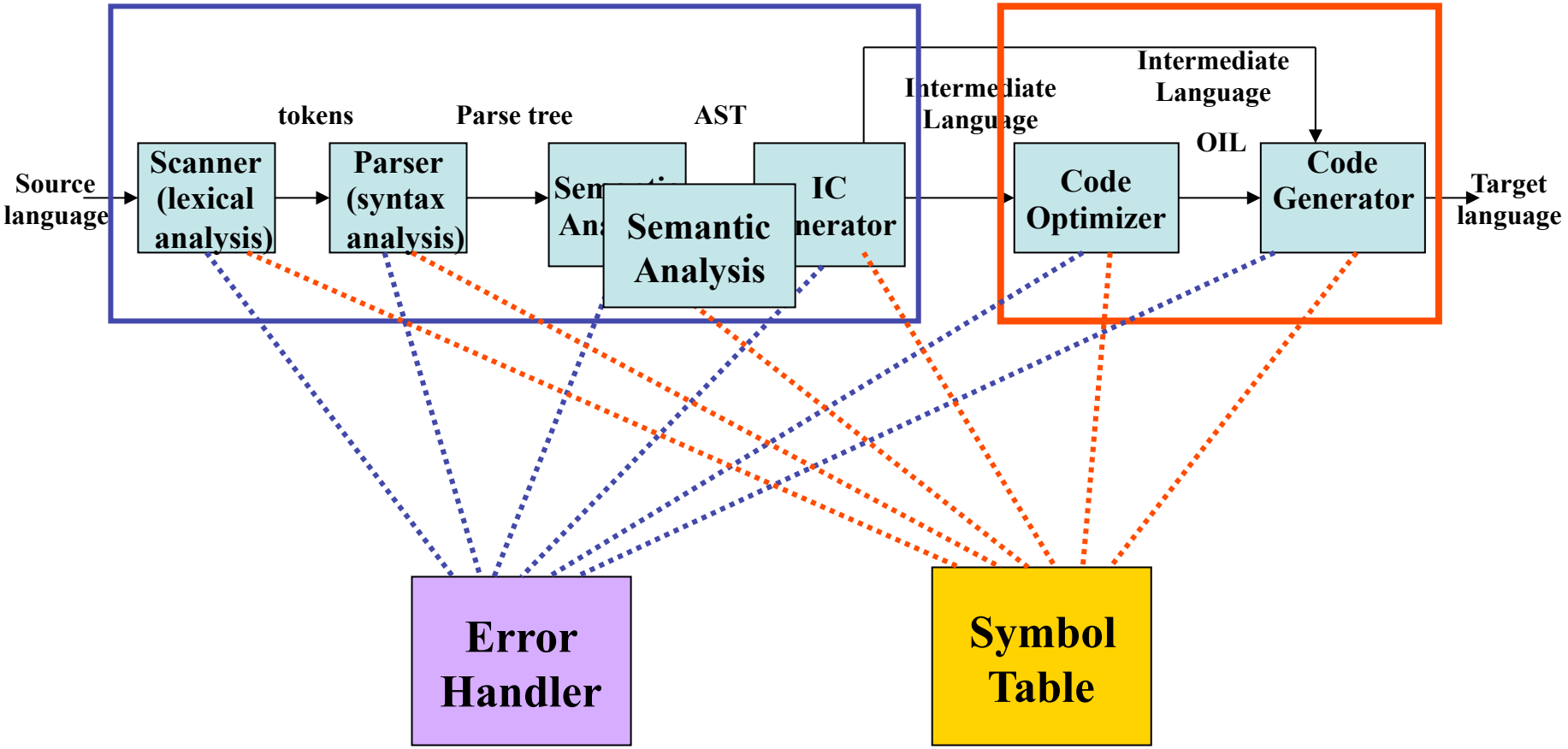
Software Engineering Lab.
The University of Aizu
Japan

Semantic Analysis

Compiler Architecture



Semantic Analysis



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

Semantic Analysis

- 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

Semantic Analysis

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

Symbol Table

Symbol table

- A compile-time data structure used to map names into declarations
- An environment that stores information about identifiers
- A data structure that captures scope information
- Each entry in symbol table contains
  - The name of an identifier
  - Its kind (variable/method/field…)
  - Type
  - Additional properties, e.g, **final**, **public**
- One symbol table for each scope

Symbol Table

- Primary data structure inside a compiler
- Stores information about the symbols in the input program including:
  - Type (or class)
  - Size (if not implied by type)
  - Scope
- Scope represented explicitly or implicitly (based on table structure)
- Classes can also be represented by structure – one difference = information about classes must persist after have left scope
- Used in all phases of the compiler

Symbol table structure

- Assign variables to storage classes that prescribe scope, visibility, and lifetime
  - **scope rules prescribe the symbol table structure**
  - **scope: unit of static program structure with one or more variable declarations**
  - **scope may be nested**
    - **Pascal: procedures are scoping units**
    - **C: blocks, functions, files are scoping units**
- Visibility, lifetimes, global variables
- Automatic or stack storage
- Static variables

Symbol Table Object

Symbol table functions are called during parsing:

- Insert(x) –*A new symbol is defined*
- Delete(x) –*The lifetime of a symbol ends*
- Lookup(x) –*A symbol is used*
- EnterScope(s) – *A new scope is entered*
- ExitScope(s) – *A scope is left*

Symbol Table Issues

- A major consideration in designing a symbol table is that insertion and retrieval should be as fast as possible
- One dimensional table: search is very slow
- Balanced binary tree: quick insertion, searching and retrieval; extra work required to keep the tree balanced
- Hash tables: quick insertion, searching and retrieval; extra work to compute hash keys
- Hashing with a chain of entries is generally a good approach

