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, …)

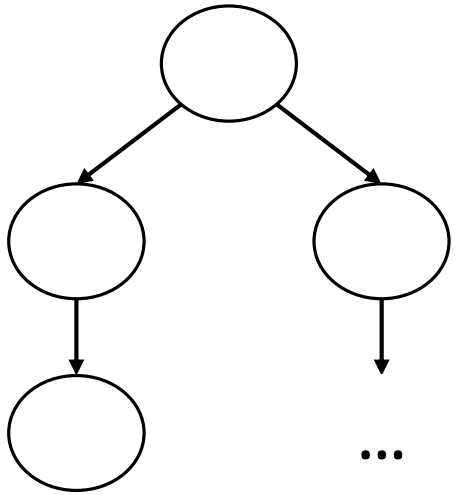
Visitor Design Pattern

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

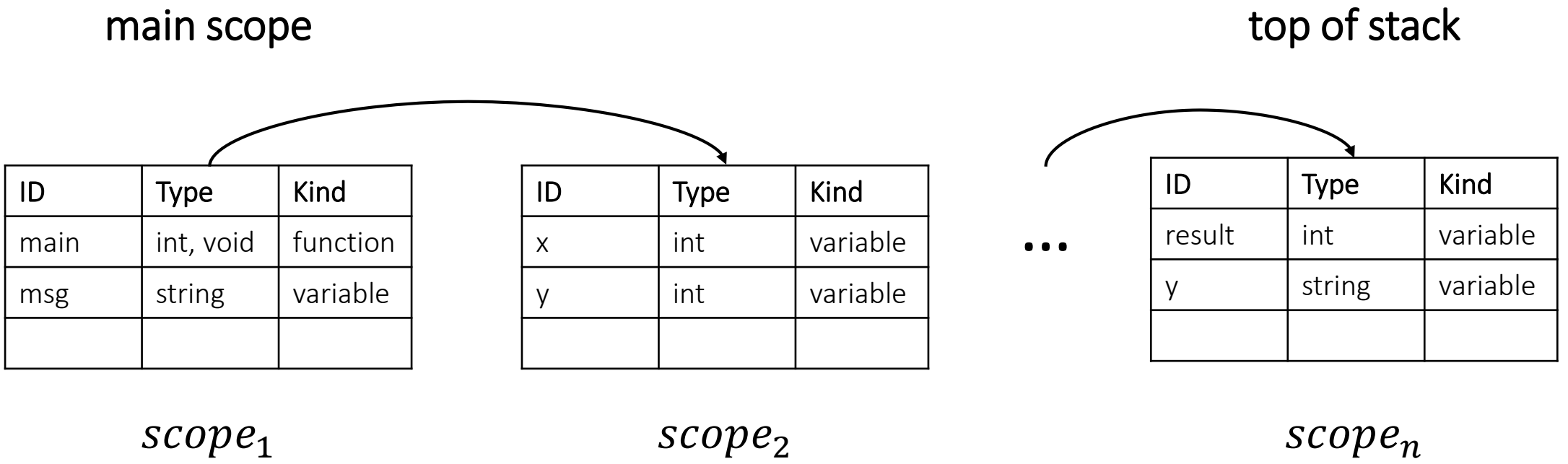
Printing the AST

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

Symbol Table

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

Symbol Table



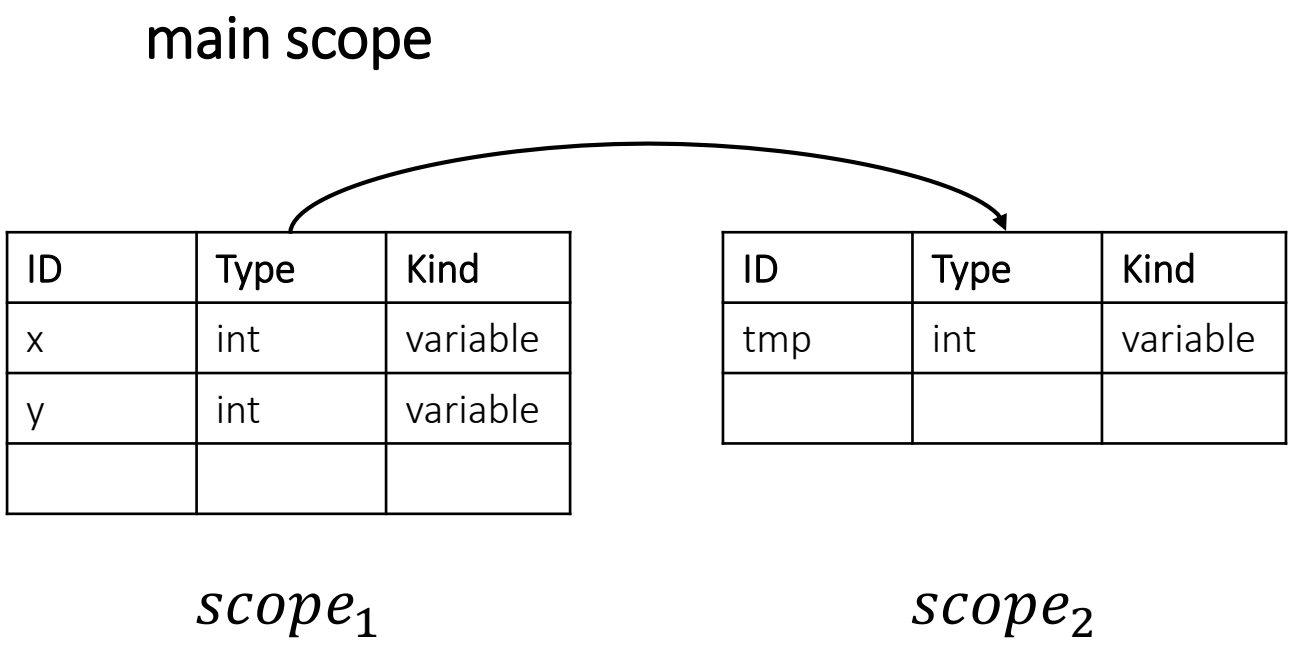
Symbol Table Operations

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

Symbol Table: Insert

Example:

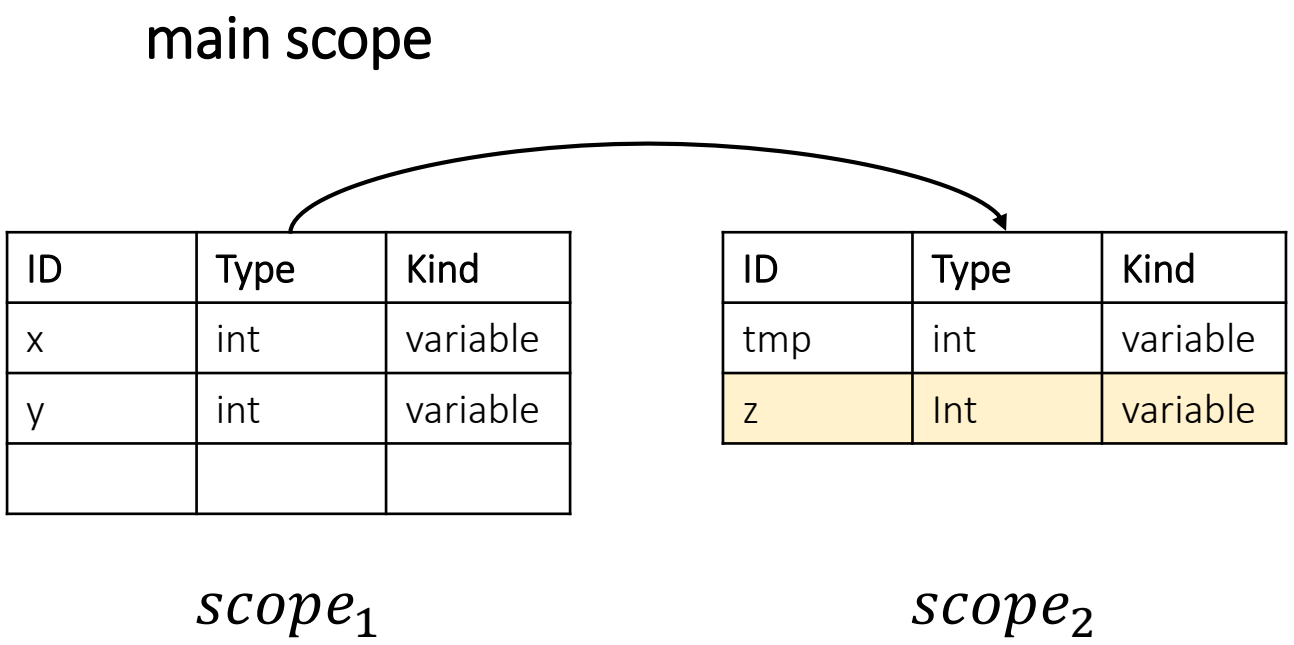
- Insert(z, int, variable)



Symbol Table: Insert

Example:

- Insert(z, int, variable)

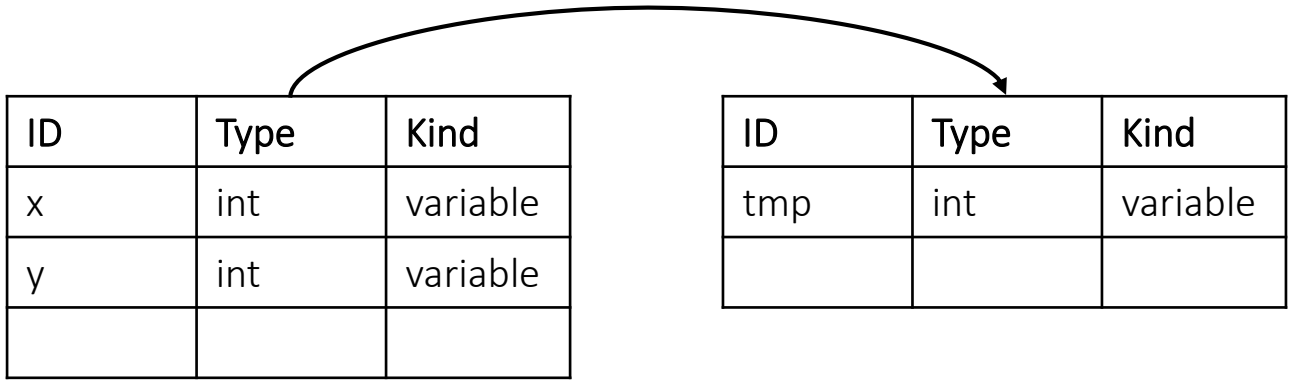


Symbol Table: Lookup

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

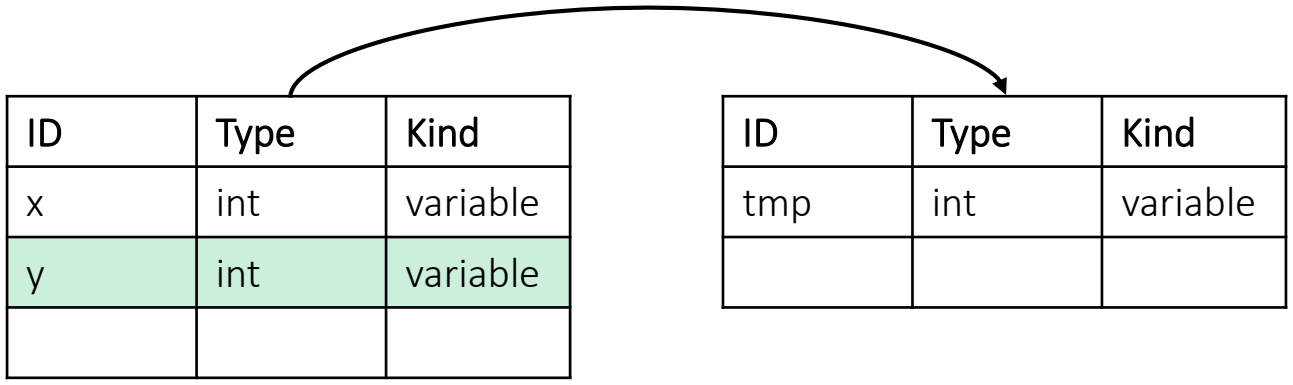
$scope_2$

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

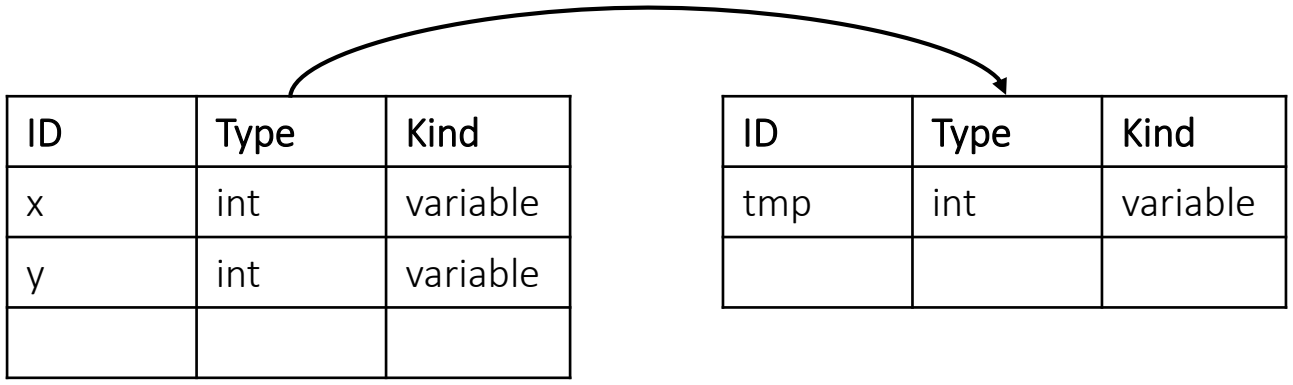
$scope_1$

ID	Type	Kind
tmp	int	variable

$scope_2$

Symbol Table: Enter

main scope



ID	Type	Kind
x	int	variable
y	int	variable

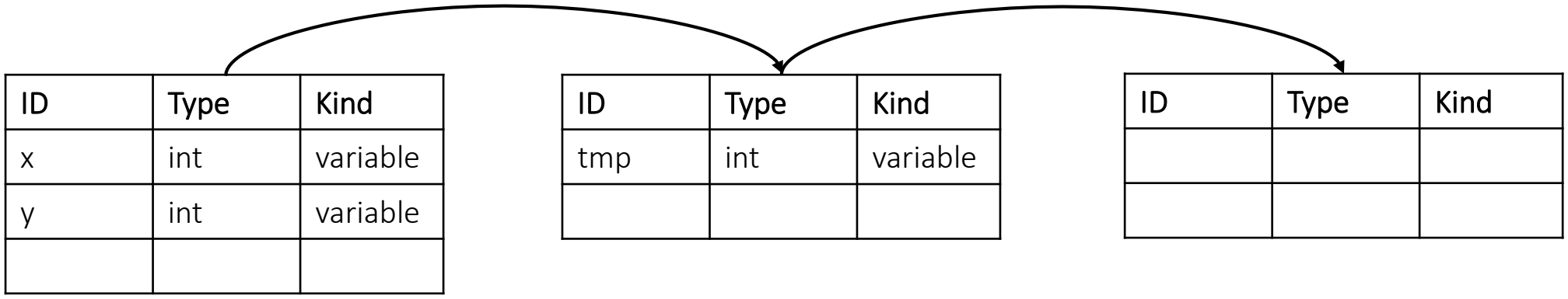
ID	Type	Kind
tmp	int	variable

$scope_1$

$scope_2$

Symbol Table: Enter

main scope



ID	Type	Kind
x	int	variable
y	int	variable

$scope_1$

ID	Type	Kind
tmp	int	variable

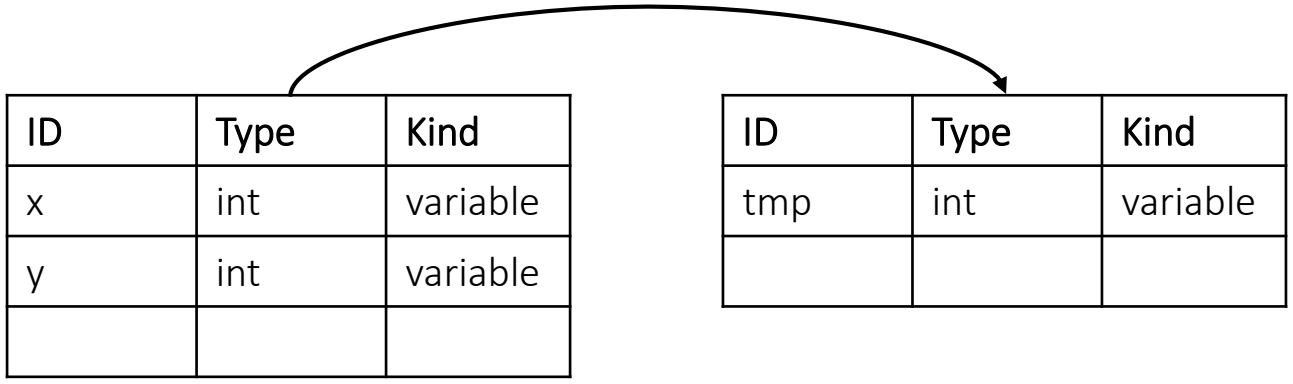
$scope_2$

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

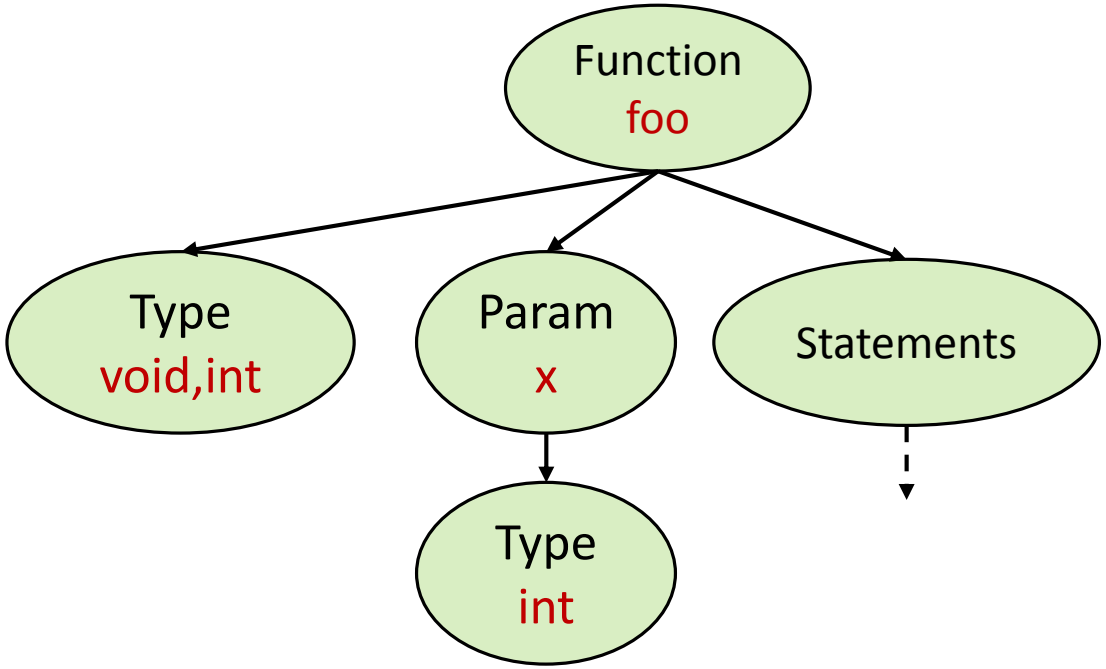
ID	Type	Kind
x	int	variable
y	int	variable

$scope_1$

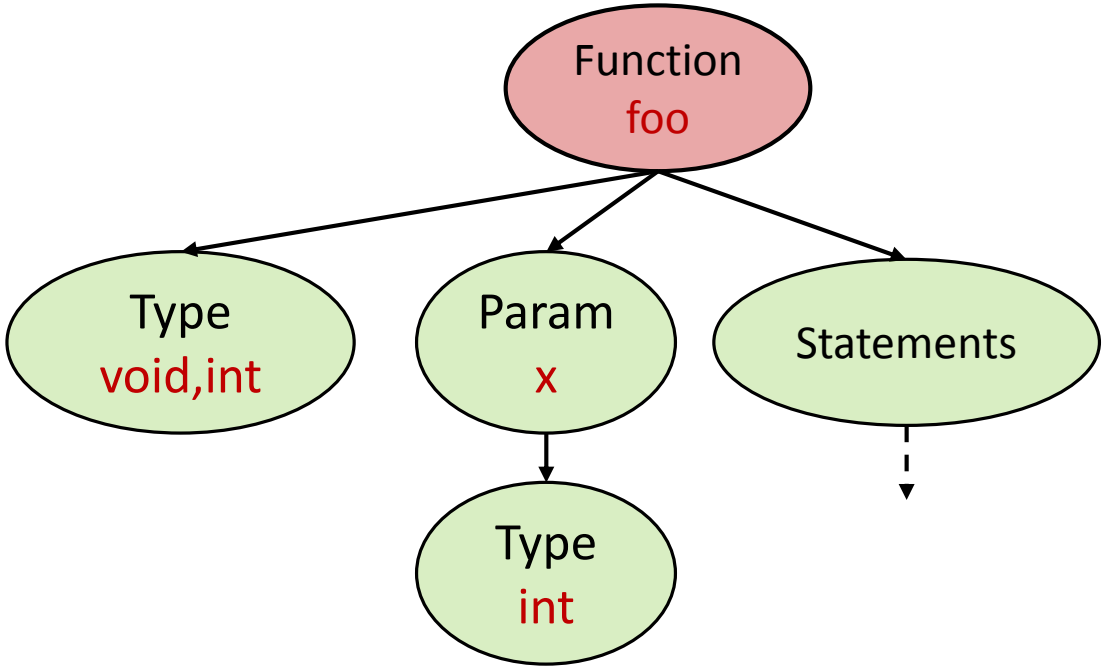
Symbol Table Construction

- 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";
  }
}
```

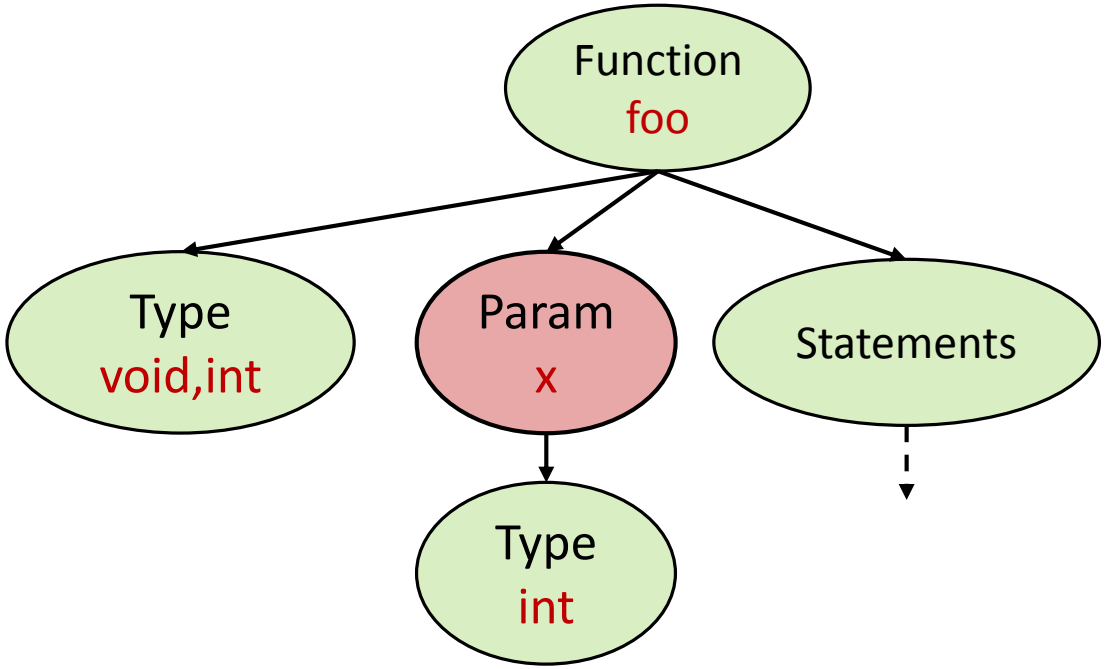


```
void foo(int x) {
  int a = 1;
  int b = 2;
  if (x) {
    string a = "abc";
  }
}
```



ID	Type	Kind
foo	void,int	function

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



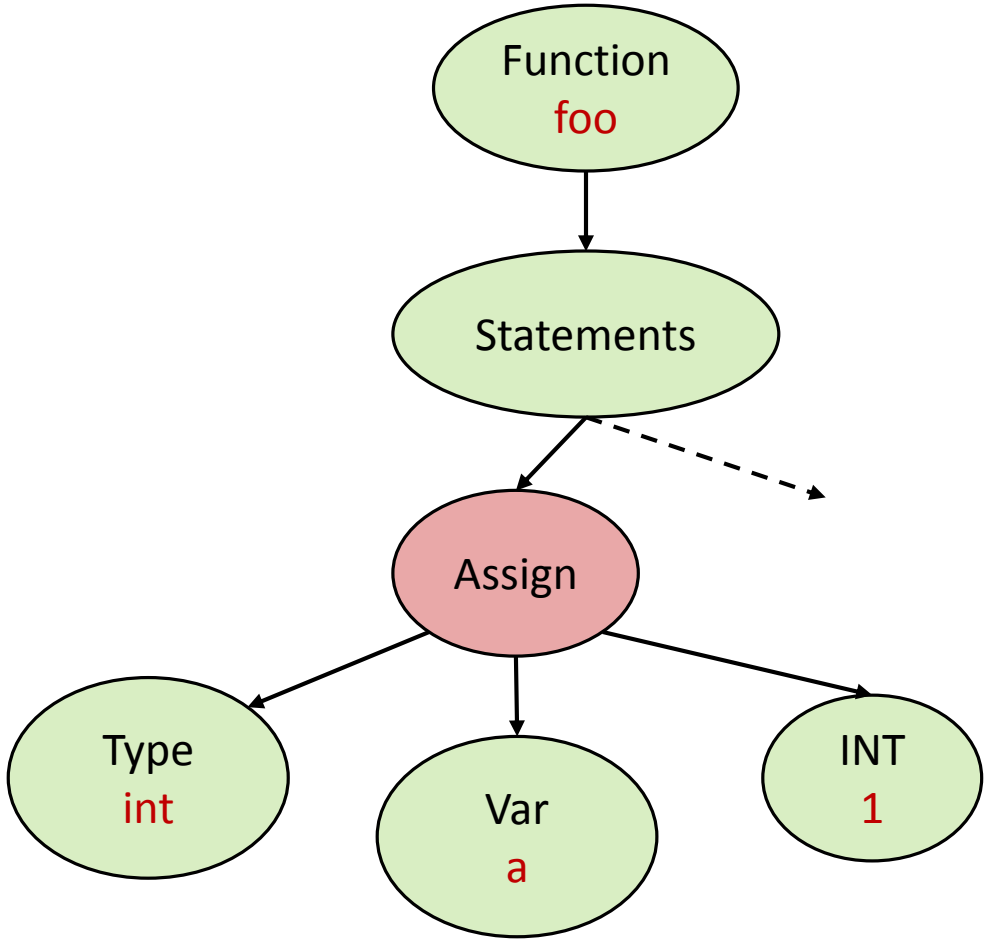
ID	Type	Kind
foo	void,int	function

ID	Type	Kind
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

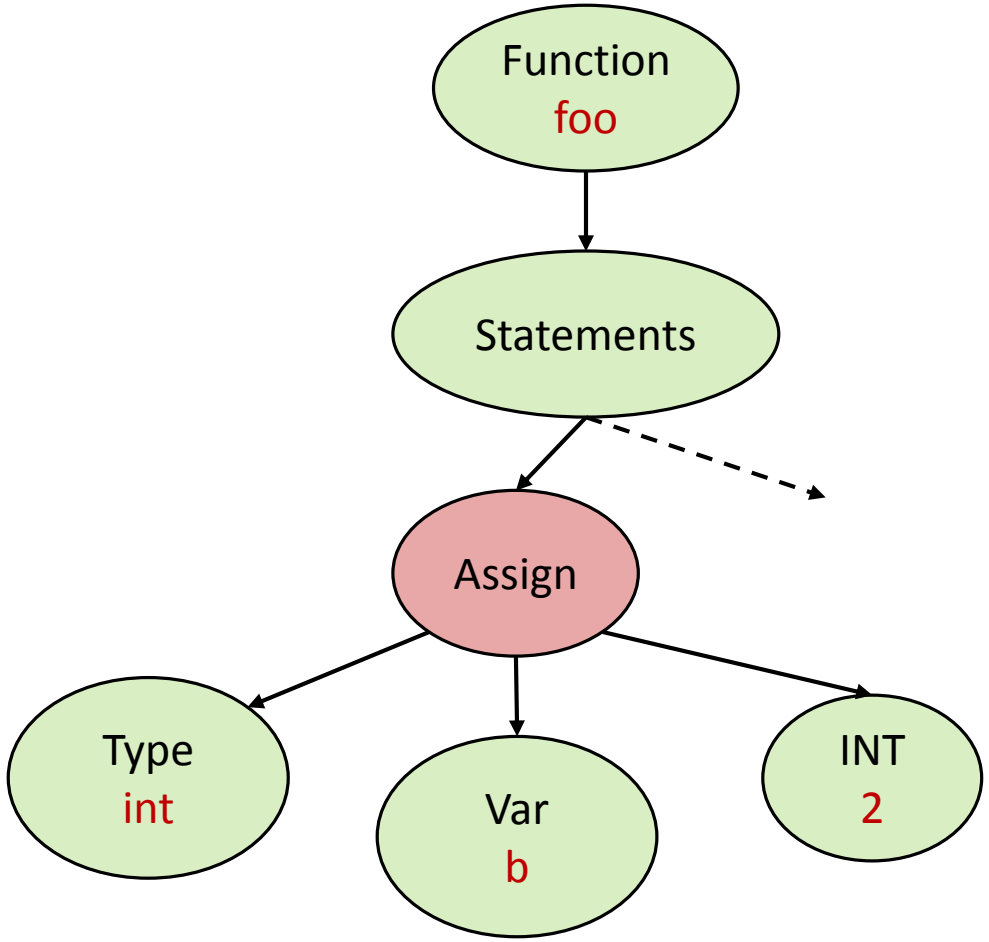
ID	Type	Kind
x	int	variable
a	int	variable



```
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

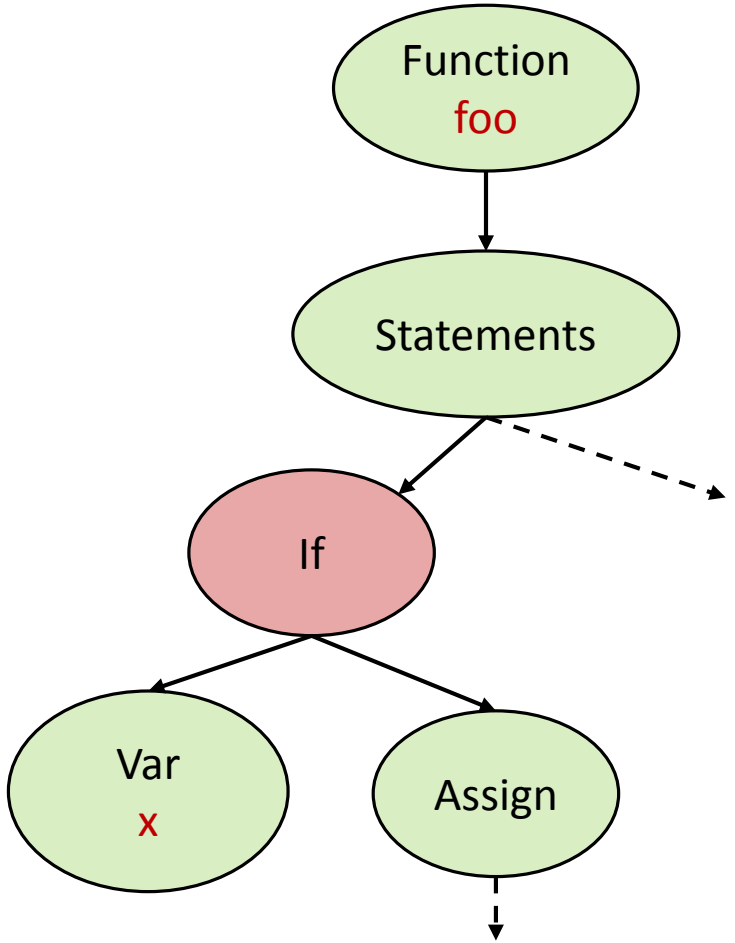


```
void foo(int x) {
  int a = 1;
  int b = 2;
  if (x) {
    string a = "abc";
  }
}
```

ID	Type	Kind
foo	void,int	function

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

ID	Type	Kind

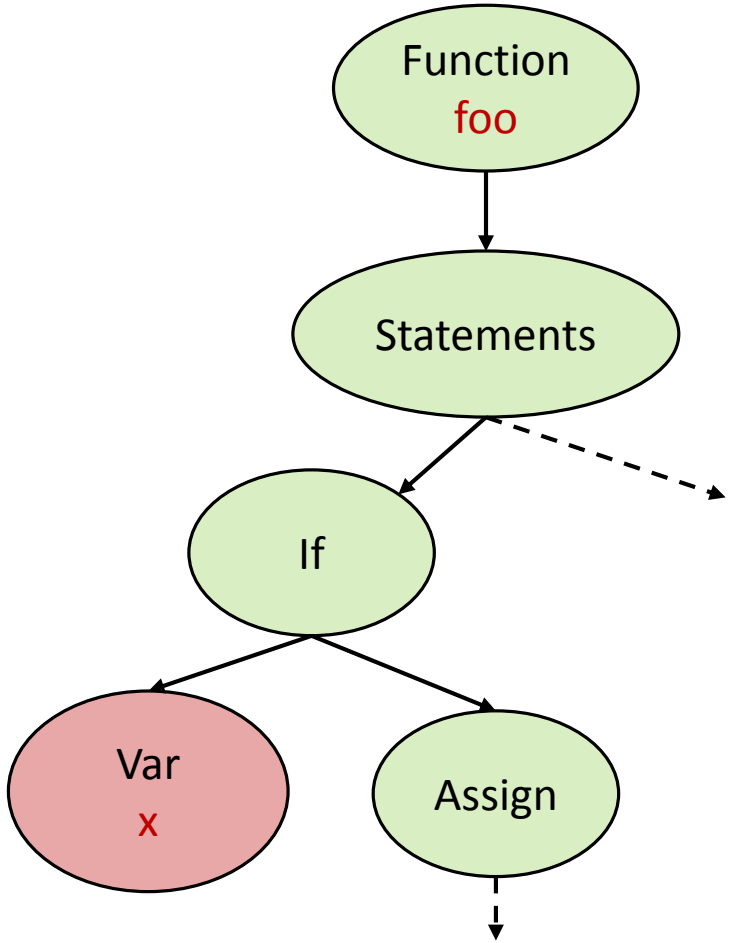


```
void foo(int x) {
  int a = 1;
  int b = 2;
  if (x) {
    string a = "abc";
  }
}
```

ID	Type	Kind
foo	void,int	function

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

ID	Type	Kind

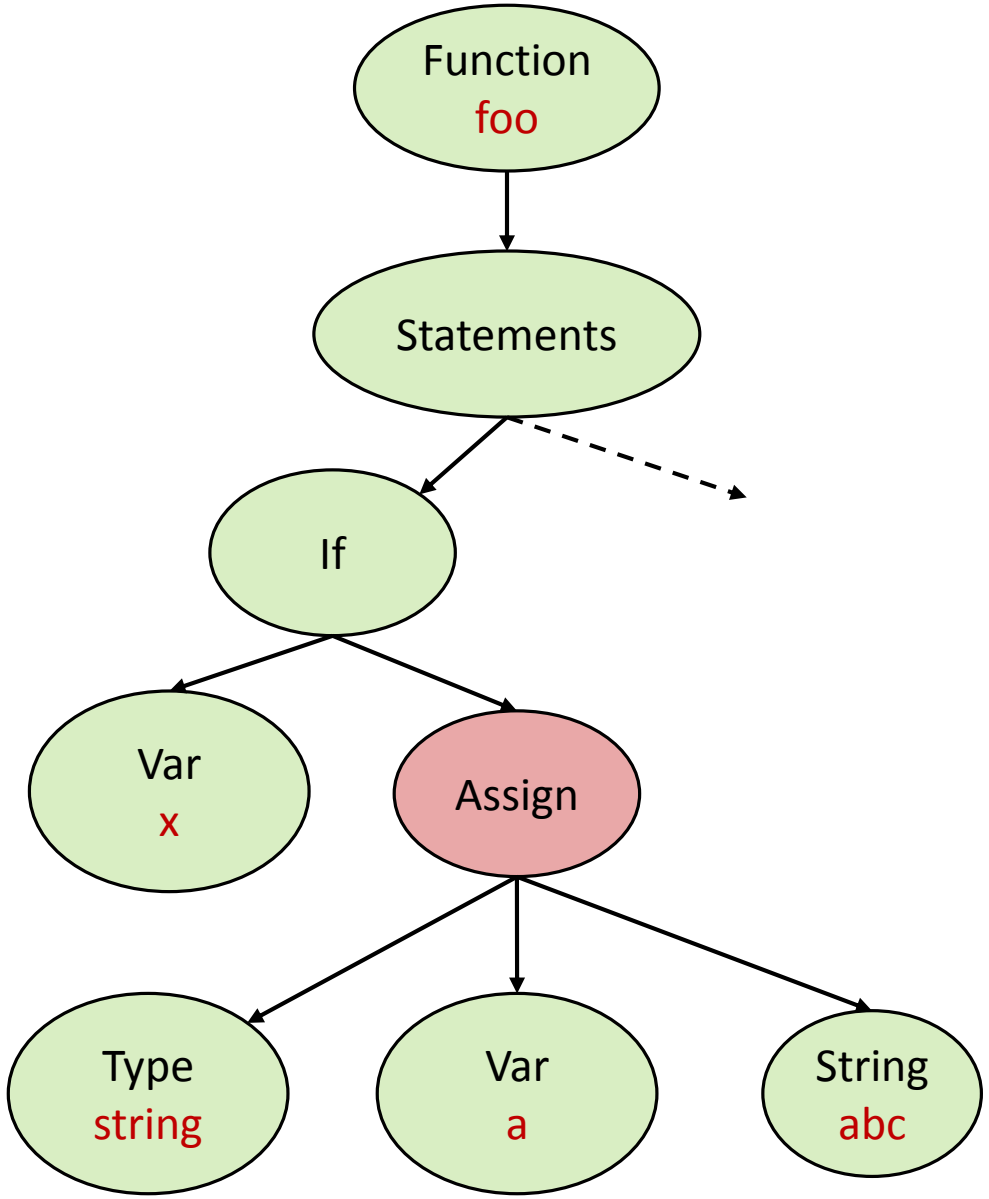


```
void foo(int x) {
  int a = 1;
  int b = 2;
  if (x) {
    string a = "abc";
  }
}
```

ID	Type	Kind
foo	void,int	function

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

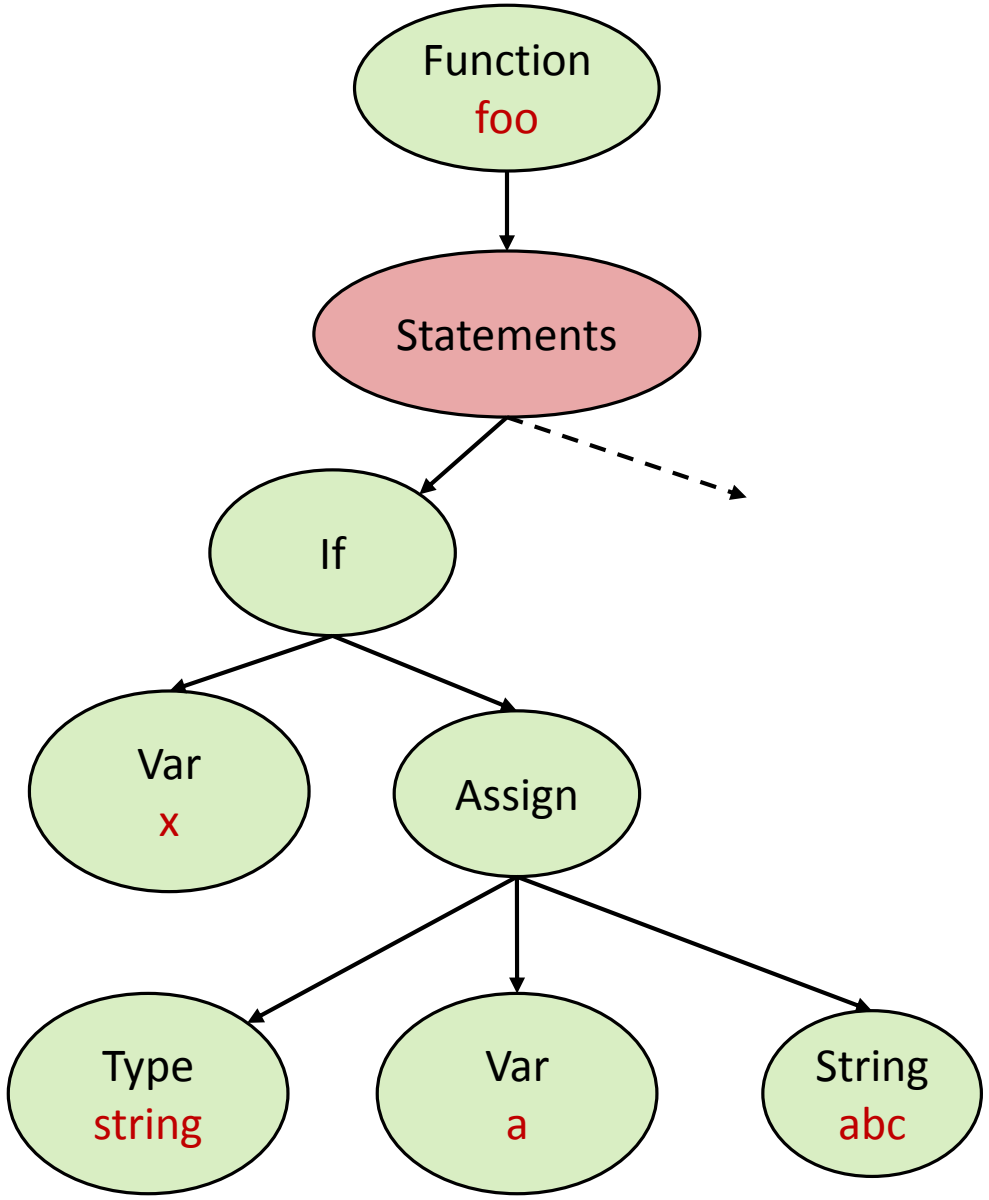
ID	Type	Kind
a	string	variable



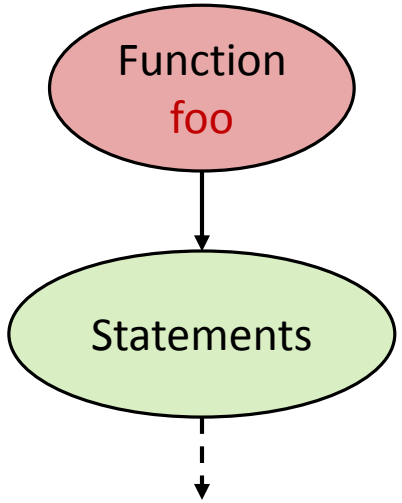
```
void foo(int x) {
  int a = 1;
  int b = 2;
  if (x) {
    string a = "abc";
  }
}
```

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:

$visit(node)$:

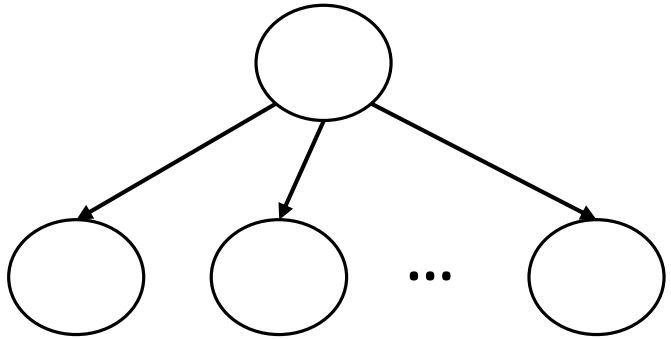
$\mathrm{t}_1 = visit(node.child_1)$

…

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

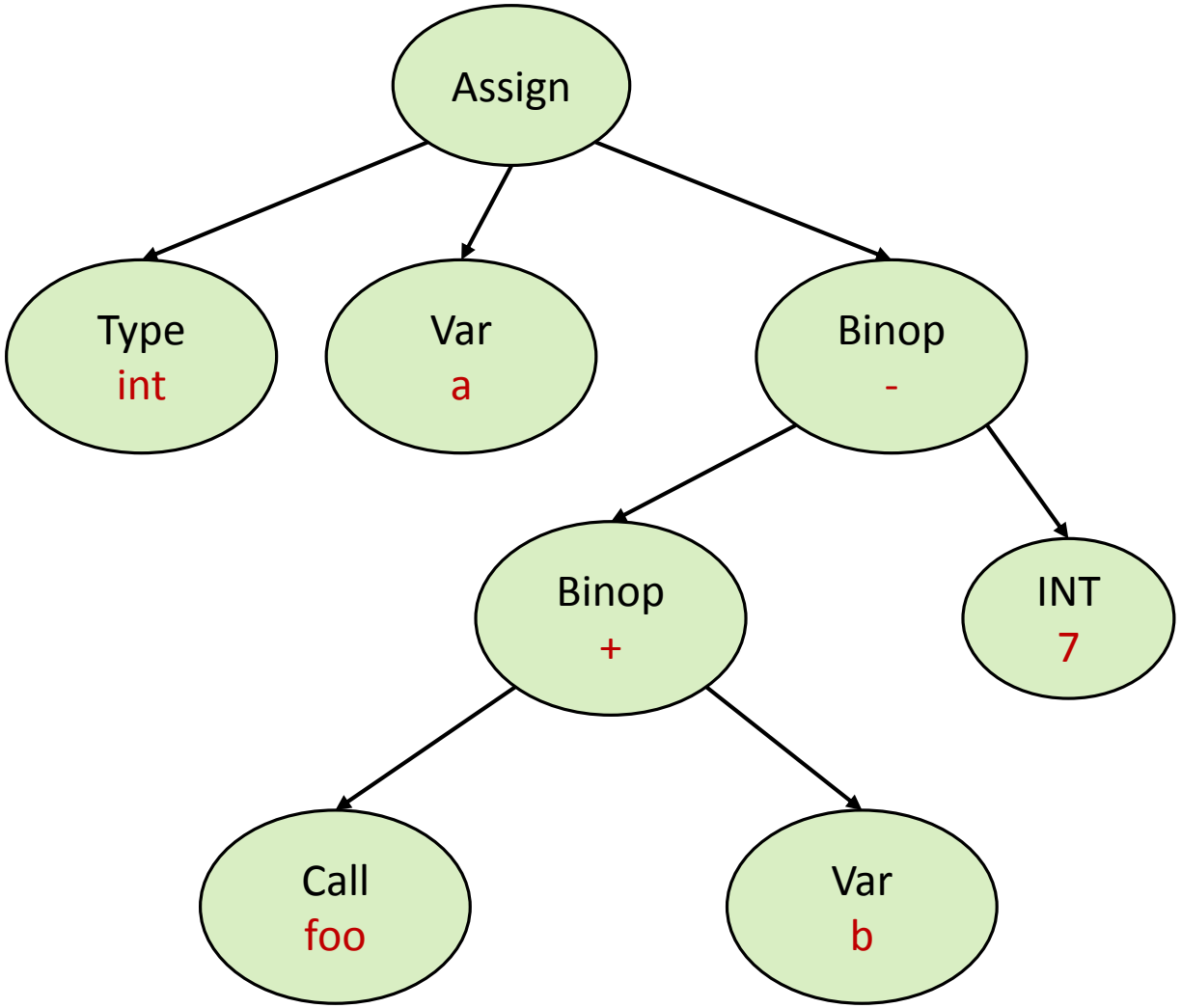
$return \underbrace{compute\_type(t_1, \ldots, t_n)}$

