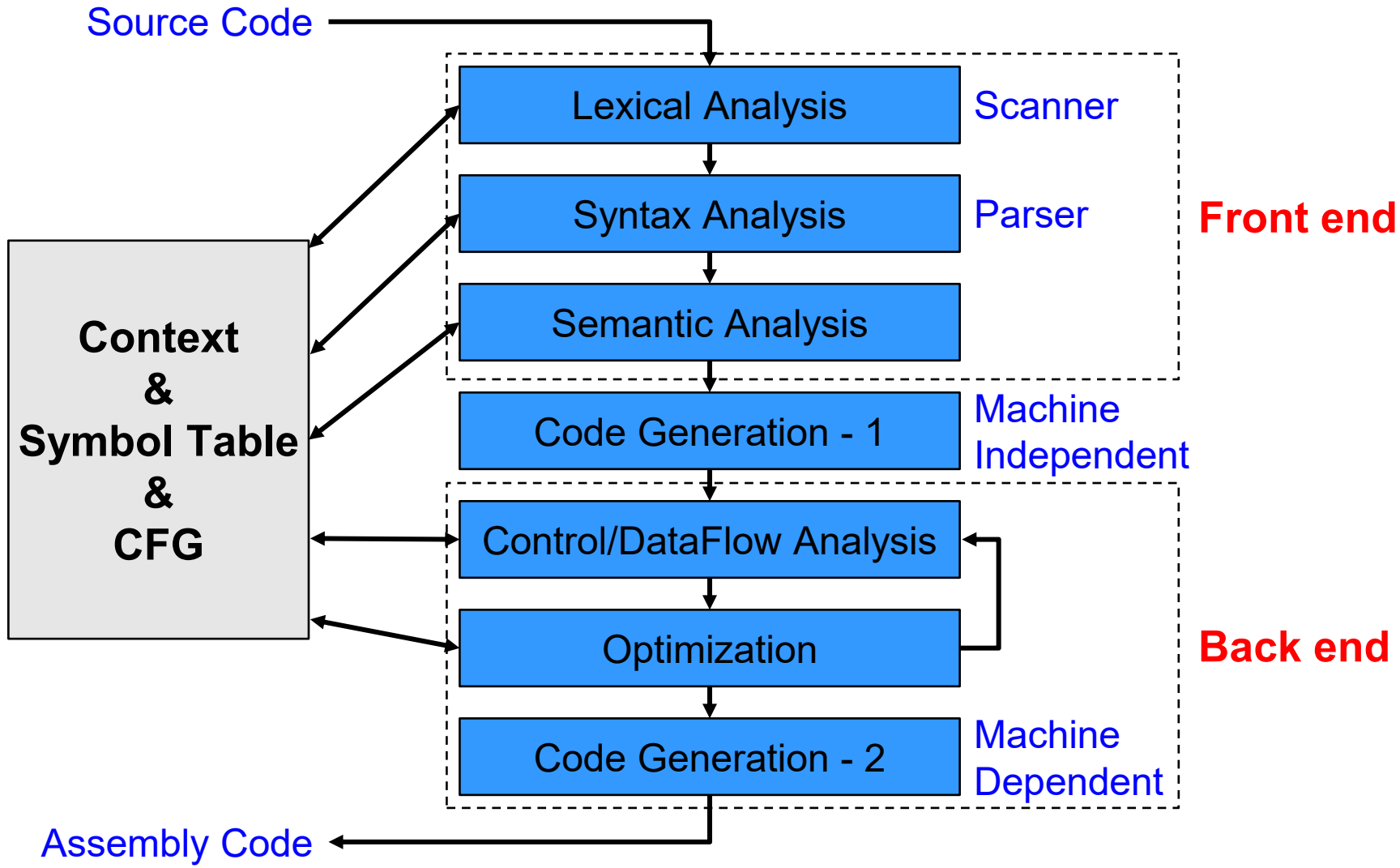
5. Semantic Analysis

2025 Fall

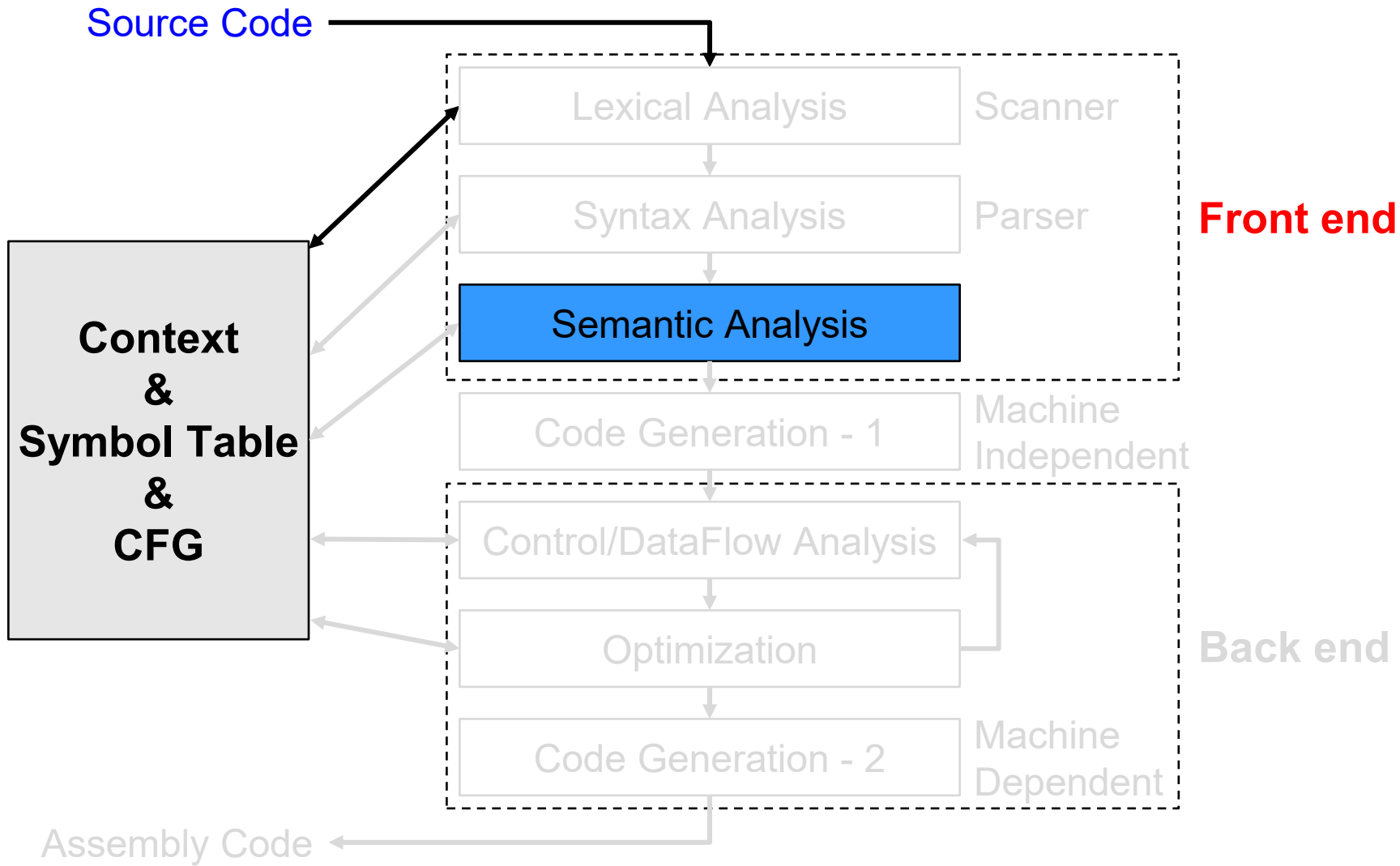
Hunjun Lee

Hanyang University

Semantic Analysis



Semantic Analysis



Semantic Analysis

- **Even after passing the lexical analysis and syntax analysis, there are still errors**
  - Correct usage of variables, objects, functions ,…
- **Semantic analysis ensures that the program satisfies a set of rules regarding the usage of programming constructs**
  - Are all identifiers declared before used?
  - Types?
  - Inheritance relationships?
  - Single definition of each class / methods
  - …

Class Exercise

- **Classify the grammar errors as lexical, syntax, semantic, or correct**

```
int a;
a = 1.0;
```

```
{
   int a;
   a = 1;
}
{
   a = 2;
}
```

```
int foo(int a)
{
 foo = 3;
}
```

```
1 int x;
x = 2;
```

```
int foo(int a)
{
   a = 3;
}
```

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

```
in$ a;
a = 1;
```

Categories of Semantic Analysis

- **Scopes: characterizes the declaration of identifiers and their usages**

- **Types: characterizes the types of values corresponding to variables, statements, expressions, …**

Categories of Semantic Analysis

- **Scopes: characterizes the declaration of identifiers and their usages**

- **Types: characterizes the types of values corresponding to variables, statements, expressions, …**

Scope Information - 1

- **Lexical scope: textual region in the program**
  - Ex) Statement block, function body, method body, source file, whole program

- **Scope of an identifier: the lexical scope its declaration refers to**
  - Definition of a variable is resolved by searching its containing block or function (if fails, search for the outer block and so on)

Scope Information - 2

```
extern int a;
...
int b;
...
{ int c;
     ...
      { int d;
      ...
      }
     ...
}
```

Scope a:
entire program

Scope b:
current file

Scope c: {~}

Scope d: {~}

```
int foo(int n) {
...
}

void boo() {
  goto lab;
  ...
  lab: i++;
  ... goto lab;
  ...
}
```

Scope n:
int foo {~}

Scope lab:
int boo {~}
(conventional labels
have function scope)

Symbol Tables

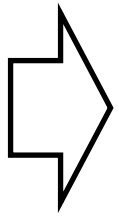
- **Semantic checks refer to properties of identifiers in the program – scope or type**
- **Need an environment to store identifier info (symbol table)**
- **Each entry contains name of an identifier + additional info**

Name	Kind	Type
foo	func	int → int
m	arg	int
n	arg	int
tmp	var	char

Scope Information Using Symbol Tables

- **Generate symbol tables according to lexical scope hierarchies**
  - Each table contains the symbols declared in a lexical scope

```
int x;

void f (int m) {
    float x, y; …;
    {int i,j; …;}
    {int x,k; …;}
}

int g (int n) {
    char t;
    …;
}
```



Global:

x	var	int
f	func	int → void
g	func	int → int

Function f:

m	arg	int
x	var	float
y	var	float

Function g:

n	arg	int
t	var	char

Function f.0:

i	var	int
j	var	int

Function f.1:

x	var	int
k	var	int

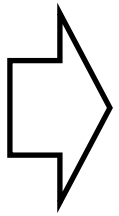
Identifiers with the Same Name

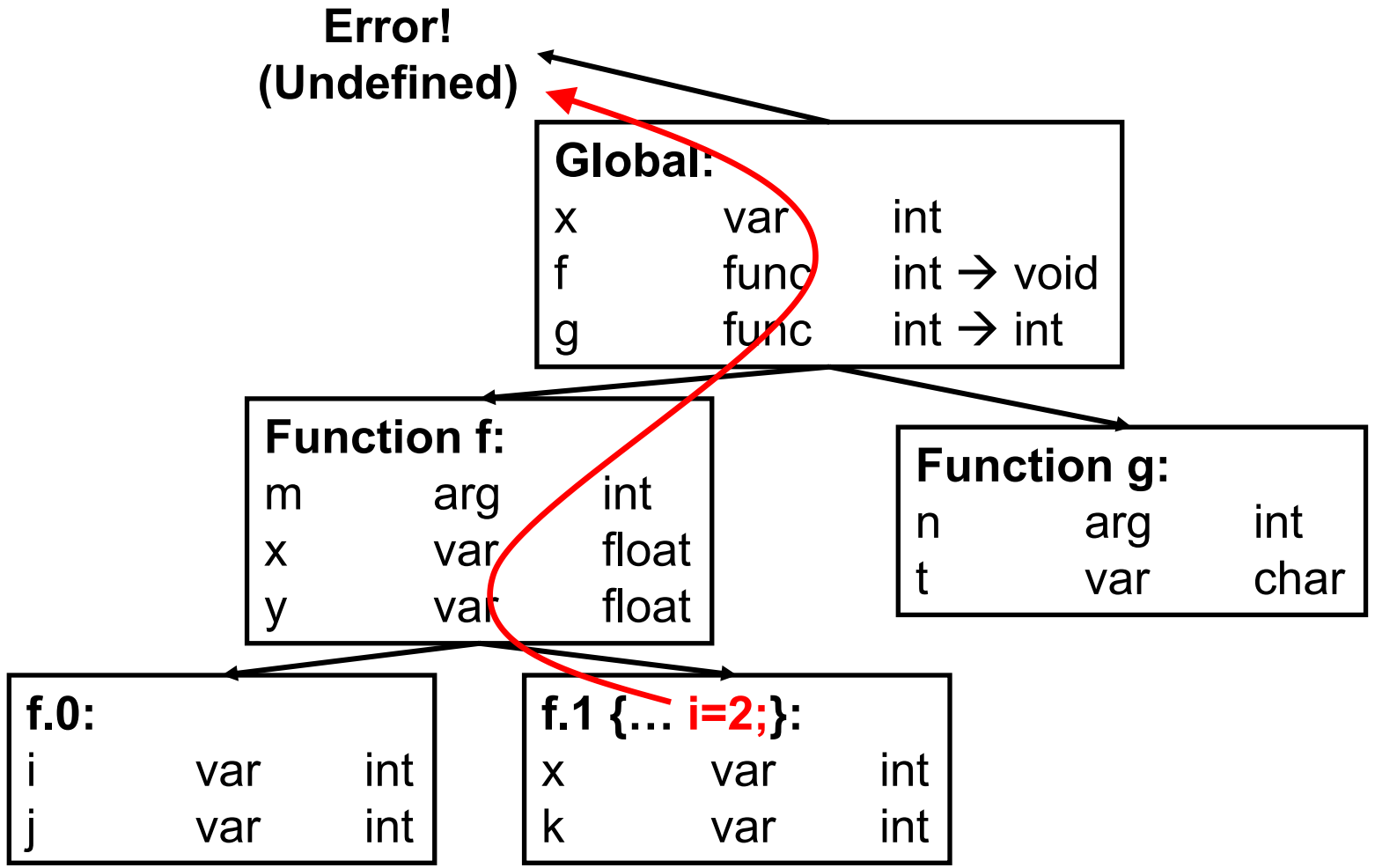
- **The hierarchical structure of symbol tables automatically solves the problem resolving name collision**

- **Traverse the hierarchy upward to determine the declaration**
  - Start from the current scope
  - Go up the hierarchy until finding an identifier with the same name

Example - 1

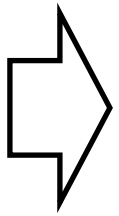
```
int x;

void f (int m) {
    float x, y; …;
    {int i,j; …;}
    {int x,k; i=2;}
}

int g (int n) {
    char t;
    …;
}
```

