Semantic Analysis

TEACHING ASSISTANT: DAVID TRABISH

Semantic Analysis

Perform various checks:

- Type checking
  - 1 + “1”
- Scopes
  - Undefined variables
- Other
  - Division by zero
  - Visibility semantics in classes (public, private, …)

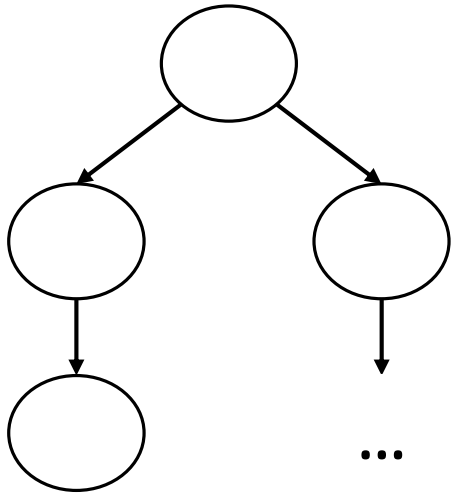
Visitor Design Pattern

Perform computations over tree-like data structures

$visit(node)$:

// do something with node

$r_1 = visit(node.child_1)$
$r_2 = visit(node.child_2)$
…
// do something with $r_1, r_2, \ldots$



Visitor Design Pattern: Example

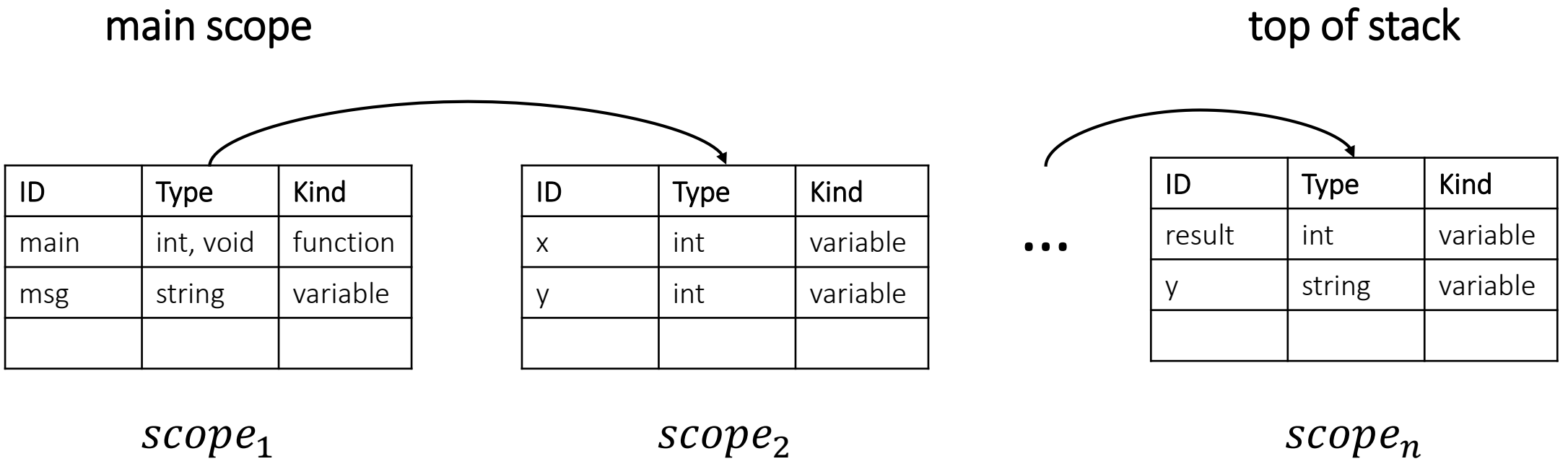
Printing the AST

$visit(node)$:
    $print(node)$
    $for\ child\ in\ node.children$:
        $visit(child)$

Symbol Table

- Stack of scopes
- Each scope contains information about identifiers
  - Name
  - Type (int, string, …)
  - Kind (variable, function, method, …)

Symbol Table



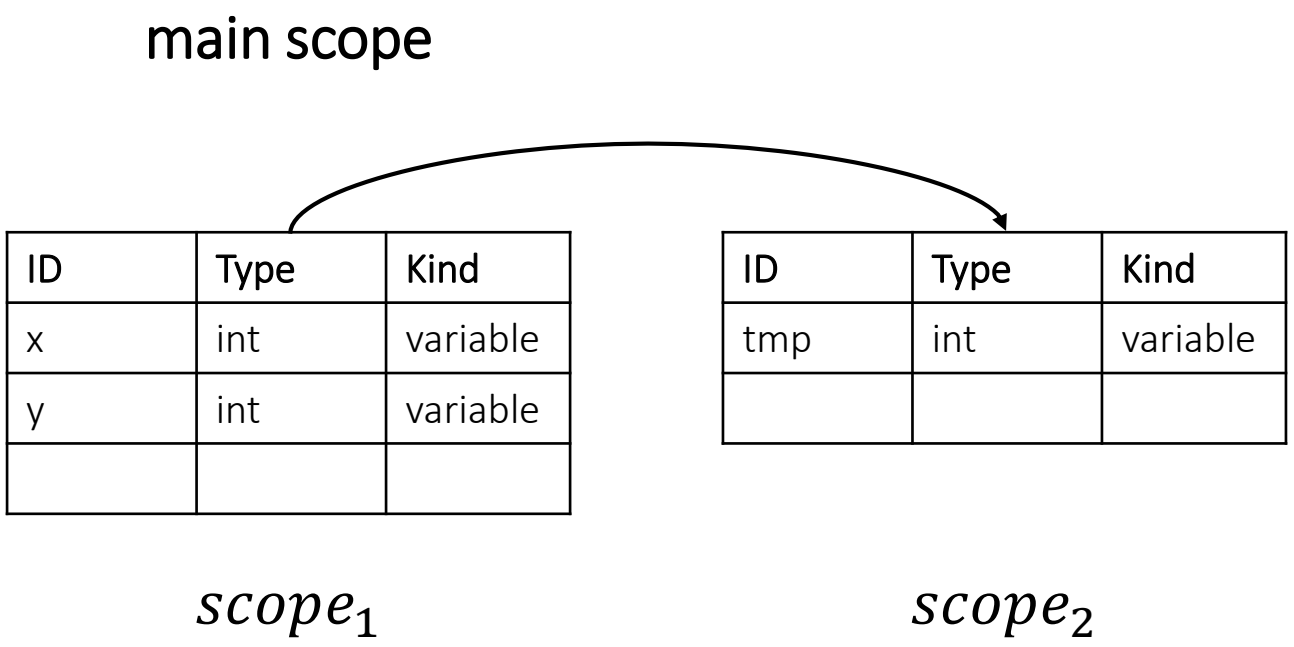
Symbol Table Operations

- **Insert** symbol
- **Lookup** symbol
- **Enter** scope
- **Exit** scope

Symbol Table: Insert

Example:

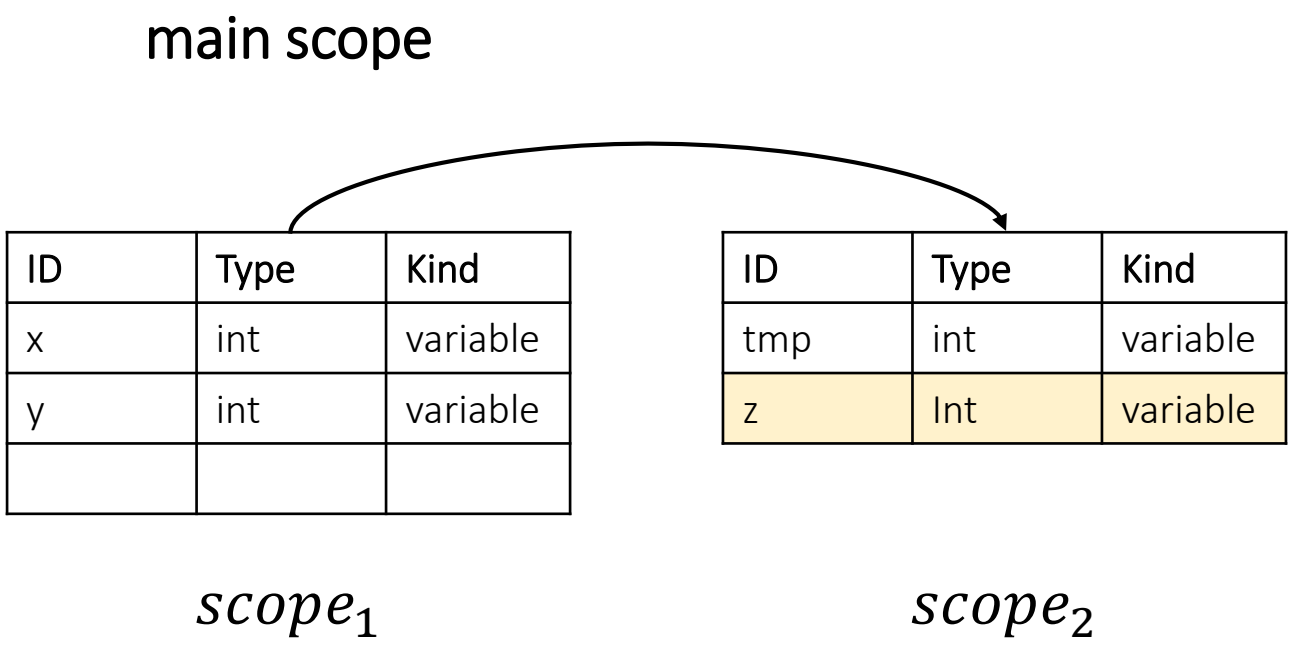
- Insert(z, int, variable)



Symbol Table: Insert

Example:

- Insert(z, int, variable)

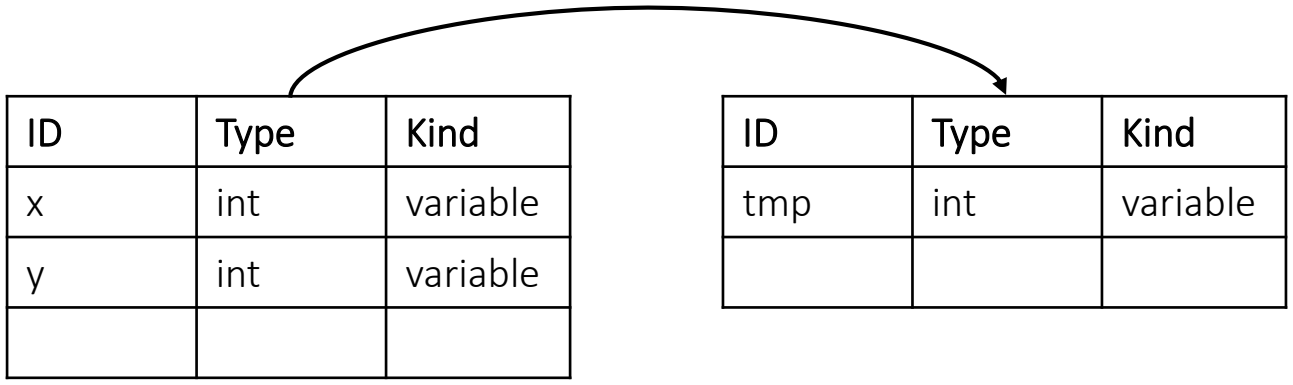


Symbol Table: Lookup

Example:

- Lookup(y)
  - Start from the top of the stack, return **first** match

main scope



ID	Type	Kind
x	int	variable
y	int	variable

$scope_1$

ID	Type	Kind
tmp	int	variable

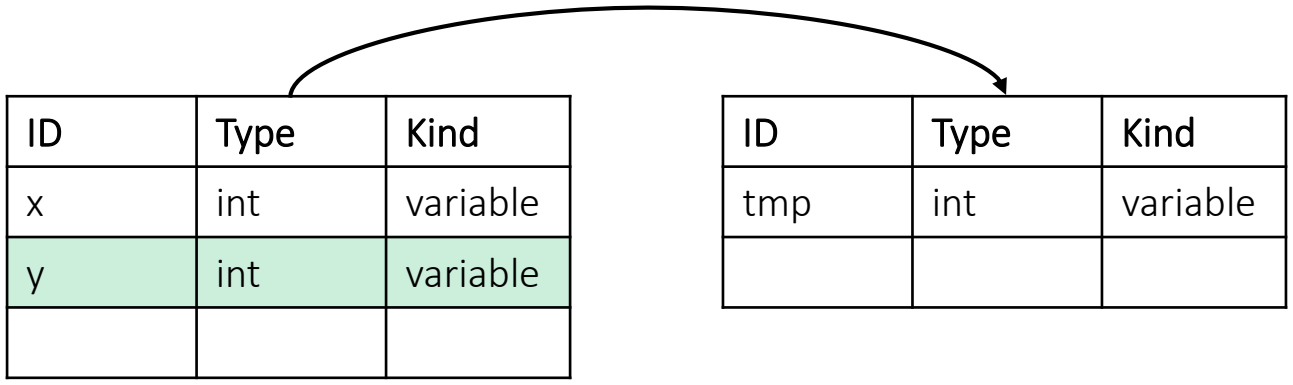
$scope_2$

Symbol Table: Lookup

Example:

- Lookup(y)
  - Start from the top of the stack, return **first** match

main scope



ID	Type	Kind
x	int	variable
y	int	variable

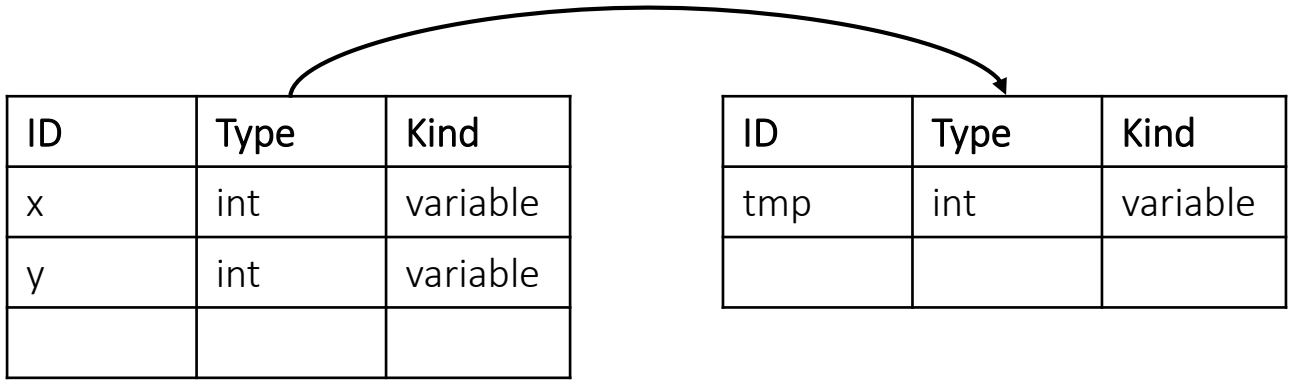
$scope_1$

ID	Type	Kind
tmp	int	variable

$scope_2$

Symbol Table: Enter

main scope



ID	Type	Kind
x	int	variable
y	int	variable

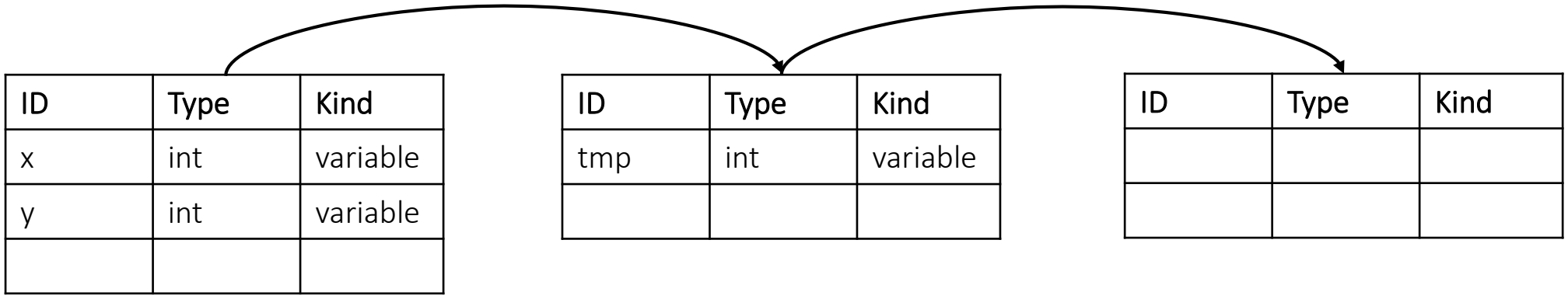
ID	Type	Kind
tmp	int	variable

$scope_1$

$scope_2$

Symbol Table: Enter

main scope



ID	Type	Kind
x	int	variable
y	int	variable

$scope_1$

ID	Type	Kind
tmp	int	variable

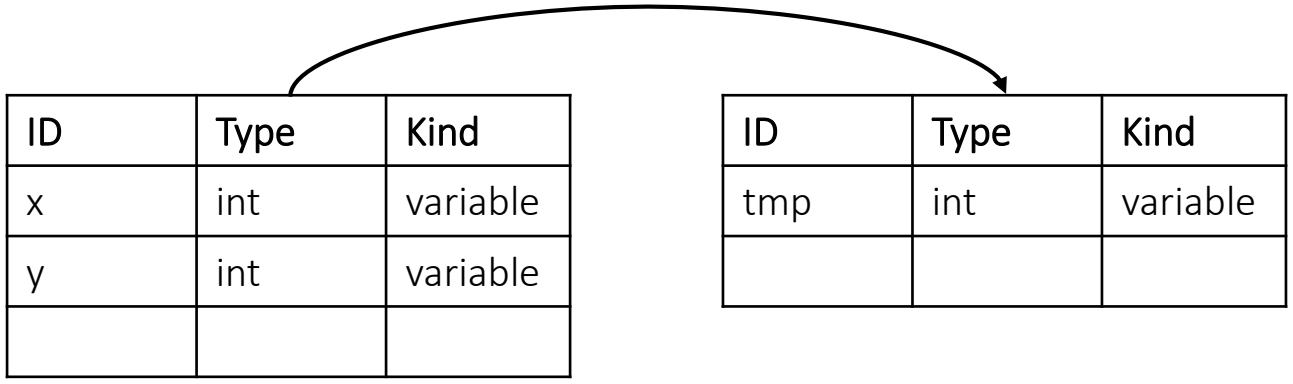
$scope_2$

ID	Type	Kind

$scope_3$

Symbol Table: Exit

main scope



ID	Type	Kind
x	int	variable
y	int	variable

ID	Type	Kind
tmp	int	variable

$scope_1$ $scope_2$

Symbol Table: Exit

main scope

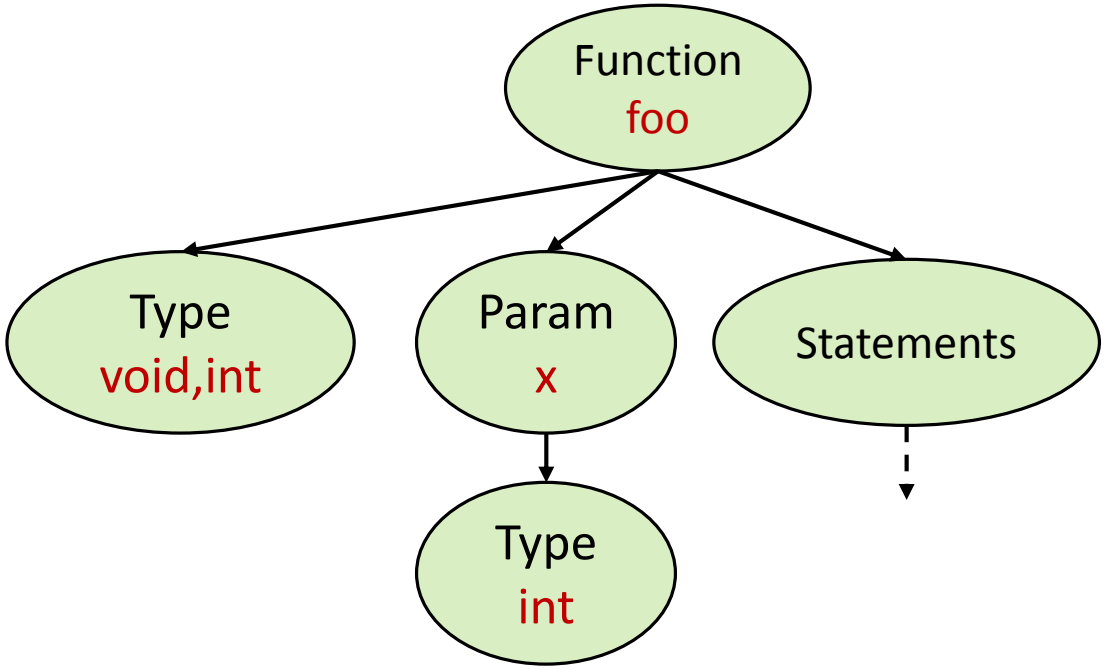
ID	Type	Kind
x	int	variable
y	int	variable

$scope_1$

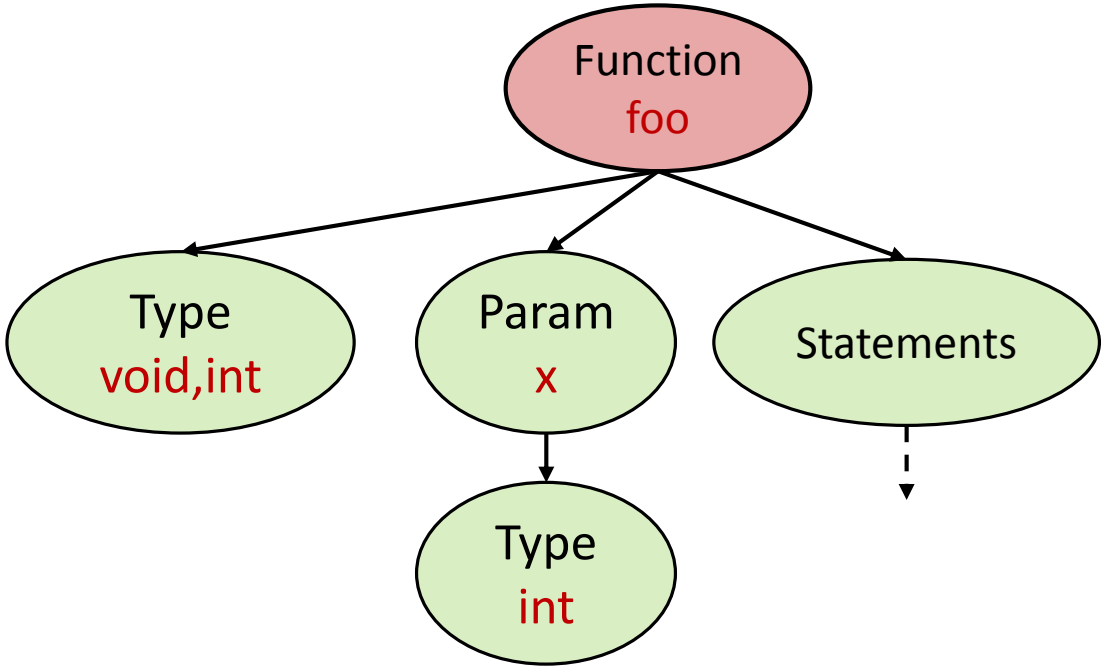
Symbol Table Construction

- Identifier declaration
  - Insert
- Identifier reference
  - Lookup
- When visiting a new block
  - Enter
- When leaving a block
  - Exit

```
void foo(int x) {
  int a = 1;
  int b = 2;
  if (x) {
    string a = "abc";
  }
}
```

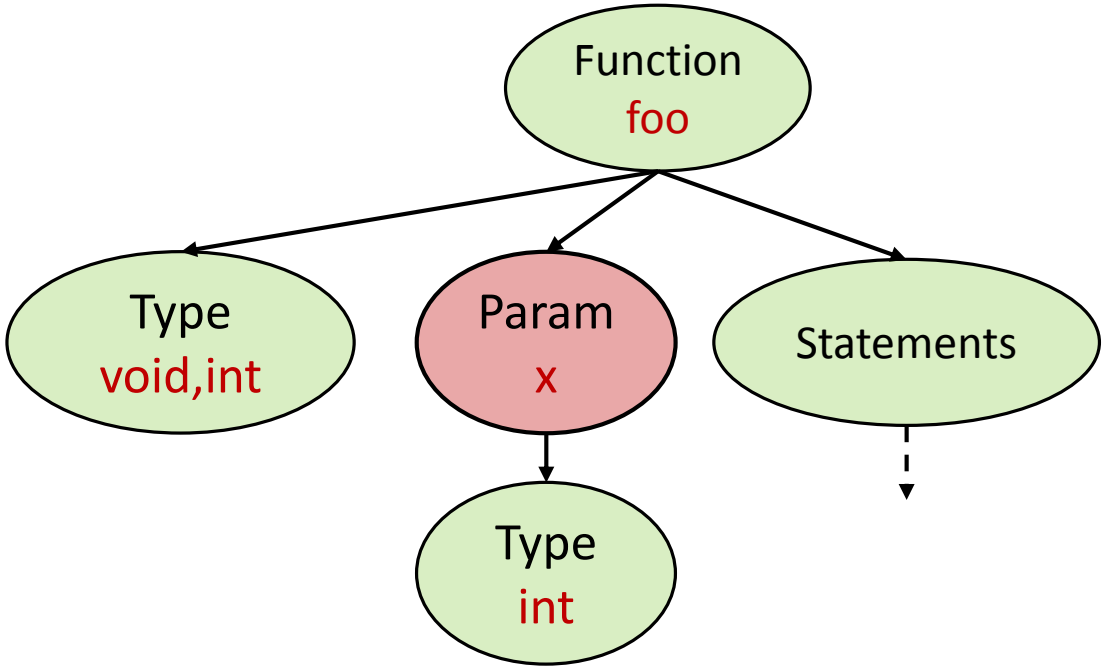


```
void foo(int x) {
  int a = 1;
  int b = 2;
  if (x) {
    string a = "abc";
  }
}
```



ID	Type	Kind
foo	void,int	function

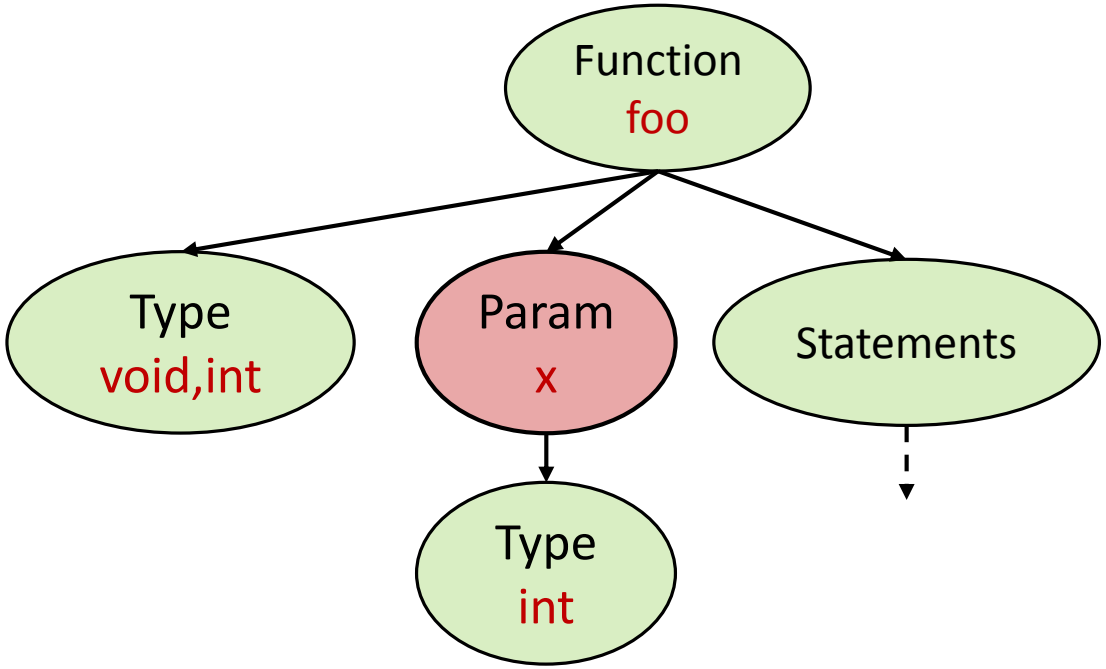
```
void foo(int x) {
  int a = 1;
  int b = 2;
  if (x) {
    string a = "abc";
  }
}
```



ID	Type	Kind
foo	void,int	function

ID	Type	Kind

```
void foo(int x) {
  int a = 1;
  int b = 2;
  if (x) {
    string a = "abc";
  }
}
```



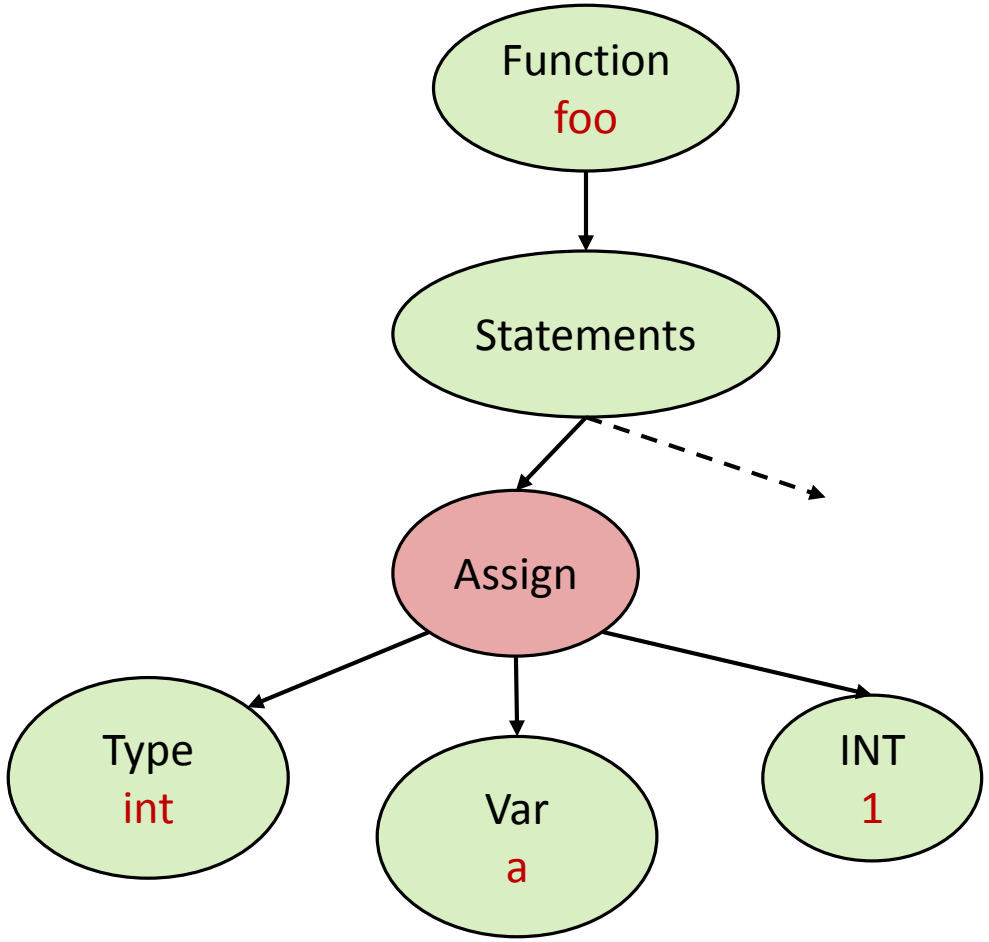
ID	Type	Kind
foo	void,int	function

ID	Type	Kind
x	int	variable

```
void foo(int x) {
  int a = 1;
  int b = 2;
  if (x) {
    string a = "abc";
  }
}
```