node specific



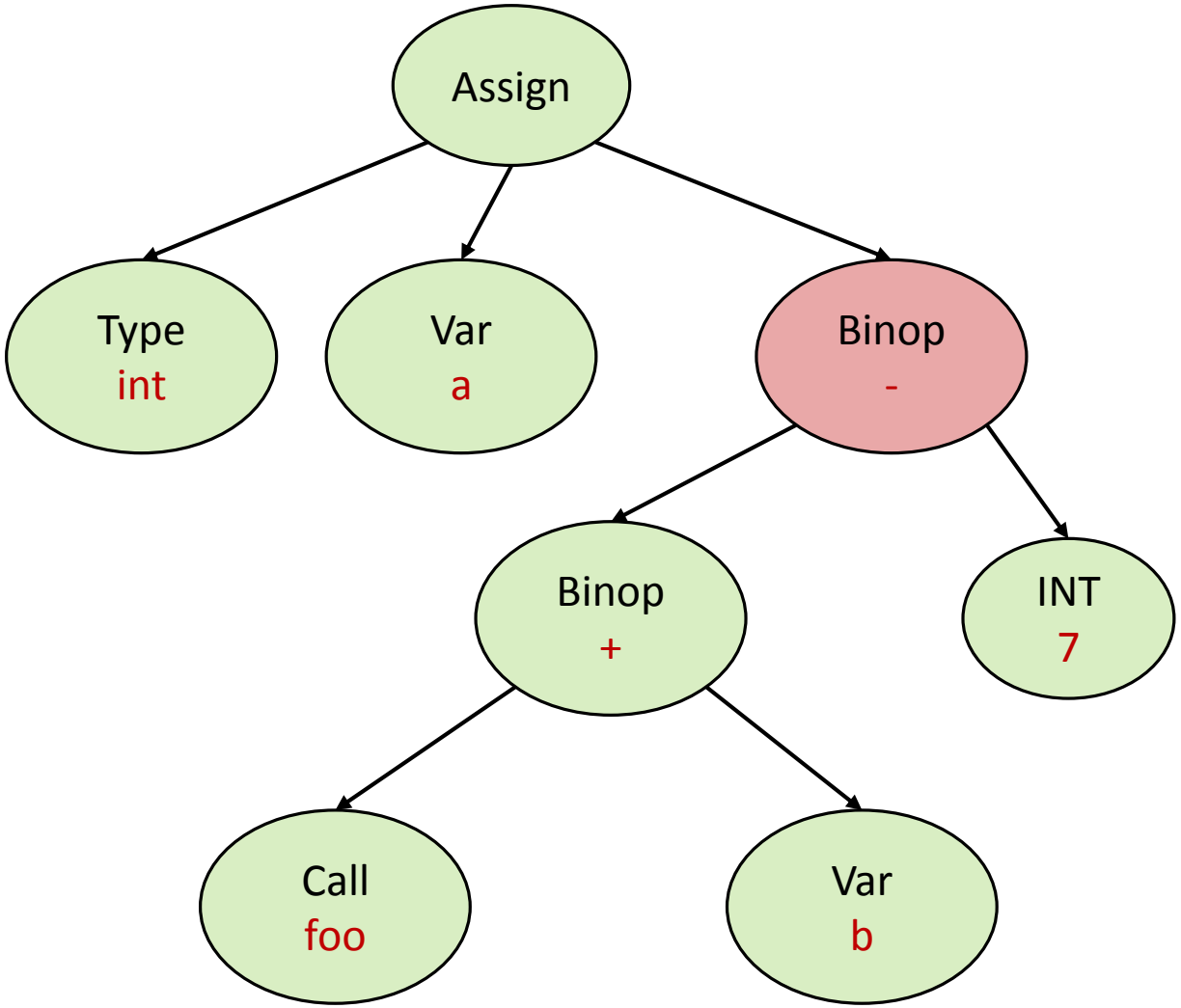
```
int a = foo() + b - 7;
...
```

ID	Type	Kind
foo	int,void	function
b	int	variable



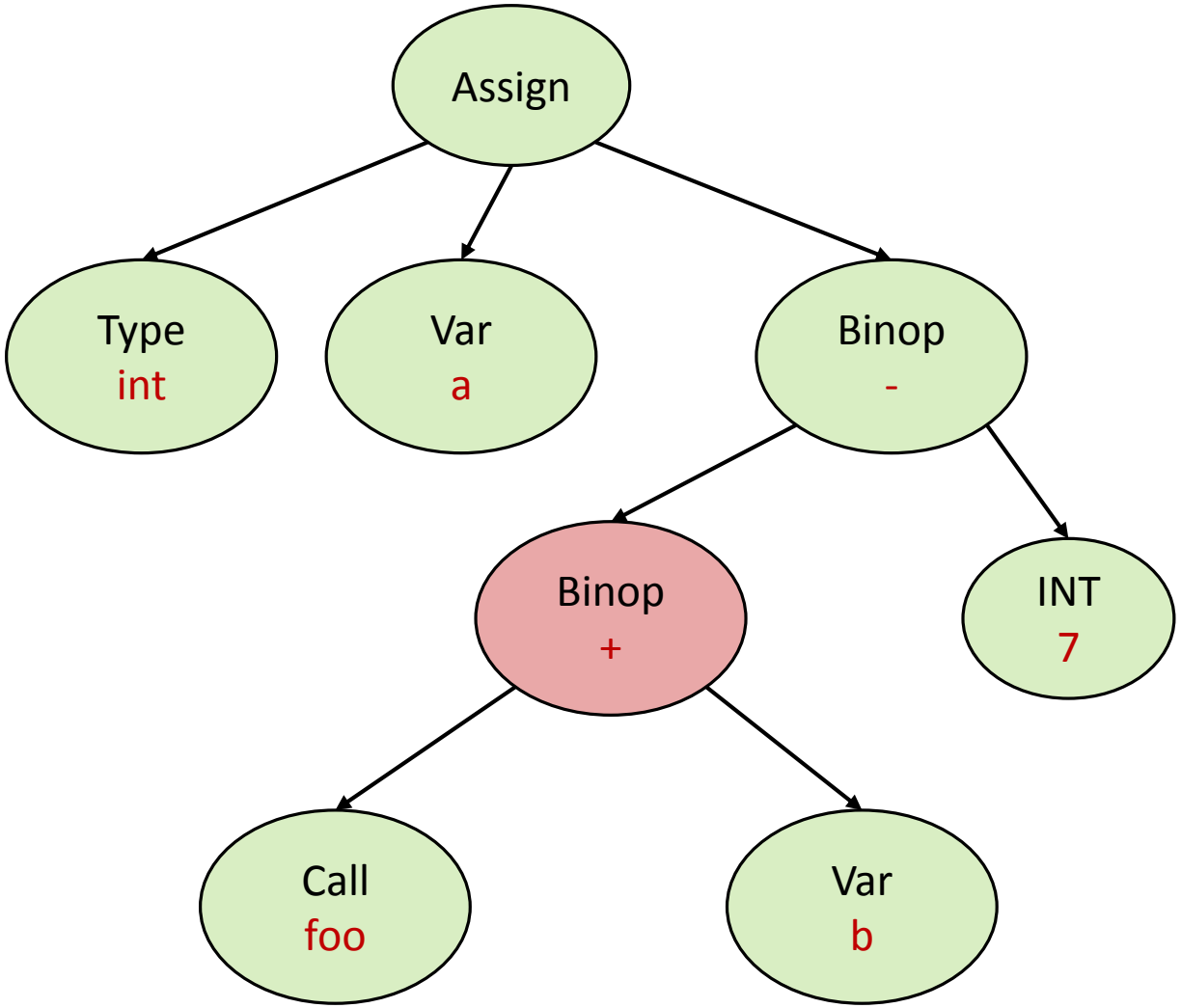
```
int a = foo() + b - 7;
...
```

ID	Type	Kind
foo	int,void	function
b	int	variable



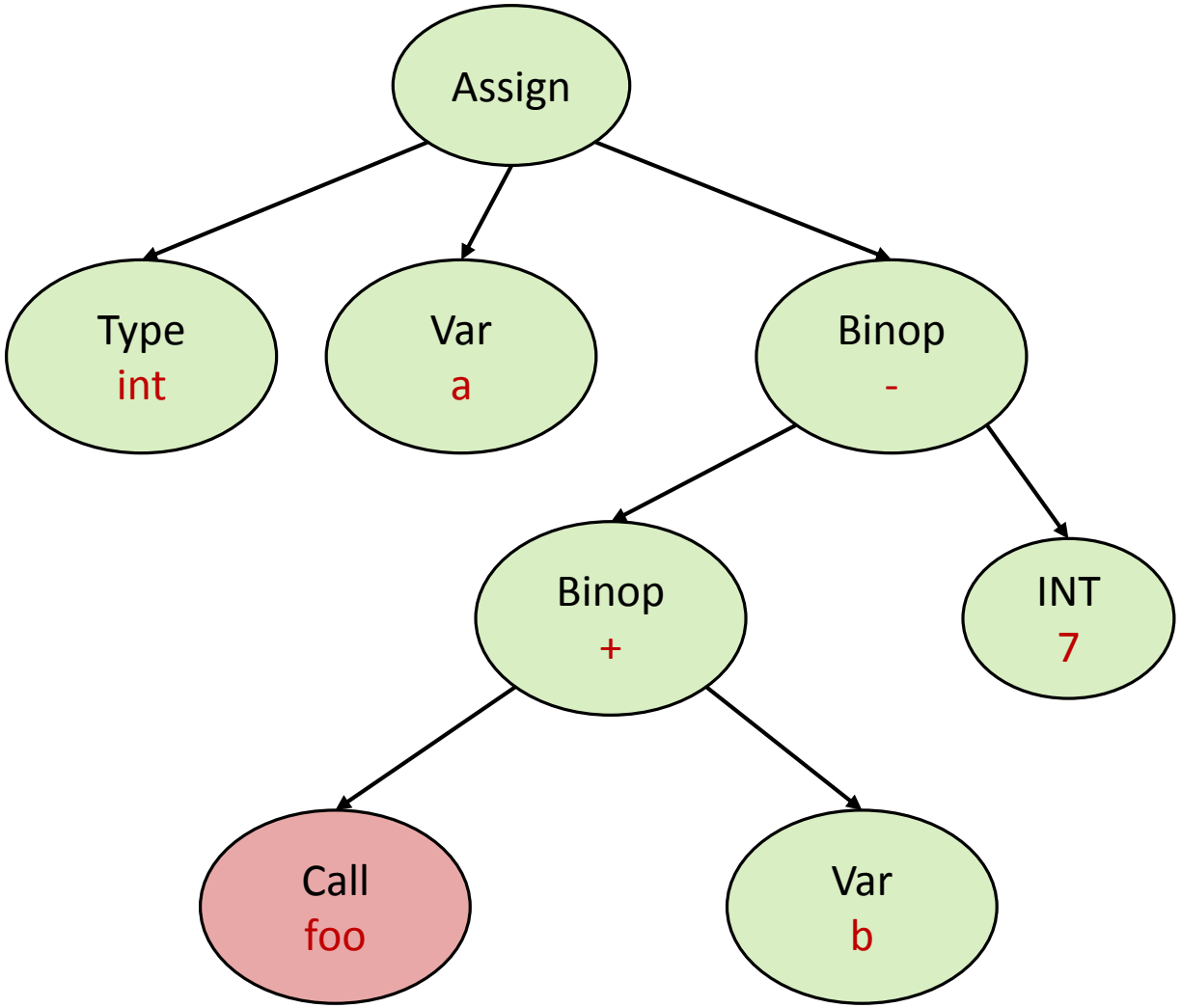
```
int a = foo() + b - 7;
...
```

ID	Type	Kind
foo	int,void	function
b	int	variable



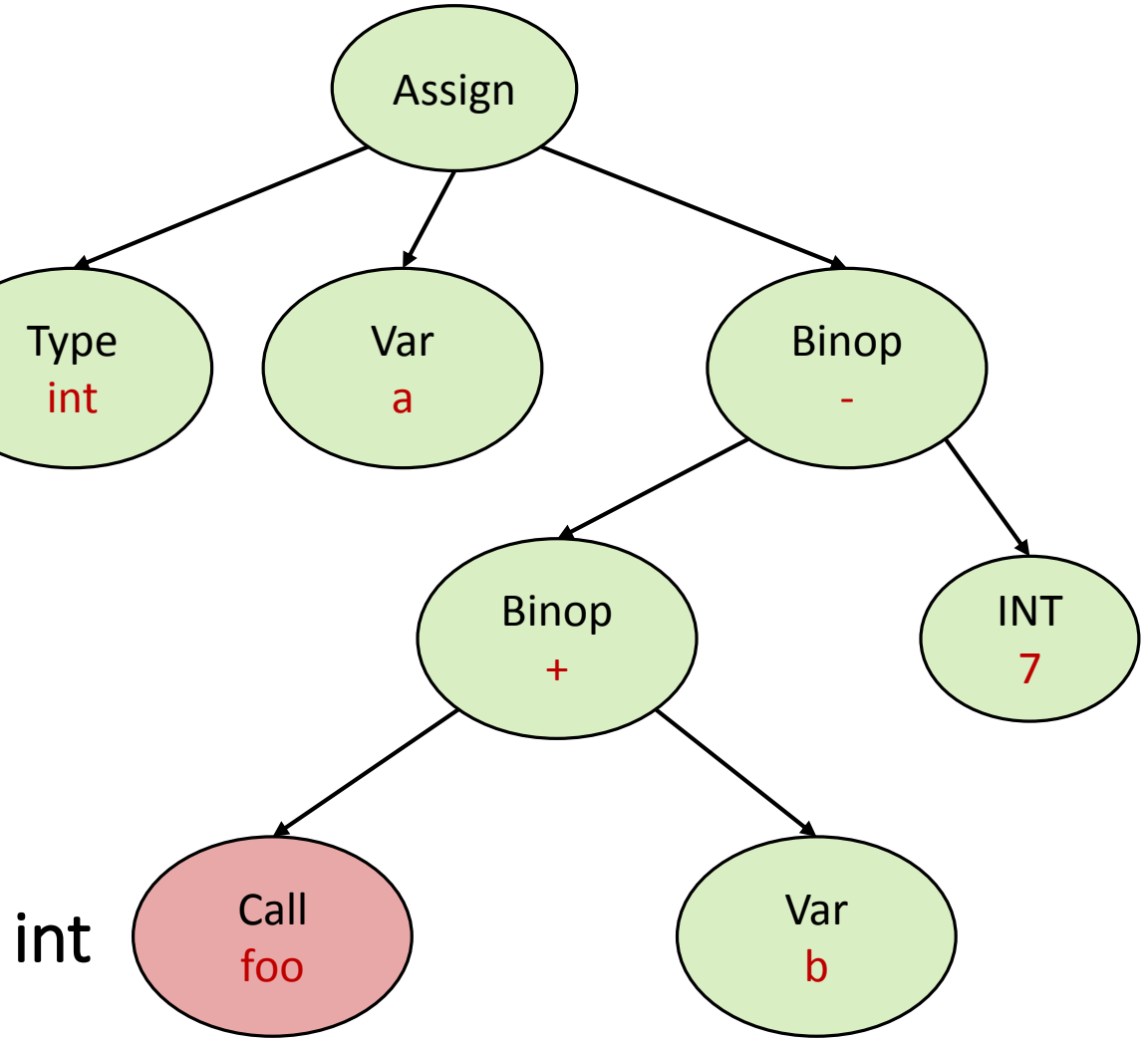
```
int a = foo() + b - 7;
...
```

ID	Type	Kind
foo	int,void	function
b	int	variable



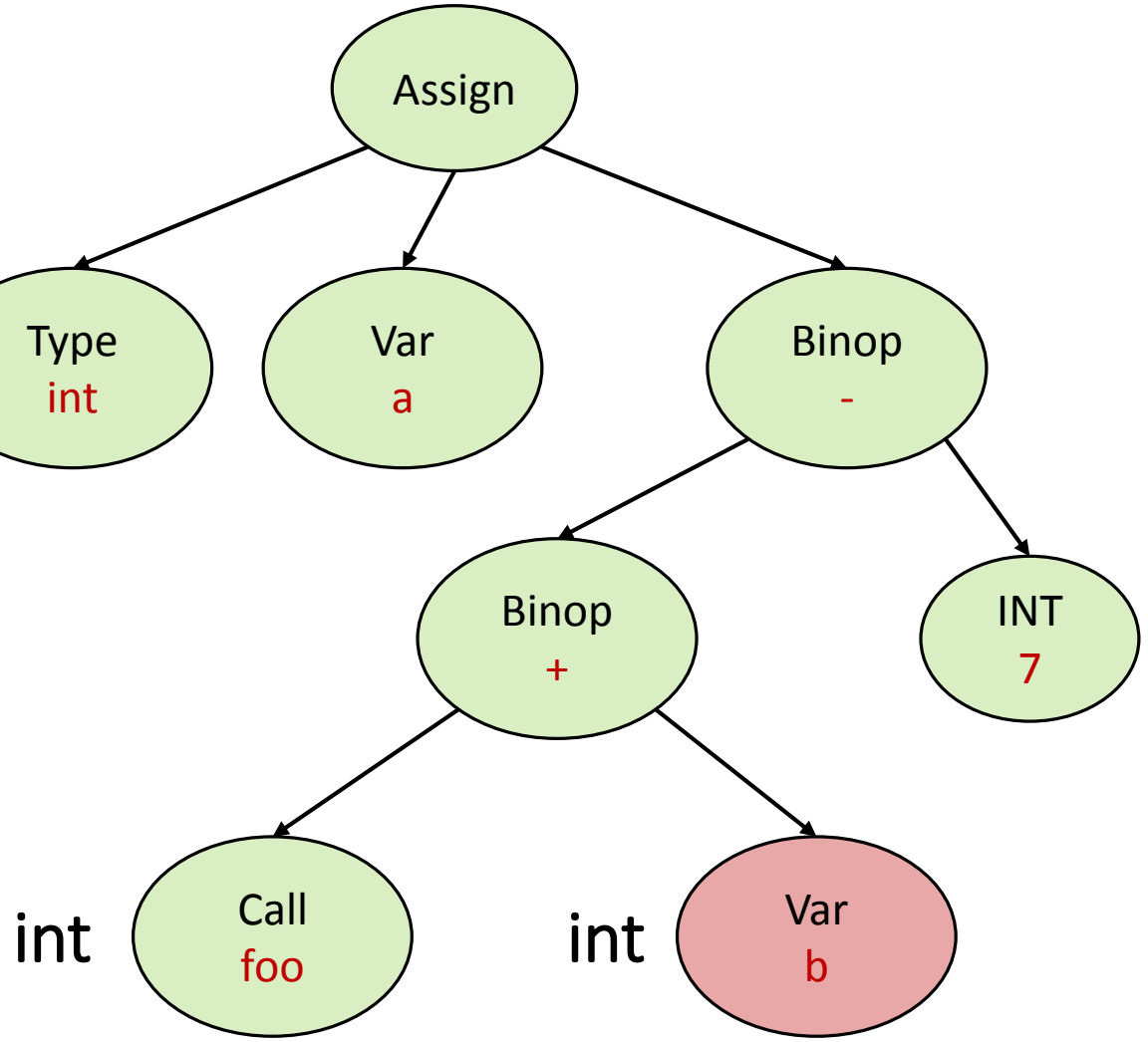
```
int a = foo() + b - 7;
...
```

ID	Type	Kind
foo	int,void	function
b	int	variable



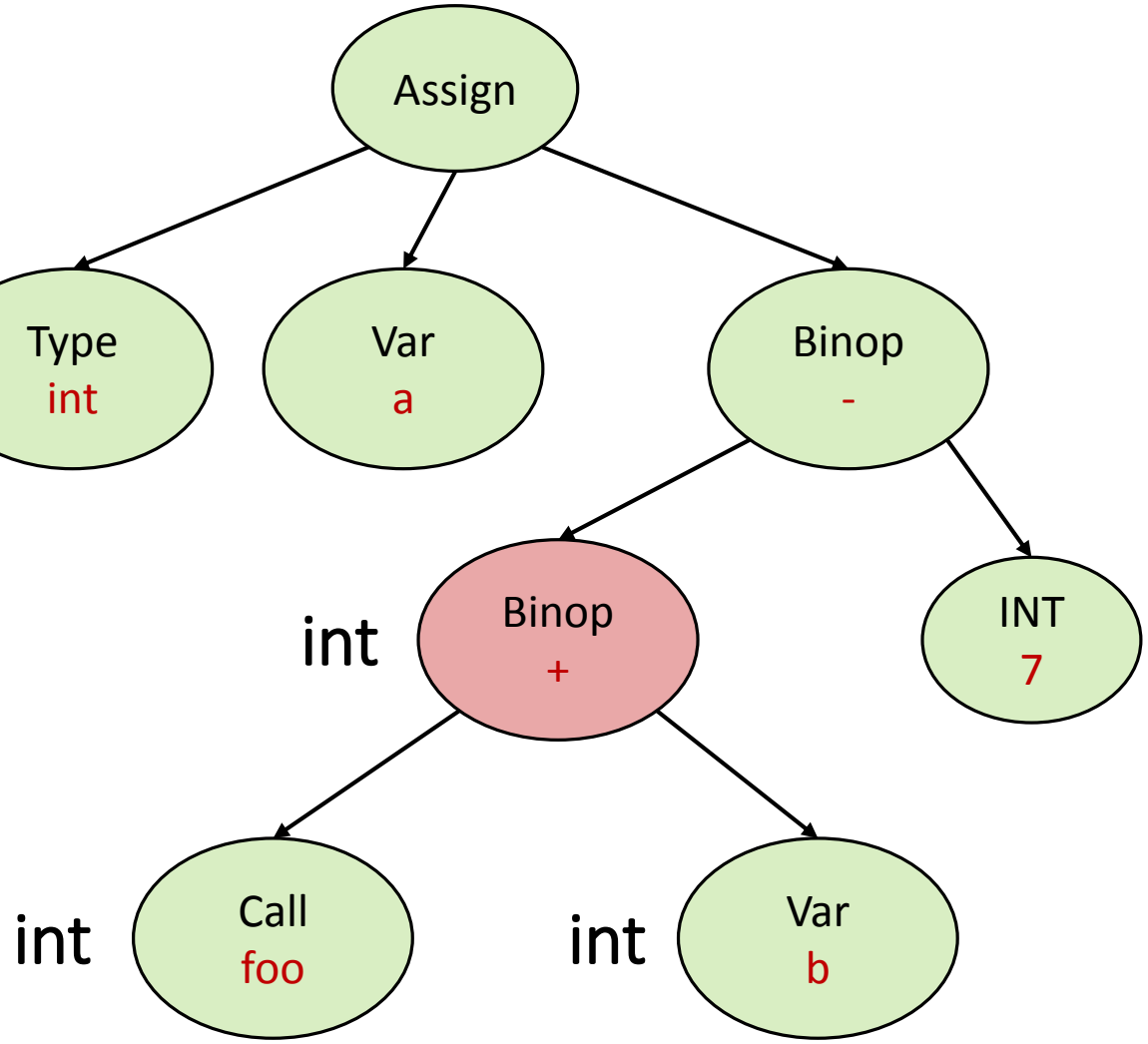
```
int a = foo() + b - 7;
...
```

ID	Type	Kind
foo	int,void	function
b	int	variable



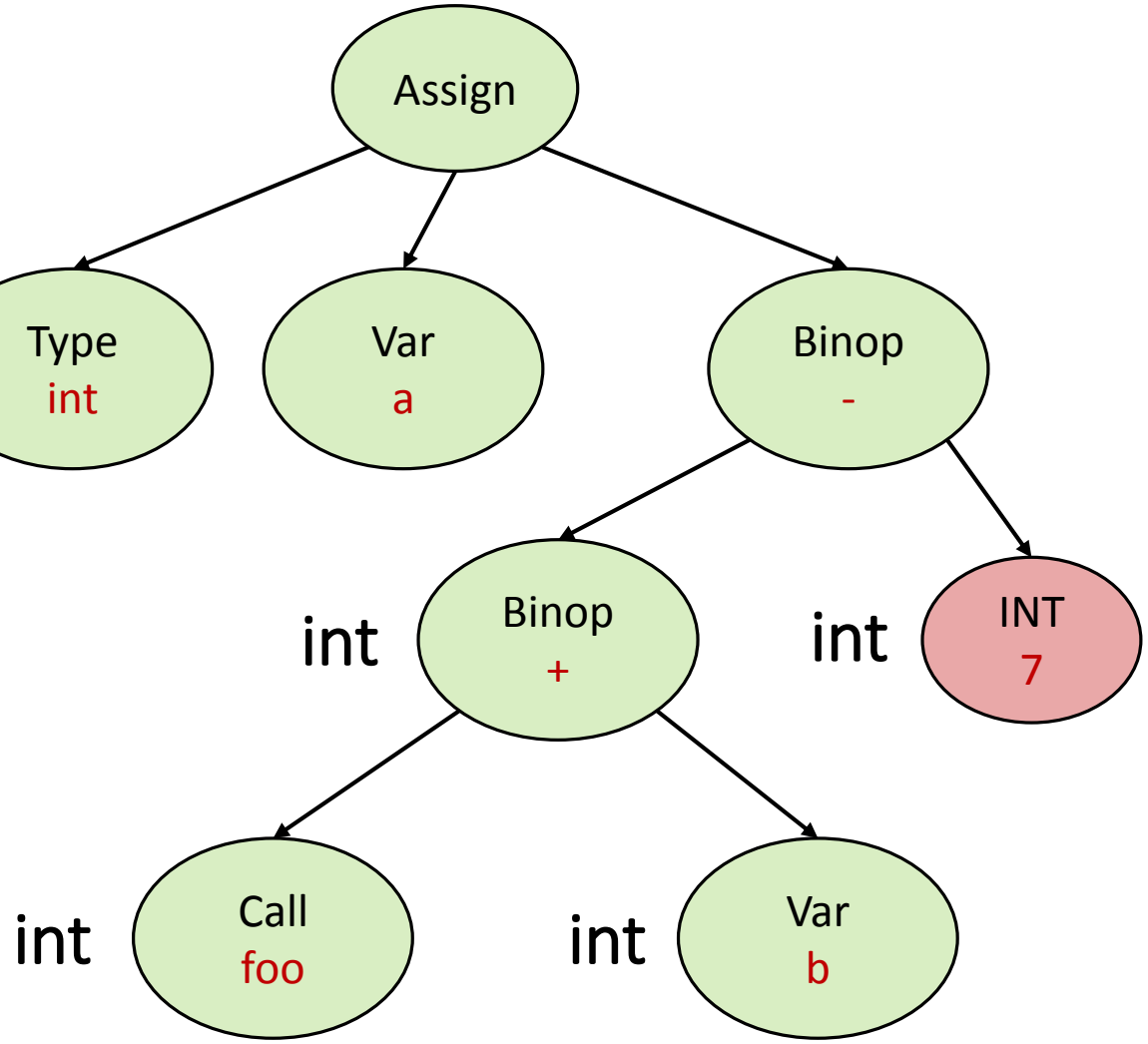
```
int a = foo() + b - 7;
...
```

ID	Type	Kind
foo	int,void	function
b	int	variable



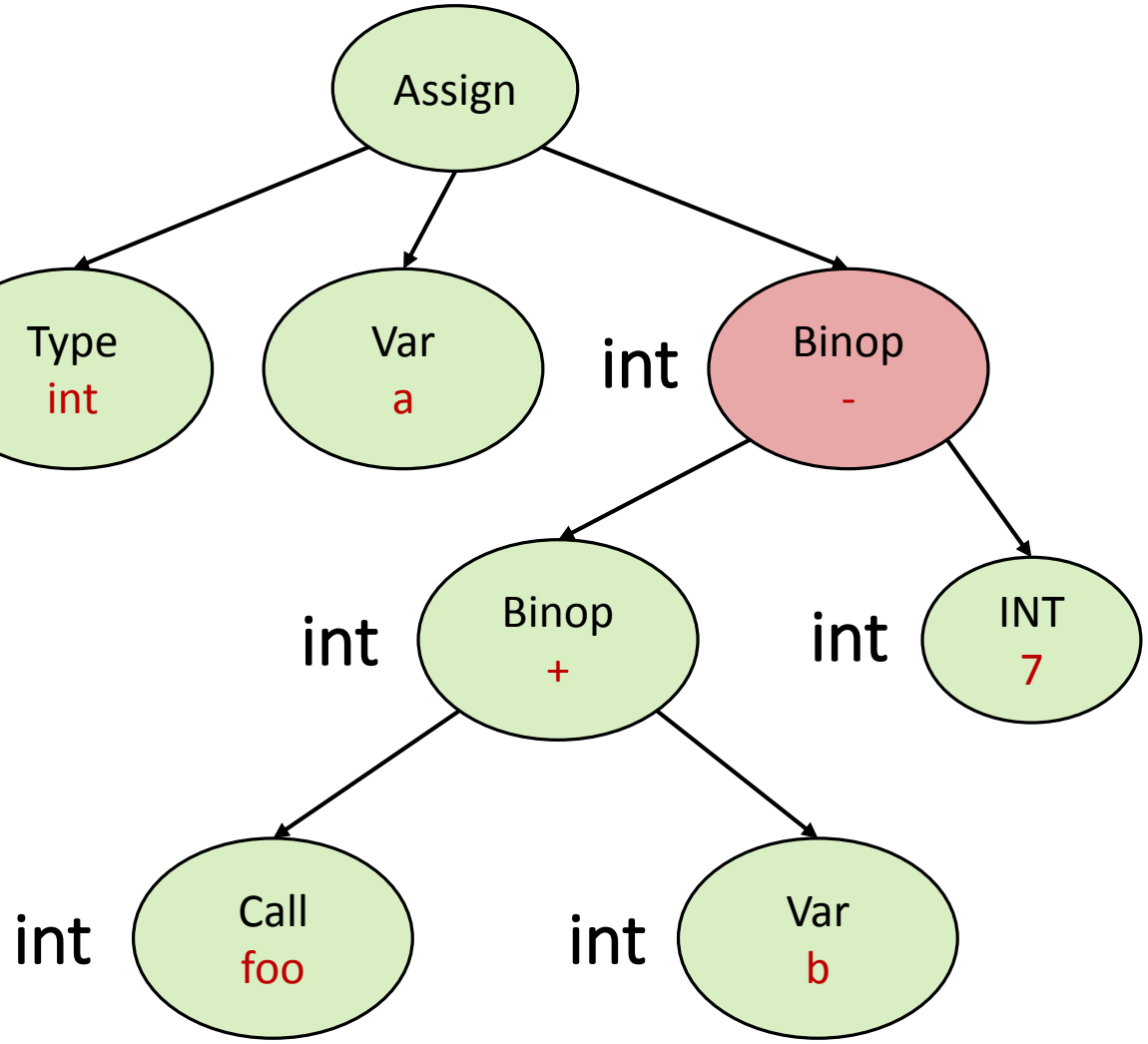
```
int a = foo() + b - 7;
...
```

ID	Type	Kind
foo	int,void	function
b	int	variable



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

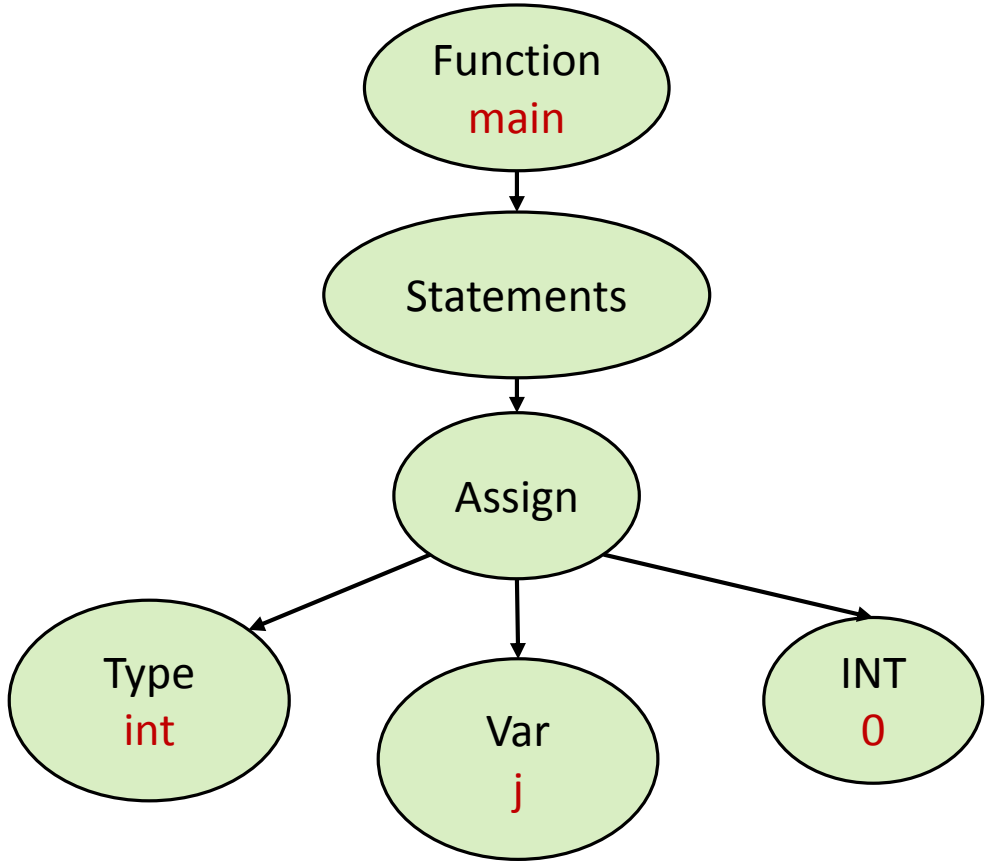
ID	Type	Kind
foo	int,void	function
b	int	variable



Examples

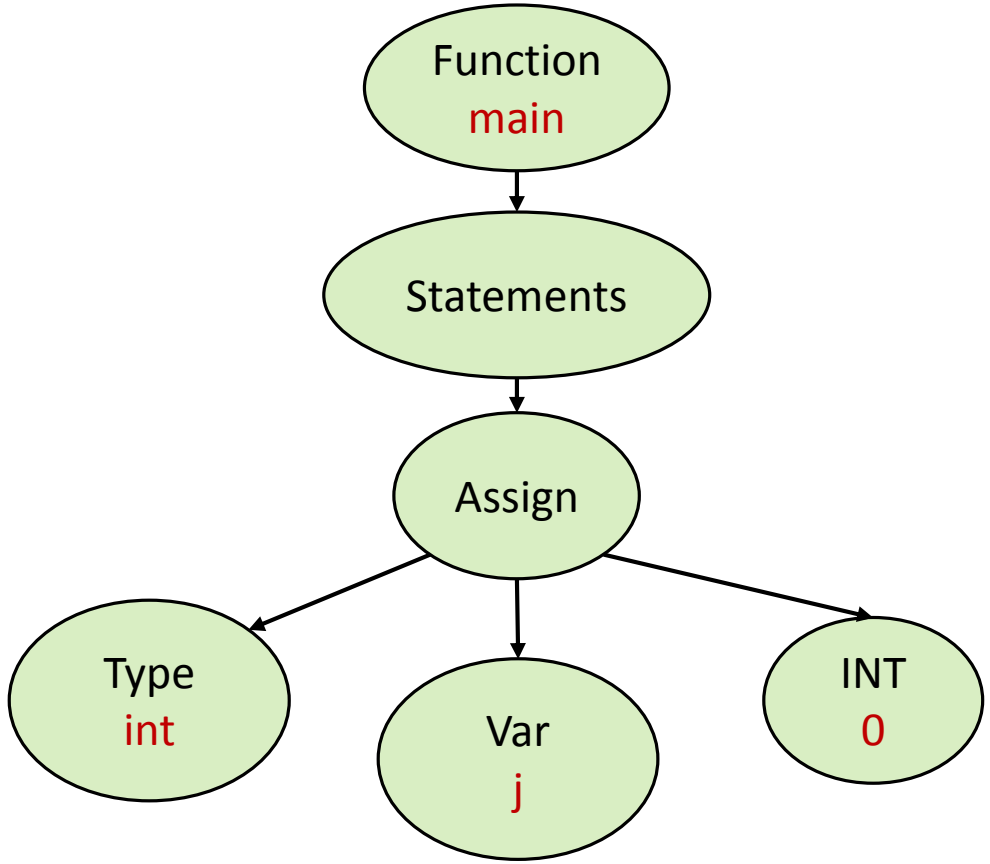
Assignments

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



Assignments

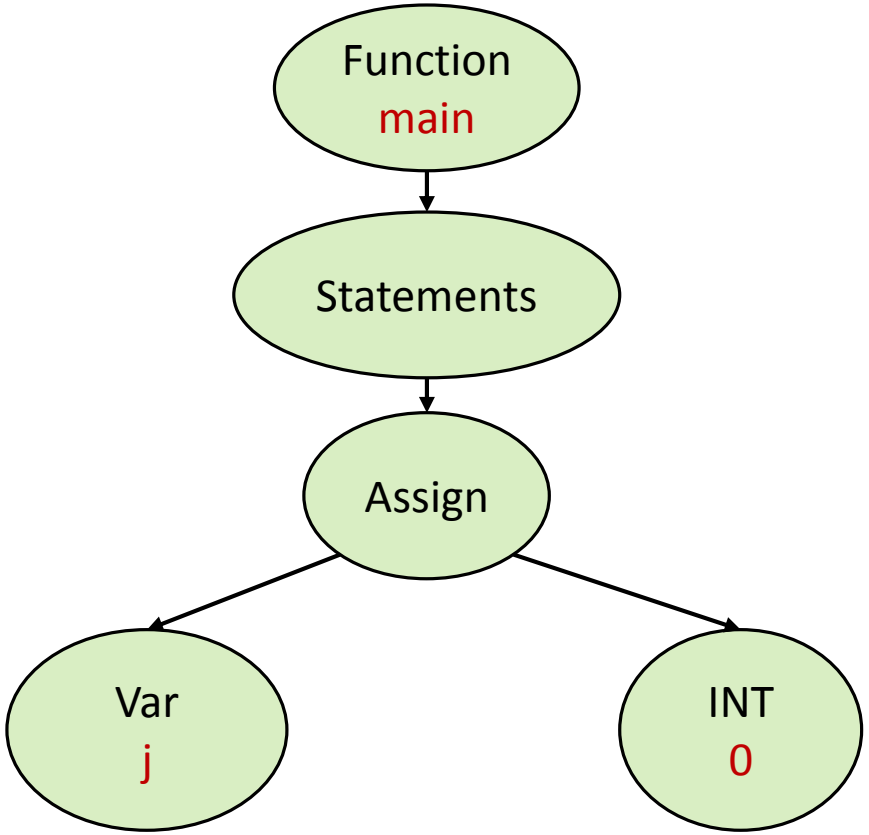
```
void main() {
  int j = 0;
}
```



Valid

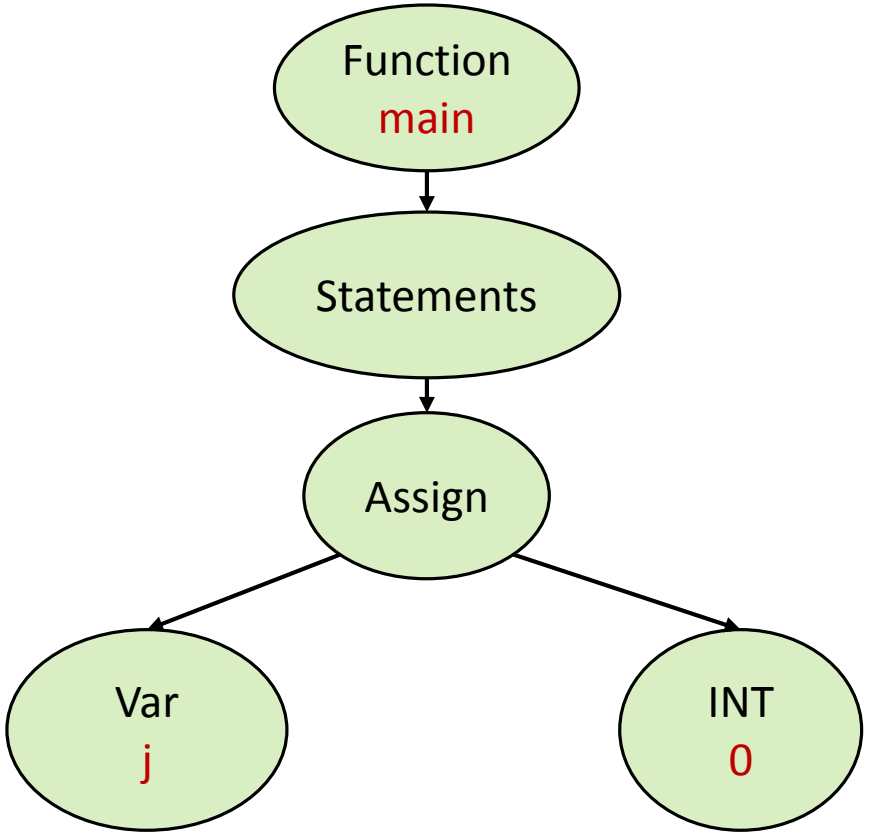
Assignments

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



Assignments

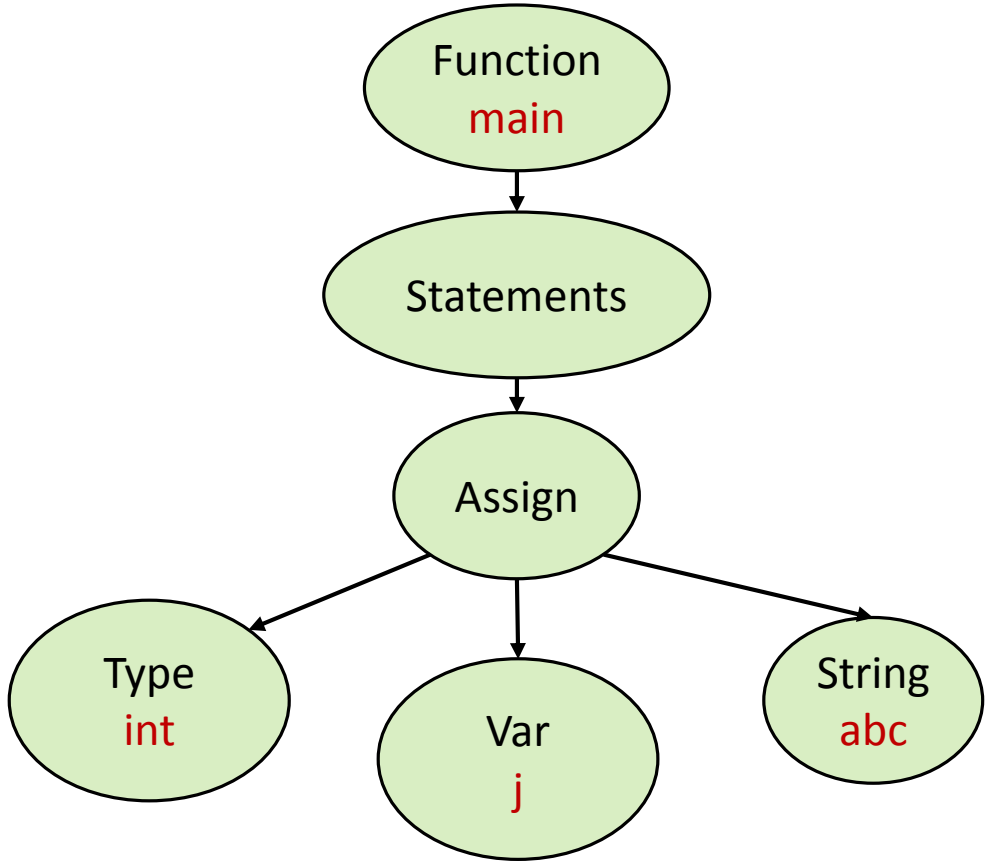
```
void main() {
  j = 0;
}
```



Invalid

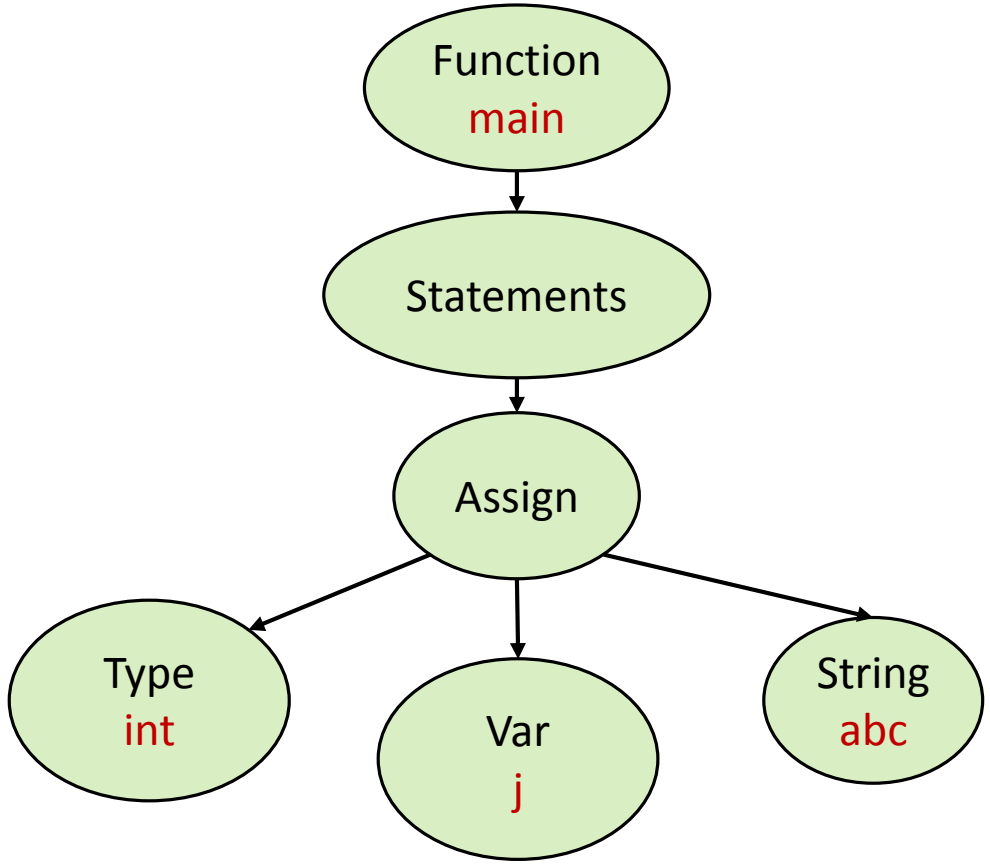
Assignments

```
void main() {
  int j = "abc";
}
```



Assignments

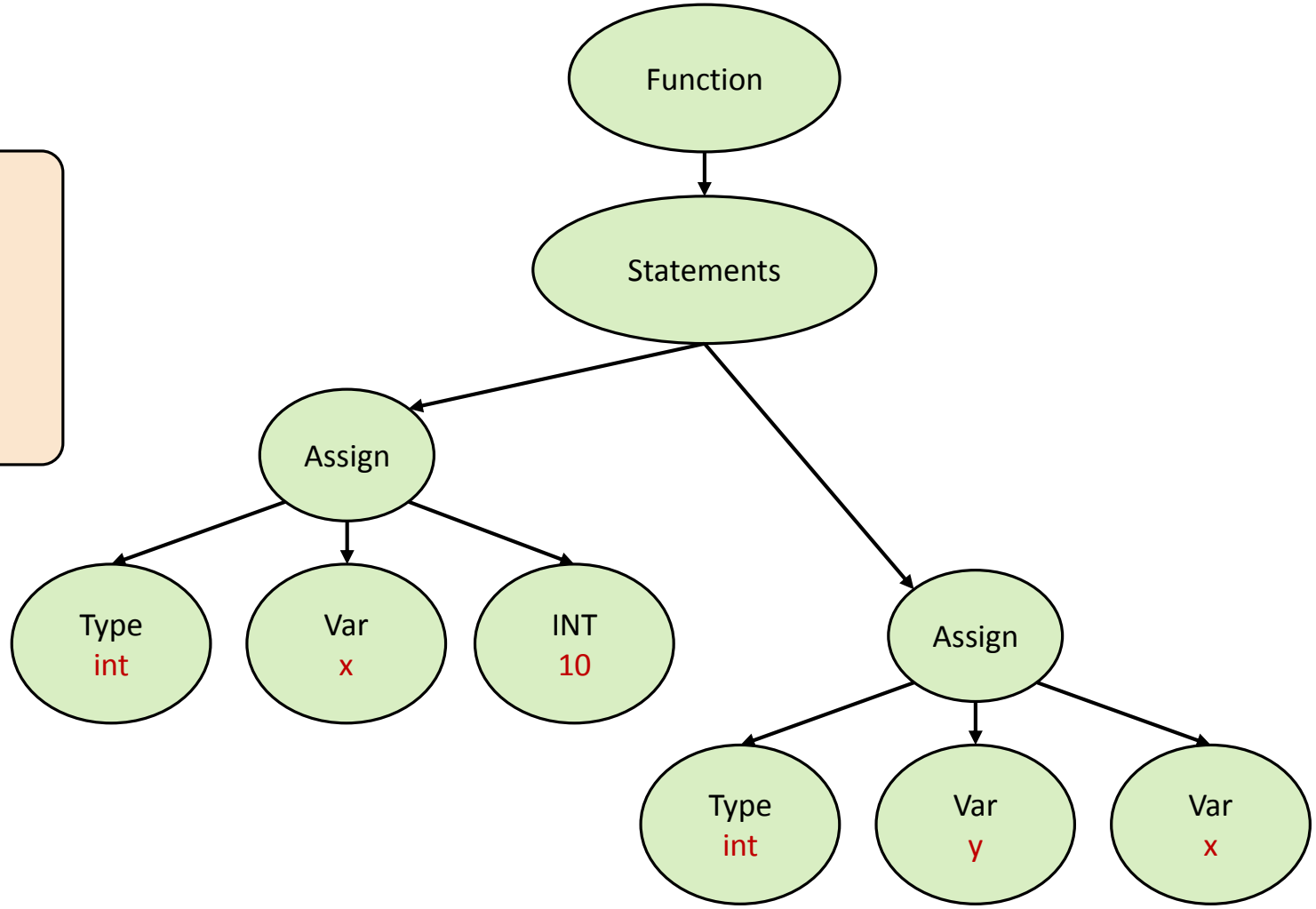
```
void main() {
  int j = "abc";
}
```



Invalid

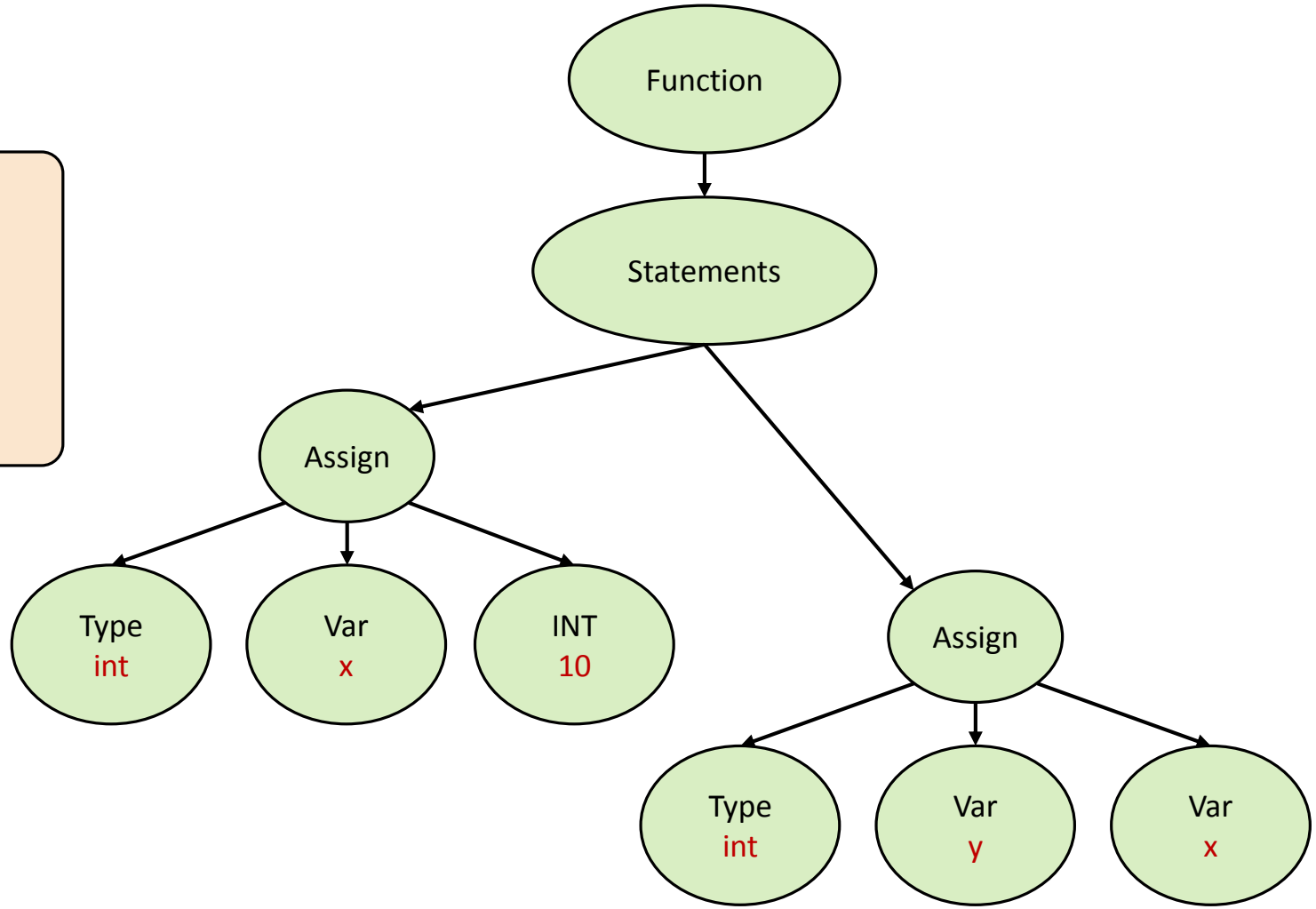
Assignments

```
void main() {
  int x = 10;
  int y = x;
}
```



Assignments

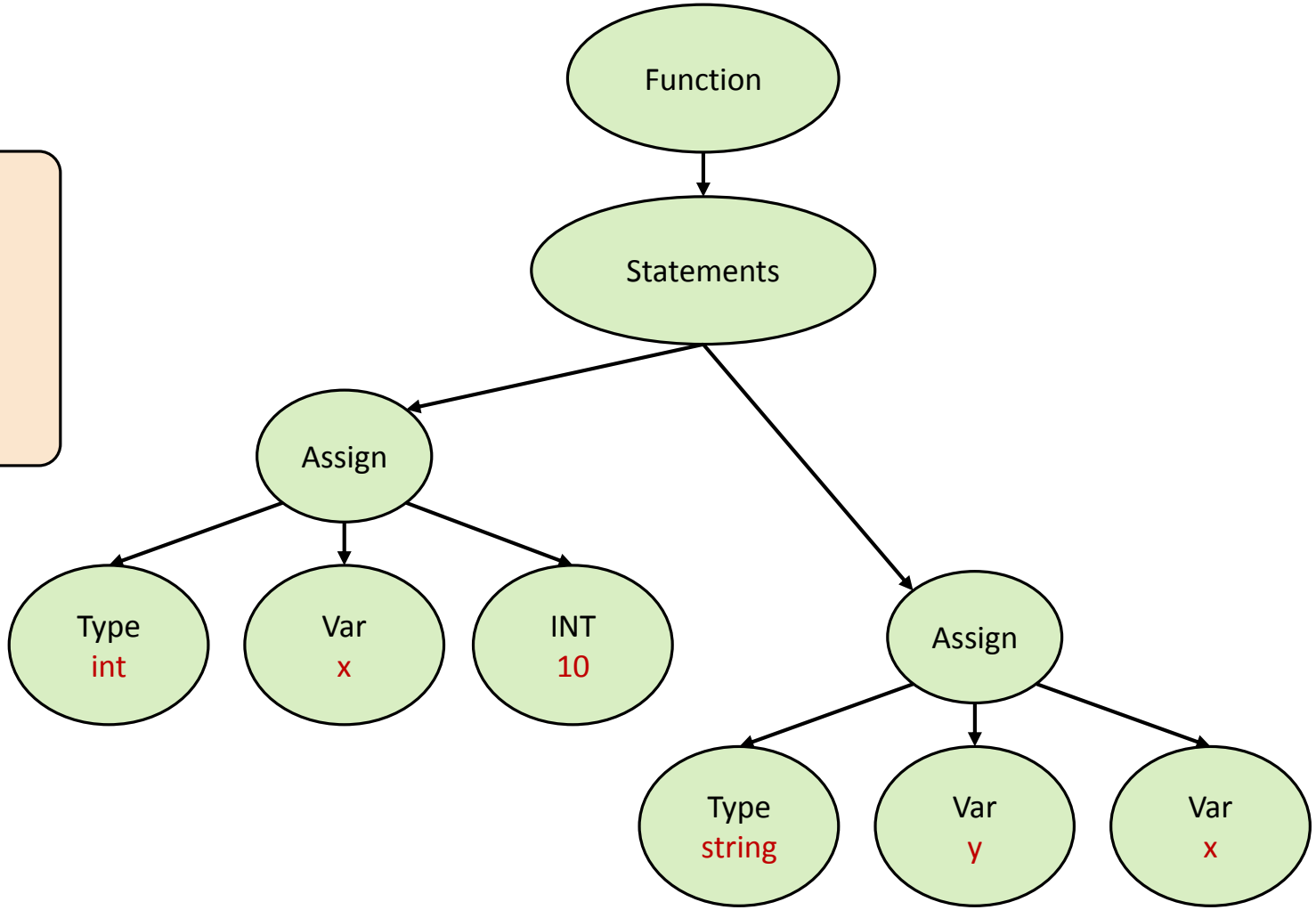
```
void main() {
  int x = 10;
  int y = x;
}
```



Valid

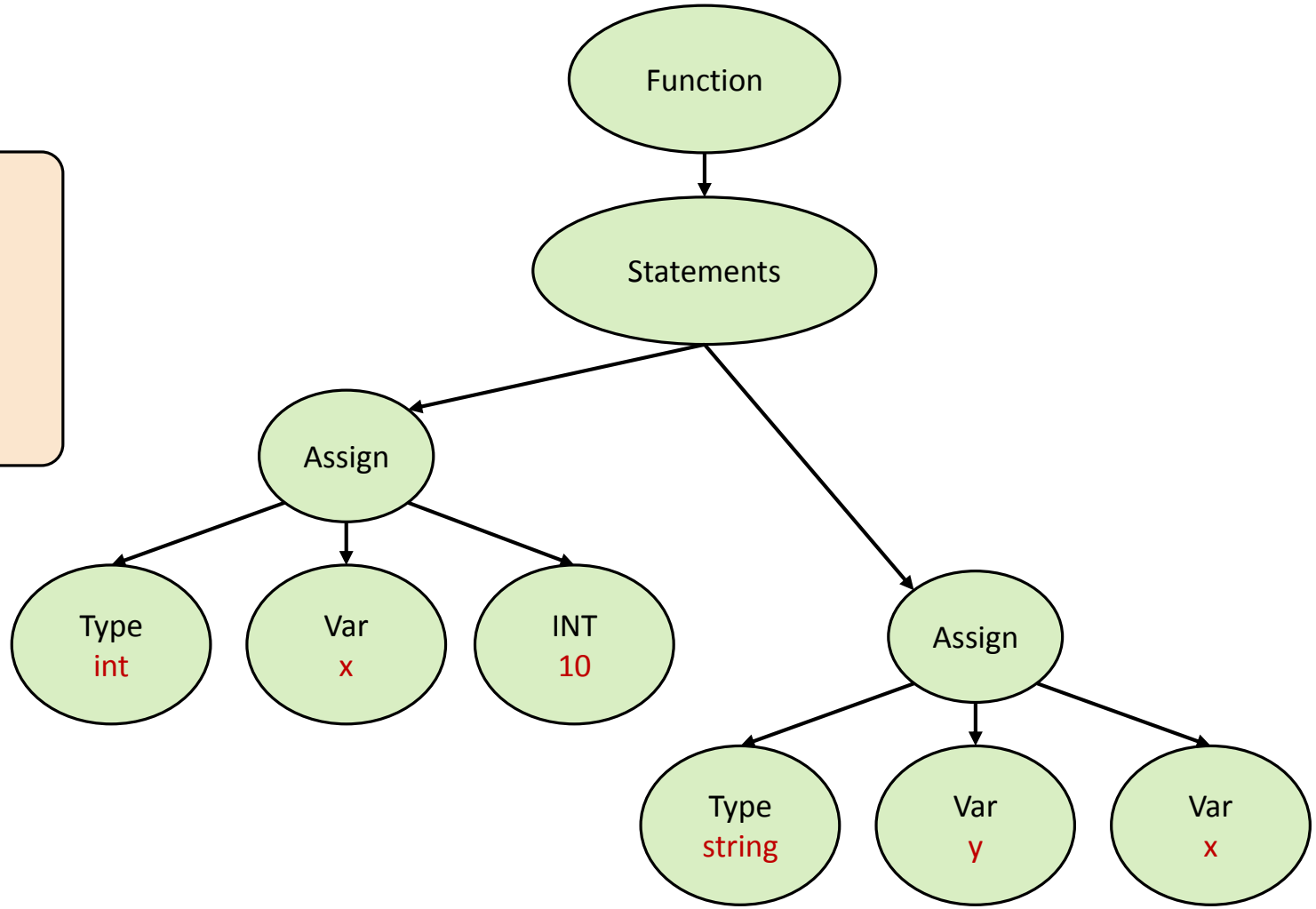
Assignments

```
void main() {
  int x = 10;
  string y = x;
}
```



Assignments

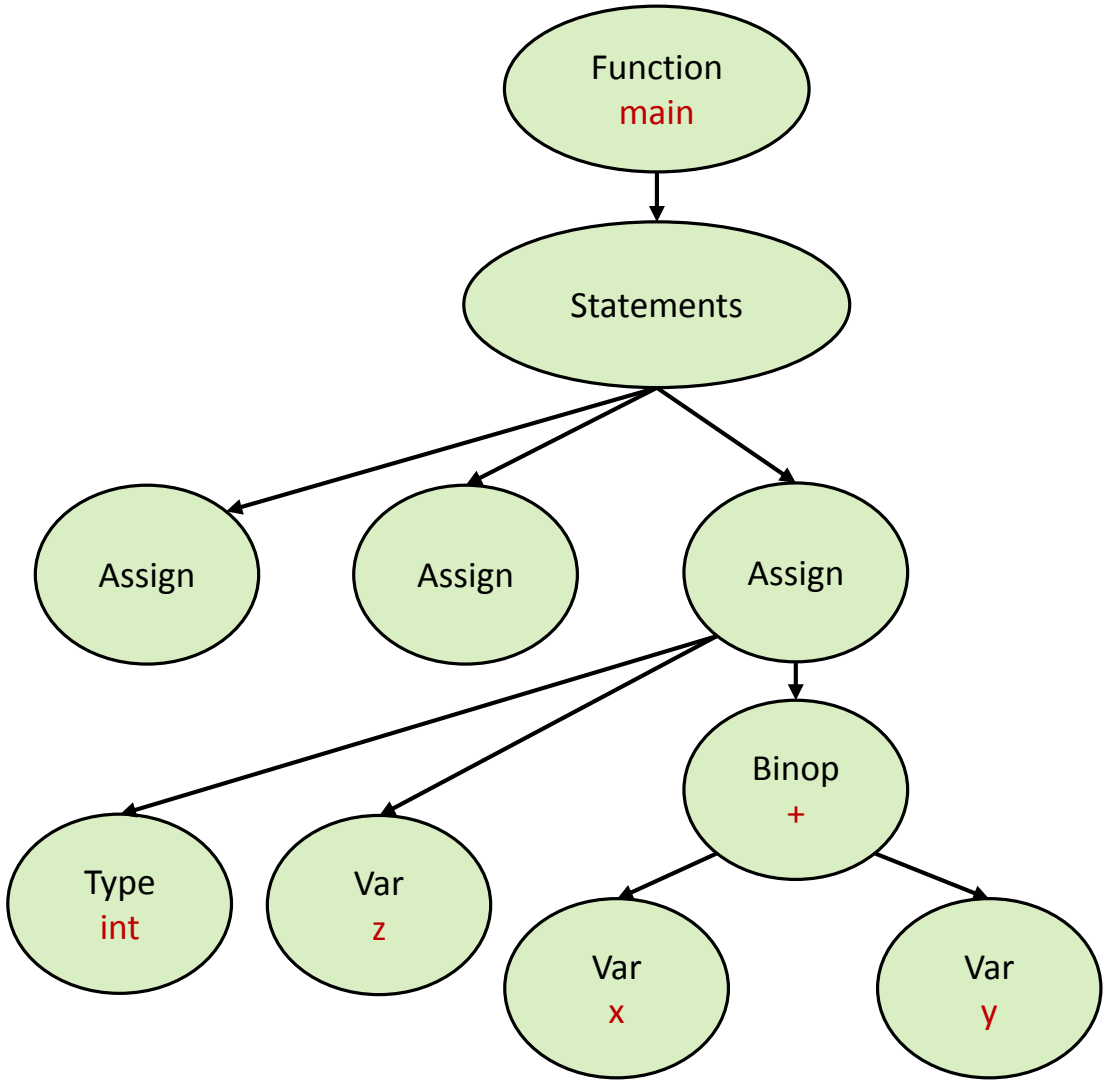
```
void main() {
   int x = 10;
   string y = x;
}
```



Invalid

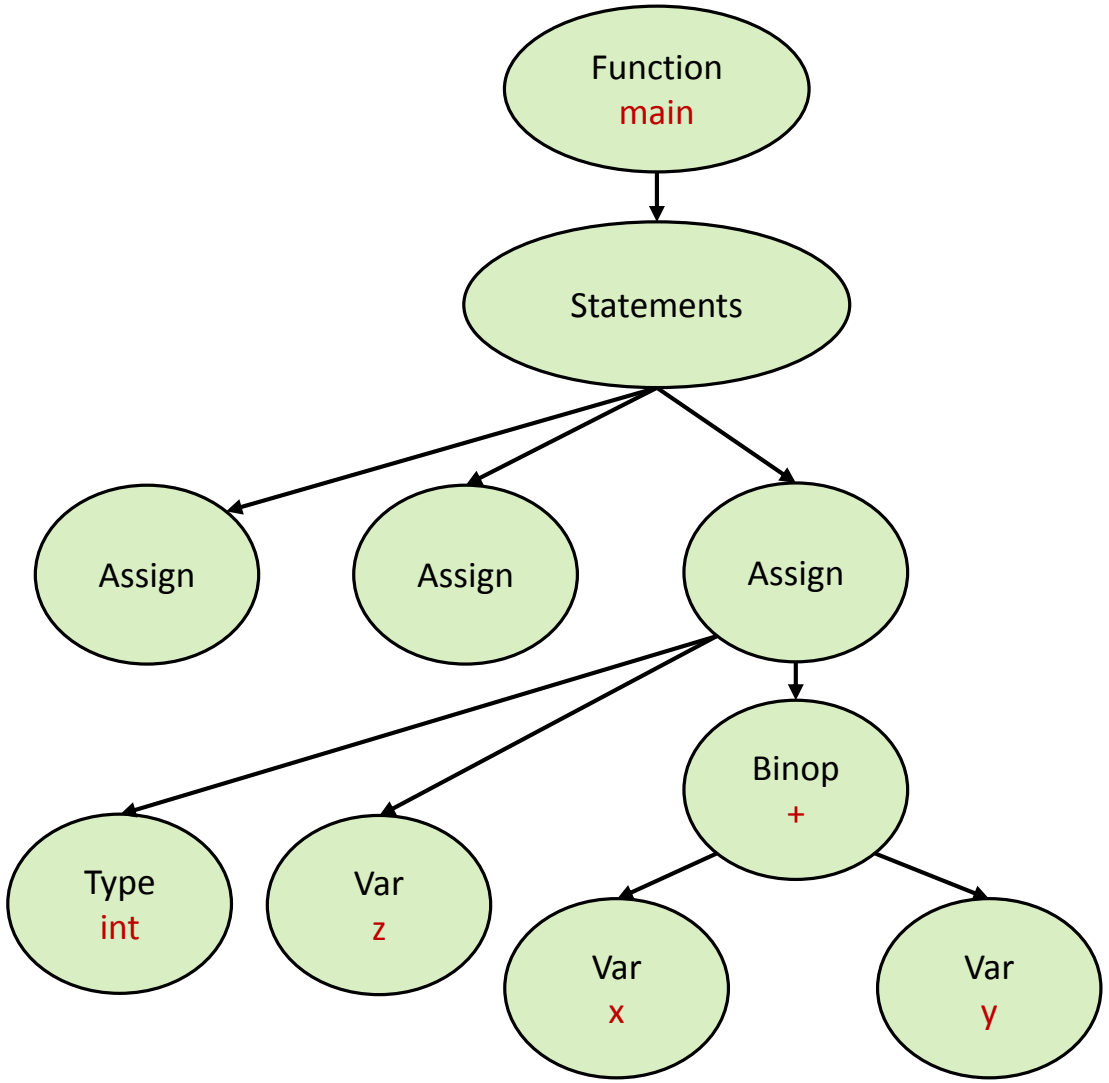
Binary Operations

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



Binary Operations

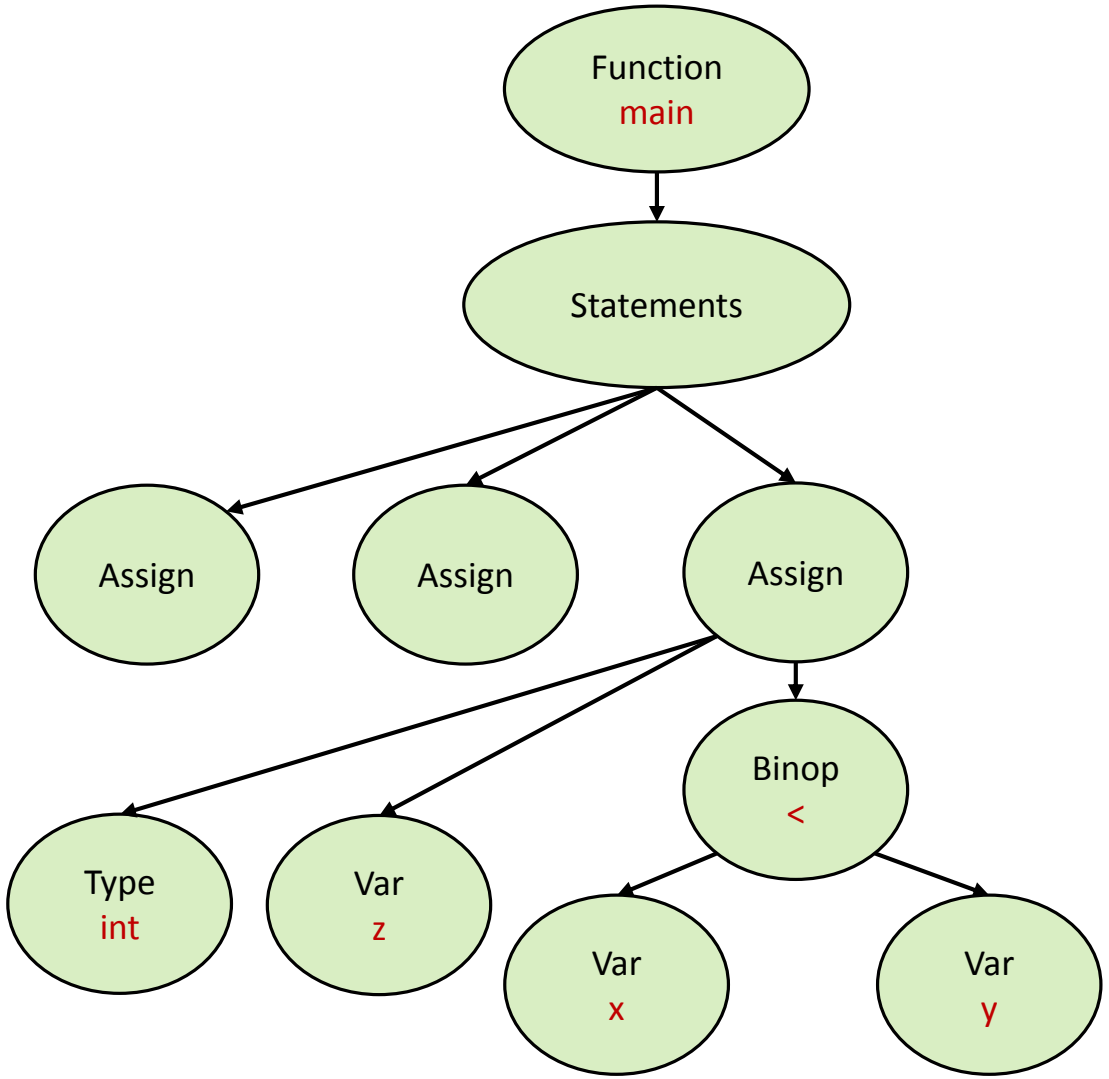
```
void main() {
  int x = 1;
  int y = 2;
  int z = x + y;
}
```



Valid

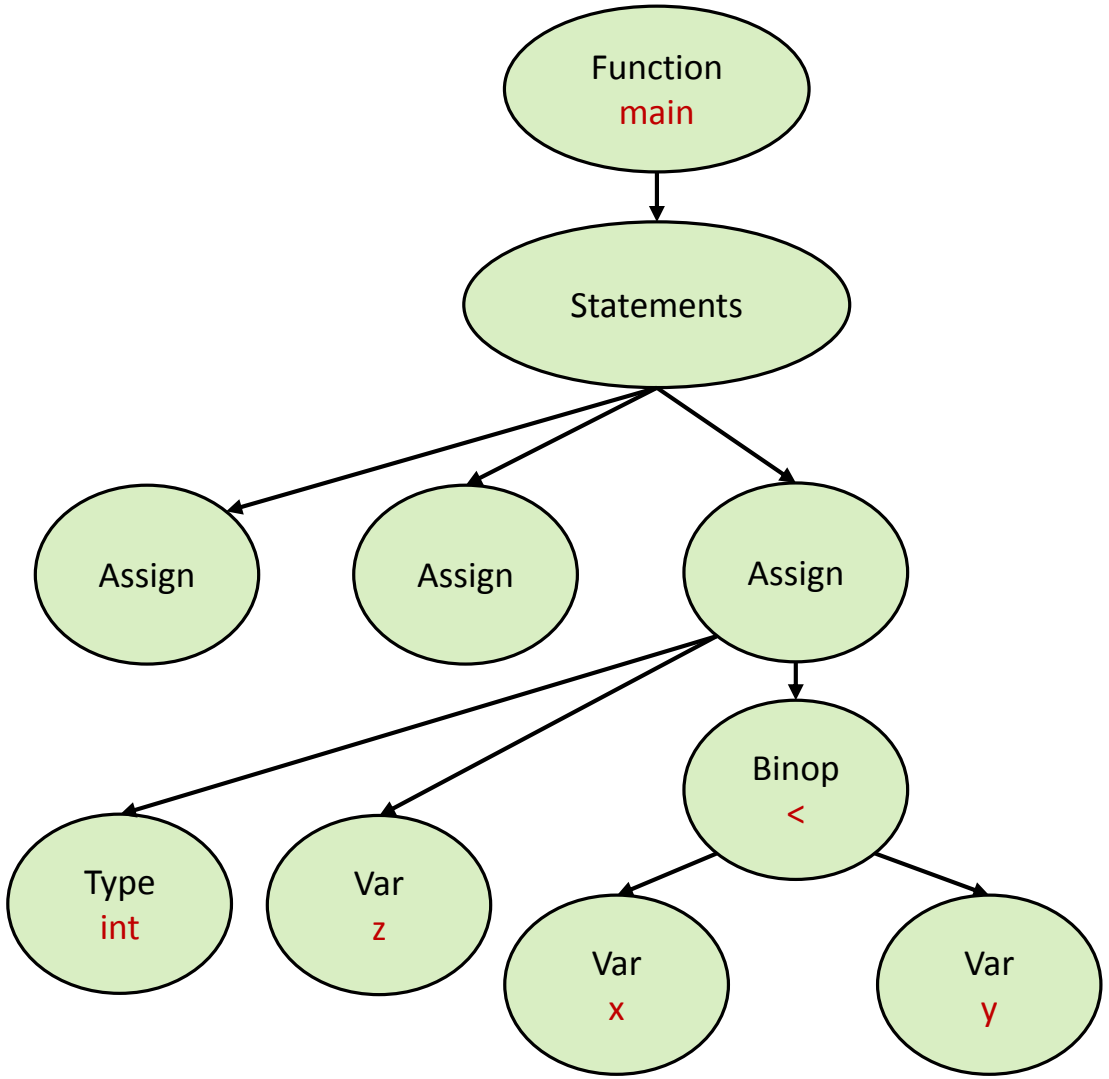
Binary Operations

```
void main() {
   int x = 1;
   string y = "A";
   int z = x < y;
}
```



Binary Operations

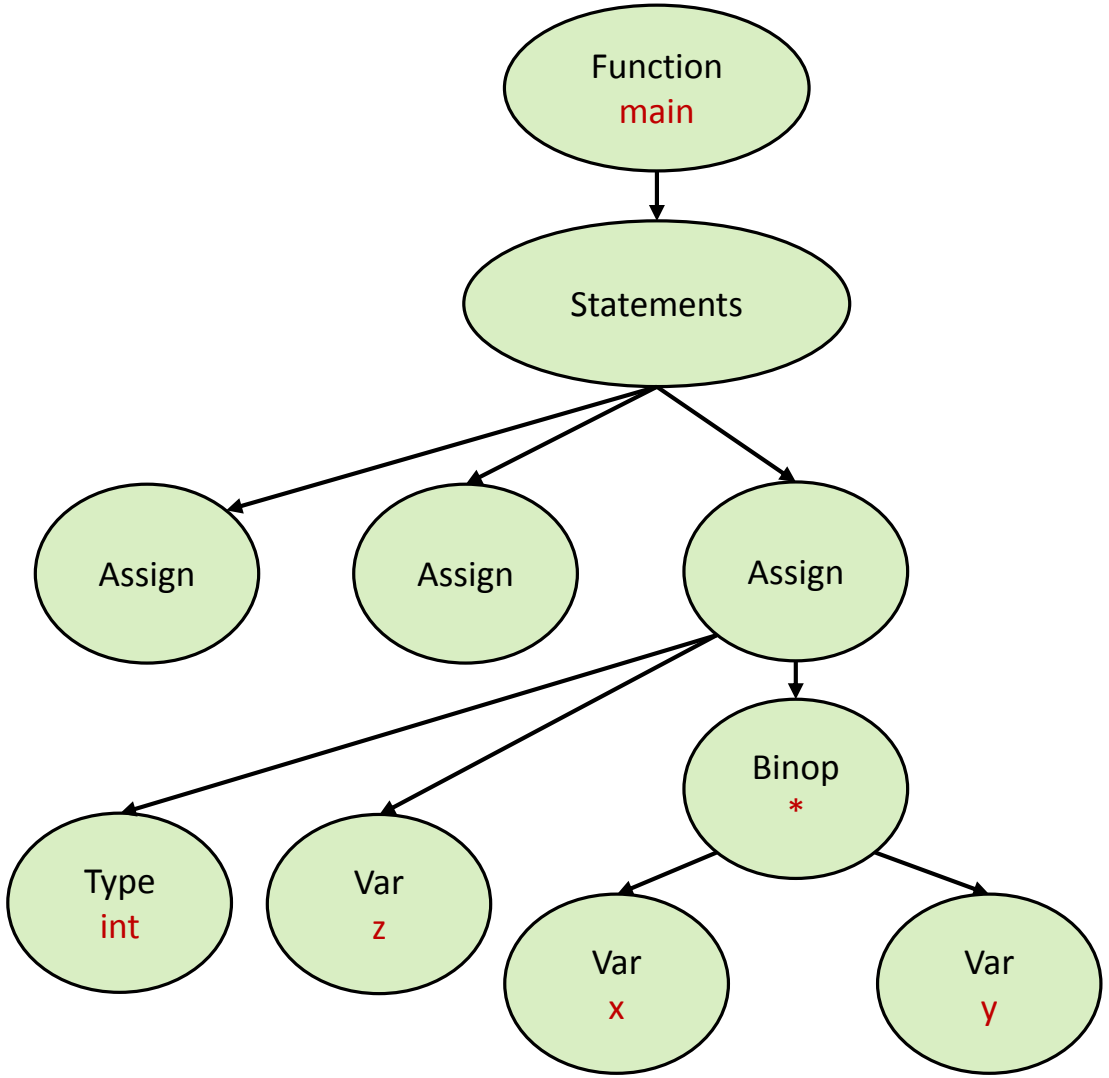
```
void main() {
   int x = 1;
   string y = "A";
   int z = x < y;
}
```