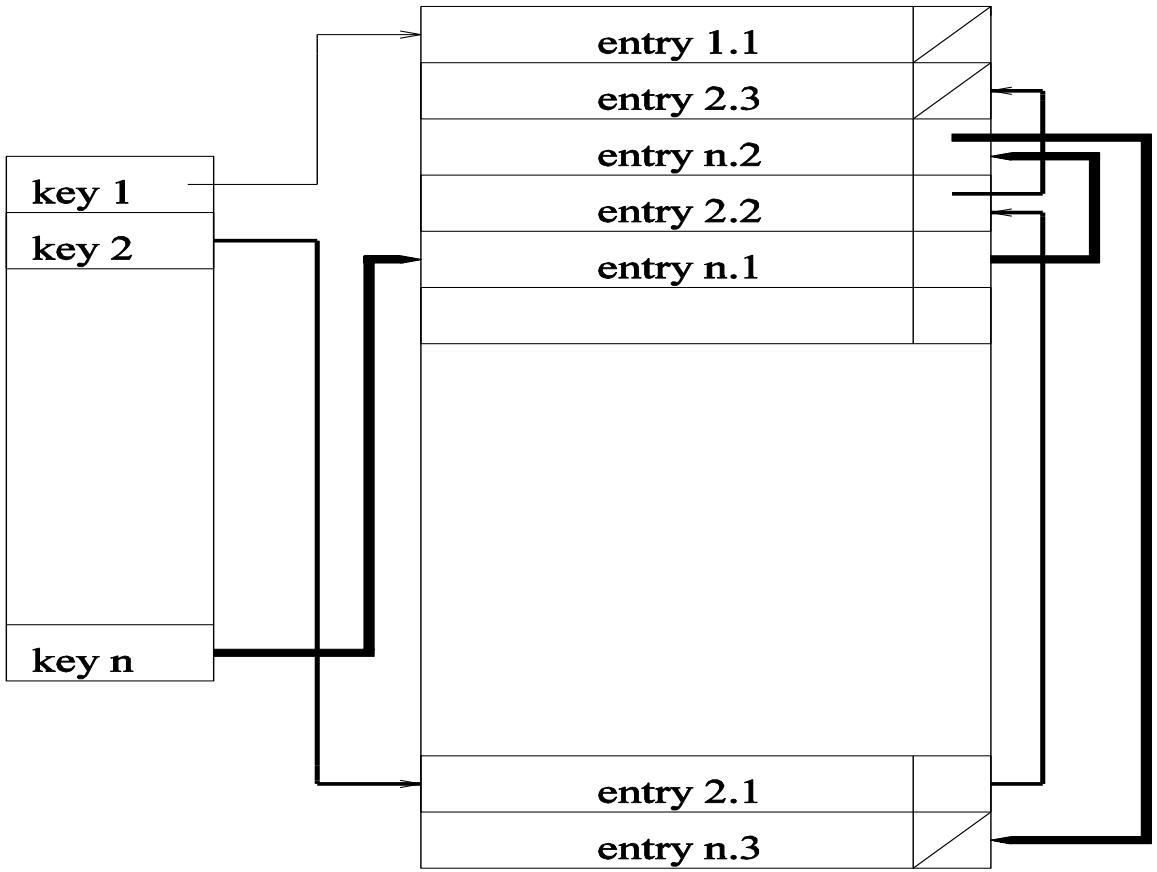
Symbol Table Implementation

- Variety of choices, including arrays, lists, trees, heaps, hash tables, …
- Different structures may be used for local tables versus tables representing scope.
- Each table in the hierarchy could be implemented using **`java.util.HashMap`**

Symbol Table Implementation

- Scopes implemented using symbol tables
- Data-structure for “look-up”
  - key – identifier
  - value – type of identifier, other semantic properties

Hashed local symbol table



Symbol Table - Example 1

```
class Test {
  int a = 39;
  int test(){
    int b = 3 ;
    a = a + b ;
  };
};
```

Symbol	Kind	Type	Properties
a	var	Int	…
b	var	Int	…
test	method	-> Int	…

Symbol Table - Example 1

```
class Test {
  int a = 39;
  int test(){
    String a = "hello";
    int b = 3;
    b = +b;
  };
};
```

Symbol	Kind	Type	Properties
a	var	Int	...
b	var	Int	...
test	method	-> Int	...
a	var	String	...