ID	Type	Kind
foo	void,int	function

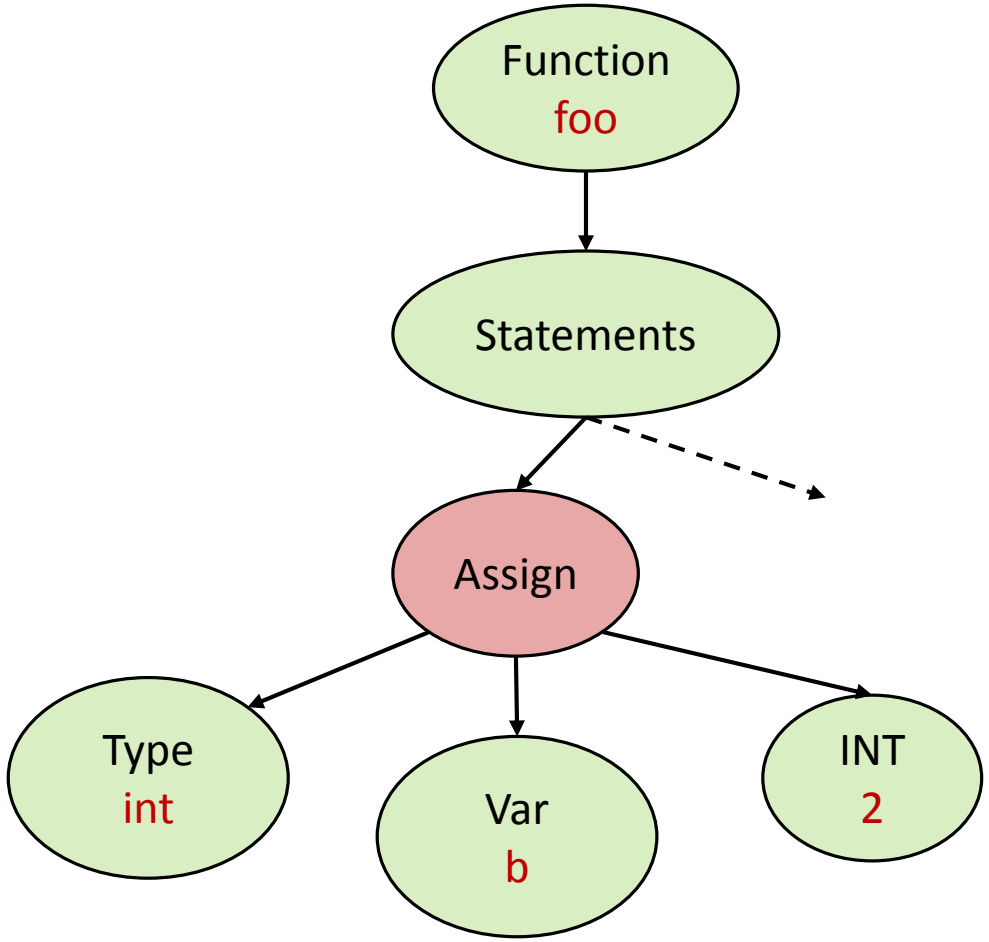
ID	Type	Kind
x	int	variable
a	int	variable



```
void foo(int x) {
  int a = 1;
  int b = 2;
  if (x) {
    string a = "abc";
  }
}
```

ID	Type	Kind
foo	void,int	function

ID	Type	Kind
x	int	variable
a	int	variable
b	int	variable

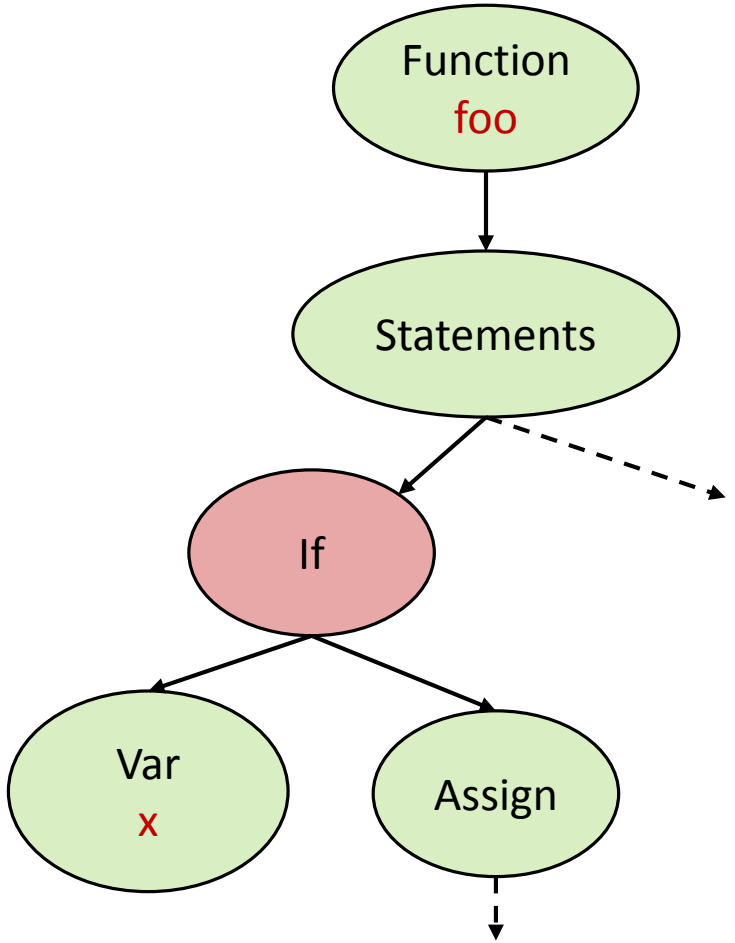


```
void foo(int x) {
  int a = 1;
  int b = 2;
  if (x) {
    string a = "abc";
  }
}
```

ID	Type	Kind
foo	void,int	function

ID	Type	Kind
x	int	variable
a	int	variable
b	int	variable

ID	Type	Kind

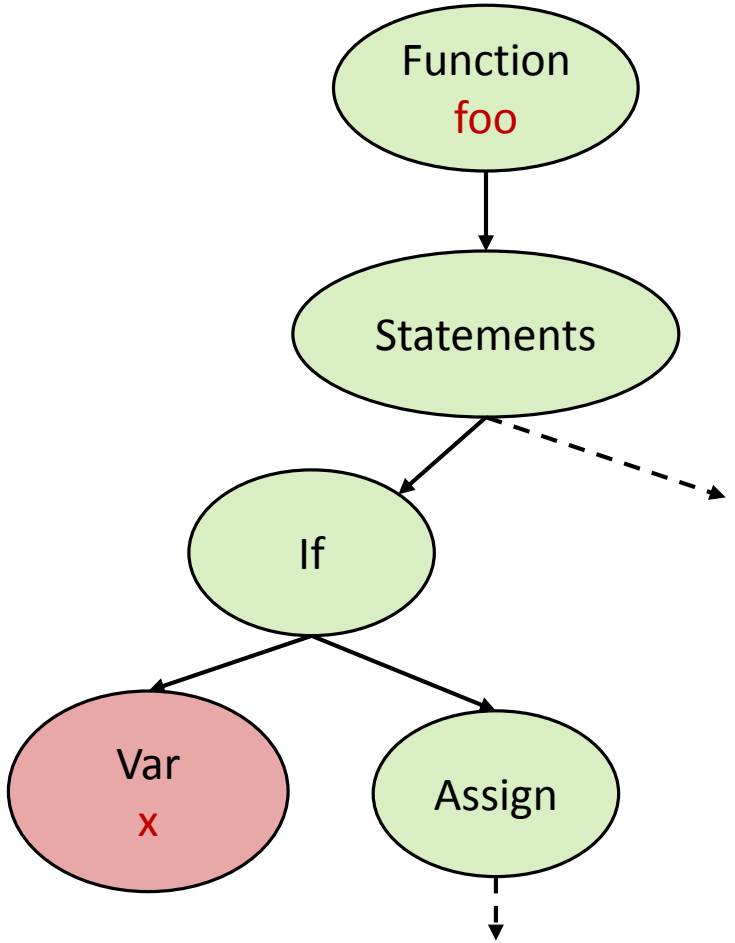


```
void foo(int x) {
  int a = 1;
  int b = 2;
  if (x) {
    string a = "abc";
  }
}
```

ID	Type	Kind
foo	void,int	function

ID	Type	Kind
x	int	variable
a	int	variable
b	int	variable

ID	Type	Kind

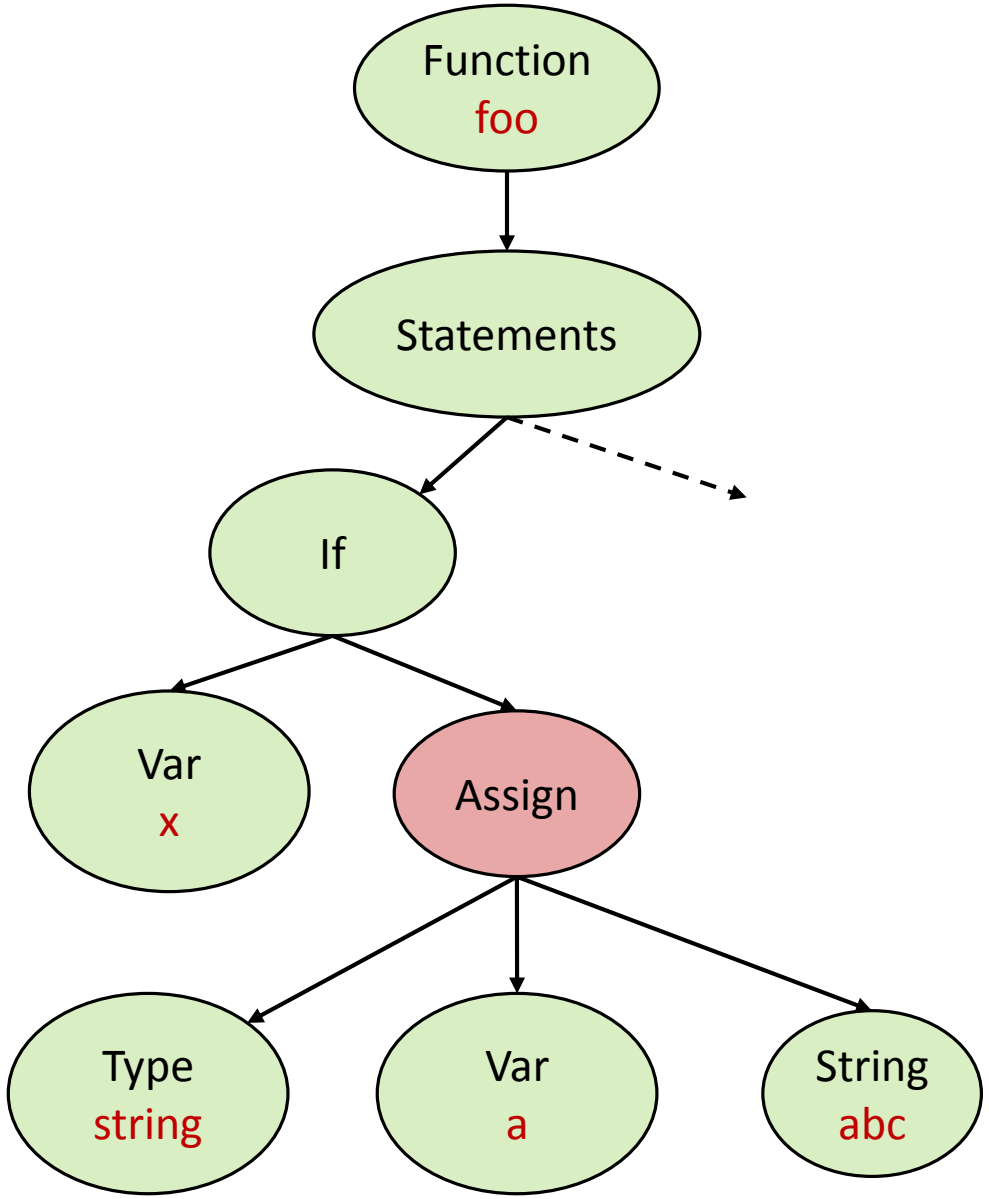


```
void foo(int x) {
  int a = 1;
  int b = 2;
  if (x) {
    string a = "abc";
  }
}
```

ID	Type	Kind
foo	void,int	function

ID	Type	Kind
x	int	variable
a	int	variable
b	int	variable

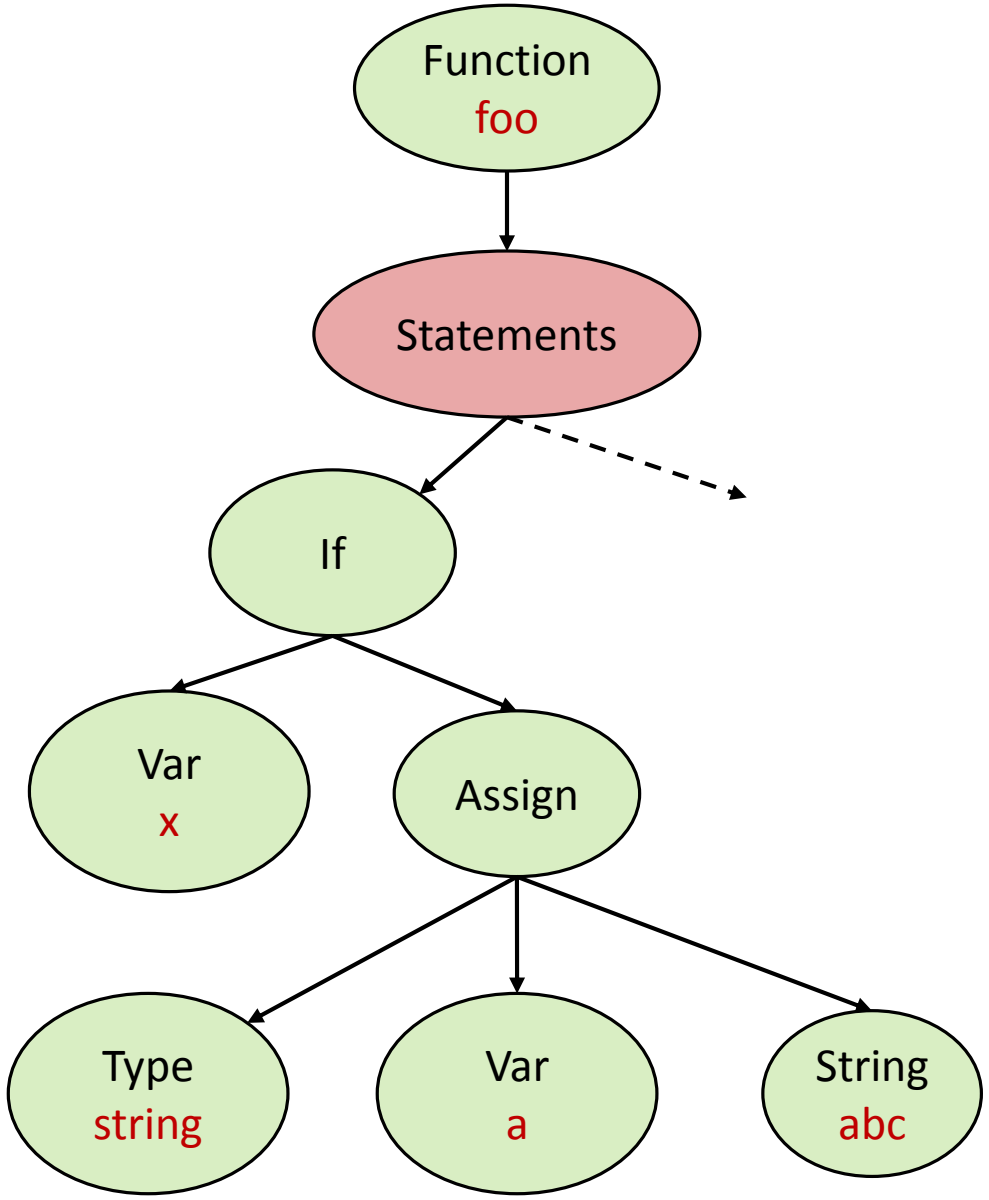
ID	Type	Kind
a	string	variable



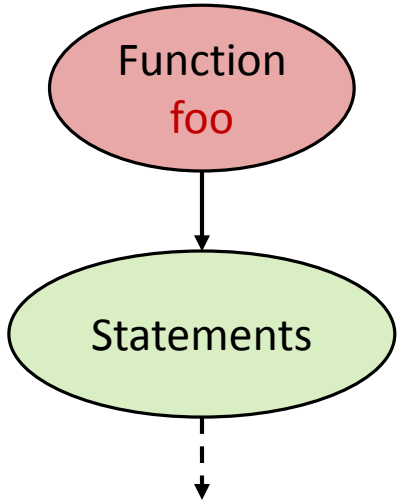
```
void foo(int x) {
  int a = 1;
  int b = 2;
  if (x) {
    string a = "abc";
  }
}
```

ID	Type	Kind
foo	void,int	function

ID	Type	Kind
x	int	variable
a	int	variable
b	int	variable



```
void foo(int x) {
   int a = 1;
   int b = 2;
   if (x) {
      string a = “abc”;
   }
}
```



ID	Type	Kind
foo	void,int	function

```
void foo(int x) {
  int a = 1;
  int b = 2;
  if (x) {
    string a = “abc”;
  }
}
```

Type Checking

Goals:

- Type correctness of expressions
- Compute type of expressions

Performed using:

- AST visitor
- Symbol table

Type Checking

Basic algorithm:

$visit(node)$:

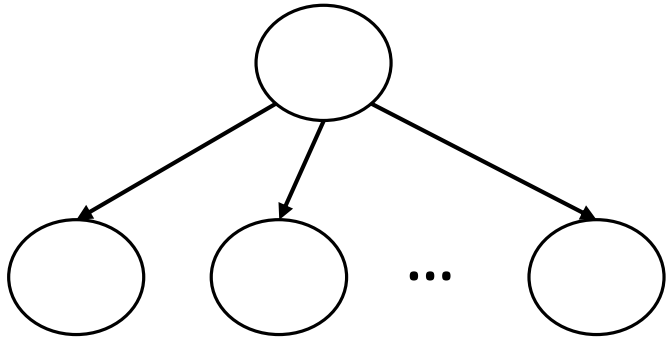
$\mathrm{t}_1 = visit(node.child_1)$

…

$\mathrm{t}_\mathrm{n} = visit(node.child_n)$

$return \underbrace{compute\_type(t_1, \ldots, t_n)}$

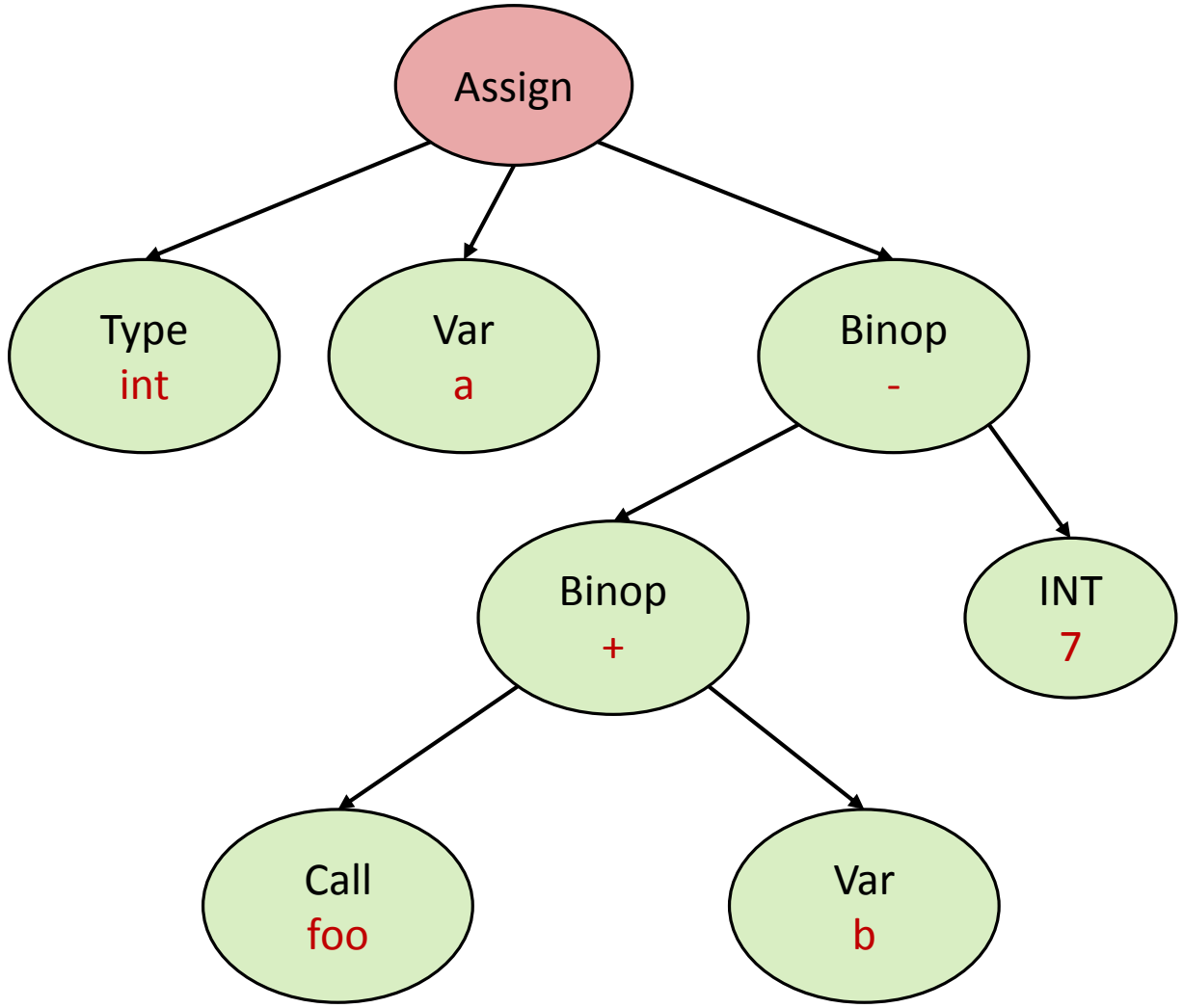
node specific



```
int a = foo() + b - 7;
...
```

ID	Type	Kind
foo	int,void	function

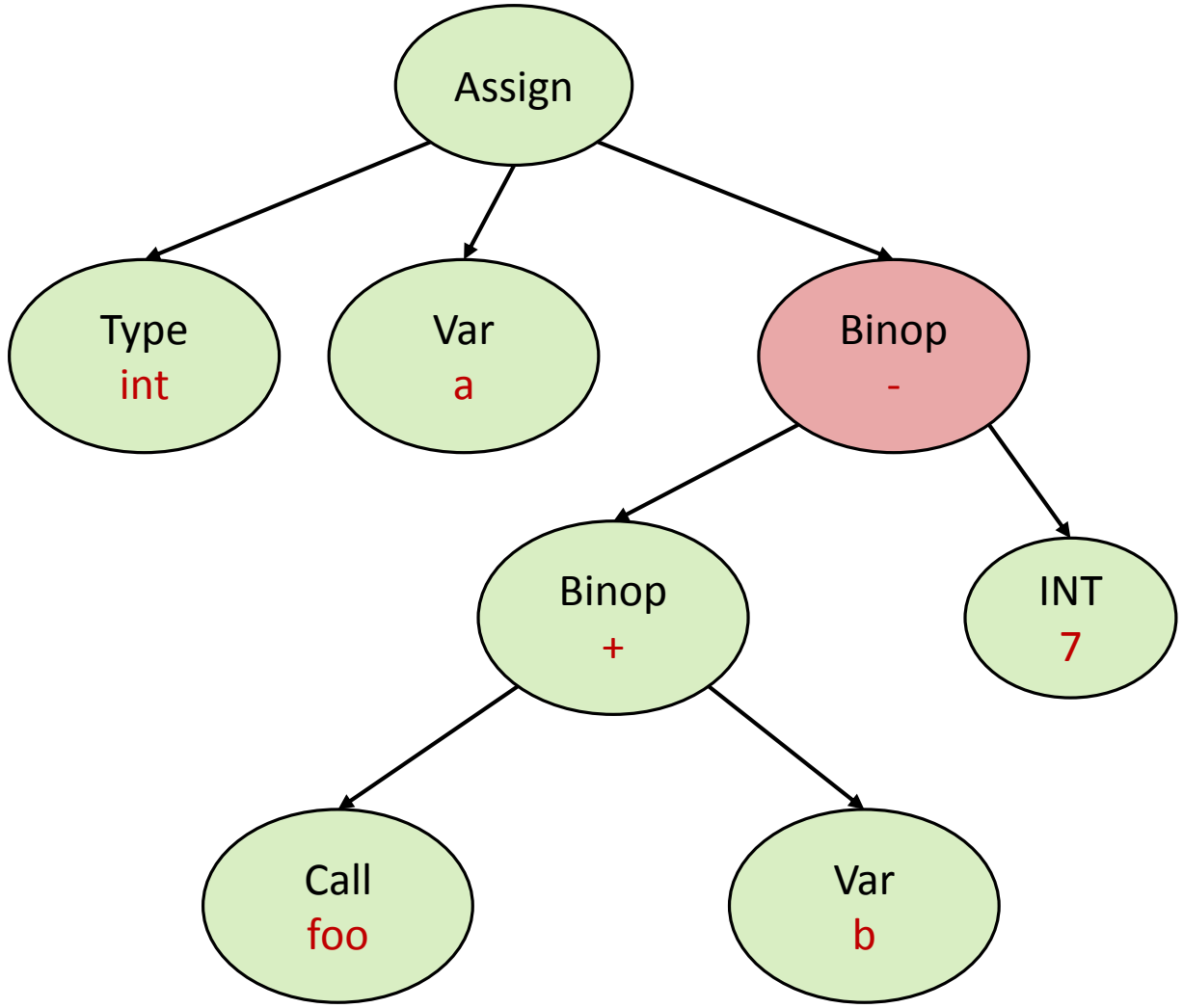
ID	Type	Kind
b	int	variable



```
int a = foo() + b - 7;
...
```

ID	Type	Kind
foo	int,void	function

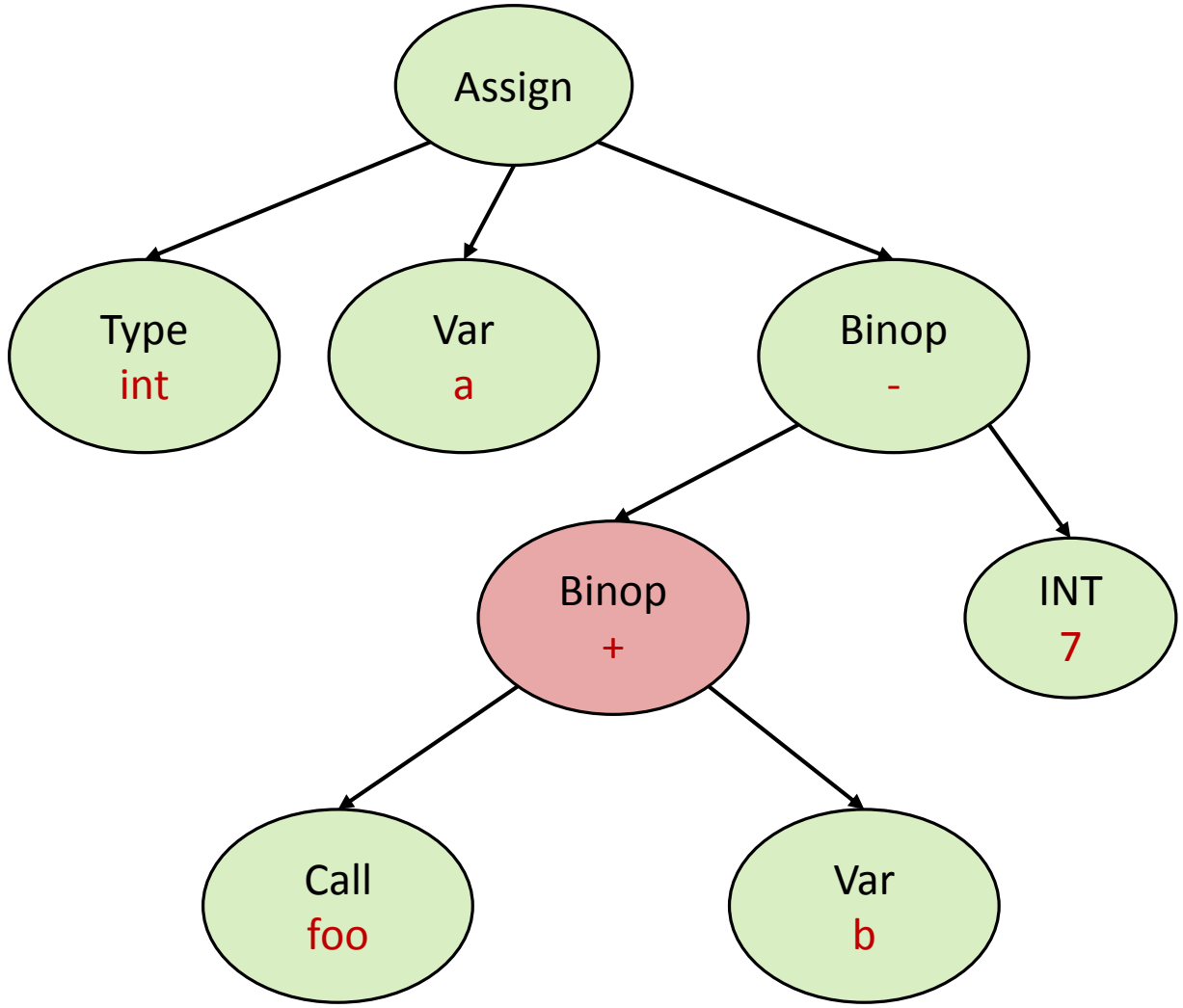
ID	Type	Kind
b	int	variable



```
int a = foo() + b - 7;
...
```