Invalid

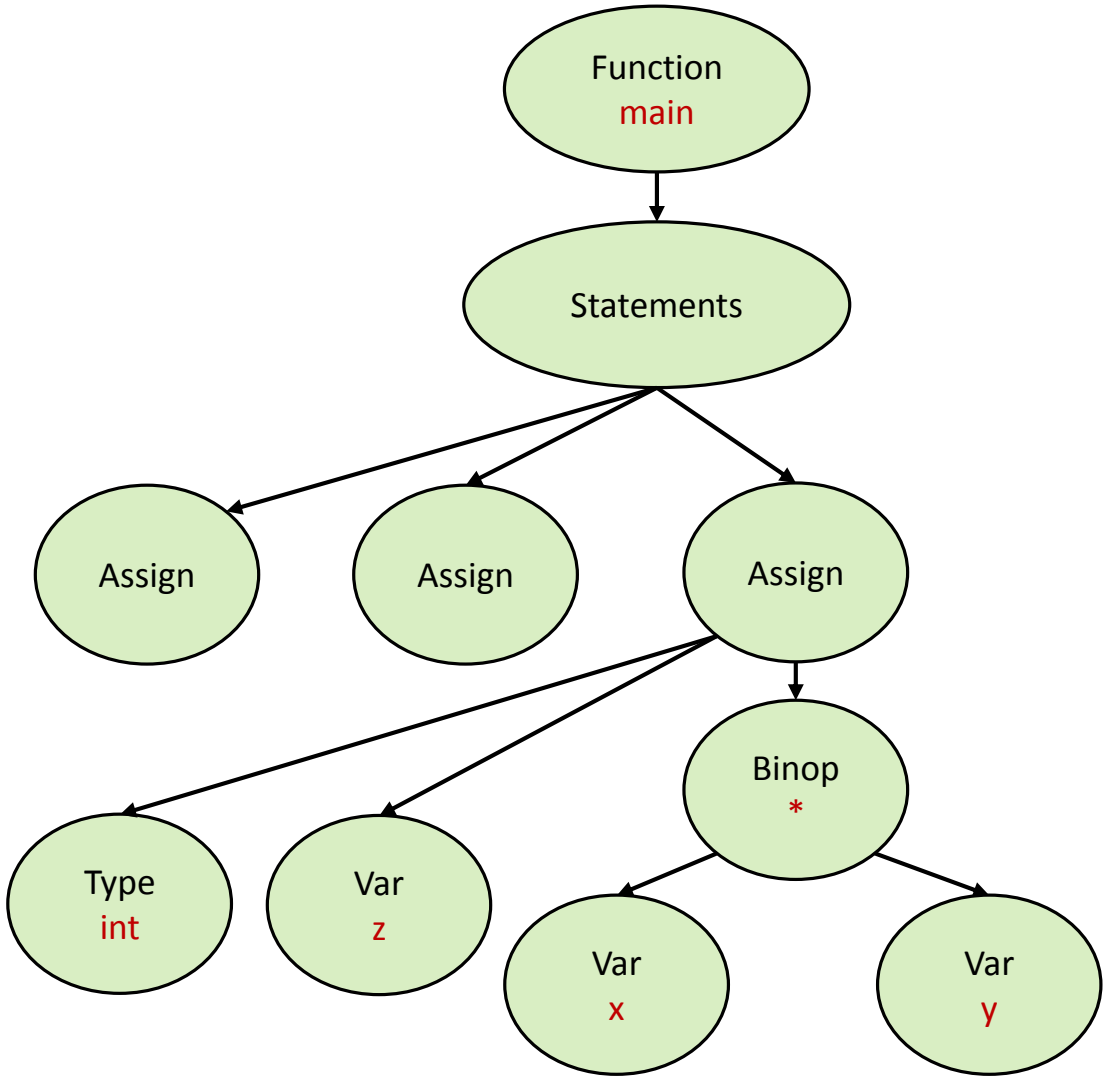
Binary Operations

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



Binary Operations

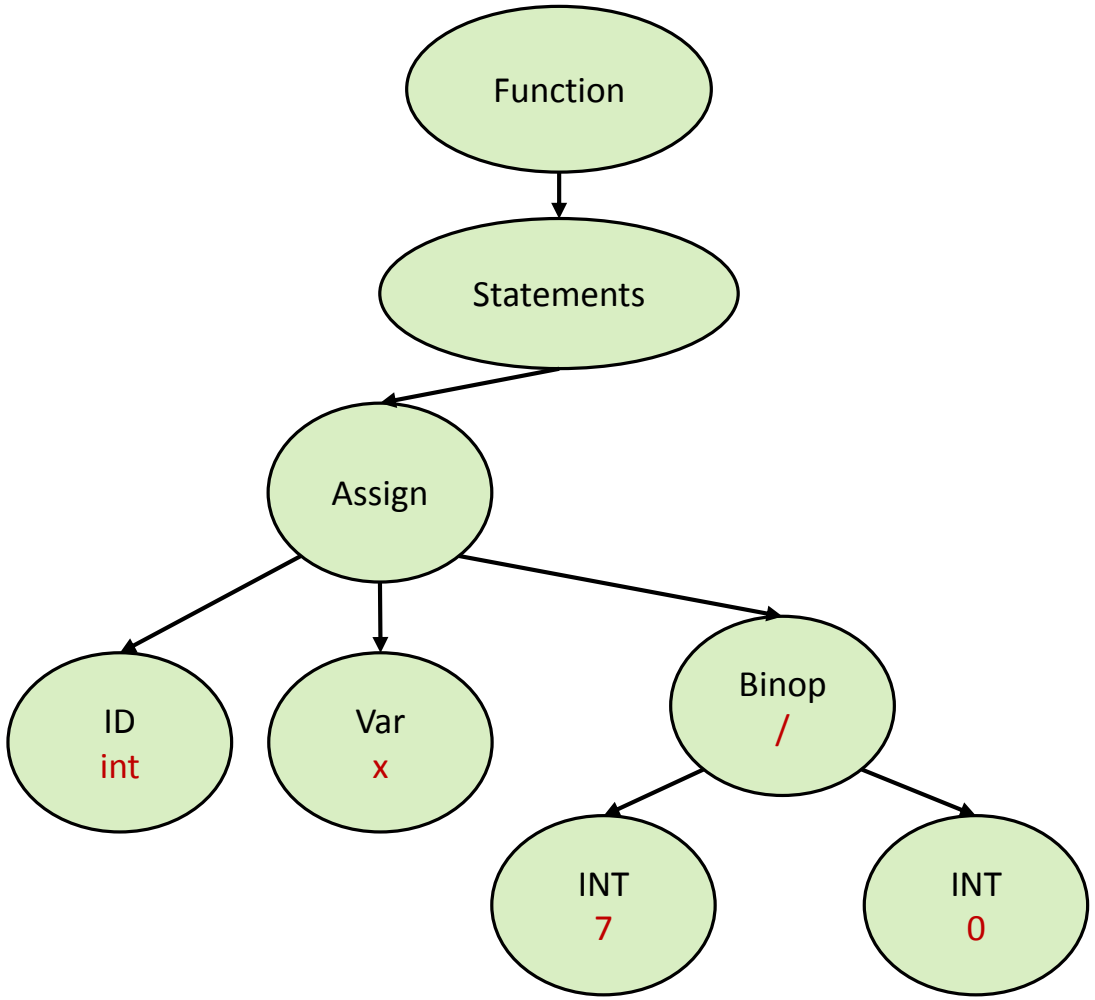
```
void main() {
   string x = "A";
   string y = "B";
   string z = x * y;
}
```



Invalid

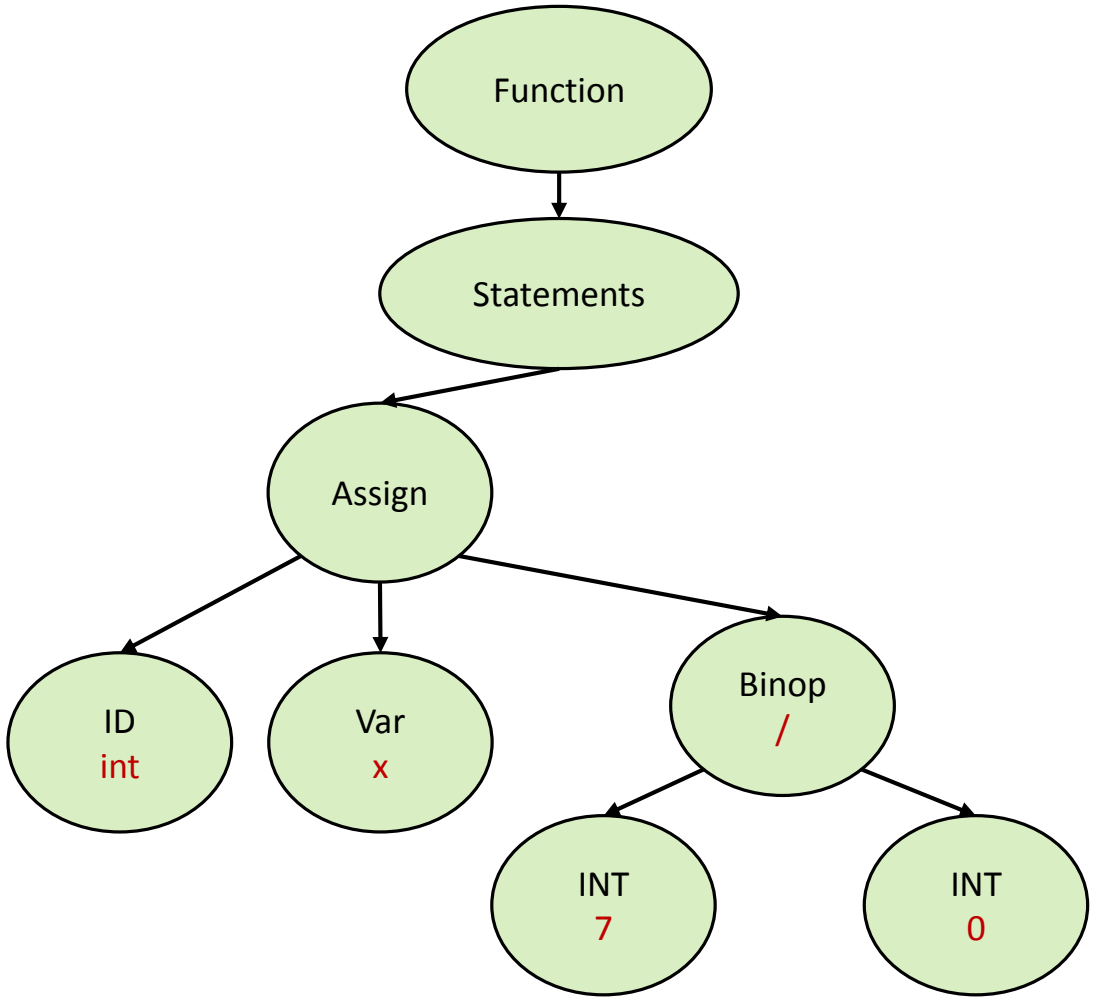
Binary Operations

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



Binary Operations

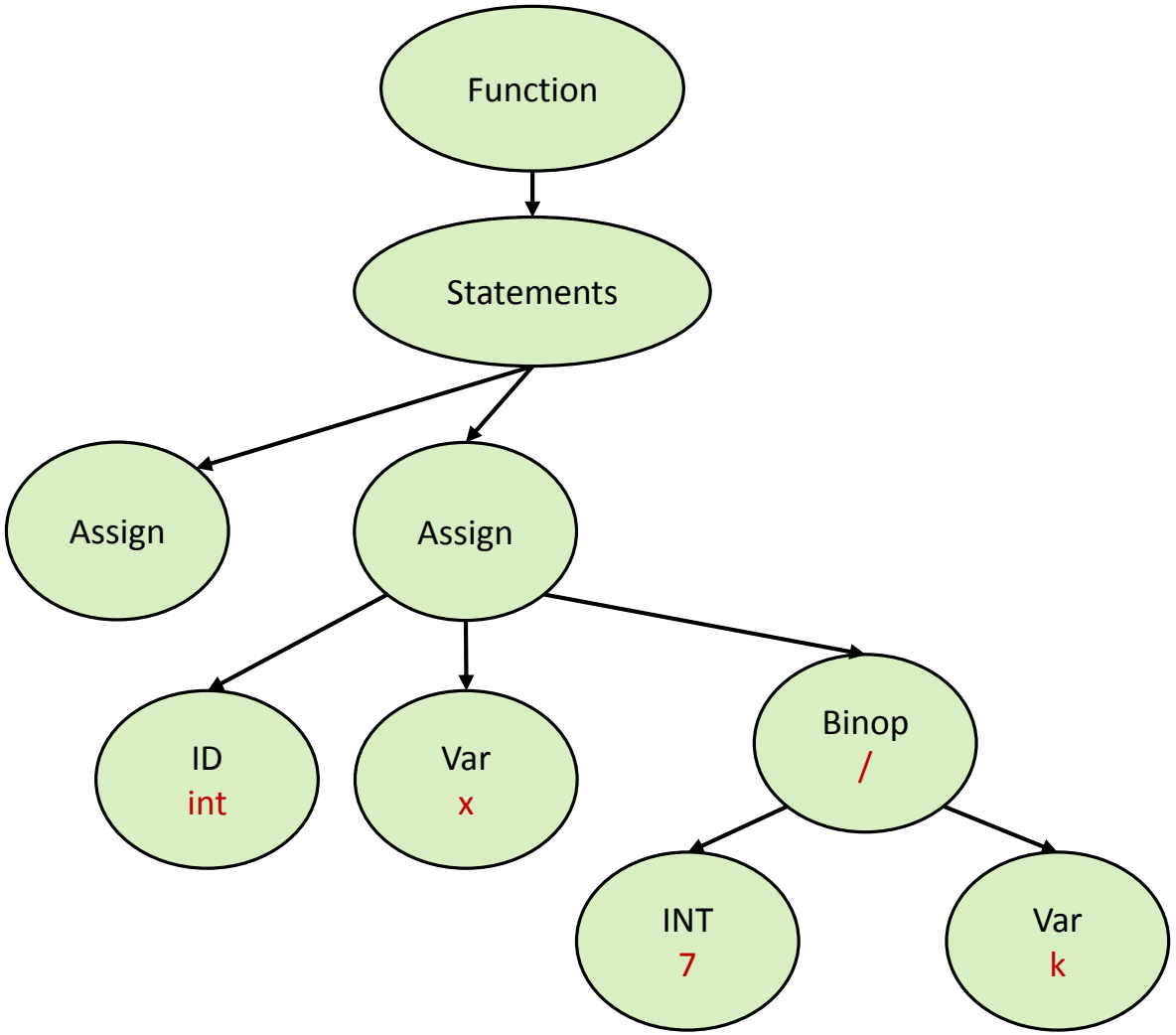
```
void main() {
  int x = 7 / 0;
}
```



Invalid

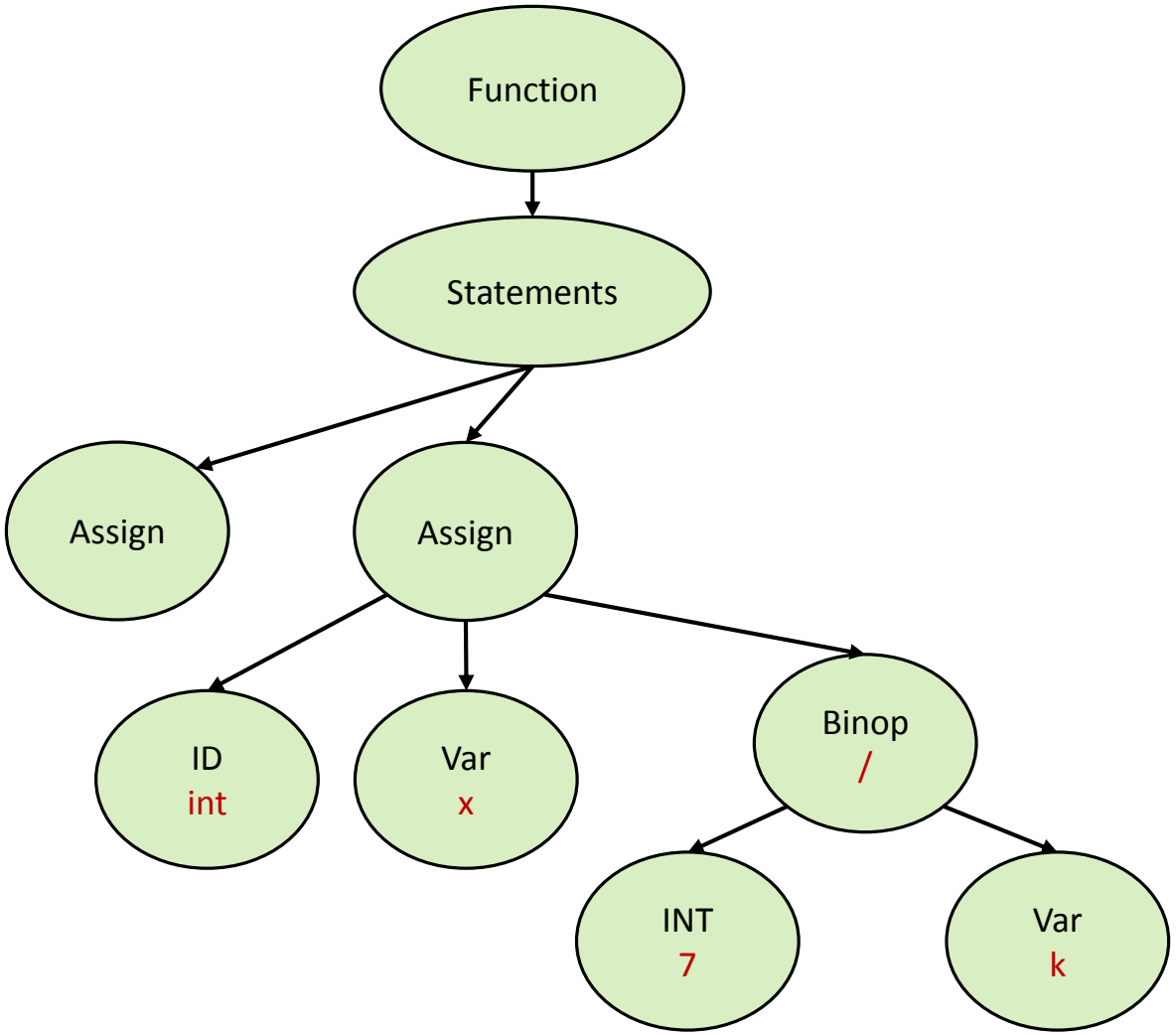
Binary Operations

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



Binary Operations

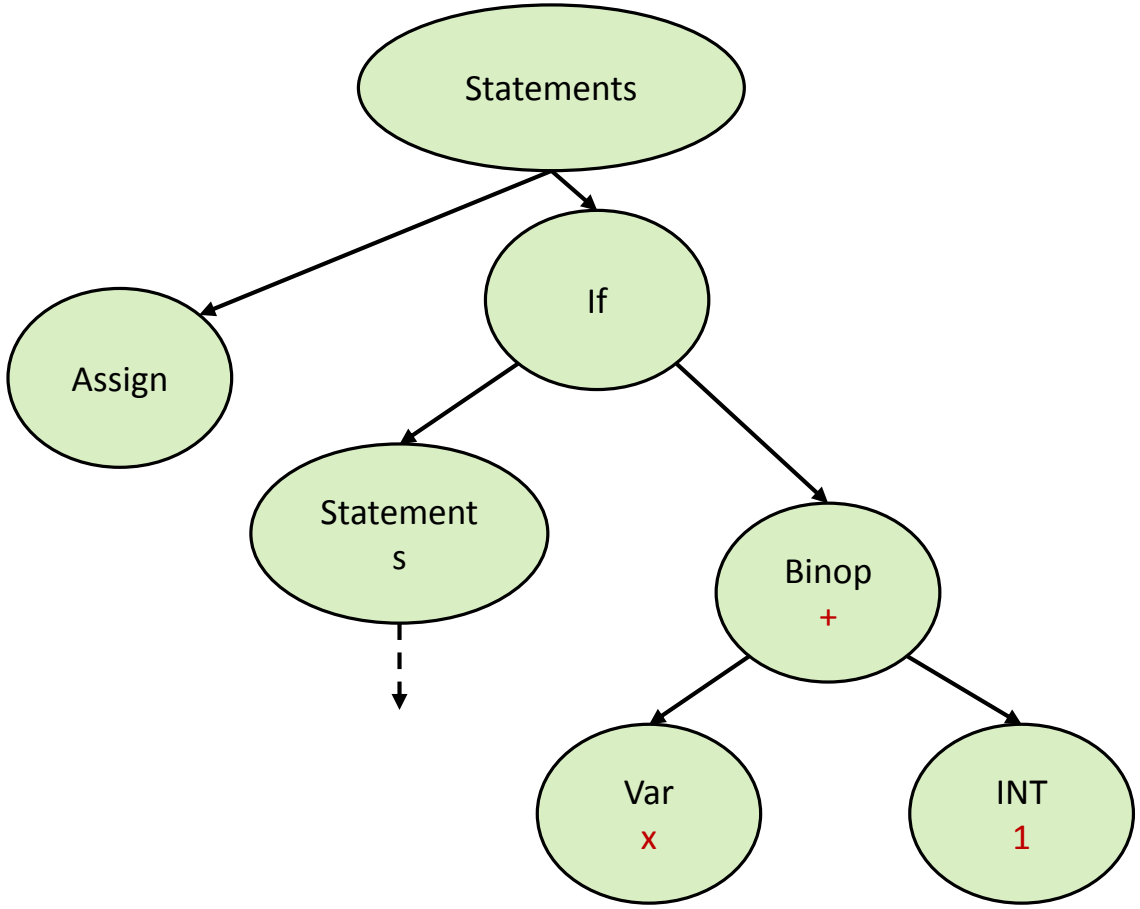
```
void main() {
  int k = 0;
  int x = 7 / k;
}
```



Depends

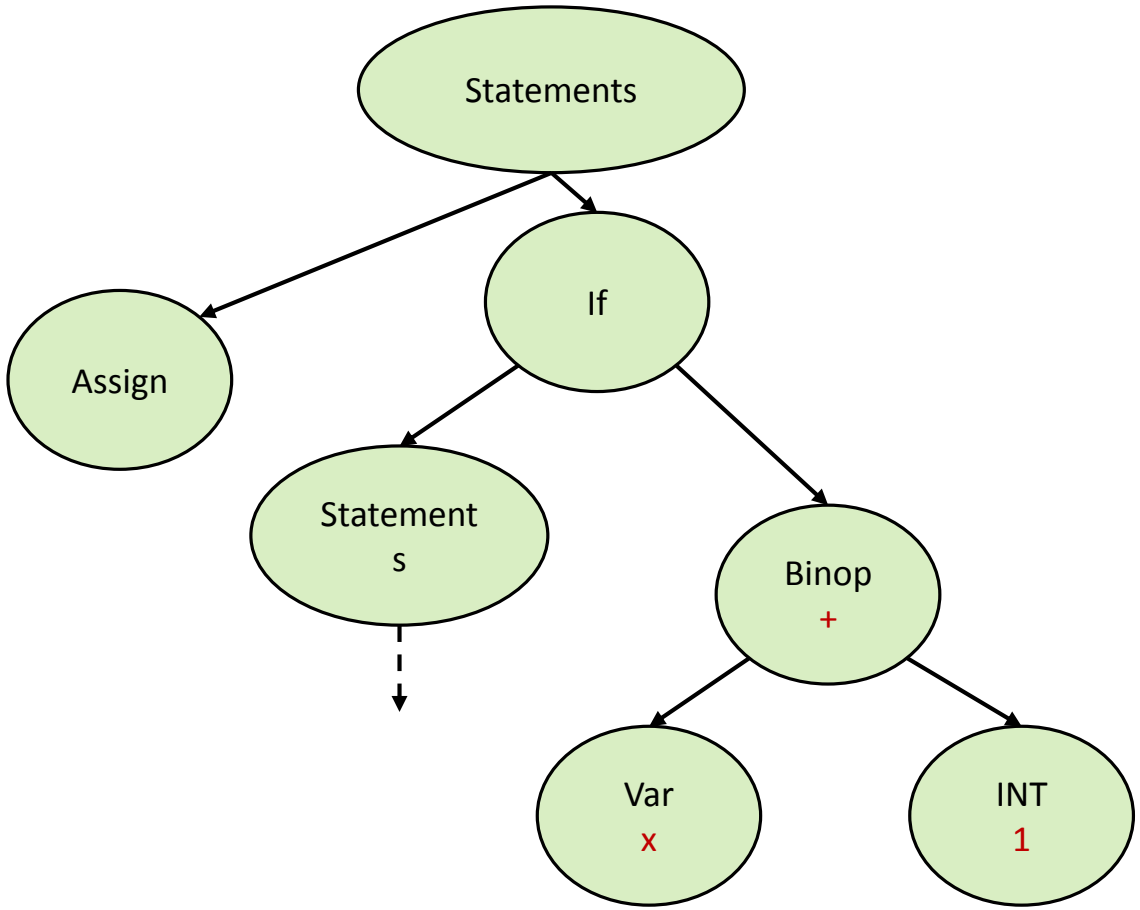
If, While, …

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



If, While, …

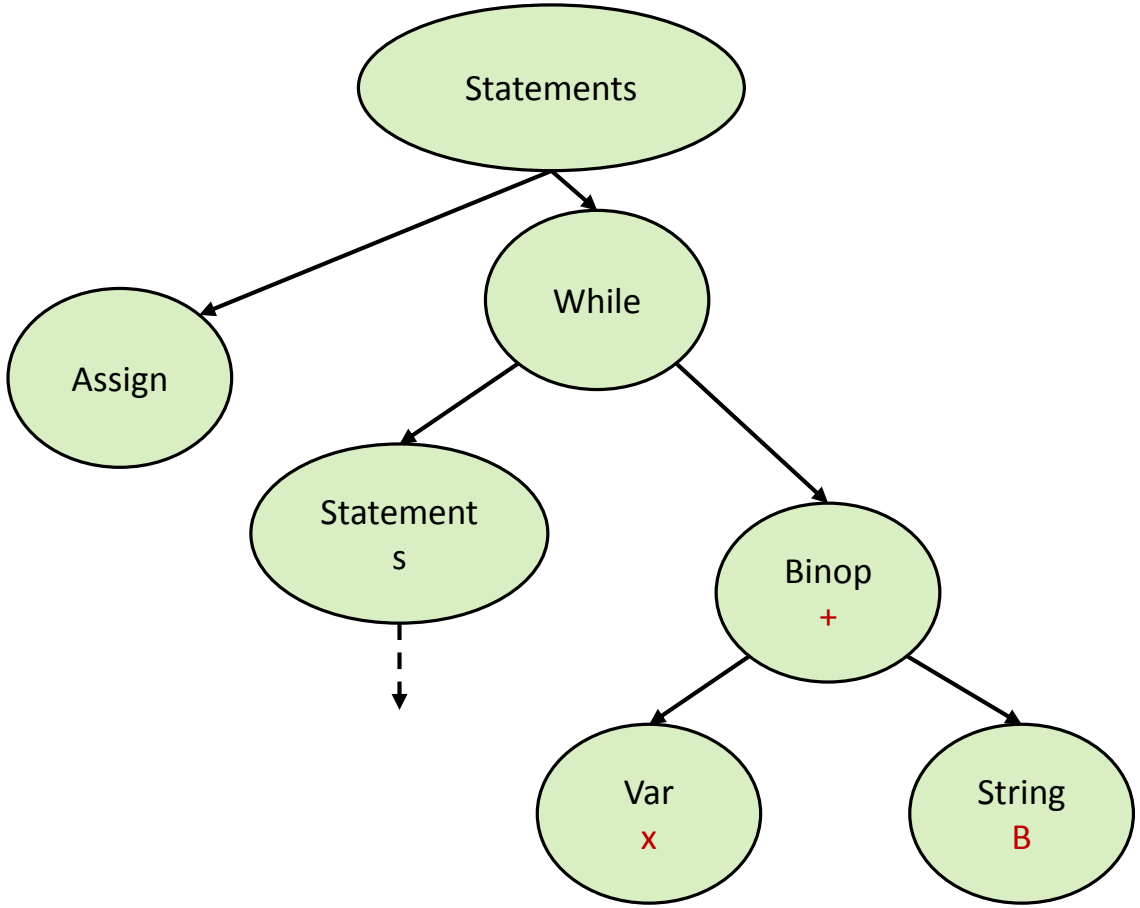
```
void main() {
  int x = 1;
  if (x + 1) {
    int z = 2;
  }
}
```



Valid

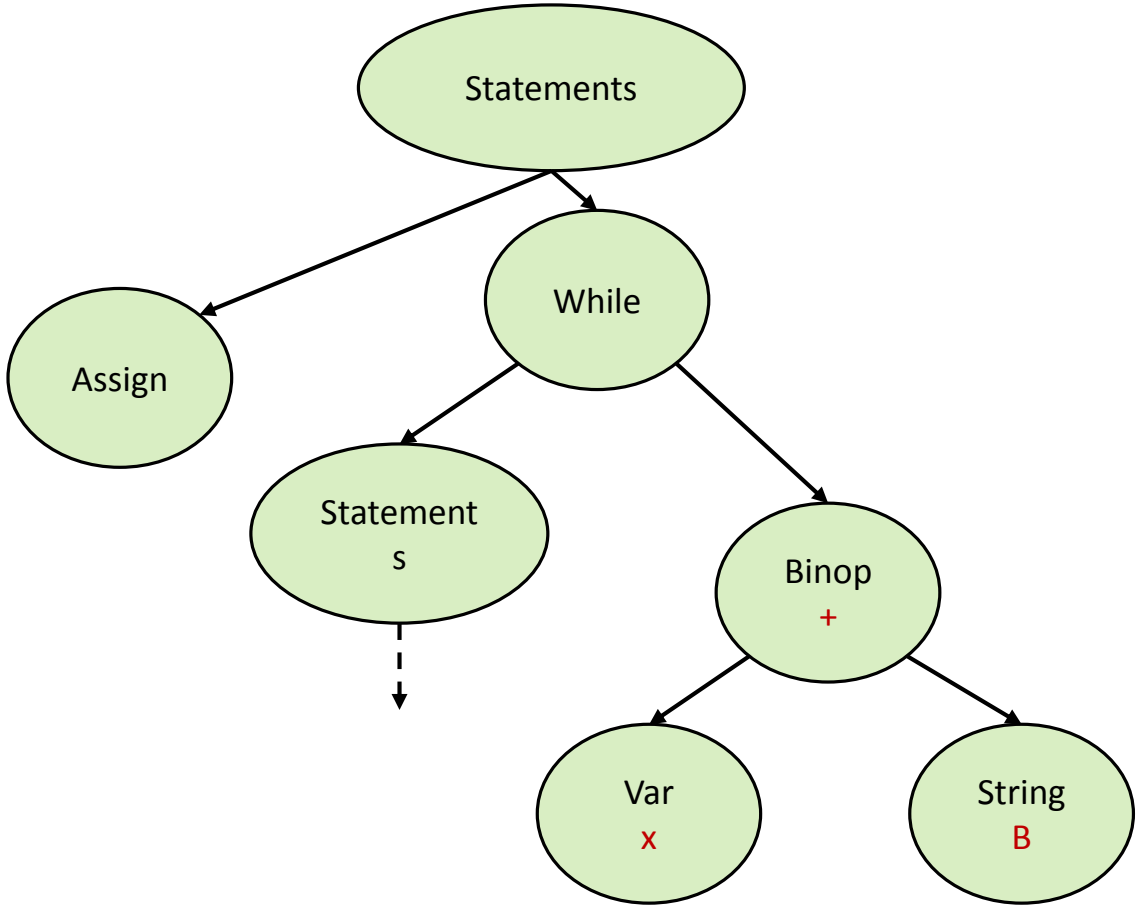
If, While, …

```
void main() {
  string x = "A";
  while (x + "B") {
    int z = 2;
  }
}
```



If, While, …

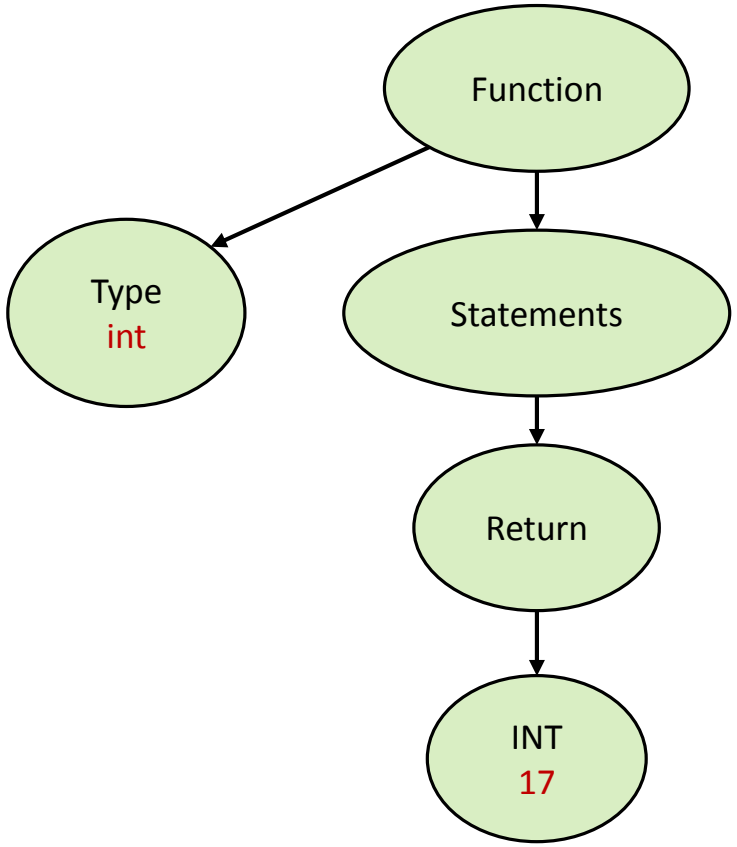
```
void main() {
  string x = "A";
  while (x + "B") {
    int z = 2;
  }
}
```



Invalid

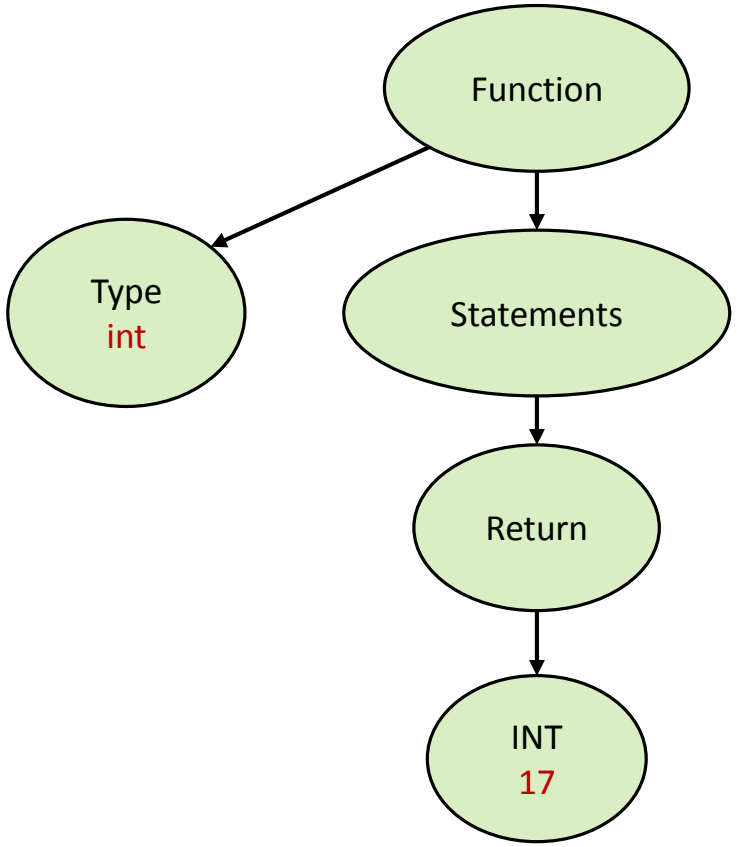
Return Statement

```
int main() {
  return 17;
}
```



Return Statement

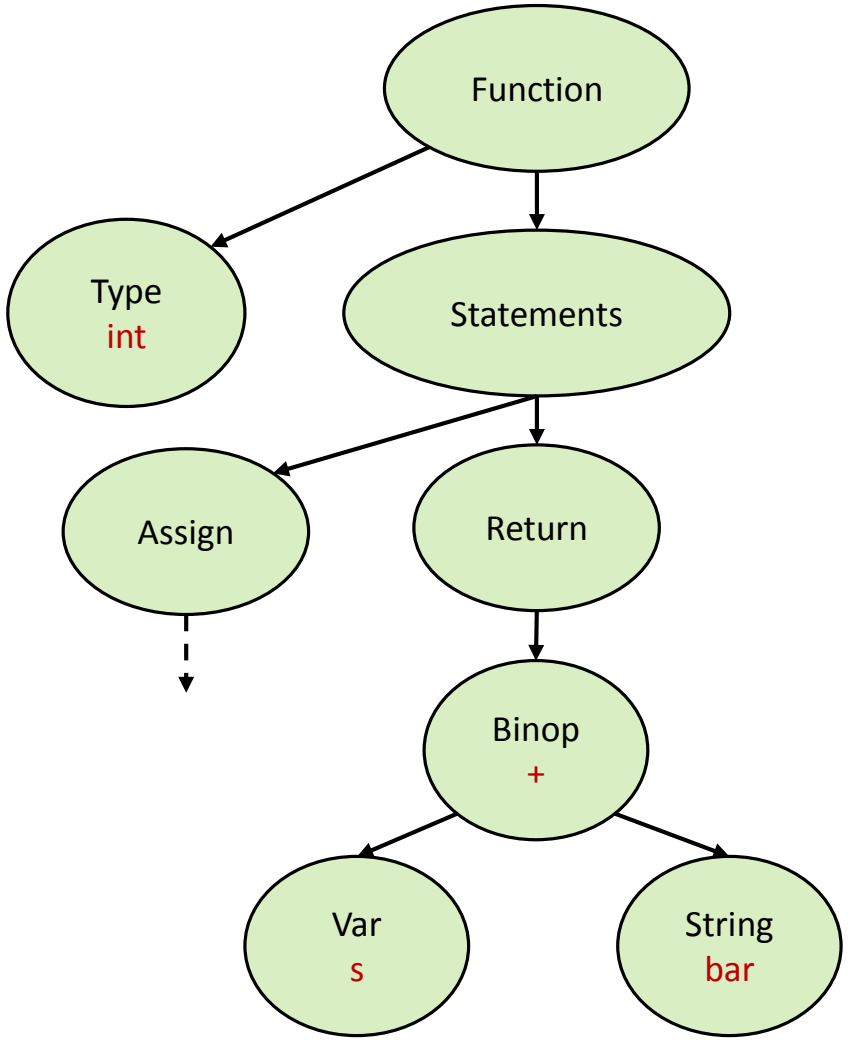
```
int main() {
  return 17;
}
```



Valid

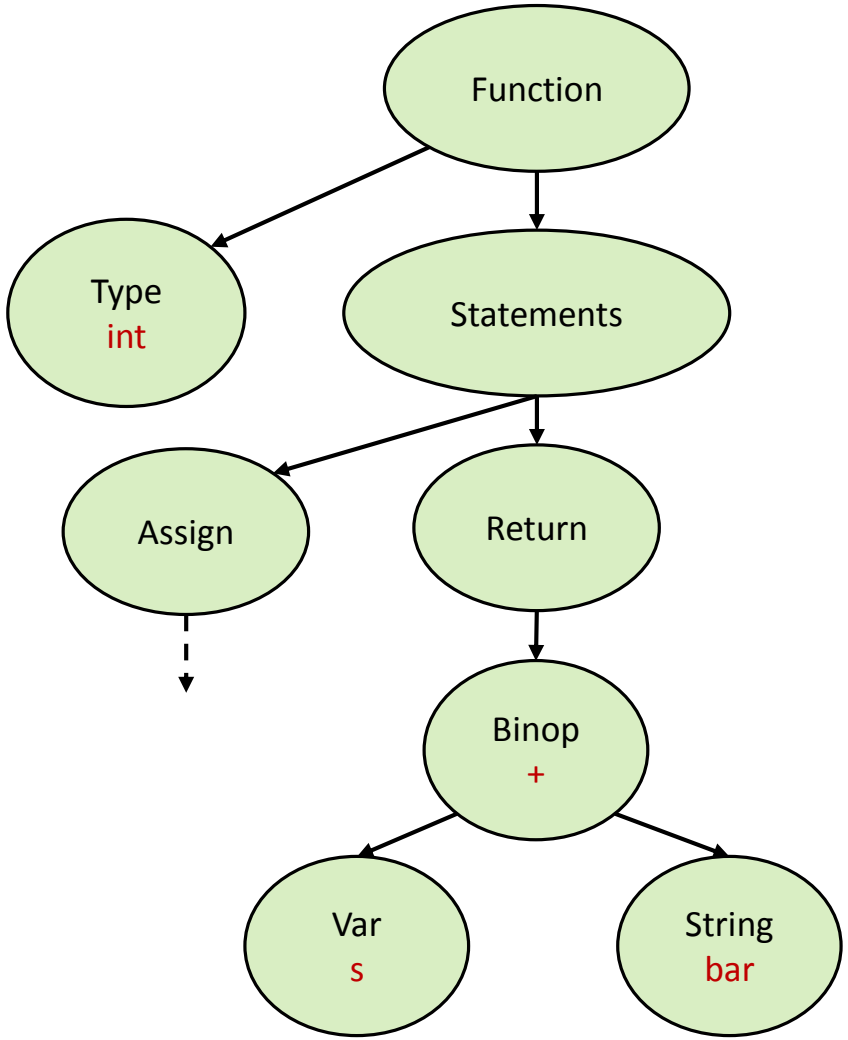
Return Statement

```
int main() {
  string s = “foo”
  return s + “bar”;
}
```



Return Statement

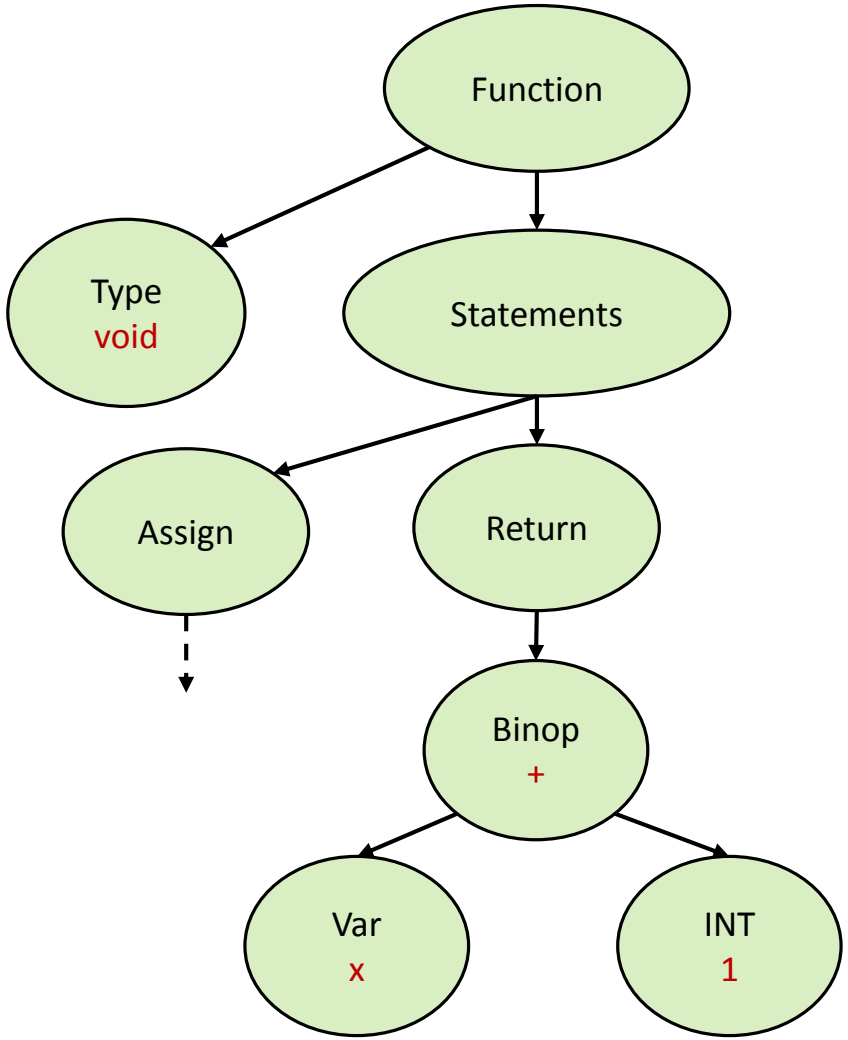
```
int main() {
  string s = "foo"
  return s + "bar";
}
```



Invalid

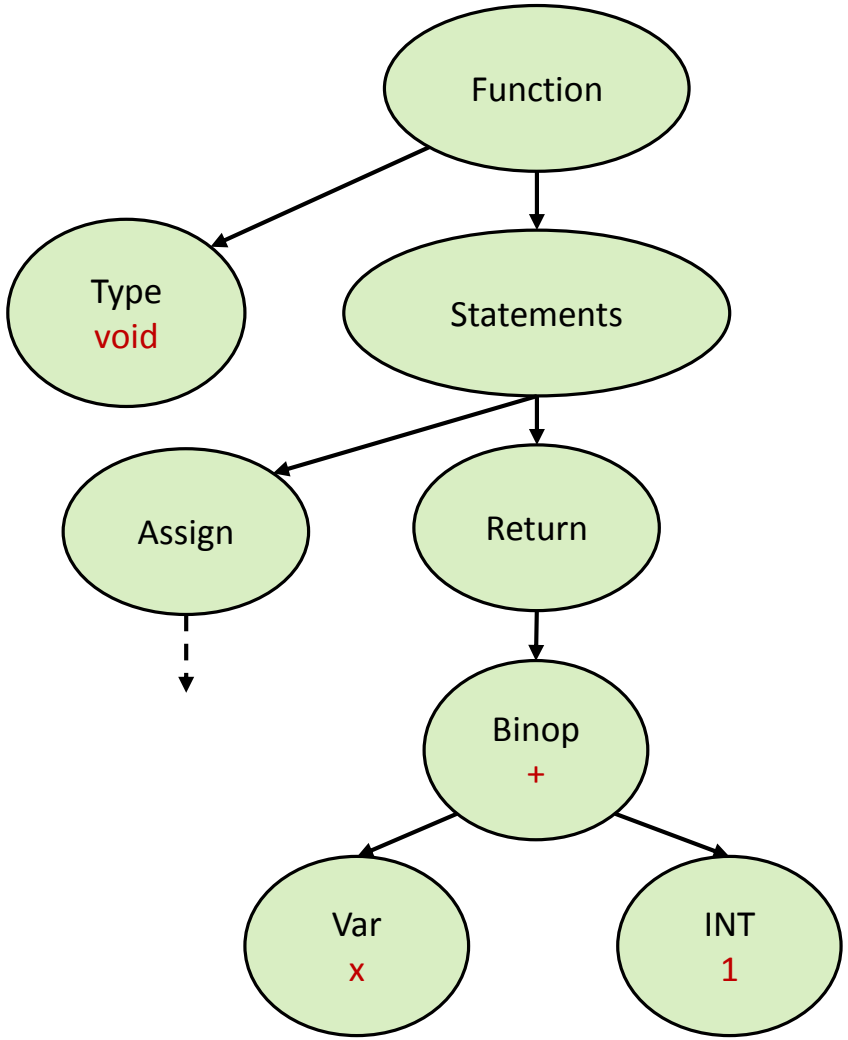
Return Statement

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



Return Statement

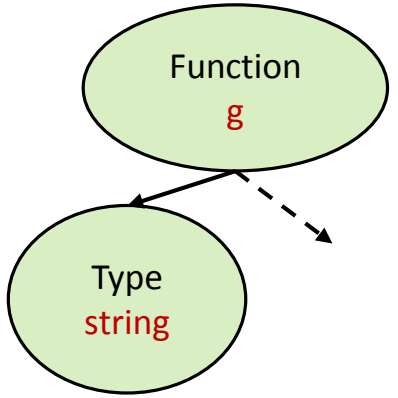
```
void main() {
  int x = 1;
  return x + 1;
}
```

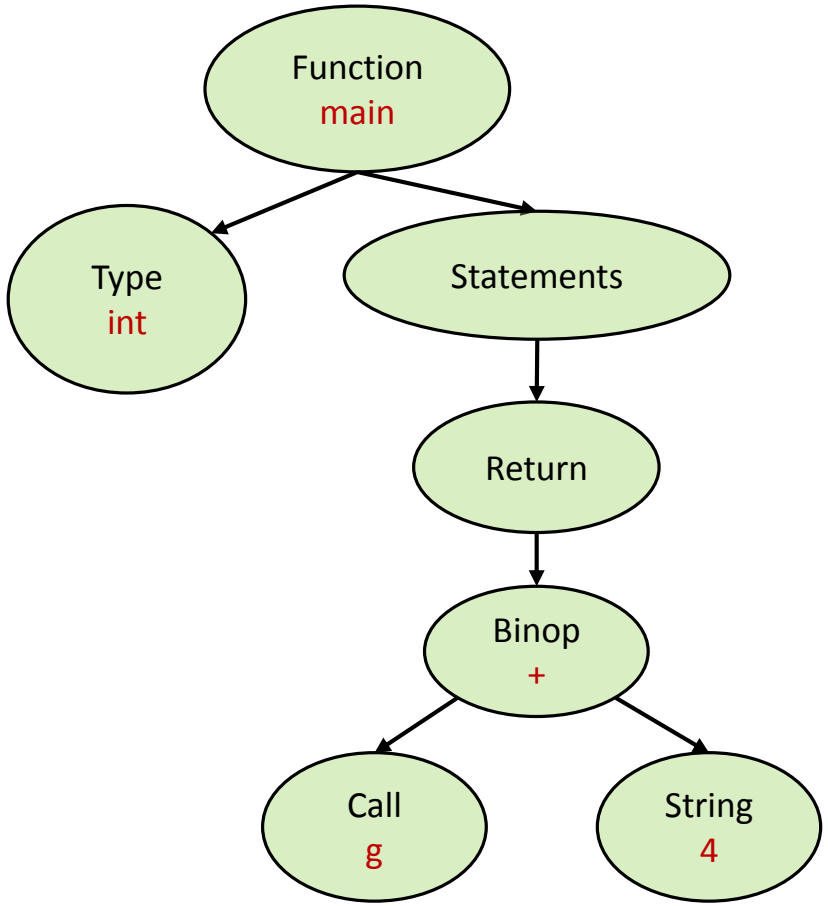


Invalid

Return Statement

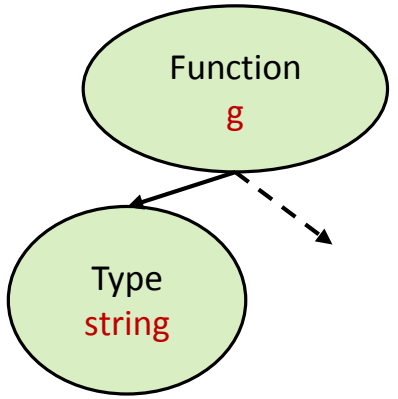
```
string g() {
  return "123";
}
int main() {
  return g() + "4";
}
```

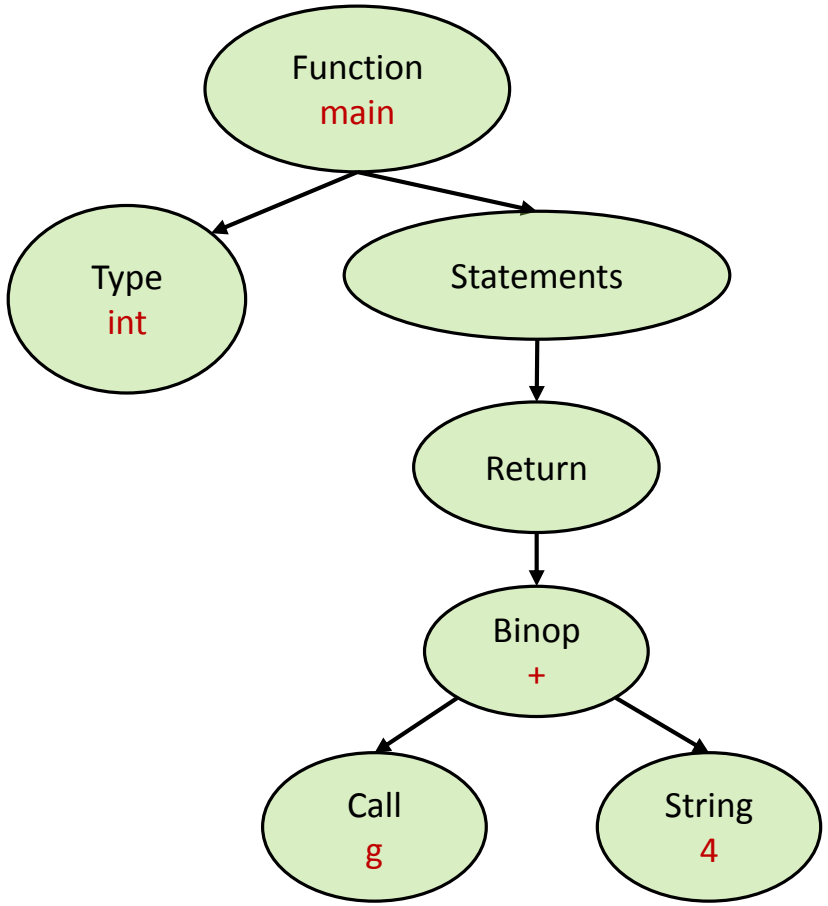




Return Statement

```
string g() {
  return "123";
}
int main() {
  return g() + "4";
}
```

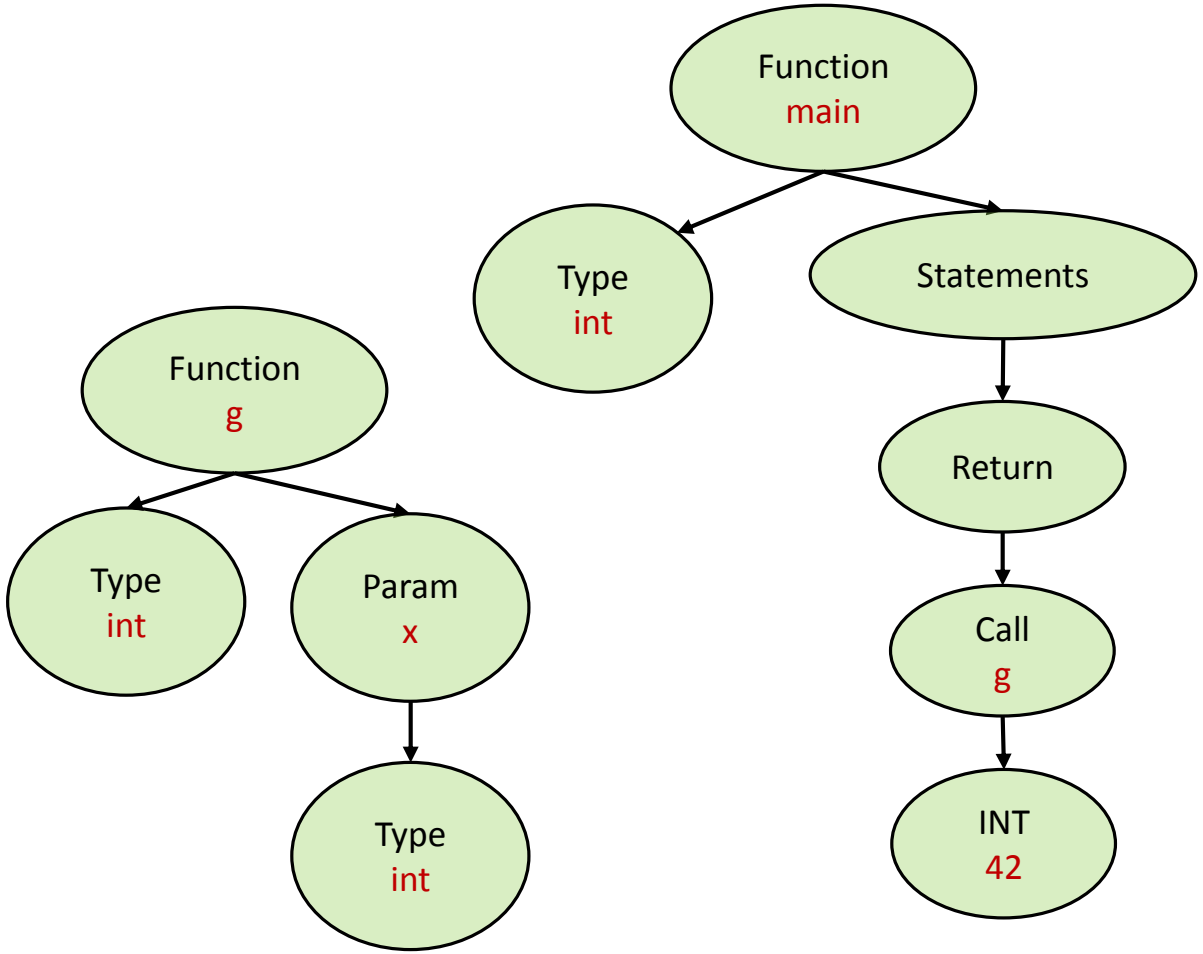




Invalid

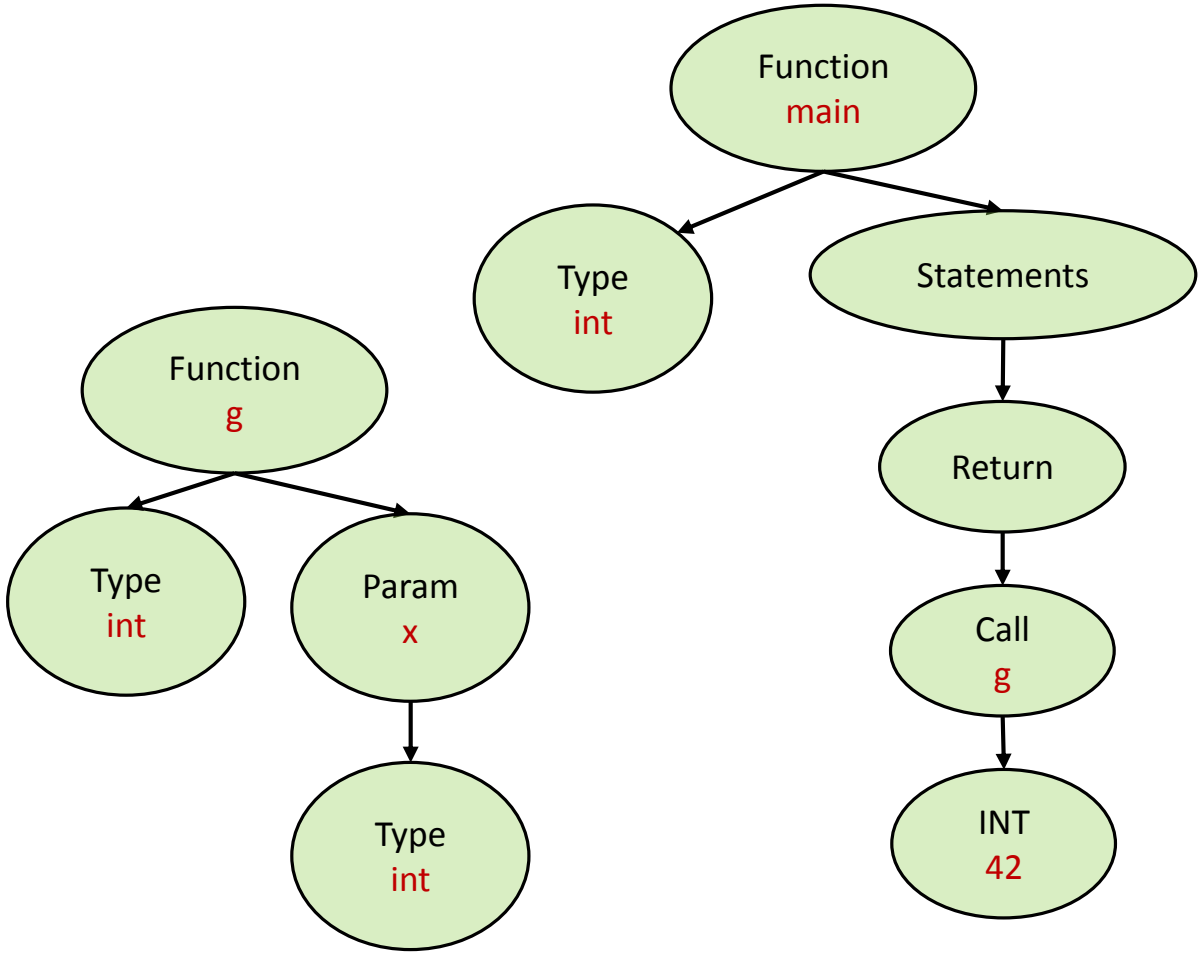
Function Calls

```
int g(int x) {
  return x + 1;
}
int main() {
  return g(42);
}
```