Implementing Scopes

```
Int x = 0 ;
e = e + x;
```

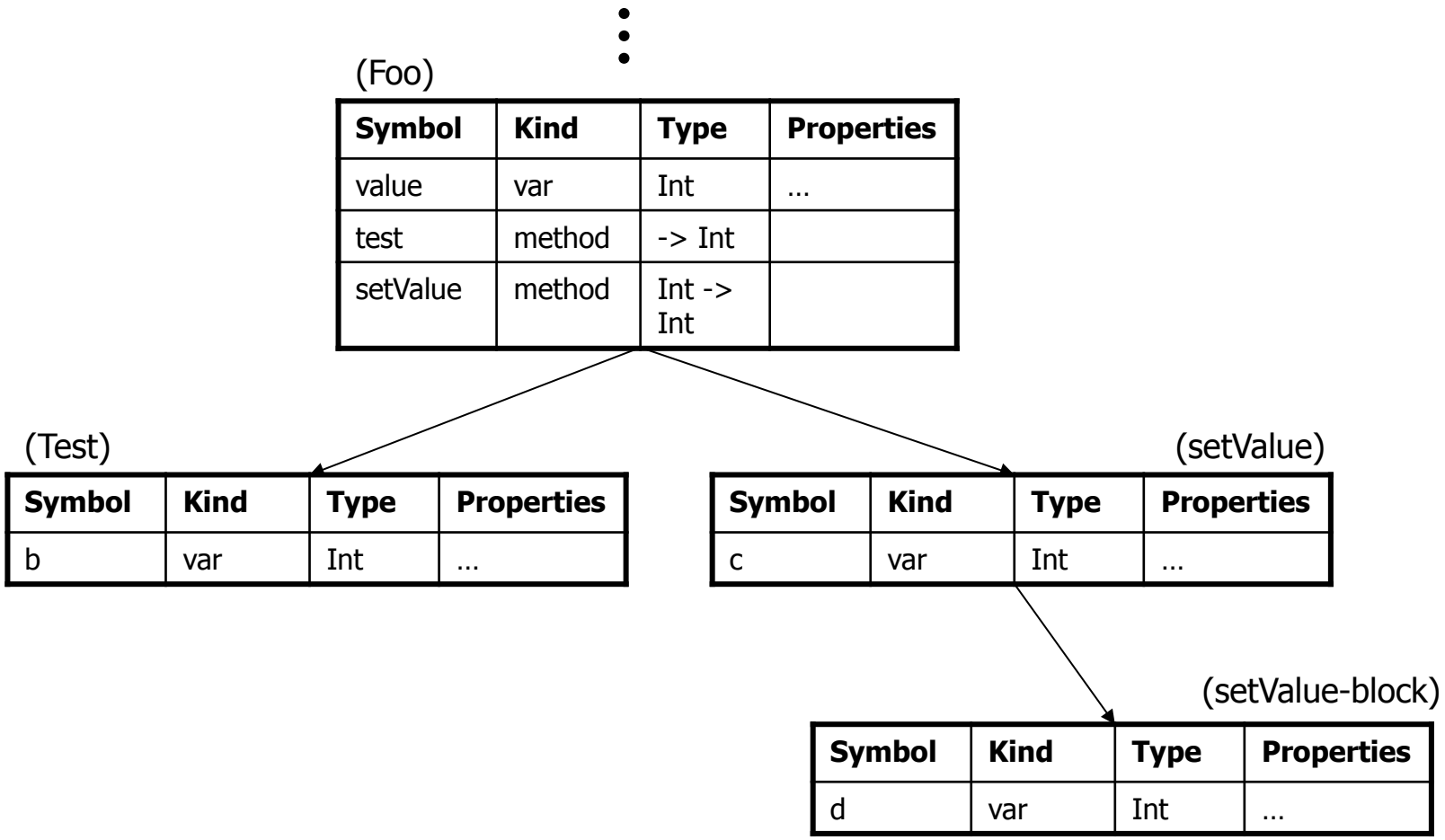
- before processing e
  - add definition of x to current definitions
  - override any other definition of x
- after processing e
  - remove definition of x
  - restore old definition of x

Symbol Table – Example 2

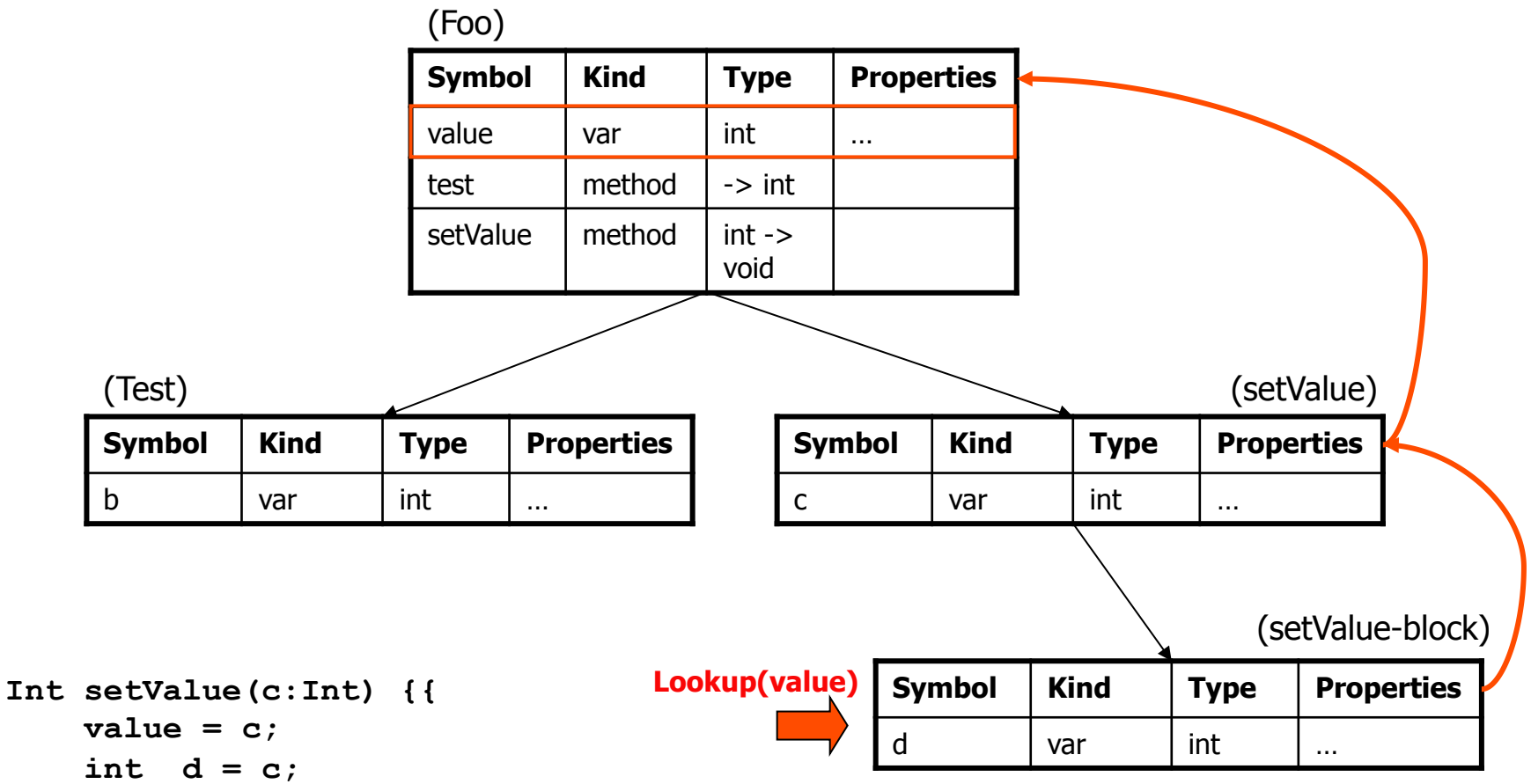
```
class Foo {
  int value = 39;
  int test() {
    int b = 3;
    value =+ b;
  };
  int setValue(c:Int)
    value = c;
    int d = c;
    c = c + d;
    value = c;

  }};
};
```