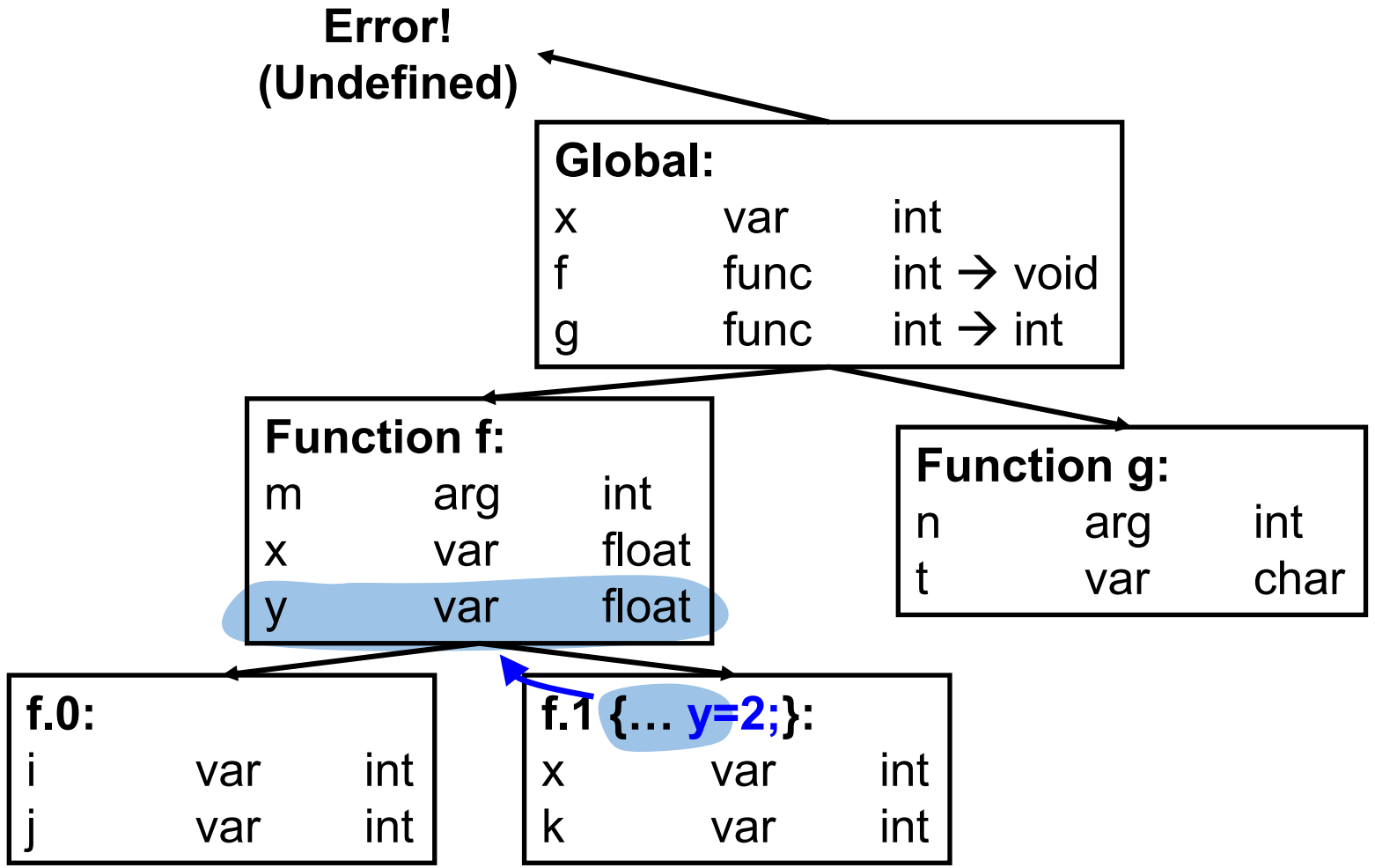




Example - 2

```
int x;

void f (int m) {
    float x, y; …;
    {int i,j; …;}
    {int x,k; y=2;}
}

int g (int n) {
    char t;
    …;
}
```

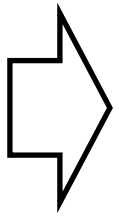


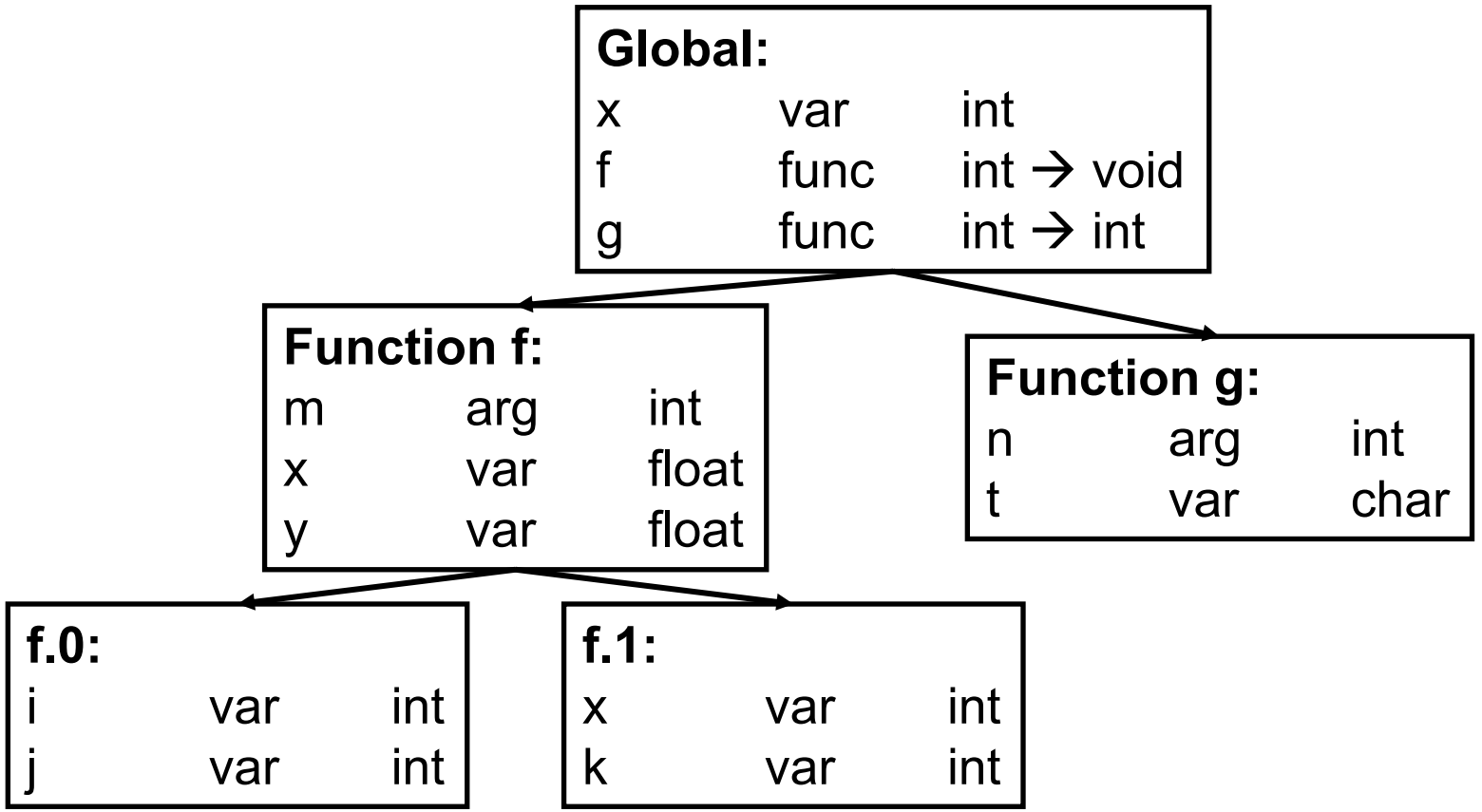


Class Exercise

- **Associate each definition of x with its appropriate symbol entry**

```
int x;

void f (int m) {
    float x, y; …;
    {int i,j; x=1;}
    {int x,k; x=2;}
}

int g (int n) {
    char t;
    x = 3;
}
```





Implementing Symbol Table

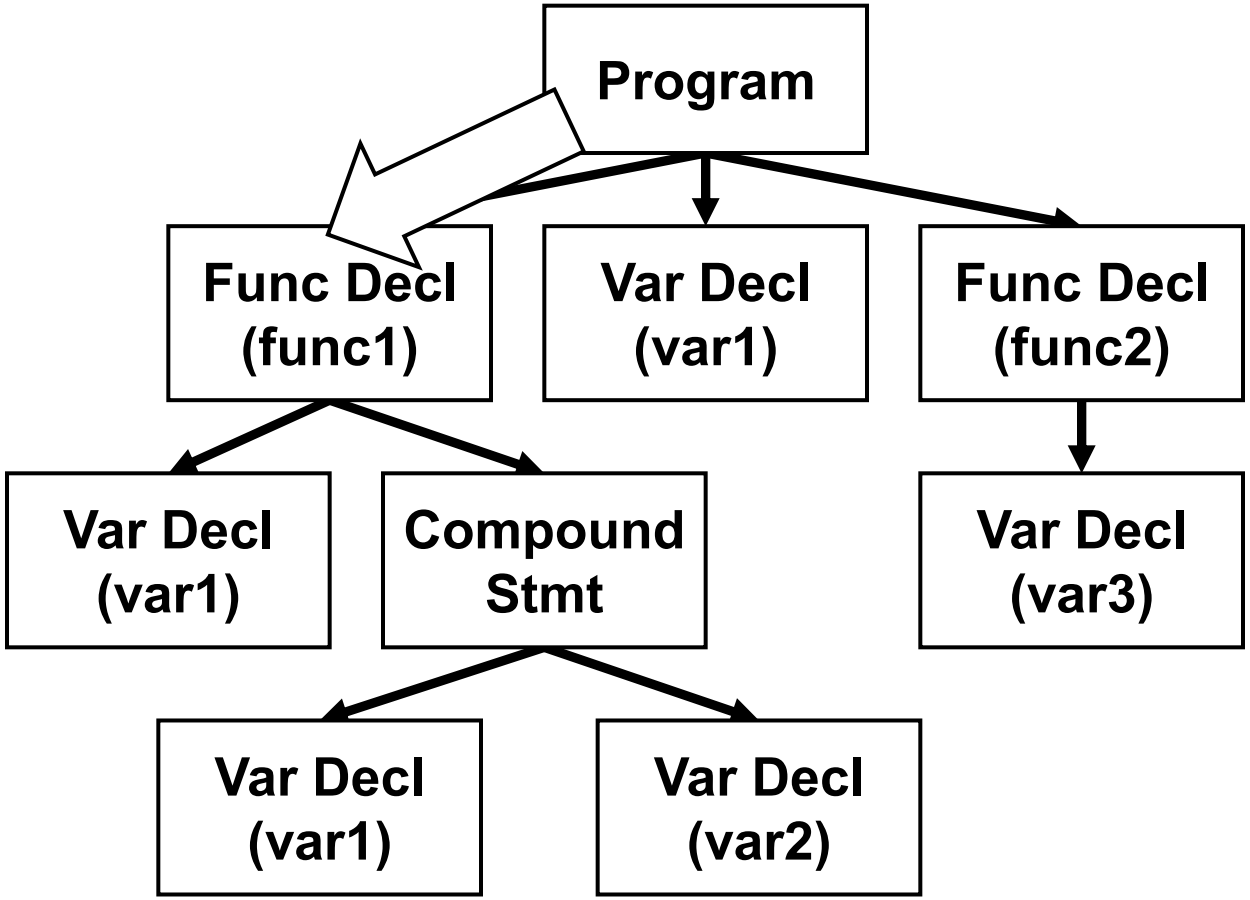
- **Five operations:**
  - Insert scope: start a new nested scope
  - Exit scope: exit the current scope
  - Find symbol(x): Search for x in the hierarchy
  - Add symbol(x): Add a symbol x to the table
  - Check scope(x): Check if x is defined in the current scope (optional)

- **We can build the symbol tables during parsing (or after constructing the AST)**

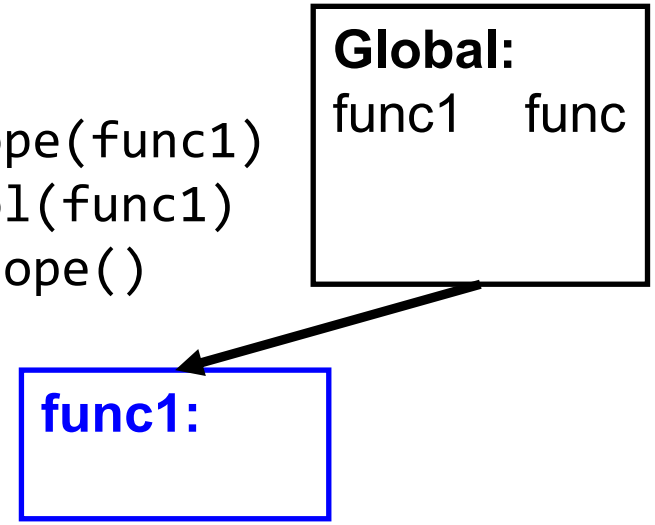
- **We should generate the symbol tables before semantic analysis**