Function Calls

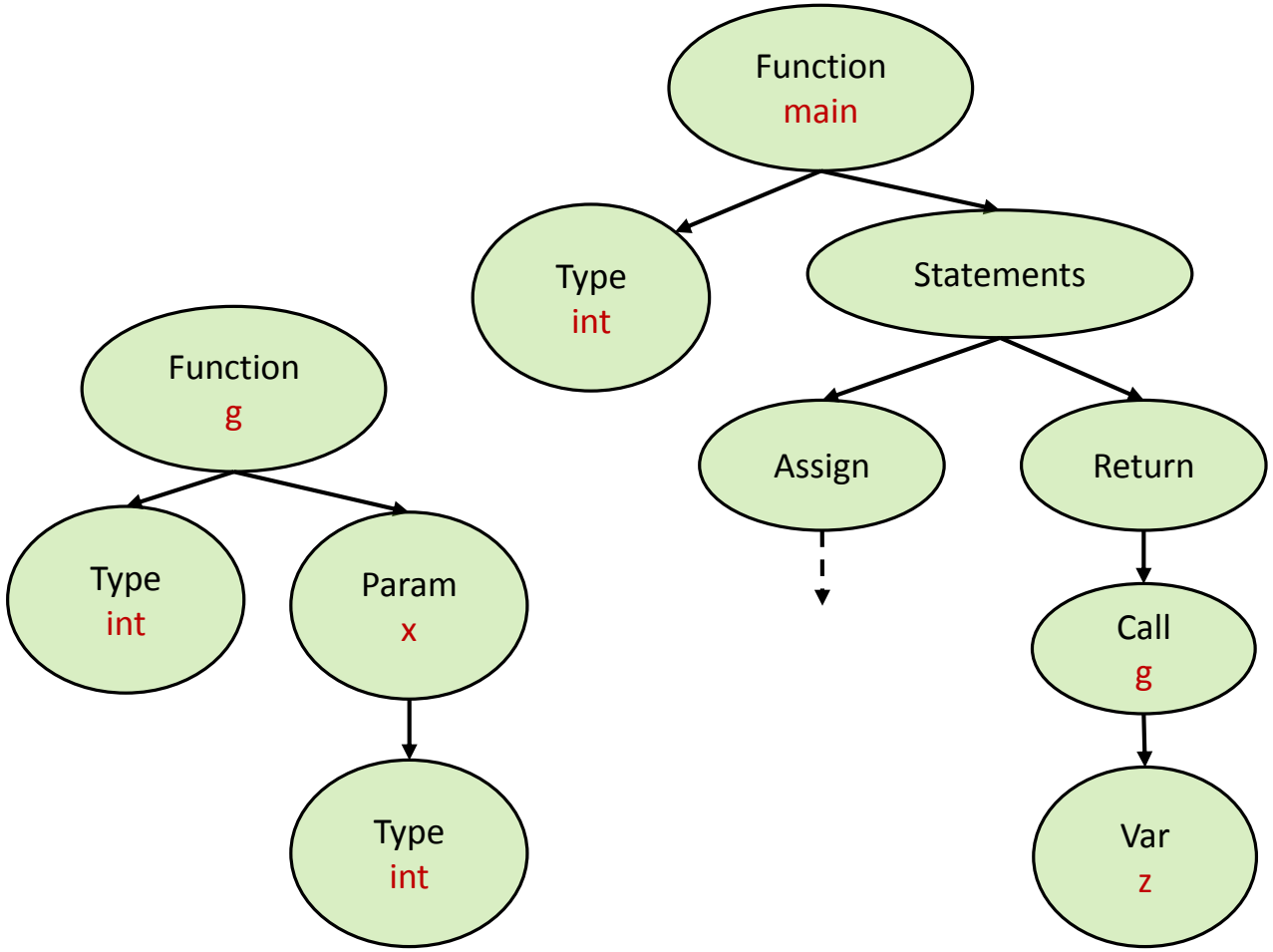
```
int g(int x) {
  return x + 1;
}
int main() {
  return g(42);
}
```



Valid

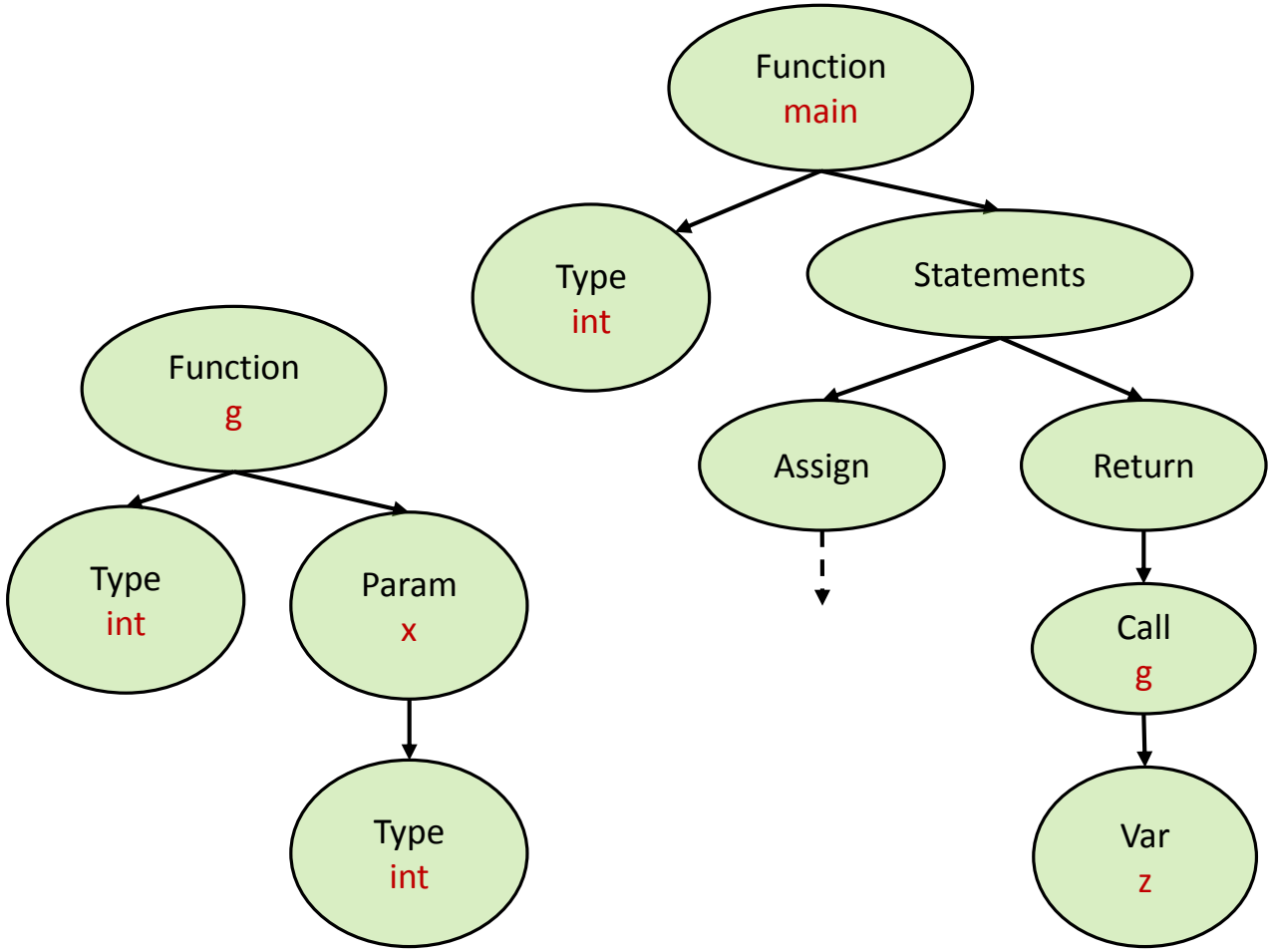
Function Calls

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



Function Calls

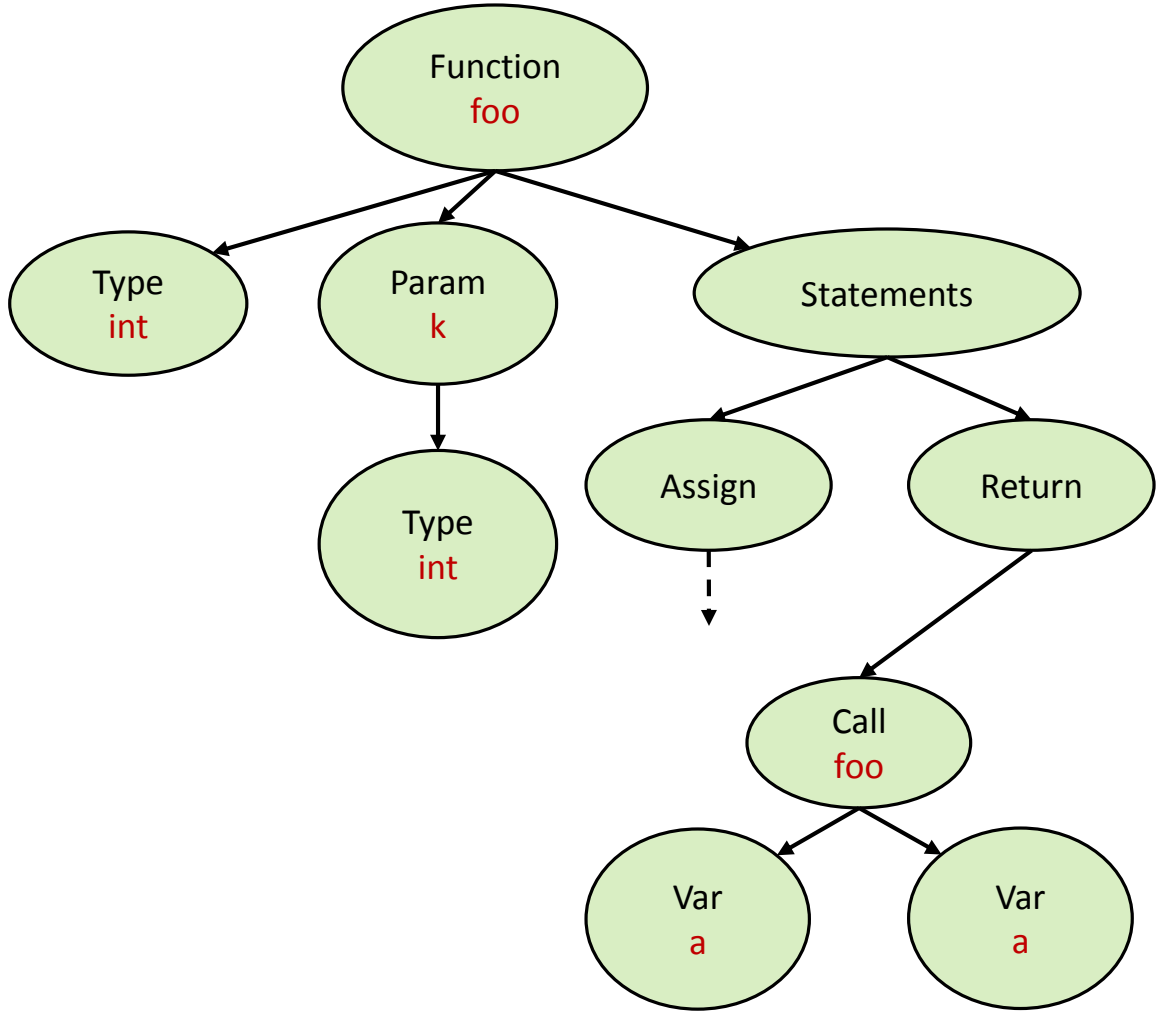
```
int g(int x) {
  return x + 1;
}
int main() {
  string z = "..."
  return g(z);
}
```



Invalid

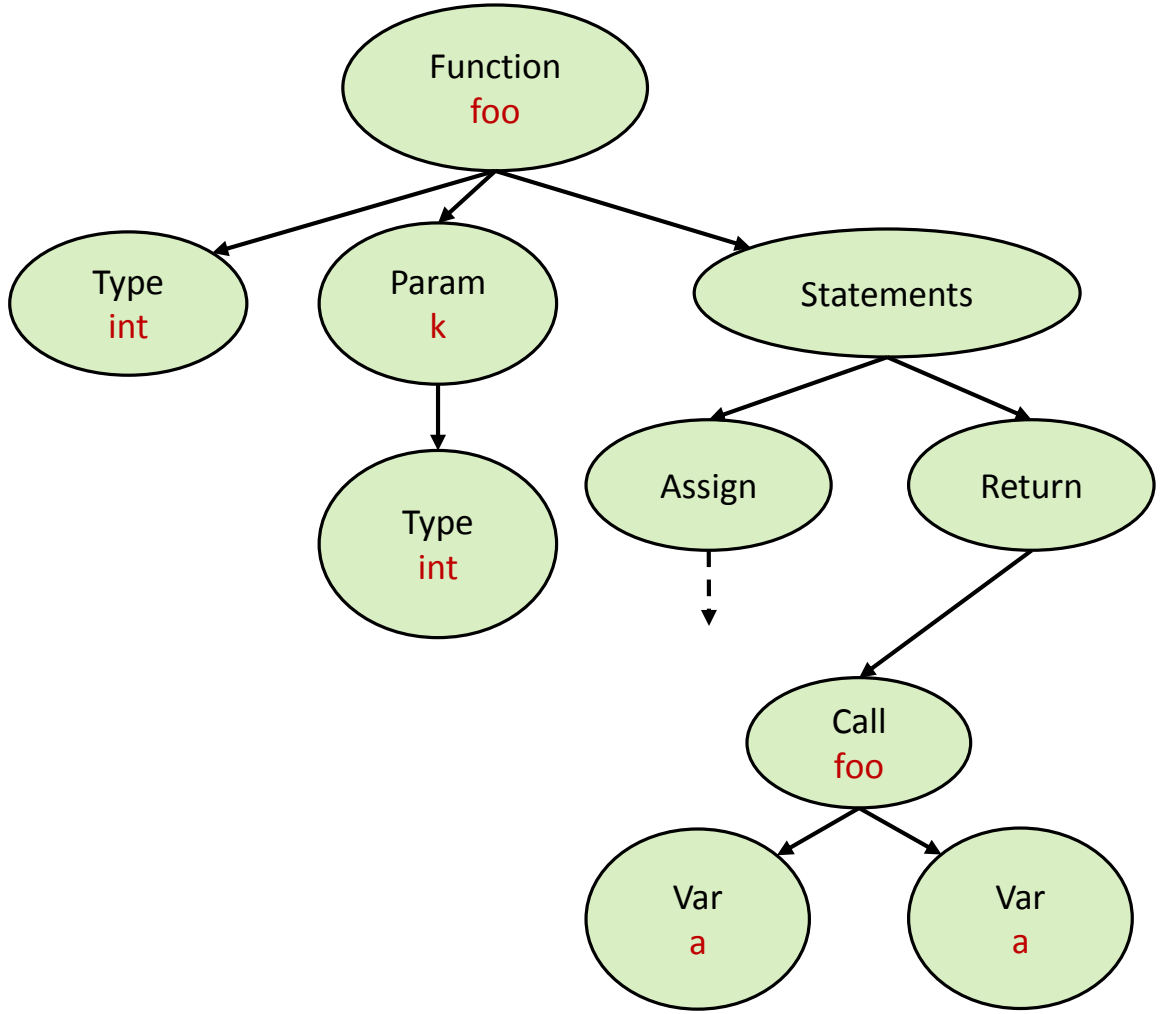
Function Calls

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



Function Calls

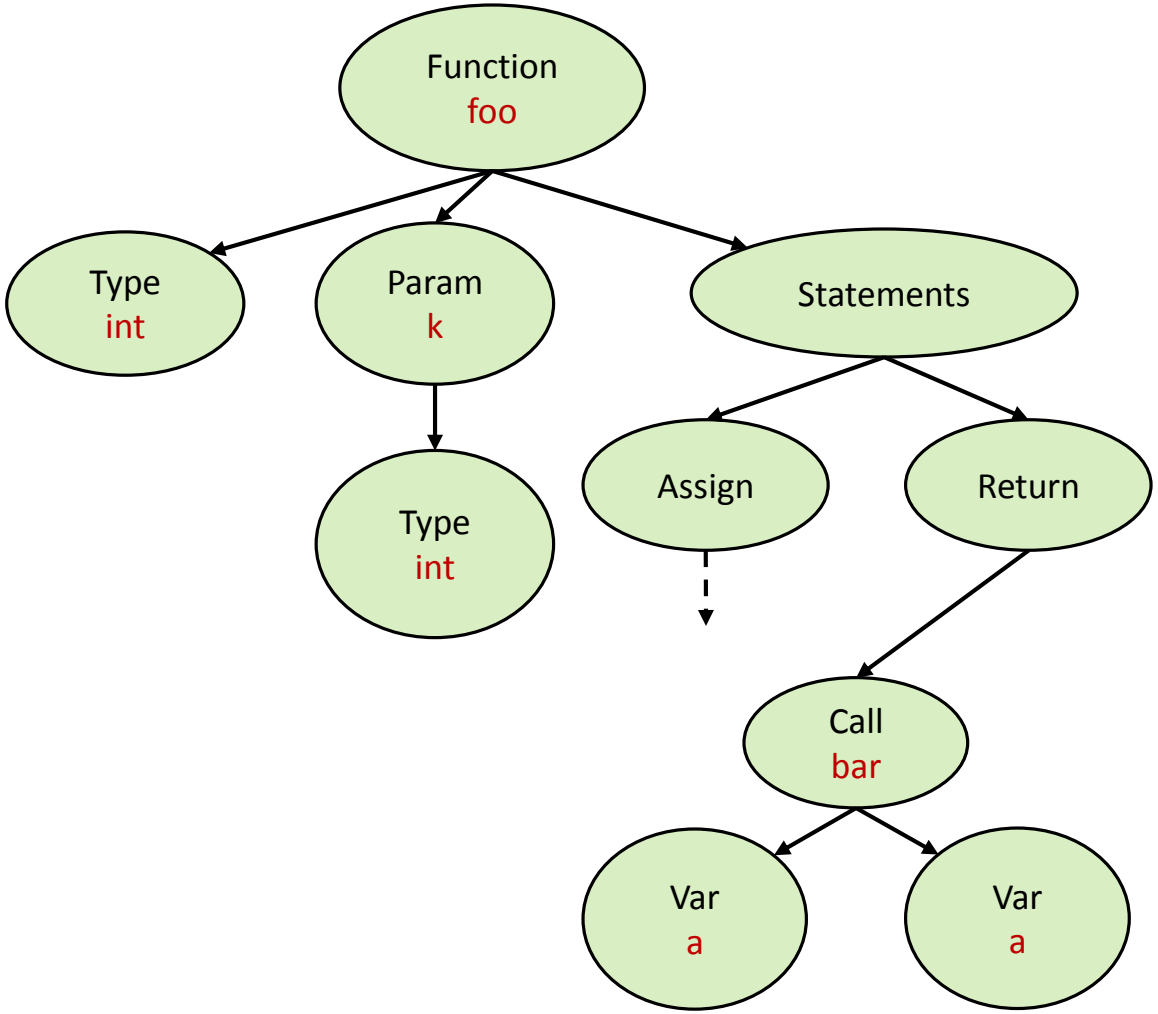
```
int foo(int k) {
  int a = k * 10;
  return foo(a, a);
}
```



Invalid

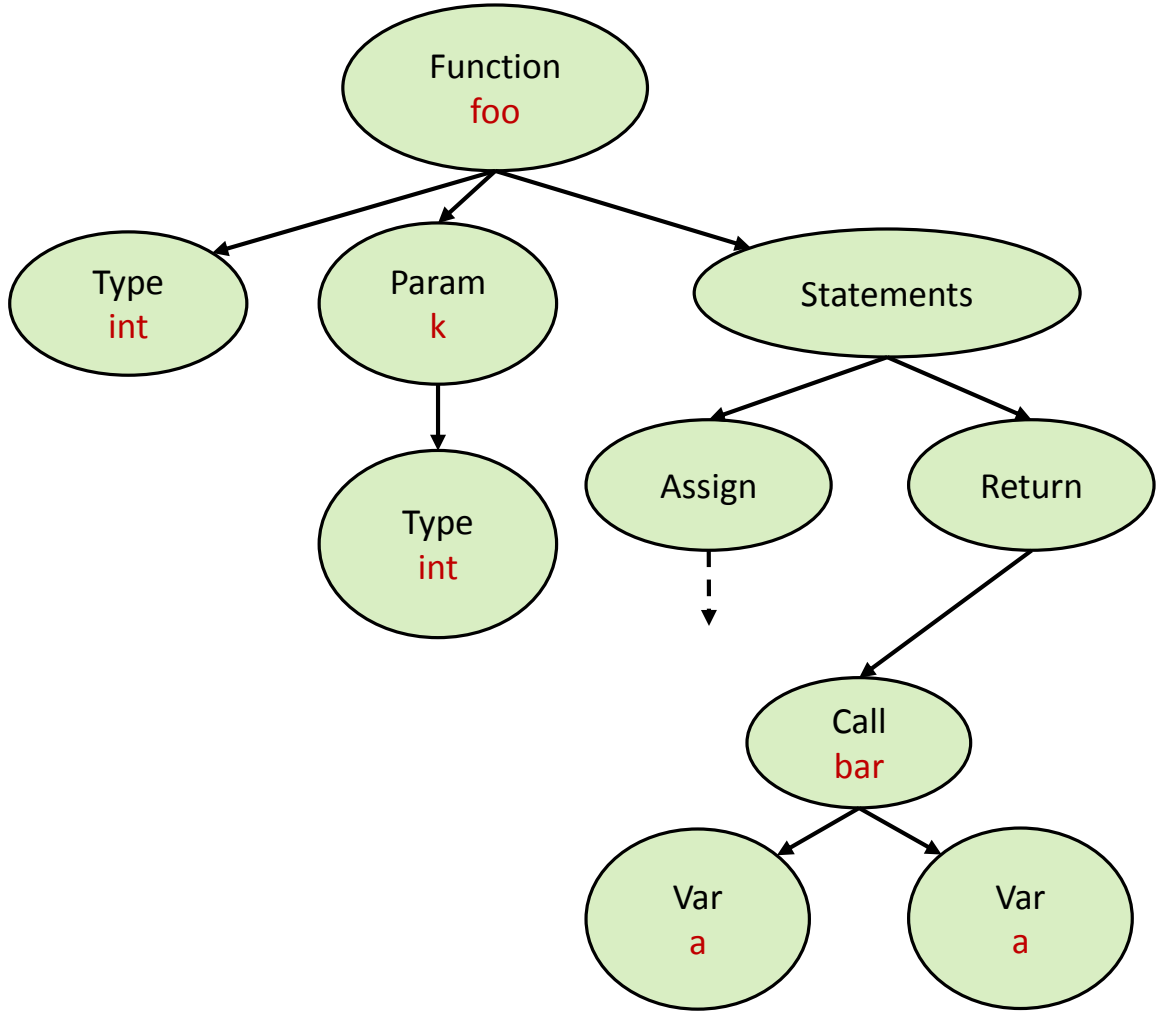
Function Calls

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



Function Calls

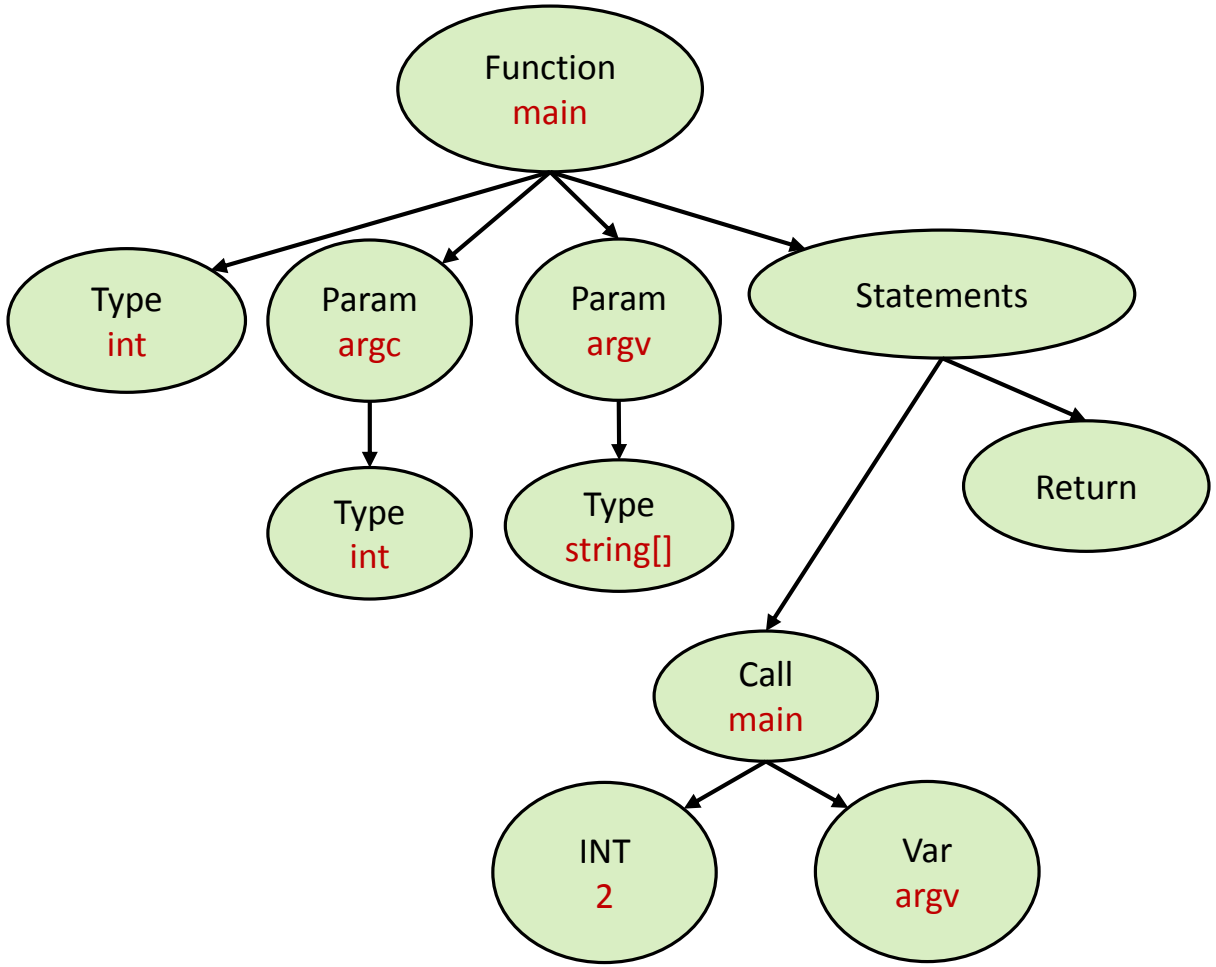
```
int foo(int k) {
  int a = k * 10;
  return bar(a, a);
}
```



Invalid

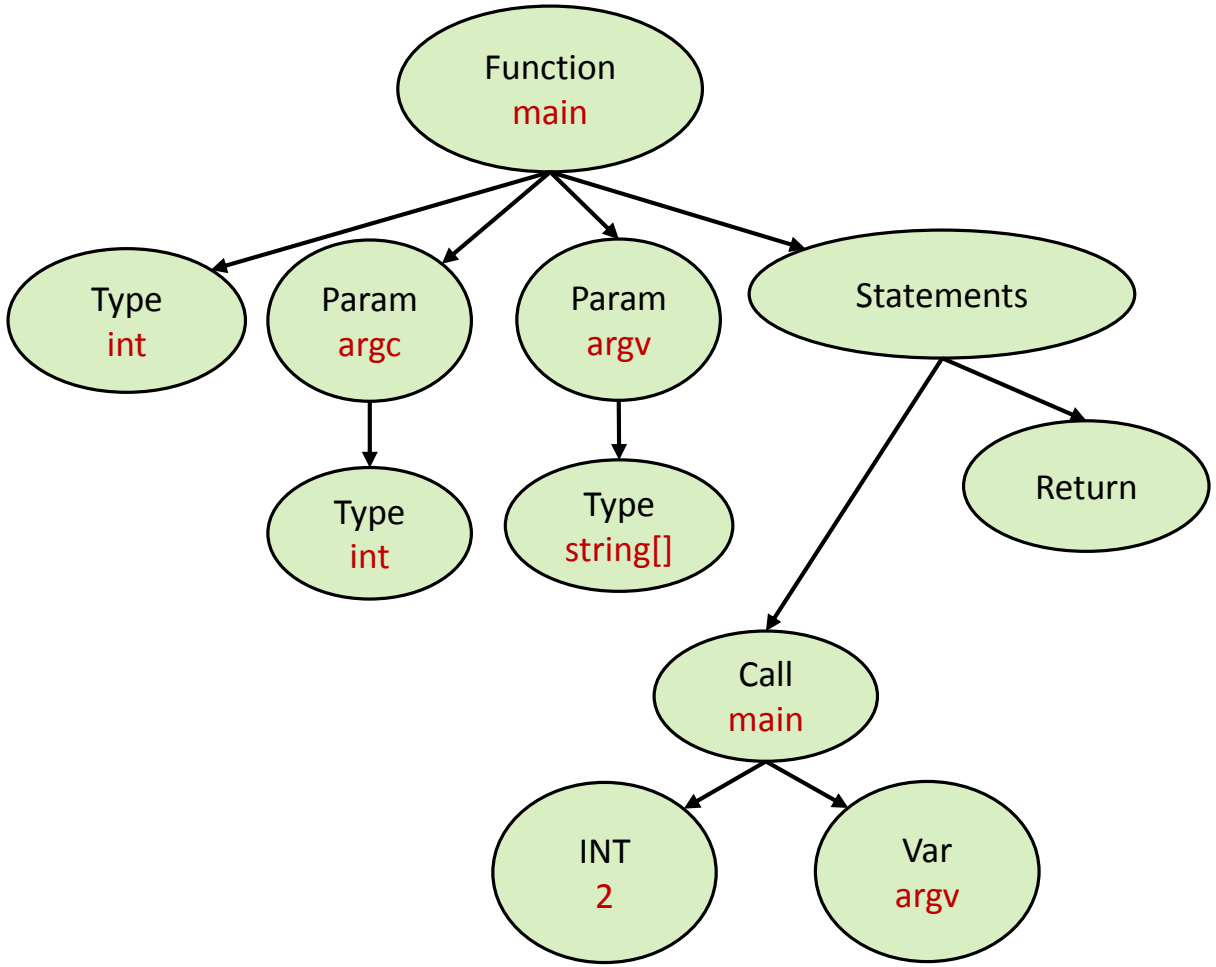
Function Calls

```
int main(int argc,
         string argv[]){
  main(2, argv);
  return 0;
}
```



Function Calls

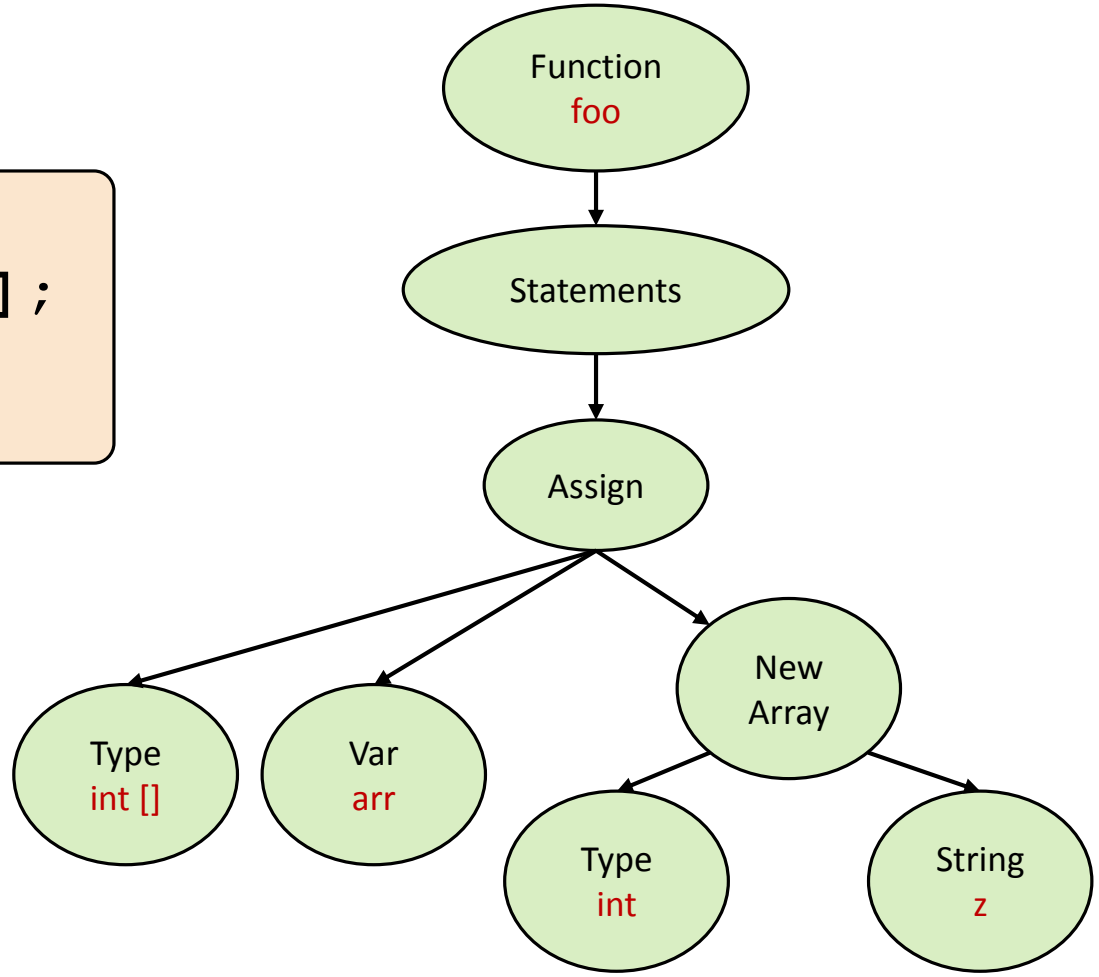
```
int main(int argc,
         string argv[]){
  main(2, argv);
  return 0;
}
```



Valid

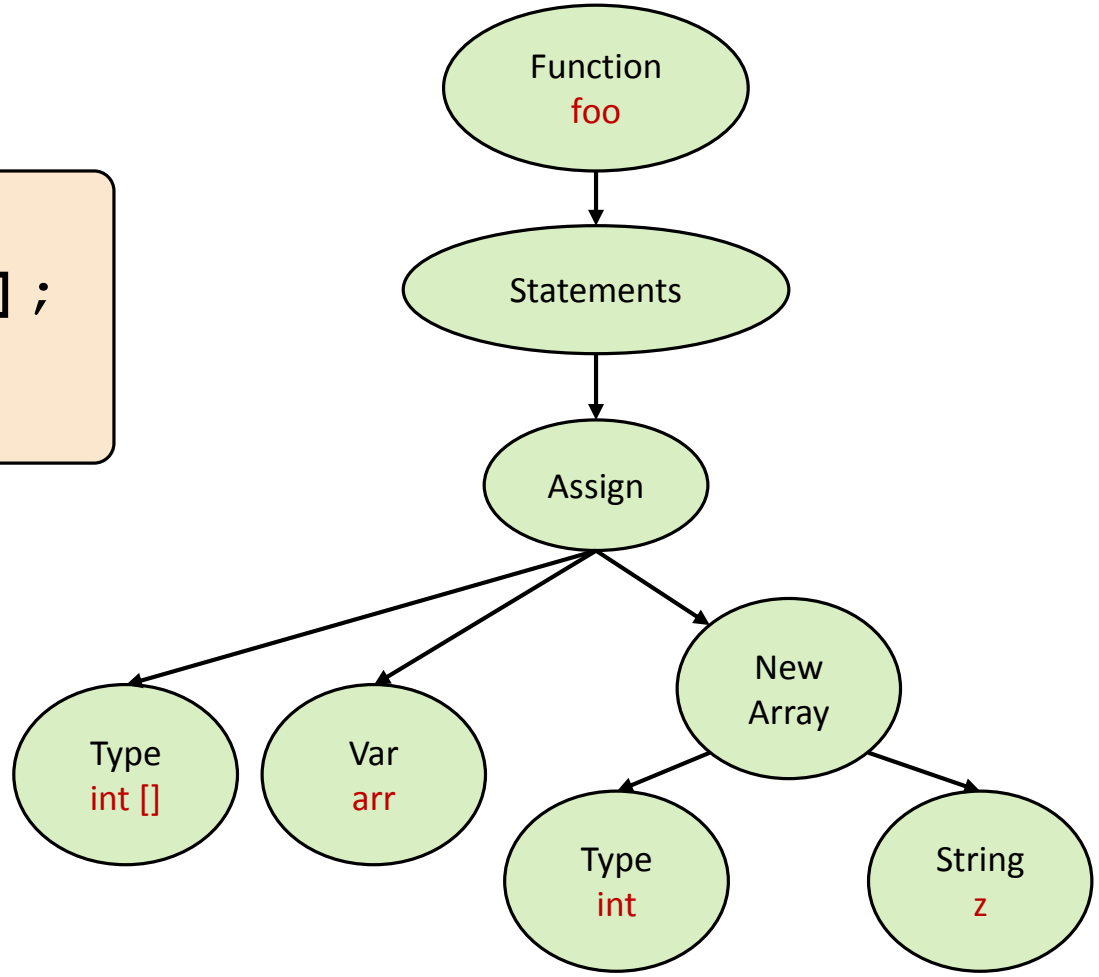
Arrays

```
void foo(void) {
  int[] arr = new int[“z”];
}
```



Arrays

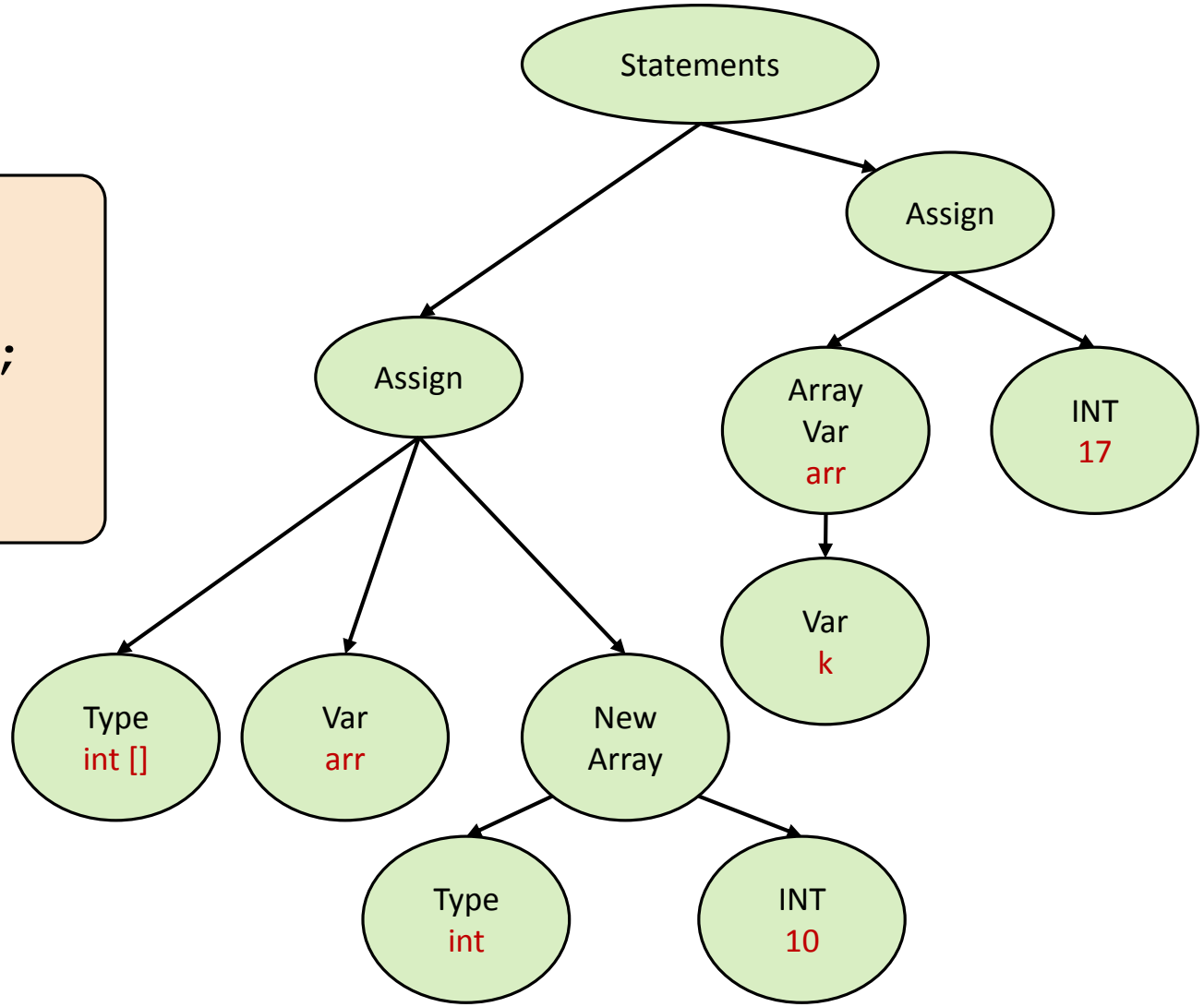
```
void foo(void) {
  int[] arr = new int[“z”];
}
```



Invalid

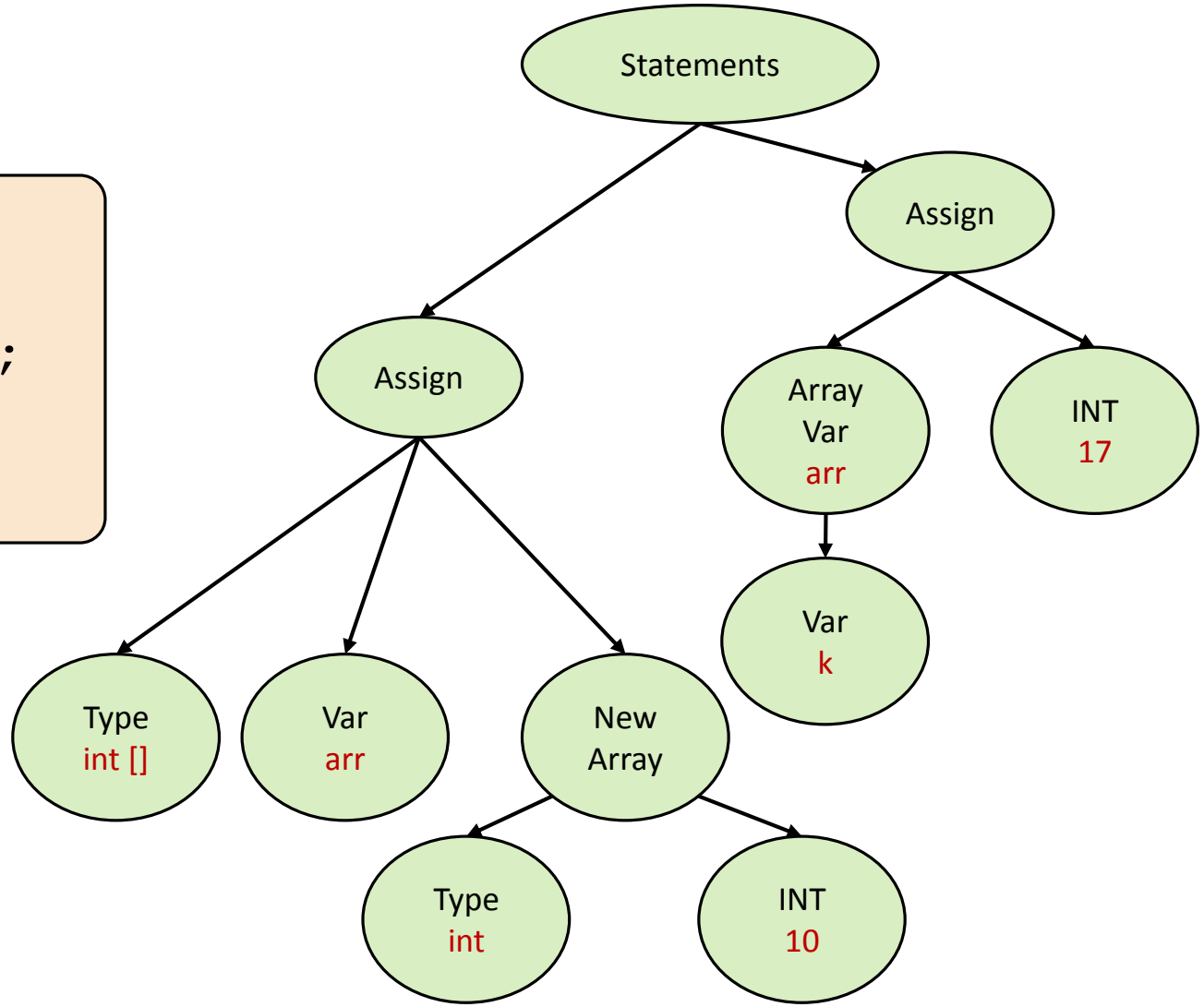
Arrays

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



Arrays

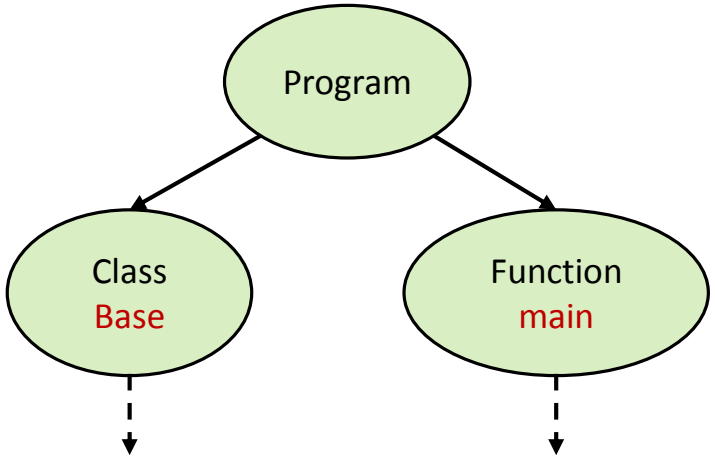
```
void foo(int d) {
  int k = 3;
  int[] arr = new int[10];
  arr[k] = 17;
}
```



Valid

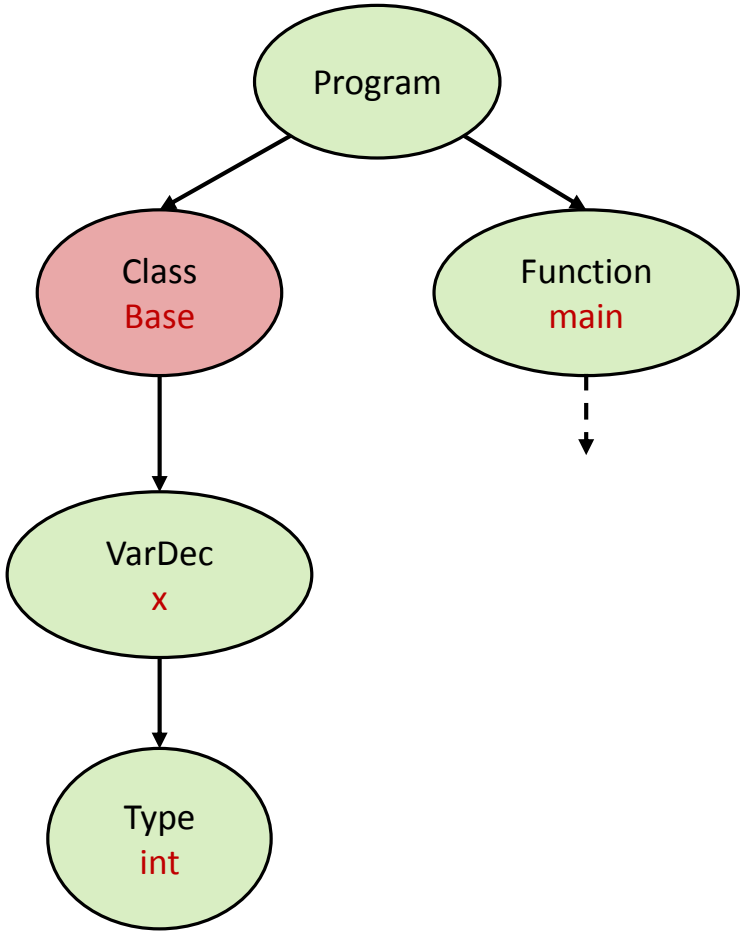
Classes

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



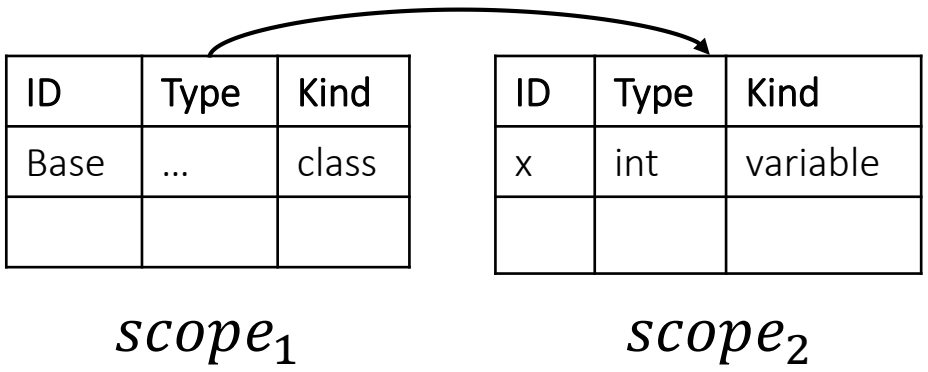
Classes

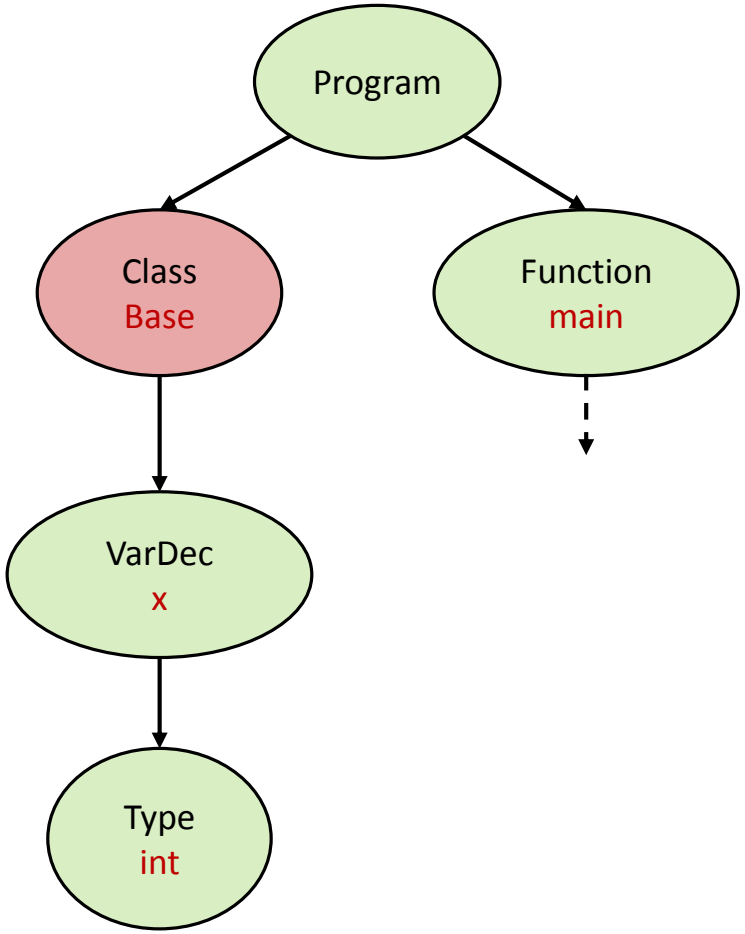
```
class Base {
  int x;
}
void main() {
  Base o = new Base;
  o.y = 1;
}
```



Classes

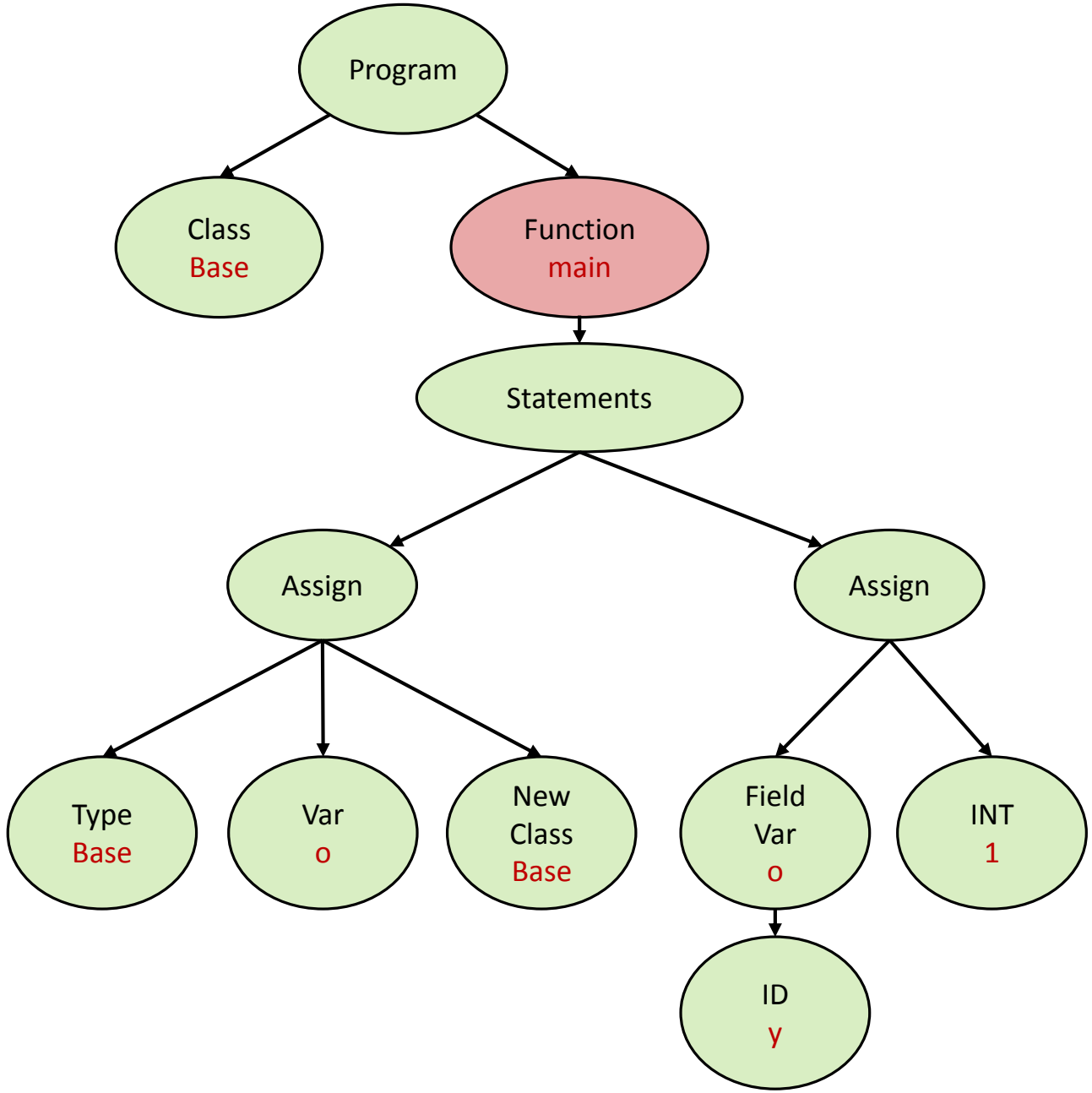
```
class Base {
  int x;
}
void main() {
  Base o = new Base;
  o.y = 1;
}
```





Classes

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



Classes

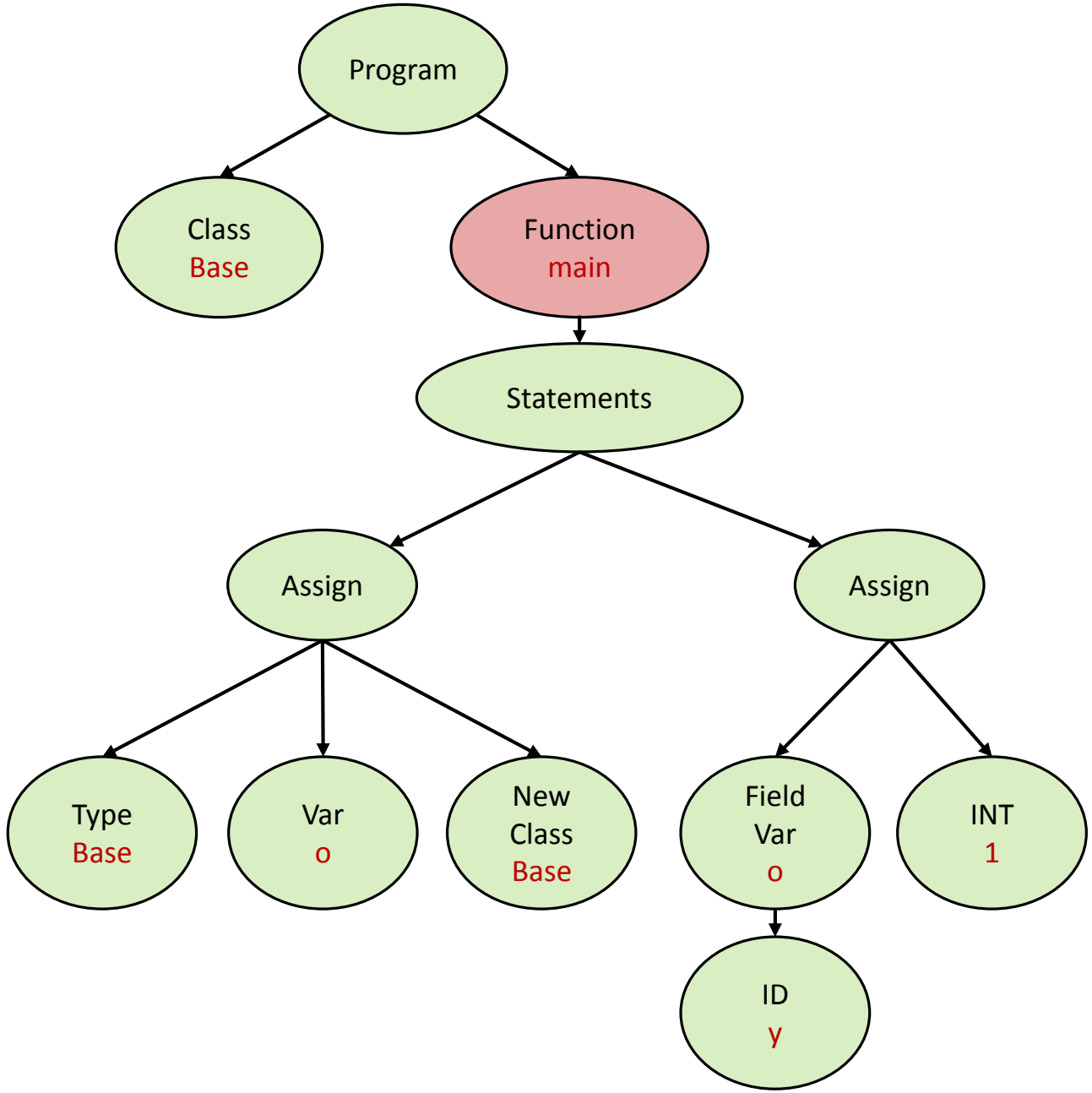
```
class Base {
  int x;
}
void main() {
  Base o = new Base;
  o.y = 1;
}
```

ID	Type	Kind
Base	…	class
main	…	function

$scope_1$

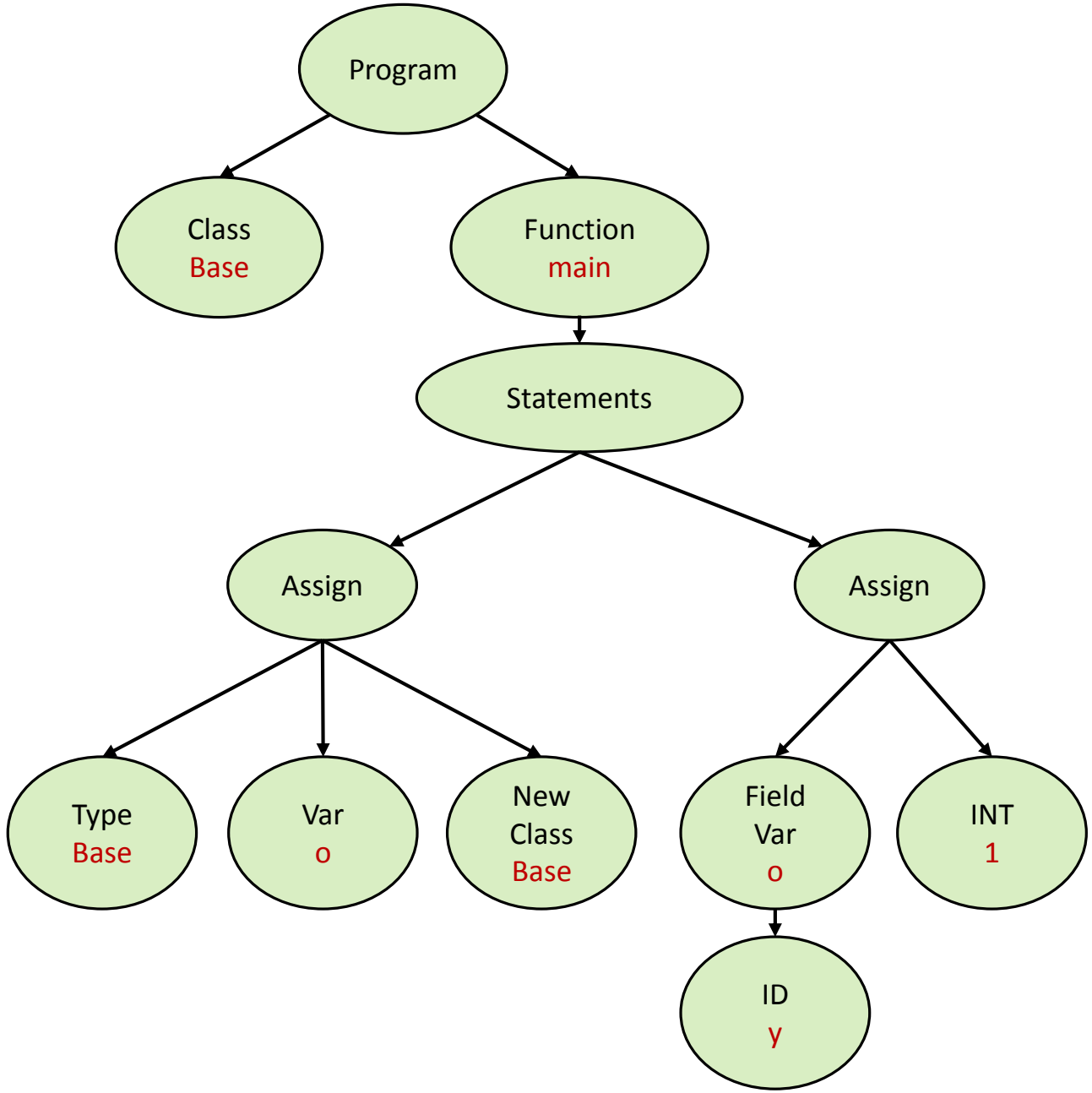
ID	Type	Kind
o	Base	variable

$scope_2$



Classes

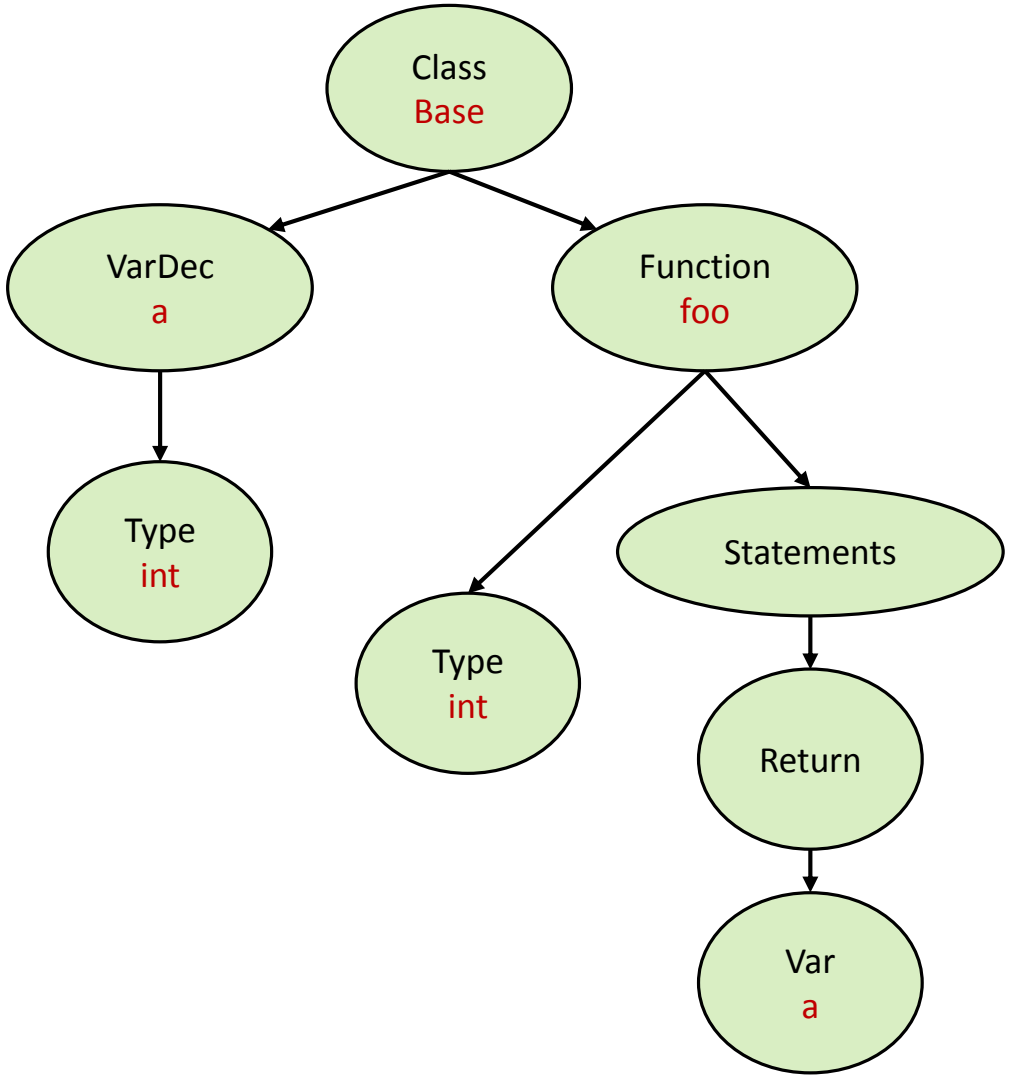
```
class Base {
  int x;
}
void main() {
  Base o = new Base;
  o.y = 1;
}
```