Symbol Table – Example 2

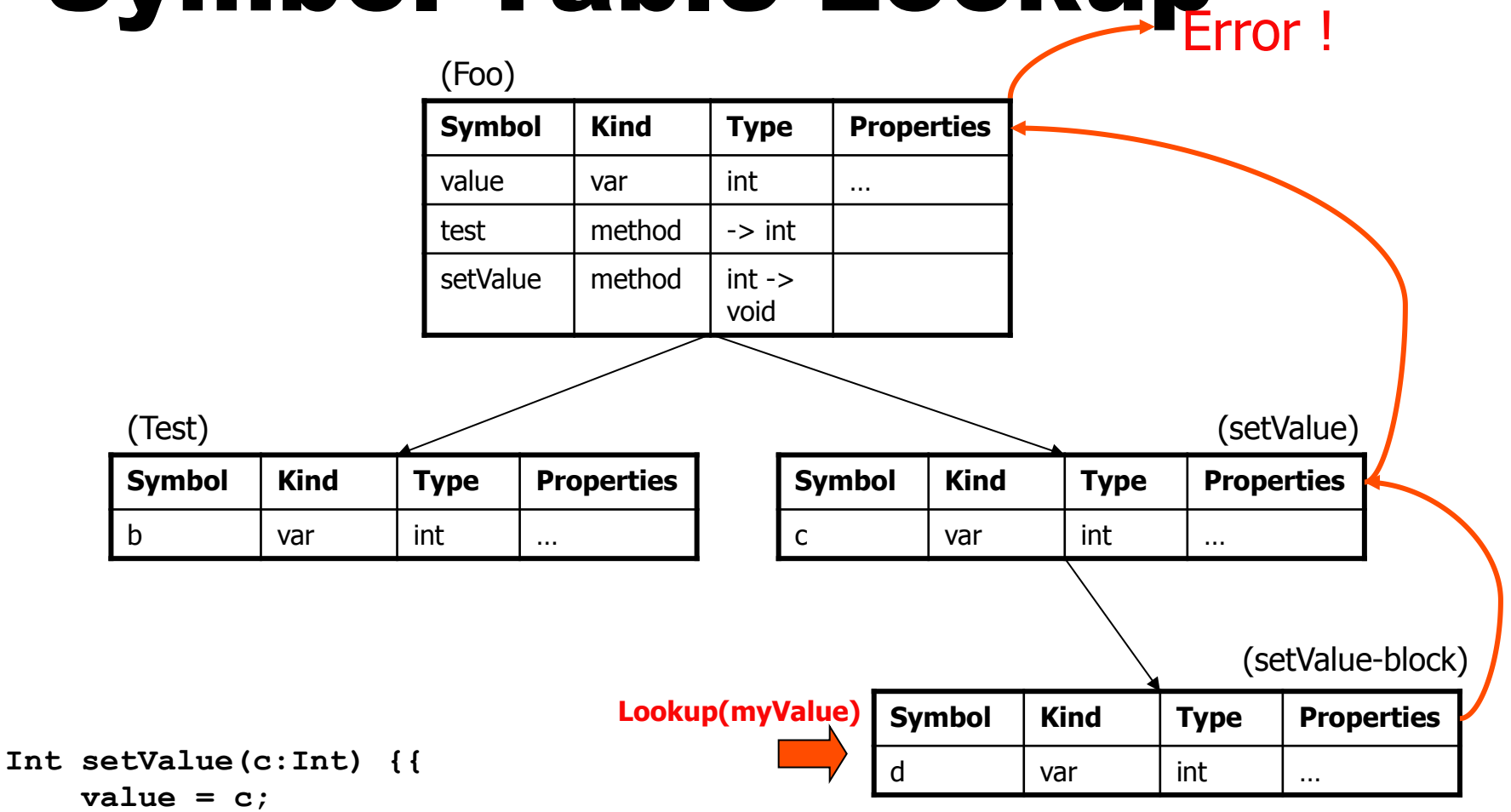


Symbol Table Lookup




```
Int setValue(c:Int) {{
    value = c;
    int  d = c;
    {
      c ← c + d;
      value ← c;
    };
  }};
```