Build Symbol Table

- We generate the symbol table when traversing over the AST

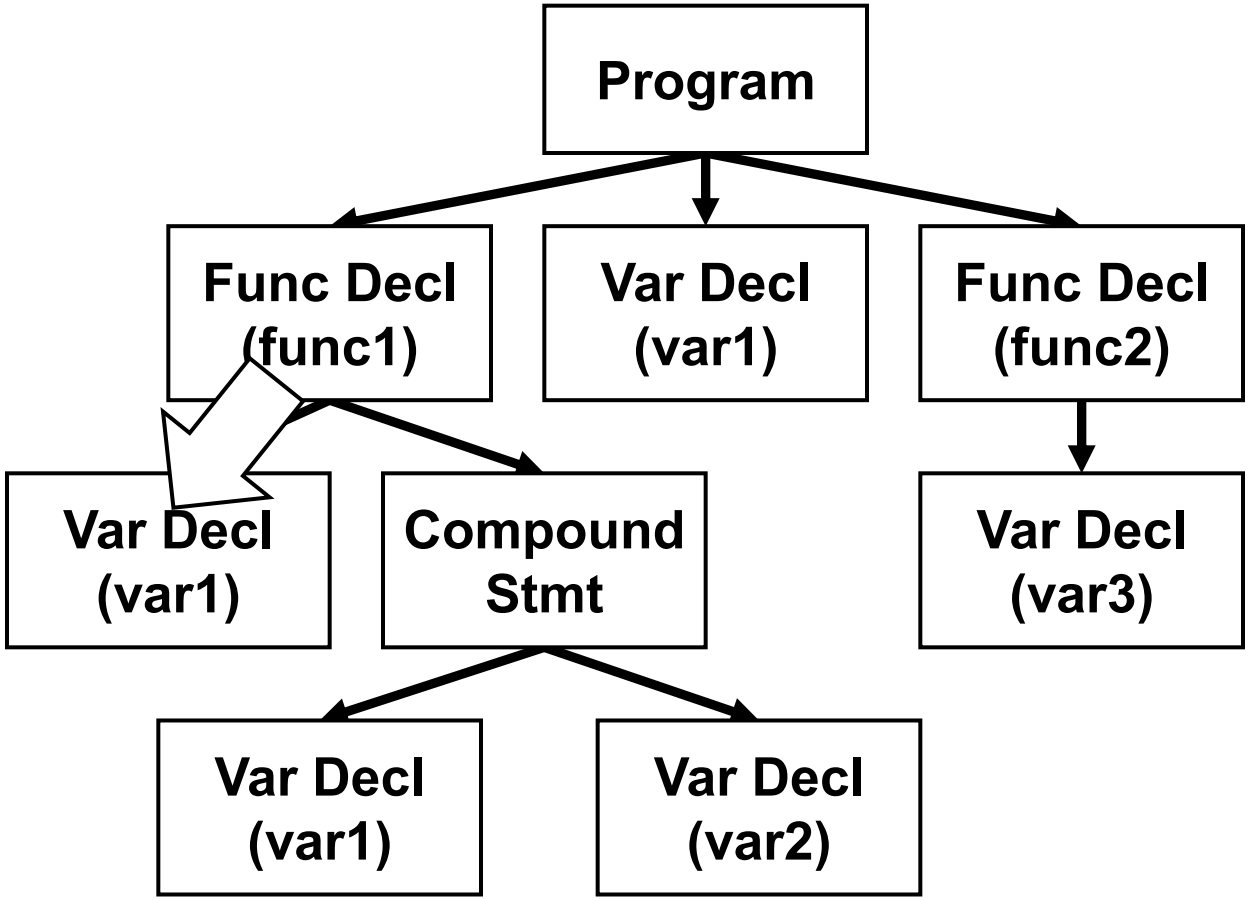


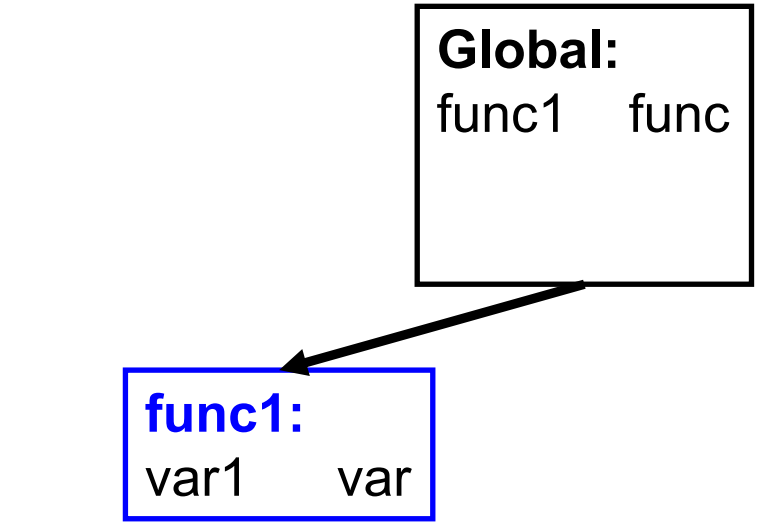

```
check_scope(func1)
add_symbol(func1)
Insert_scope()
```



Build Symbol Table

- We generate the symbol table when traversing over the AST

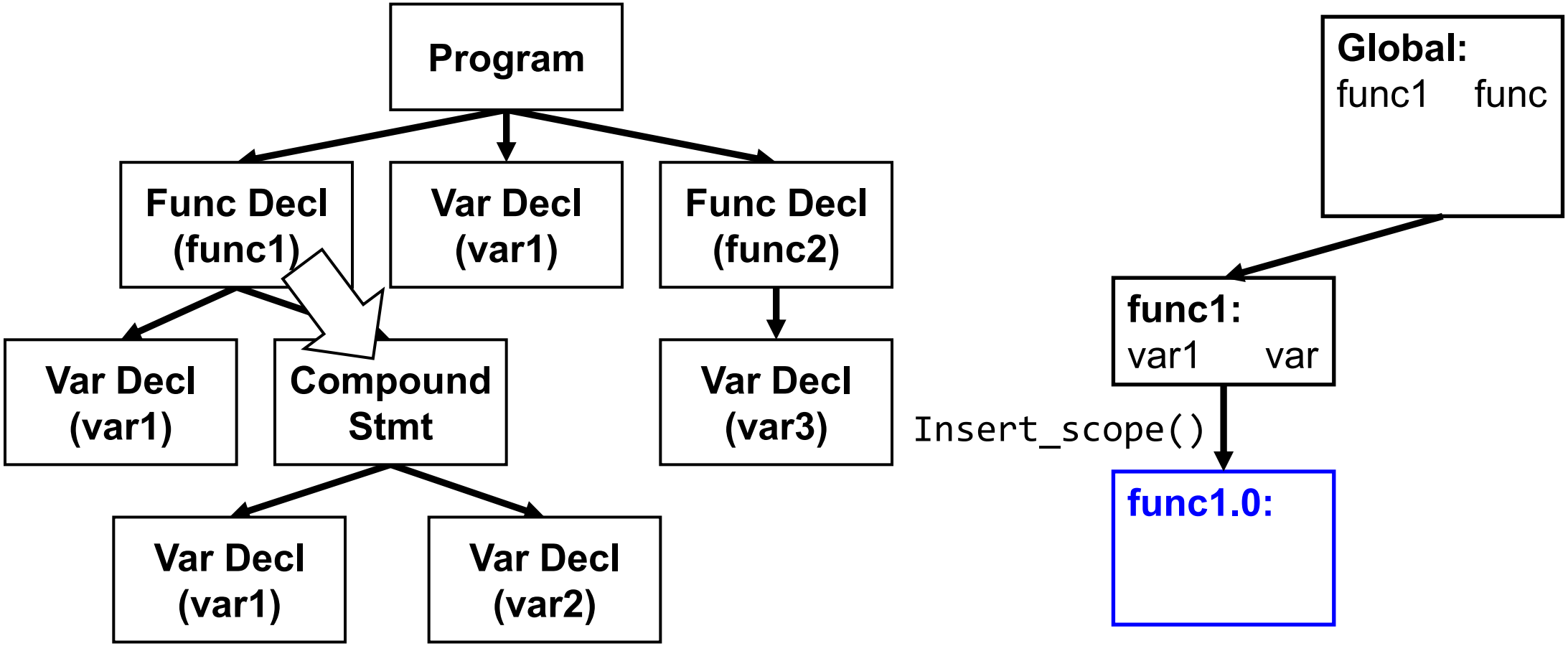





```
check_scope(var1)
add_symbol(var1)
```

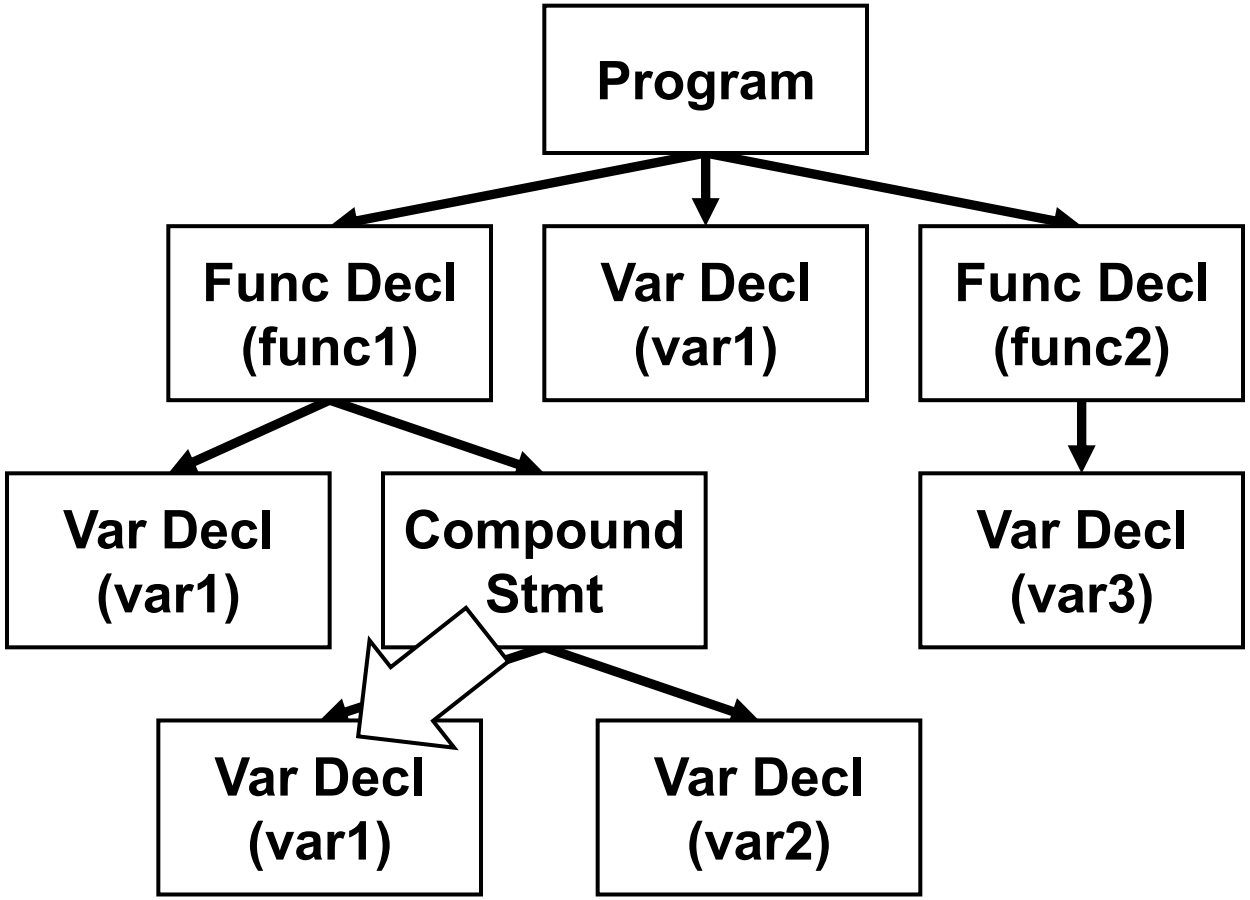
Build Symbol Table

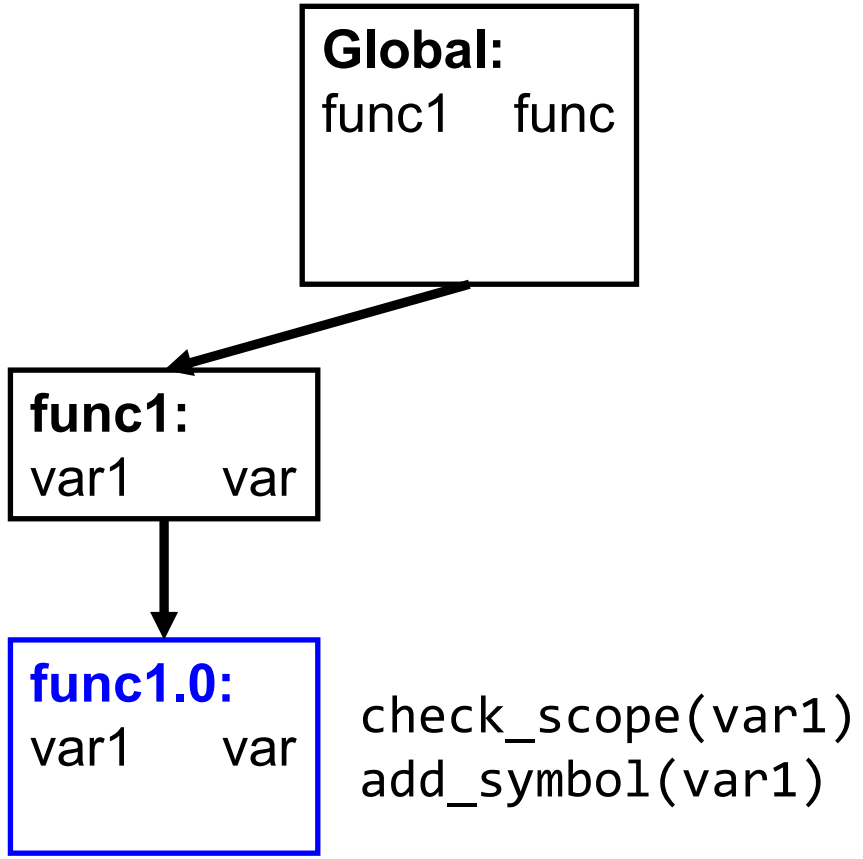
- We generate the symbol table when traversing over the AST



Build Symbol Table

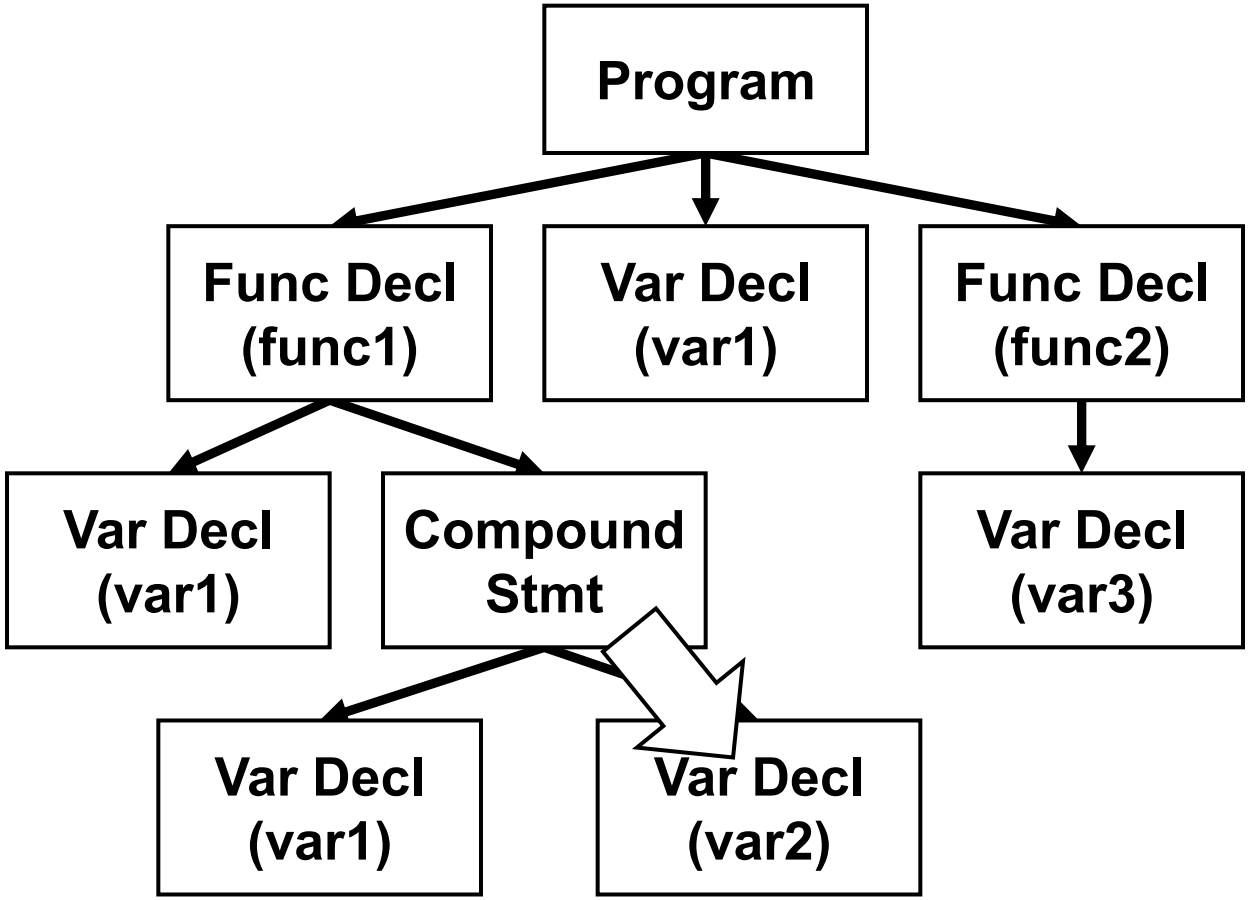
- We generate the symbol table when traversing over the AST