Invalid

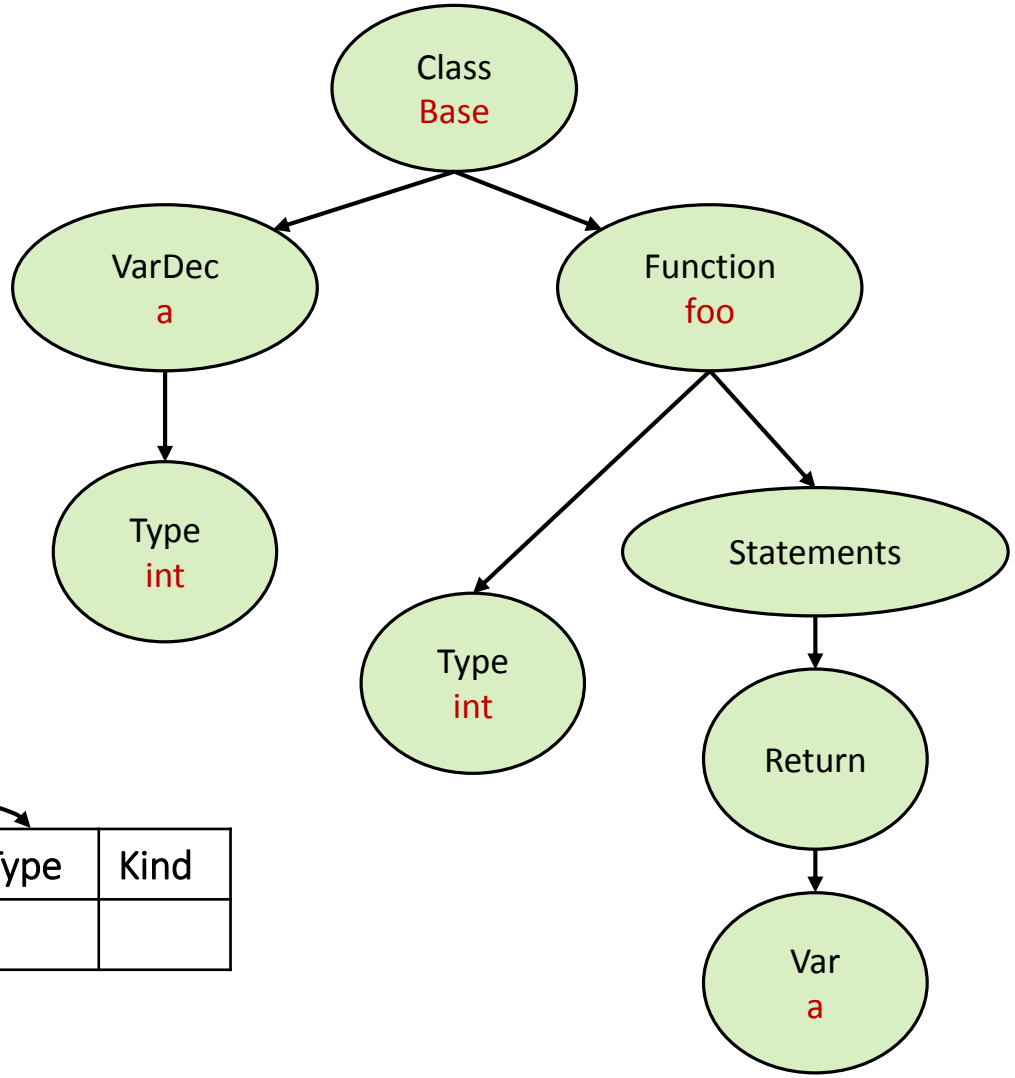
Classes

```
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

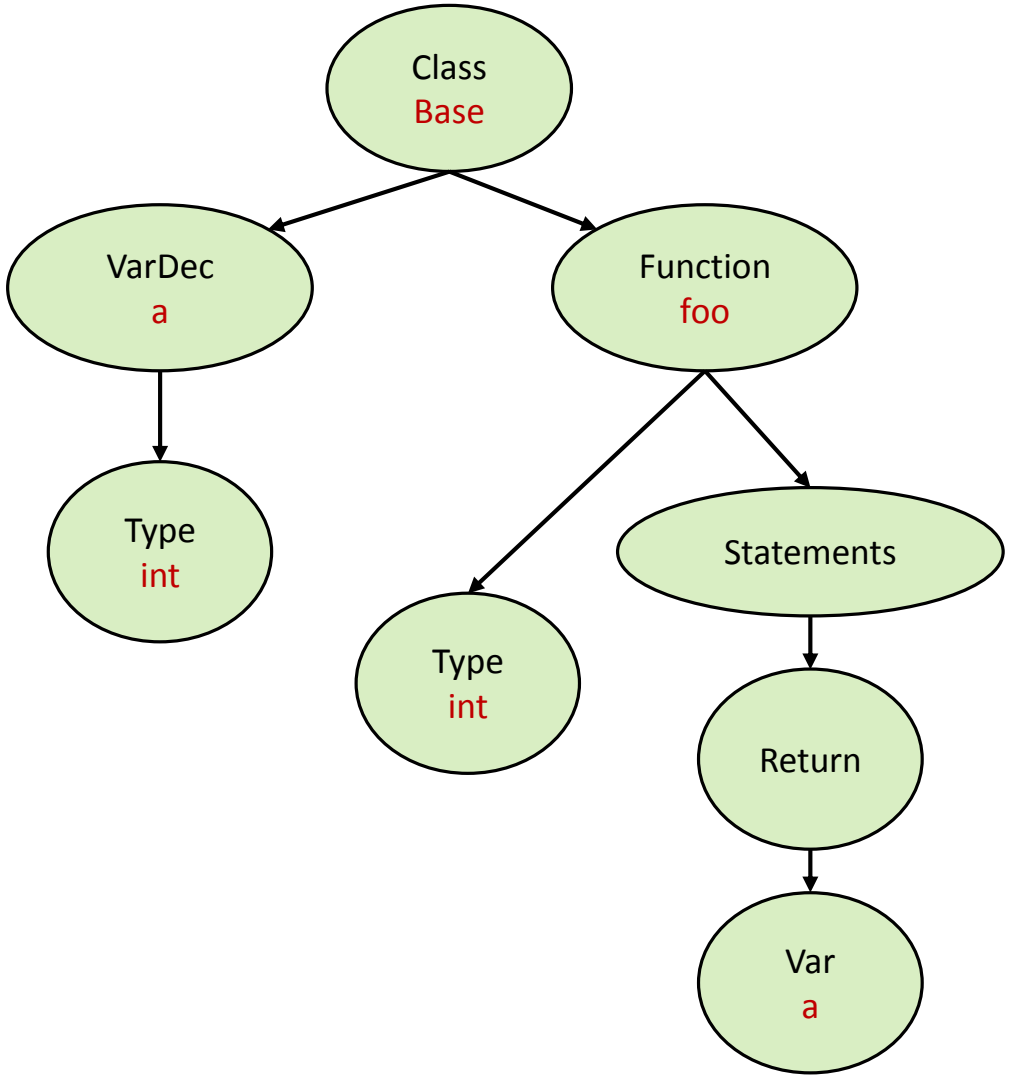
$scope_2$

ID	Type	Kind

$scope_3$

Classes

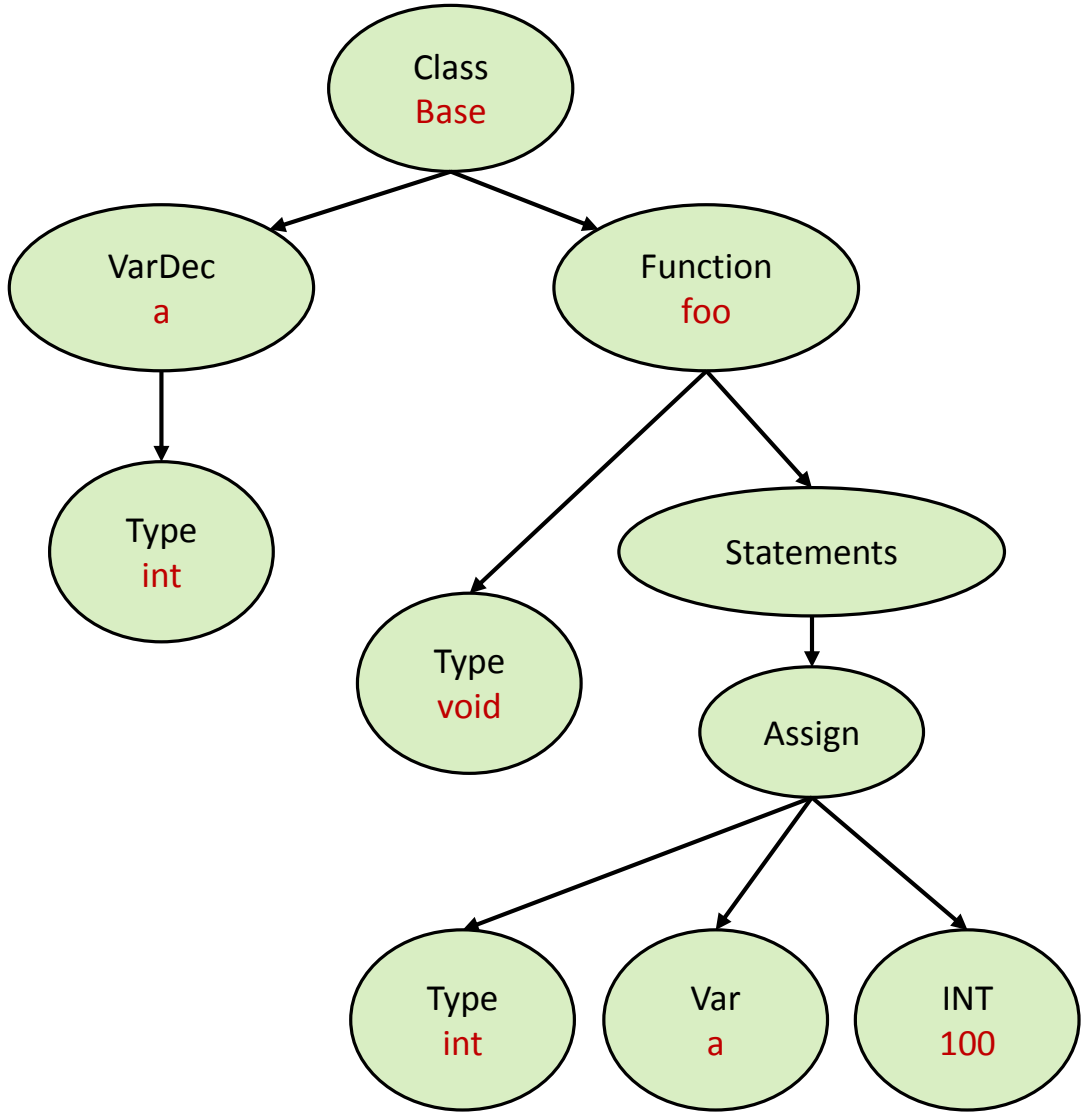
```
class Base {
  int a;
  int foo() {
    return a;
  }
}
```



Valid

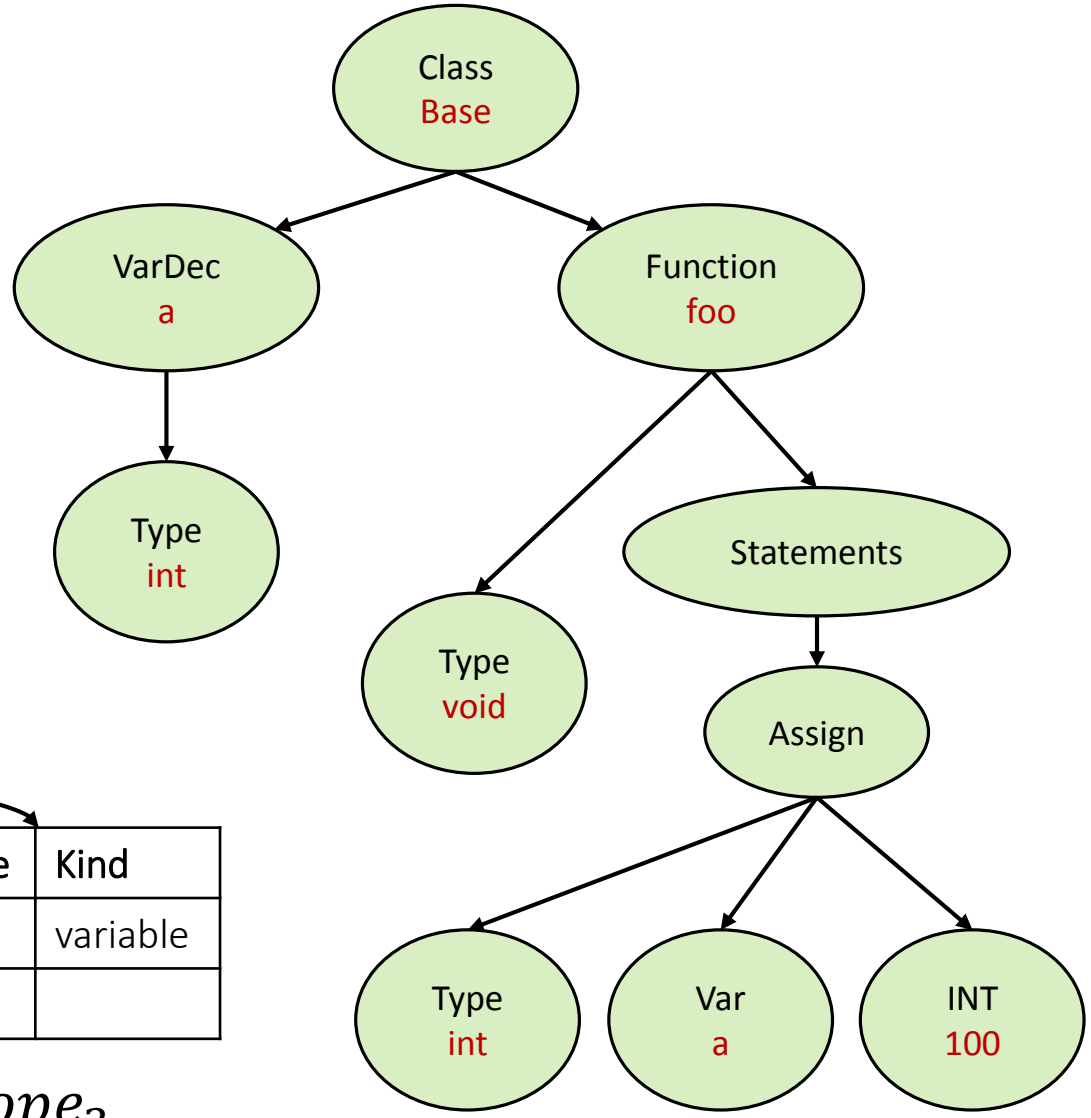
Classes

```
class Base {
  int a;
  void foo() {
    int a = 100;
  }
}
```



Classes

```
class Base {
  int a;
  void foo() {
    int a = 100;
  }
}
```



ID	Type	Kind
Base	…	class

$scope_1$

ID	Type	Kind
a	int	variable
foo	…	function

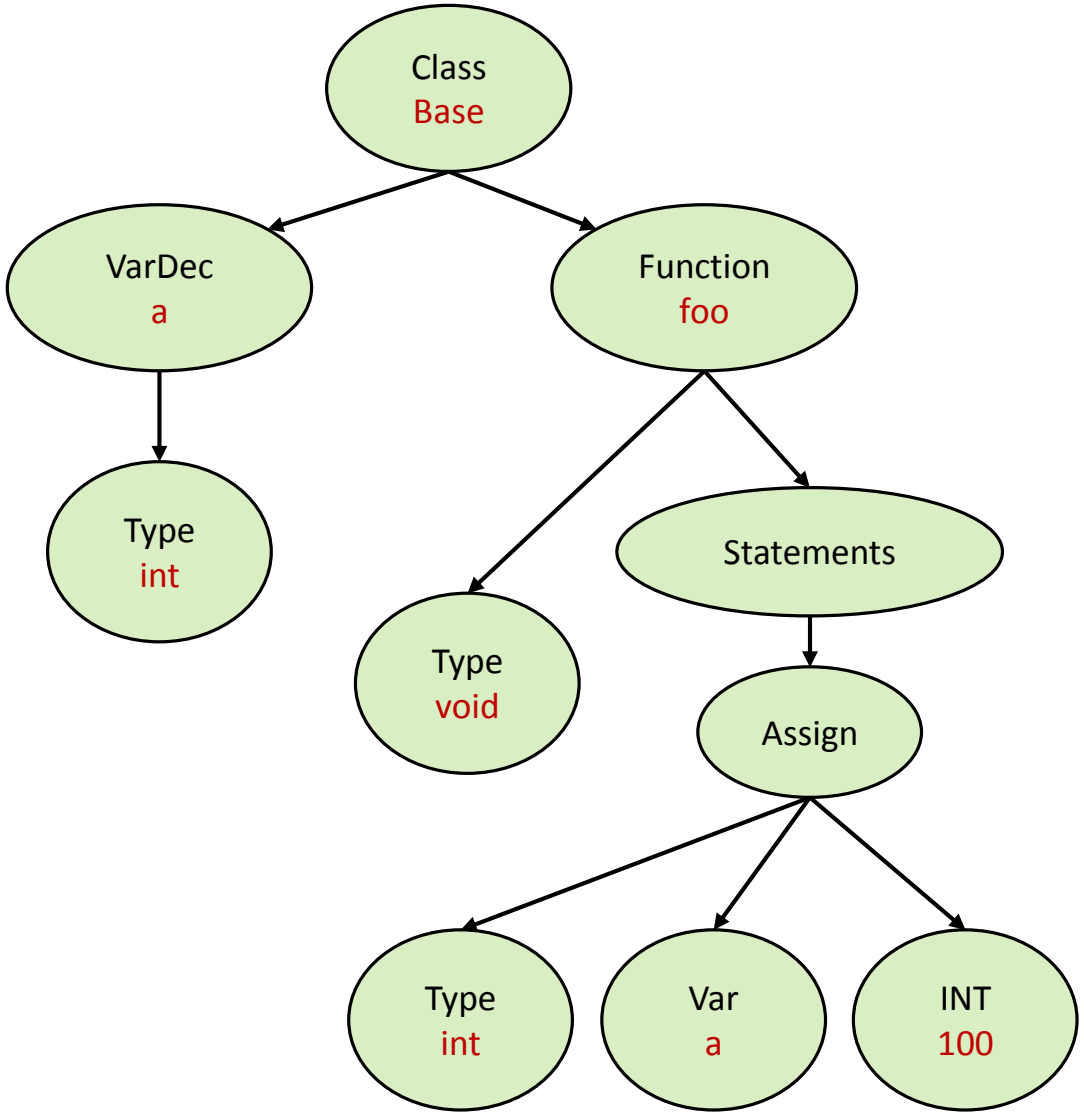
$scope_2$

ID	Type	Kind
a	int	variable

$scope_3$

Classes

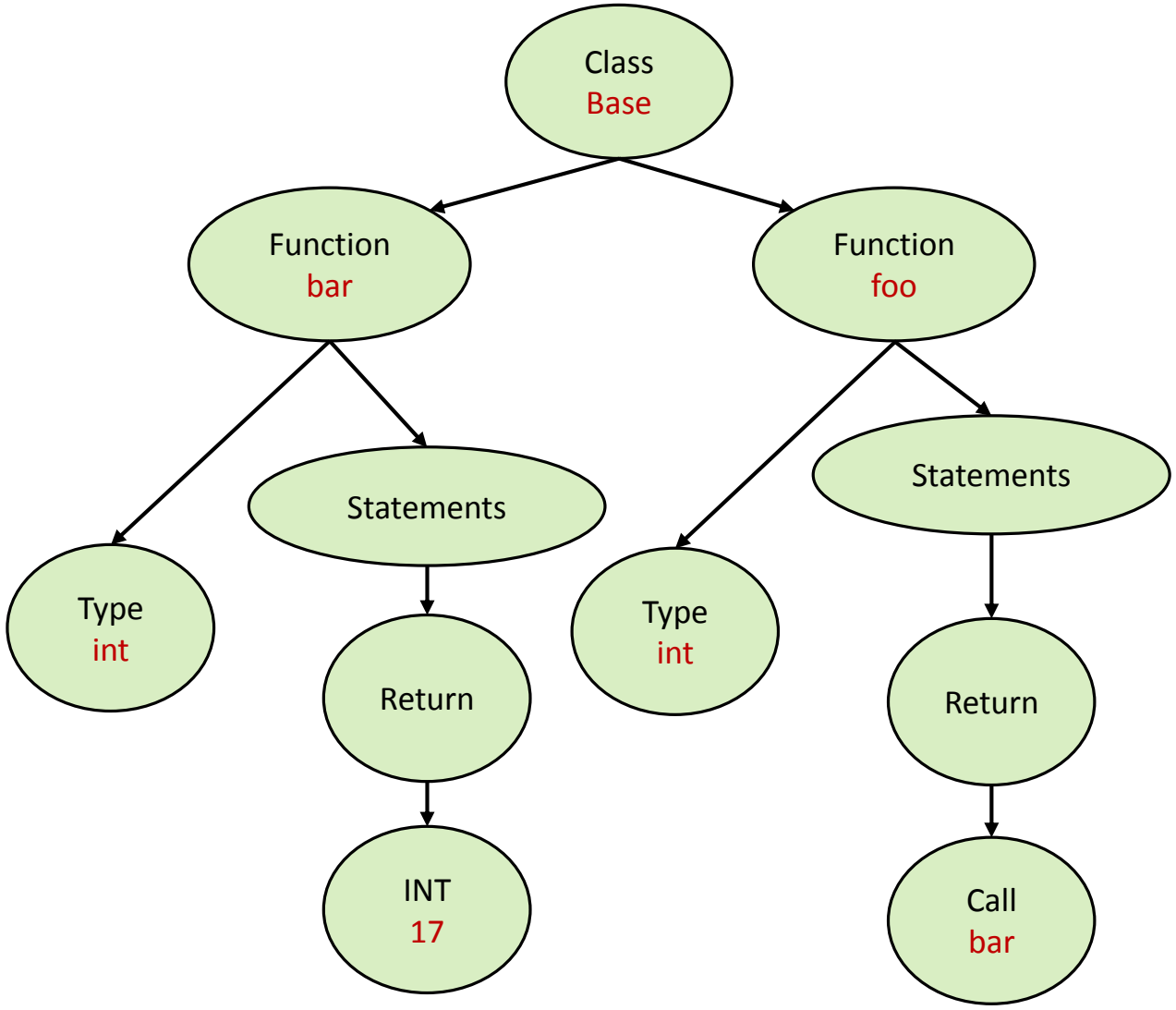
```
class Base {
   int a;
   void foo() {
      int a = 100;
   }
}
```



Valid

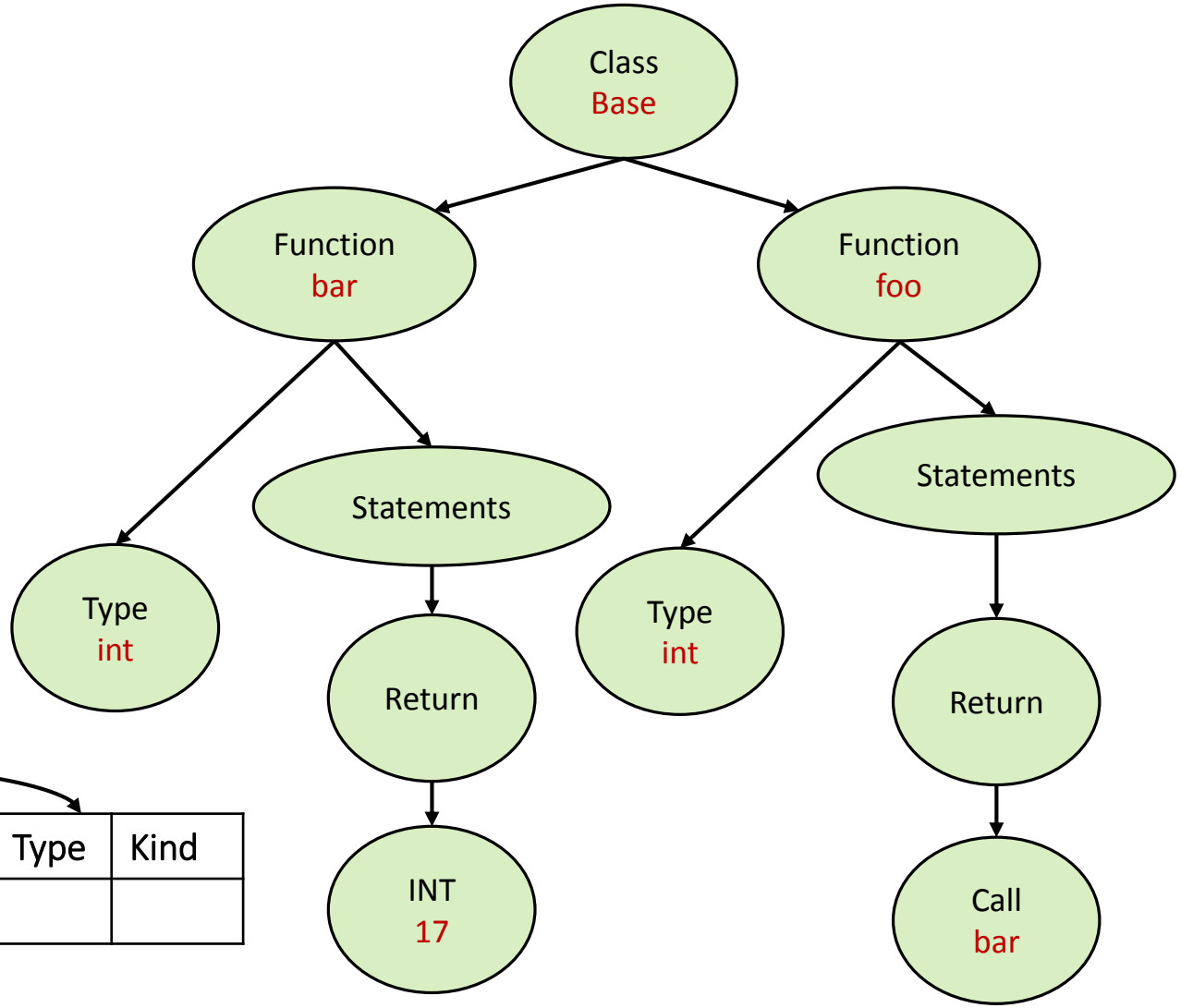
Classes

```
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

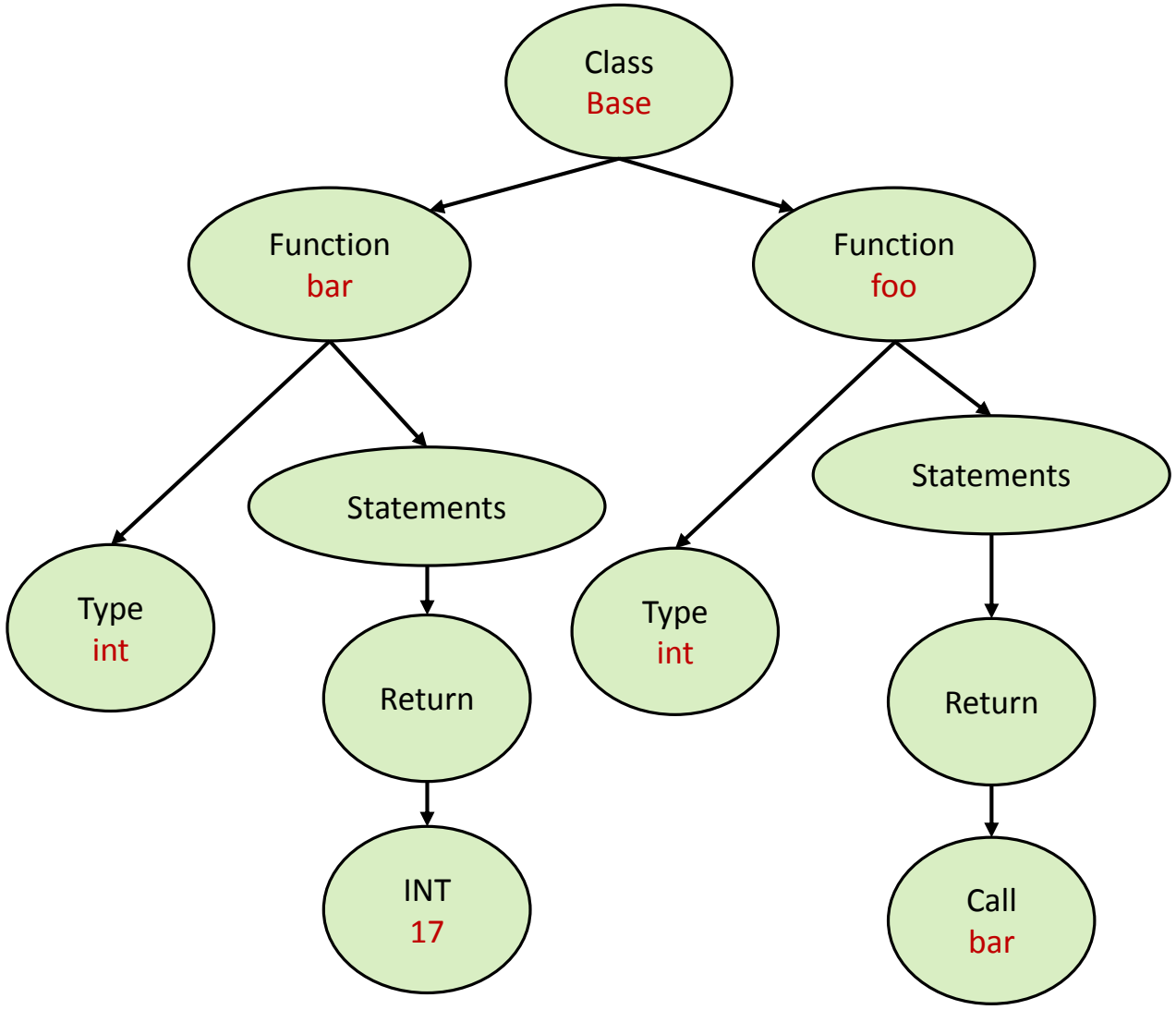
$scope_2$

ID	Type	Kind

$scope_3$

Classes

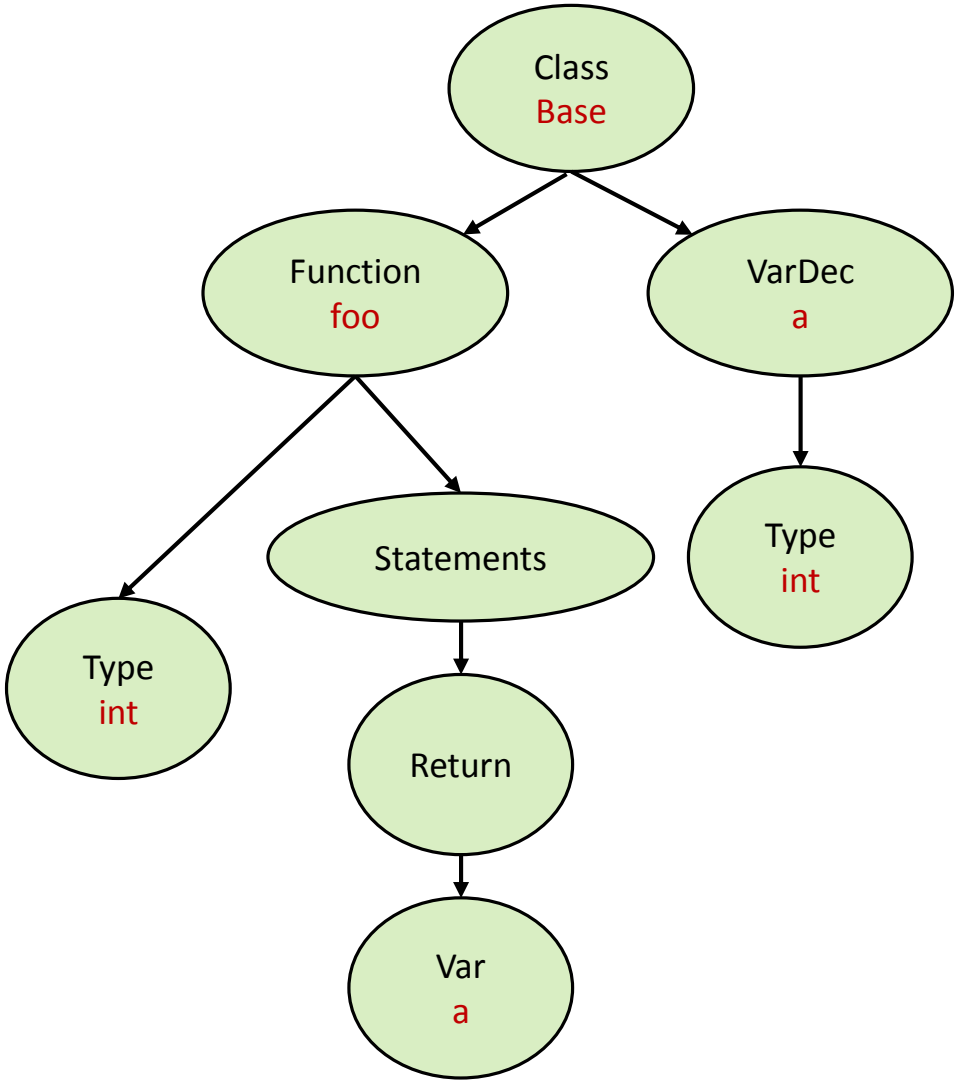
```
class Base {
  int bar() {
    return 17;
  }
  int foo() {
    return bar();
  }
}
```



Valid

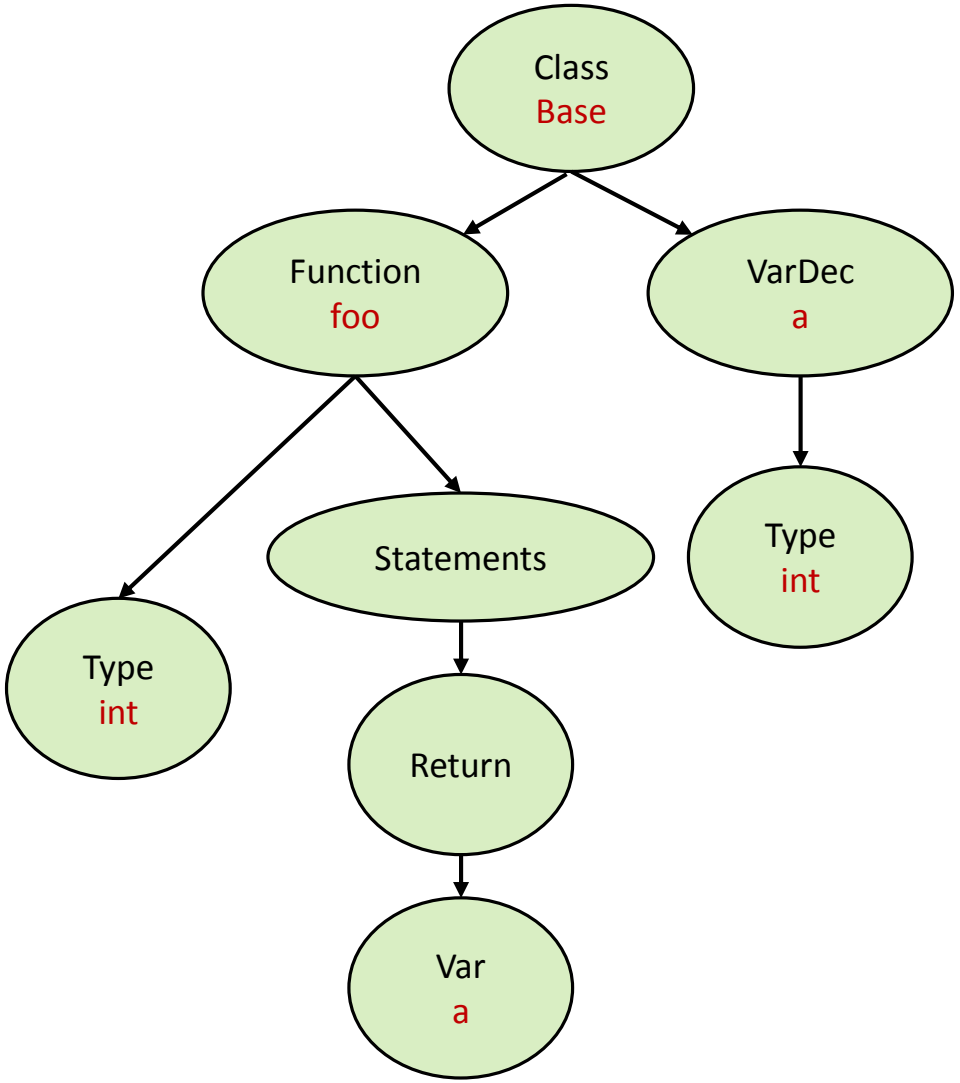
Classes

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



Classes

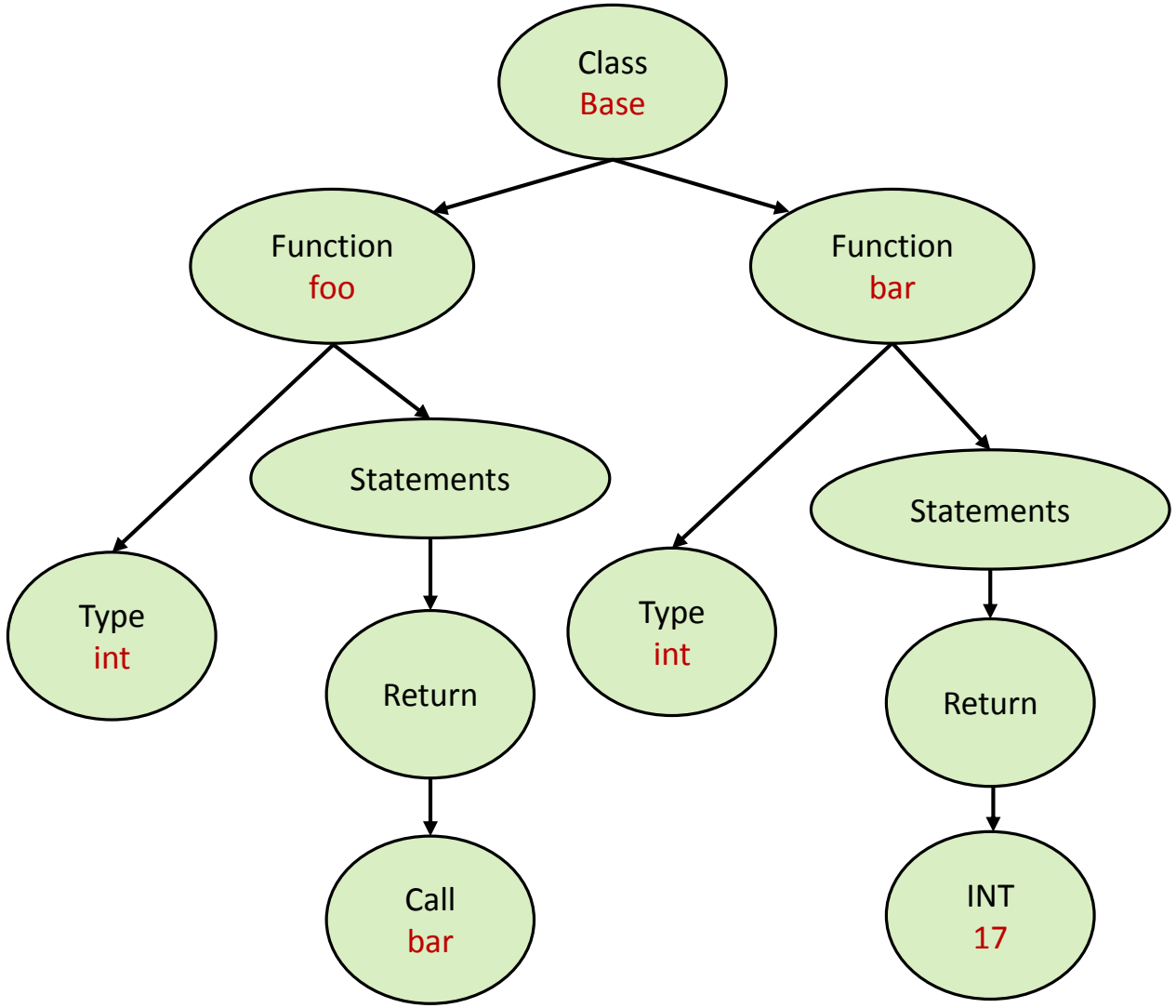
```
class Base {
  int foo() {
    return a;
  }
  int a;
}
```



Invalid

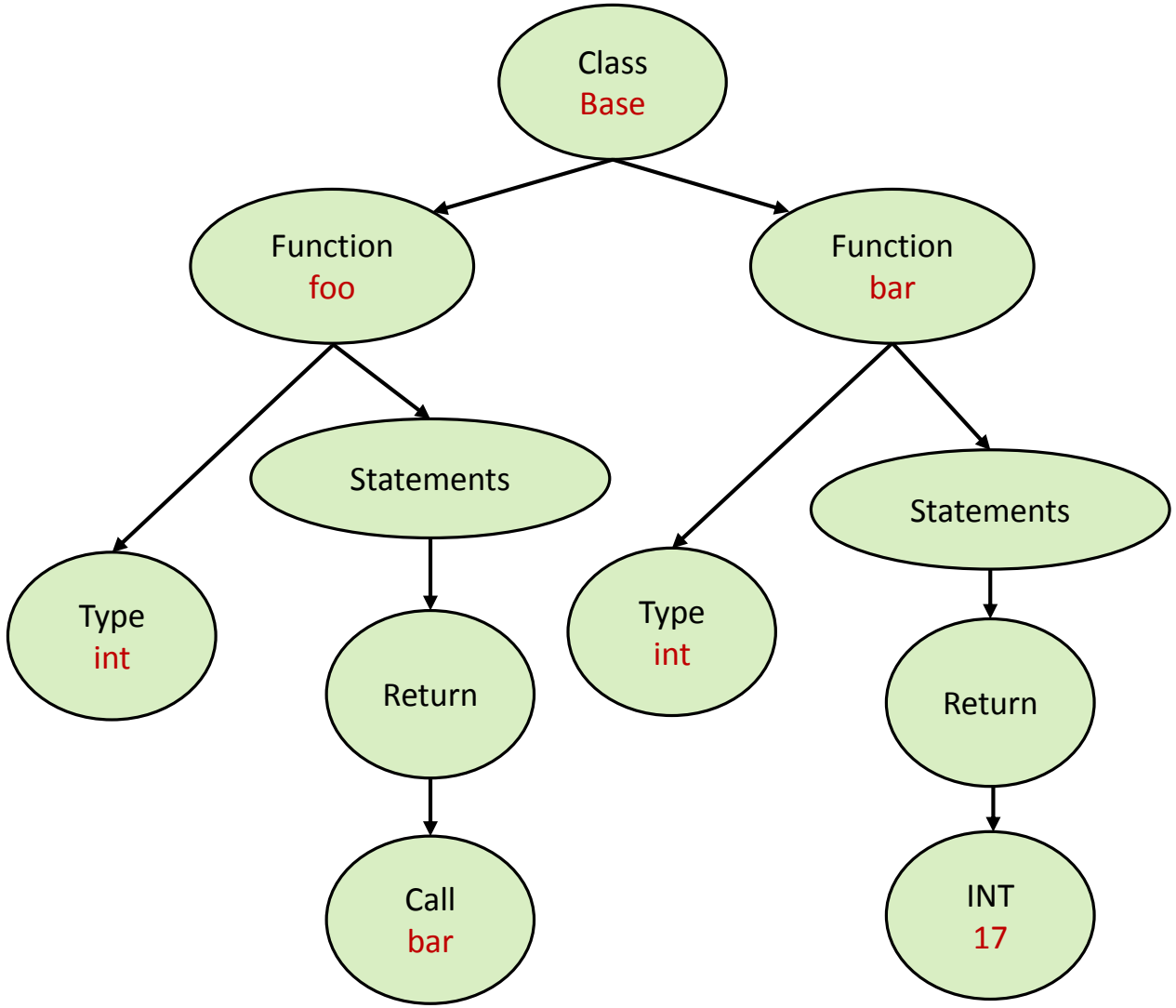
Classes

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



Classes

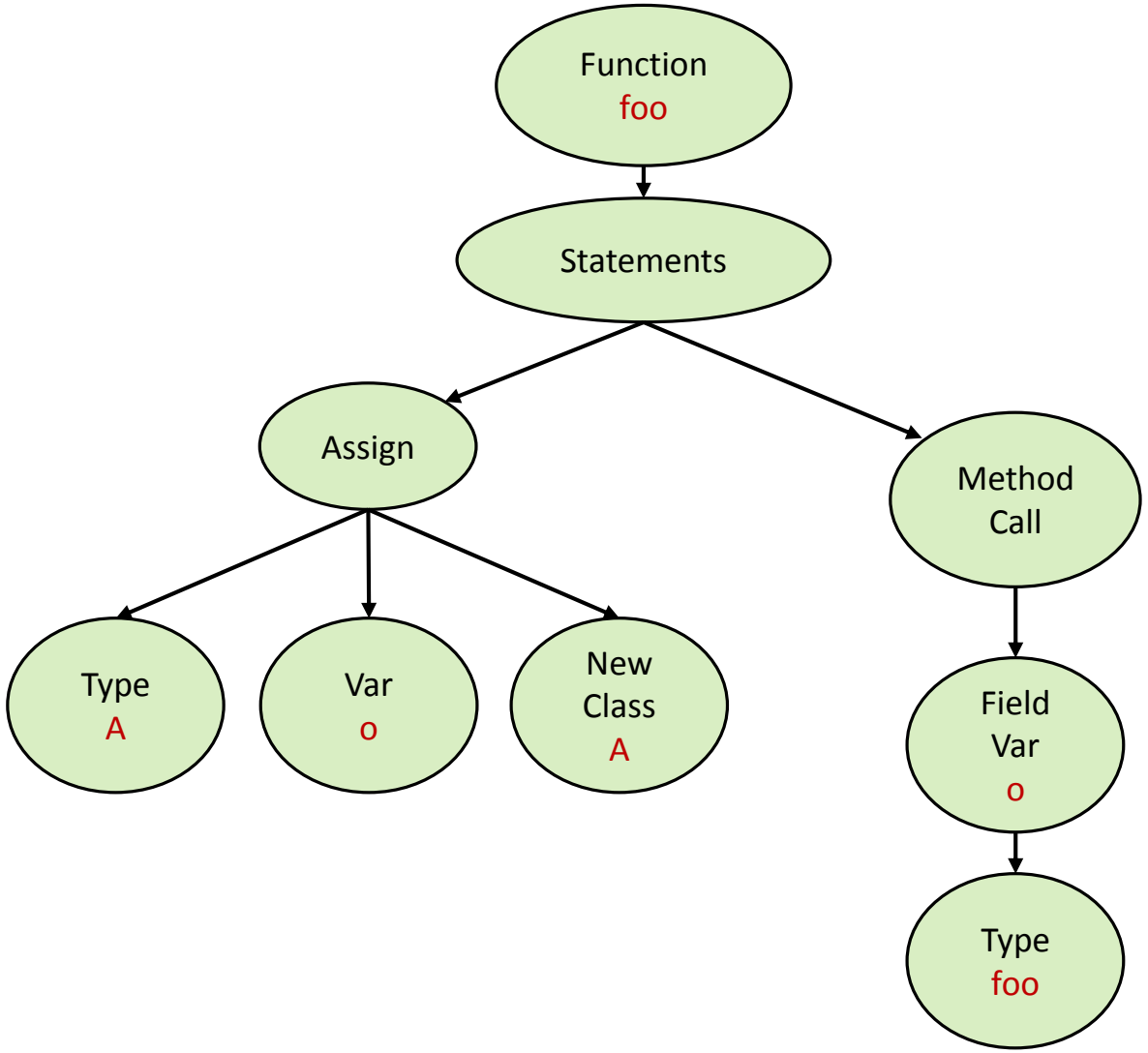
```
class Base {
  int foo() {
    return bar();
  }
  int bar() {
    return 17;
  }
}
```



Invalid

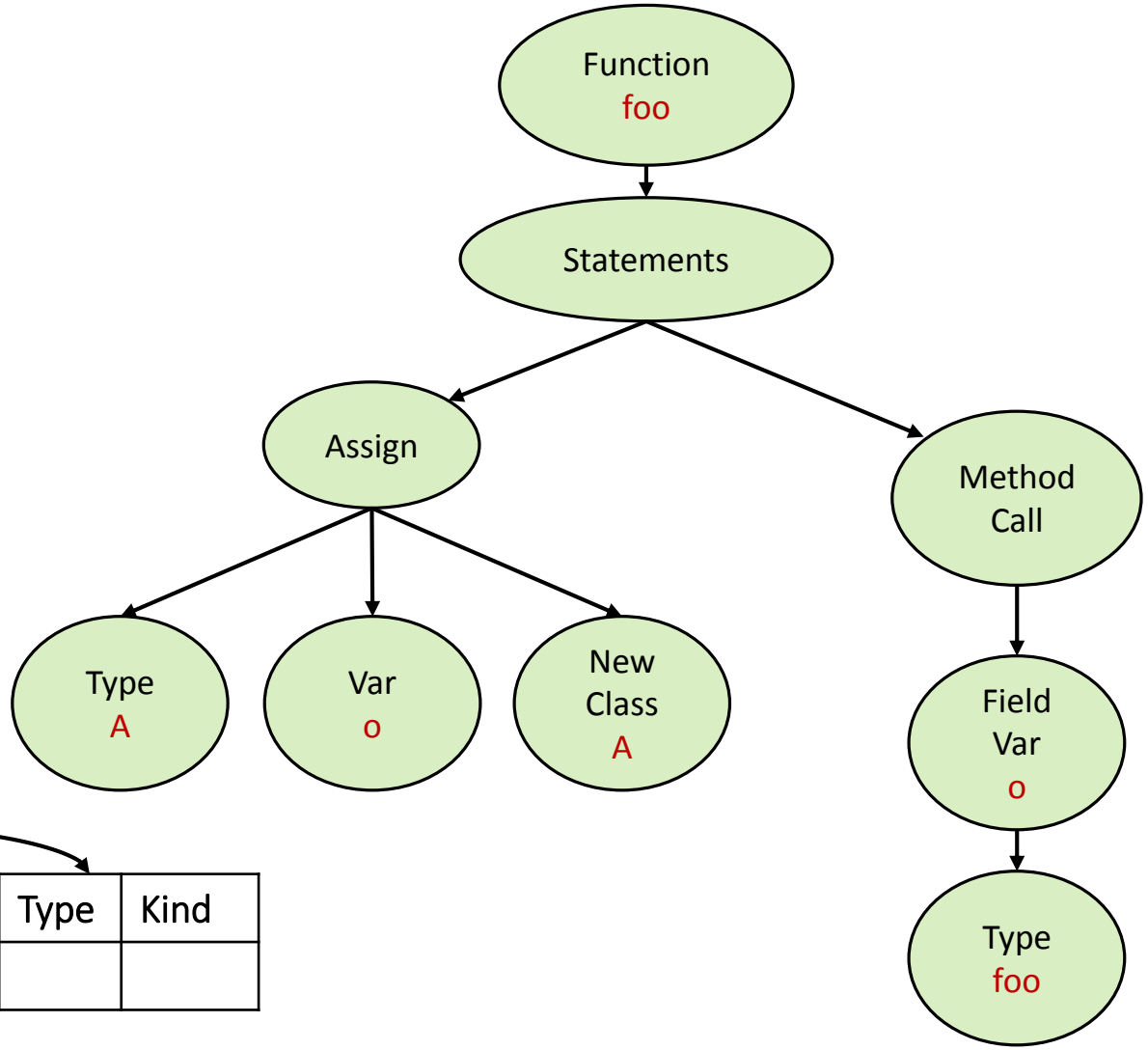
Classes

```
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

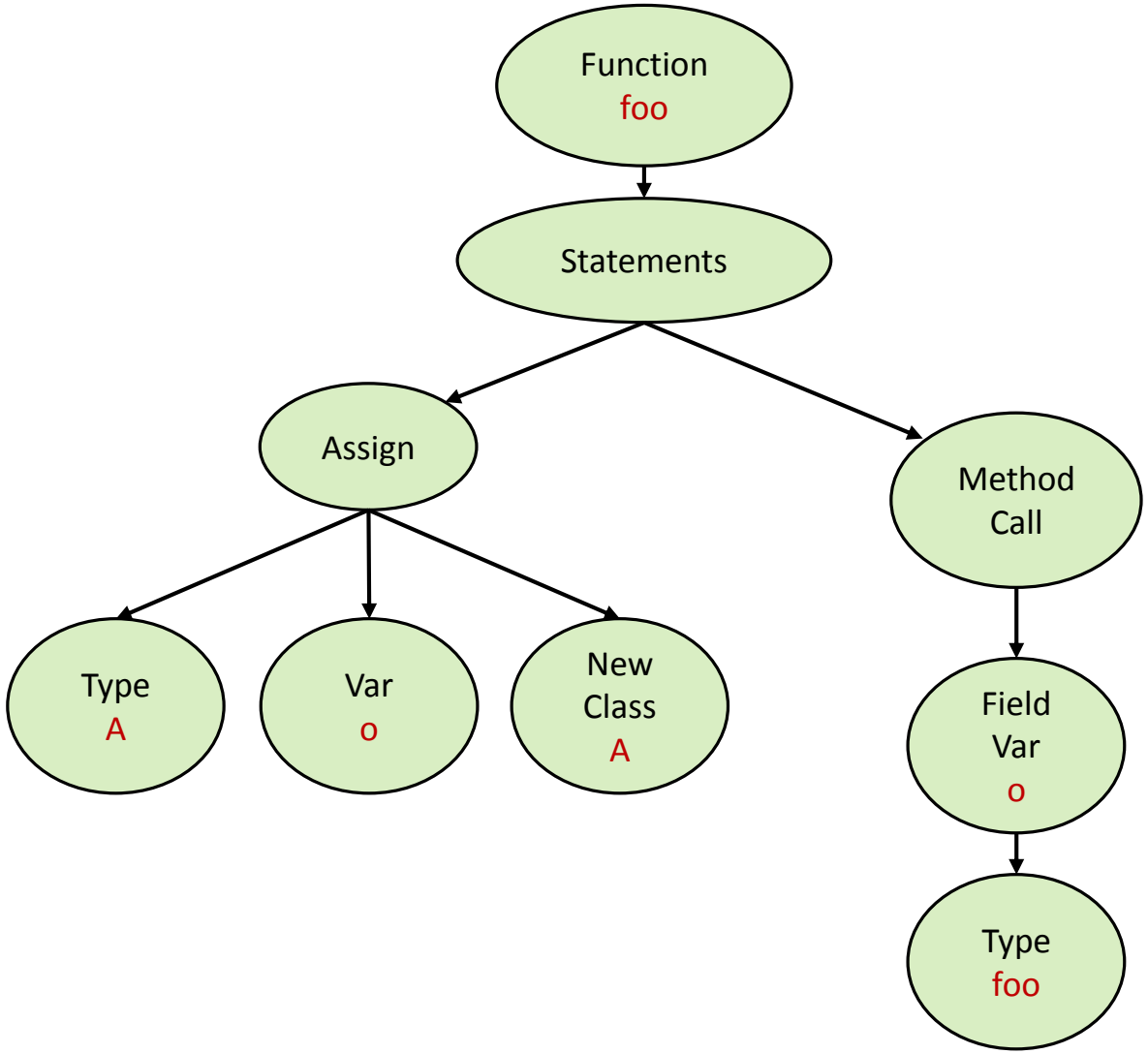
$scope_2$

ID	Type	Kind

$scope_3$

Classes

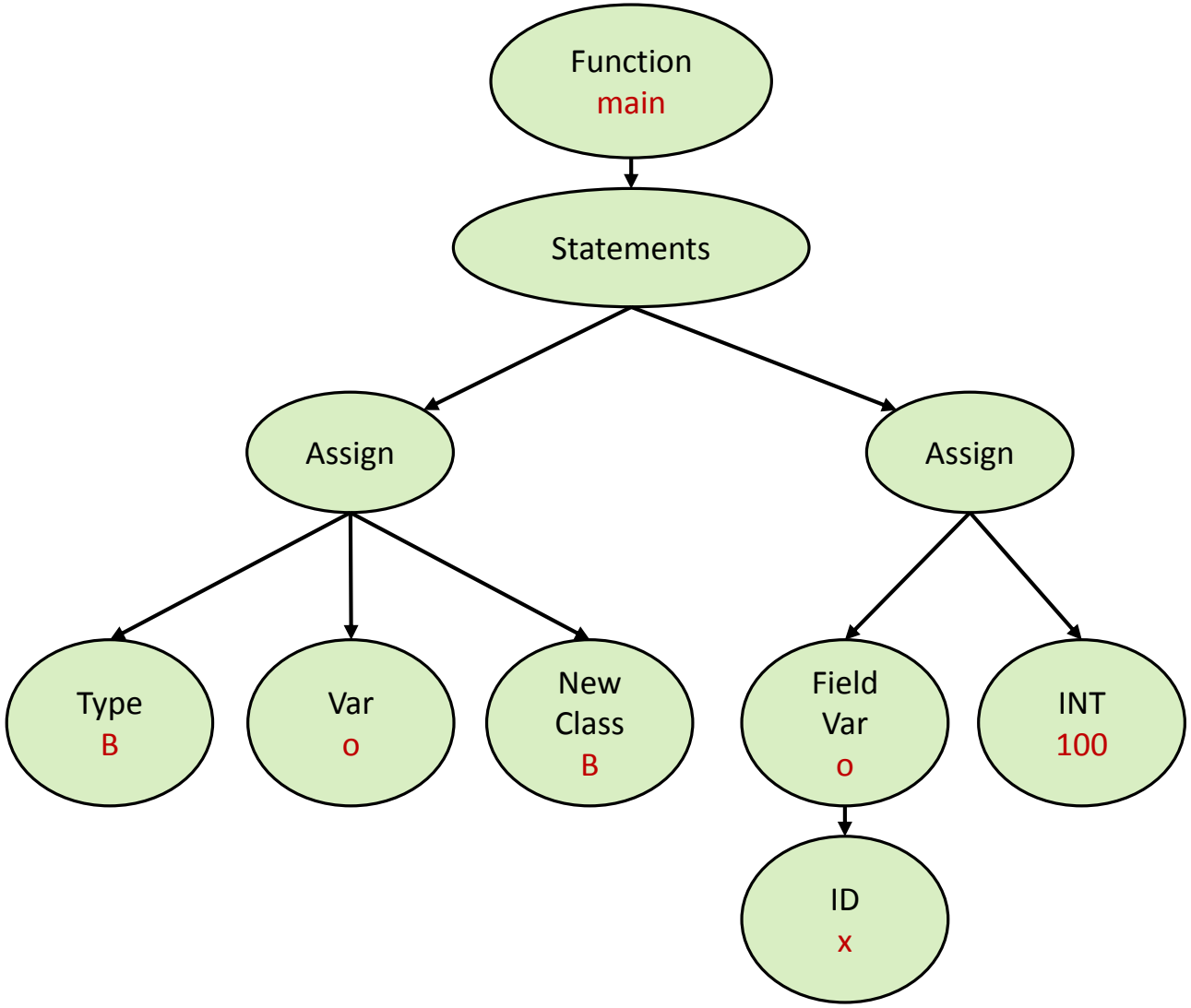
```
class A {
  void foo() {
    A o = new A;
    o.foo();
  }
}
```



Valid

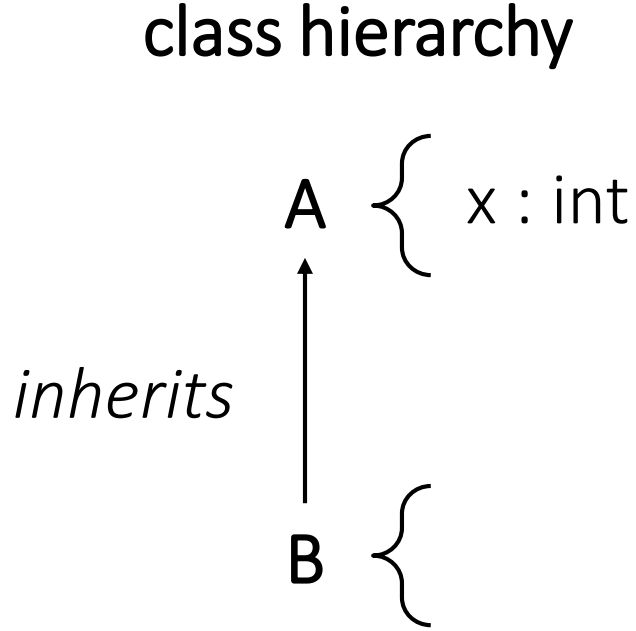
Inheritance