Symbol Table Lookup

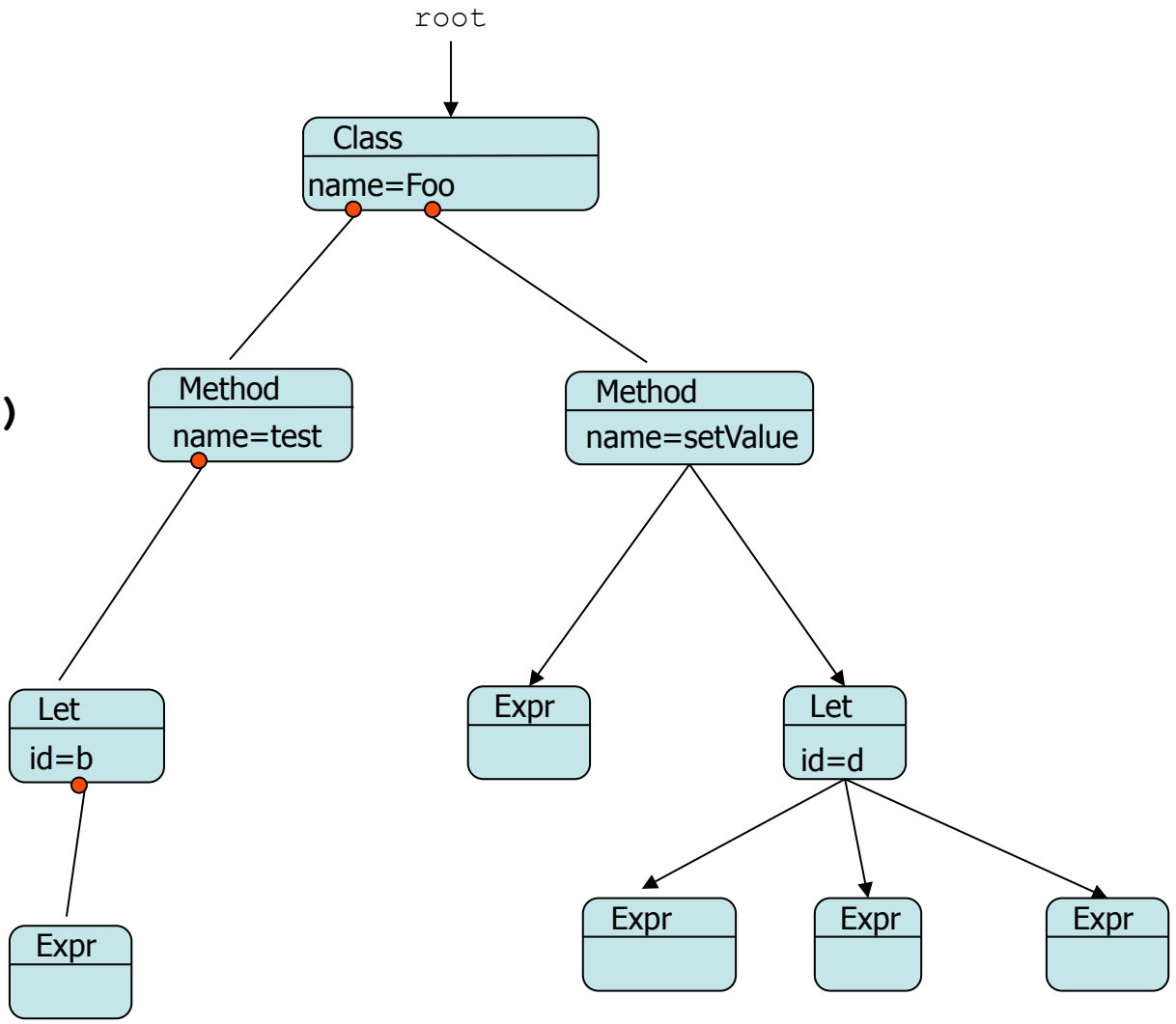



```
Int setValue(c:Int) {{
    value = c;
    int  d = c ;
     {
      c = c + d;
      myValue = c;
    };
  }};
```