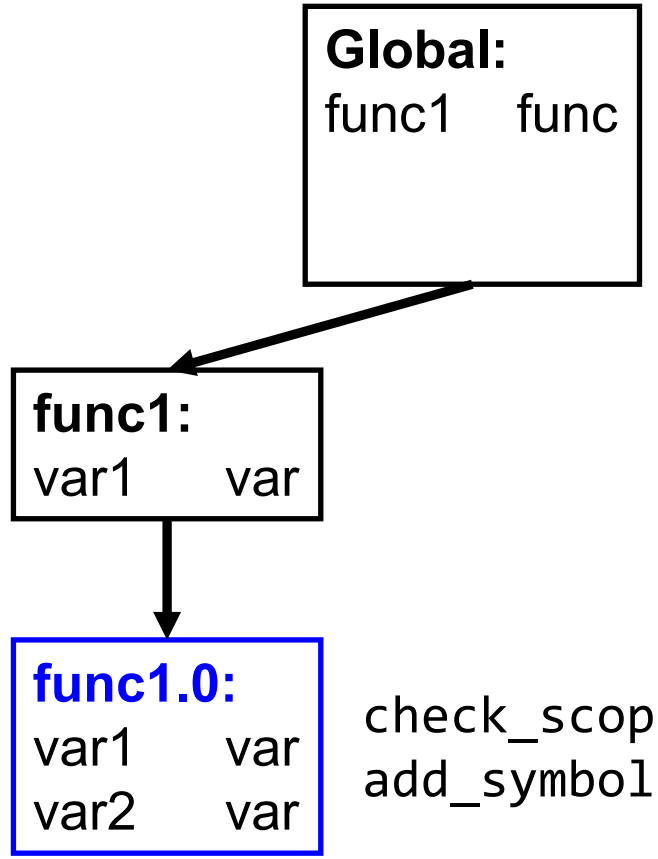




Build Symbol Table

- We generate the symbol table when traversing over the AST

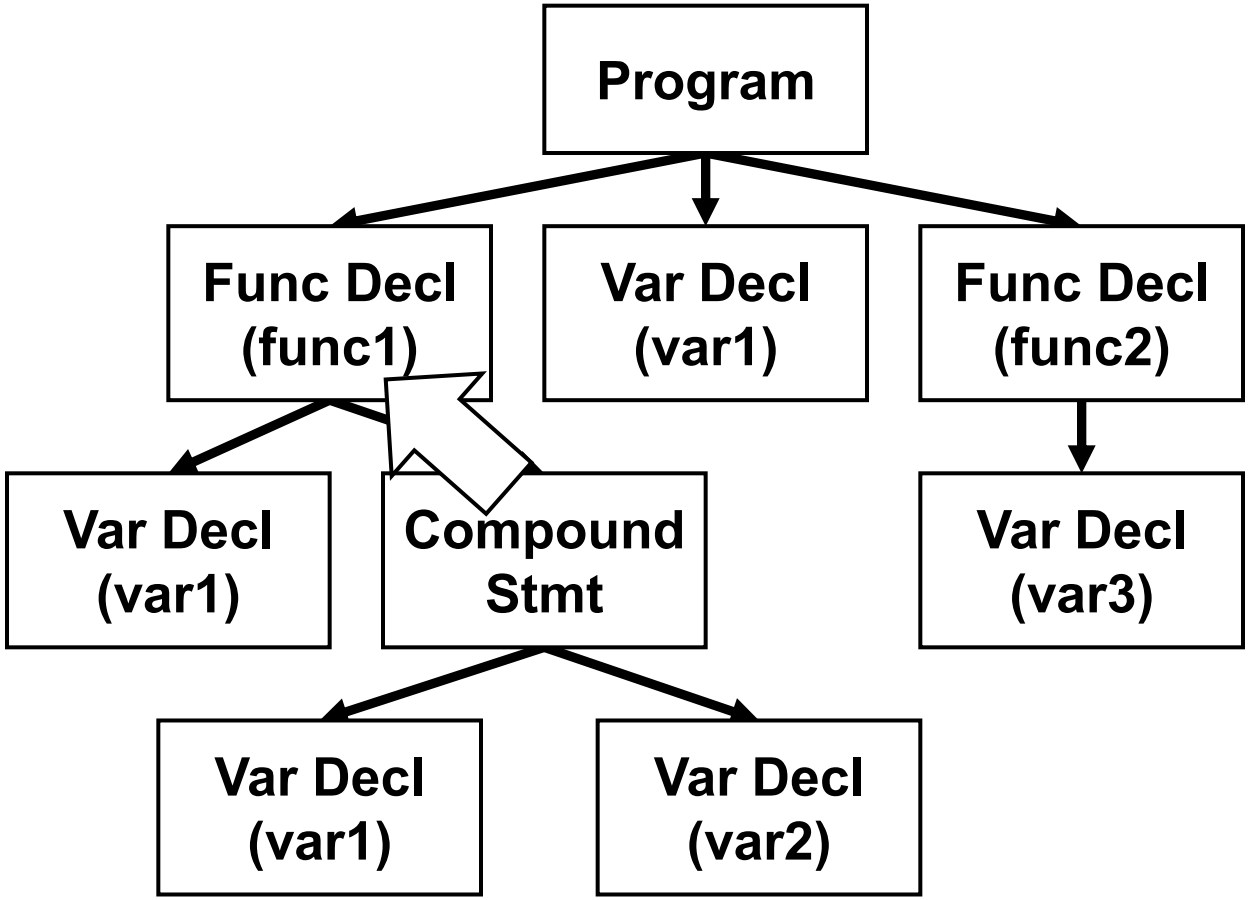


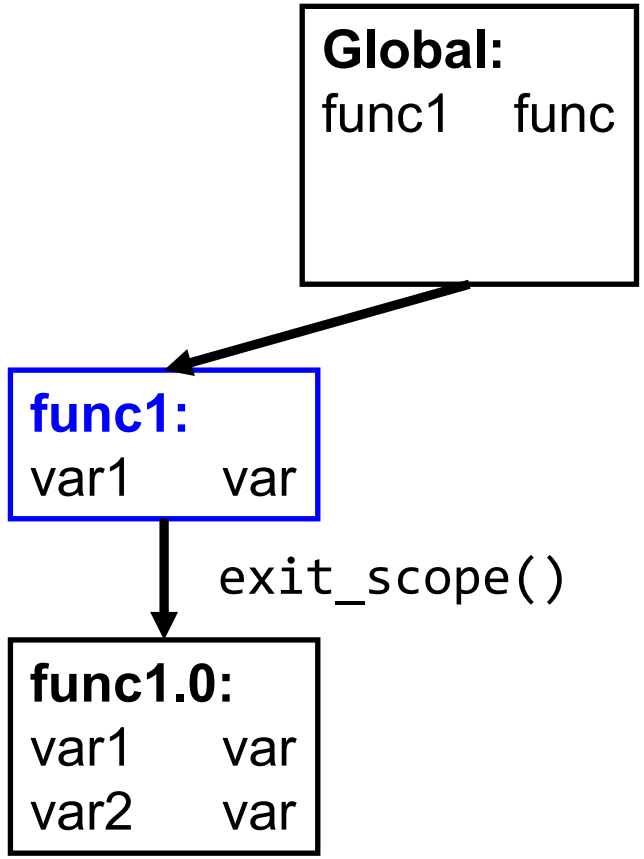



```
check_scope(var2)
add_symbol(var2)
```

Build Symbol Table

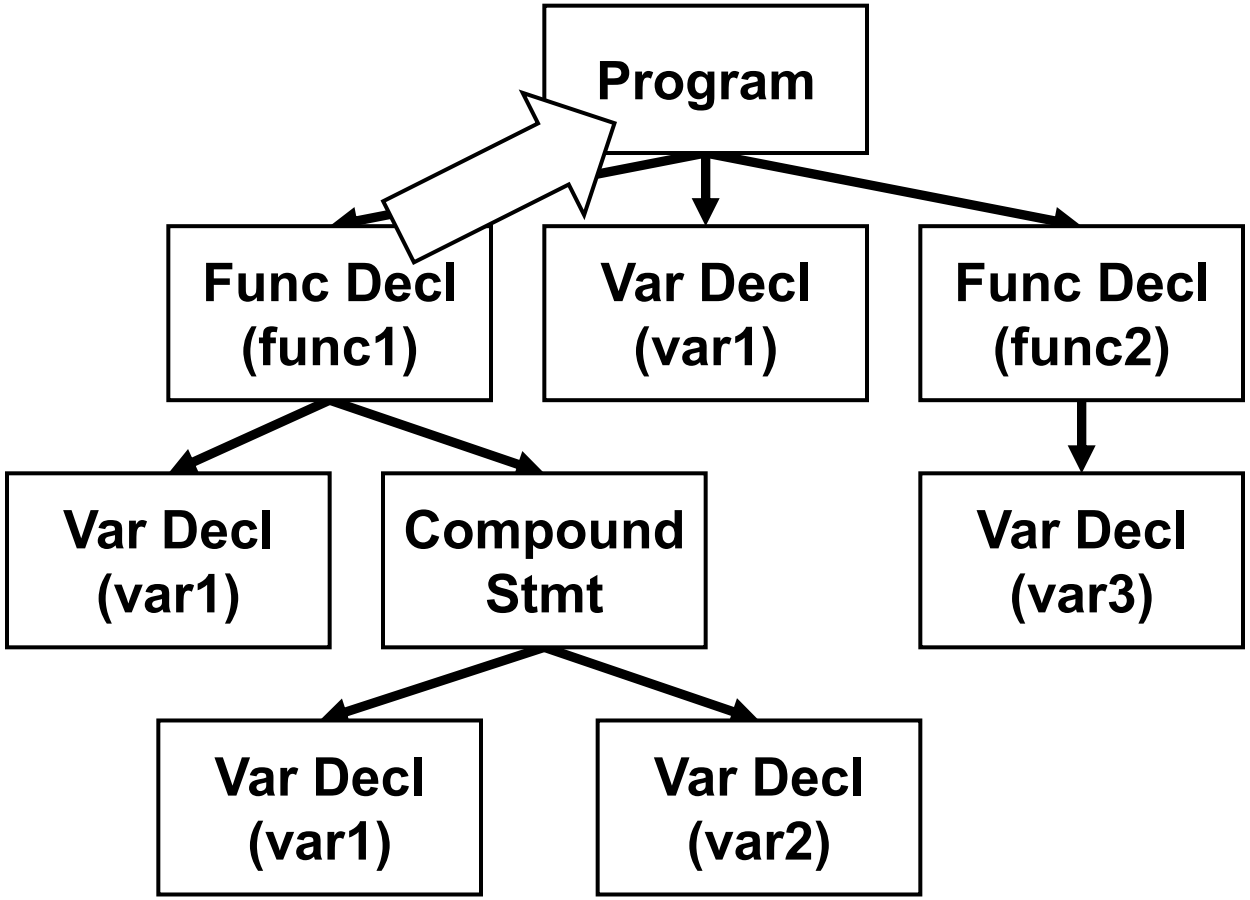
- We generate the symbol table when traversing over the AST

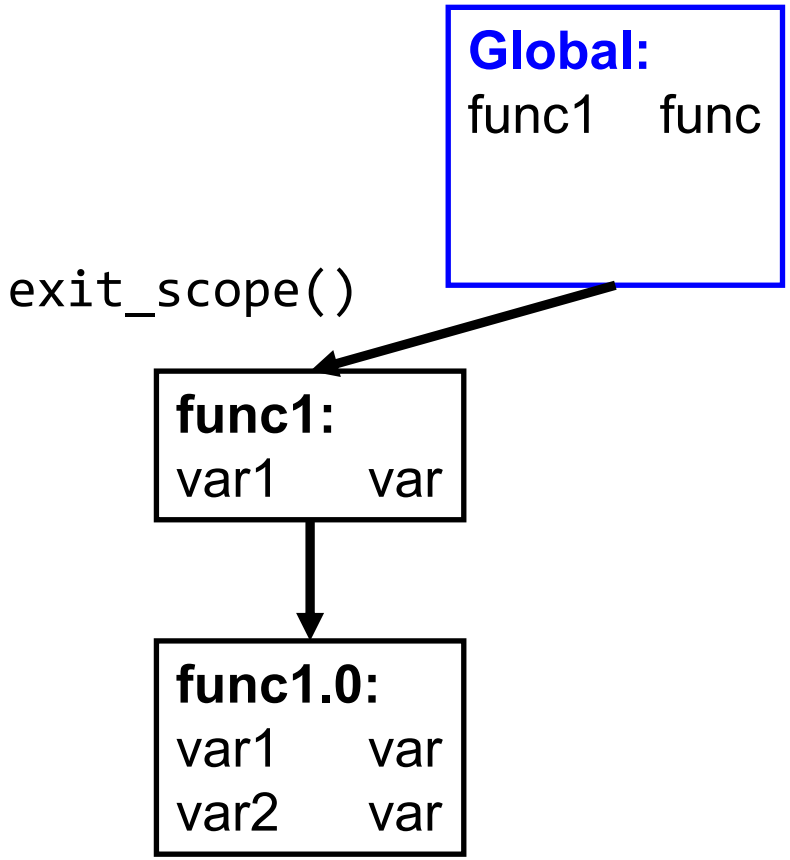




Build Symbol Table

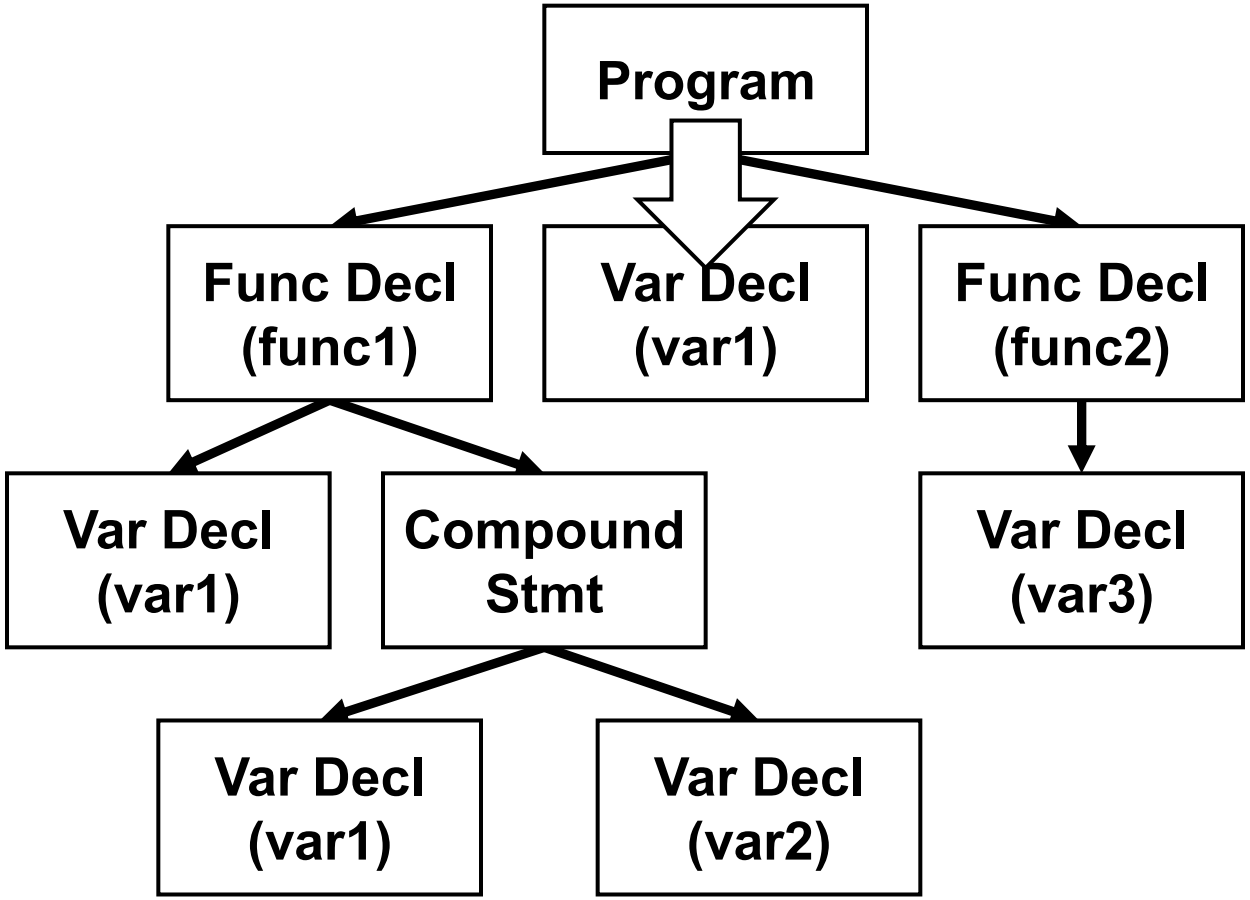
- We generate the symbol table when traversing over the AST