ID	Type	Kind
foo	int,void	function

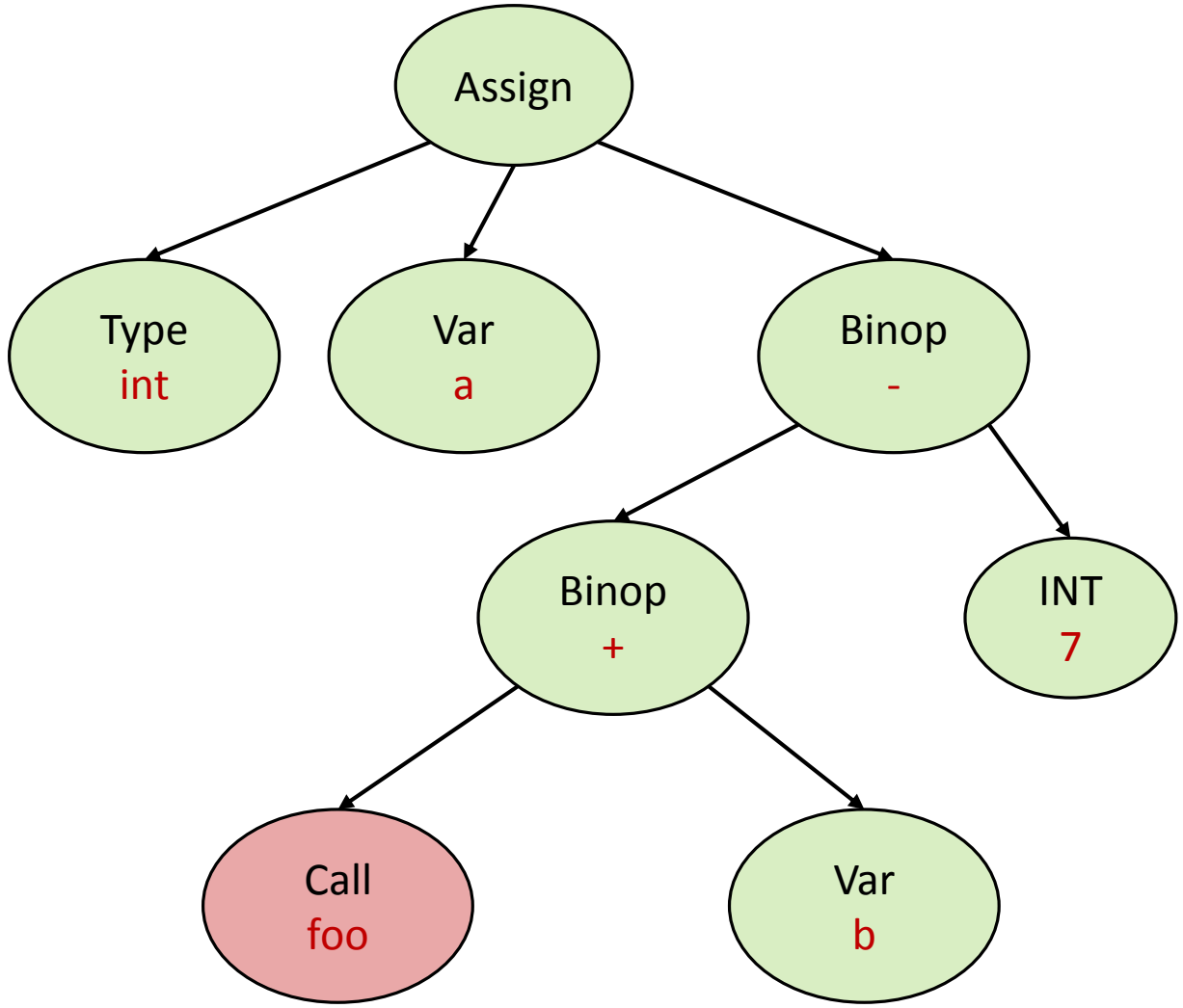
ID	Type	Kind
b	int	variable



```
int a = foo() + b - 7;
...
```

ID	Type	Kind
foo	int,void	function

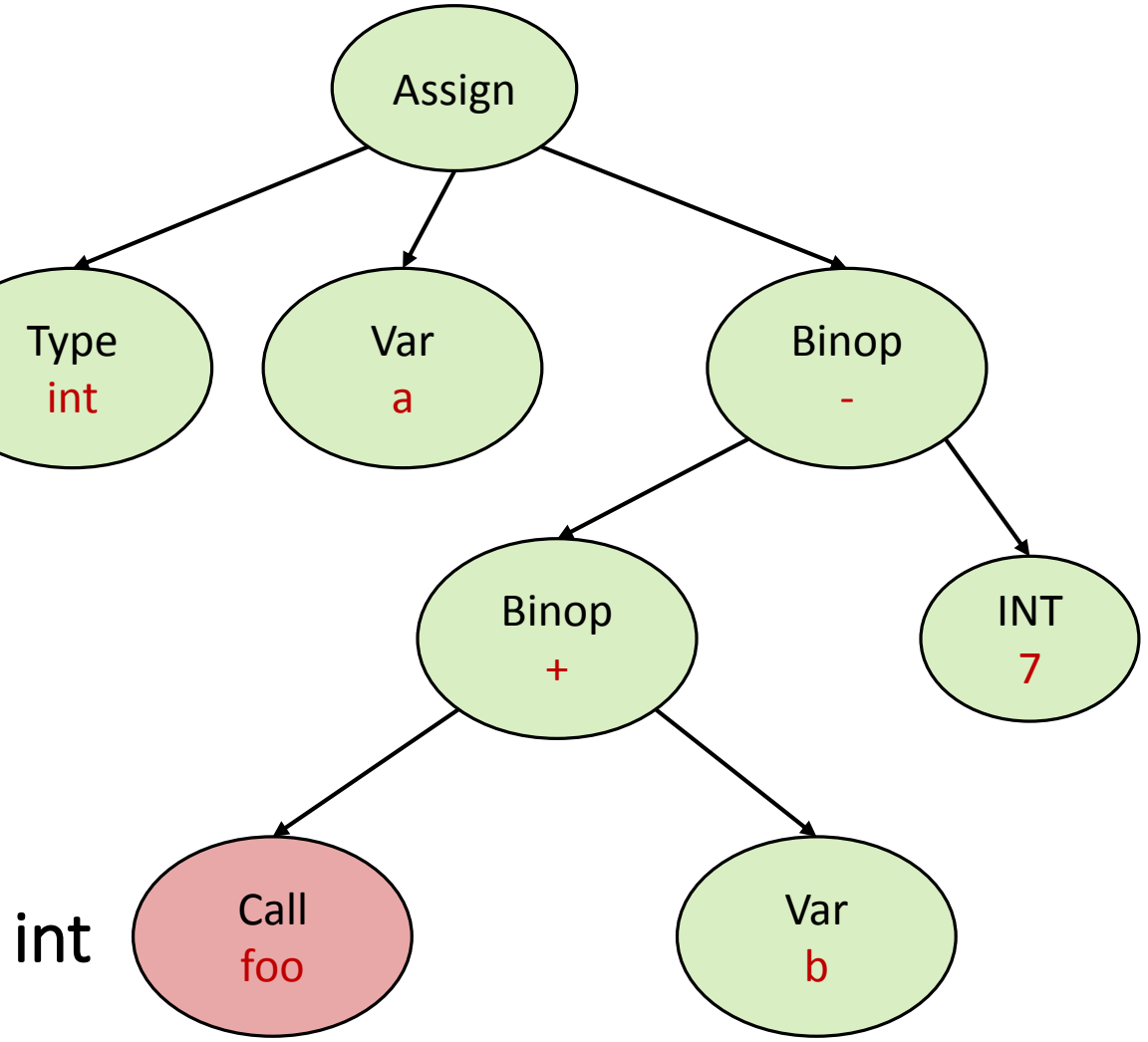
ID	Type	Kind
b	int	variable



```
int a = foo() + b - 7;
...
```

ID	Type	Kind
foo	int,void	function

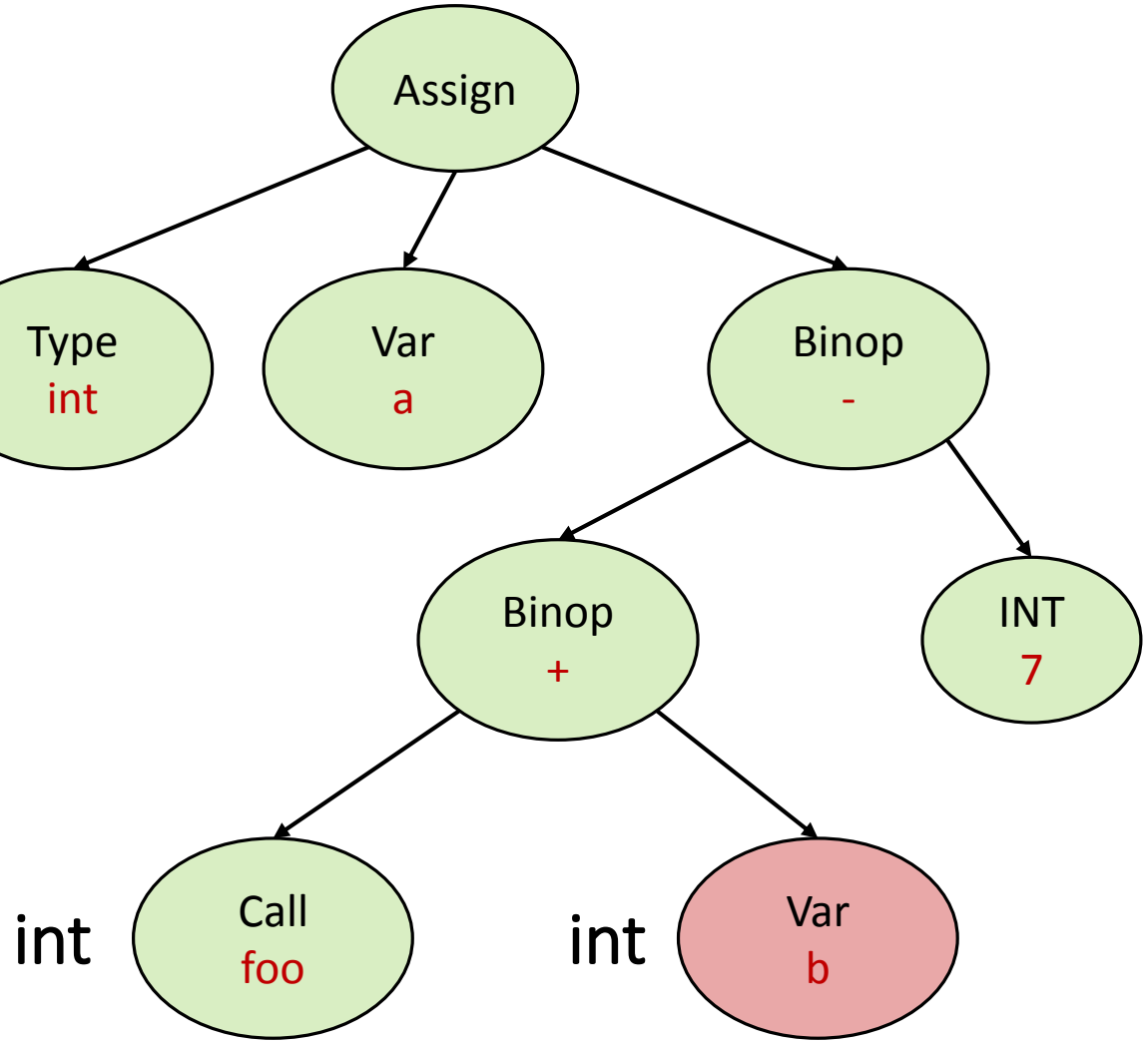
ID	Type	Kind
b	int	variable



```
int a = foo() + b - 7;
...
```

ID	Type	Kind
foo	int,void	function

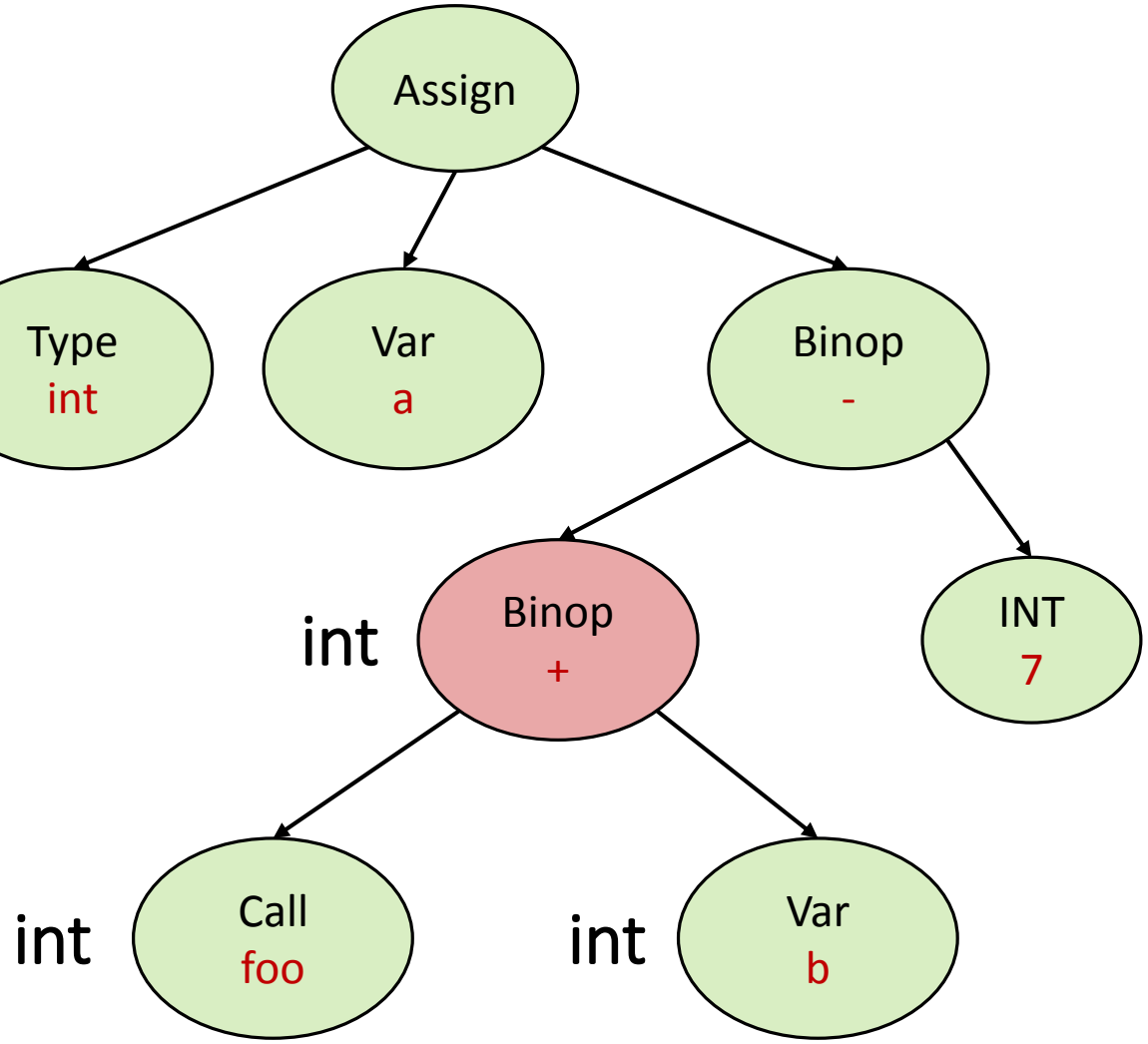
ID	Type	Kind
b	int	variable



```
int a = foo() + b - 7;
...
```

ID	Type	Kind
foo	int,void	function

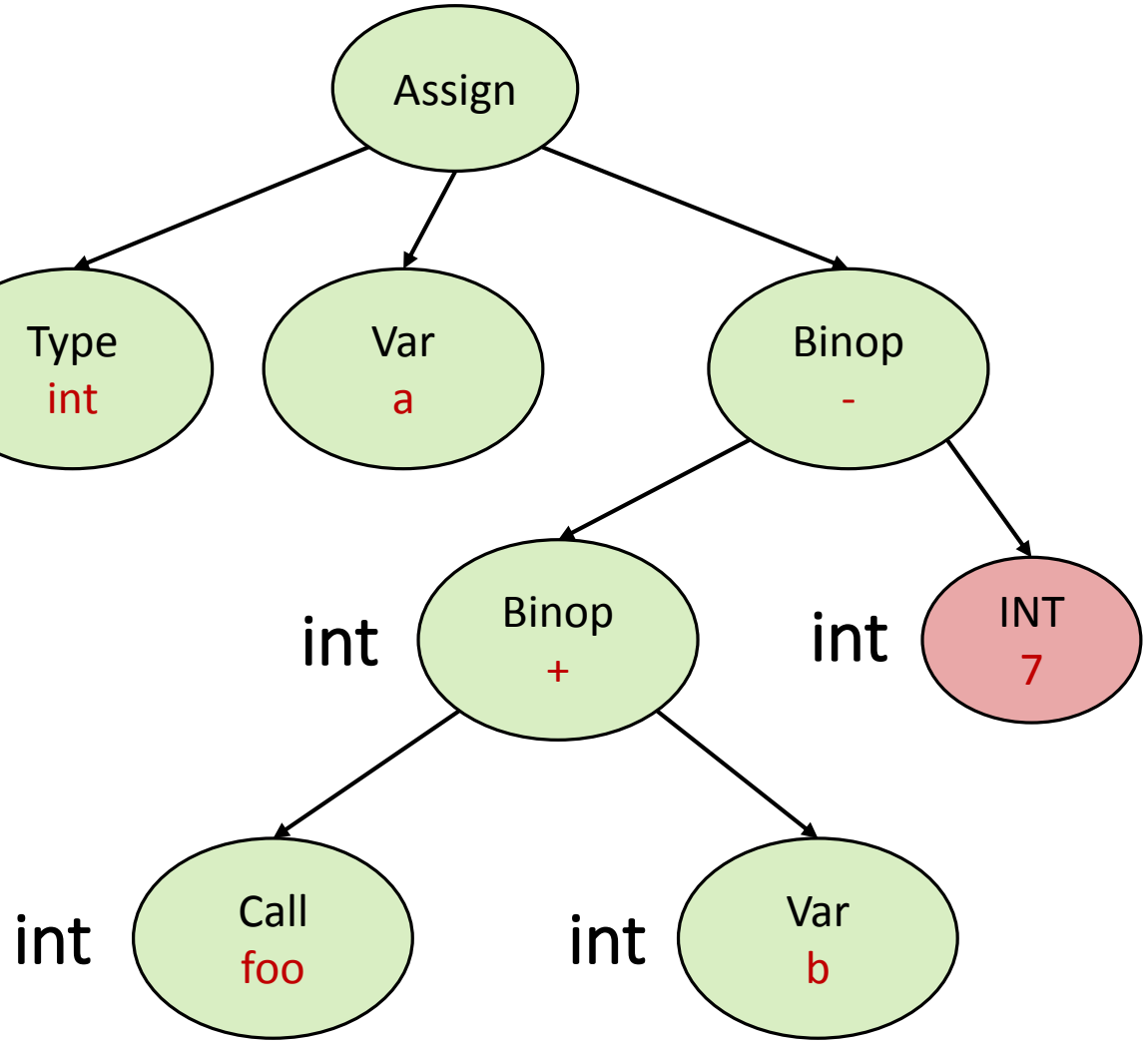
ID	Type	Kind
b	int	variable



```
int a = foo() + b - 7;
...
```

ID	Type	Kind
foo	int,void	function

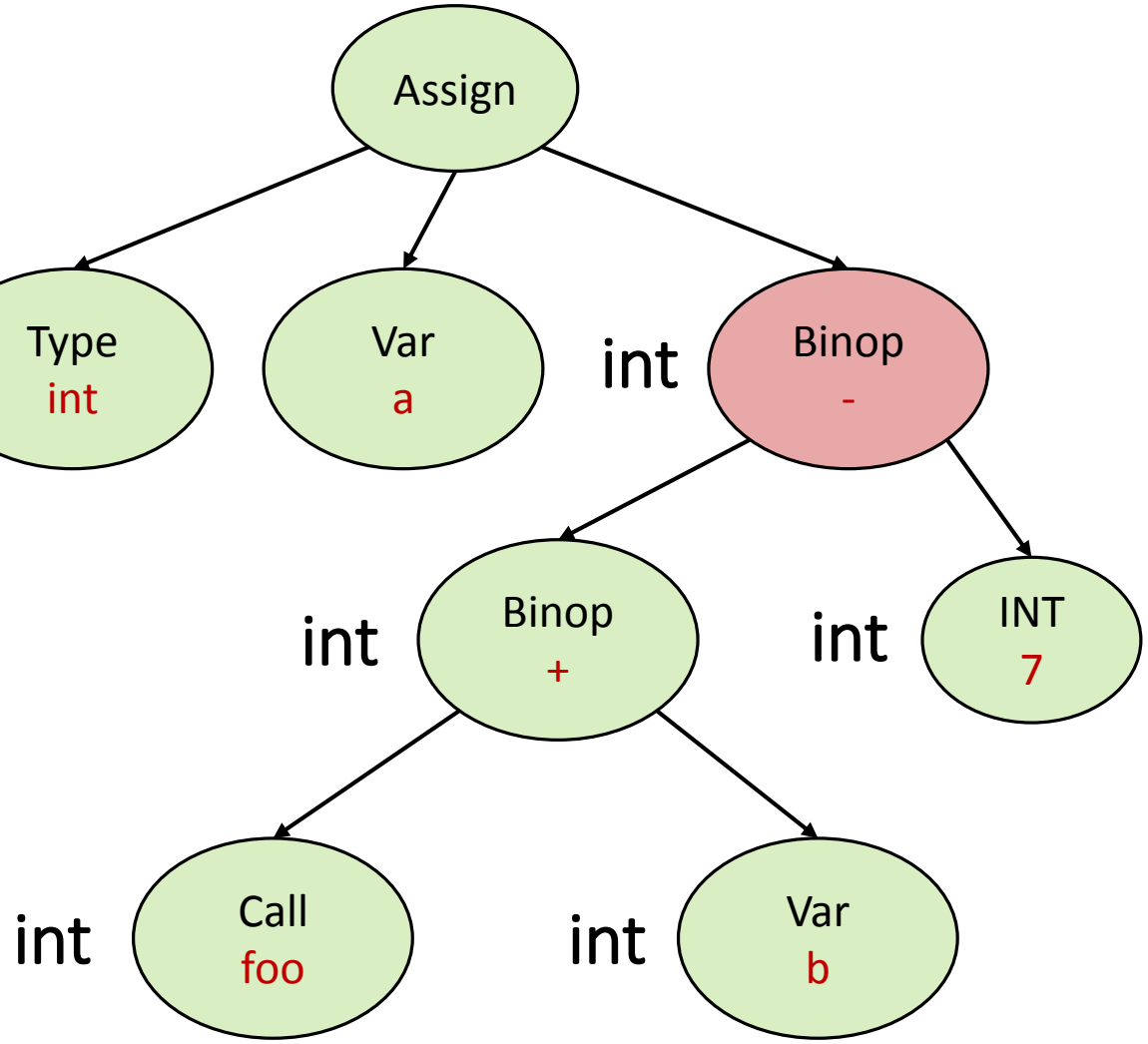
ID	Type	Kind
b	int	variable



```
int a = foo() + b - 7;
...
```

ID	Type	Kind
foo	int,void	function

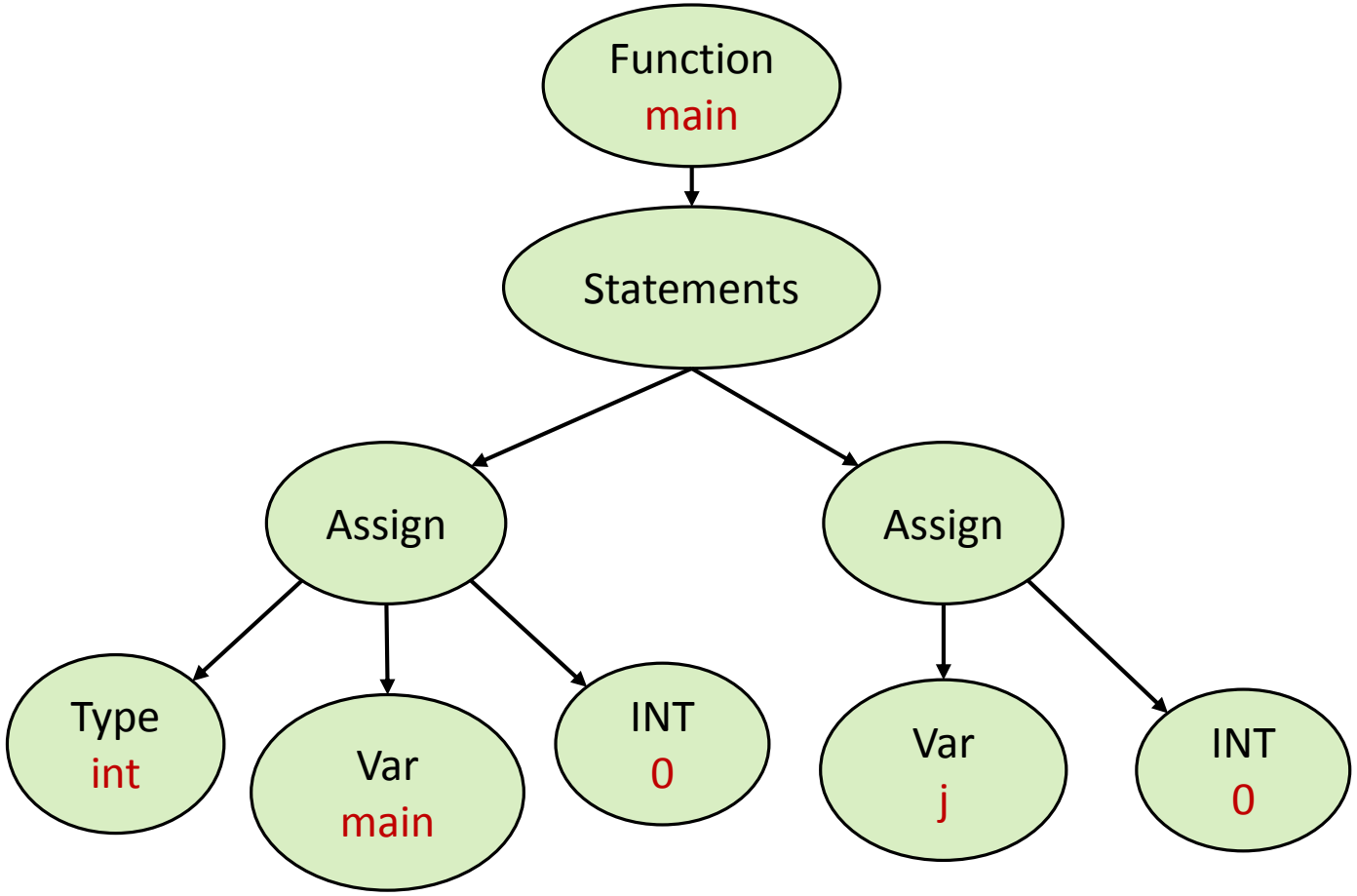
ID	Type	Kind
b	int	variable



Examples

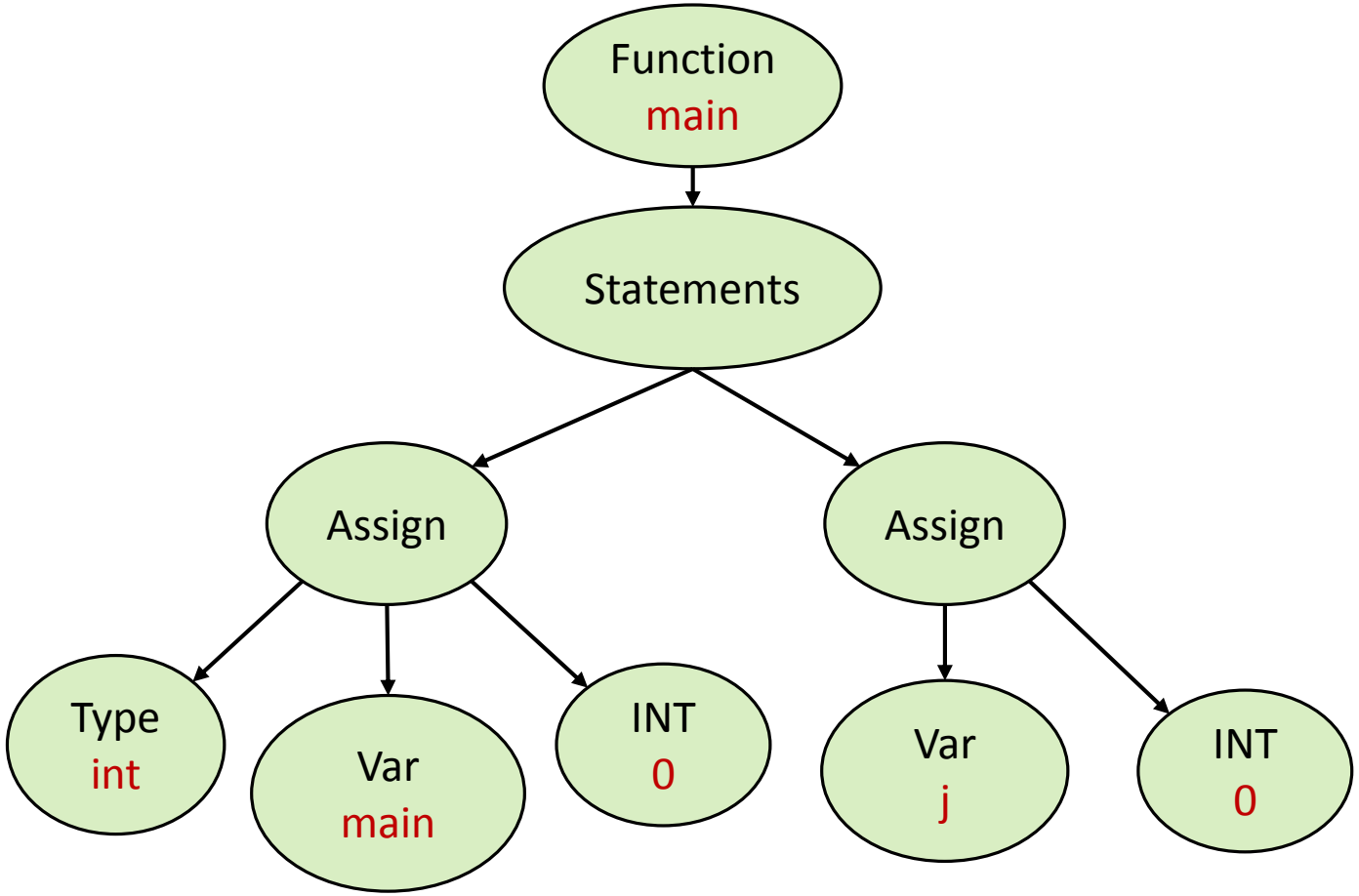
Assignments

```
void main() {
   int main = 0
   j = 0;
}
```