Symbol Table Construction

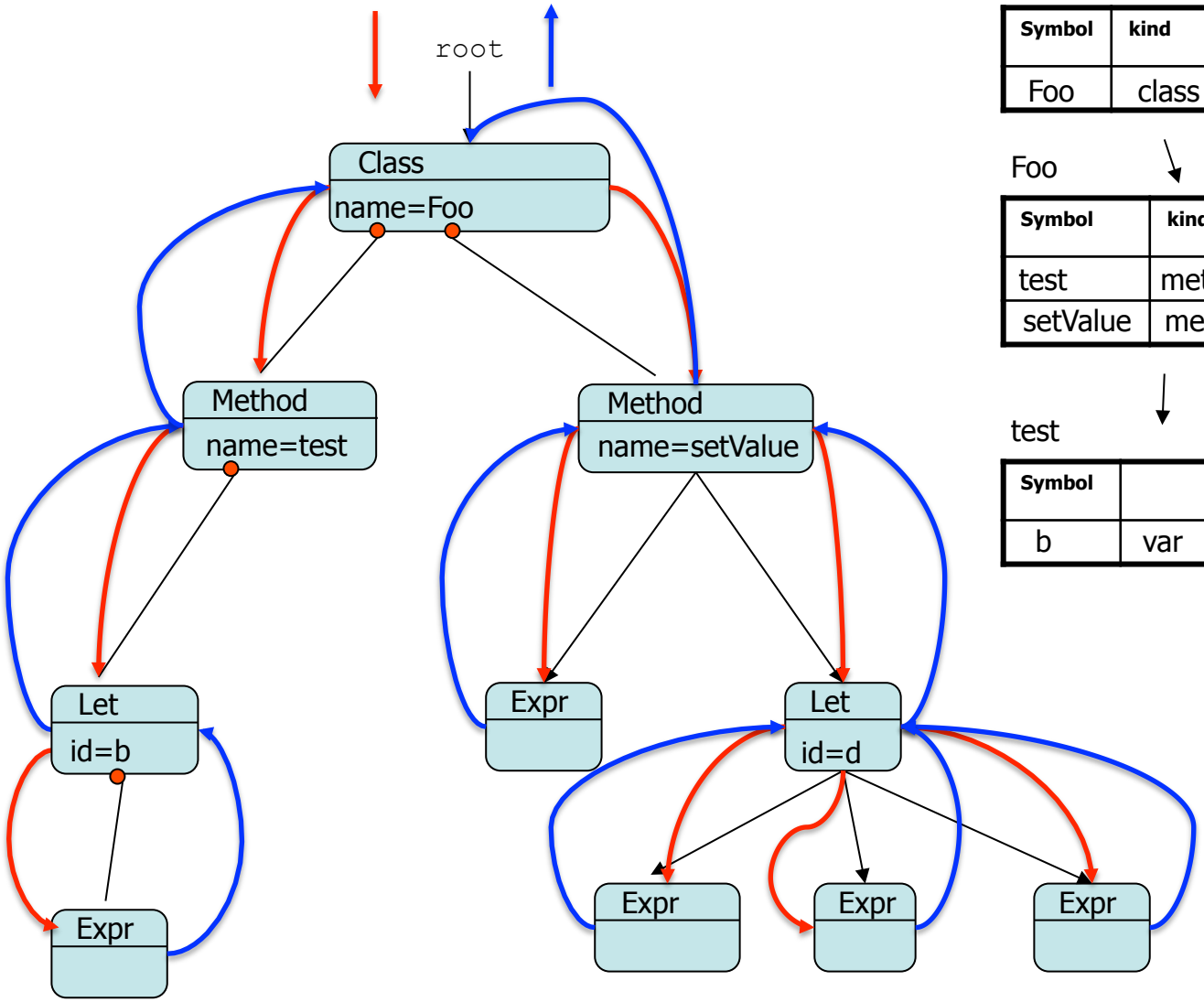
```
class Foo {
  int value = 39;
  int test() {
    int b = 3;
    value =+ b;
  };
  int setValue(c:Int)
    value = c;
    int d = c;
    c = c + d;
    value = c;

  }};
};
```



(some details omitted)