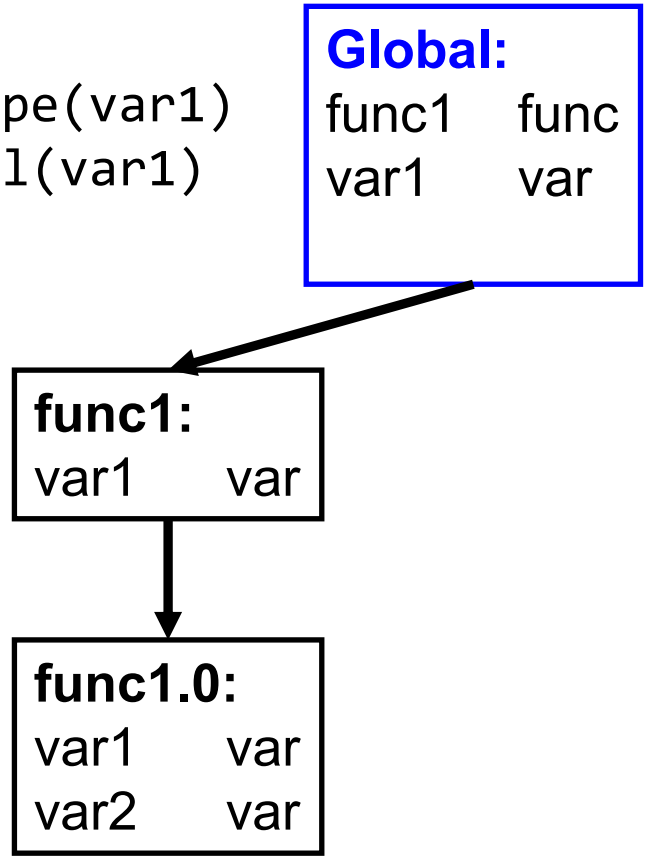


Build Symbol Table

- We generate the symbol table when traversing over the AST

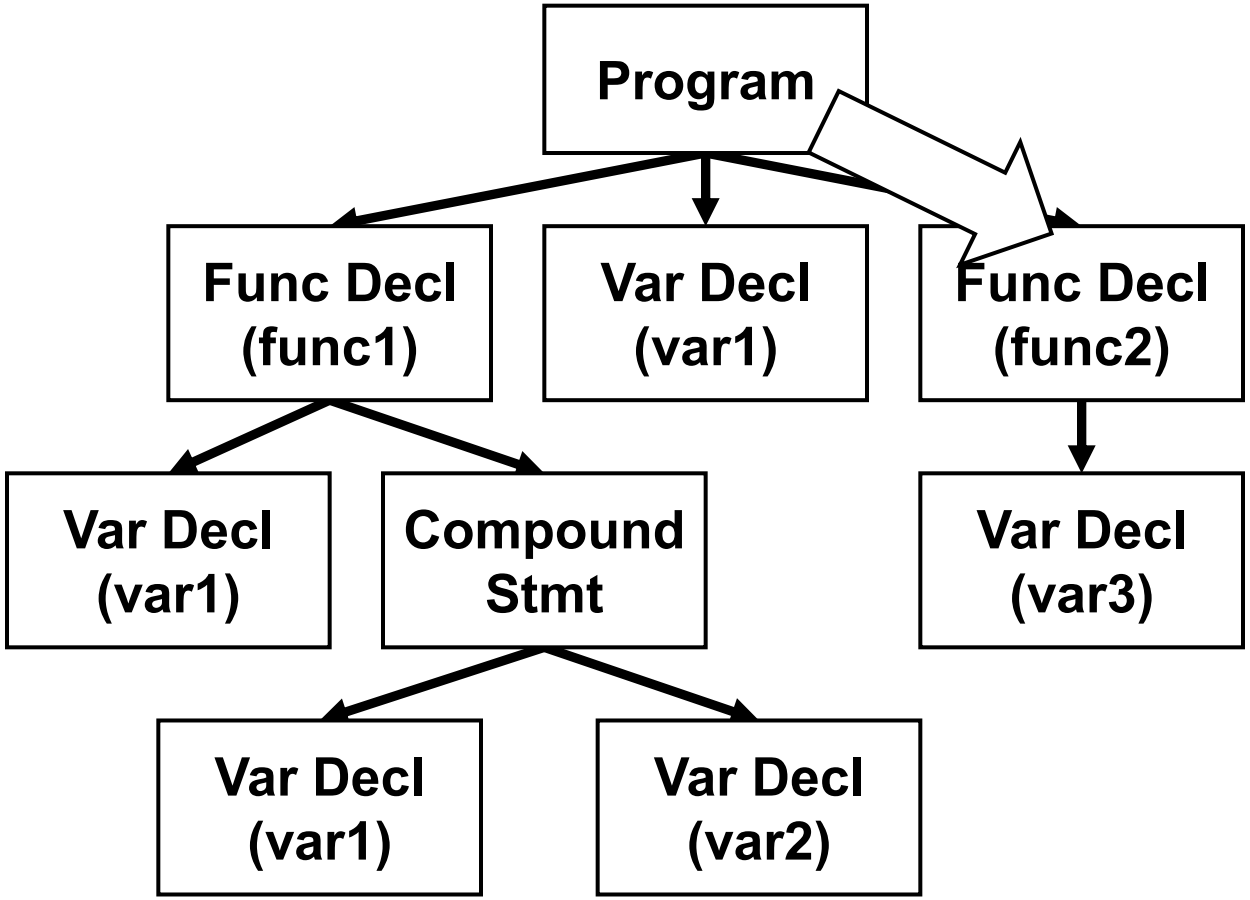



```
check_scope(var1)
add_symbol(var1)
```

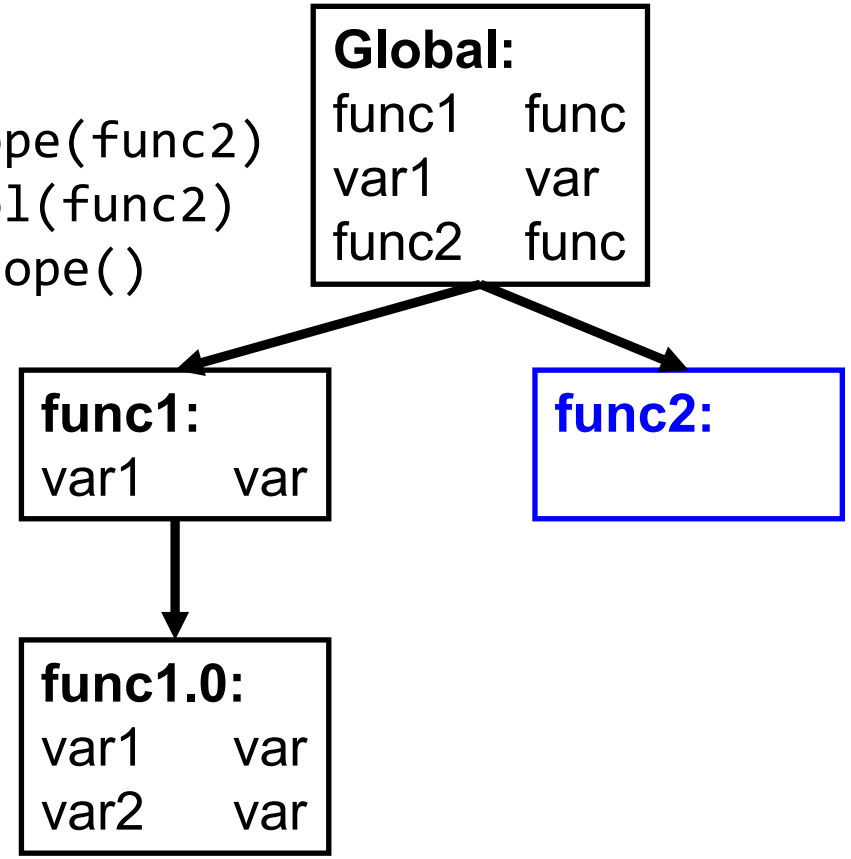


Build Symbol Table

- We generate the symbol table when traversing over the AST

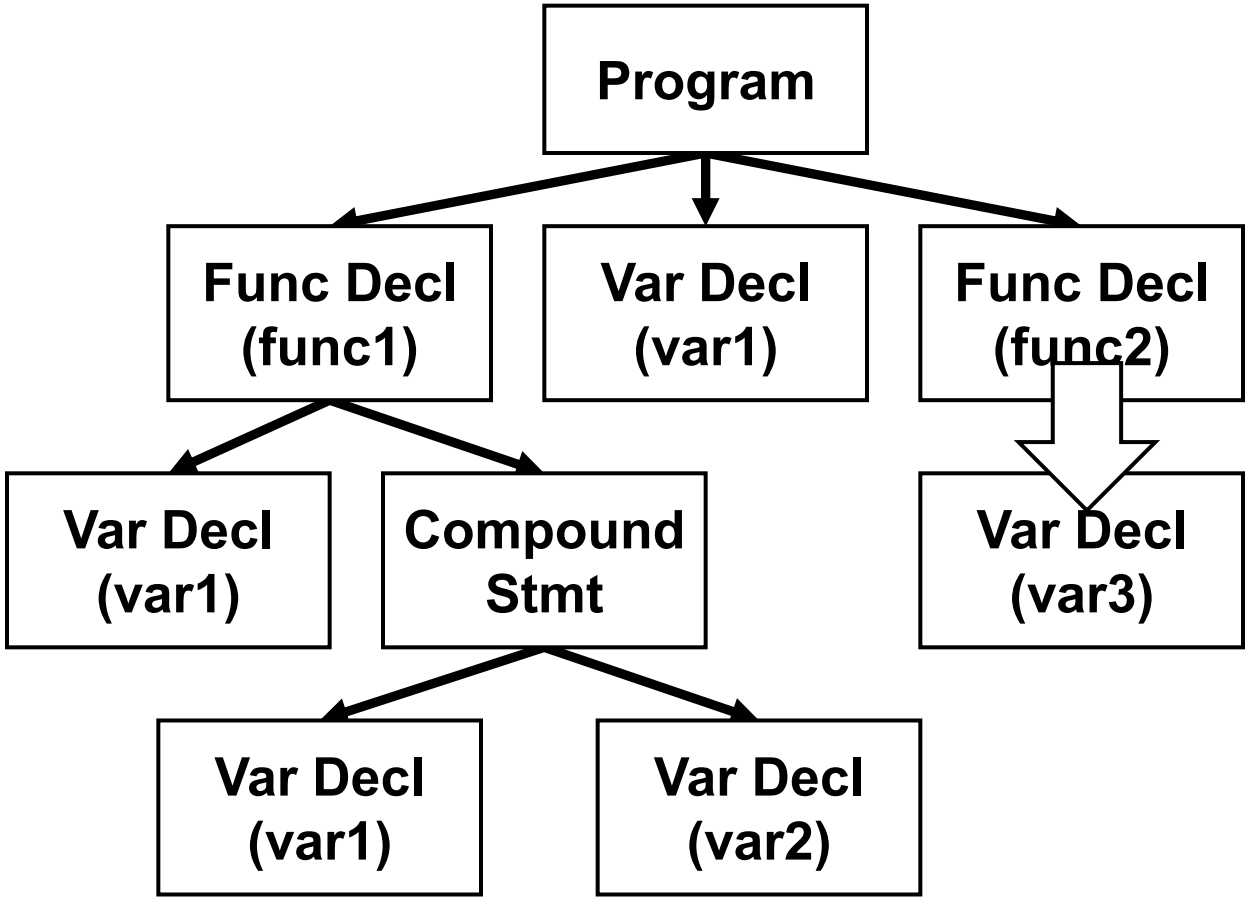


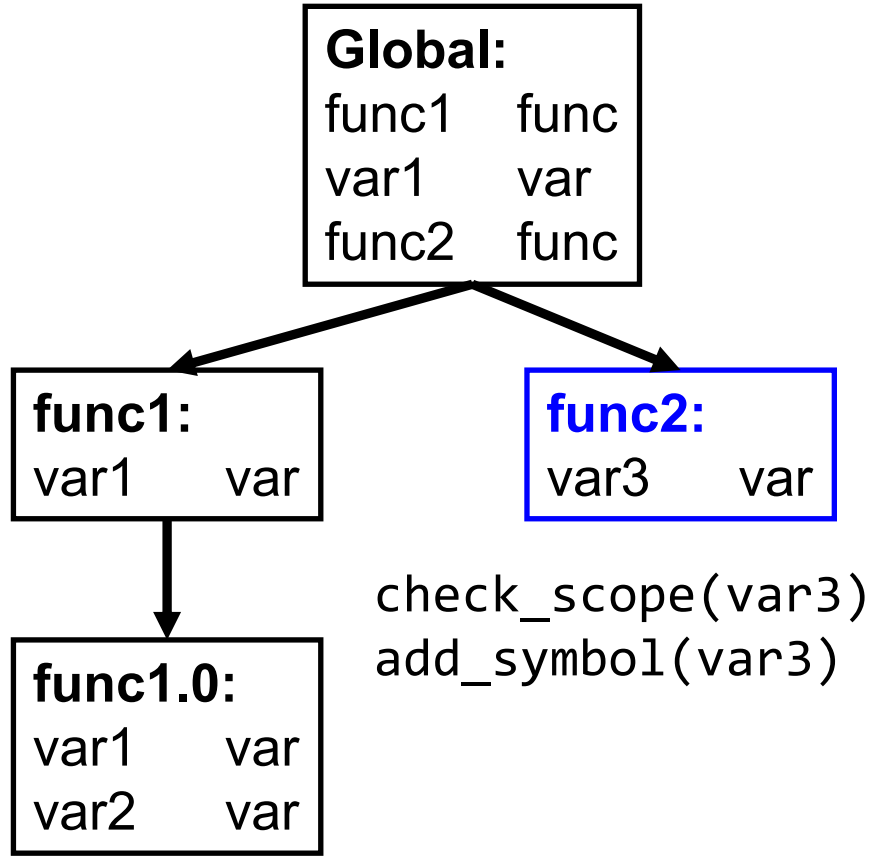

```
check_scope(func2)
add_symbol(func2)
Insert_scope()
```