Assignments

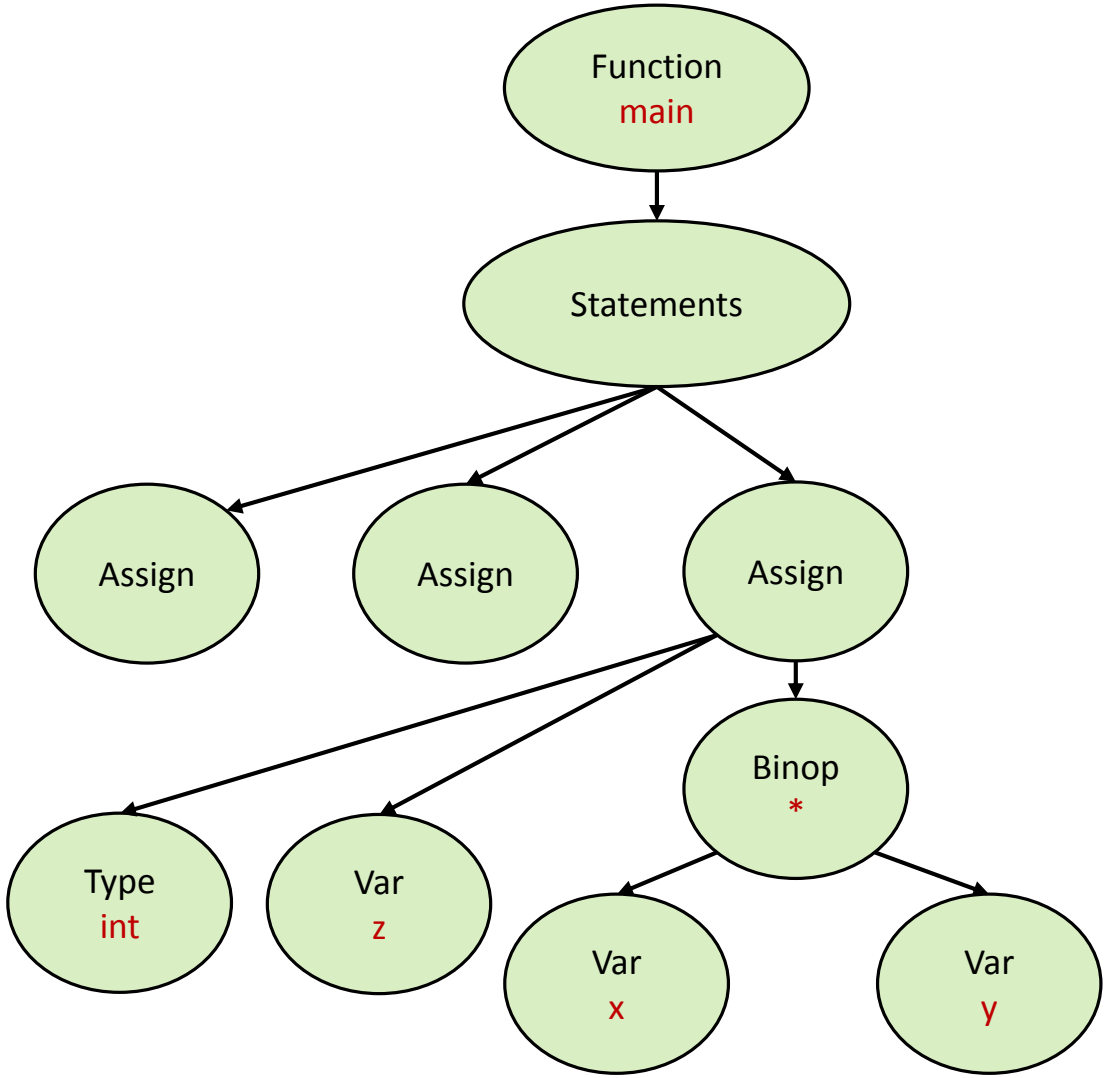
```
void main() {
   int main = 0
   j = 0;
}
```



Invalid

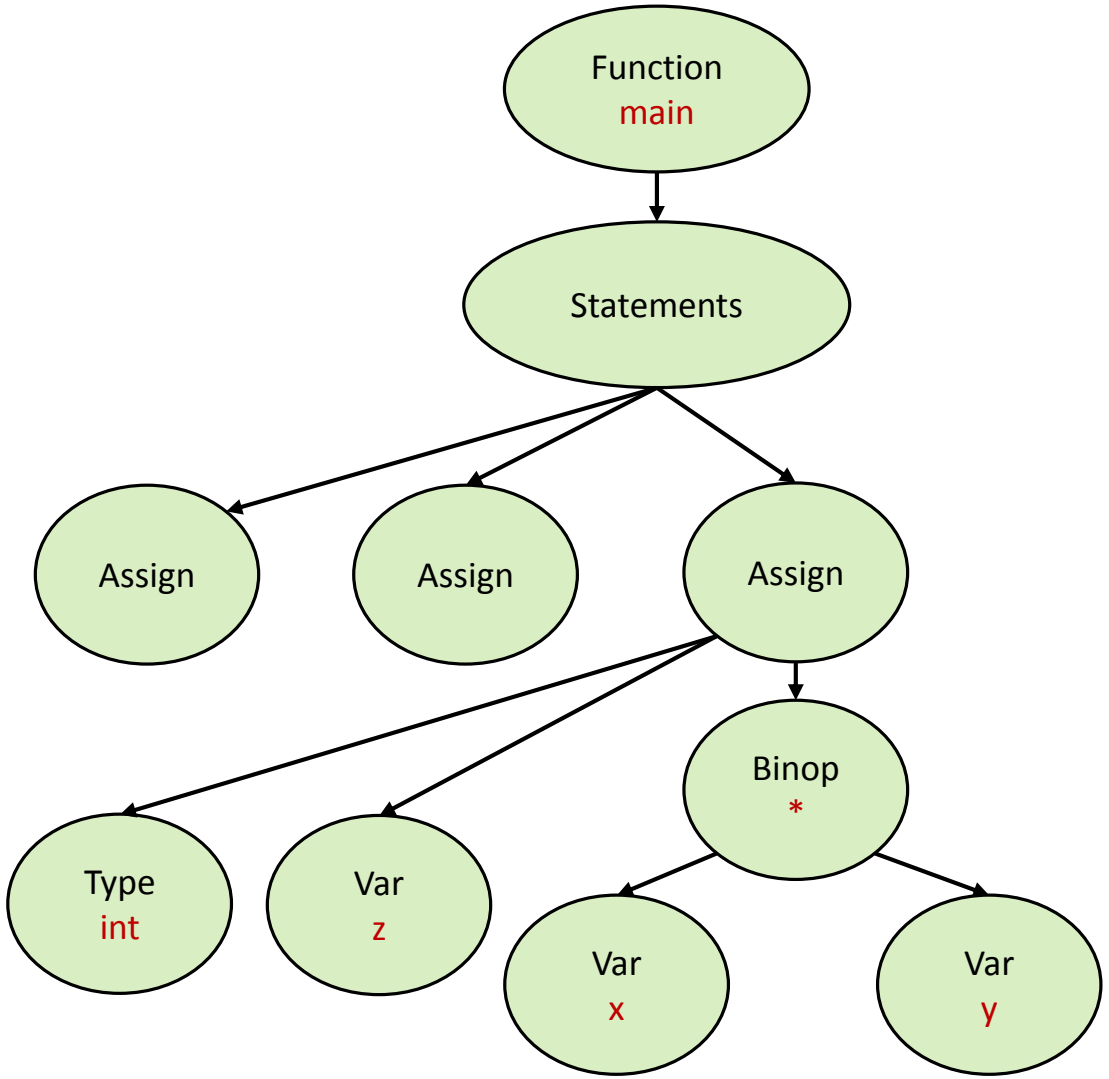
Binary Operations

```
void main() {
   string x = "A";
   string y = "B";
   string z = x * y;
}
```



Binary Operations

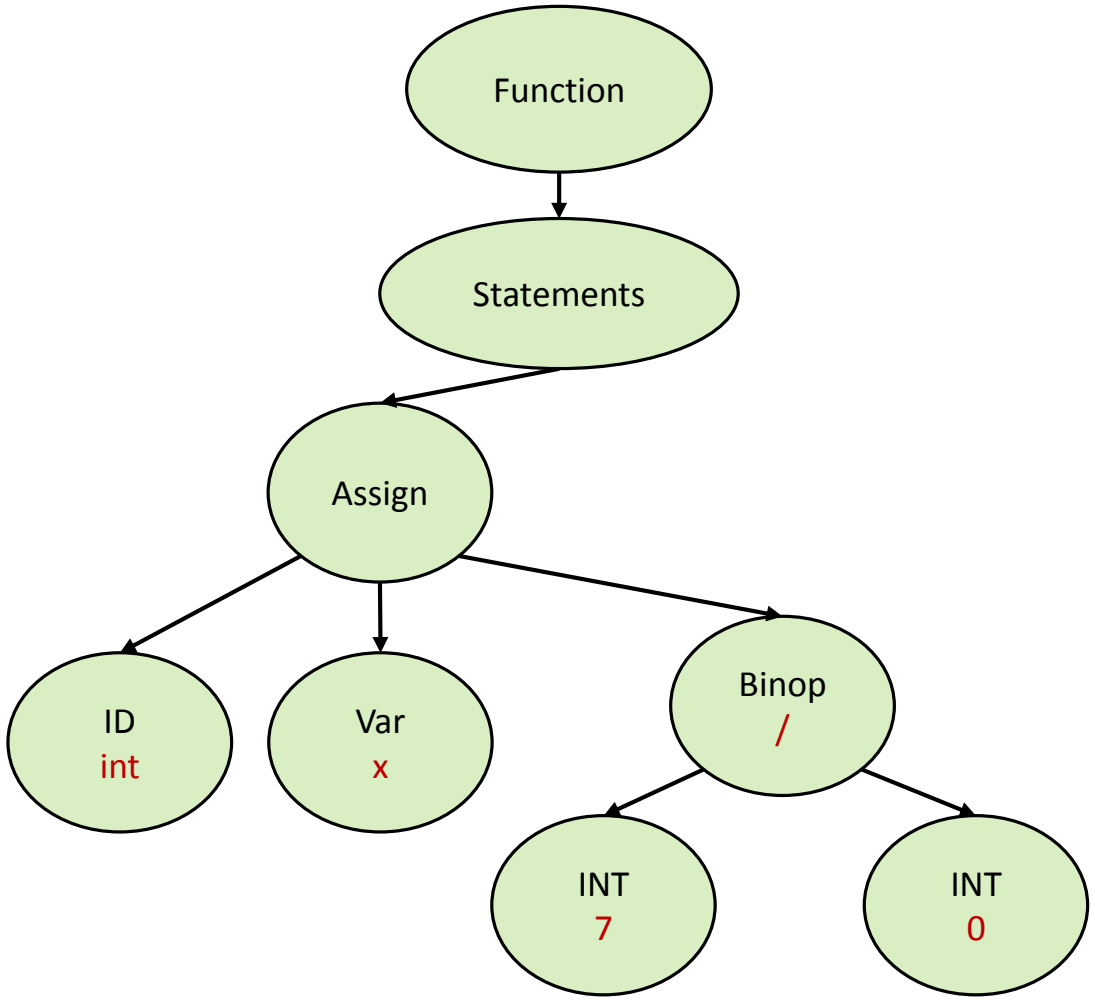
```
void main() {
   string x = "A";
   string y = "B";
   string z = x * y;
}
```



Invalid

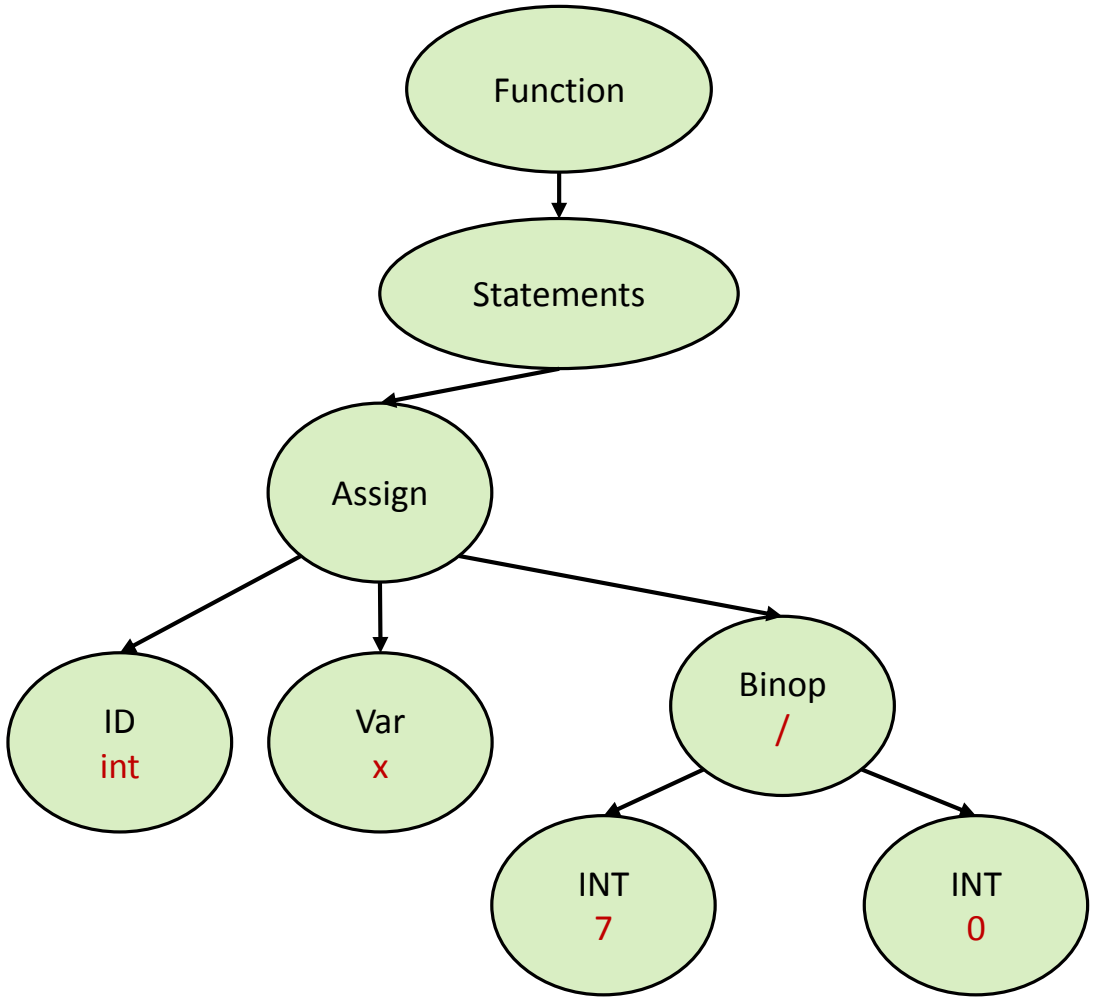
Binary Operations

```
void main() {
  int x = 7 / 0;
}
```



Binary Operations

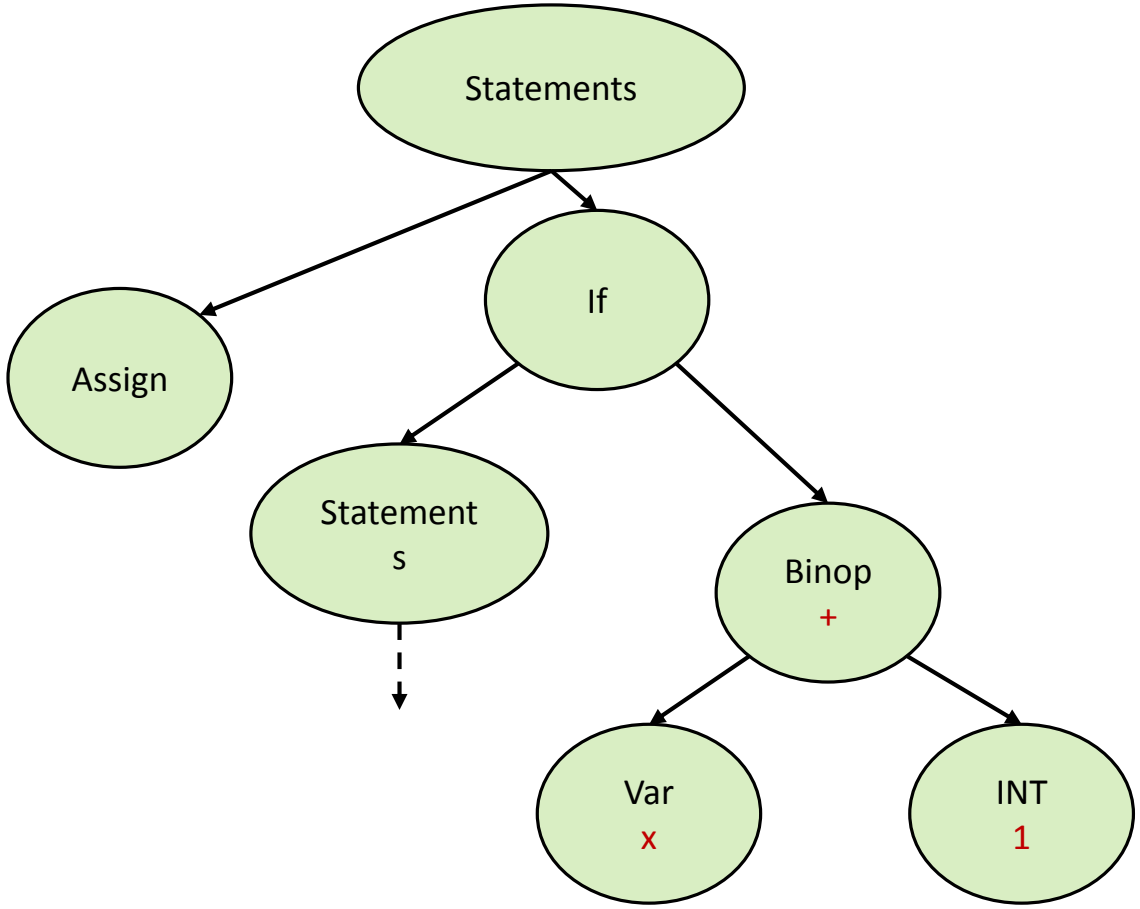
```
void main() {
  int x = 7 / 0;
}
```



Invalid

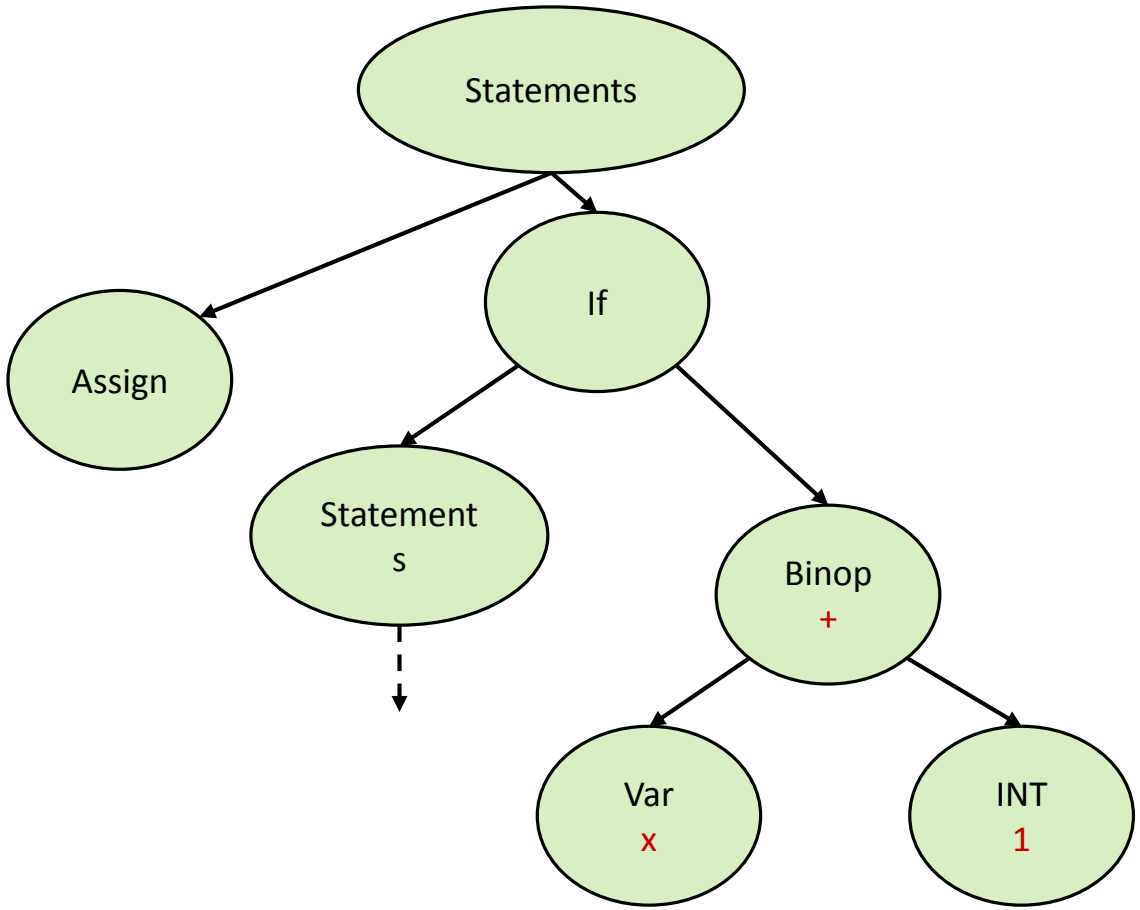
If, While, …

```
void main() {
  int x = 1;
  if (x + 1) {
    int z = 2;
  }
}
```



If, While, …

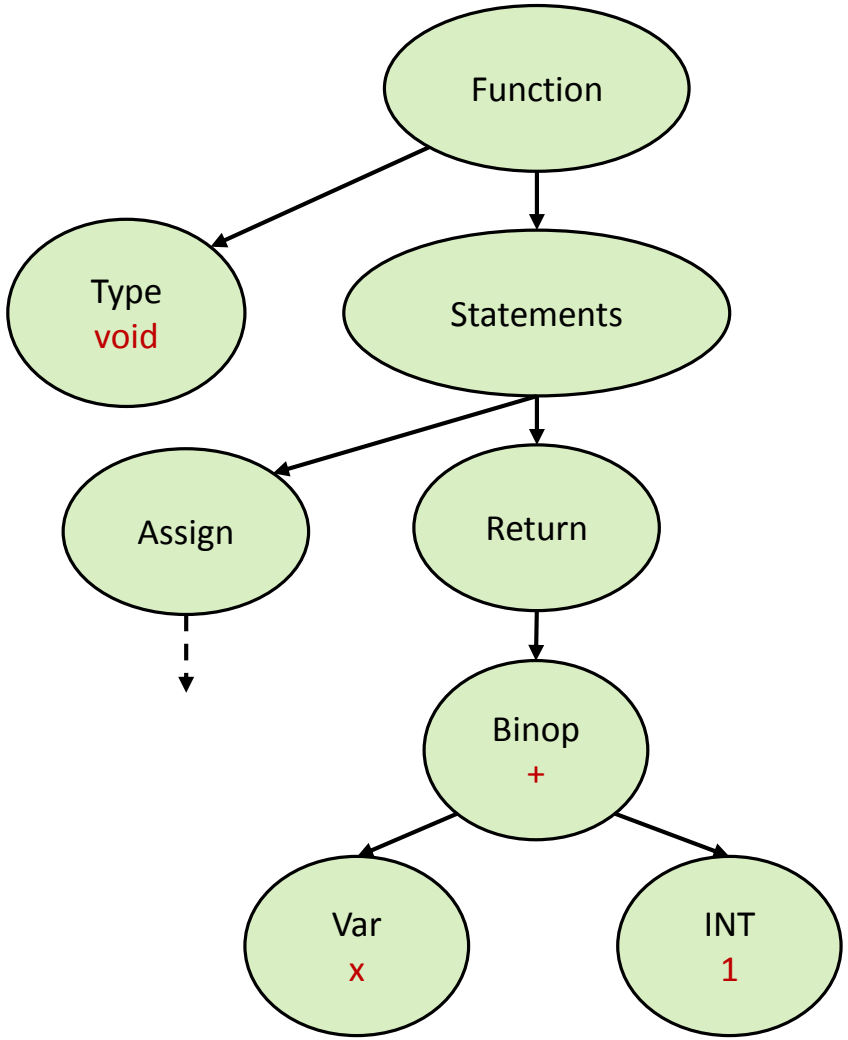
```
void main() {
  int x = 1;
  if (x + 1) {
    int z = 2;
  }
}
```



Valid

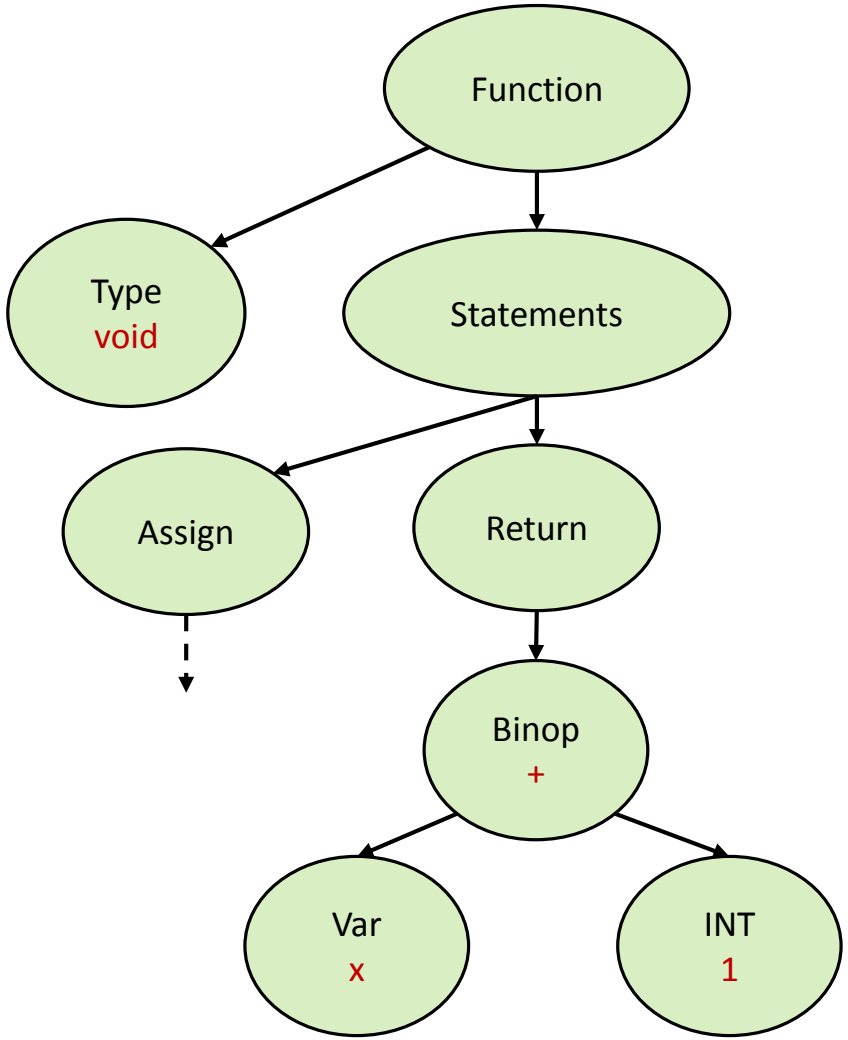
Return Statement

```
void main() {
  int x = 1;
  return x + 1;
}
```