Symbol Table Construction via AST Traversal



globals

Symbol	kind		
Foo	class		

Foo

Symbol	kind		
test	method		
setValue	method		

test

Symbol			
b	var		

setValue

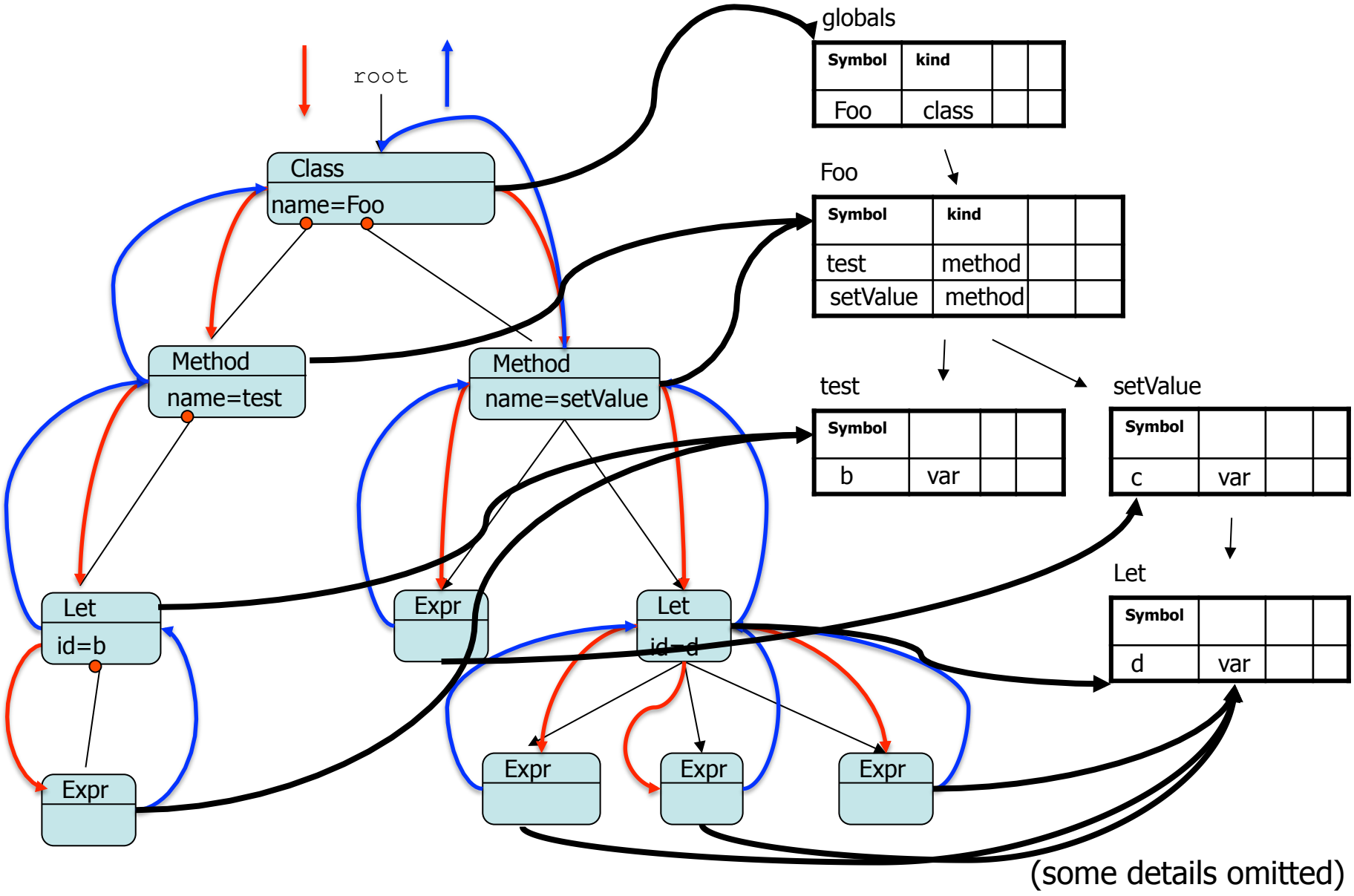
Symbol			
c	var		

Let

Symbol			
d	var		

(some details omitted)

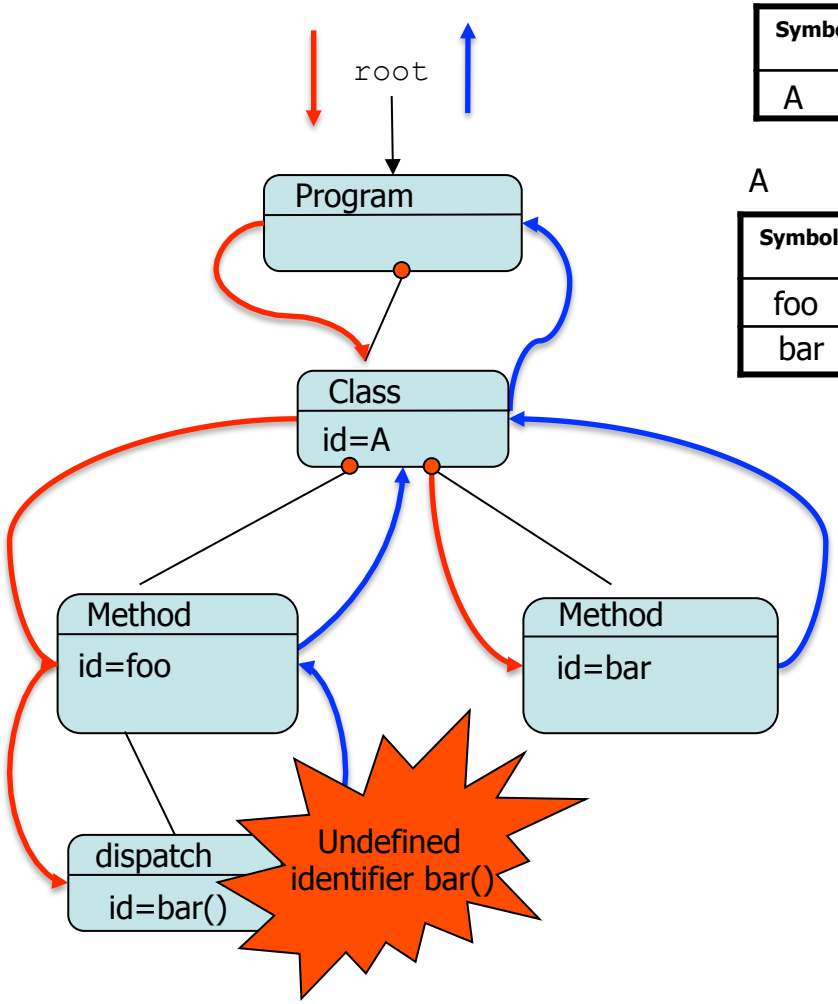
Symbol Table Construction via AST Traversal



(some details omitted)

Symbol Tables (cont'd)

```
class A {
   foo() {
        bar();
   }
   bar() {
        …
   }
}
```



globals

Symbol	kind		
A	class		

A

Symbol	kind		
foo	method		
bar	method		

Symbol Tables - Naïve solution

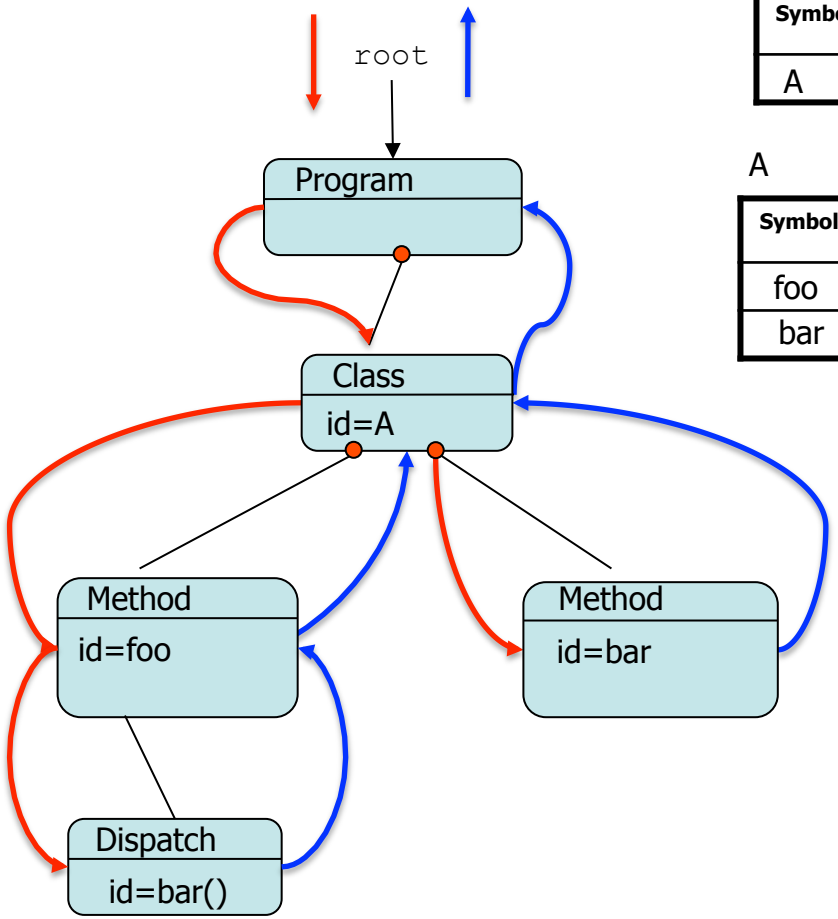
- Building visitor
  - Propagates (at least) a reference to the symbol table of the current scope
  - In some cases have to use type information (inherits)
- Checking visitor
  - On visit to node – perform check using symbol tables
    - Resolve identifiers
  - try to find symbol in table hierarchy
  - In some cases have to use global type table and type information
    - You may postpone these checks