Build Symbol Table

- We generate the symbol table when traversing over the AST





Function Declaration and Usage Types

- **There are two methods to implement scopes**

C-Minus …

```
int foo(int x, int y) {
    …
}

int boo(int z) {
    int i, j;
    foo(i, j);
}

…
```

Declare the functions before usage

Python …

```
int boo(int z) {
    int i, j;
    foo(i, j);
}

int foo(int x, int y) {
    …
}

…
```

Can use functions before declaration

C …

```
int boo(int z);

int foo(int x, int y) {
    …
}

int boo(int z) {
    int i, j;
    foo(i, j);
}
```

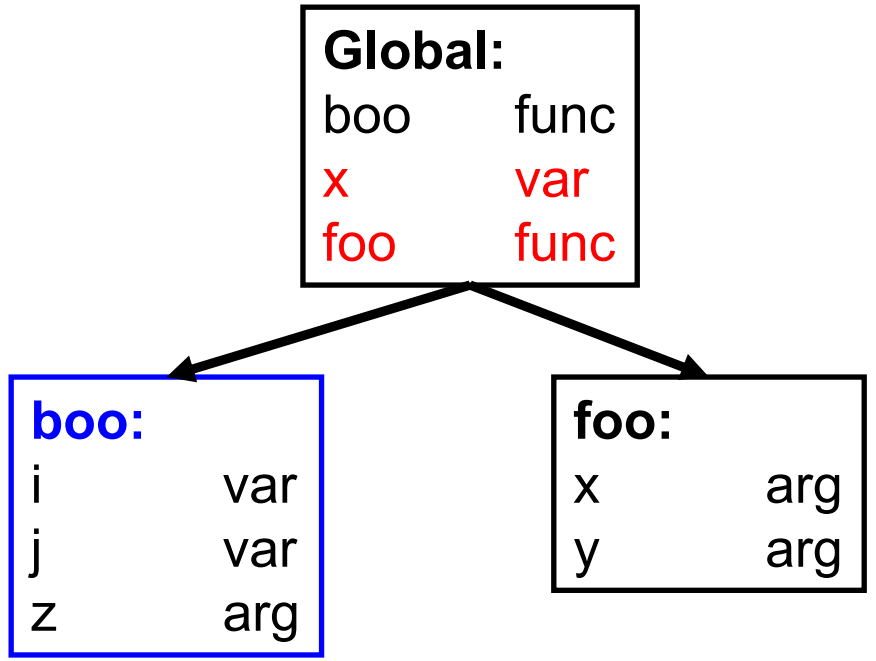
Separate body declaration

Scope Analysis and Symbol Table

- **If we use a full symbol table, we should make a separate scope**

```
int boo(int z) {
    int i, j;
    foo(i, j);
}

int x;
int foo(int x, int y) {
    …
}

…
```

For C-minus, foo should not exist in the global scope



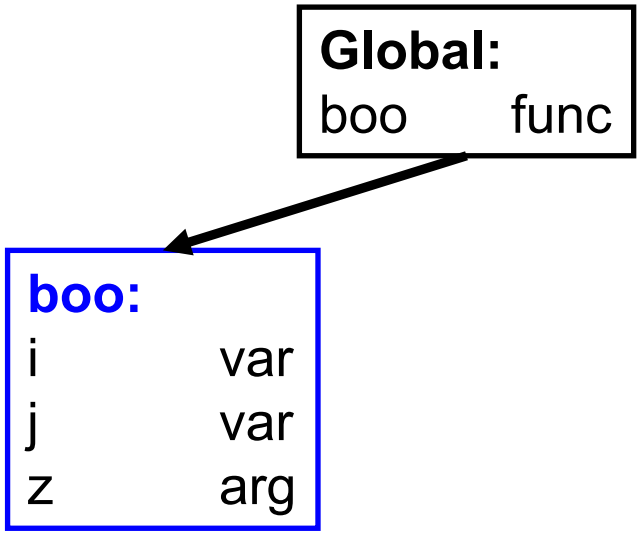
Scope Analysis and Symbol Table

- **Option #1: Generate a symbol table and do scope analysis**

- **Option #2: Simultaneously generate a symbol table and do scope analysis**

```
int boo(int z) {
    int i, j;
    foo(i, j);
}

int x;
int foo(int x, int y) {
    …
}

…
```

Search for the declaration using a partial table



Categories of Semantic Analysis

- Scopes: characterizes the declaration of identifiers and their usages

- **Types: characterizes the types of values corresponding to variables, statements, expressions, …**

Type Information

- **Type checking: set of rules which ensures the type consistency of different constructs in the program**
  - Ex) The program should not add integer and string variables

- **Type inferencing: fill missing type information**
  - Ex) If you add integer and integer, the result should be integer

- **We will use the term interchangeably**

```
int x;
float y;
double z = x + y;
```

Type Checking

- **Semantic checks to enforce the type safety of the program**

- **There are various examples**
  - Unary and binary operators (e.g. +, ==, [ ]) must receive operands of the proper type
  - Functions must be invoked with the right number and type of arguments
  - Return statements must agree with the return type
  - In assignments, assigned value must be compatible with type of variable on LHS
  - Class members accessed appropriately

Three Language Types

- **Statically typed: all or almost all type checking is done at compile time (e.g., C, Java)**
- **Dynamically typed: almost all type checking is done as part of program execution (e.g., Python, Lisp, Perl)**
- **Untyped: no type checking (e.g., machine code)**

Type Checking Processors - 1

- **Dynamic type checking incurs type checking overhead**

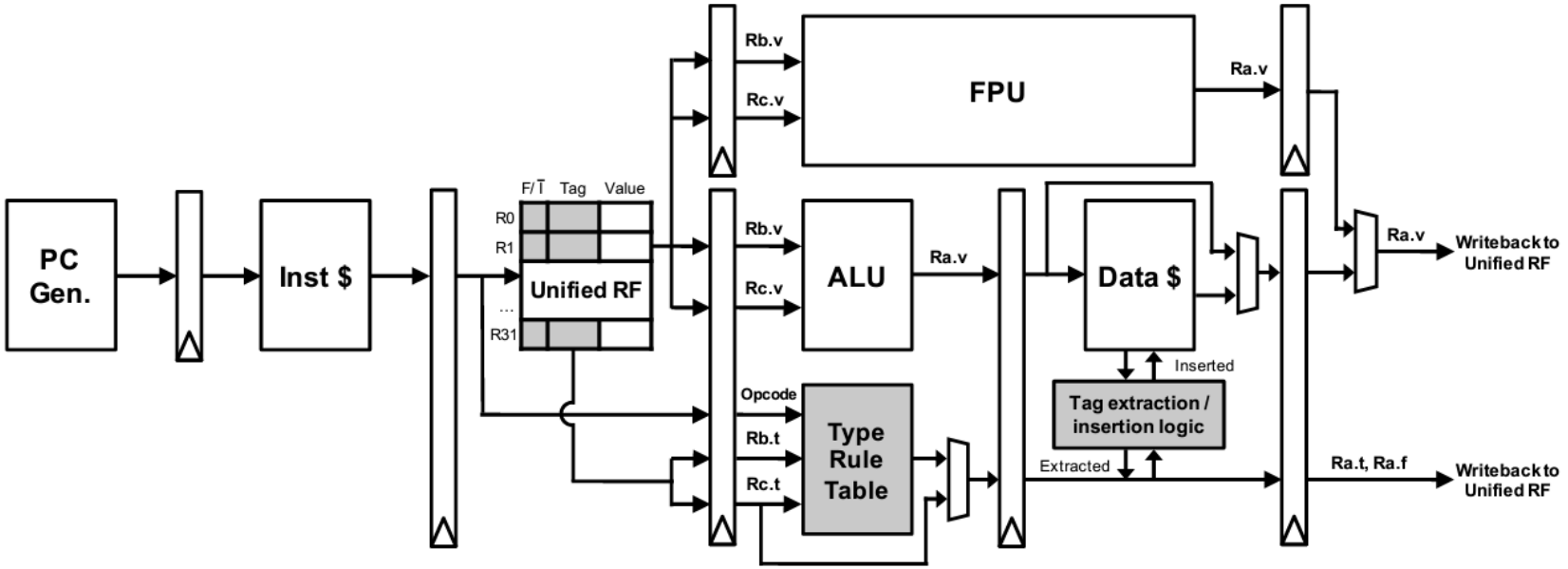
```
1   function add(x,y) return x+y end
2   add(1,2)          --(INTEGER) 3
3   add(1,2.2)        --(FLOAT) 3.2
4   add(1.1,2)        --(FLOAT) 3.1
5   add(1.1,2.2)      --(FLOAT) 3.3
6   add("1","2")      --(FLOAT) 3.0
7   add("a","b")      --error
```

```
6     case ADD:
7      // load pointers of RB and RC to rb and rc
8      Value *rb = RB(bc); Value *rc = RC(bc);
9      Number nb, nc;
10     // check if rb and rc is integer
11     // if so, calculate sum and set type
12     if(isInt(rb) && isInt(rc)){
13      type(ra) = INTEGER;
14      ival(ra) = ival(rb) + ival(rc);
15     // convert rb and rc to float
16     // if successful, caculate sum and set type
17     }else if(toNumber(rb,&nb)&&toNumber(rc,&nc)){
18      type(ra) = FLOAT;
19      fval(ra) = nb + nc;
20     }else{
21      /* handle exception cases */
22     }
23    break;
```

* Typed Architectures: Architectural Support for Lightweight Scripting

Type Checking Processors - 2

- **There is an effort to minimize the type checking overhead by changing the processor**



Super fun concept, but not the scope of this class

* Typed Architectures: Architectural Support for Lightweight Scripting

Static Type Checking

- **Does not require additional type checking instructions at runtime**
  - Greatly reduces the additional overhead for type checking
- **Guarantees that the executions are safe at compile time**
- **Modern languages require both static and dynamic type checking**
  - Union, Void pointer, …

What Are Types?

- **A type indicates a description of a set of values and a set of allowed operations on those values**
  - They are essentially a predicate on values (e.g., “int x” means that -2^31 <= x < 2^31
- **Type expressions: Describe the possible types in the program**
  - Ex) int, char*, array[], object, etc.
- **Type system: Defines types for language constructs (think about nodes in AST)**
  - Ex) expressions, statements

Type Expressions

- **Language type systems have basic types (aka: primitive types or ground types)**
  - E.g., int, char, double

- **Build type expressions using basic types:**
  - Type constructors
    - Array types
    - Structure/object types
    - Pointer types
  - Type aliases
  - Function types

Type Comparison Implementation

- **Option 1: Implement a method Equals(T1, T2)**
  - Must compare type trees of T1 and T2
  - For object-oriented languages: also need sub-typing, SubtypeOf(B, A)
    - "B" is a subtype of "A" if every function that can be invoked on an object of "A" can also be invoked on an object of type "B" (Bool is a subtype of Int)

- **Option 2: Use unique objects for each distinct type**
  - Each type expression (e.g., array[int]) resolved to same type object everywhere
  - Faster type comparison: can use ==
  - Object-oriented: check subtyping of type objects

Type Comparison Variants

- **Structural equivalence**
  - The two types are the same if they have the same structure (memory layout)
- **Name equivalence**
  - The two types are the same if they have the same type name
  - If the two types have different names, they are not the same even if they have the same memory layout

```
typedef struct cs_student {
        int number;
        char name[20];
        int grade;
} cs_stu;
```

```
typedef struct ds_student {
        int number;
        char name[20];
        int grade;
} ds_stu;
```

Type Checking Methodology

- **Type checking = verifying whether expressions are given proper types**
  - Ex) check if integer + integer results in integer

- **Option 1: Implement using syntax-directed definitions (SDD)**
  - The type checking is done during parsing

- **Option 2: First build the AST, then implement type checking by recursive traversal of the AST nodes:**

Type Checking Methodology

- **Type checking = verifying whether expressions are given proper types**
  - Ex) check if integer + integer results in integer

- **Option 1: Implement using syntax-directed definitions (SDD)**
  - The type checking is done during parsing

- **Option 2: First build the AST, then implement type checking by recursive traversal of the AST nodes:**

Syntax-Directed Definition (SDD) - 1

- **SDD associates semantic rules for the productions in the CFG**

Production Rules

```
L → E $
E → E1 + T

E → T
T → T1 * F

…
```

Semantic Rules

```
L.type =   E.type
E.type =   if (E1.type == T.type) E1.type
           else TypeCheckError(“+”)
E.type =   T.type
T.type =   if (T1.type == F.type) T1.type
           else TypeCheckError(“*”)
…
```

Syntax-Directed Definition (SDD) - 2

- **Check types based on the sematic rules associated w/ the production rules**

```
/* for a non-terminal type type */
L : E EOF
       {$$.type = $1.type;};
E : E1 + T
       {if ($1.type == INT && $3.type == INT)
              $$.type = INT;
        else
              TypeCheckError("+");};
E : E1 + T
       {if ($1.type == INT && $3.type == INT)
              $$.type = INT;
        else
              TypeCheckError("+");};
```

Type Checking Methodology

- **Type checking = verifying whether expressions are given proper types**
  - Ex) check if integer + integer results in integer

- **Option 1: Implement using syntax-directed definitions (SDD)**
  - The type checking is done during parsing

- **Option 2: First build the AST, then implement type checking by recursive traversal of the AST nodes:**

AST Traversal

- **Option 2: First build the AST, then implement type checking by recursive traversal of the AST nodes:**

```
static void traverse( TreeNode * t,
                      void (* preProc) (TreeNode *),
                      void (* postProc) (TreeNode *) )
{ if (t != NULL)
  { preProc(t);
    { int i;
      for (i=0; i < MAXCHILDREN; i++)
        traverse(t->child[i],preProc,postProc);
    }
    postProc(t);
    traverse(t->sibling,preProc,postProc);
  }
}
```

Type Checking During AST Traversal

- **Type declarations add new identifiers and their types in the symbol tables**

- **Class definitions must be added to symbol table:**

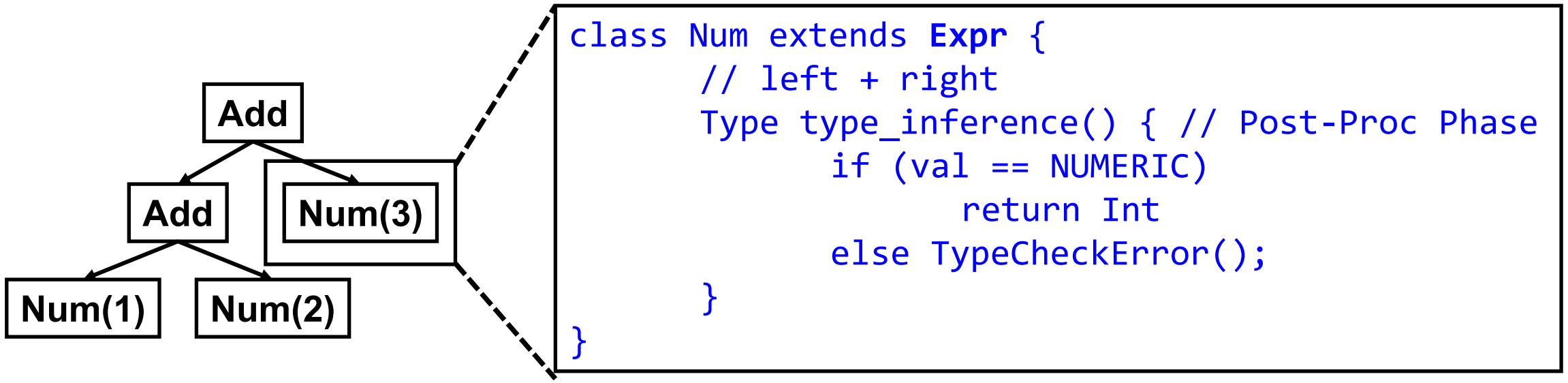
```
class_defn : CLASS ID {decls} ;
```

- **Forward references require multiple passes over AST to collect legal names**

```
class A {B b; }
class B { ... }
```