Return Statement

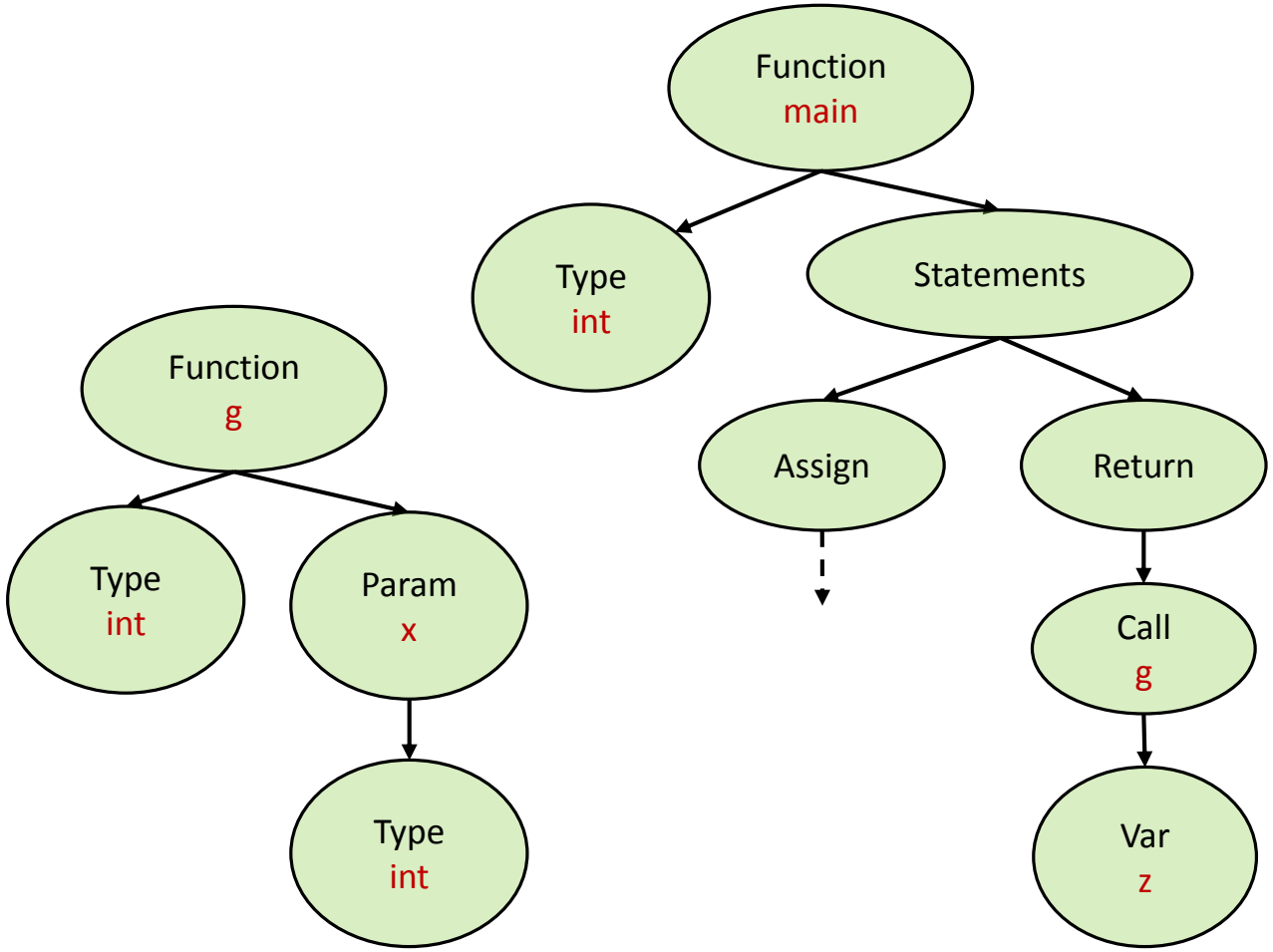
```
void main() {
  int x = 1;
  return x + 1;
}
```



Invalid

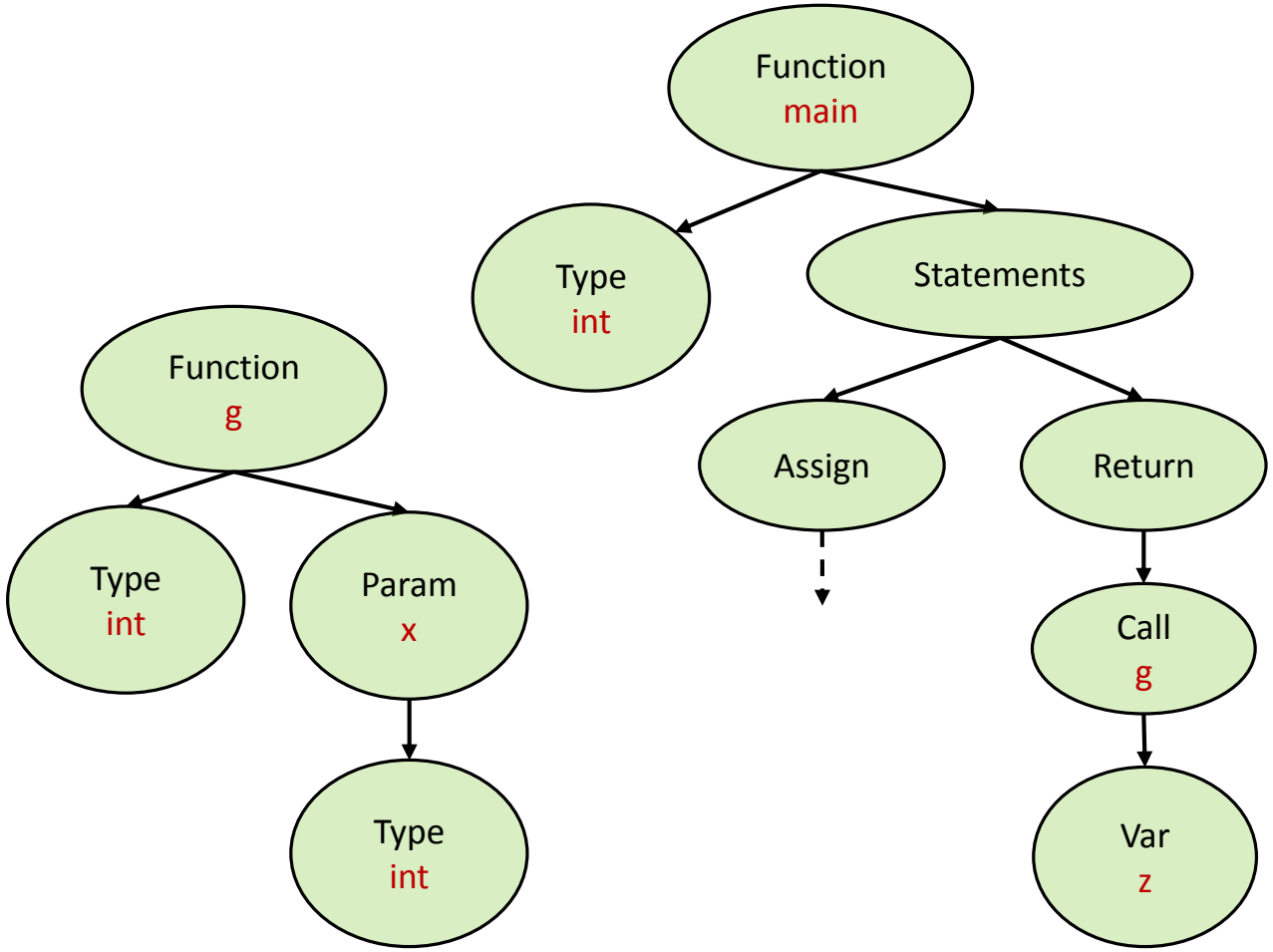
Function Calls

```
int g(int x) {
  return x + 1;
}
int main() {
  string z = "..."
  return g(z);
}
```



Function Calls

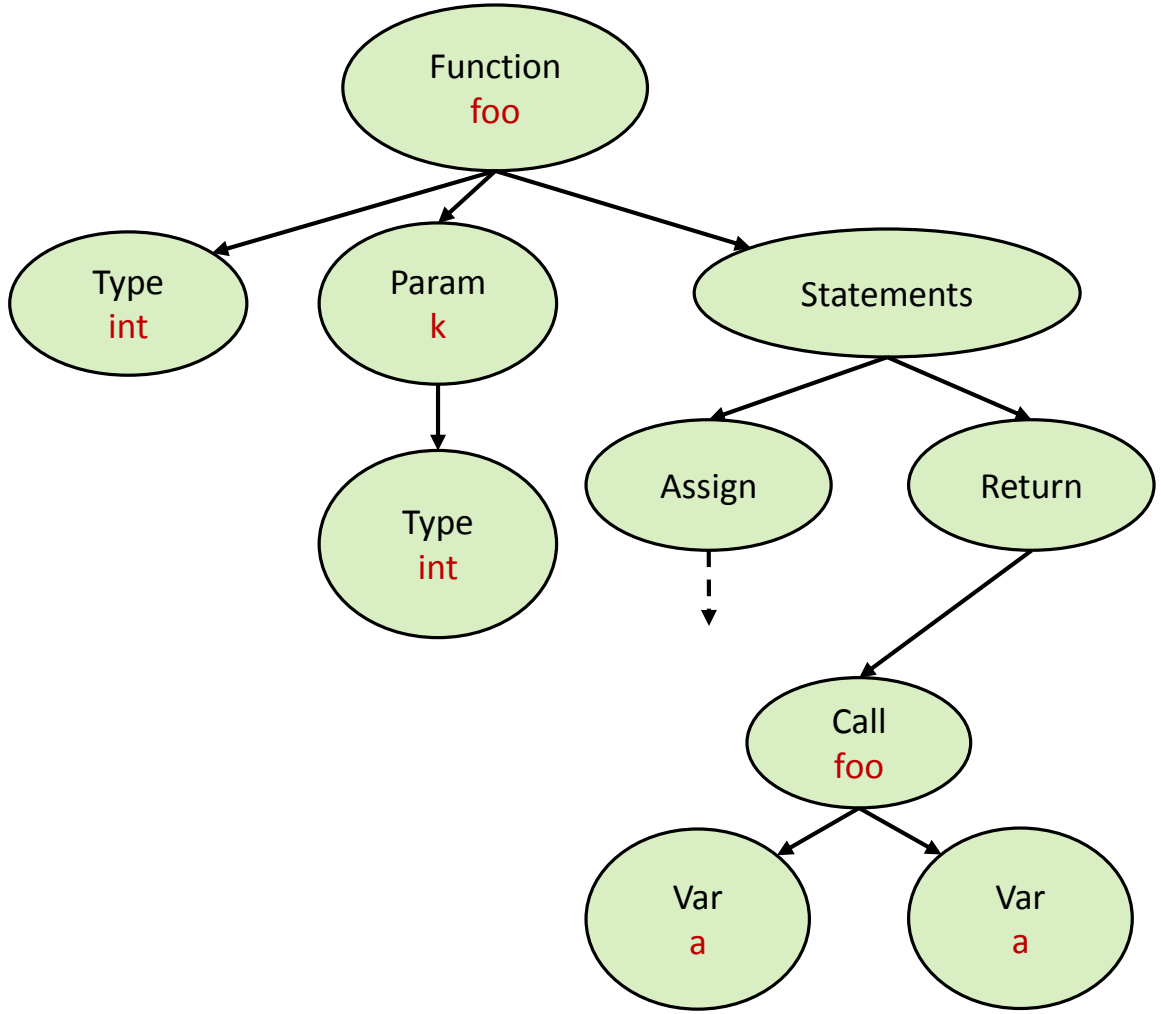
```
int g(int x) {
  return x + 1;
}
int main() {
  string z = "..."
  return g(z);
}
```



Invalid

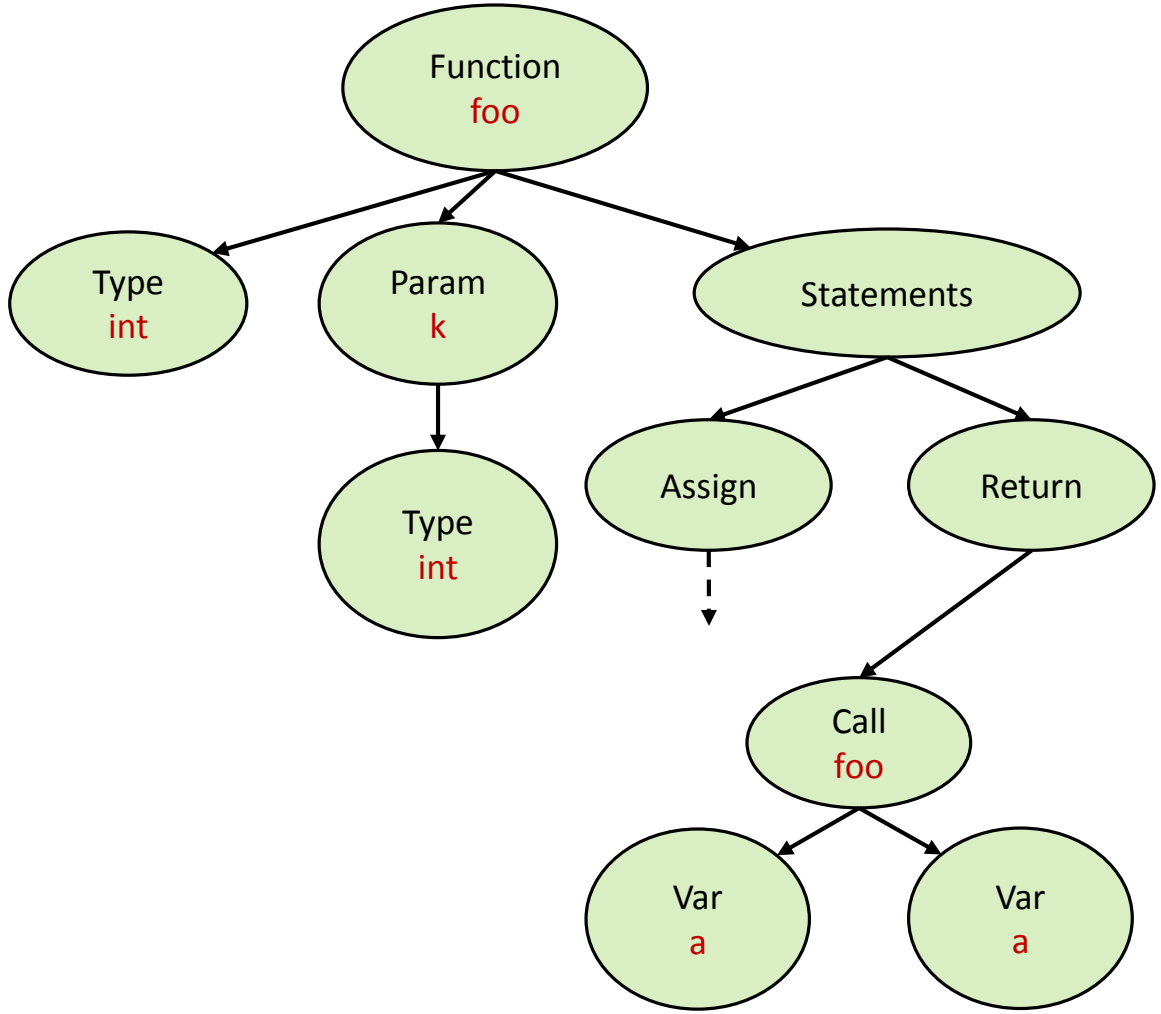
Function Calls

```
int foo(int k) {
  int a = k * 10;
  return foo(a, a);
}
```



Function Calls

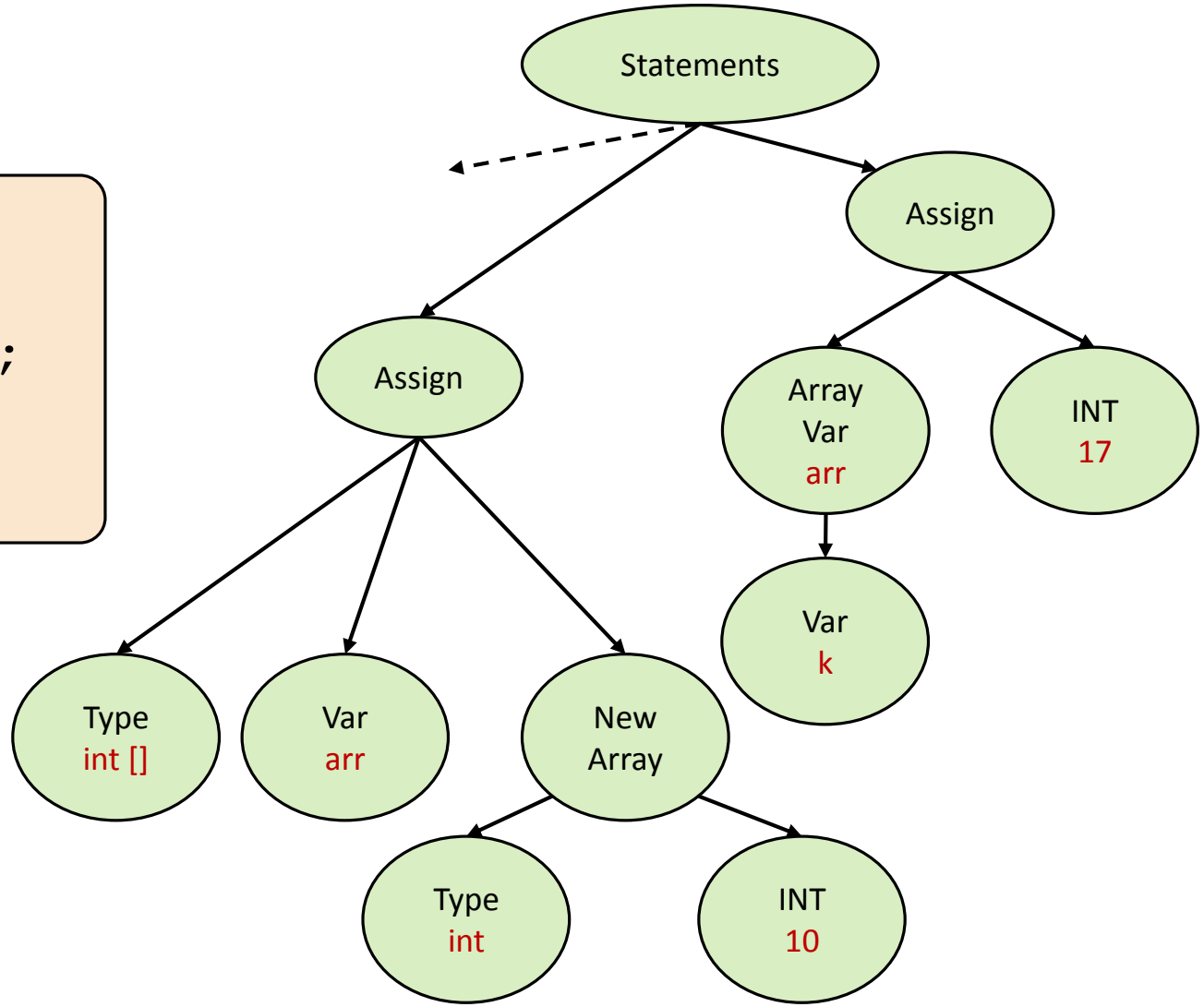
```
int foo(int k) {
  int a = k * 10;
  return foo(a, a);
}
```



Invalid

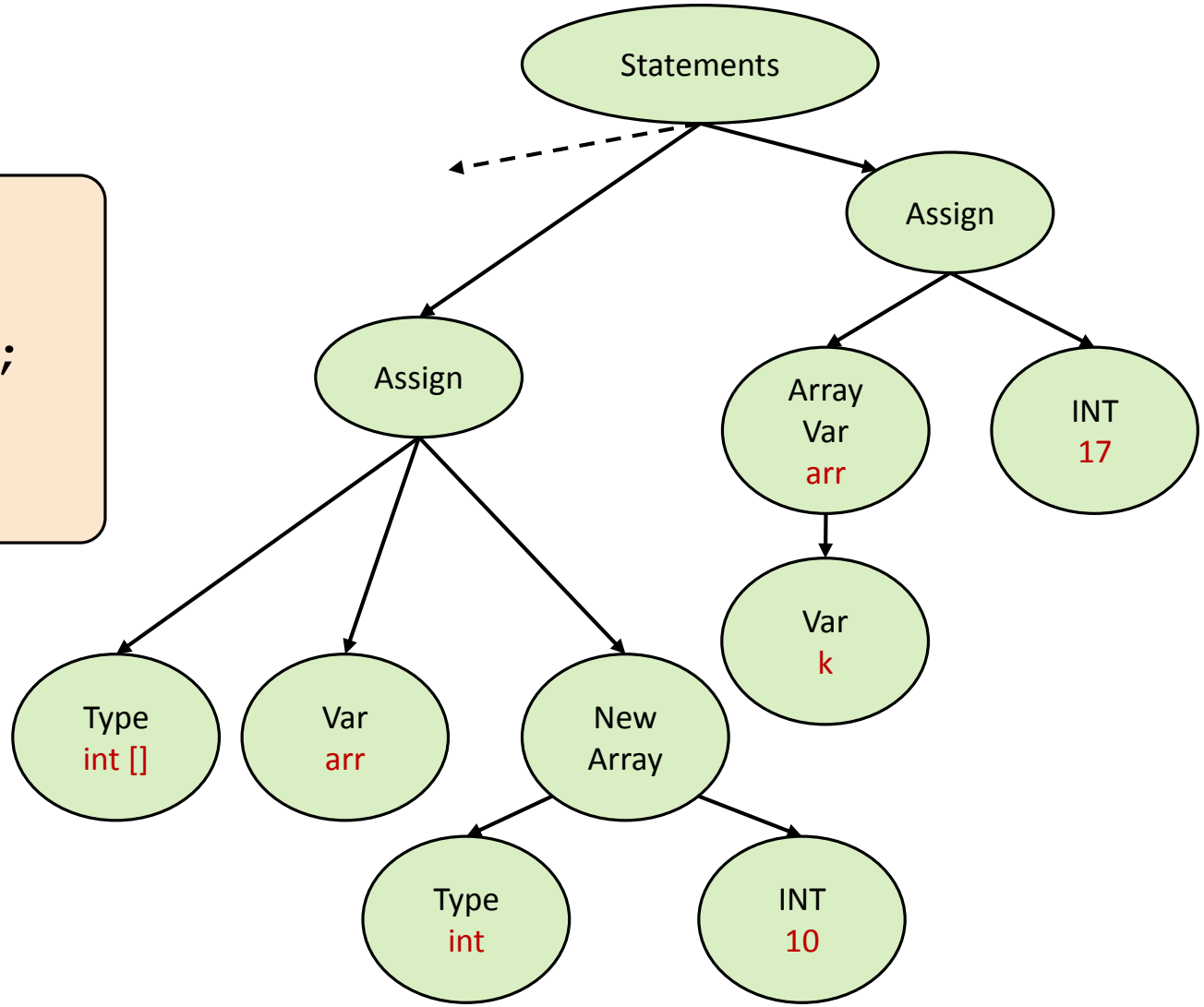
Arrays

```
void foo(int d) {
  int k = 3;
  int[] arr = new int[10];
  arr[k] = 17;
}
```



Arrays

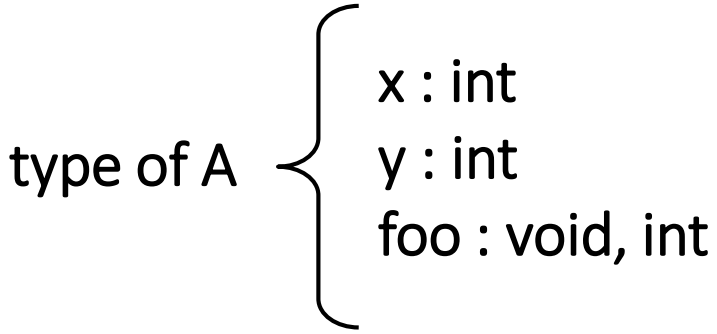
```
void foo(int d) {
  int k = 3;
  int[] arr = new int[10];
  arr[k] = 17;
}
```



Valid

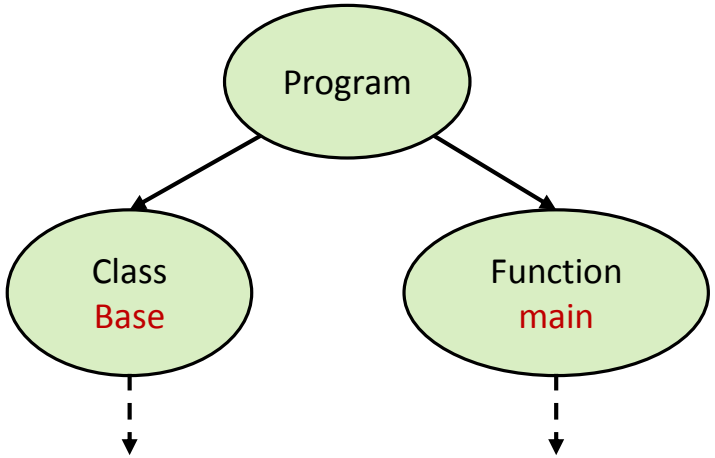
Classes

```
class A {
  int x;
  int y;
  void foo(int) { }
}
```



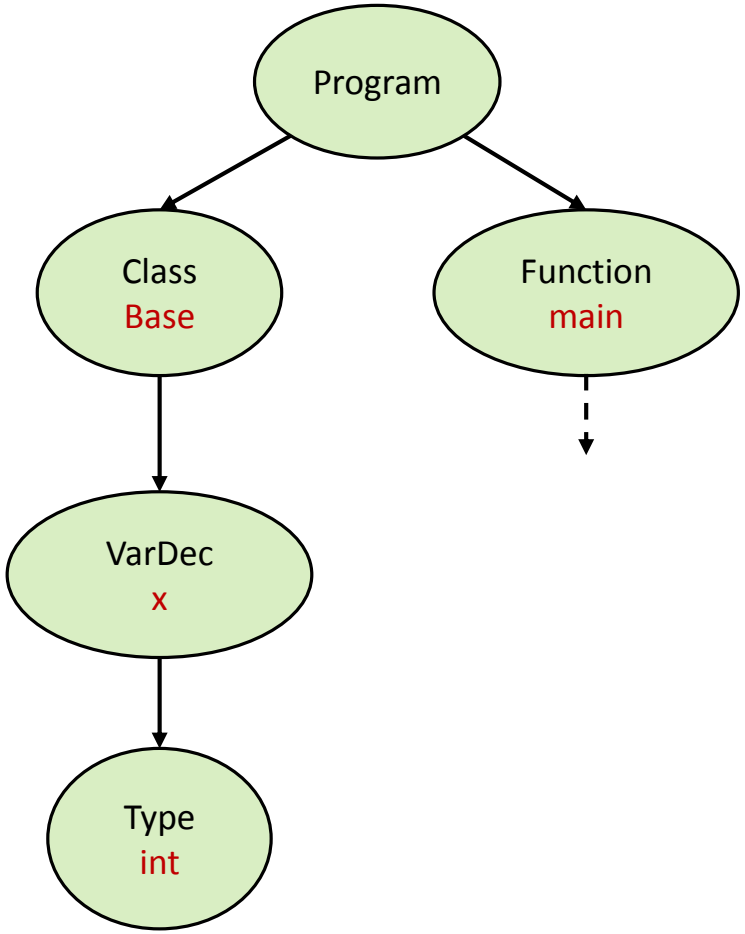
Classes

```
class Base {
  int x;
}
void main() {
  Base o = new Base;
  o.y = 1;
}
```



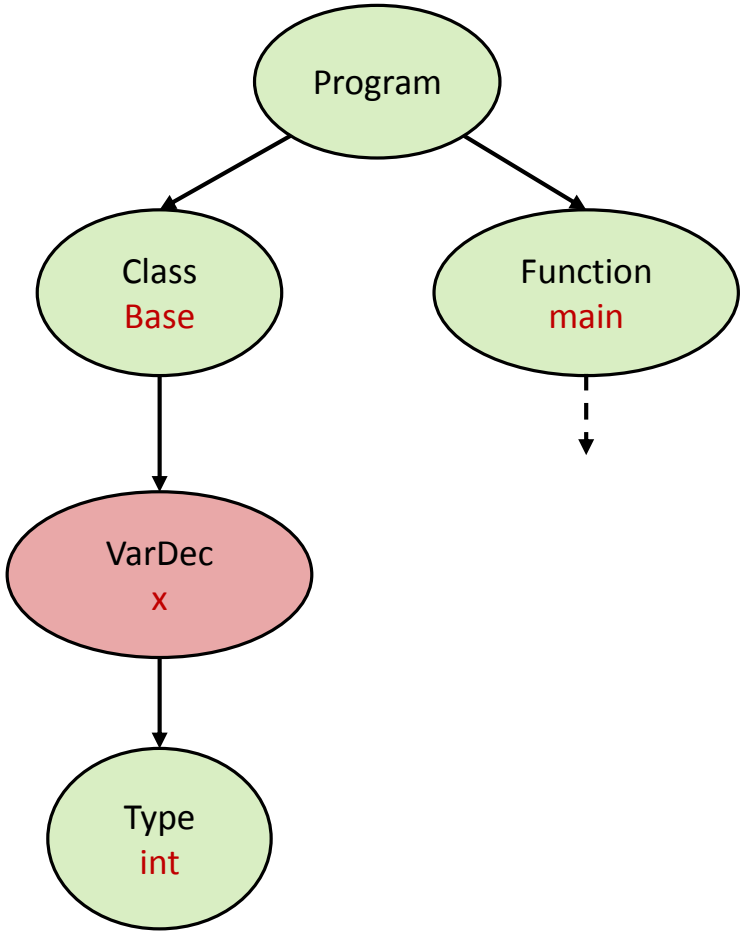
Classes

```
class Base {
  int x;
}
void main() {
  Base o = new Base;
  o.y = 1;
}
```



Classes

```
class Base {
  int x;
}
void main() {
  Base o = new Base;
  o.y = 1;
}
```