```
class A {
  int x;
}
class B extends A { }
void main() {
  B o = new B;
  o.x = 100;
}
```



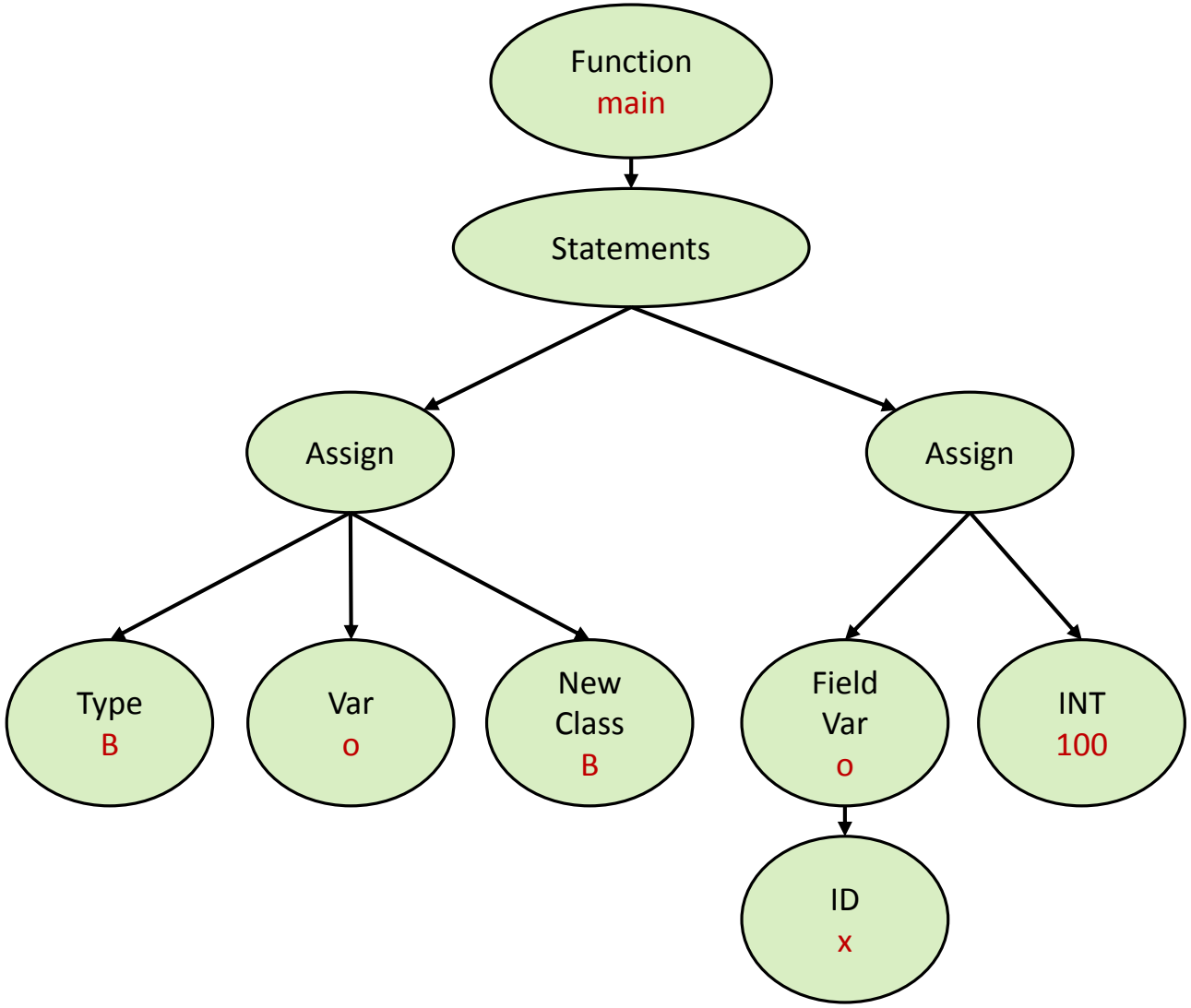
Inheritance

```
class A {
  int x;
}
class B extends A { }
void main() {
  B o = new B;
  o.x = 100;
}
```



Inheritance

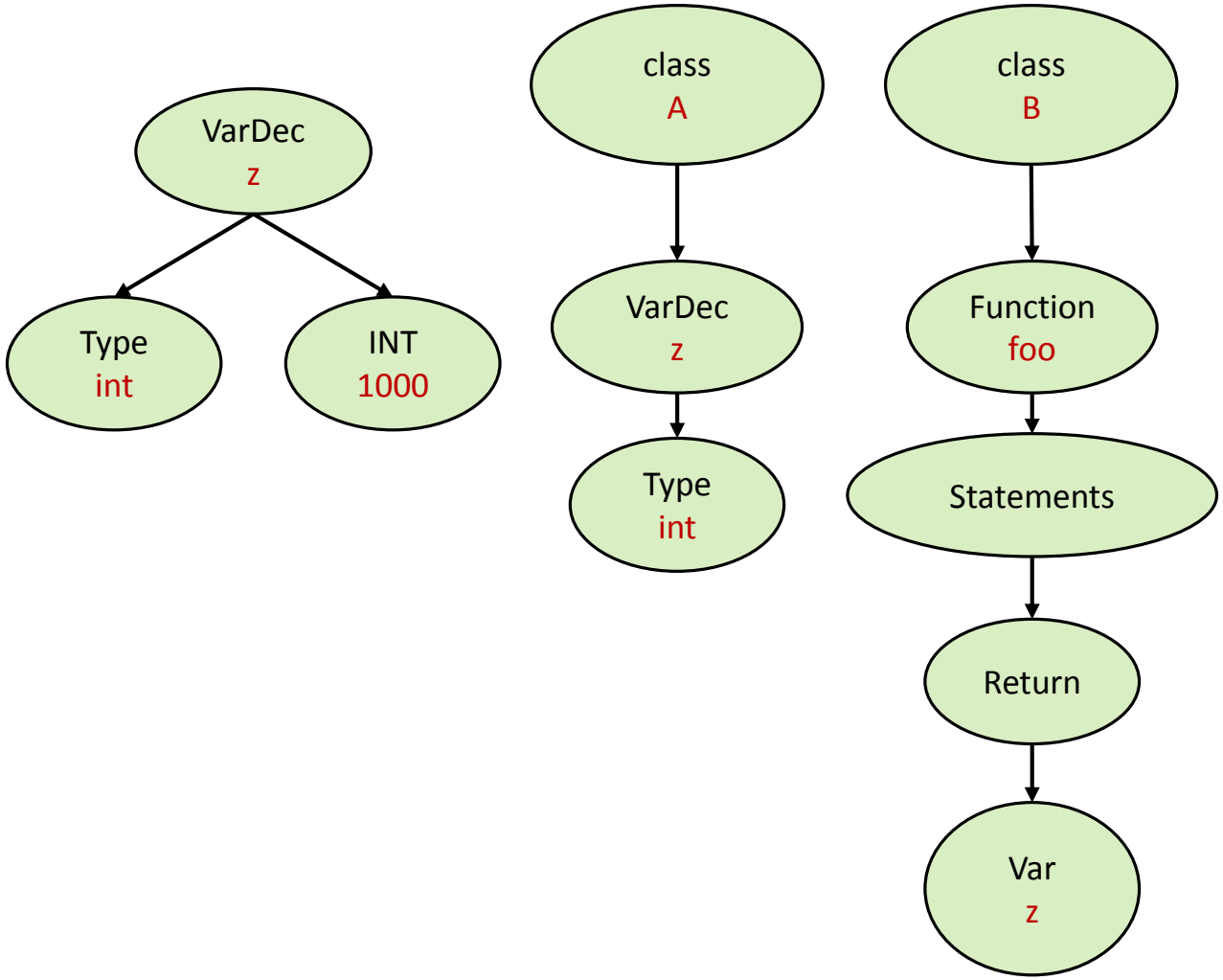
```
class A {
  int x;
}
class B extends A { }
void main() {
  B o = new B;
  o.x = 100;
}
```



Valid

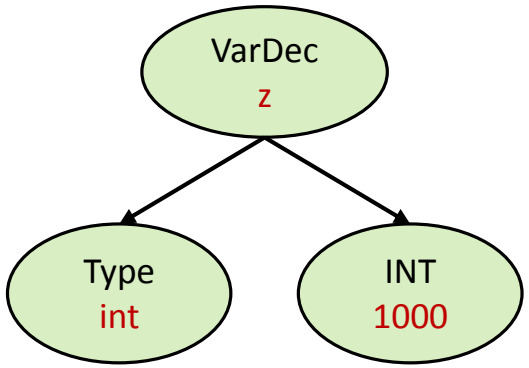
Inheritance

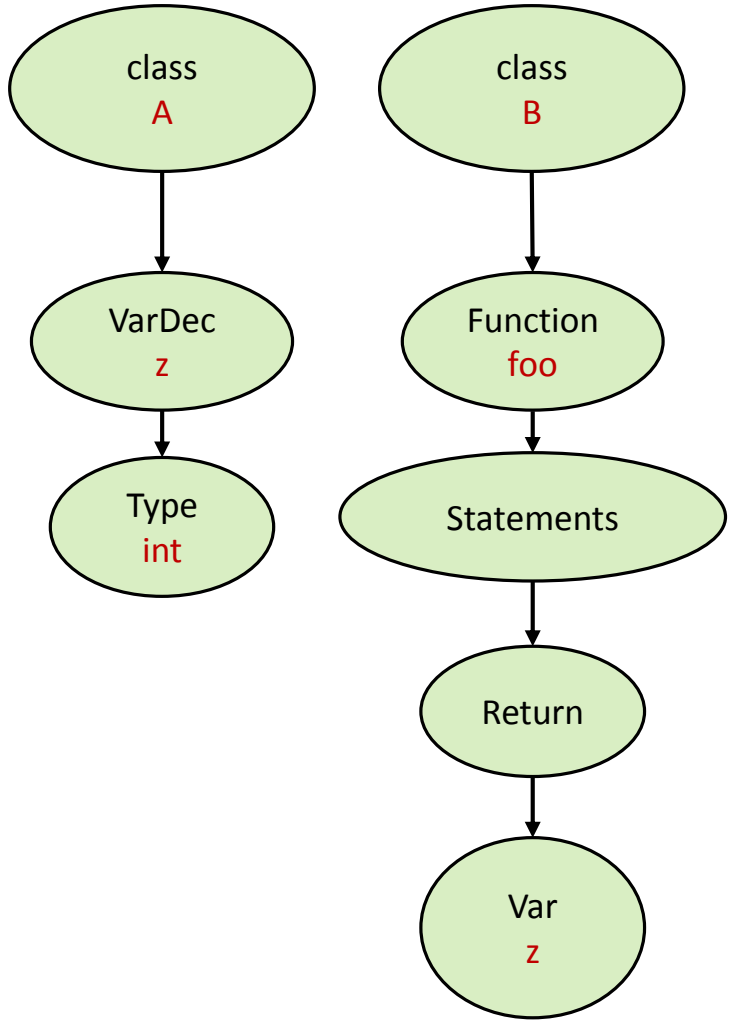
```
class A {
  int z;
}
class B extends A {
  int foo() {
    return z;
  }
}
```



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

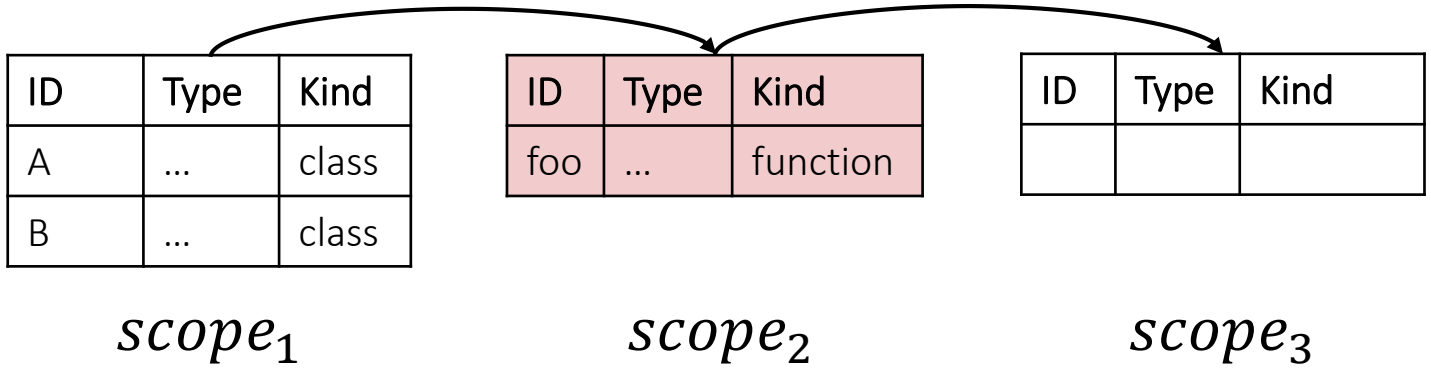
$scope_2$

ID	Type	Kind

$scope_3$

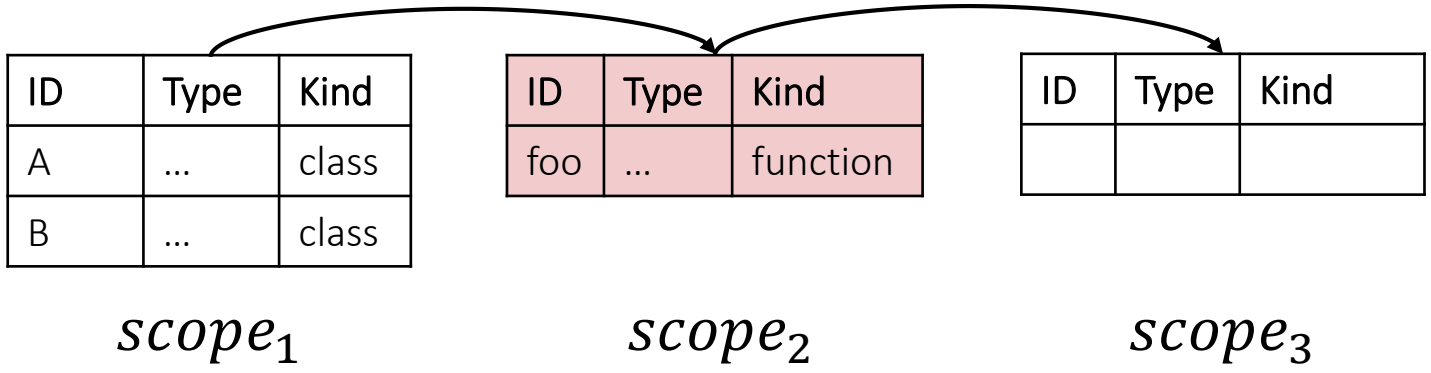
Inheritance

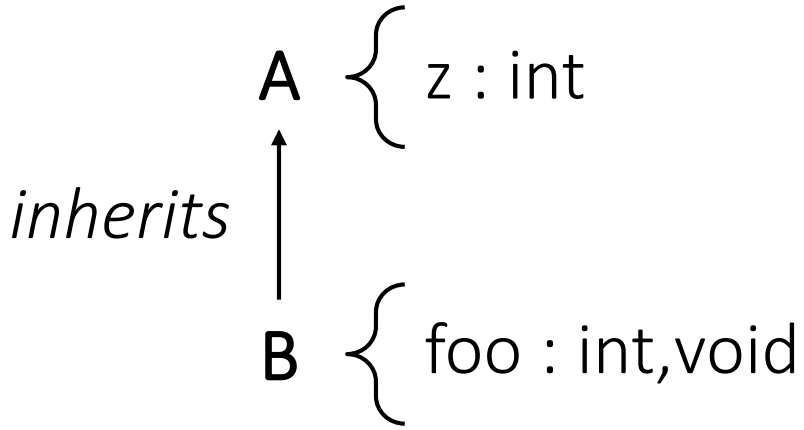
```
class A {
  int z;
}
class B extends A {
  int foo() {
    return z;
  }
}
```



Inheritance

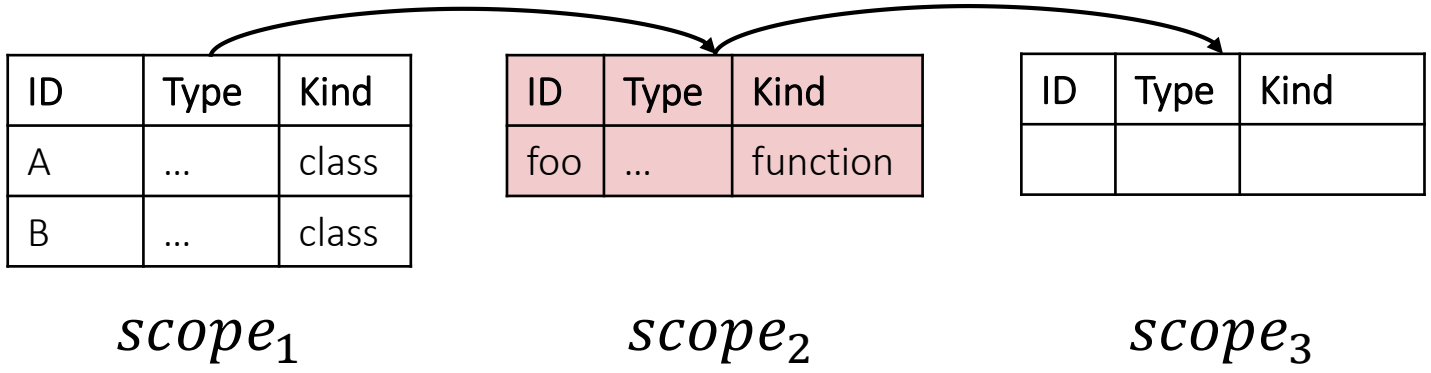
```
class A {
  int z;
}
class B extends A {
  int foo() {
    return z;
  }
}
```

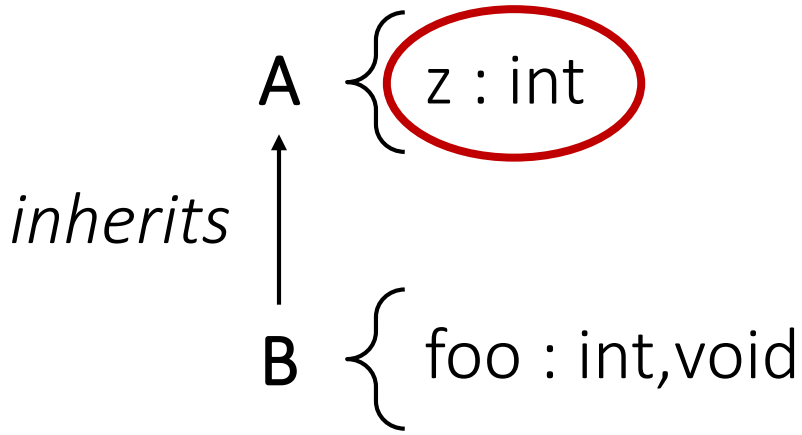




Inheritance

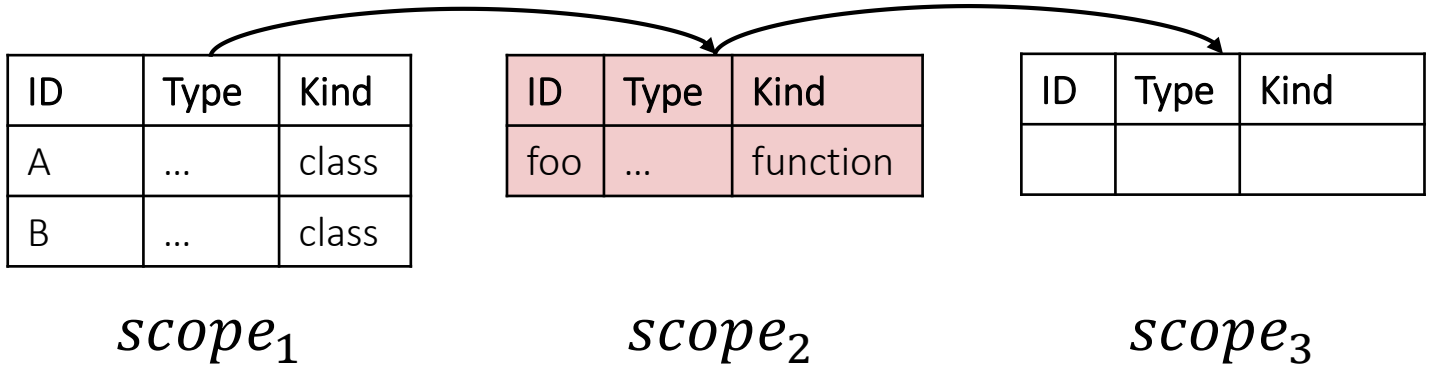
```
class A {
  int z;
}
class B extends A {
  int foo() {
    return z;
  }
}
```

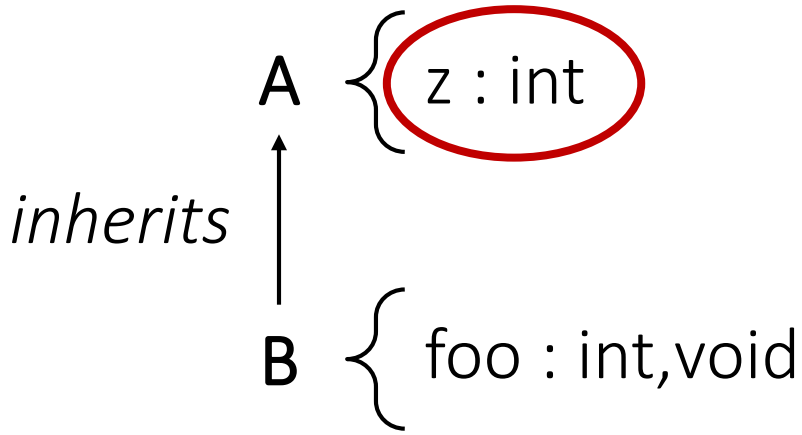




Inheritance

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

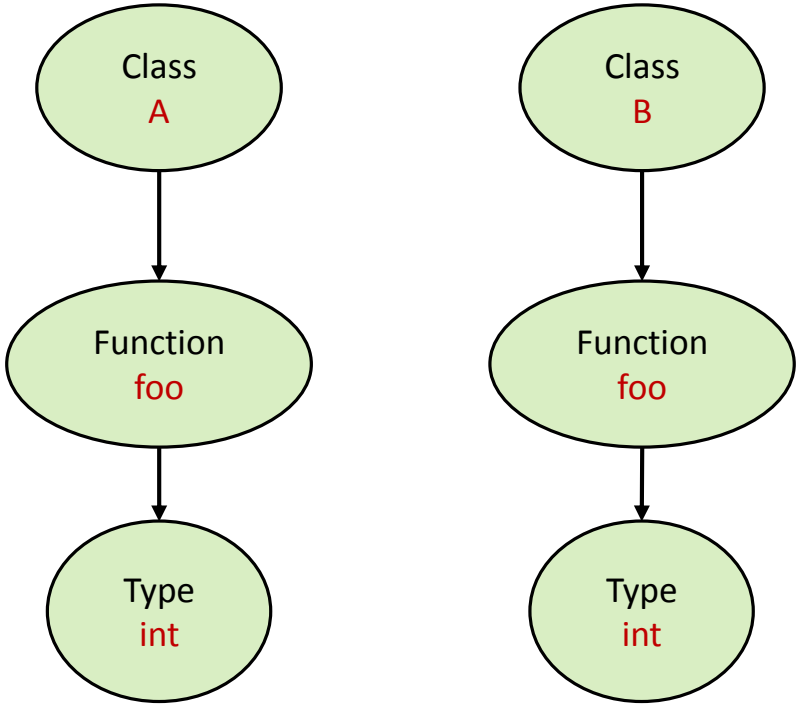




Valid

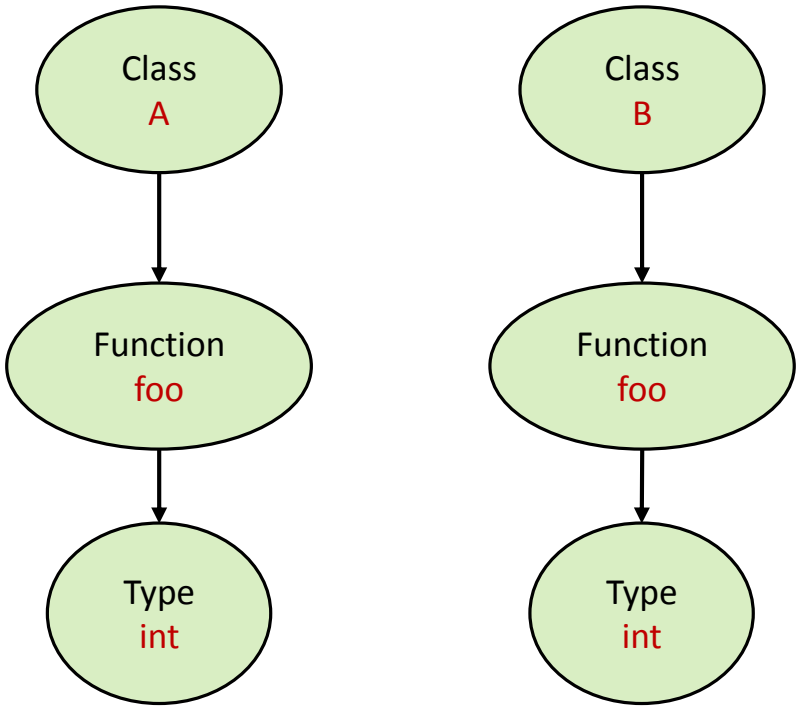
Inheritance

```
class A {
  int foo() {
    return 17;
  }
}
class B extends A {
  int foo() {
    return 18;
  }
}
```



Inheritance

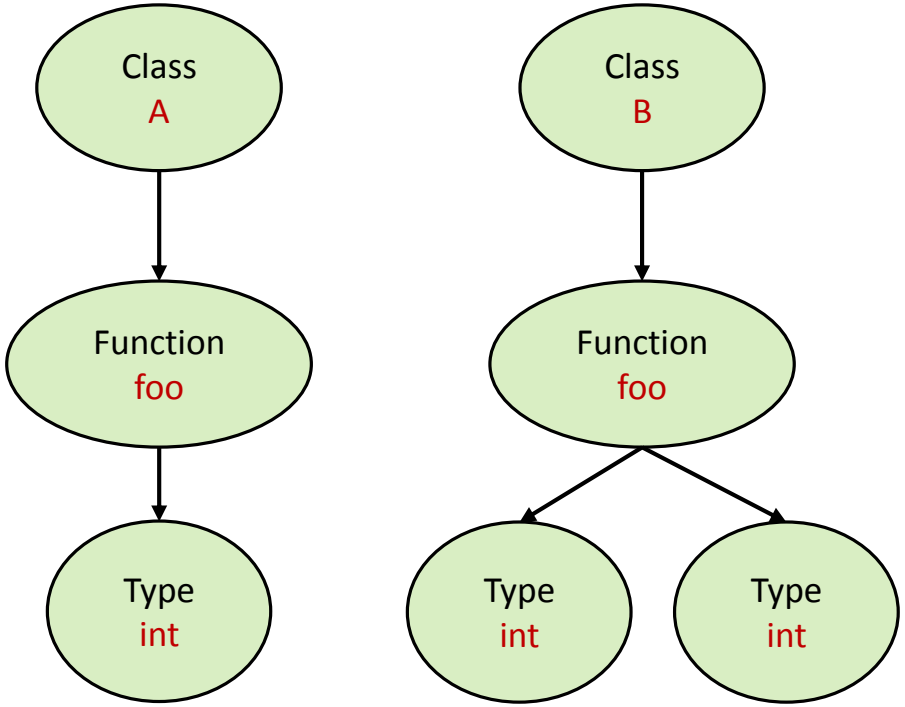
```
class A {
  int foo() {
    return 17;
  }
}
class B extends A {
  int foo() {
    return 18;
  }
}
```



Valid

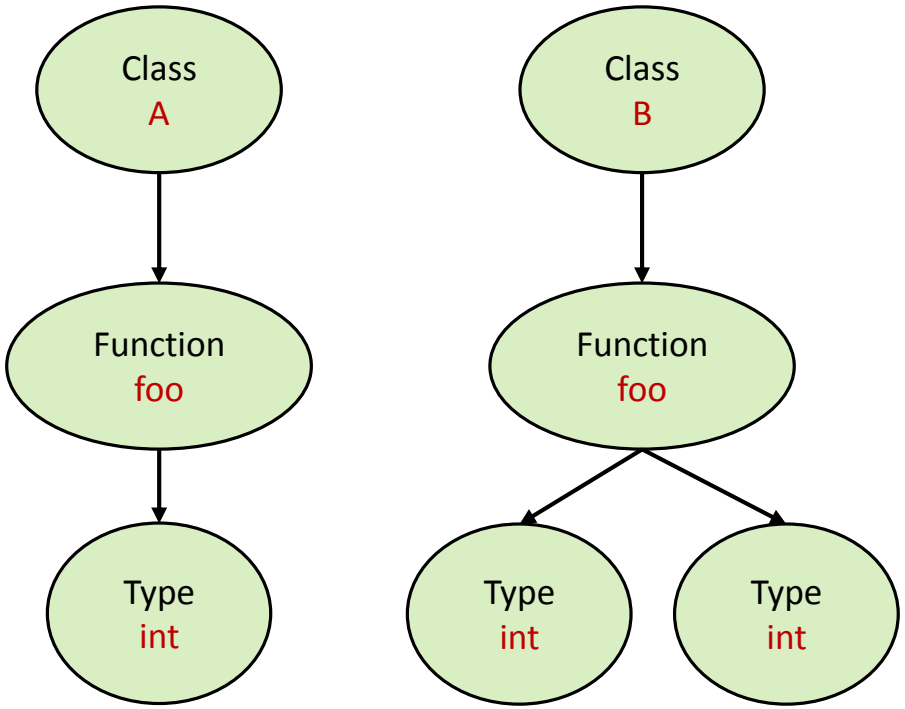
Inheritance

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



Inheritance

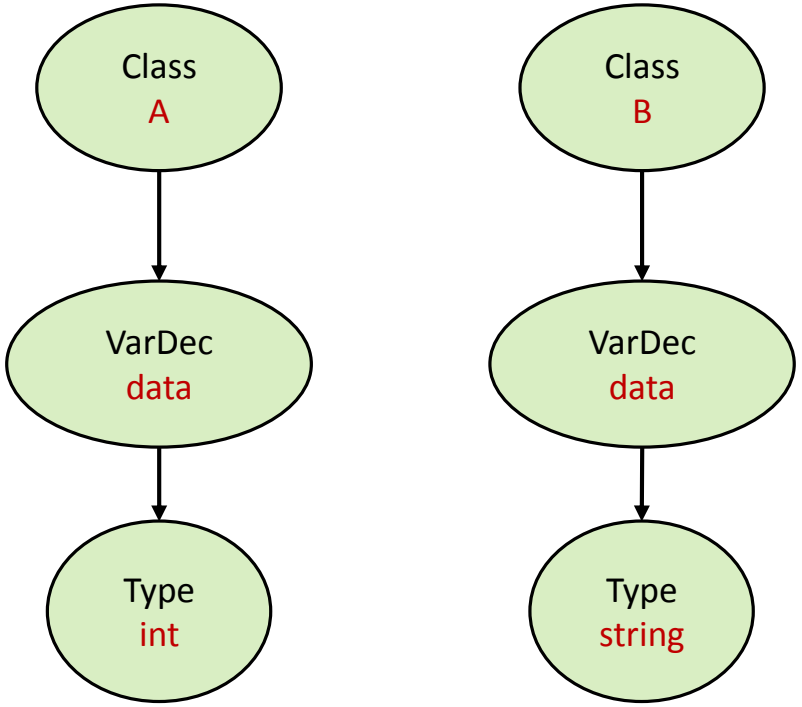
```
class A {
  int foo() {
    return 17;
  }
}
class B extends A {
  int foo(int x) {
    return x + 1;
  }
}
```



Invalid

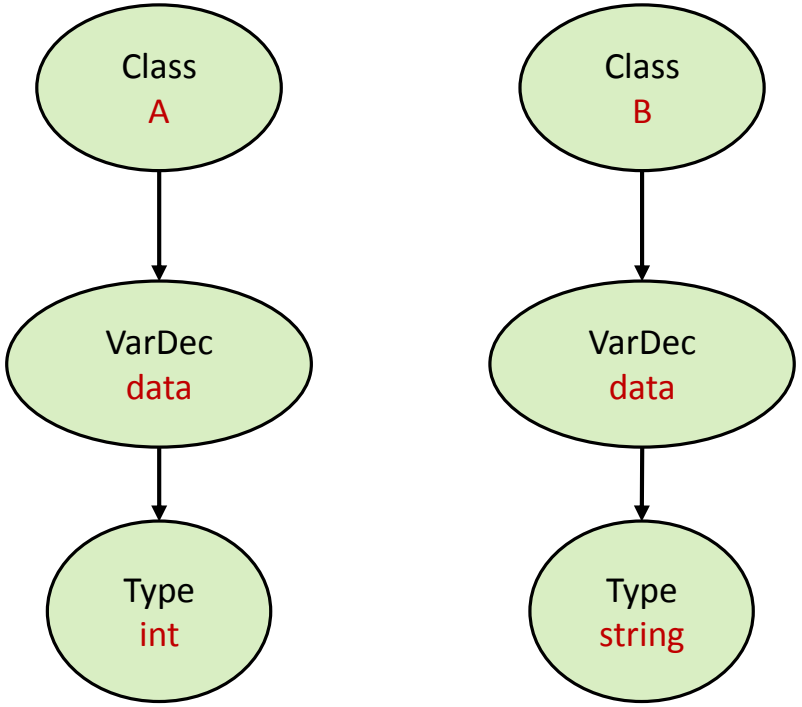
Inheritance

```
class A {
   int data;
}
class B extends A {
   string data;
}
```



Inheritance

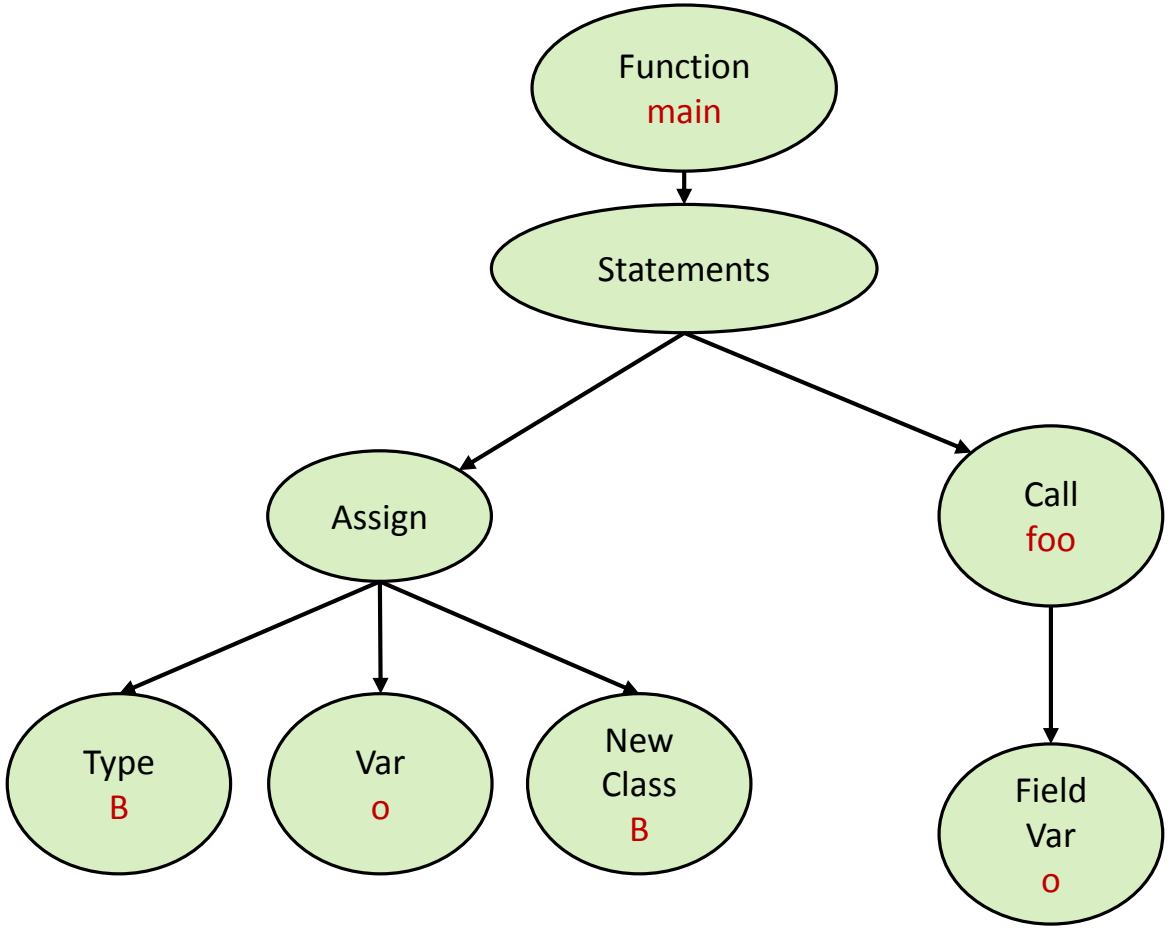
```
class A {
   int data;
}
class B extends A {
   string data;
}
```



Invalid

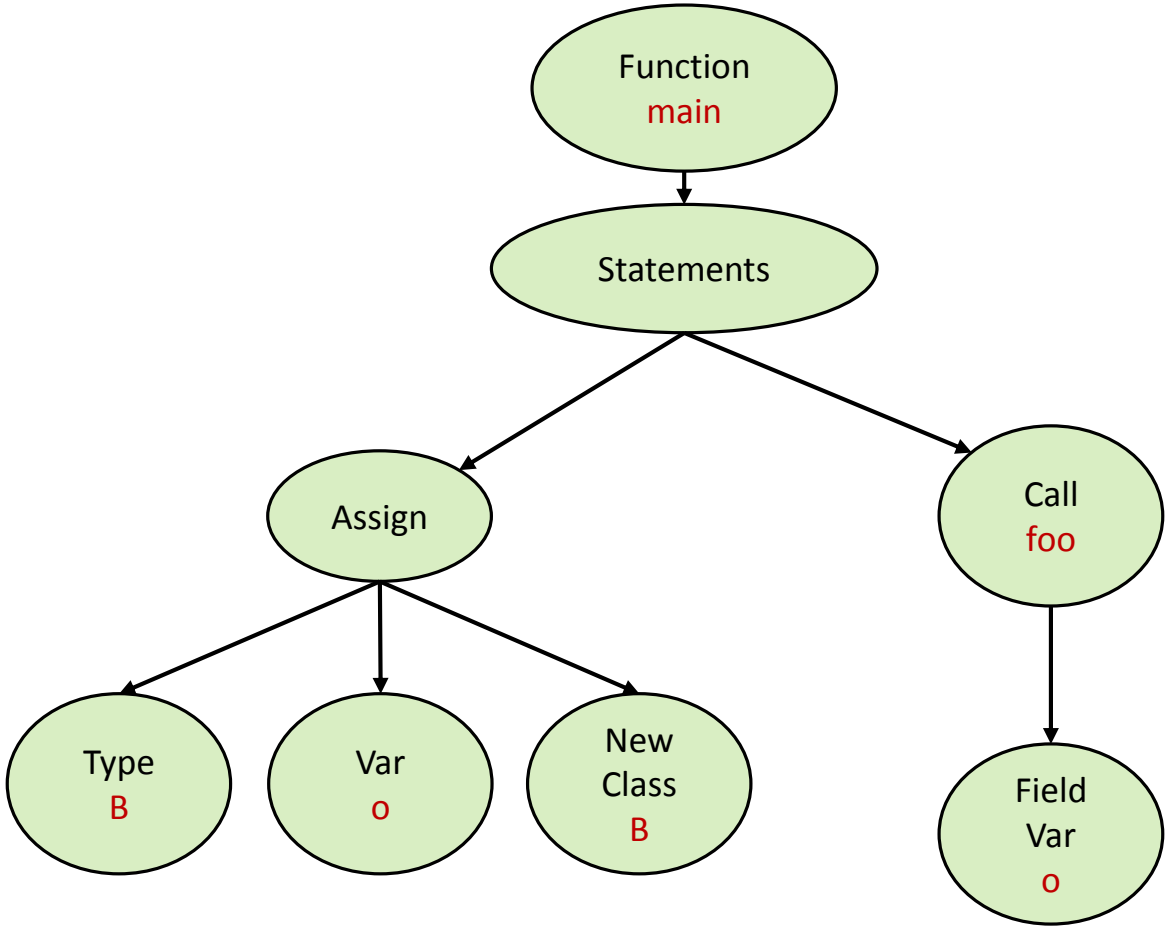
Inheritance

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



Inheritance

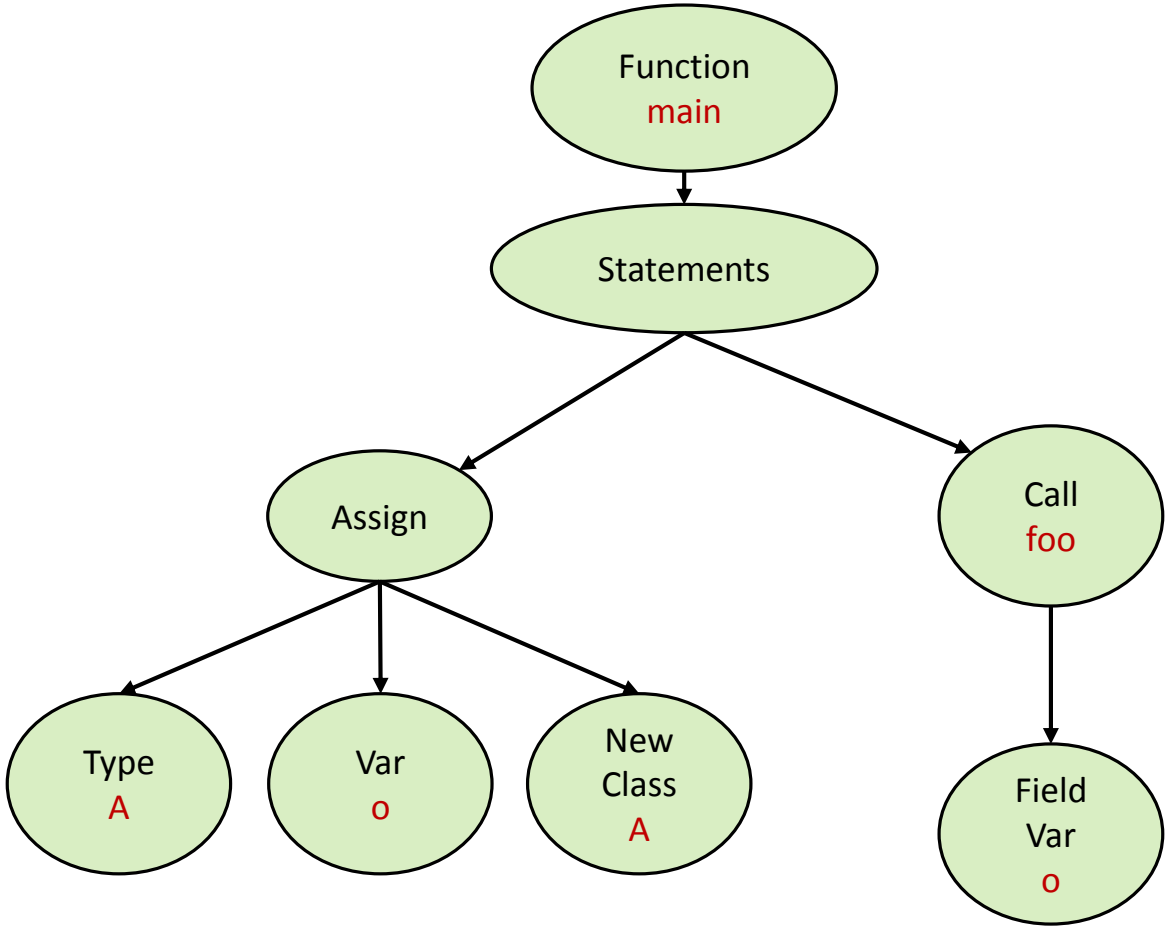
```
class A { }
class B extends A { }
void foo(A a) { }
void main() {
  B o = new B;
  foo(o);
}
```



Valid

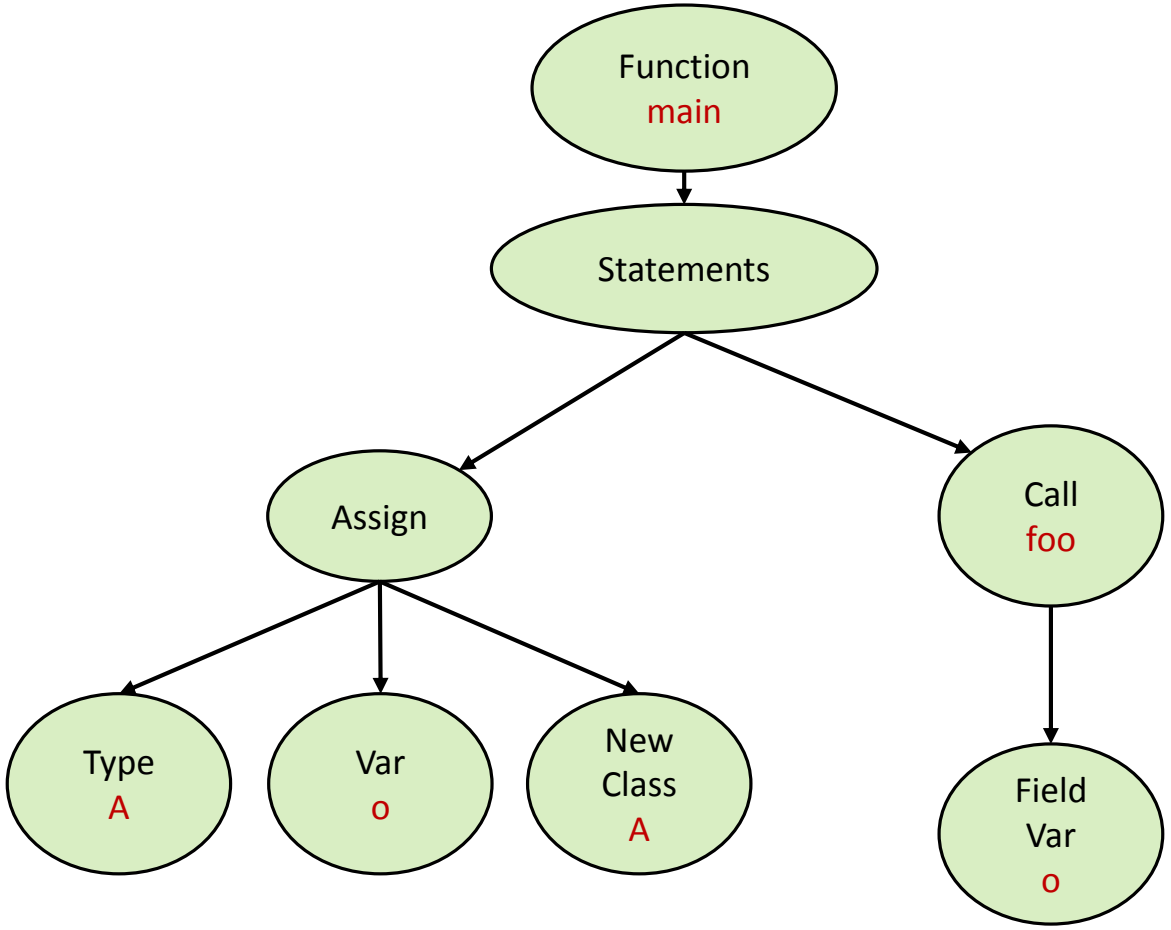
Inheritance

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



Inheritance

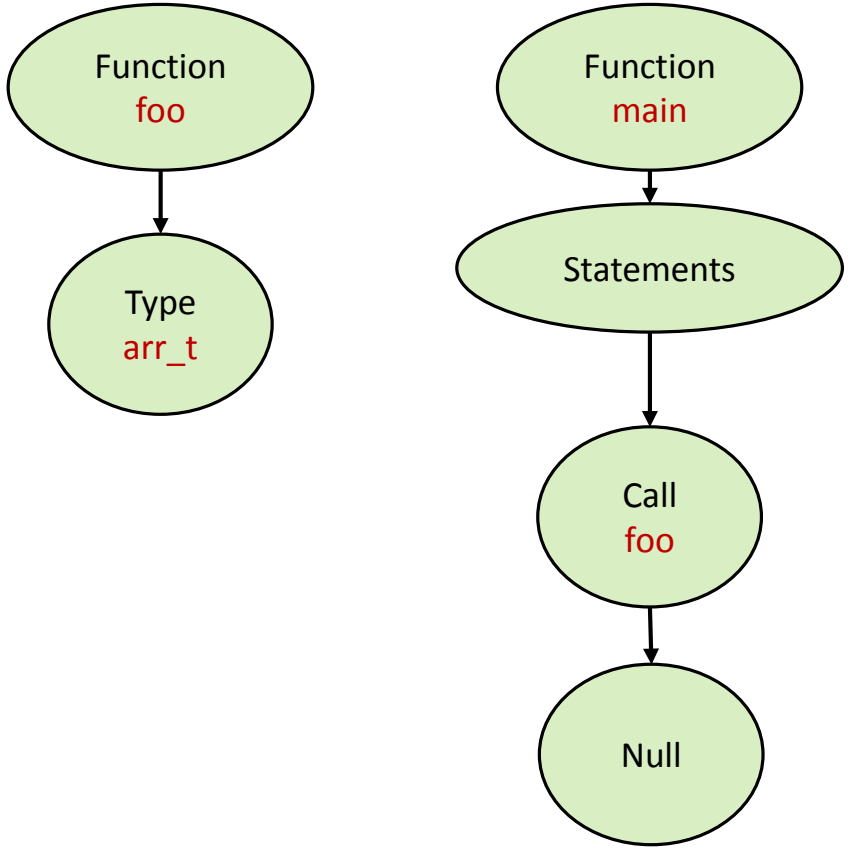
```
class A { }
class B extends A { }
void foo(B b) { }
void main() {
  A o = new A;
  foo(o);
}
```



Invalid

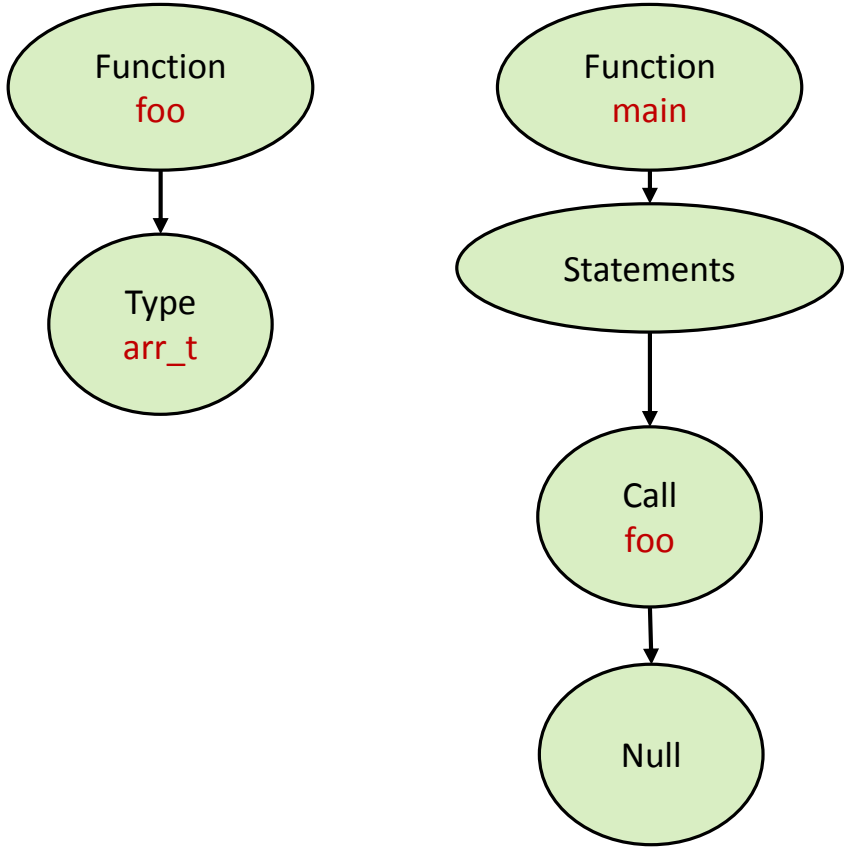
Null

```
typedef int arr_t[];
void foo(arr_t a) { }
void main() {
  foo(null);
}
```



Null

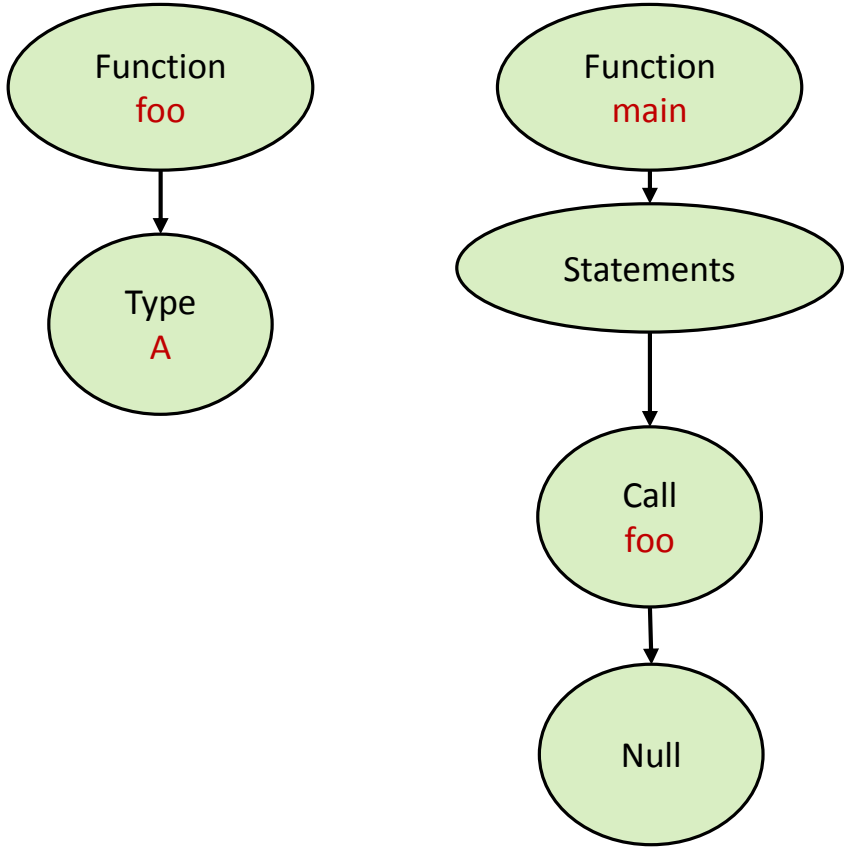
```
typedef int arr_t[];
void foo(arr_t a) { }
void main() {
  foo(null);
}
```



Valid

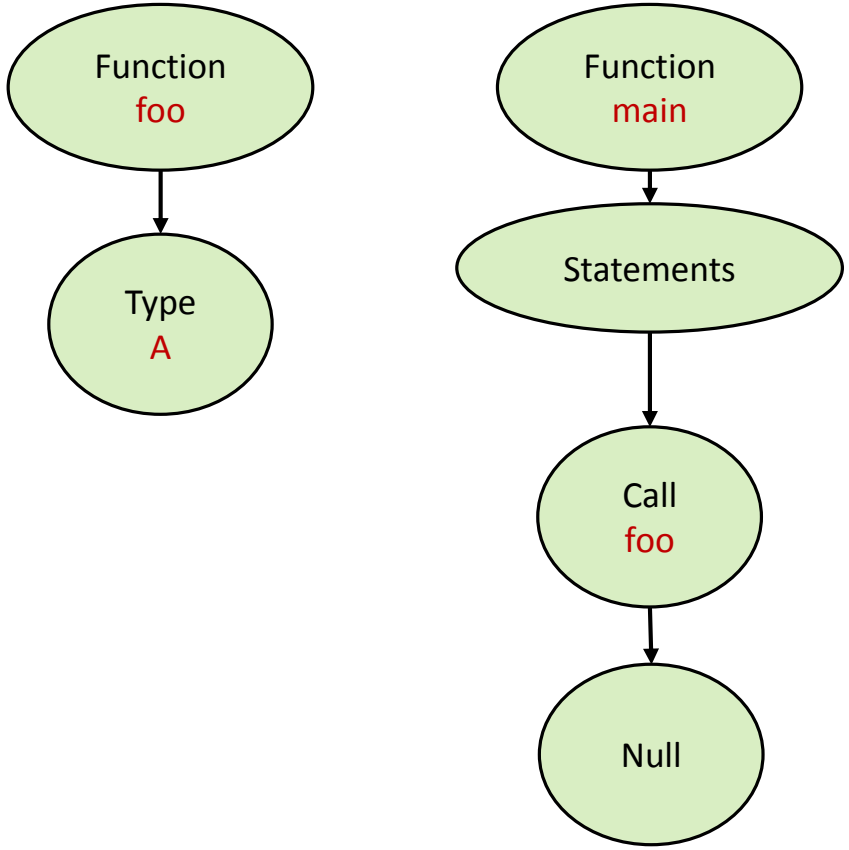
Null

```
class A { };
void foo(A a) { }
void main() {
  foo(null);
}
```



Null

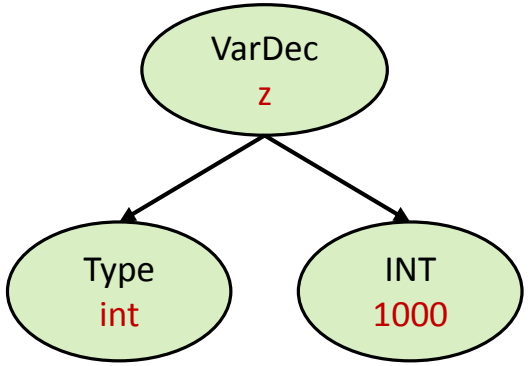
```
class A { };
void foo(A a) { }
void main() {
  foo(null);
}
```

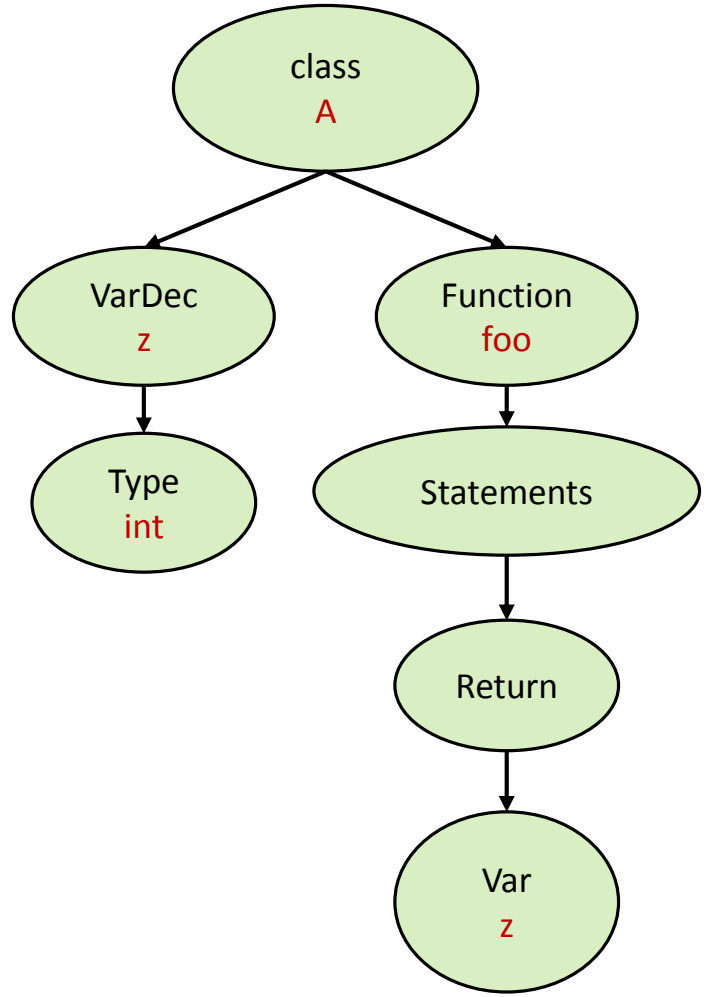


Valid

Scopes

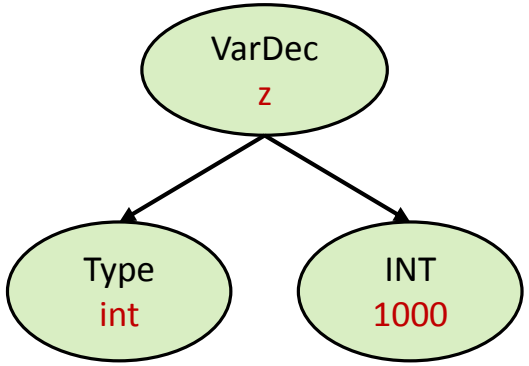
```
int z = 1000;
class A {
  int z;
  int foo() {
    return z;
  }
}
```

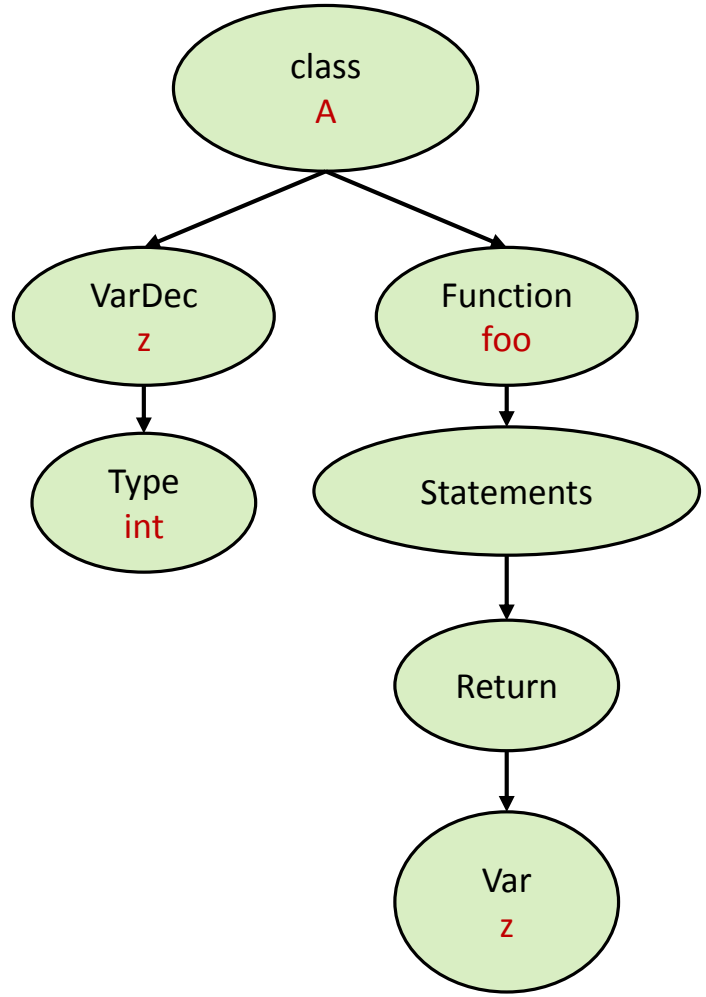




Scopes

```
int z = 1000;
class A {
  int z;
  int foo() {
    return z;
  }
}
```





ID	Type	Kind
z	int	variable
A	…	class

$scope_1$

ID	Type	Kind
z	int	variable
foo	…	function

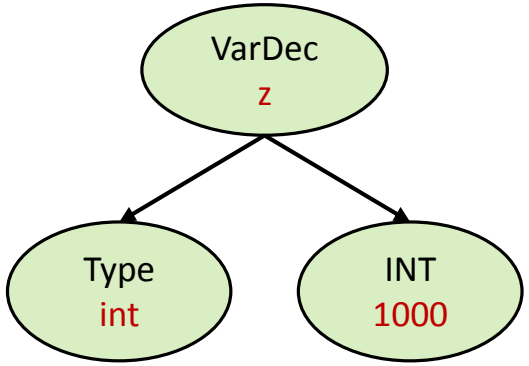
$scope_2$

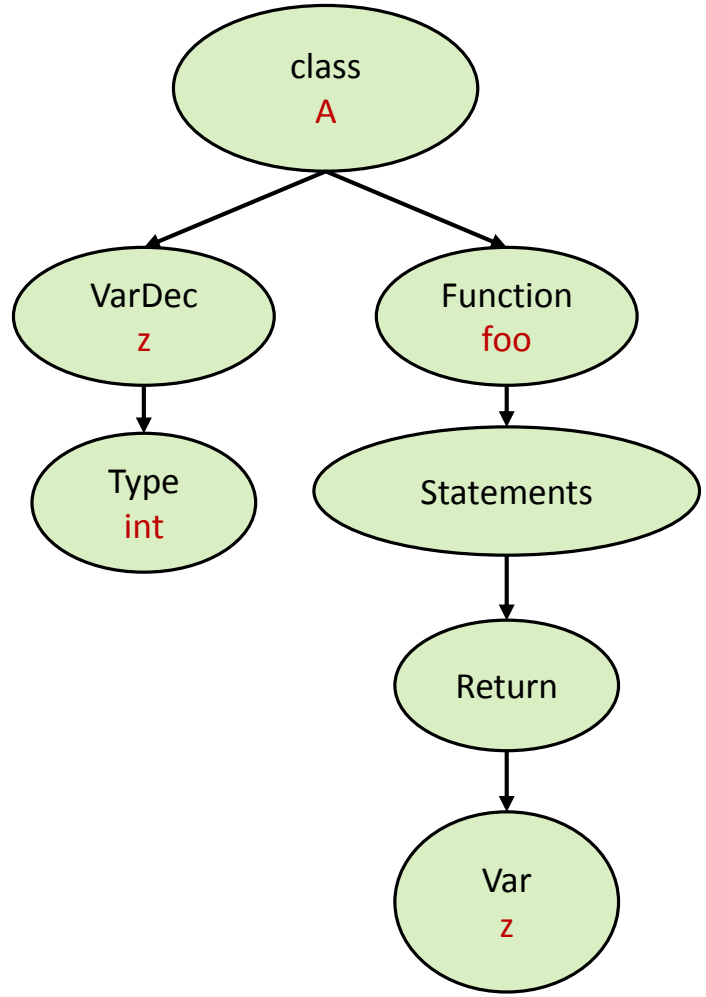
ID	Type	Kind

$scope_3$

Scopes