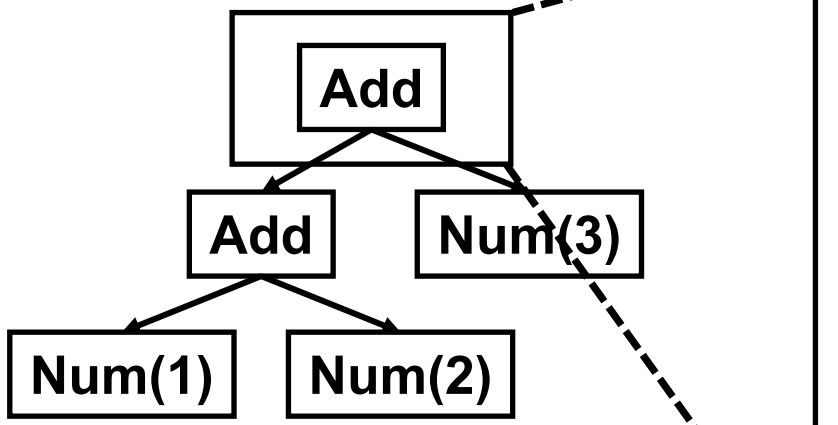
AST Traversal

- **Option 2: First build the AST, then implement type checking by recursive traversal of the AST nodes:**



AST Traversal

- **Option 2: First build the AST, then implement type checking by recursive traversal of the AST nodes:**




```
class Add extends Expr {
        // left + right
        Type type_inference() { // Post-Proc Phase
                Type t1 = left.type;
                Type t2 = right.type;
                if (t1 == Int && t2 == Int)
                        return Int
                else TypeCheckError("+");
        }
}
```

Specifying Type Rules

- **We have seen two examples of specification methods in the previous lectures**
  - Lexical analysis: regular expressions
  - Syntax analysis: context-free grammars

- **We use "logical rules of inference" as a formal specification method for type checking**

Inference Rules - 1

- **Inference rules have the form**
  - If hypotheses are true, then conclusion is true
- **Type checking is done by applying the inference rules**
  - As e1 and e2 have certain types, then e3 has a certain type
- **Important building blocks (notations)**
  - Symbol $\wedge$ indicates “and”
  - Symbol $\Longrightarrow$ indicates “if-then”
  - x:T indicates “x has type T”

Inference Rules - 2

- **If e1 has type int and e2 has type int, then e1 + e2 has type int**
  - e1 : int Λ e2 : int ⟹ e1 + e2 : int

- **Generally speaking, the above rule can have the form:**
  - hypothesis1 Λ hypothesis2 Λ … ⟹ conclusion

Inference Rules - 3

- By tradition, inference rules are written as follows:

$$\frac{\vdash \text{hypothesis1} \quad \vdash \text{hypothesis2} \quad \vdash \text{hypothesis3} \ \ldots}{\vdash \text{conclusion}} \ \text{[Rule name]}$$

- The notation $\vdash$ indicates "it is provable that …"

Soundness - 1

- **Static semantics: formal notation which describes type judgements**
  - "e : T" means e is a "sound" expression of type T
  - e is typable if there is some type T such that e : T

- **Type judgement examples:**
  - 2 : int
  - True : bool
  - 2 * (3 + 4) : int
  - "Hello" : string

Soundness - 2

- **Statements can be “expressions” (i.e., represent values)**

- **Use type judgements for statements:**
  - If (b) 2 else 3 : int // some languages utilizes if-else as an expression
    - Python: x = 2 if (b) else 3
  - X == 10 : bool
  - B = true, y = 2 : int
    - Ex) result = (a /= 2, b++); // returns preincremented b and divides a by 2
- **For non-expression statements, use a special unit type (void or empty type)**
  - S : void (means that S is a sound statement with no result type)

Sound Inference Rules

$$\frac{\text{i is an integer literal}}{\vdash \text{i} : \text{int}}\ \textbf{[Int]} \qquad \frac{\vdash \text{e1} : \text{int} \quad \vdash \text{e2} : \text{int}}{\vdash \text{e1} + \text{e2} : \text{int}}\ \textbf{[Int Add]}$$

- **An inference rule is sound if**
  - Whenever $\vdash$ e : T, e evaluates to a type T

- **We only use sound rules, but some are better than others**

$$\frac{\text{i is an integer literal}}{\vdash \text{i} : \text{object}}$$

Proof Tree

- **As in the parse tree in the syntax analysis, there exists a proof tree**

$$\dfrac{\dfrac{\text{1 is an integer literal}}{\vdash 1 : \text{int}}\,\textbf{[Int]} \qquad \dfrac{\text{2 is an integer literal}}{\vdash 2 : \text{int}}\,\textbf{[Int]}}{\vdash 1 + 2 : \text{int}}\,\textbf{[Int Add]}$$

Type Checking - 2

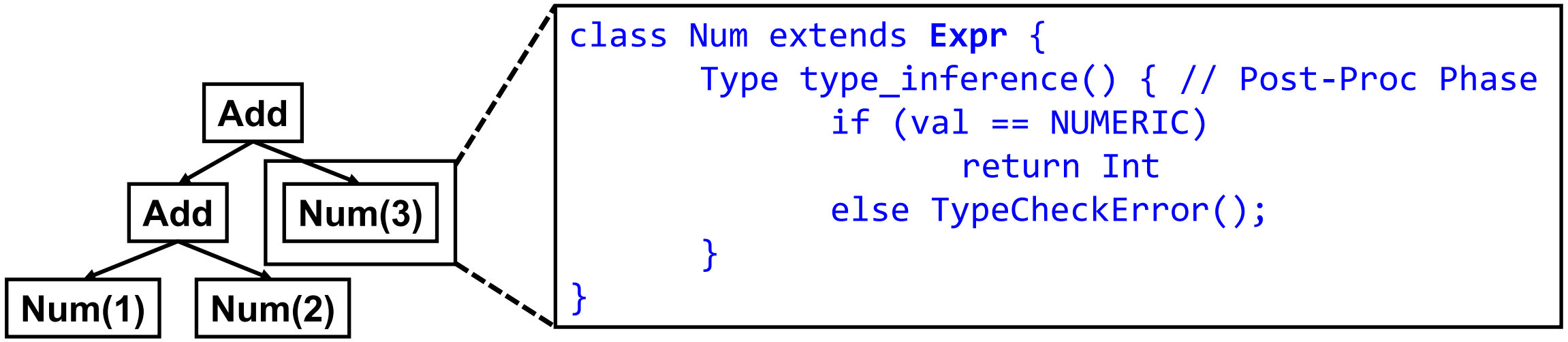
- **Type checking is a process of proving e : T**
  - Proof tree is on the structure of the AST (noticed?)
  - One type rule is used for each AST node

- **Types are checked in a bottom-up (post processing) pass over the AST node**
  - For each node e, hypotheses are the proofs of type e's subexpression and the conclusion is the type of e

$$\frac{\vdash 1 : \text{int} \quad \textbf{sub-e1} \qquad \vdash 2 : \text{int} \quad \textbf{sub-e2}}{\vdash 1 + 2 : \text{int} \quad \textbf{e}} \; \textbf{[Int Add]}$$

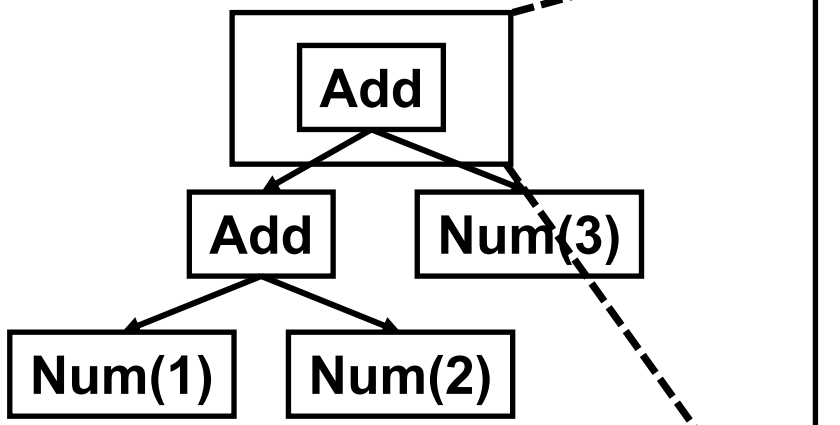
AST Traversal

- **Option 2: First build the AST, then implement type checking by recursive traversal of the AST nodes:**



AST Traversal

- **Option 2: First build the AST, then implement type checking by recursive traversal of the AST nodes:**




```
class Add extends Expr {
        // left + right
        Type type_inference() { // Post-Proc Phase
                Type t1 = left.type;
                Type t2 = right.type;
                if (t1 == Int && t2 == Int)
                        return Int
                else TypeCheckError("+");
        }
}
```

More on Inference Rules - 1

$$\frac{s \text{ is a string literal}}{\vdash s : string}\ [String]$$

$$\frac{}{\vdash false : bool}\ [False]$$

$$\frac{}{\vdash new\ T : T}\ [New]$$

$$\frac{\vdash e : bool}{\vdash !e : bool}\ [Not]$$

Class Exercise

- **Define the inference rule for the while statement**

$$\frac{\text{FILLME}}{\vdash \text{while (e1) \{ e2 \} : FILLME}}\ [\text{while}]$$

Class Exercise

- **Define the inference rule for the while statement**

$$\frac{\vdash e1 : Bool \qquad \vdash e2 : T}{\vdash \text{while } (e1) \{ e2 \} : Void} \text{ [while]}$$

More on Inference Rules - 2

- **We "may" use if-then-else statement to return an interesting typed value**

$$\frac{\vdash e1 : bool \quad \vdash e2: T \quad \vdash e3: T}{\vdash \text{if } e1 \text{ then } e2 \text{ else } e3 : T} \text{ [If-then-else]}$$

- **What about if-then statement?**
  - You should discourage programmers to rely on [If-then] to return a type

$$\frac{\vdash e1 : bool \quad \vdash e2: T}{\vdash \text{if } e1 \text{ then } e2 : void} \text{ [If-then]}$$

Class Exercise

There are following typing rules

$$\frac{}{\vdash 0 : \text{int}}$$

$$\frac{}{\vdash \text{true} : \text{bool}} \qquad \frac{}{\vdash \text{false} : \text{bool}}$$

$$\frac{\vdash e1 : \text{bool} \quad e2 : T \quad e3 : T}{\vdash \text{if } e1 \text{ then } e2 \text{ else } e3 : T}\ \textbf{[if-then-else]}$$

$$\frac{\vdash e1 : \text{bool}}{\vdash \text{succ } e1 : \text{int}}\ \textbf{[succ]} \qquad \frac{\vdash e1 : \text{int}}{\vdash \text{pred } e1 : \text{int}}\ \textbf{[pred]}$$

$$\frac{\vdash e1 : \text{int}}{\vdash \text{iszero } e1 : \text{bool}}\ \textbf{[iszero]}$$

There are following CFGs

```
exp ➔ true
      | false
      | if exp then exp else exp
      | 0
      | succ exp
      | pred exp
      | iszero exp
```

Proof Tree for:

(1) if iszero 0 then 0 else pred 0 : int
(2) pred succ iszero pred 0 : int

Class Exercise

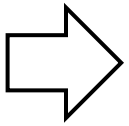
$$\dfrac{\dfrac{\dfrac{}{\vdash 0 : \text{int}}}{\vdash \text{iszero } 0 : \text{bool}} \qquad \dfrac{}{\vdash 0 : \text{int}} \qquad \dfrac{\dfrac{}{\vdash 0 : \text{int}}}{\vdash \text{pred } 0 : \text{int}}}{\vdash \text{if iszero } 0 \text{ then } 0 \text{ else pred } 0 : \text{int}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash 0 : \text{int}}}{\vdash \text{pred } 0 : \text{int}}}{\vdash \text{iszero pred } 0 : \text{bool}}}{\vdash \text{succ iszero pred } 0 : \text{int}}}{\vdash \text{pred succ iszero pred } 0 : \text{int}}$$

Problems of the Inference Rules

- **How should we define the type of a variable reference?**
  - The reference does not provide enough information to give “x” a type

```
int x;
…
x;
…
```



$$\frac{x \text{ is a variable}}{\vdash x : ?}\ [\text{Var}]$$

We should provide more information to the rules!

Type Environment - 1

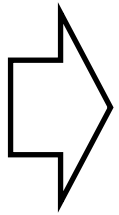
- **A type environment gives types for "free variables"**
  - A type environment is a function from Object identifiers to types
  - A variable is free in an expression is it is not defined within the expression
    - For expression "x", x is free
    - For expression "x+y", x and y are free
    - For expression "int z = x", only x is free

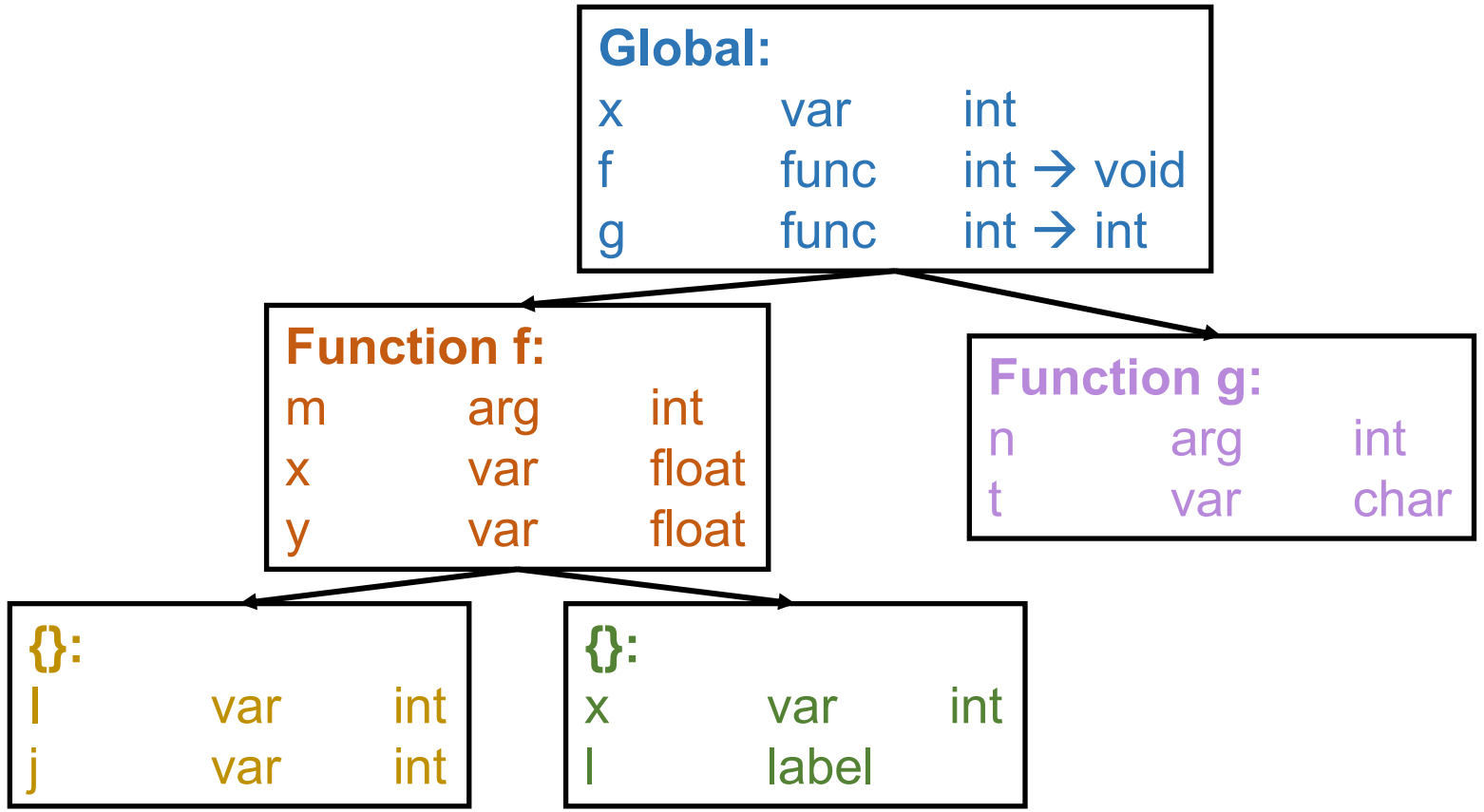
We need information for these variables for type checking