ID	Type	Kind
Base	…	class

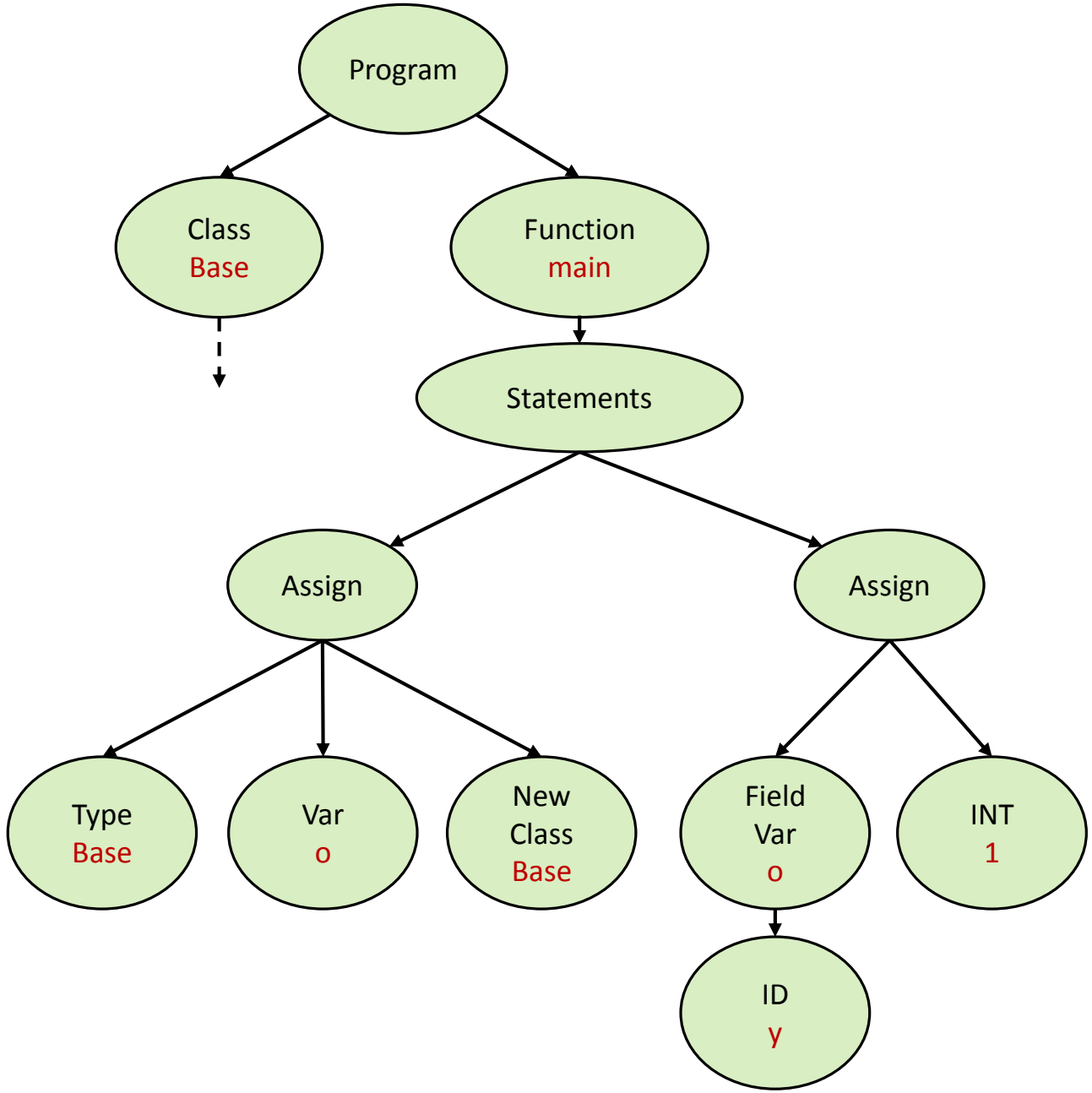
$scope_1$

ID	Type	Kind
x	int	variable

$scope_2$

Classes

```
class Base {
  int x;
}
void main() {
  Base o = new Base;
  o.y = 1;
}
```



Classes

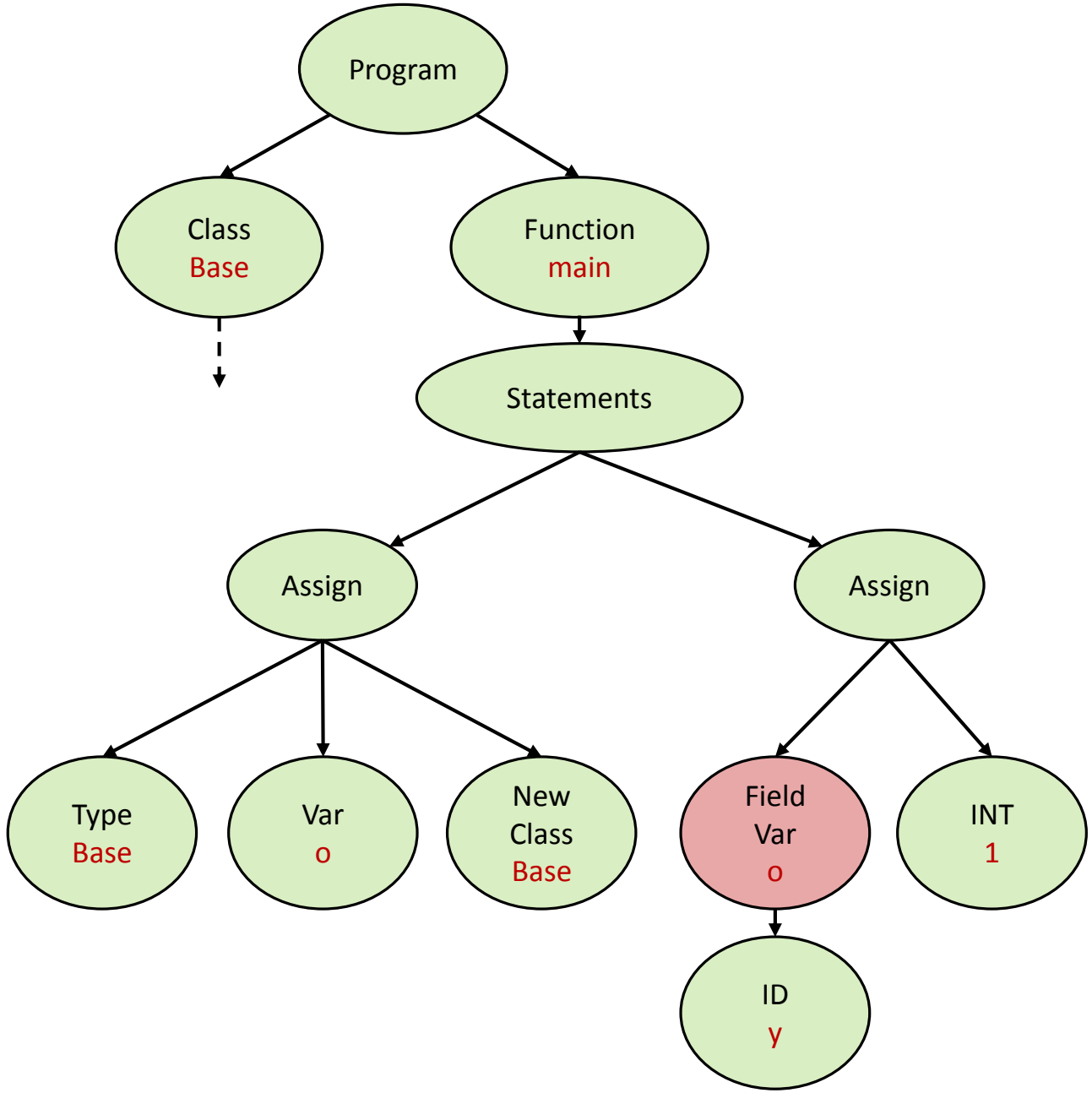
```
class Base {
  int x;
}
void main() {
  Base o = new Base;
  o.y = 1;
}
```

ID	Type	Kind
Base	…	class
main	…	function

$scope_1$

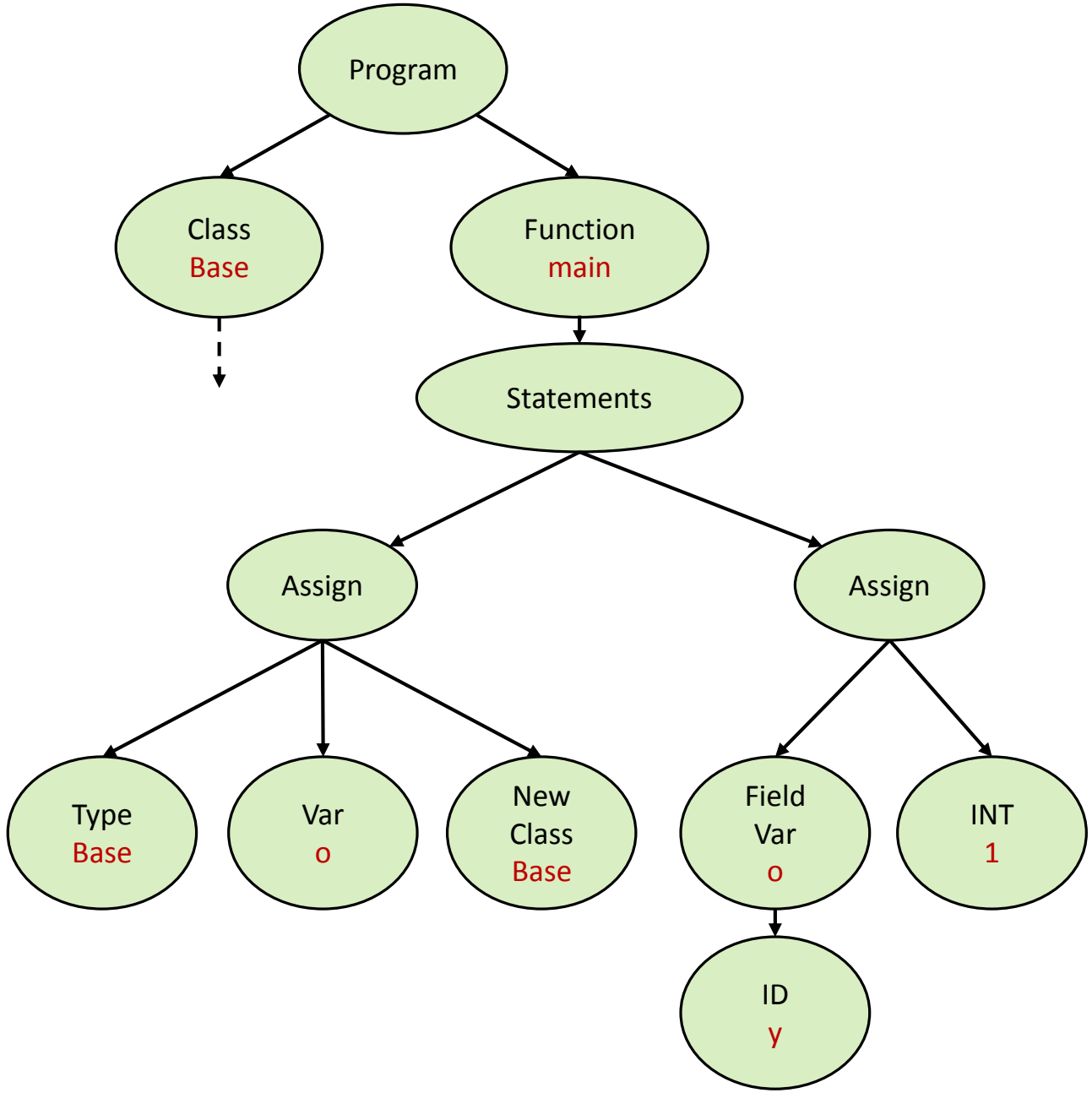
ID	Type	Kind
o	Base	variable

$scope_2$



Classes

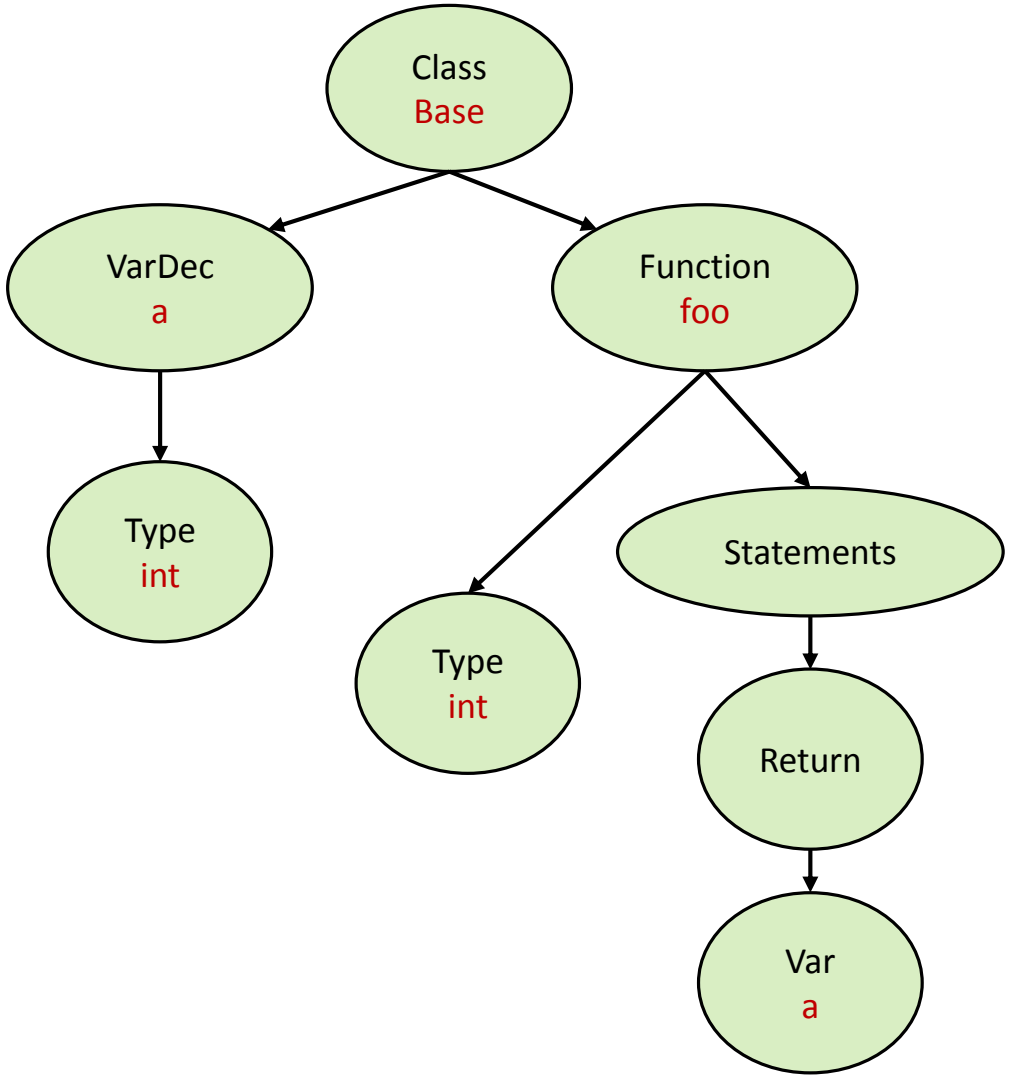
```
class Base {
  int x;
}
void main() {
  Base o = new Base;
  o.y = 1;
}
```



Invalid

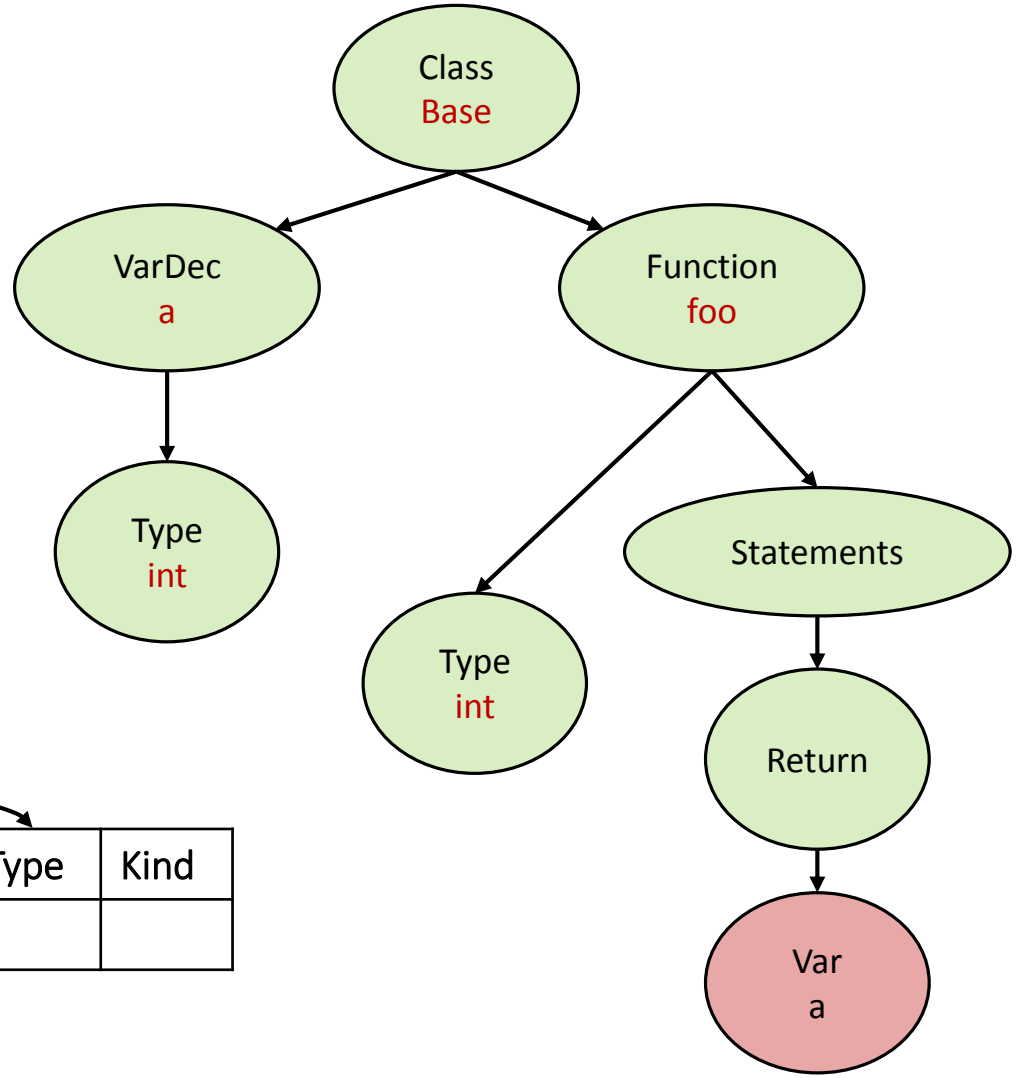
Classes

```
class Base {
  int a;
  int foo() {
    return a;
  }
}
```



Classes

```
class Base {
  int a;
  int foo() {
    return a;
  }
}
```



ID	Type	Kind
Base	…	class

$scope_1$

ID	Type	Kind
a	int	variable
foo	…	function

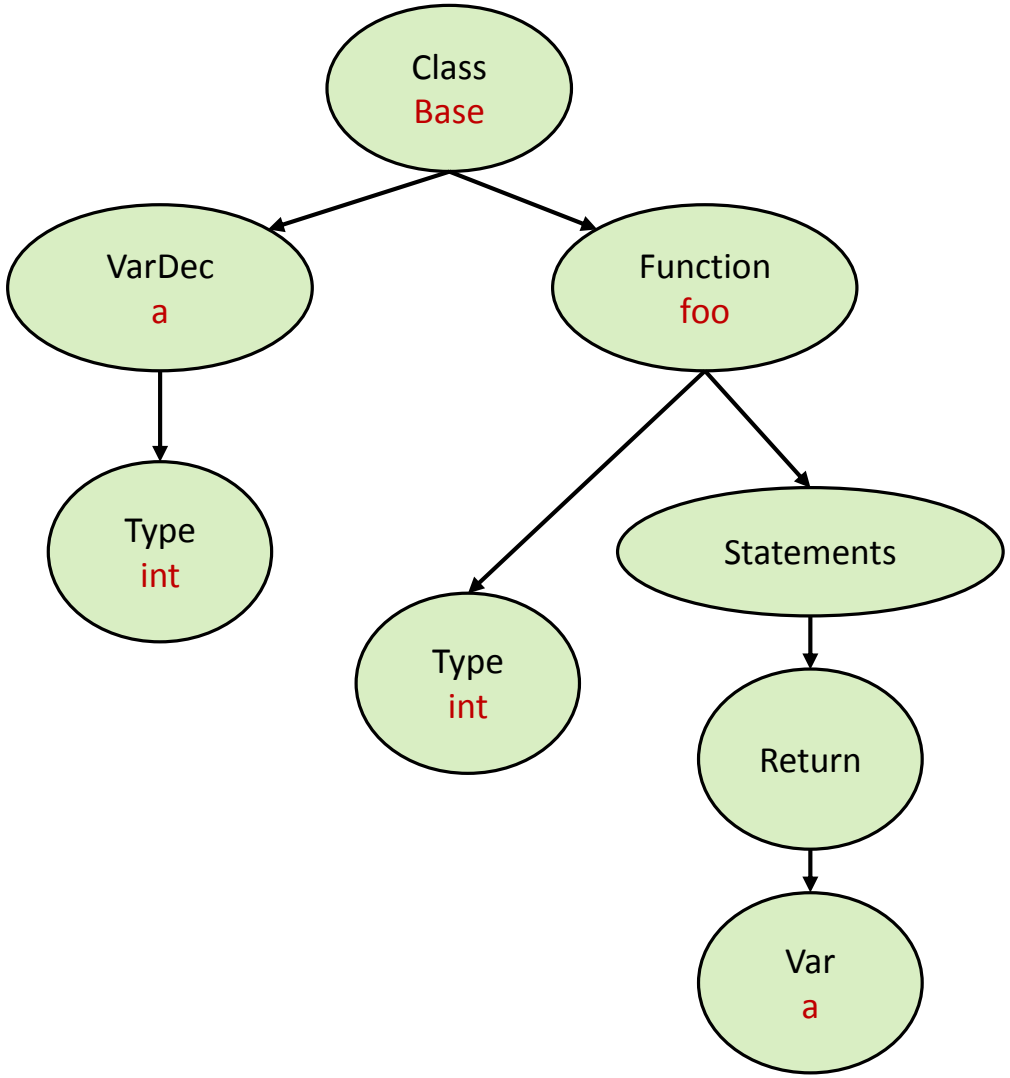
$scope_2$

ID	Type	Kind

$scope_3$

Classes

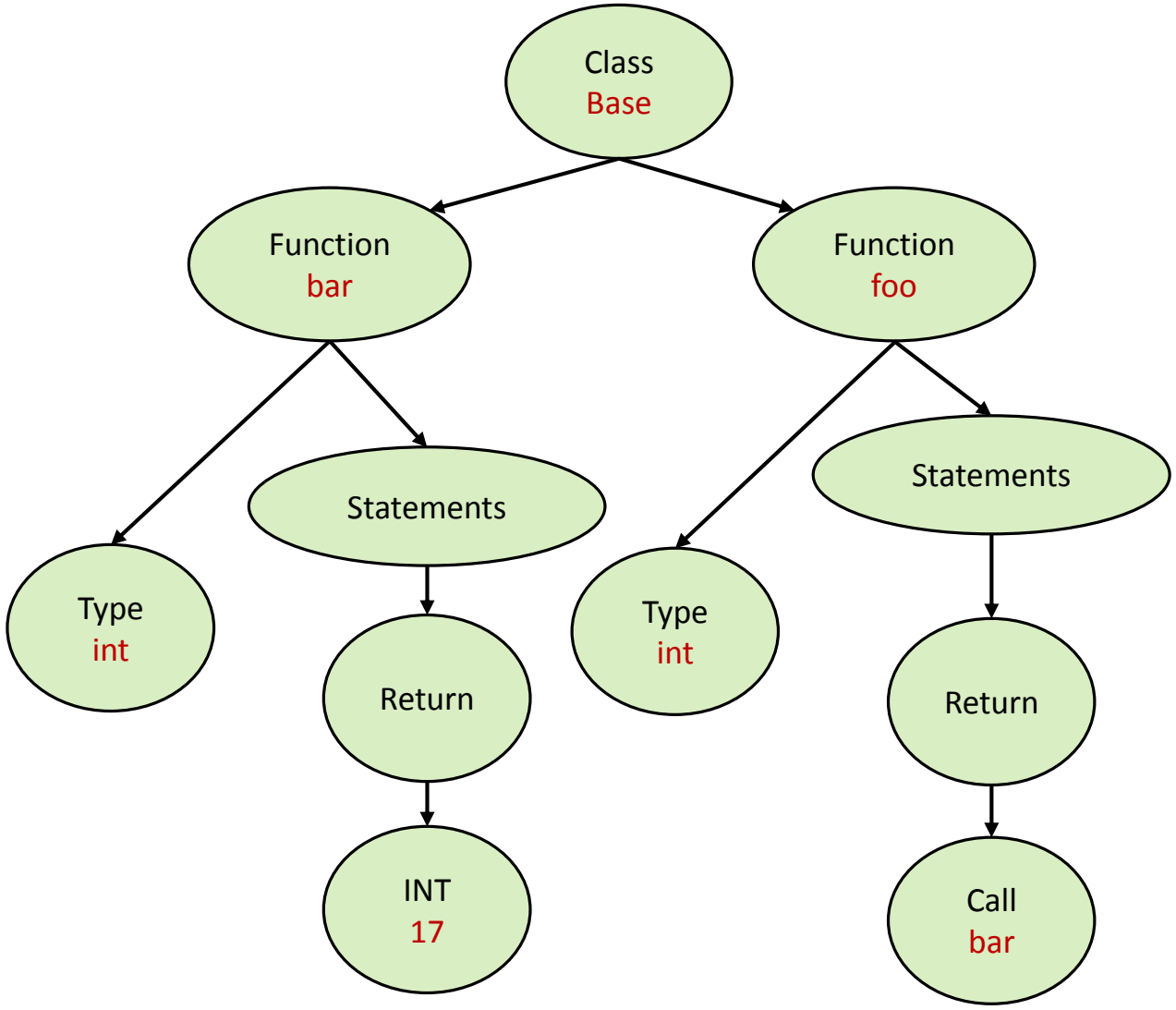
```
class Base {
  int a;
  int foo() {
    return a;
  }
}
```



Valid

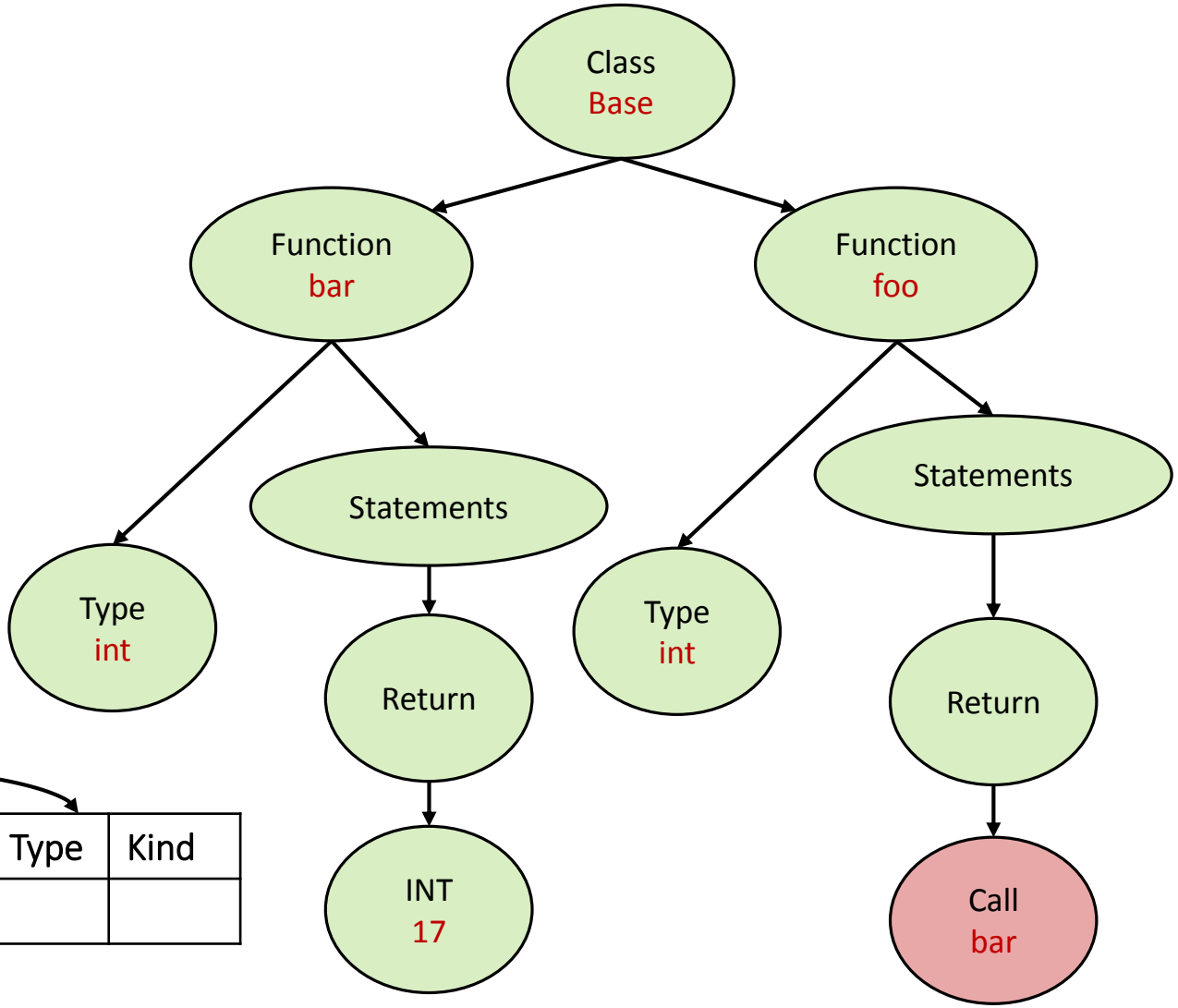
Classes

```
class Base {
  int bar() {
    return 17;
  }
  int foo() {
    return bar();
  }
}
```



Classes

```
class Base {
  int bar() {
    return 17;
  }
  int foo() {
    return bar();
  }
}
```