```
int z = 1000;
class A {
  int z;
  int foo() {
    return z;
  }
}
```

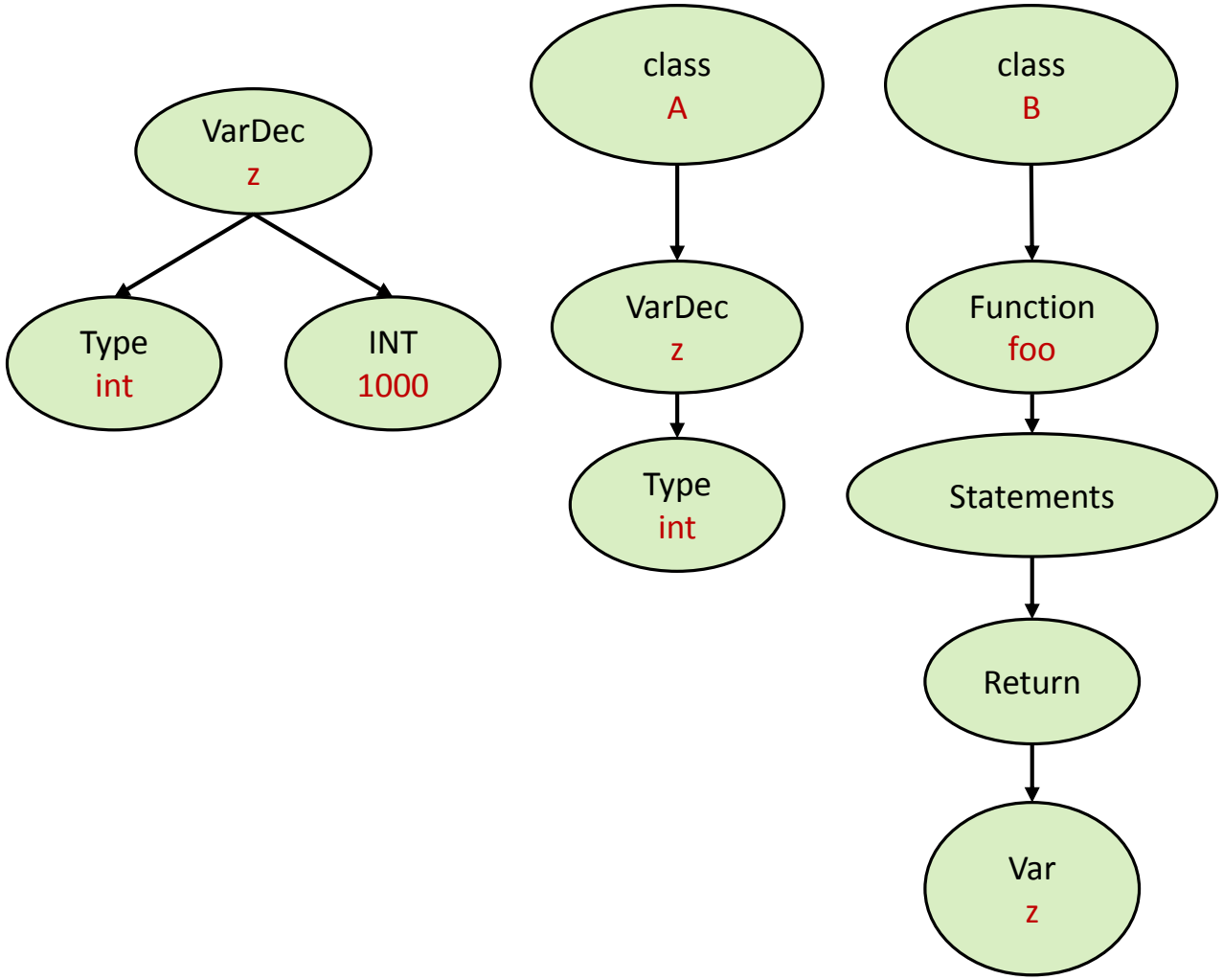




Valid

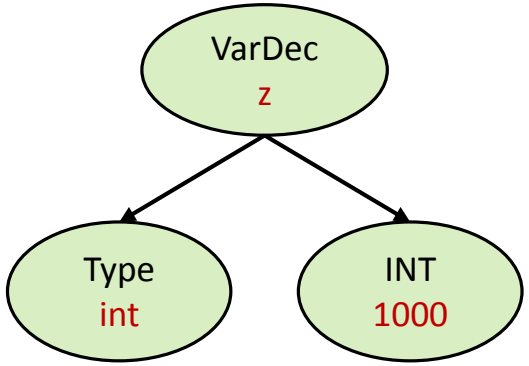
Scopes

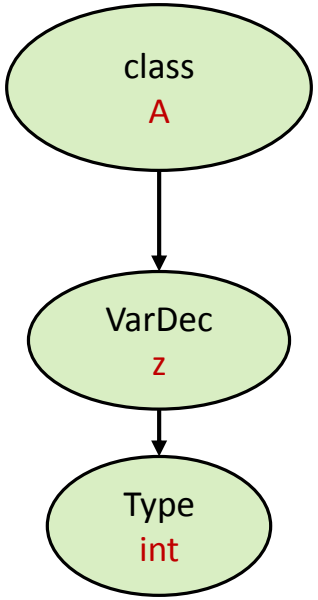
```
int z = 1000;
class A {
  int z;
}
class B extends A {
  int foo() {
    return z;
  }
}
```

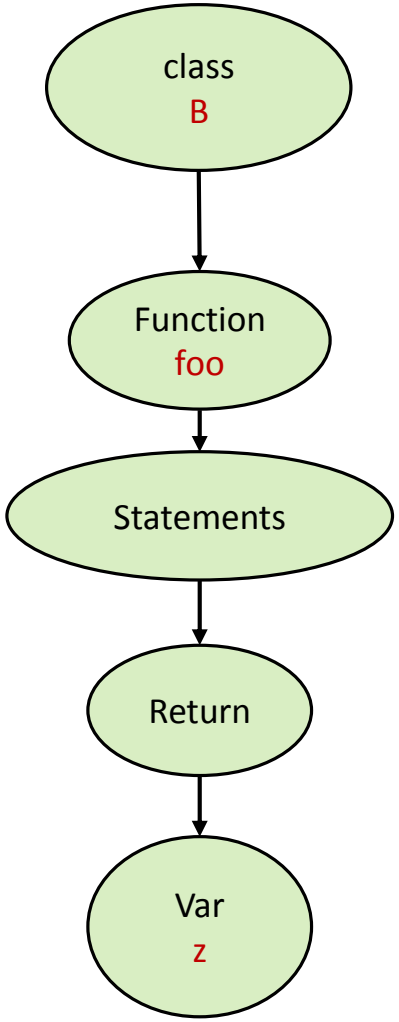


Scopes

```
int z = 1000;
class A {
  int z;
}
class B extends A {
  int foo() {
    return z;
  }
}
```







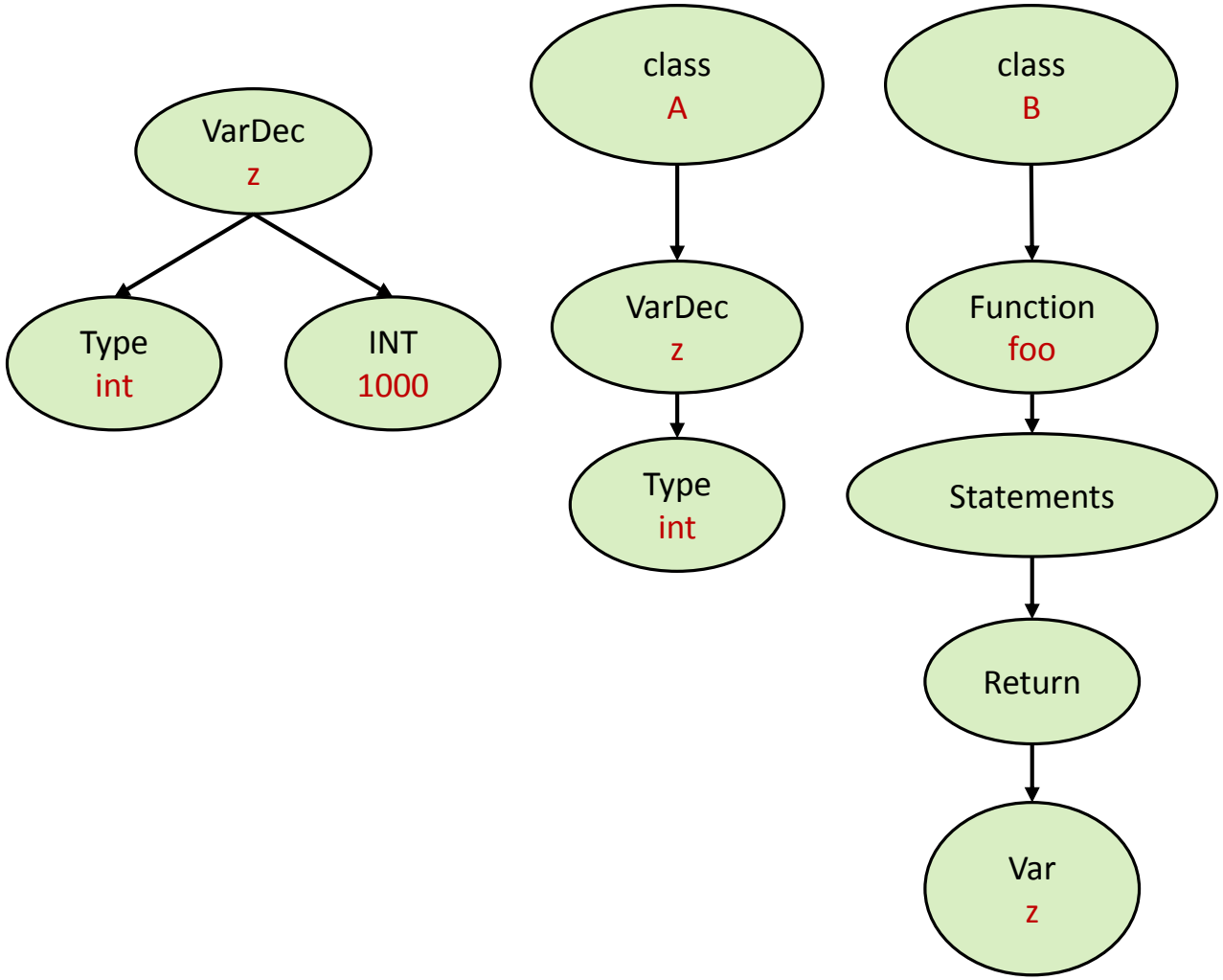
ID	Type	Kind
z	int	variable
A	…	class
B	…	class

ID	Type	Kind
foo	…	function

ID	Type	Kind

Scopes

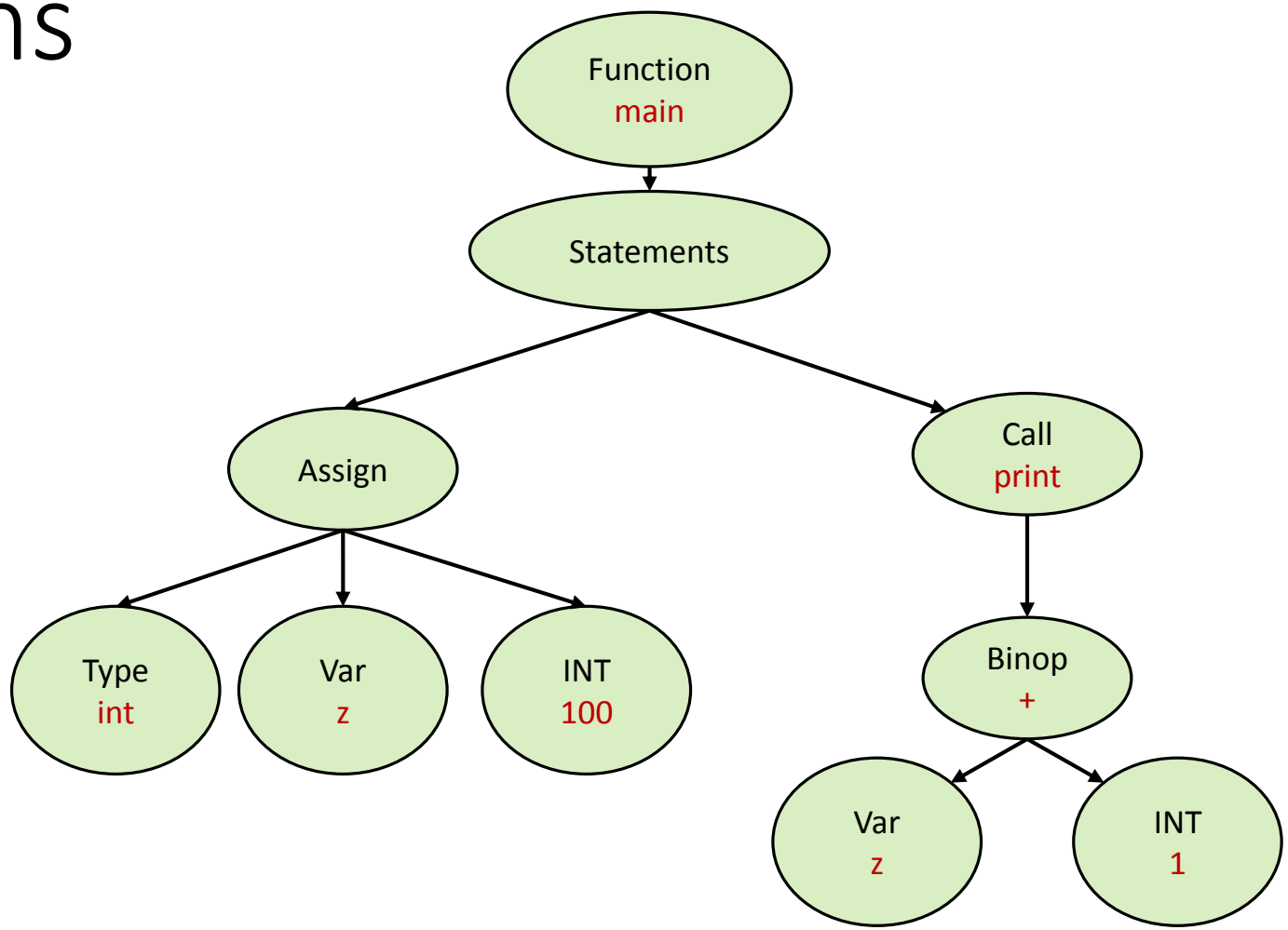
```
int z = 1000;
class A {
  int z;
}
class B extends A {
  int foo() {
    return z;
  }
}
```



Valid

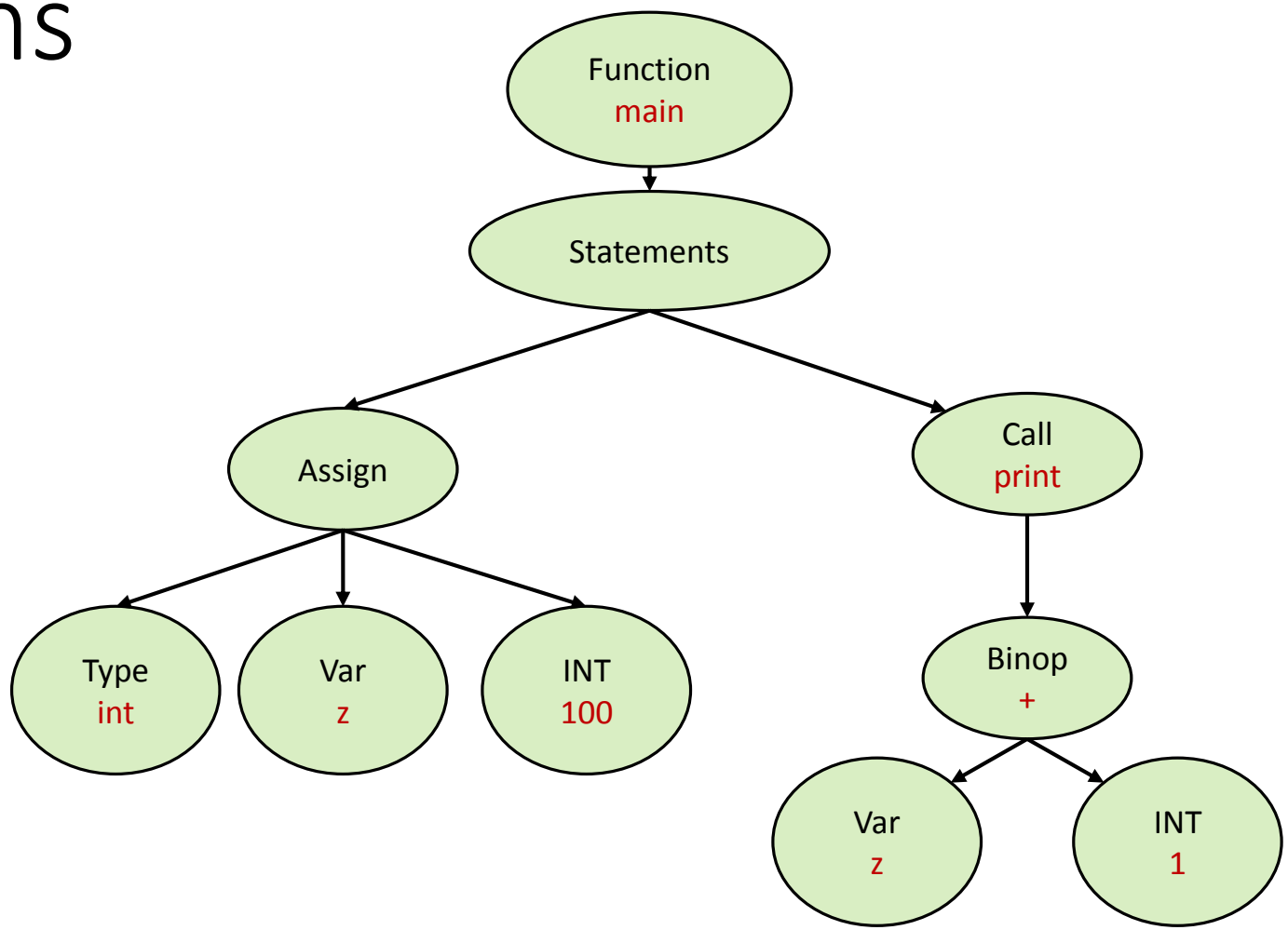
Library Functions

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



Library Functions

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



Valid

Exam Question

We extend the language with automatic type inference:

- Can use **auto** when the declaration has an initial value

```
auto i := 8 + 100;
auto s := “1234”;
class A {}
A a := new A;
auto b := a;
```

Describe the changes required in:

- Lexical analysis
- Syntactic analysis
- Semantic analysis

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
    }

    // return the corresponding type
  }
}
```

Implementation

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

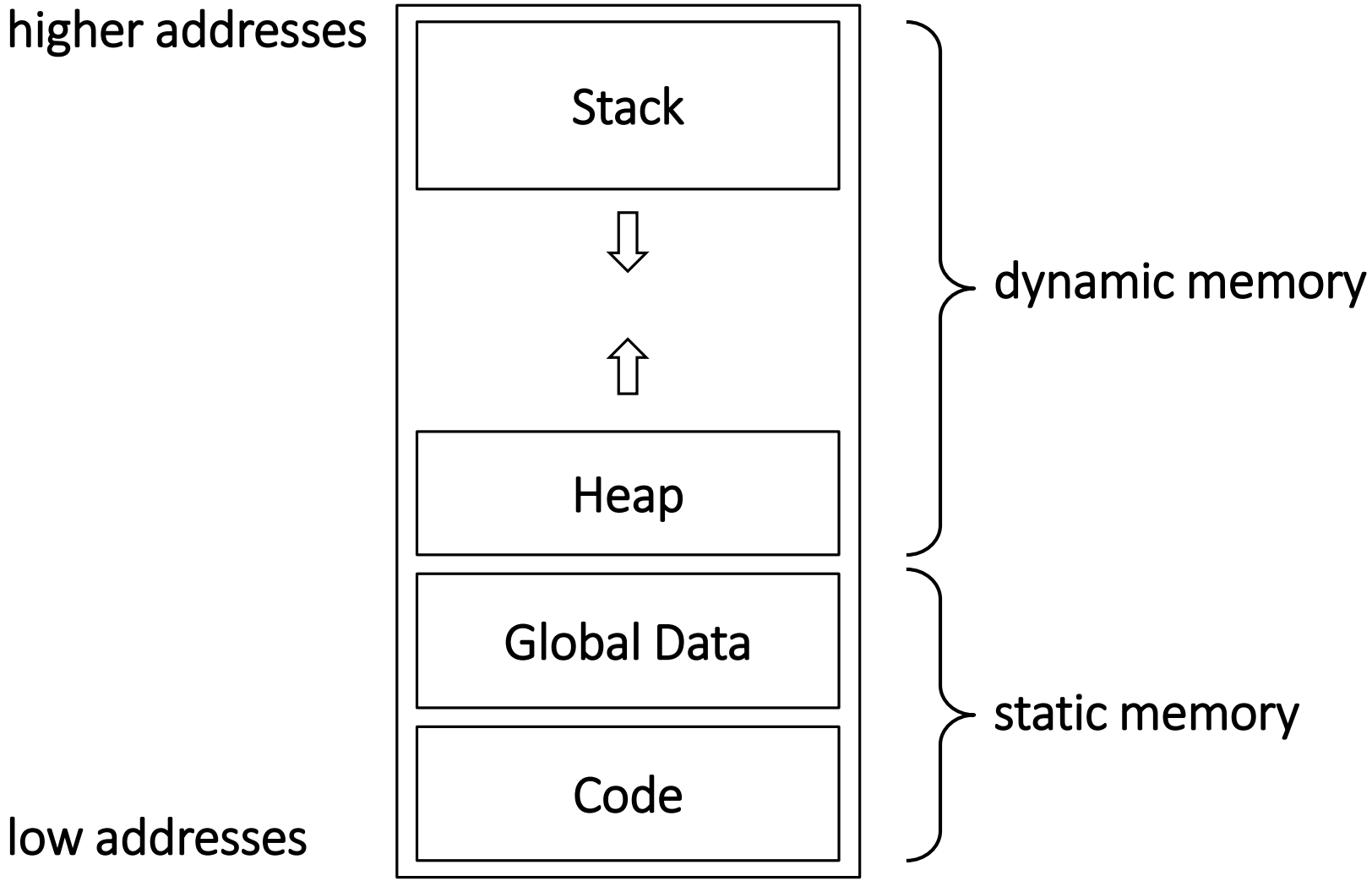
AST Annotations

AST Annotations

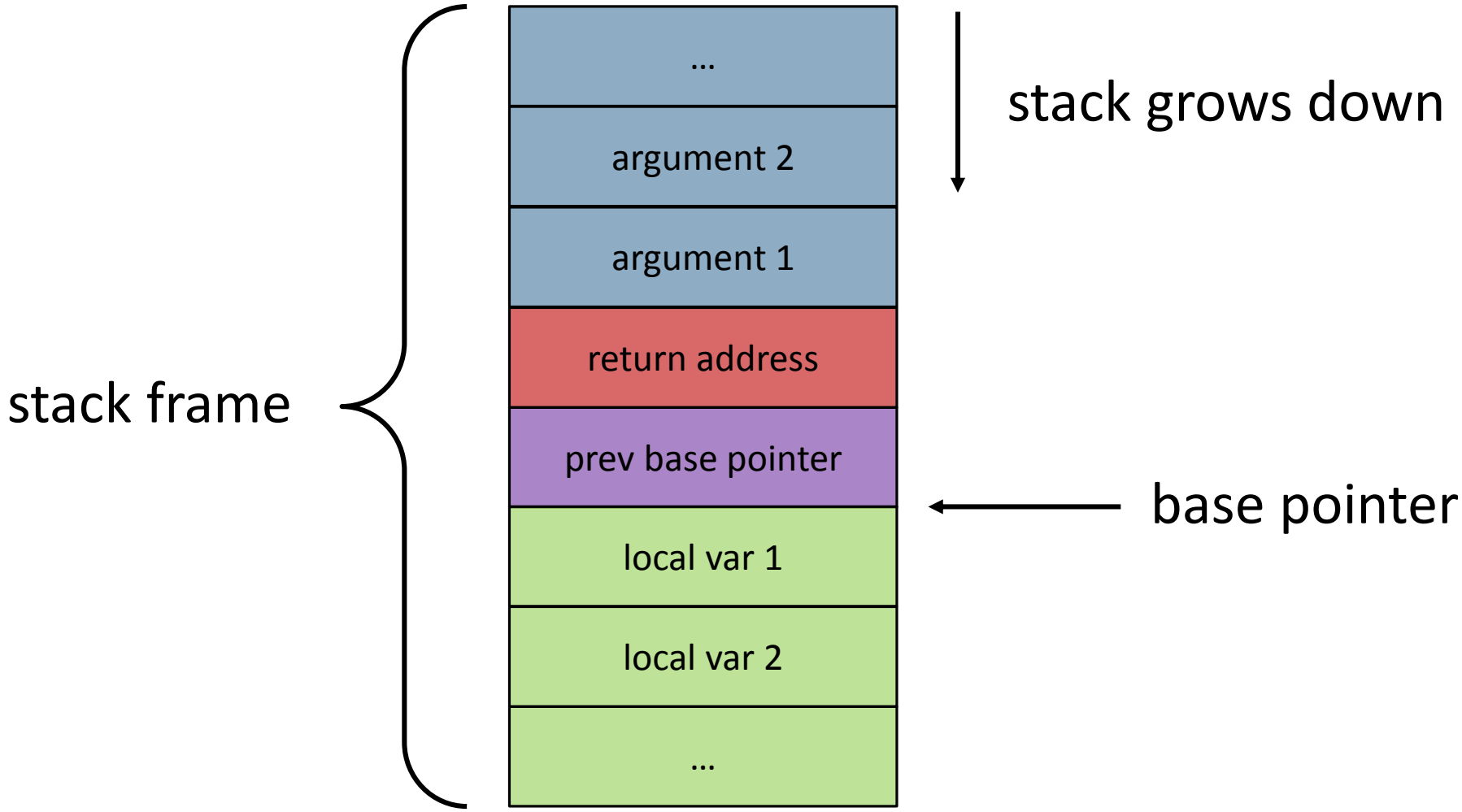
Annotate the AST with information needed for **code generation**:

- Variable offsets
- Parameter offsets
- Class layouts
- Type sizes

Runtime Memory Layout



Stack


...
argument 2
argument 1
return address
prev base pointer
local var 1
local var 2
...
stack grows down
stack frame
base pointer

MIPS: Instructions Set

MIPS has 32 registers:

- t0, ..., t9 (general purpose)
- a0, a1, a2, a3 (arguments)
- v0 (return value)
- sp (stack pointer)
- fp (frame pointer)
- ra (return address)
- …

MIPS: Instructions Set

Setting registers values:

- li
- move

```
li $t0, 3
move $t1, $t2
```

MIPS: Instructions Set

Arithmetic instructions operate on registers and constants:

- add, sub, …

```
add $t2, $t0, $t1
sub $t3, $t1, 7
```

MIPS: Instructions Set

Read from memory:

- lw

```
lw $t0,0($t1)
lw $t0,12($t1)
lw $t0,-8($t1)
```

MIPS: Instructions Set

Write to memory:

- sw

```
sw $t0,0($t1)
sw $t0,12($t1)
sw $t0,-8($t1)
```

MIPS: Instructions Set

Branches:

- jal?

`jal` …

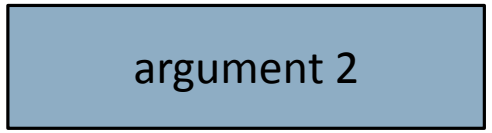
Stack

```
int f(int x, int y){
  int z = x + y;
  return z;
}
int g() {
  int x = f(10, 20)
}
```

```
f:
subu $sp, $sp, 4
sw $ra, 0($sp)
subu $sp, $sp, 4
sw $fp, 0($sp)
move $fp, $sp
sub $sp, $sp, 16
lw $t0, 8($fp)
lw $t1, 12($fp)
add $t2, $t0, $t1
sw $t2, -4($fp)
lw $v0, -4($fp)
move $sp, $fp
lw $fp, 0($sp)
lw $ra, 4($sp)
addu $sp, $sp, 8
jr $ra
```

```
g:
...
li $t0, 20
subu $sp, $sp, 4
sw $t0, 0($sp)
li $t0, 10
subu $sp, $sp, 4
sw $t0, 0($sp)
jal f
addu $sp, $sp, 8
move $t0, $v0
...
```

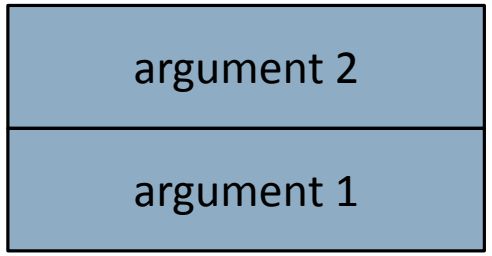
Stack




```
f:
subu $sp, $sp, 4
sw $ra, 0($sp)
subu $sp, $sp, 4
sw $fp, 0($sp)
move $fp, $sp
sub $sp, $sp, 16
lw $t0, 8($fp)
lw $t1, 12($fp)
add $t2, $t0, $t1
sw $t2, -4($fp)
lw $v0, -4($fp)
move $sp, $fp
lw $fp, 0($sp)
lw $ra, 4($sp)
addu $sp, $sp, 8
jr $ra
```

```
g:
...
li $t0, 20
subu $sp, $sp, 4
sw $t0, 0($sp)
li $t0, 10
subu $sp, $sp, 4
sw $t0, 0($sp)
jal f
addu $sp, $sp, 8
move $t0, $v0
...
```

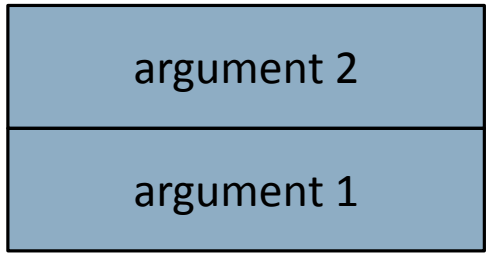
Stack




```
f:
subu $sp, $sp, 4
sw $ra, 0($sp)
subu $sp, $sp, 4
sw $fp, 0($sp)
move $fp, $sp
sub $sp, $sp, 16
lw $t0, 8($fp)
lw $t1, 12($fp)
add $t2, $t0, $t1
sw $t2, -4($fp)
lw $v0, -4($fp)
move $sp, $fp
lw $fp, 0($sp)
lw $ra, 4($sp)
addu $sp, $sp, 8
jr $ra
```

```
g:
...
li $t0, 20
subu $sp, $sp, 4
sw $t0, 0($sp)
li $t0, 10
subu $sp, $sp, 4
sw $t0, 0($sp)
jal f
addu $sp, $sp, 8
move $t0, $v0
...
```

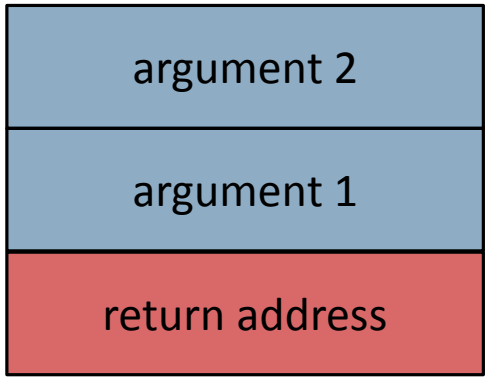
Stack




```
f:
subu $sp, $sp, 4
sw $ra, 0($sp)
subu $sp, $sp, 4
sw $fp, 0($sp)
move $fp, $sp
sub $sp, $sp, 16
lw $t0, 8($fp)
lw $t1, 12($fp)
add $t2, $t0, $t1
sw $t2, -4($fp)
lw $v0, -4($fp)
move $sp, $fp
lw $fp, 0($sp)
lw $ra, 4($sp)
addu $sp, $sp, 8
jr $ra
```

```
g:
...
li $t0, 20
subu $sp, $sp, 4
sw $t0, 0($sp)
li $t0, 10
subu $sp, $sp, 4
sw $t0, 0($sp)
jal f
addu $sp, $sp, 8
move $t0, $v0
...
```

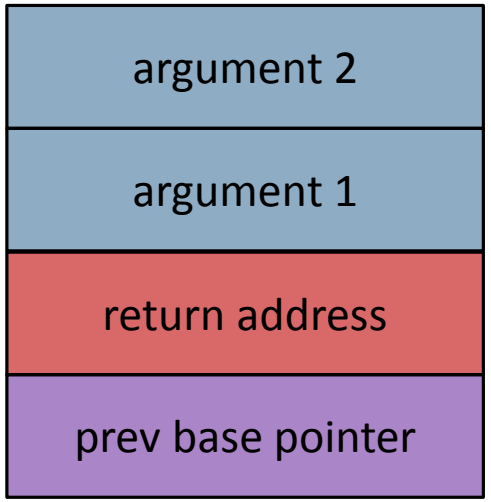
Stack




```
f:
subu $sp, $sp, 4
sw $ra, 0($sp)
subu $sp, $sp, 4
sw $fp, 0($sp)
move $fp, $sp
sub $sp, $sp, 16
lw $t0, 8($fp)
lw $t1, 12($fp)
add $t2, $t0, $t1
sw $t2, -4($fp)
lw $v0, -4($fp)
move $sp, $fp
lw $fp, 0($sp)
lw $ra, 4($sp)
addu $sp, $sp, 8
jr $ra
```

```
g:
...
li $t0, 20
subu $sp, $sp, 4
sw $t0, 0($sp)
li $t0, 10
subu $sp, $sp, 4
sw $t0, 0($sp)
jal f
addu $sp, $sp, 8
move $t0, $v0
...
```

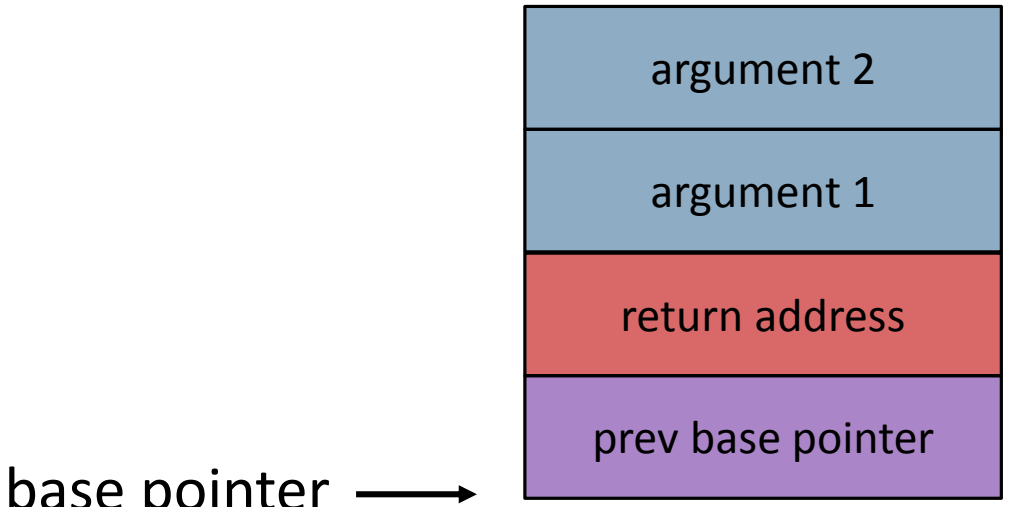
Stack




```
f:
subu $sp, $sp, 4
sw $ra, 0($sp)
subu $sp, $sp, 4
sw $fp, 0($sp)
move $fp, $sp
sub $sp, $sp, 16
lw $t0, 8($fp)
lw $t1, 12($fp)
add $t2, $t0, $t1
sw $t2, -4($fp)
lw $v0, -4($fp)
move $sp, $fp
lw $fp, 0($sp)
lw $ra, 4($sp)
addu $sp, $sp, 8
jr $ra
```

```
g:
...
li $t0, 20
subu $sp, $sp, 4
sw $t0, 0($sp)
li $t0, 10
subu $sp, $sp, 4
sw $t0, 0($sp)
jal f
addu $sp, $sp, 8
move $t0, $v0
...
```

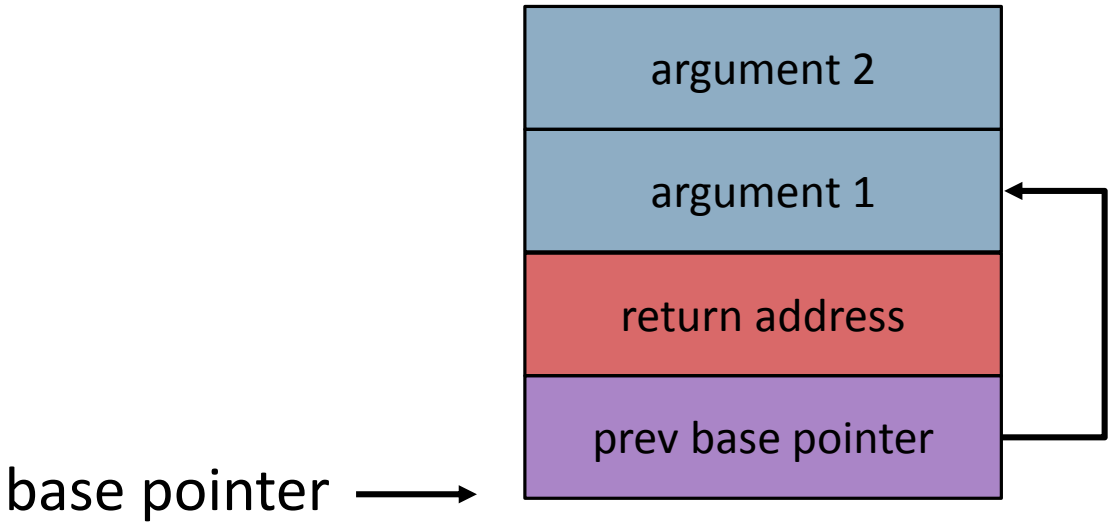
Stack




```
f:
subu $sp, $sp, 4
sw $ra, 0($sp)
subu $sp, $sp, 4
sw $fp, 0($sp)
move $fp, $sp
sub $sp, $sp, 16
lw $t0, 8($fp)
lw $t1, 12($fp)
add $t2, $t0, $t1
sw $t2, -4($fp)
lw $v0, -4($fp)
move $sp, $fp
lw $fp, 0($sp)
lw $ra, 4($sp)
addu $sp, $sp, 8
jr $ra
```

```
g:
...
li $t0, 20
subu $sp, $sp, 4
sw $t0, 0($sp)
li $t0, 10
subu $sp, $sp, 4
sw $t0, 0($sp)
jal f
addu $sp, $sp, 8
move $t0, $v0
...
```

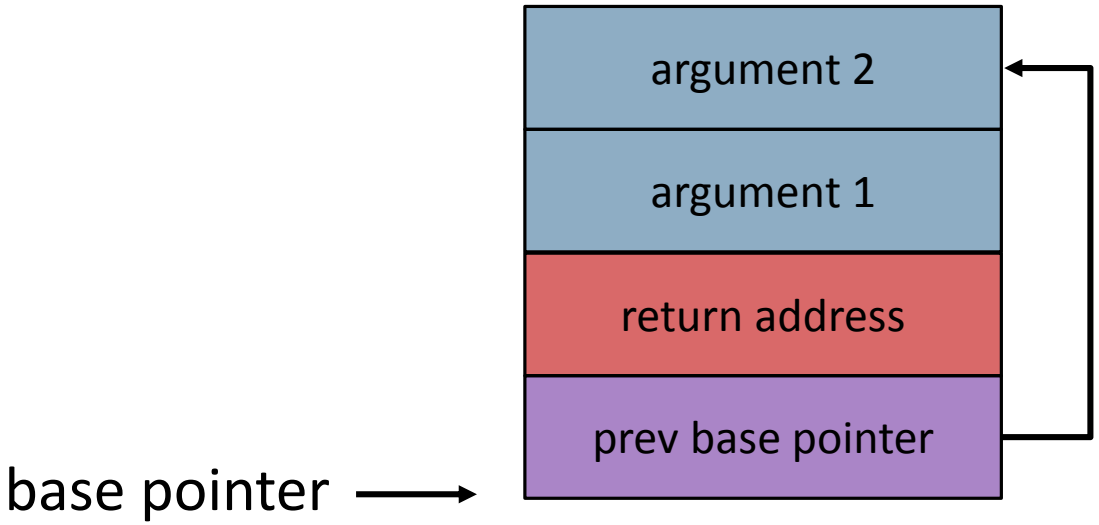
Stack




```
f:
subu $sp, $sp, 4
sw $ra, 0($sp)
subu $sp, $sp, 4
sw $fp, 0($sp)
move $fp, $sp
sub $sp, $sp, 16
lw $t0, 8($fp)
lw $t1, 12($fp)
add $t2, $t0, $t1
sw $t2, -4($fp)
lw $v0, -4($fp)
move $sp, $fp
lw $fp, 0($sp)
lw $ra, 4($sp)
addu $sp, $sp, 8
jr $ra
```

```
g:
...
li $t0, 20
subu $sp, $sp, 4
sw $t0, 0($sp)
li $t0, 10
subu $sp, $sp, 4
sw $t0, 0($sp)
jal f
addu $sp, $sp, 8
move $t0, $v0
...
```

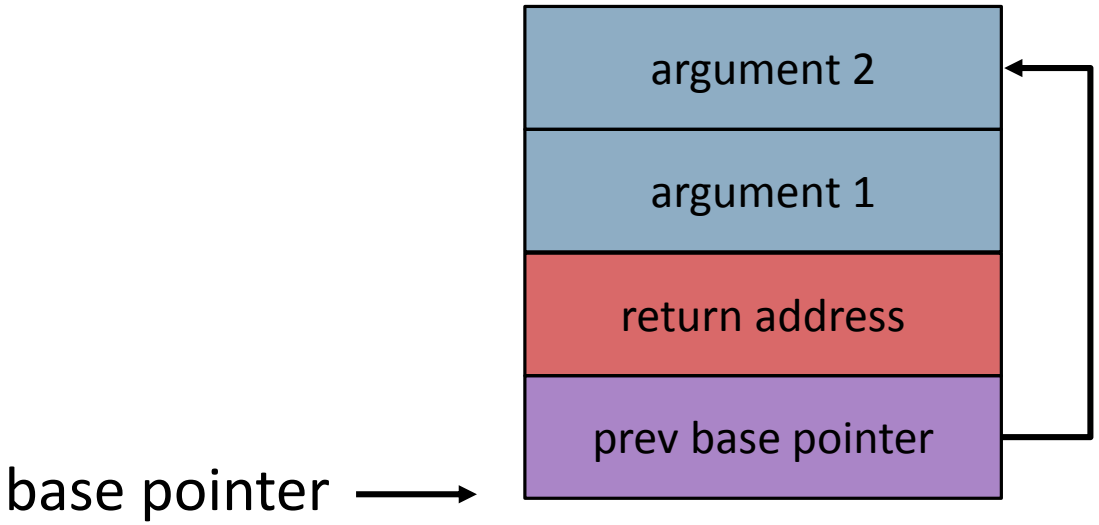
Stack




```
f:
subu $sp, $sp, 4
sw $ra, 0($sp)
subu $sp, $sp, 4
sw $fp, 0($sp)
move $fp, $sp
sub $sp, $sp, 16
lw $t0, 8($fp)
lw $t1, 12($fp)
add $t2, $t0, $t1
sw $t2, -4($fp)
lw $v0, -4($fp)
move $sp, $fp
lw $fp, 0($sp)
lw $ra, 4($sp)
addu $sp, $sp, 8
jr $ra
```

```
g:
...
li $t0, 20
subu $sp, $sp, 4
sw $t0, 0($sp)
li $t0, 10
subu $sp, $sp, 4
sw $t0, 0($sp)
jal f
addu $sp, $sp, 8
move $t0, $v0
...
```

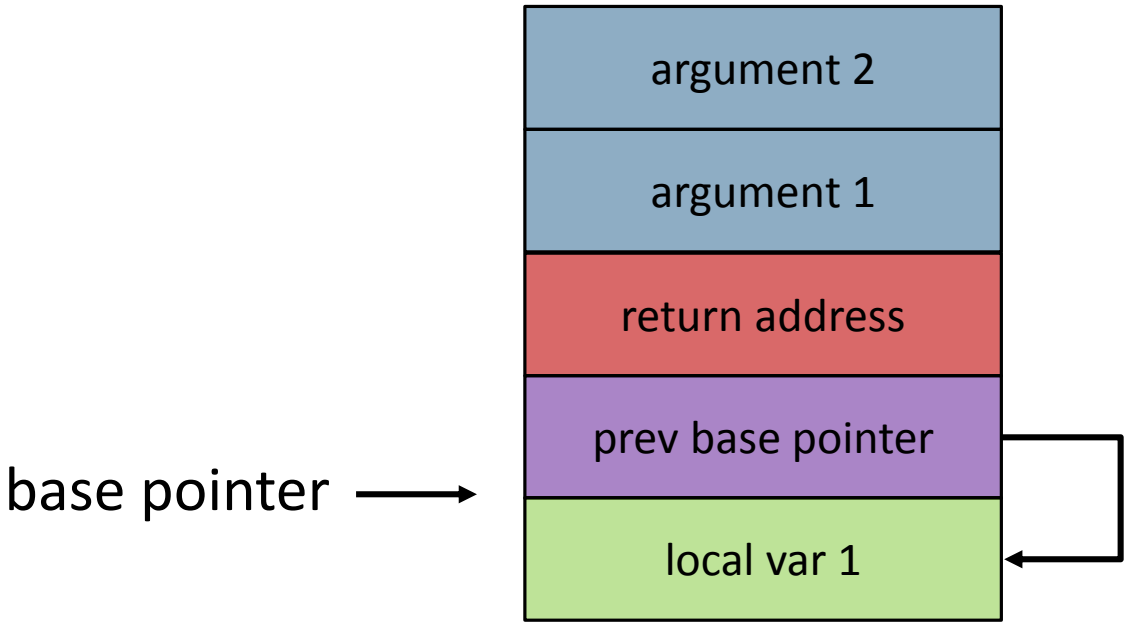
Stack




```
f:
subu $sp, $sp, 4
sw $ra, 0($sp)
subu $sp, $sp, 4
sw $fp, 0($sp)
move $fp, $sp
sub $sp, $sp, 16
lw $t0, 8($fp)
lw $t1, 12($fp)
add $t2, $t0, $t1
sw $t2, -4($fp)
lw $v0, -4($fp)
move $sp, $fp
lw $fp, 0($sp)
lw $ra, 4($sp)
addu $sp, $sp, 8
jr $ra
```

```
g:
...
li $t0, 20
subu $sp, $sp, 4
sw $t0, 0($sp)
li $t0, 10
subu $sp, $sp, 4
sw $t0, 0($sp)
jal f
addu $sp, $sp, 8
move $t0, $v0
...
```

Stack




```
f:
subu $sp, $sp, 4
sw $ra, 0($sp)
subu $sp, $sp, 4
sw $fp, 0($sp)
move $fp, $sp
sub $sp, $sp, 16
lw $t0, 8($fp)
lw $t1, 12($fp)
add $t2, $t0, $t1
sw $t2, -4($fp)
lw $v0, -4($fp)
move $sp, $fp
lw $fp, 0($sp)
lw $ra, 4($sp)
addu $sp, $sp, 8
jr $ra
```

```
g:
...
li $t0, 20
subu $sp, $sp, 4
sw $t0, 0($sp)
li $t0, 10
subu $sp, $sp, 4
sw $t0, 0($sp)
jal f
addu $sp, $sp, 8
move $t0, $v0
...
```

Variable Offsets

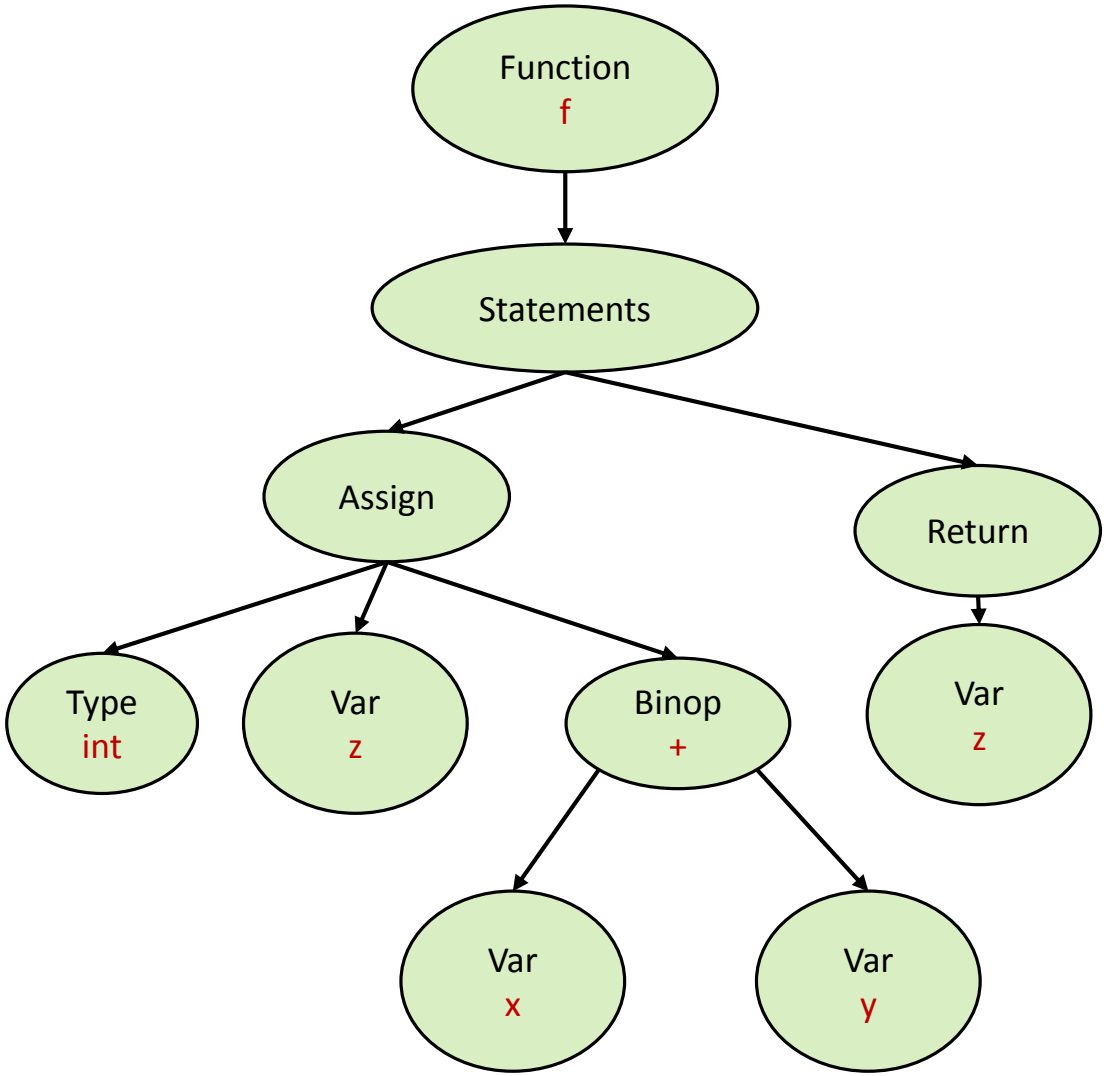
Machine code **does not** contain names of:

- Local variables
- Parameters

Instead, we use offsets **relatively** to the **stack base pointer**

Variable Offsets

```
int f(int x, int y){
  int z = x + y;
  return z;
}
```

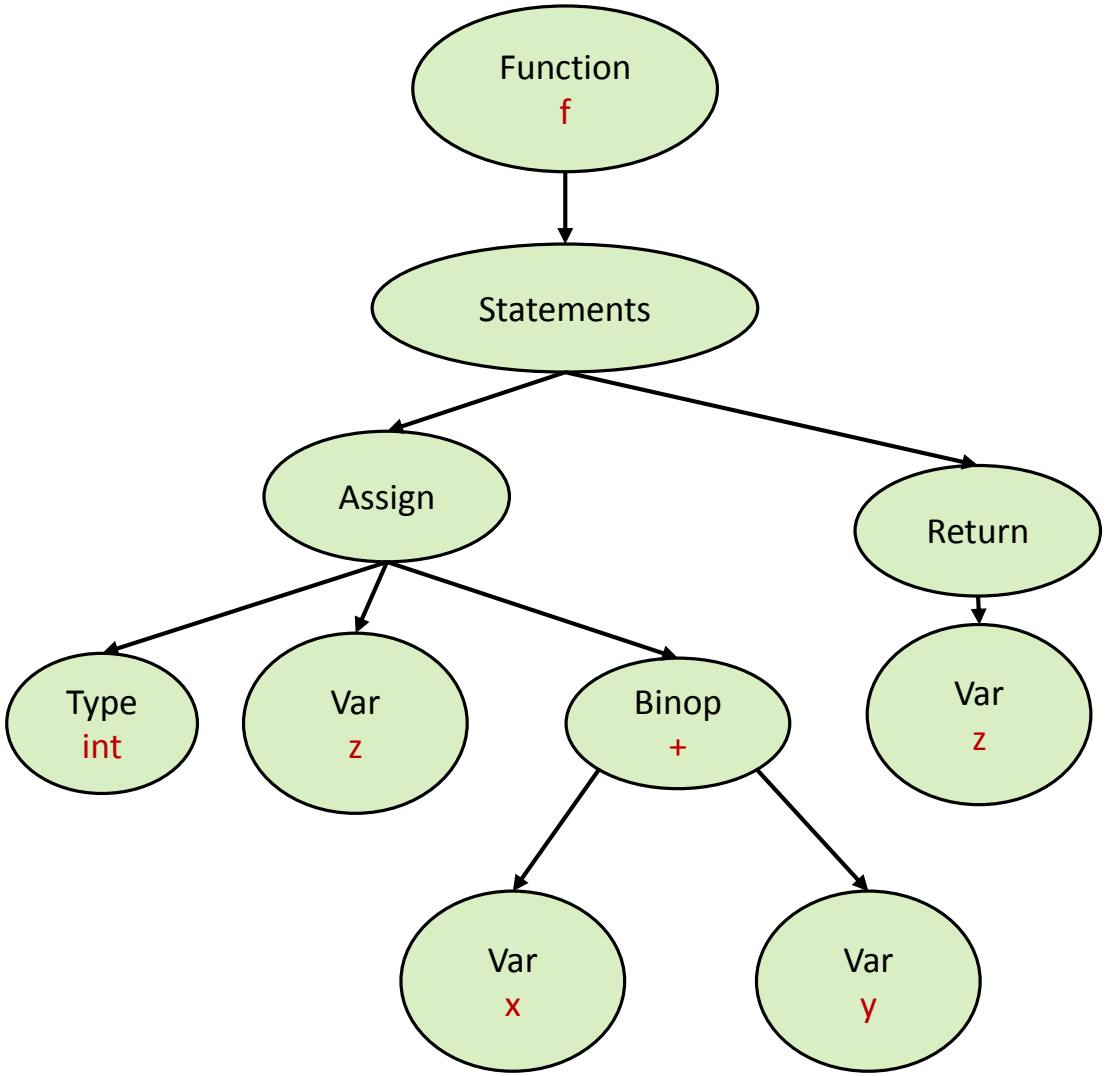


Variable Offsets

```
int f(int x, int y){
  int z = x + y;
  return z;
}
```

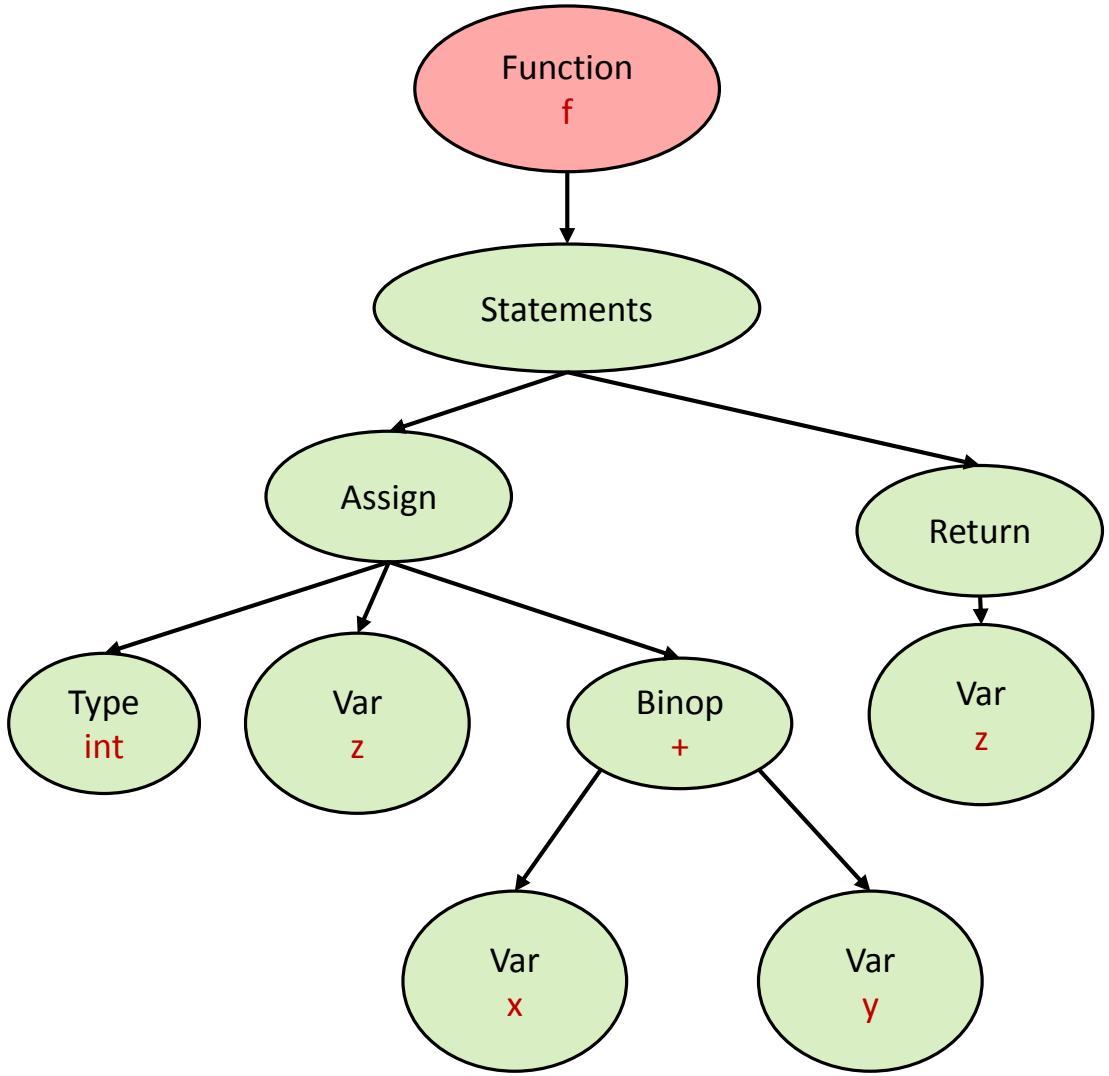
ID	Type	Kind
f	…	function

$scope_1$



Variable Offsets

```
int f(int x, int y){
  int z = x + y;
  return z;
}
```



ID	Type	Kind
f	…	function

ID	Type	Kind

$scope_1$ $scope_2$

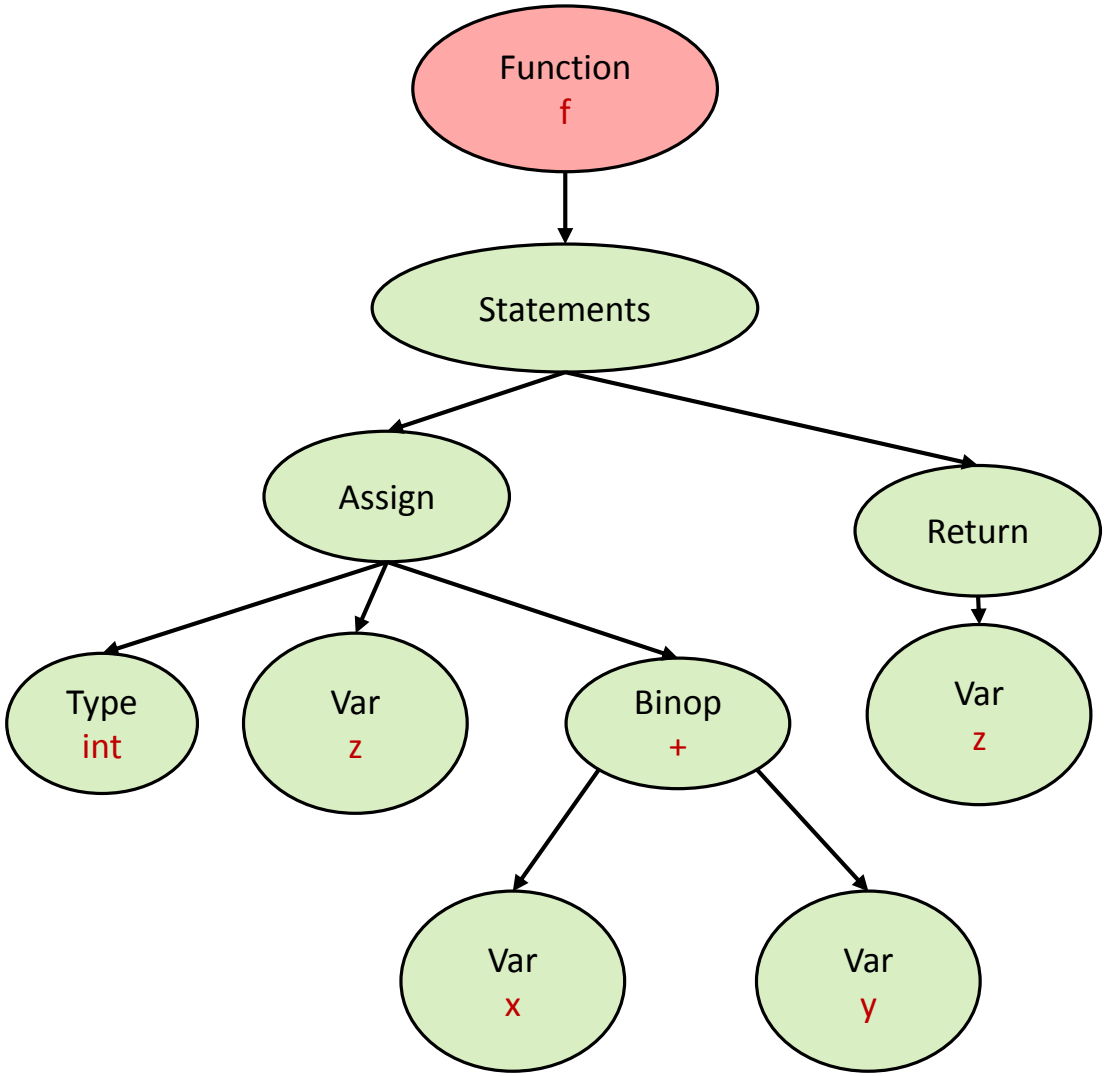
Variable Offsets

```
int f(int x, int y){
  int z = x + y;
  return z;
}
```