- **Type environment notation: O ⊢ e : T**
  - Under the assumption that variables have the types given by O, it is provable that the expression e has type T
  - O is essentially a "symbol table"

Recall: Symbol Tables

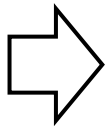
```
int x;

void f (int m) {
    float x, y; …;
    {int i,j; …;}
    {int x,k; …;}
}

int g (int n) {
    char t;
    …;
}
```





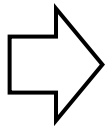
Type Environment Example

```
int x;
int y;
…
z = x + y;
…
…
…
x;
```



```
O(x) = int
O(y) = int
```

$$\frac{O \vdash x : int \quad O \vdash y : int}{O \vdash x + y : int}$$ **[Int Add]**



```
O(x) = int
```

$$\frac{O(x) = int}{O \vdash x : int}$$ **[Var]**

Complex Examples: Declaration

- **We utilize the term O[T/x] to describe context with declarations**
  - O[T/x](x) = T and O[T/x](y) = O(y) (if x != y)
  - This is essentially the same as modifying or adding an entry in a symbol table

```
for (T1 i = 0;;)
{
      exp // subpression
}
```

$$\frac{O[T1/i] \vdash exp : T2}{O \vdash for(i : T1)\ \{exp : T2\ \}}$$

Complex Examples: Class Attributes

- **We utilize the term Oc to describe class-wide scope**
  - Oc(x) = T for all attributes x:T in class C

```
Class C {
      T1 x1;
      …
}
```

$$\frac{Oc(x1) = T1 \quad Oc \vdash exp : T1}{Oc \vdash x1 = exp : T1}$$

Complex Examples: Class Methods

- **We utilize a separate scope M, to describe functions**
  - M(C, f) = (T1, T2, T3, … Tn, Tn+1) indicates that in class C, there is a method f where f(x1 : T1, x2 : T2, …, xn : Tn) return Tn+1

```
T1 e1; T2 e2; … Tn en;
T0 e0 = T0();
e0.func(e1,e2, …, en)
```

$$\frac{O, M \vdash e0:T0 \quad \ldots \quad O, M \vdash en:Tn \qquad M(T0, f) = (T1, \ldots Tn, Tn+1)}{O, M \vdash e0.func(e1,e2, \ldots,en) : Tn+1}$$

Subtyping - 1

- **There are relation on classes**
  - X ≤ X
  - X ≤ Y if X inherits from Y
  - if X ≤ Y and Y ≤ Z, then X ≤ Z
  - Ex) float ≤ double, char ≤ int

```
for (T1 x = y;…)
{
      exp // subpression
}
```

More permissive!

$$\frac{O \vdash y : T0 \quad T0 \leq T1 \quad O[T1/x] \vdash exp : T2}{O \vdash for(x : T1 = y)\ \{exp : T2\ \}}$$

Subtyping and Type Conversion

- **If a class B is a subtype of class A, all the functions that can be applied to class A can also be applied to class B**

- **We can implicitly convert a subtype to a supertype**
  - Consider char ≤ int ≤ float ≤ double, we can implicitly convert char to int, int to float, float to double
  - However, we need an explicit method (explicit conversion) when converting a supertype to a subtype

Subtyping - 2

- **Subtyping enables more complex and permissive typing**
  - If e0 then e1 else e2 (where e1 and e2 has different types)

- **We can find the smallest supertype of e1 and e2 to describe such complex expressions**
  - lub(X, Y) describes the least upper-bound of X and Y (i.e., Z) where:
    - $X \leq Z$ and $Y \leq Z$
    - Given an arbitrary Z' where $X \leq Z'$ and $Y \leq Z'$, $Z \leq Z'$

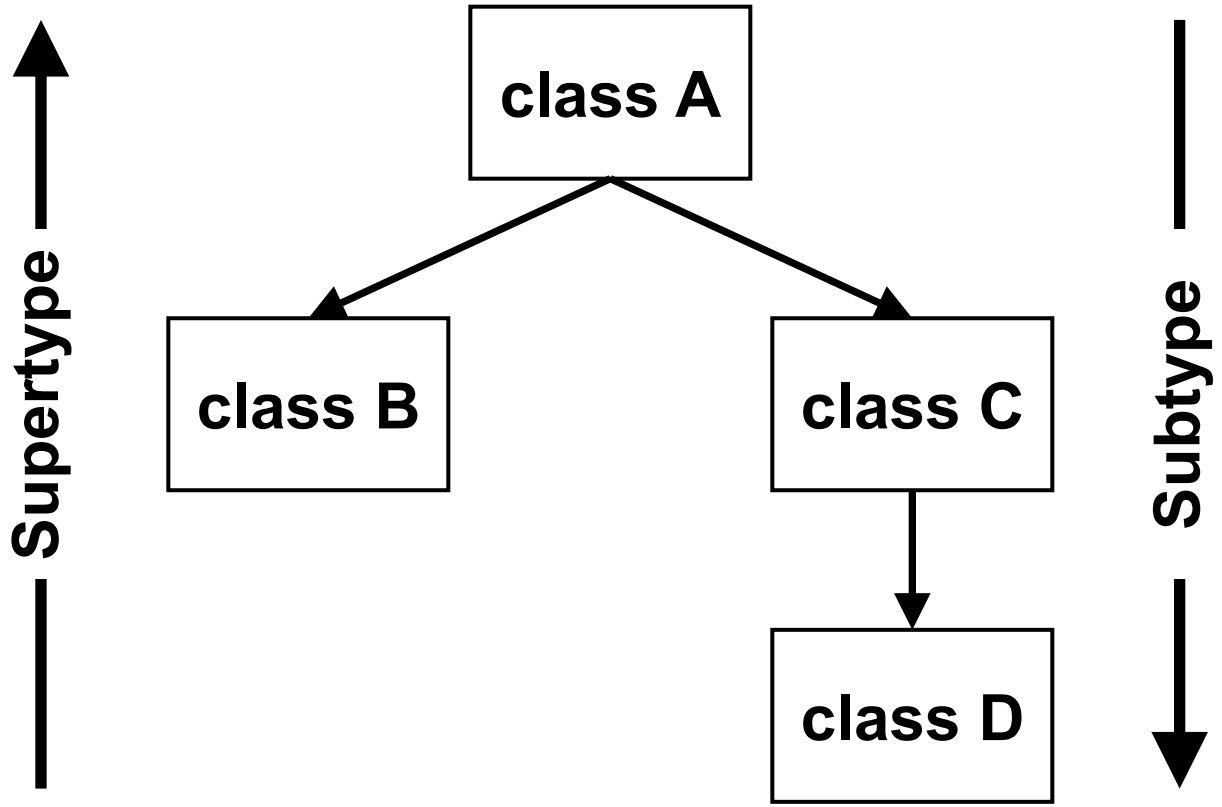
$$\frac{O \vdash e0 : bool \quad O \vdash e1 : T1 \quad O \vdash e2 : T2}{O \vdash \text{If e0 then e1 else e2} : lub(T1, T2)}$$

Inheritance and Subtyping

- **A common way to enable subtyping is via inheritance (given that there is no overriding)**

```
// Assume no overriding
class A {…}
class B inherits A {…}
class C inherits A {…}
class D inherits C {…}
```

Ex) lub(B, D) = A



Static Scoping vs. Dynamic Scoping - 1

- Let’s discuss another fun topic in scope analysis
- Static scoping: we can statically generate a symbol table, and use the table to infer where the variable was declared
- Dynamic scoping: the variable scope changes dynamically with function call

Static Scoping vs. Dynamic Scoping - 2

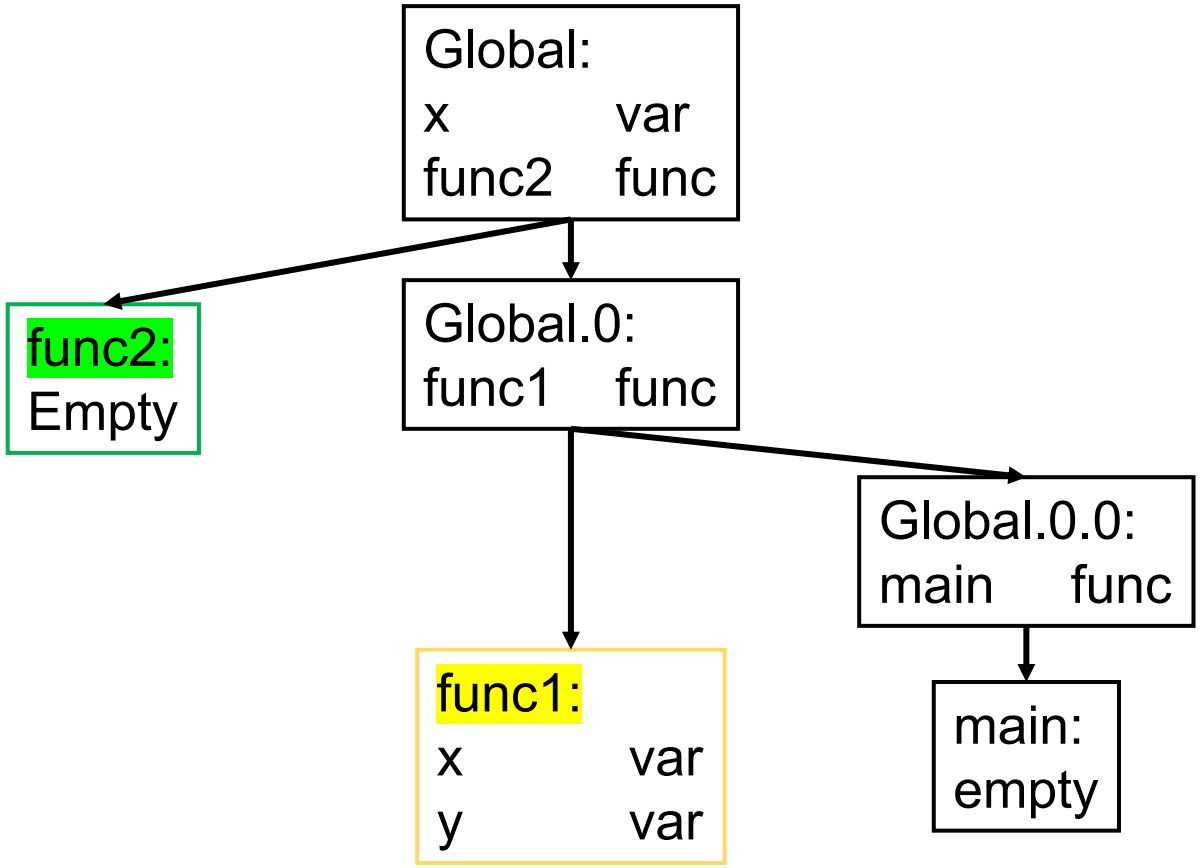
```
int x = 50;
int func2 () {
  return x;
}

int func1 () {
  int x = 10;
  int y = func2();
  return y;
}

int main () {
  func1();
}
```

What does this function return?

In Static Scoping …

- **The usage is defined at compile time using a symbol table**

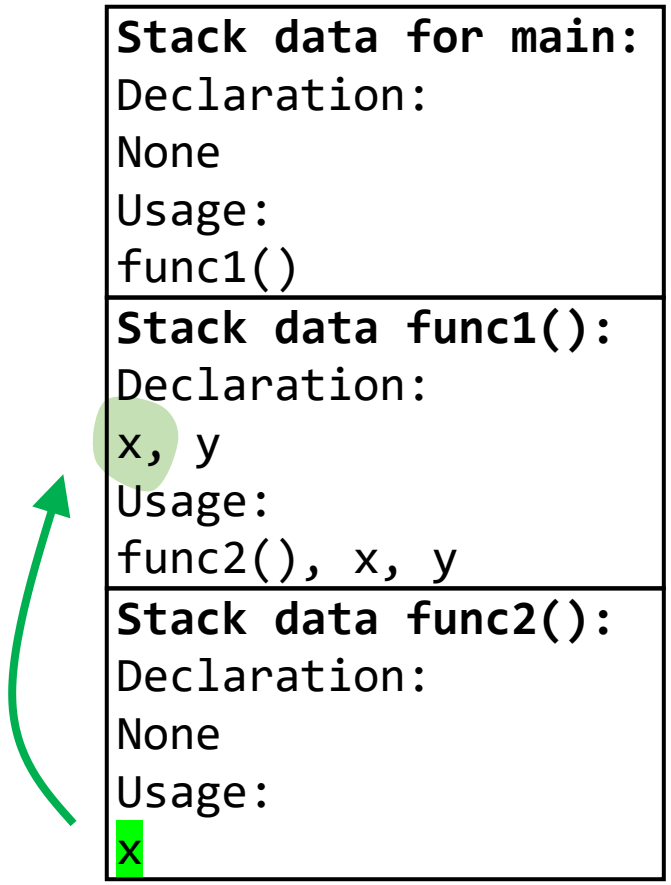
```
int x = 50;
int func2 () {
  return x;
}

int func1 () {
  int x = 10;
  int y = func2();
  return y;
}

int main () {
  func1();
}
```



In Dynamic Scoping …

- **The usage is based on the call stack (nearest declaration)**

```
int x = 50;
int func2 () {
  return x;
}

int func1 () {
  int x = 10;
  int y = func2();
  return y;
}

int main () {
  func1();
}
```



Perl: Static + Dynamic Typing

- **Perl supports both static and dynamic typing**
  - local: the variable is visible in the call stack
  - my: the variable is used only within the scope

```
$x = 50;
sub func2 () {
  return $x;
}

sub func1 () {
  local $x = 10;
  my $y = func2();
  return $y;
}

func1();
```

```
$x = 50;
sub func2 () {
  return $x;
}

sub func1 () {
  my $x = 10;
  my $y = func2();
  return $y;
}

func1();
```