ID	Type	Kind
Base	…	class

$scope_1$

ID	Type	Kind
bar	…	function
foo	…	Function

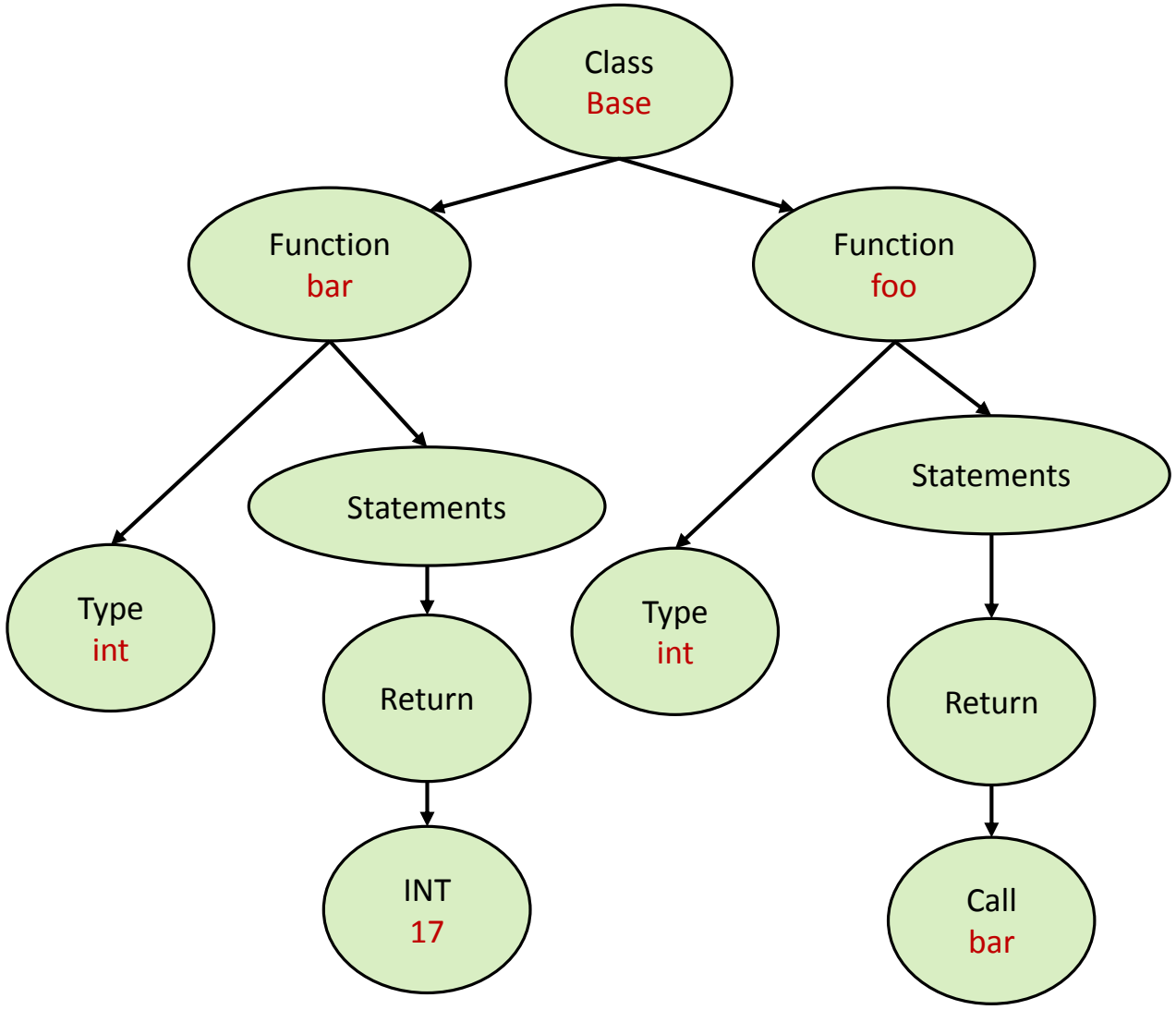
$scope_2$

ID	Type	Kind

$scope_3$

Classes

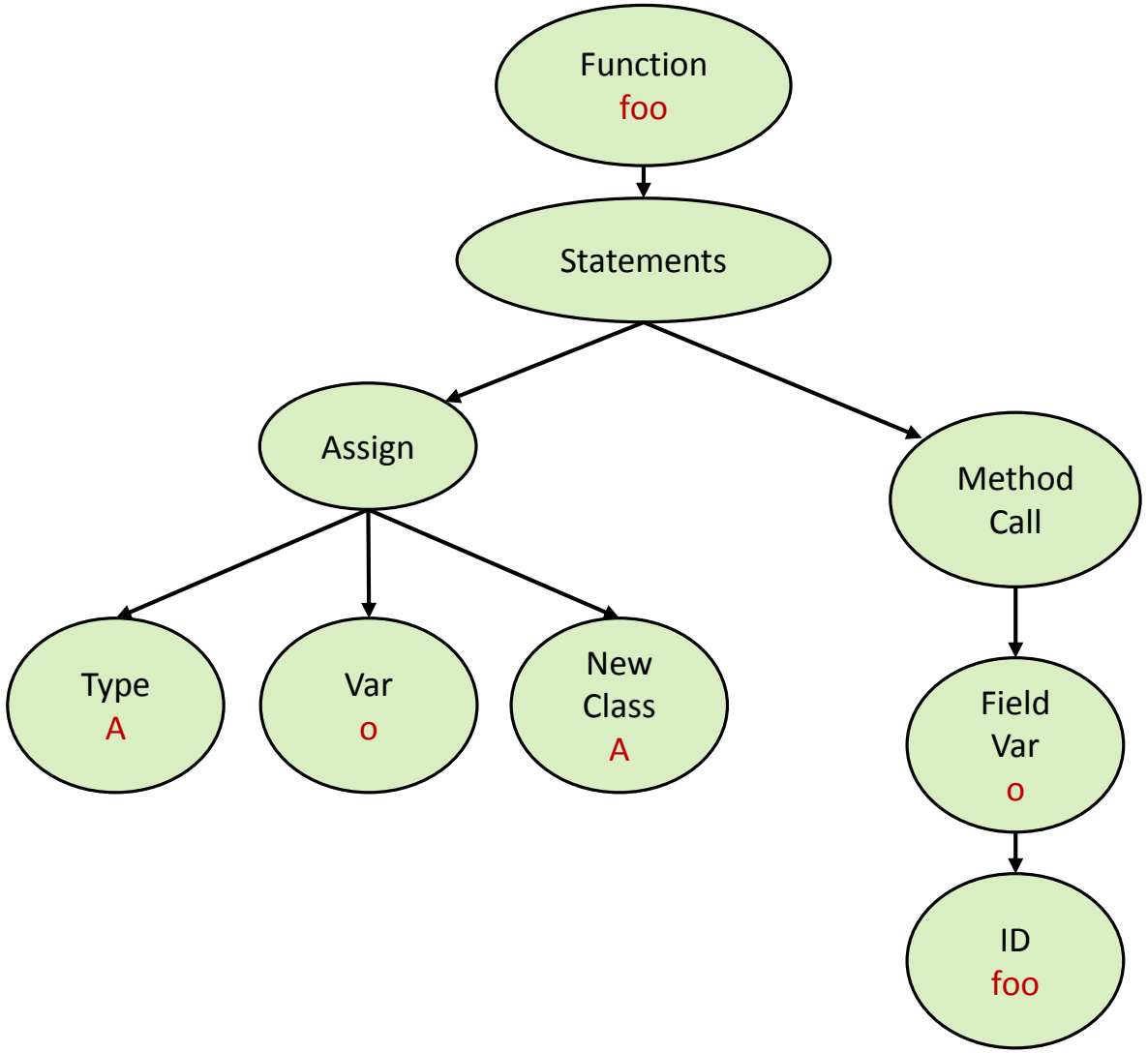
```
class Base {
  int bar() {
    return 17;
  }
  int foo() {
    return bar();
  }
}
```



Valid

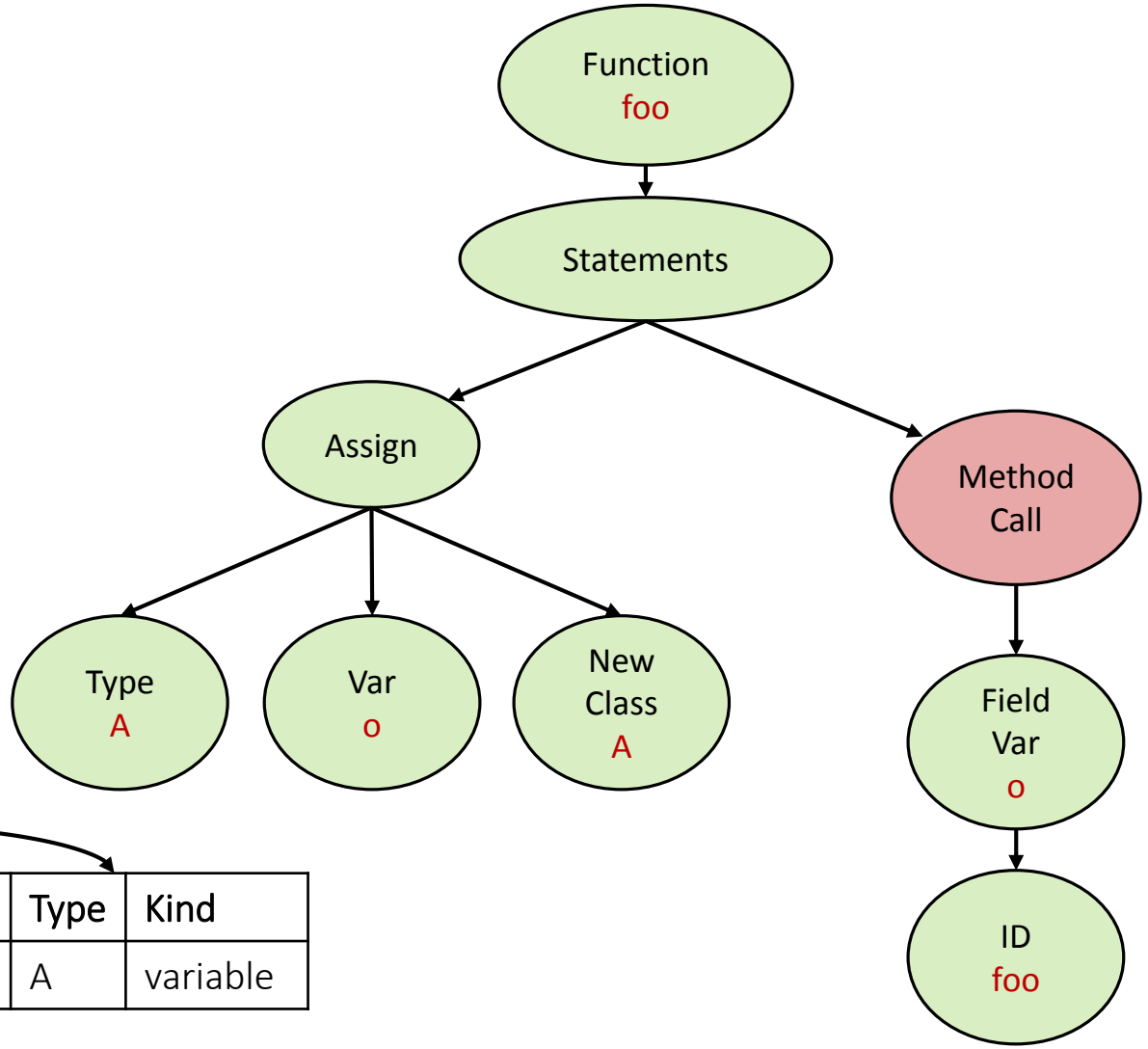
Classes

```
class A {
  void foo() {
    A o = new A;
    o.foo();
  }
}
```



Classes

```
class A {
  void foo() {
    A o = new A;
    o.foo();
  }
}
```



ID	Type	Kind
A	…	class

$scope_1$

ID	Type	Kind
foo	…	function

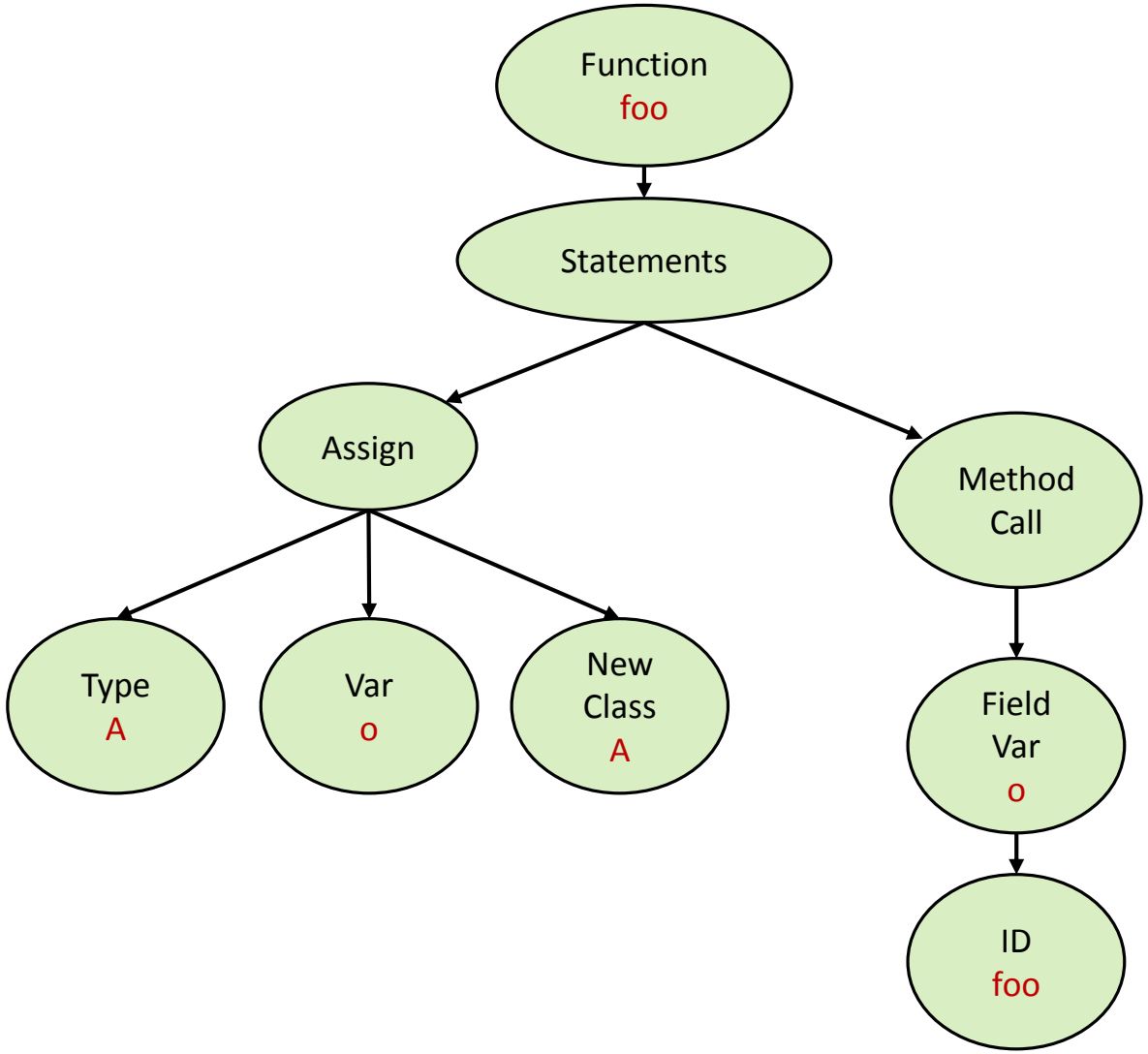
$scope_2$

ID	Type	Kind
o	A	variable

$scope_3$

Classes

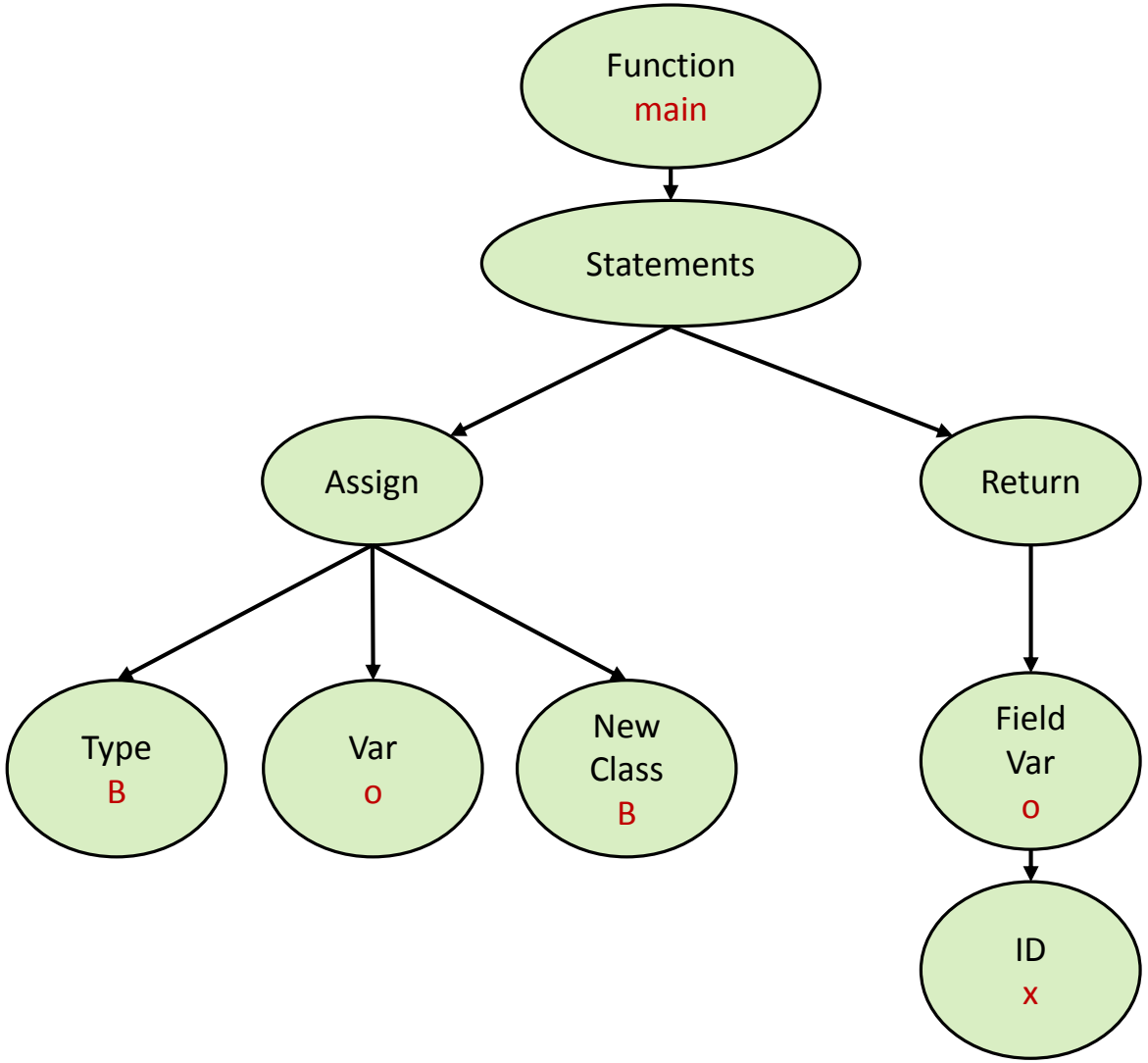
```
class A {
  void foo() {
    A o = new A;
    o.foo();
  }
}
```



Valid

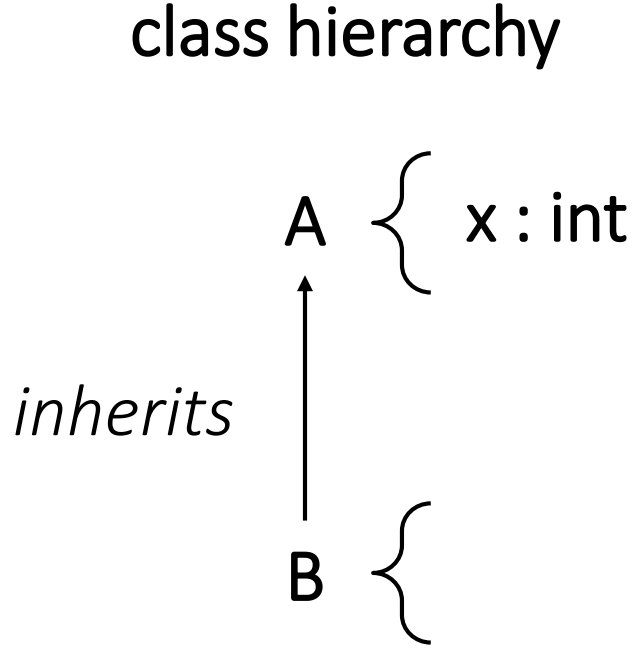
Inheritance

```
class A {
  int x;
}
class B extends A {
  int foo() {
    B o = new B;
    return o.x;
  }
}
```



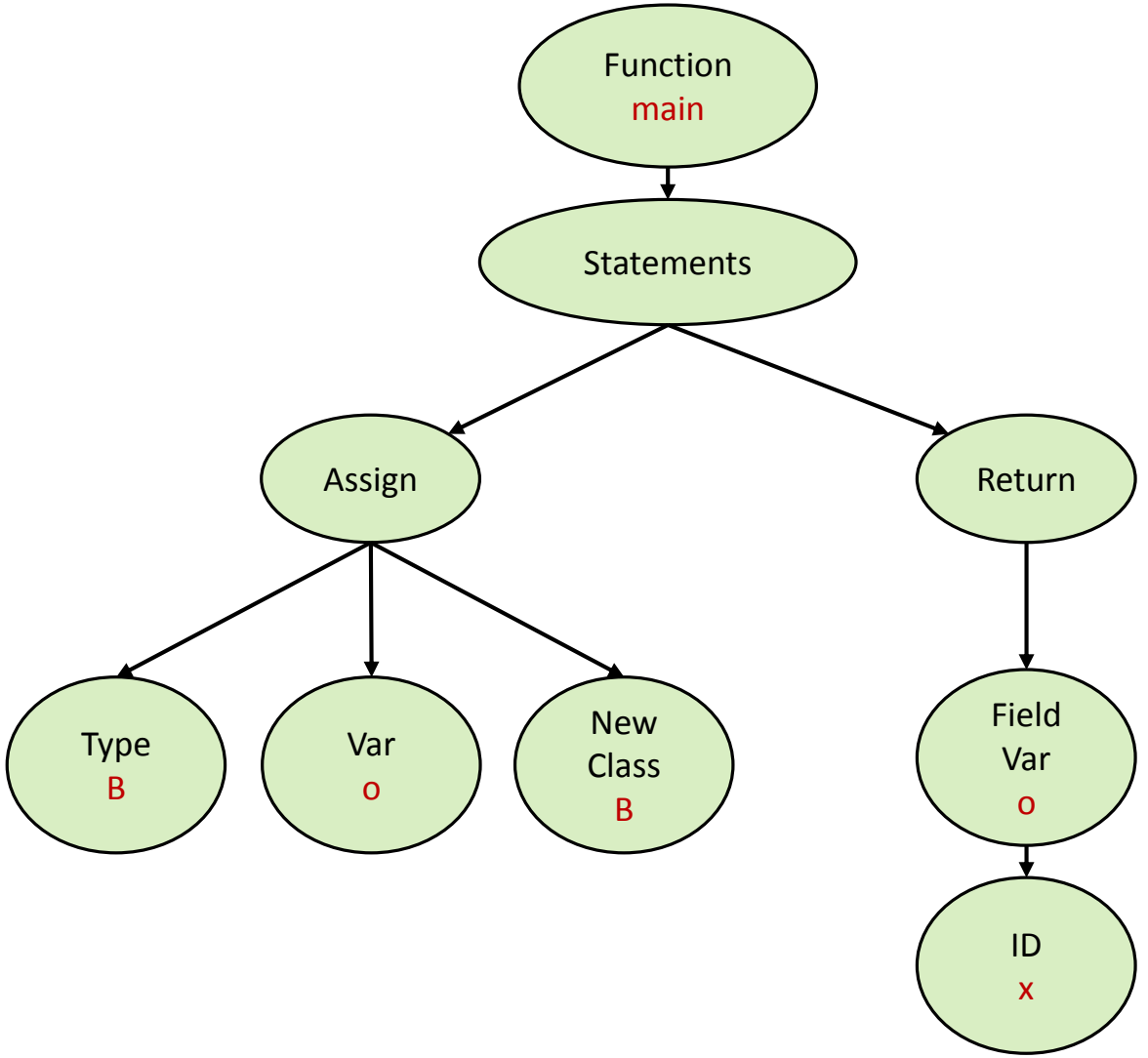
Inheritance

```
class A {
  int x;
}
class B extends A {
  int foo() {
    B o = new B;
    return o.x;
  }
}
```



Inheritance

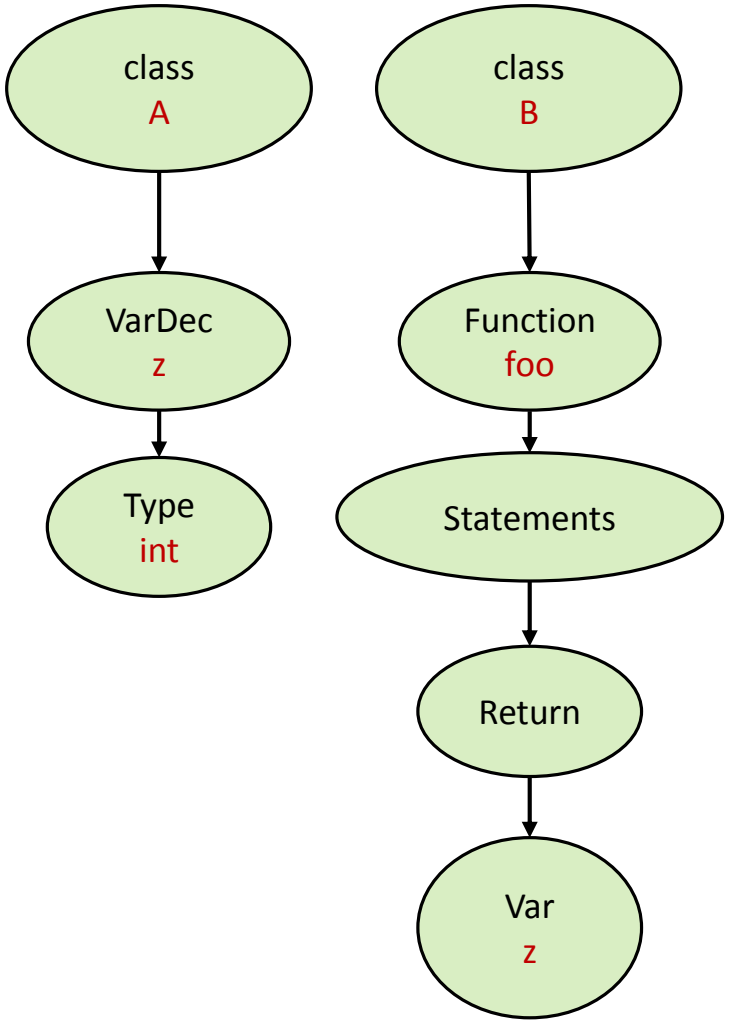
```
class A {
  int x;
}
class B extends A {
  int foo() {
    B o = new B;
    return o.x;
  }
}
```



Valid

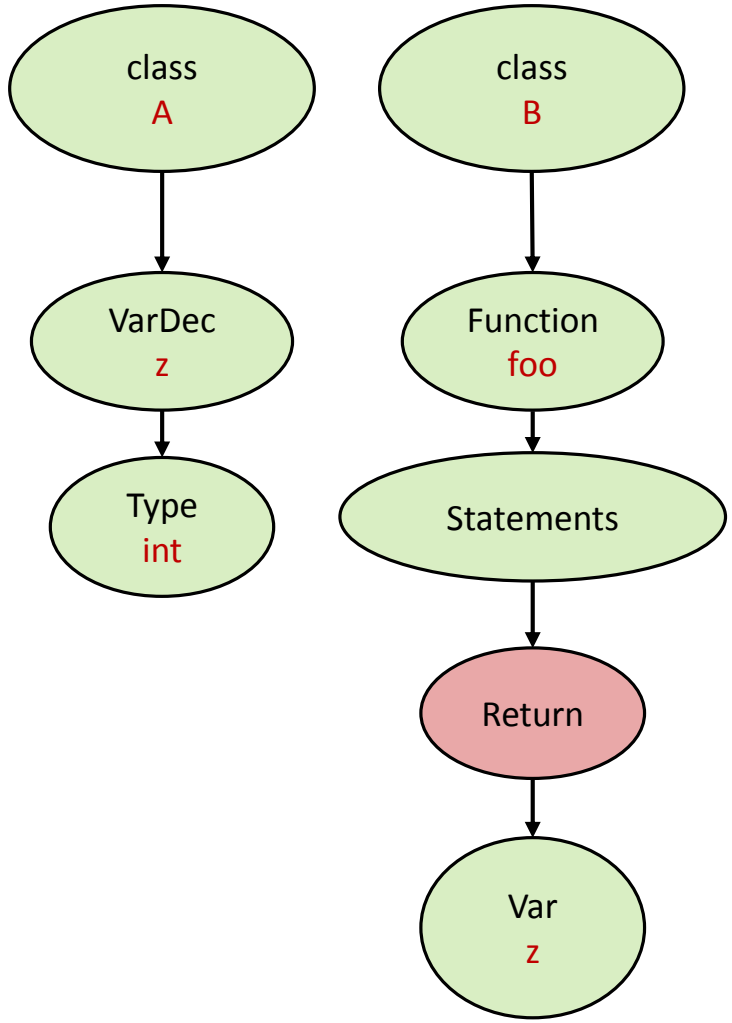
Inheritance

```
class A {
  int z;
}
class B extends A {
  int foo() {
    return z;
  }
}
```



Inheritance

```
class A {
  int z;
}
class B extends A {
  int foo() {
    return z;
  }
}
```