Symbol Tables – less naïve solution

- Use forward references
- And/or construct some of the symbol table during parsing

Symbol Tables (cont'd)

```
class A {
   foo() {
        bar();
   }
   bar() {
        …
   }
}
```



globals

Symbol	kind		
A	class		

A

Symbol	kind		FREF
foo	method		
bar	method		**true**

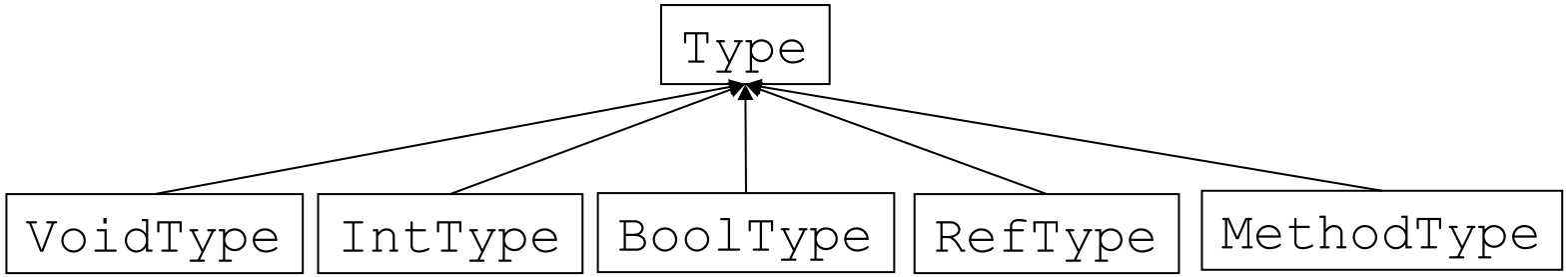
Next phase: type checking

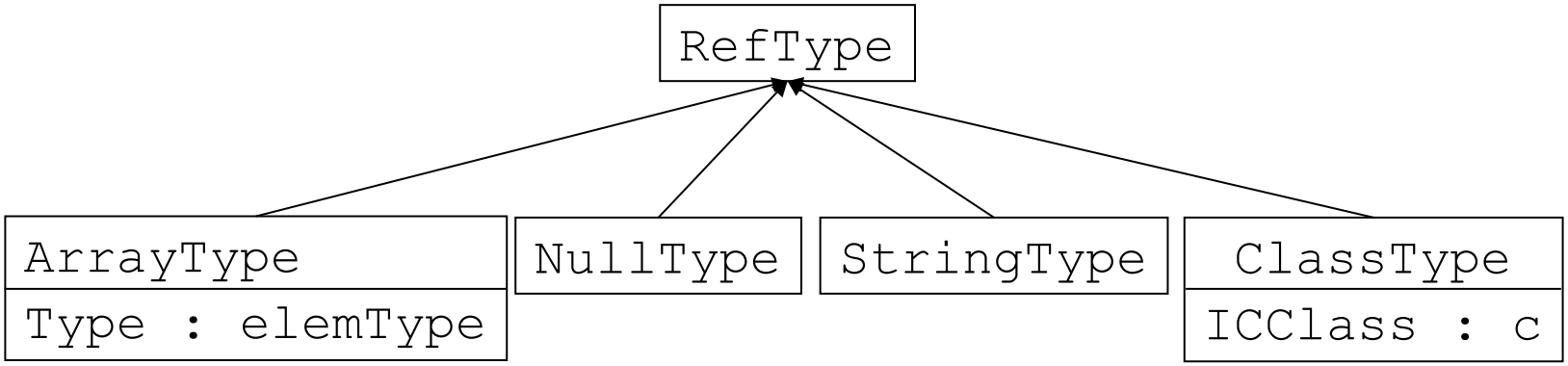
- First, record all pre-defined types (string,int,boolean,void,null)
- Second, record all user-defined types (classes, methods, arrays)
- Store all types in table
- Now, run type-checking algorithm

Type table

- Keeps a single copy for each type
  - Can compare types for equality by **==**
  - Records primitive types: int, bool, string, void, null
    - Initialize table with primitive types
  - User-defined types: arrays, methods, classes
- Used to record inheritance relation
  - Types should support **`subtypeOf(Type t)`**
- For IC enough to keep one global table
  - Static field of some class (e.g., Type)
  - In C/Java associate type table with scope

Possible type hierarchy





type ::= int | boolean | … | type `[` `]`