ID	Type	Kind
A	…	class
B	…	class

$scope_1$

ID	Type	Kind
foo	…	function

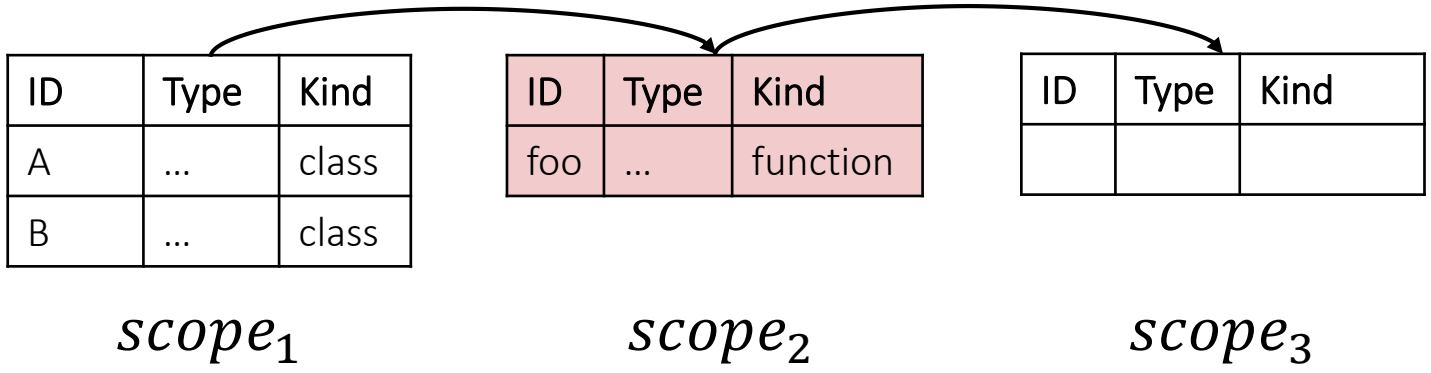
$scope_2$

ID	Type	Kind

$scope_3$

Inheritance

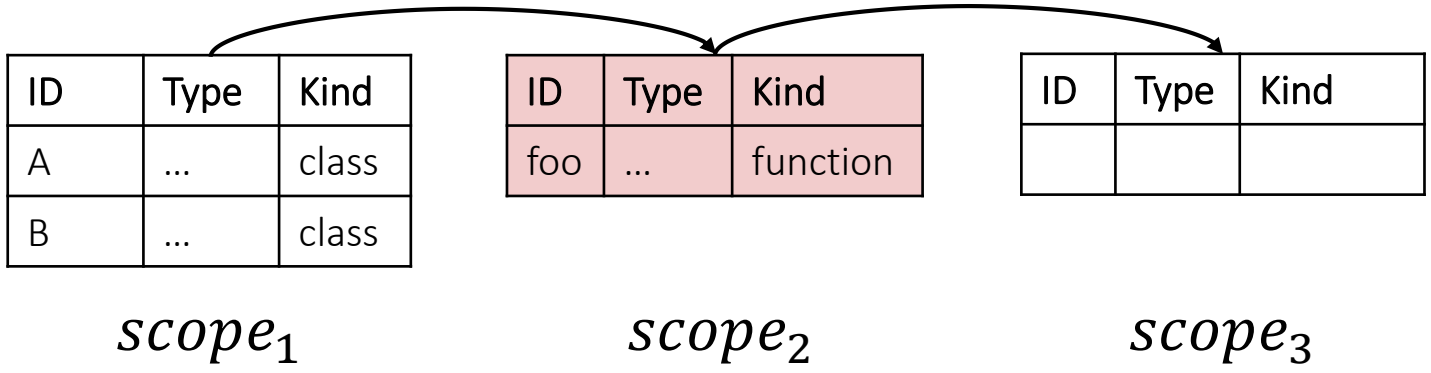
```
class A {
  int z;
}
class B extends A {
  int foo() {
    return z;
  }
}
```



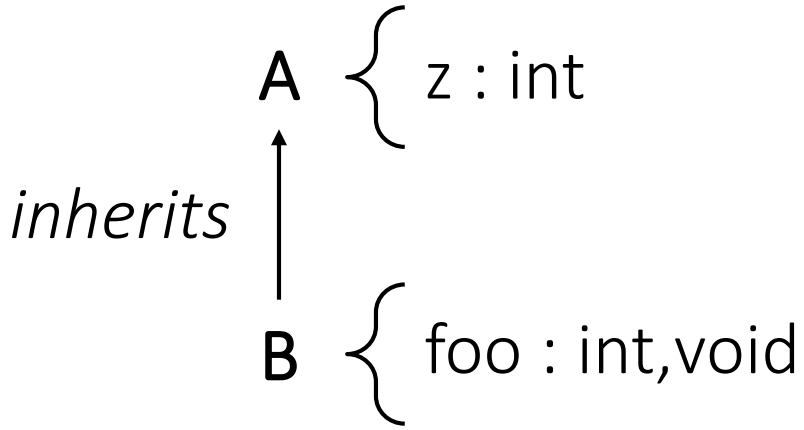
(scope of class B)

Inheritance

```
class A {
  int z;
}
class B extends A {
  int foo() {
    return z;
  }
}
```

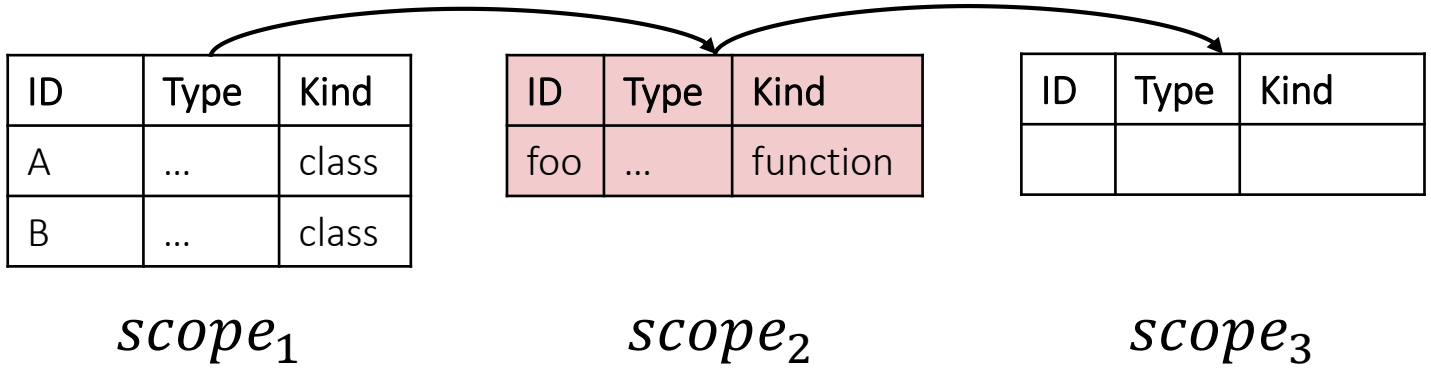


(scope of class B)

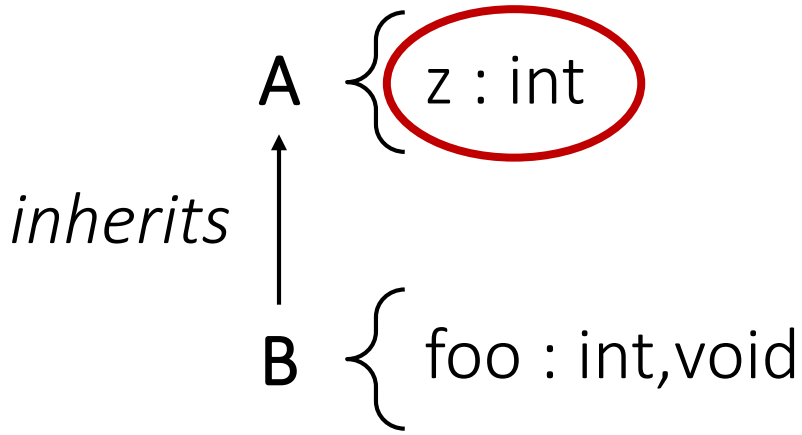


Inheritance

```
class A {
  int z;
}
class B extends A {
  int foo() {
    return z;
  }
}
```

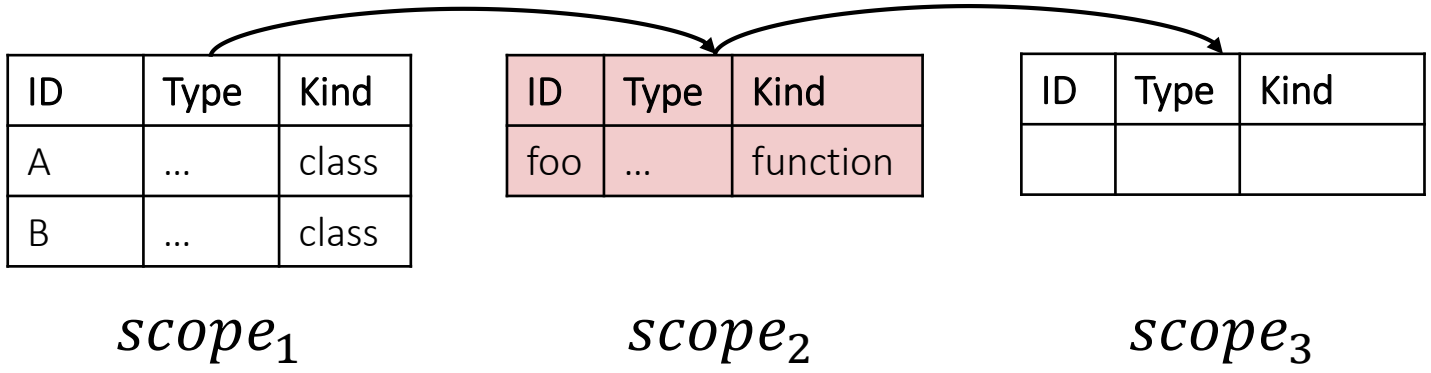


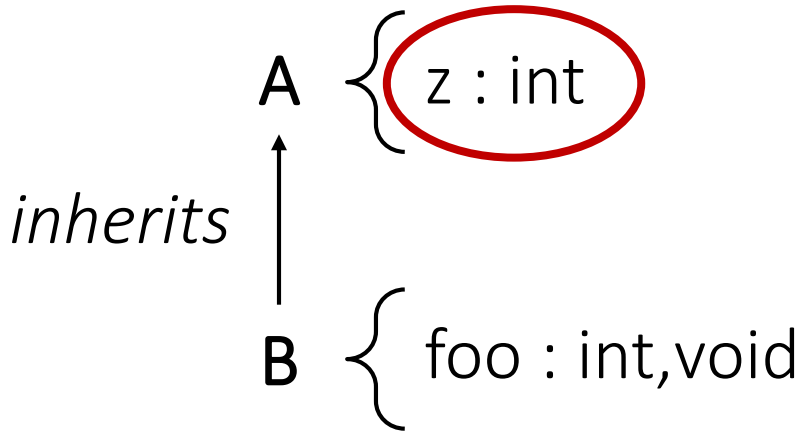
(scope of class B)



Inheritance

```
class A {
  int z;
}
class B extends A {
  int foo() {
    return z;
  }
}
```

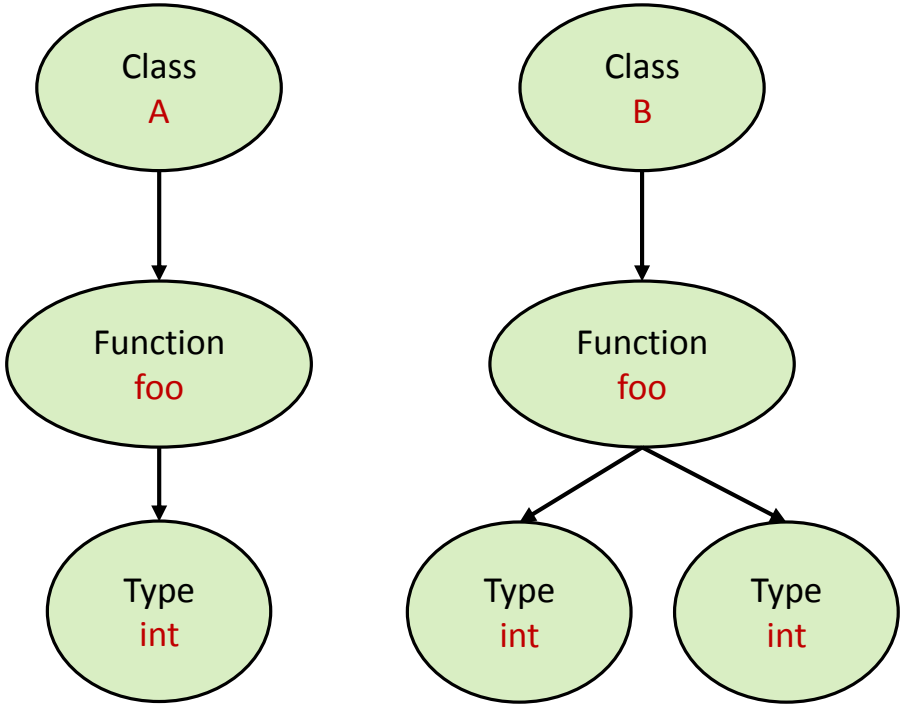




Valid

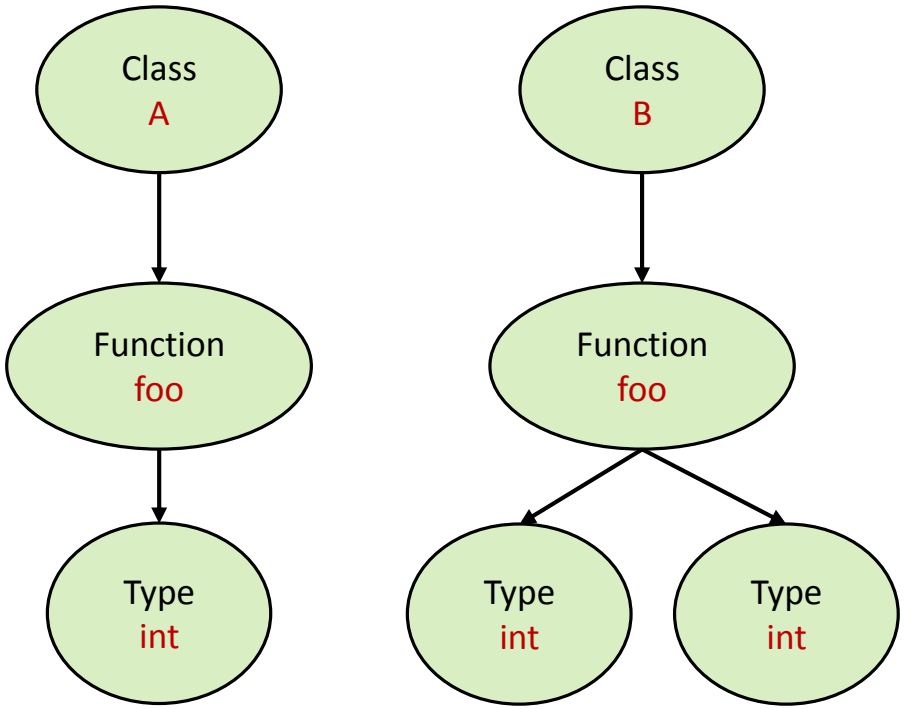
Inheritance

```
class A {
  int foo() {
    return 17;
  }
}
class B extends A {
  int foo(int x) {
    return x + 1;
  }
}
```



Inheritance

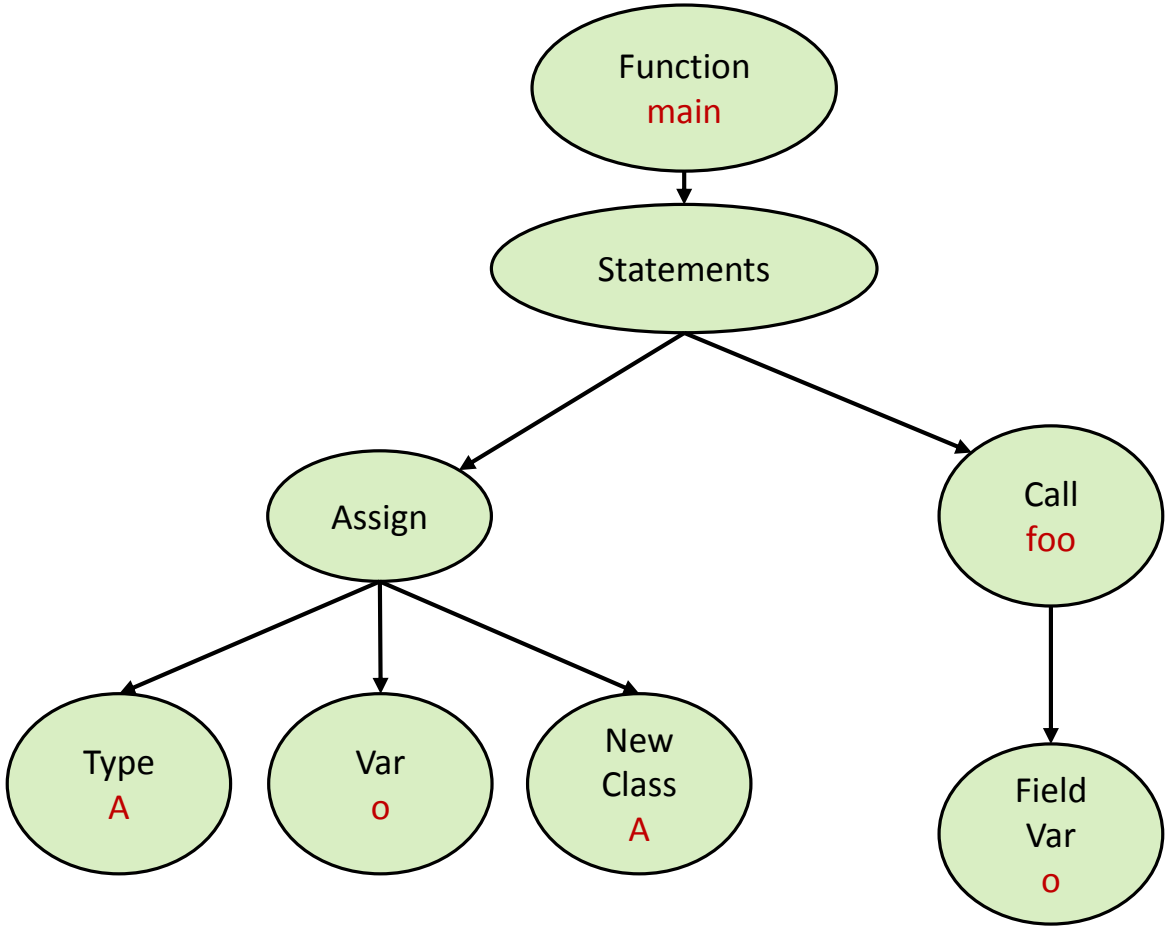
```
class A {
  int foo() {
    return 17;
  }
}
class B extends A {
  int foo(int x) {
    return x + 1;
  }
}
```



Invalid

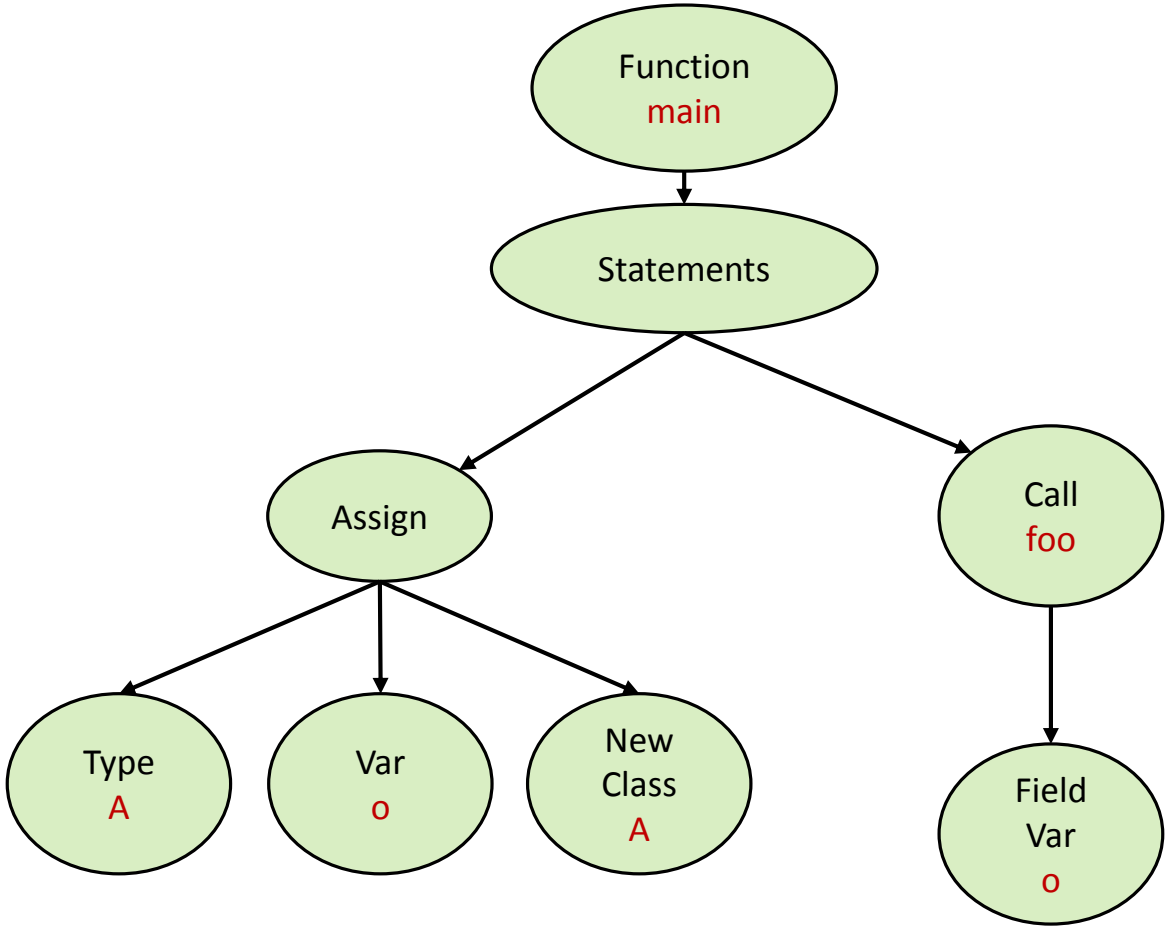
Inheritance

```
class A { }
class B extends A { }
void foo(B b) { }
void main() {
  A o = new A;
  foo(o);
}
```



Inheritance

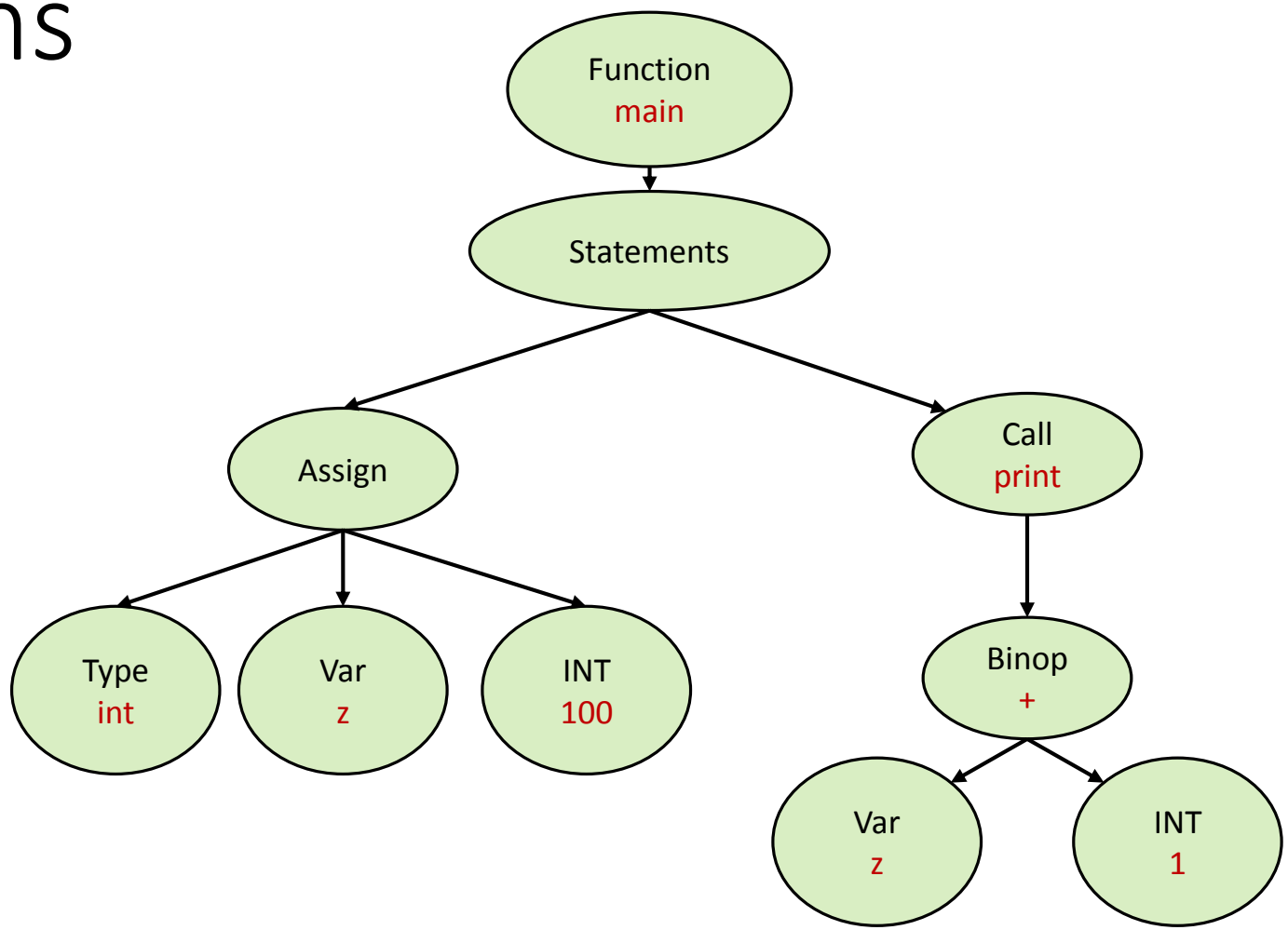
```
class A { }
class B extends A { }
void foo(B b) { }
void main() {
  A o = new A;
  foo(o);
}
```



Invalid

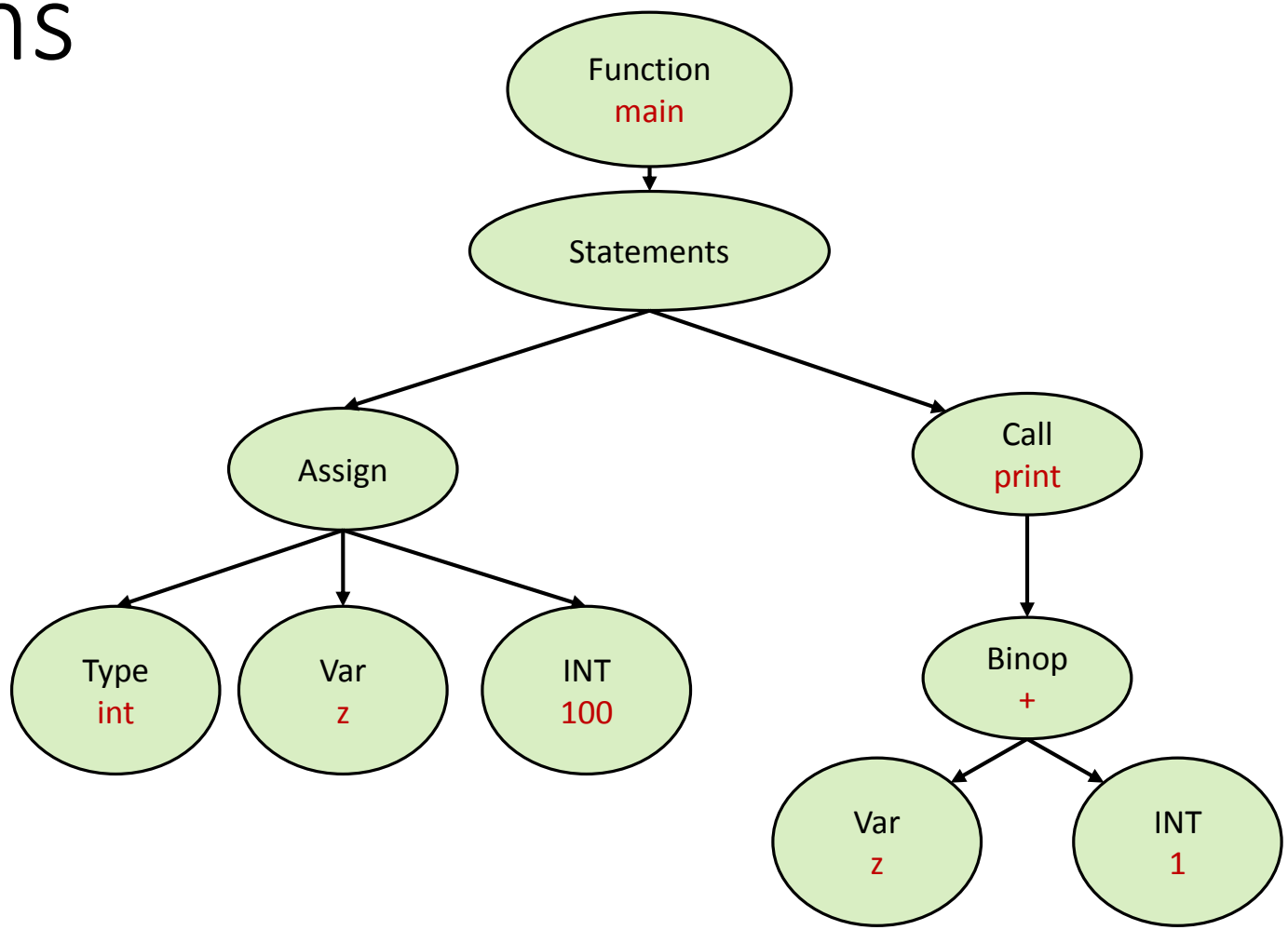
Library Functions

```
void main() {
  int z = 100;
  print(z + 1);
}
```



Library Functions

```
void main() {
  int z = 100;
  print(z + 1);
}
```



Valid

Implementation

The AST is traversed in a top-down manner:

- Each AST node class, has a **visit** API
  - Performs the relevant semantic checks
  - May call the visitors of the node's children
- The traversal starts from the root node

Implementation

```
class ASTExpBinOp {
  public ASTExp left;
  public ASTExp right;
  …
  public Type visit() {
    Type t1 = left.visit();
    Type t2 = right.visit();
    if (t1 != t2) { // error }
    // check if op is supported w.r.t. t1/t2
    return t1;
  }
}
```

Implementation

```
class ASTStatmentList {
  public ASTStatement head;
  public ASTStatmentList tail;
  …
  public Type visit() {
    if (head)
      head.visit();
    if (tail)
      tail.visit();
    return null;
  }
}
```