ID	Type	Kind
f	…	function

ID	Type	Kind
x	int	variable

$scope_1$ $scope_2$



Variable Offsets

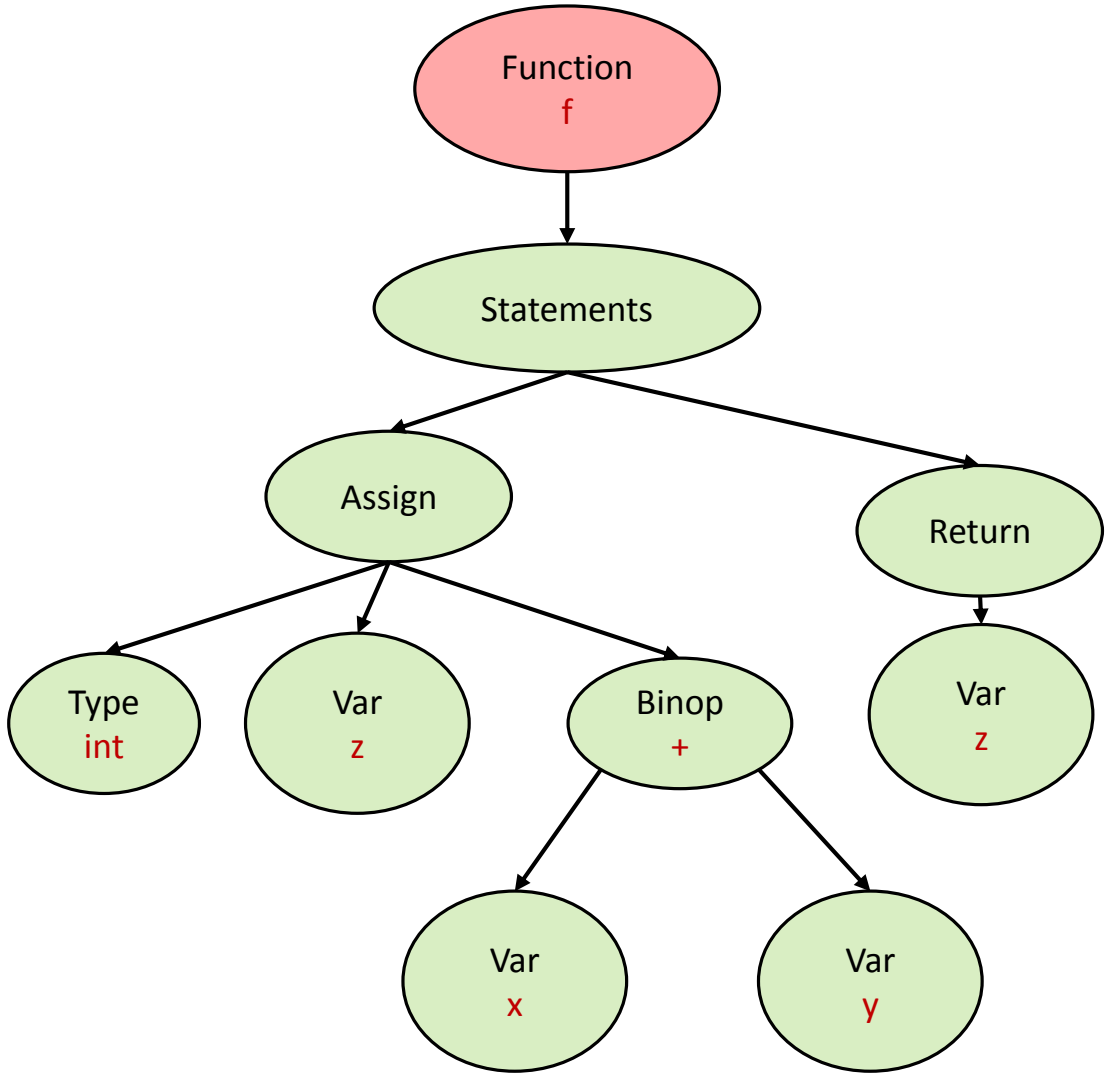
```
int f(int x, int y){
  int z = x + y;
  return z;
}
```

ID	Type	Kind
f	…	function

$scope_1$

ID	Type	Kind
x	int	variable
y	int	variable

$scope_2$



Variable Offsets

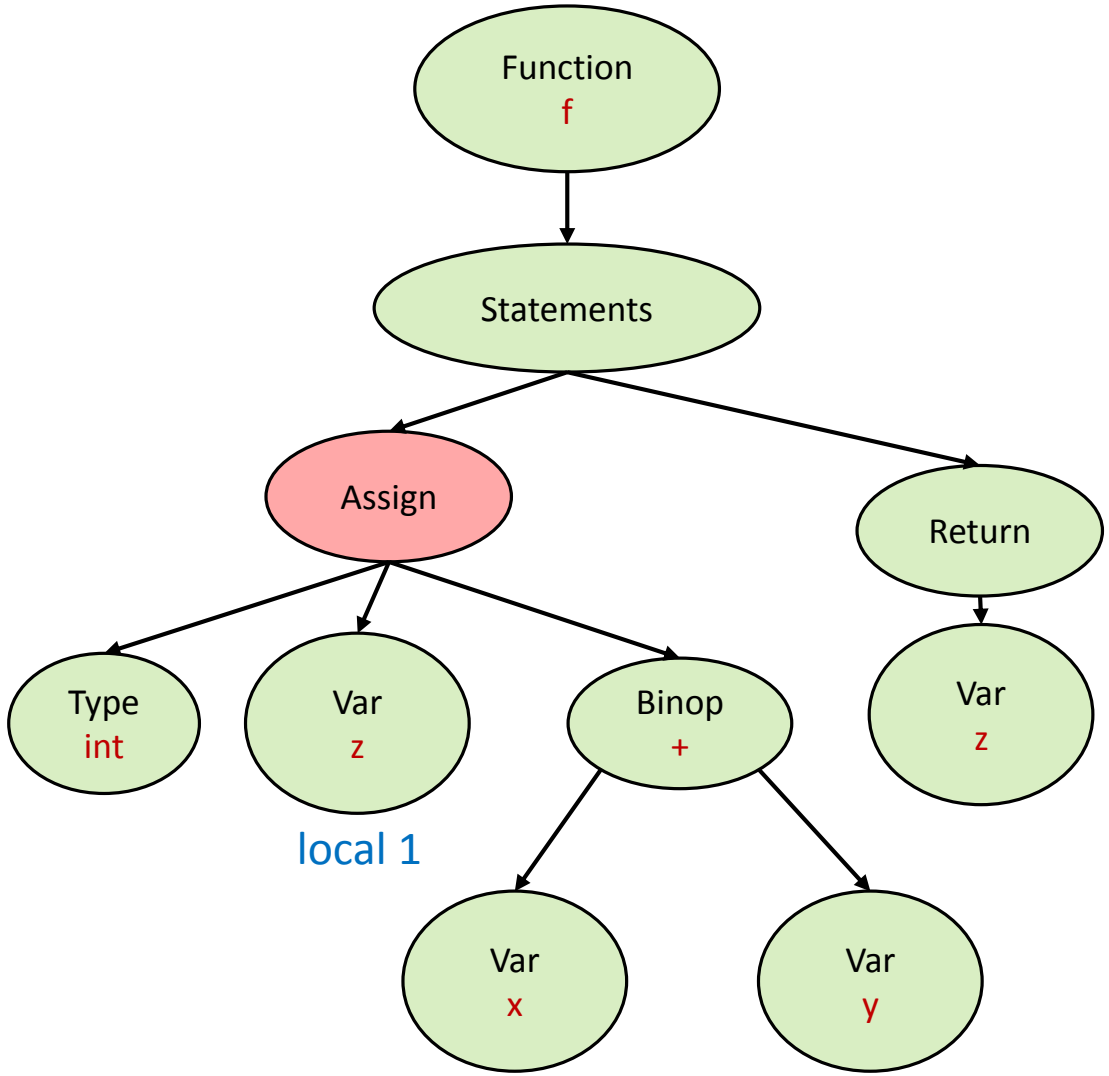
```
int f(int x, int y){
  int z = x + y;
  return z;
}
```

ID	Type	Kind
f	…	function

$scope_1$

ID	Type	Kind
x	int	variable
y	int	variable
z	int	variable

$scope_2$



Variable Offsets

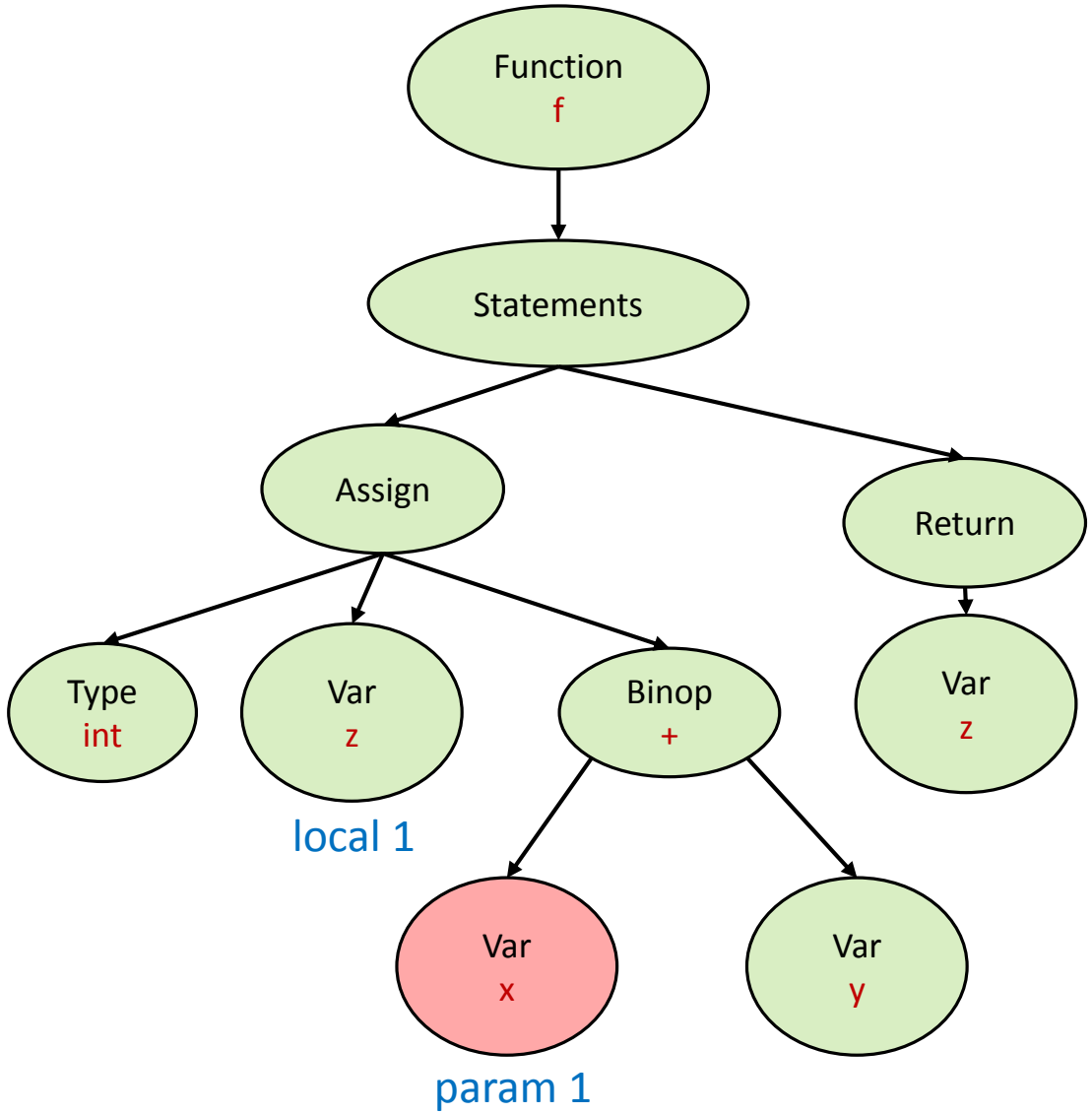
```
int f(int x, int y){
  int z = x + y;
  return z;
}
```

ID	Type	Kind
f	…	function

$scope_1$

ID	Type	Kind
x	int	variable
y	int	variable
z	int	variable

$scope_2$



Variable Offsets

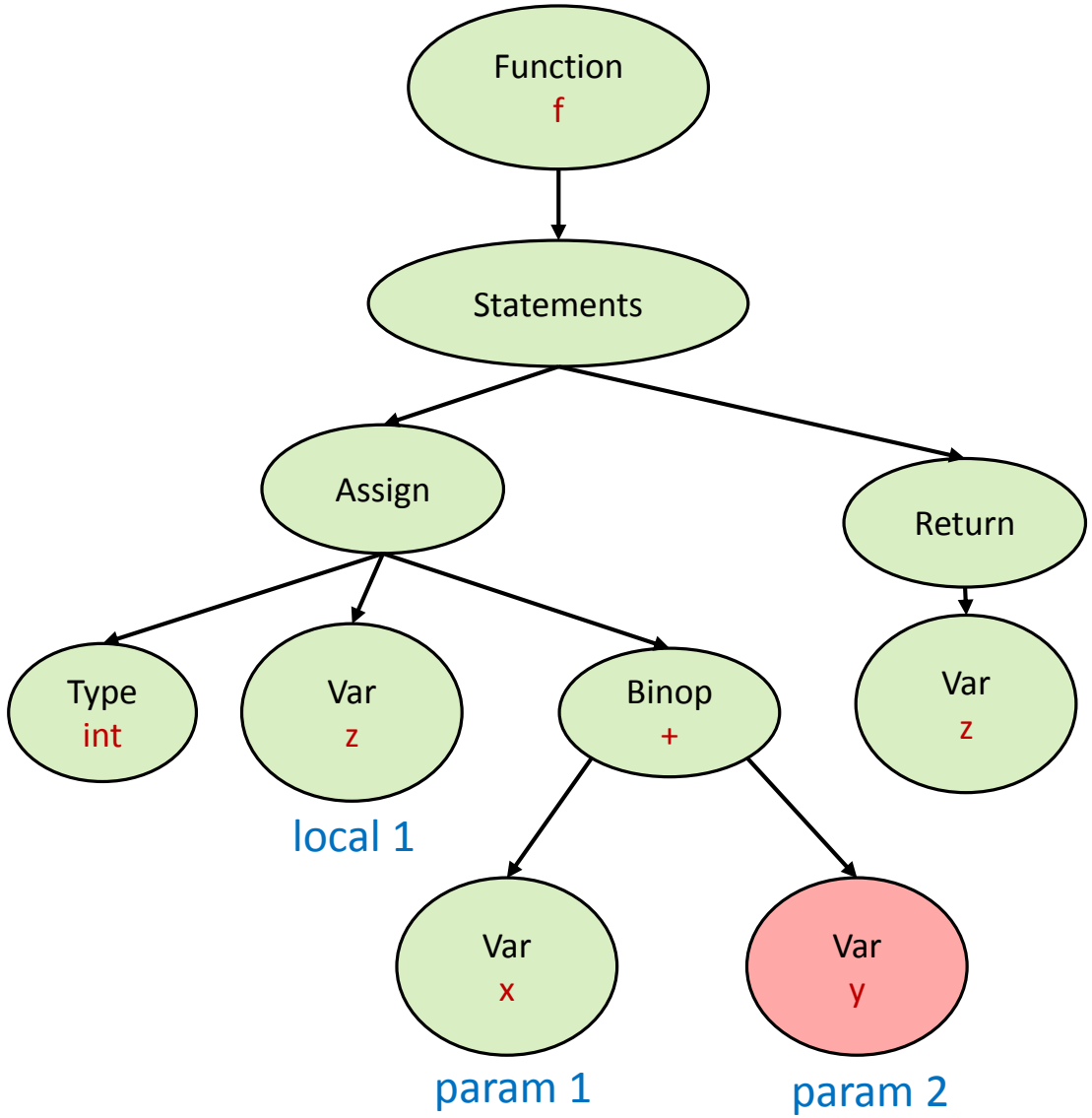
```
int f(int x, int y){
  int z = x + y;
  return z;
}
```

ID	Type	Kind
f	…	function

$scope_1$

ID	Type	Kind
x	int	variable
y	int	variable
z	int	variable

$scope_2$



Variable Offsets

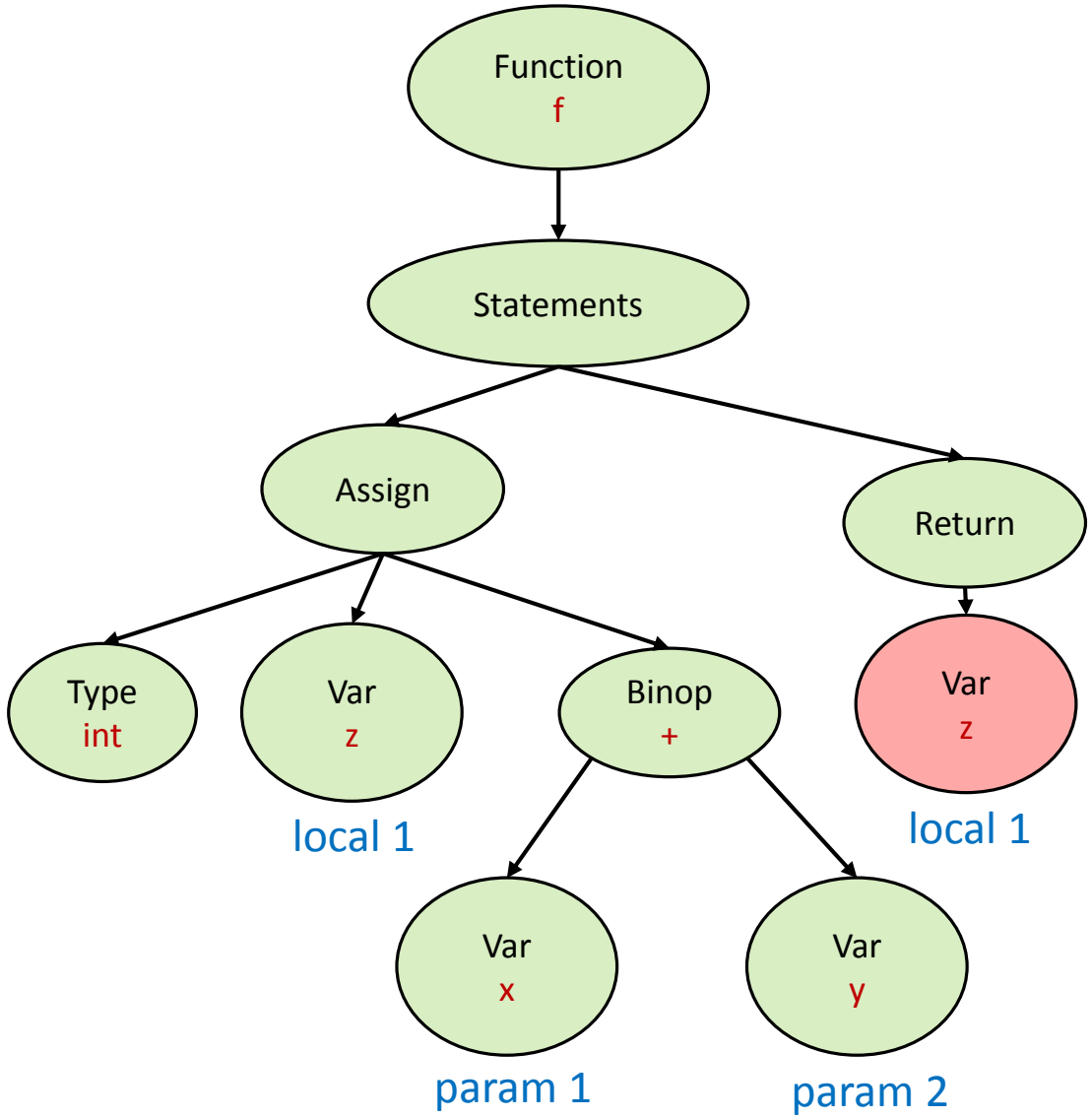
```
int f(int x, int y){
  int z = x + y;
  return z;
}
```

ID	Type	Kind
f	…	function

$scope_1$

ID	Type	Kind
x	int	variable
y	int	variable
z	int	variable

$scope_2$



Variable Offsets

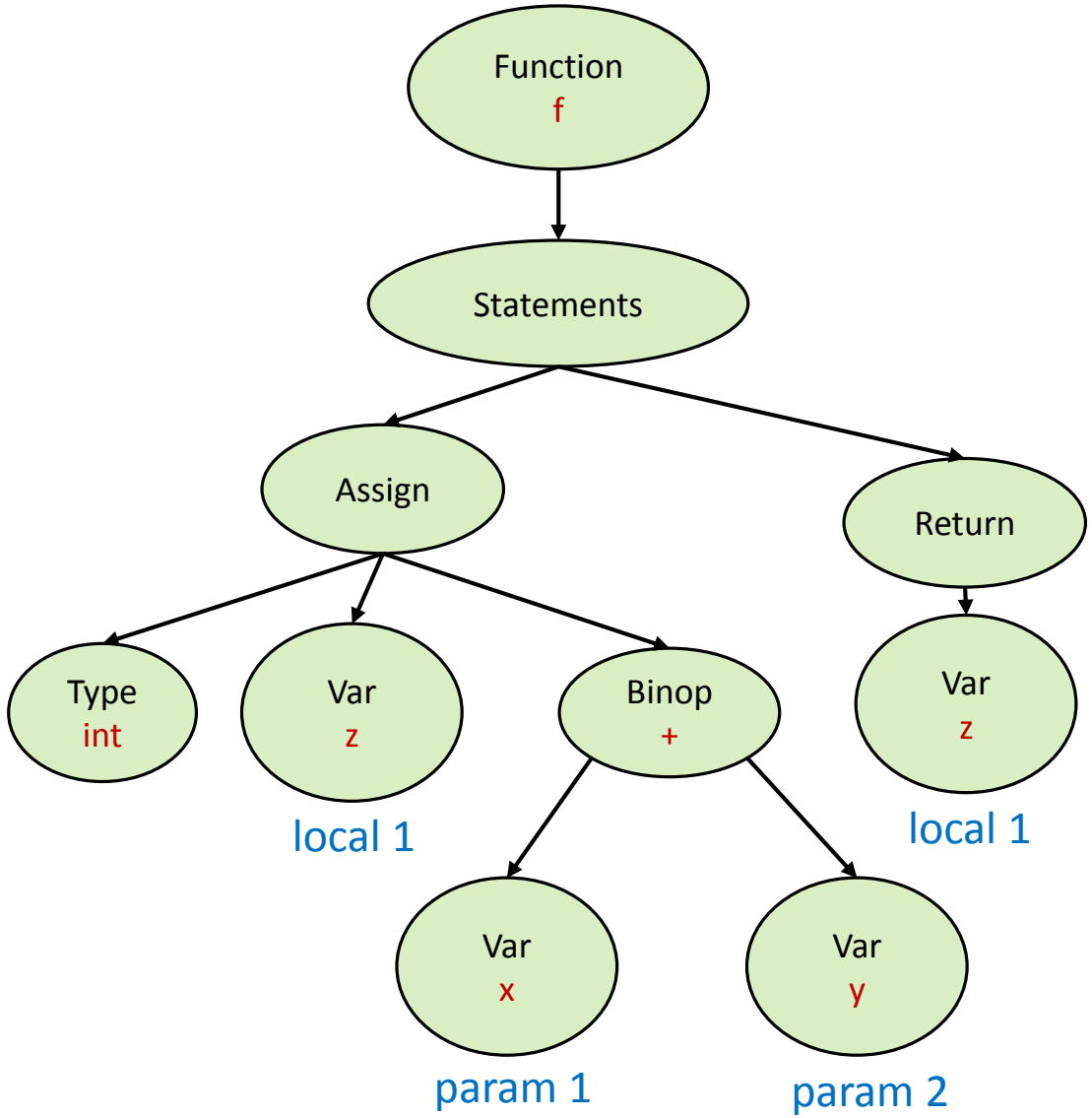
```
int f(int x, int y){
  int z = x + y;
  return z;
}
```

ID	Type	Kind
f	…	function

$scope_1$

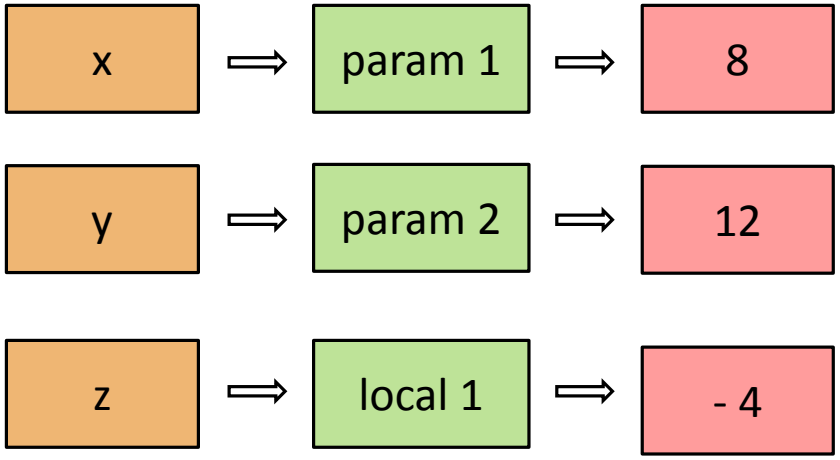
ID	Type	Kind
x	int	variable
y	int	variable
z	int	variable

$scope_2$



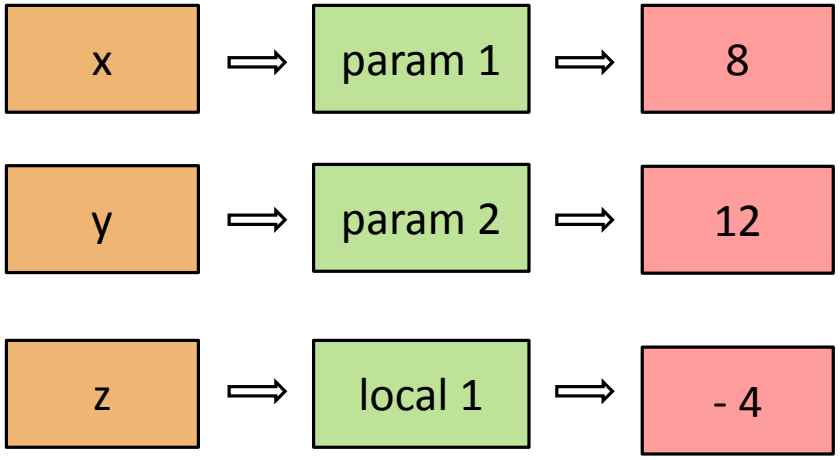
Variable Offsets

```
int f(int x, int y){
   int z = x + y;
   return z;
}
```



Variable Offsets

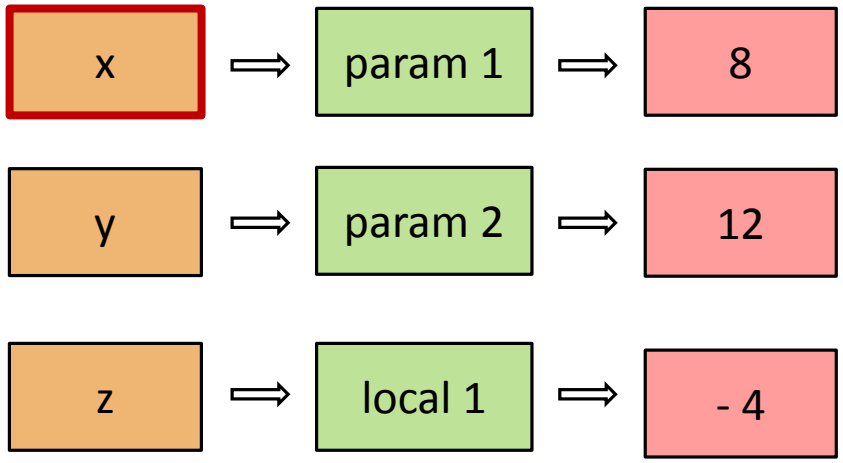
```
int f(int x, int y){
   int z = x + y;
   return z;
}
```




```
f:
...
lw $t0, 8($fp)
lw $t1, 12($fp)
add $t2, $t0, $t1
sw $t2, -4($fp)
...
```

Variable Offsets

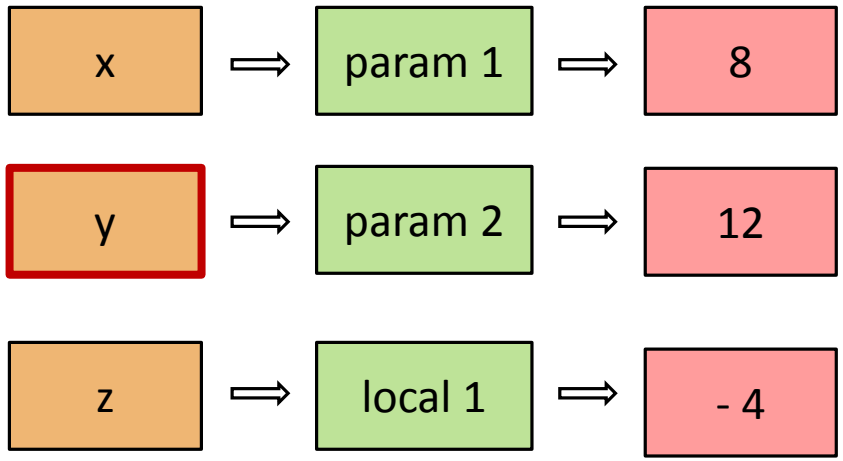
```
int f(int x, int y){
   int z = x + y;
   return z;
}
```




```
f:
...
lw $t0, 8($fp)
lw $t1, 12($fp)
add $t2, $t0, $t1
sw $t2, -4($fp)
...
```

Variable Offsets

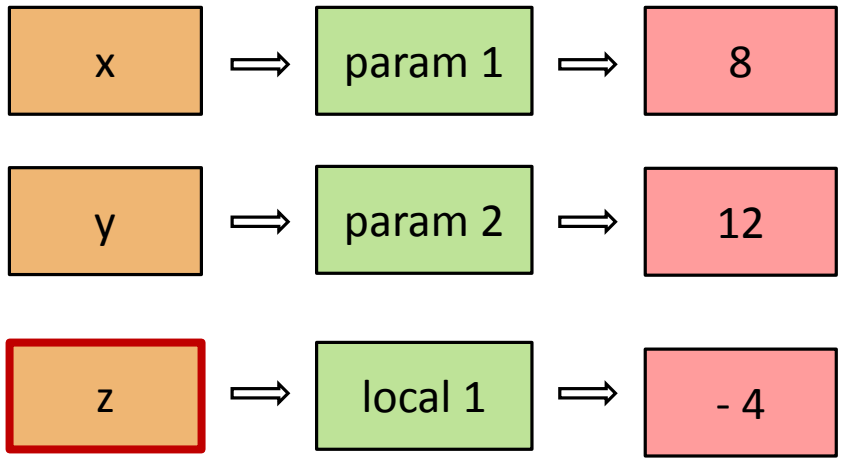
```
int f(int x, int y){
   int z = x + y;
   return z;
}
```




```
f:
...
lw $t0, 8($fp)
lw $t1, 12($fp)
add $t2, $t0, $t1
sw $t2, -4($fp)
...
```

Variable Offsets

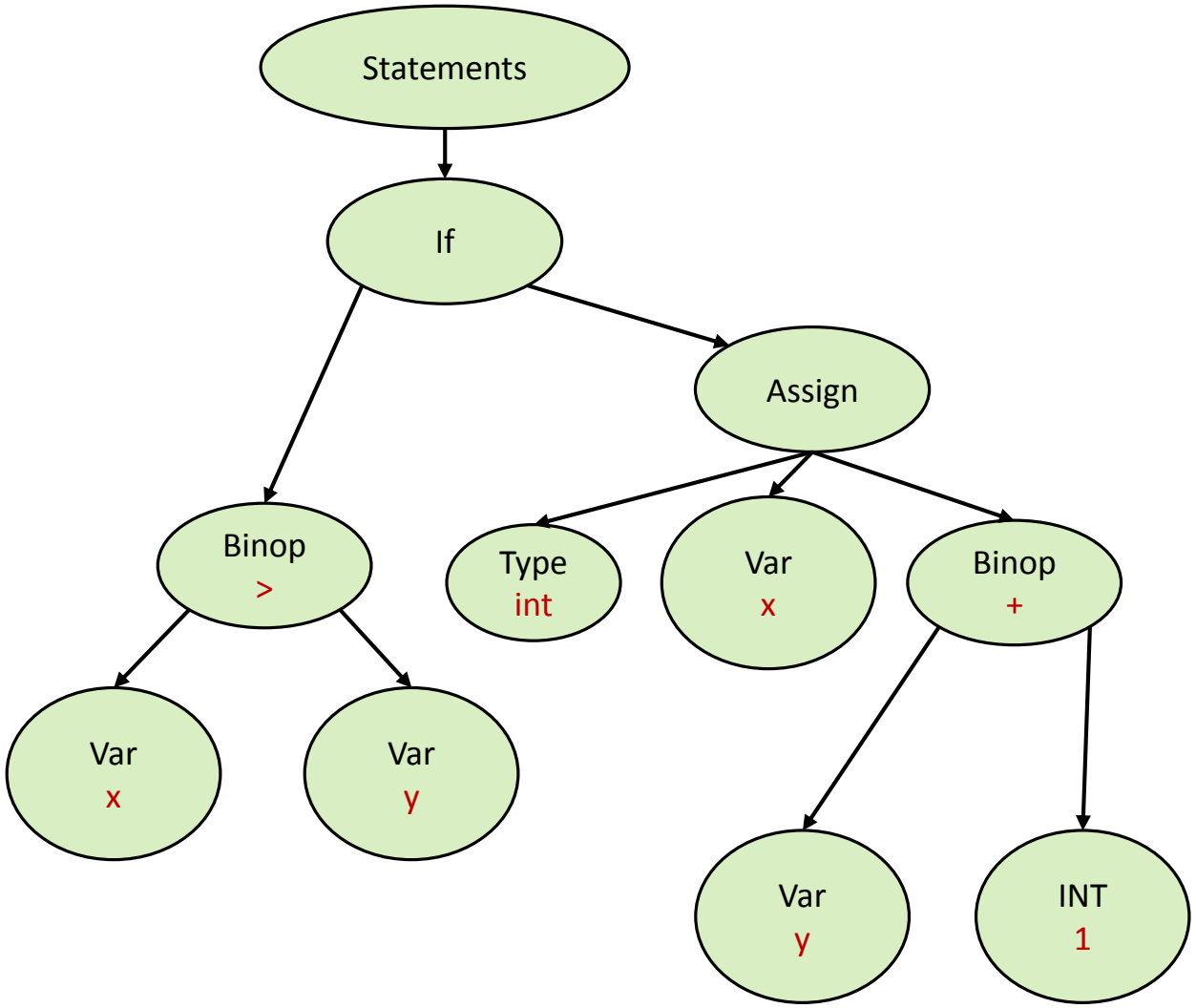
```
int f(int x, int y){
   int z = x + y;
   return z;
}
```




```
f:
...
lw $t0, 8($fp)
lw $t1, 12($fp)
add $t2, $t0, $t1
sw $t2, -4($fp)
...
```

Variable Offsets

```
void f(int x, int y){
  if (x > y) {
    int x = y + 1;
  }
}
```



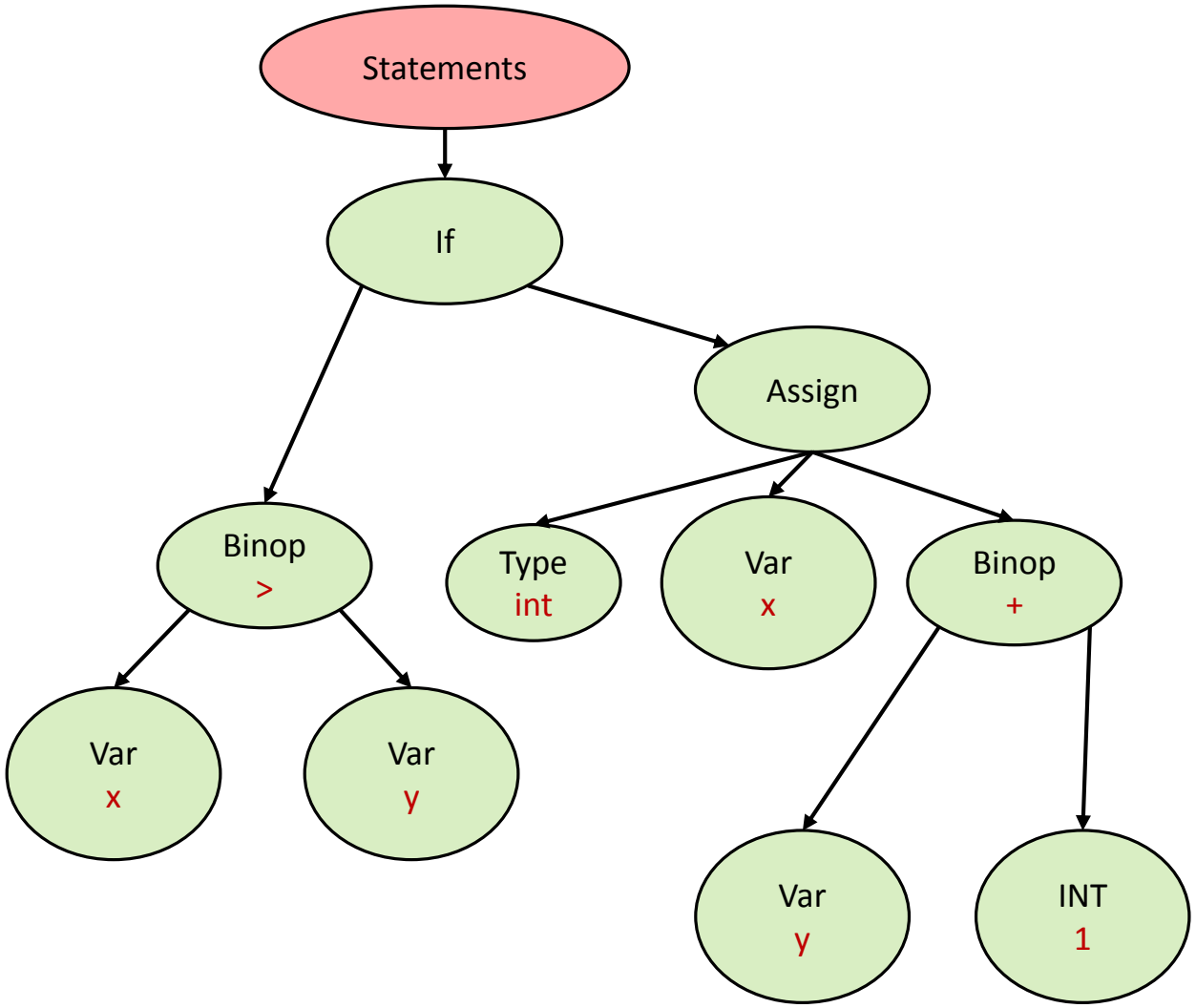
Variable Offsets

```
void f(int x, int y){
  if (x > y) {
    int x = y + 1;
  }
}
```

ID	Type	Kind
…	…	…

ID	Type	Kind
x	int	variable
y	int	variable

$scope_1$ $scope_2$



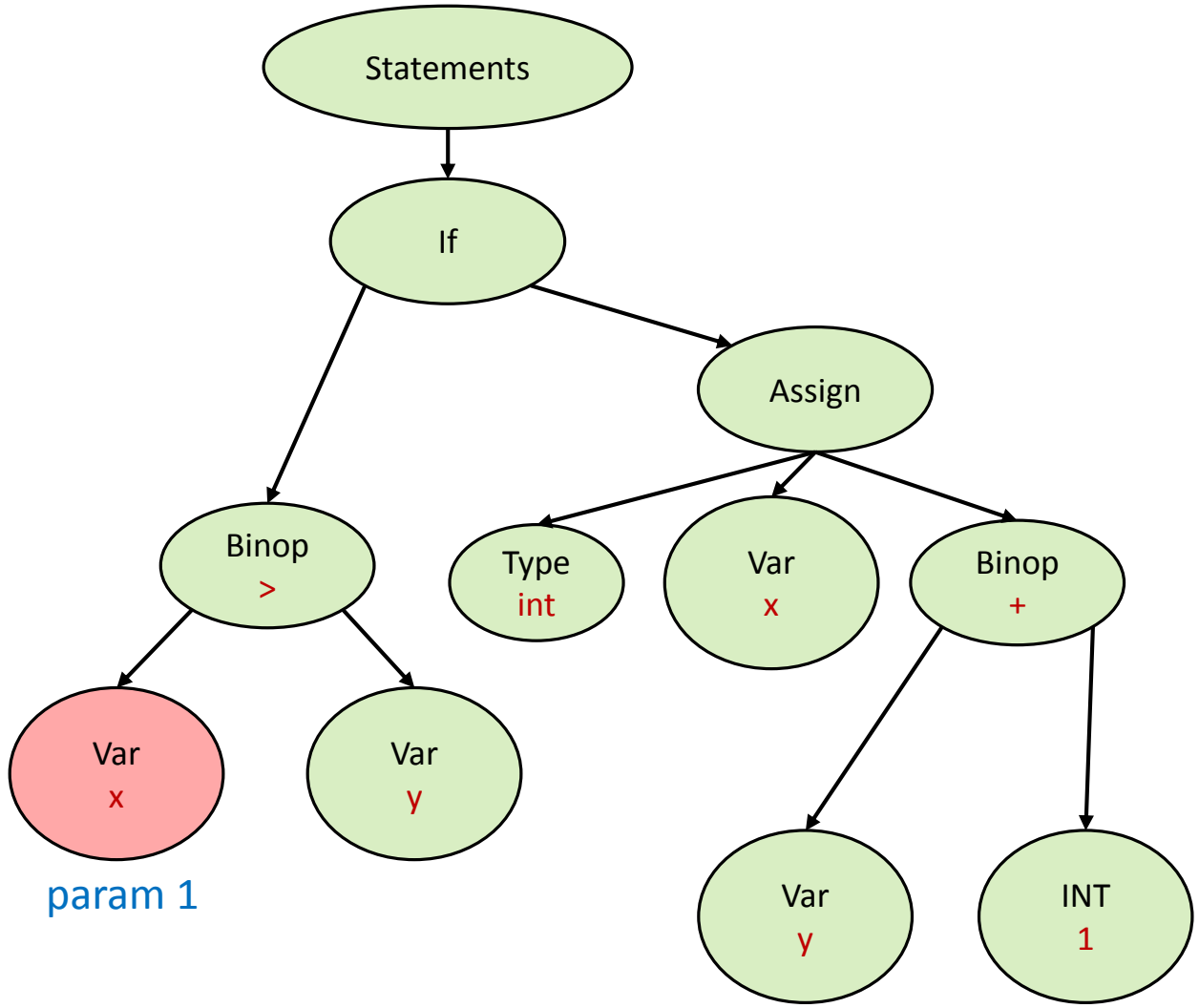
Variable Offsets

```
void f(int x, int y){
  if (x > y) {
    int x = y + 1;
  }
}
```

ID	Type	Kind
…	…	…

ID	Type	Kind
x	int	variable
y	int	variable

$scope_1$ $scope_2$



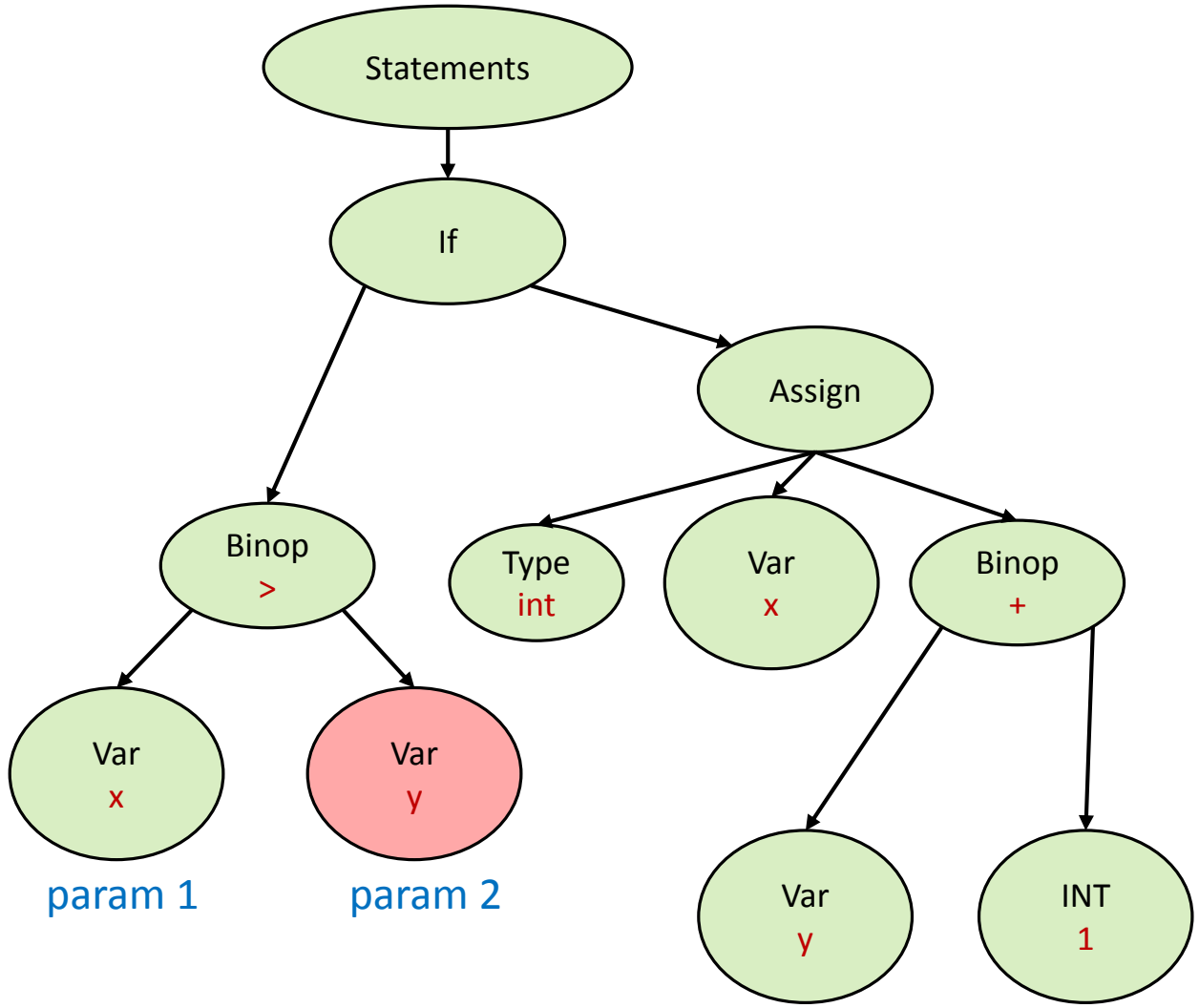
Variable Offsets

```
void f(int x, int y){
  if (x > y) {
    int x = y + 1;
  }
}
```

ID	Type	Kind
…	…	…

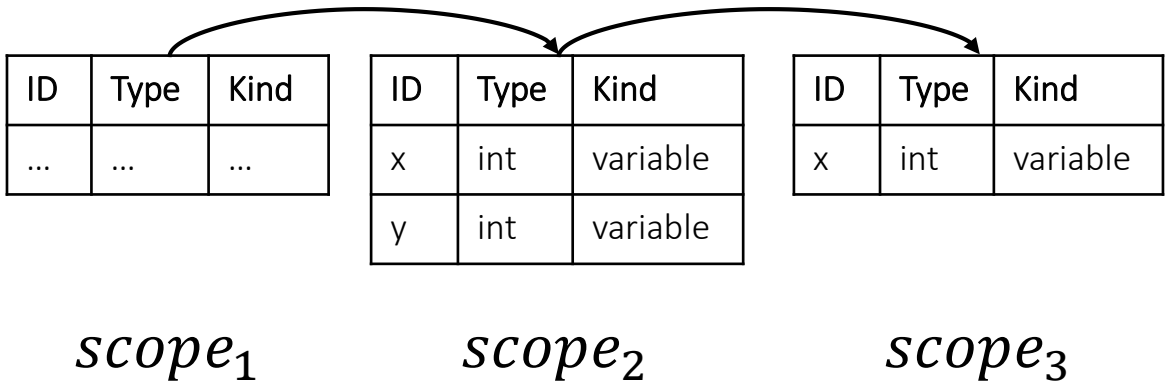
ID	Type	Kind
x	int	variable
y	int	variable

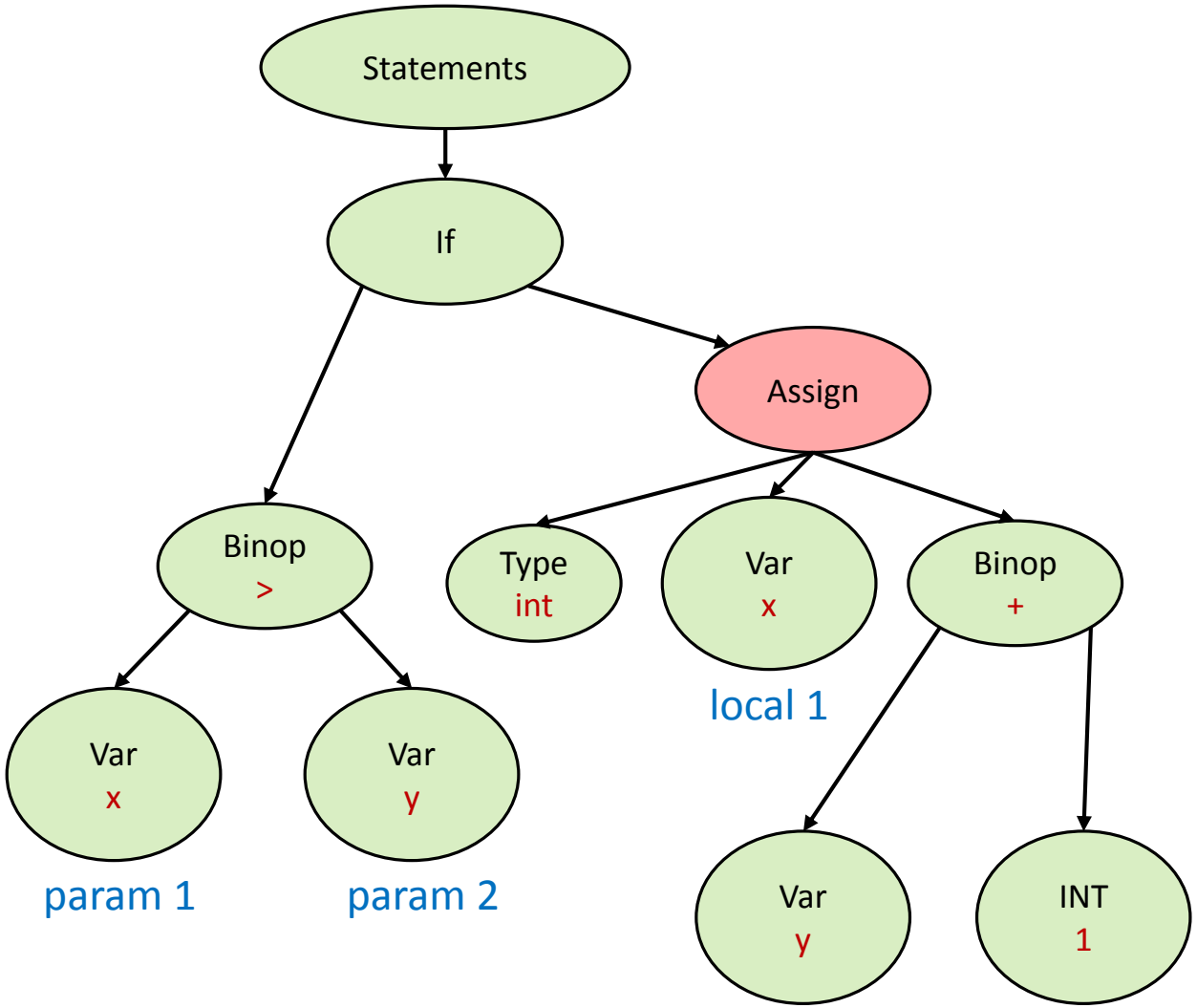
$scope_1$ $scope_2$



Variable Offsets

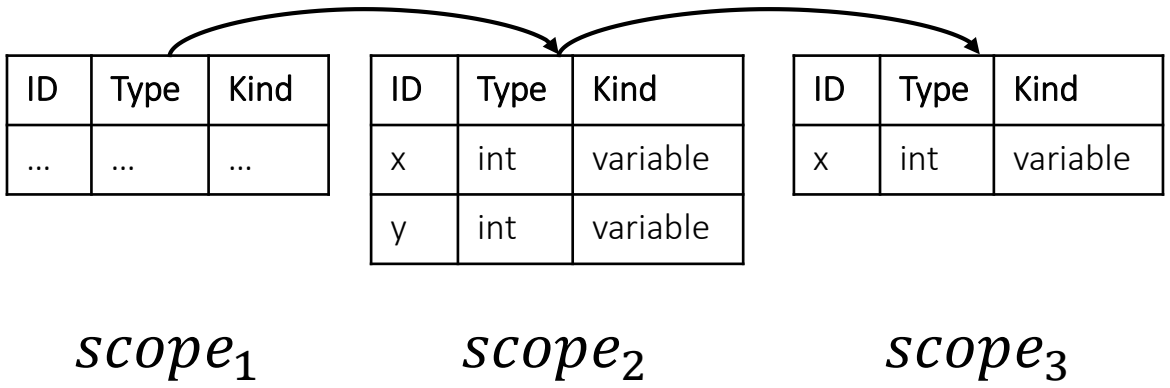
```
void f(int x, int y){
  if (x > y) {
    int x = y + 1;
  }
}
```

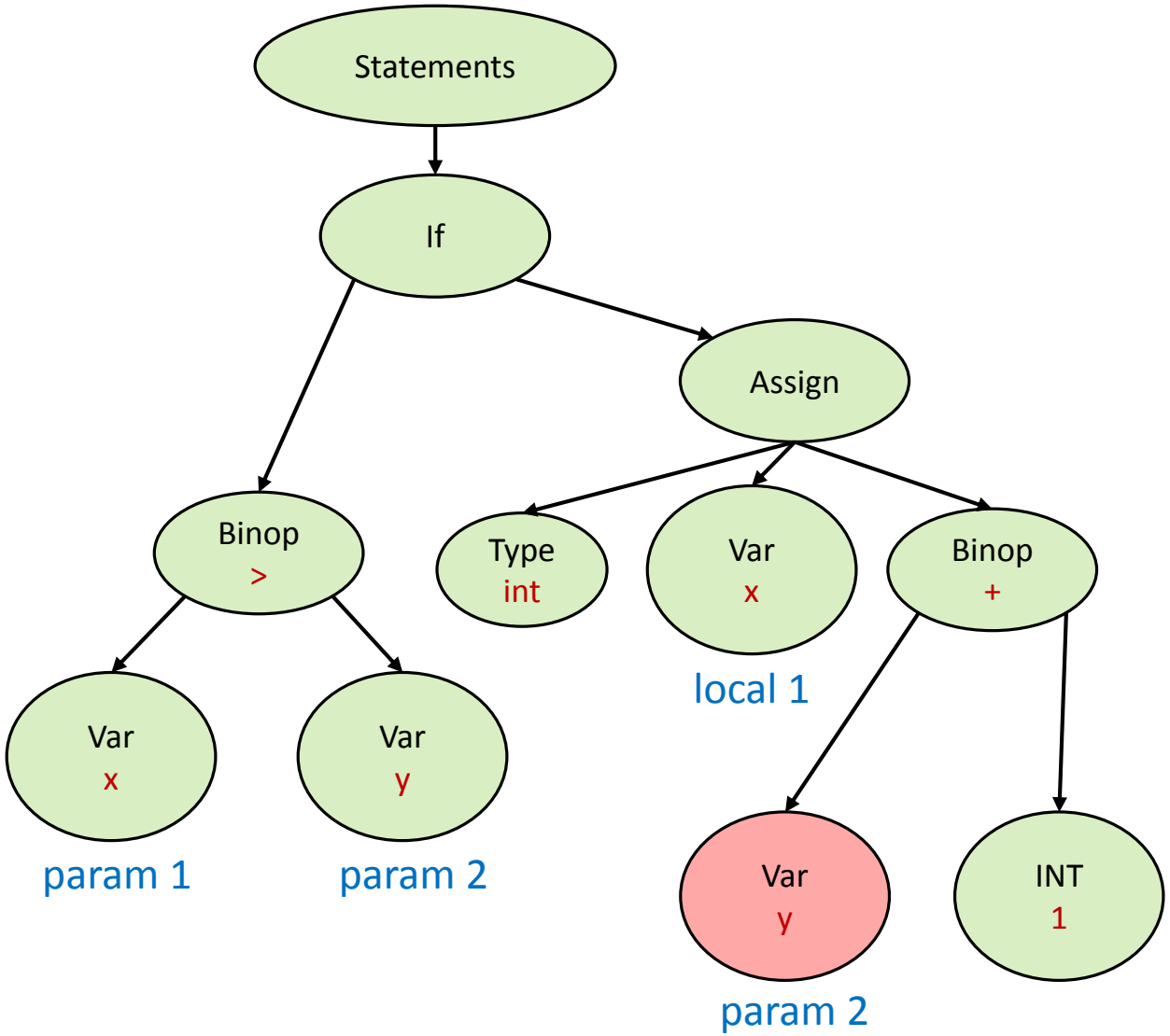




Variable Offsets

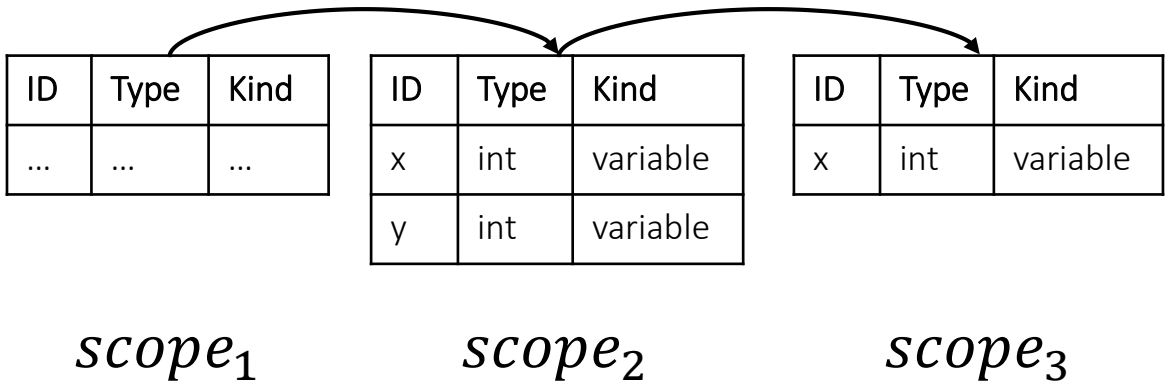
```
void f(int x, int y){
  if (x > y) {
    int x = y + 1;
  }
}
```

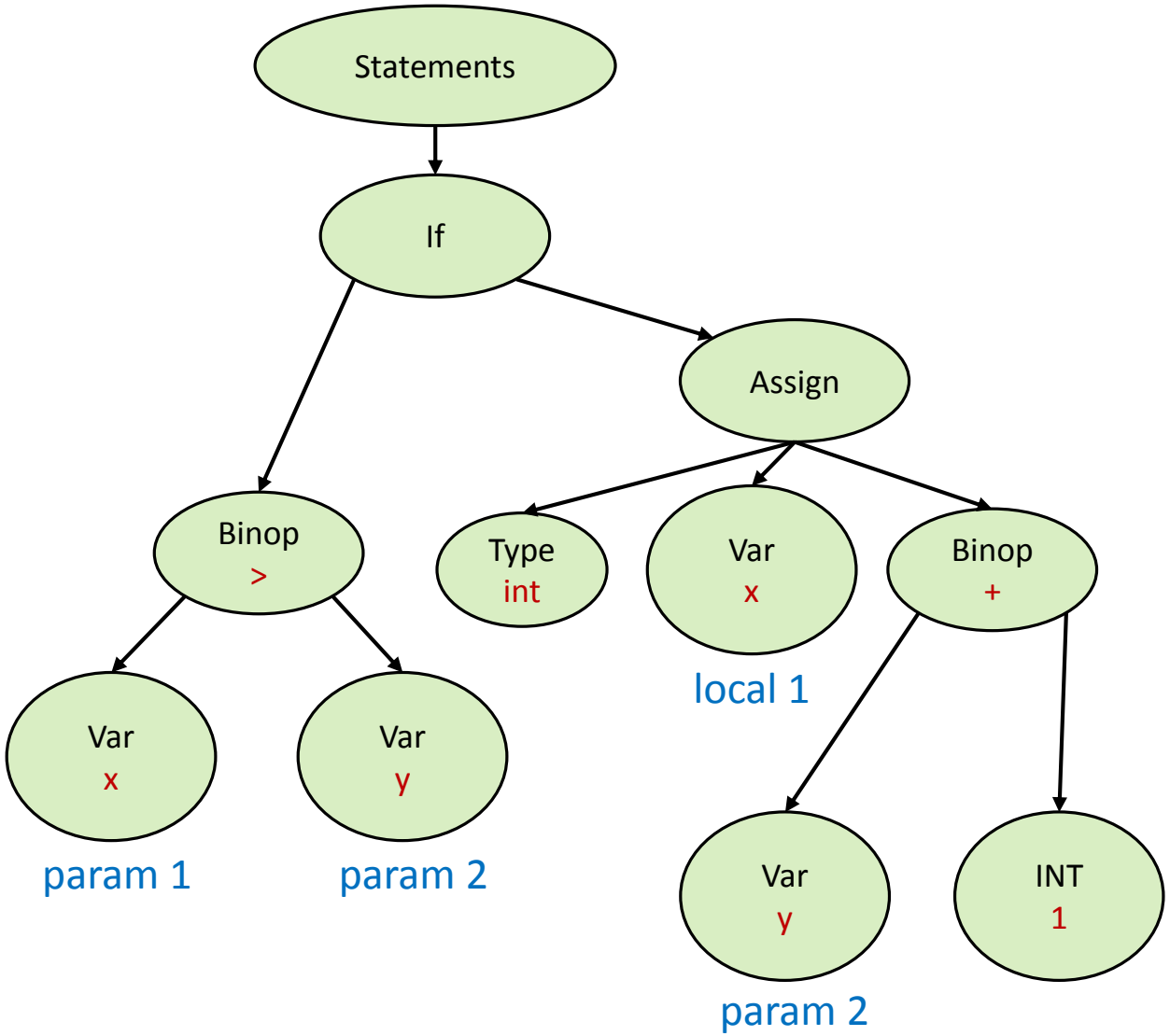




Variable Offsets

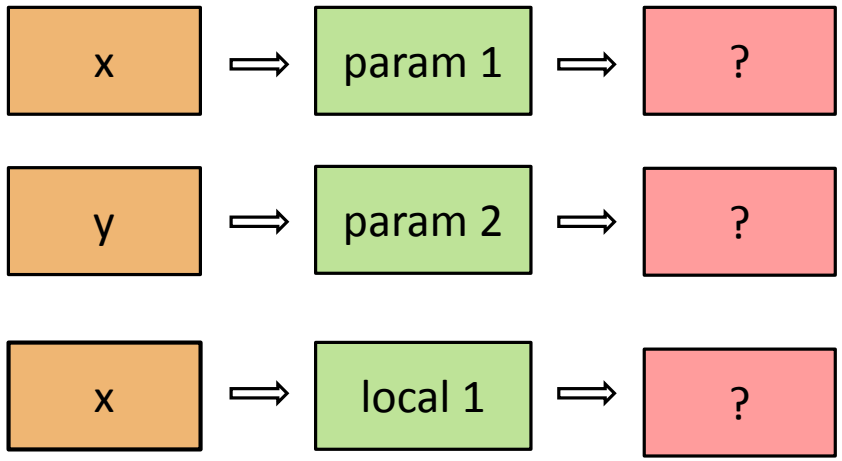
```
void f(int x, int y){
  if (x > y) {
    int x = y + 1;
  }
}
```

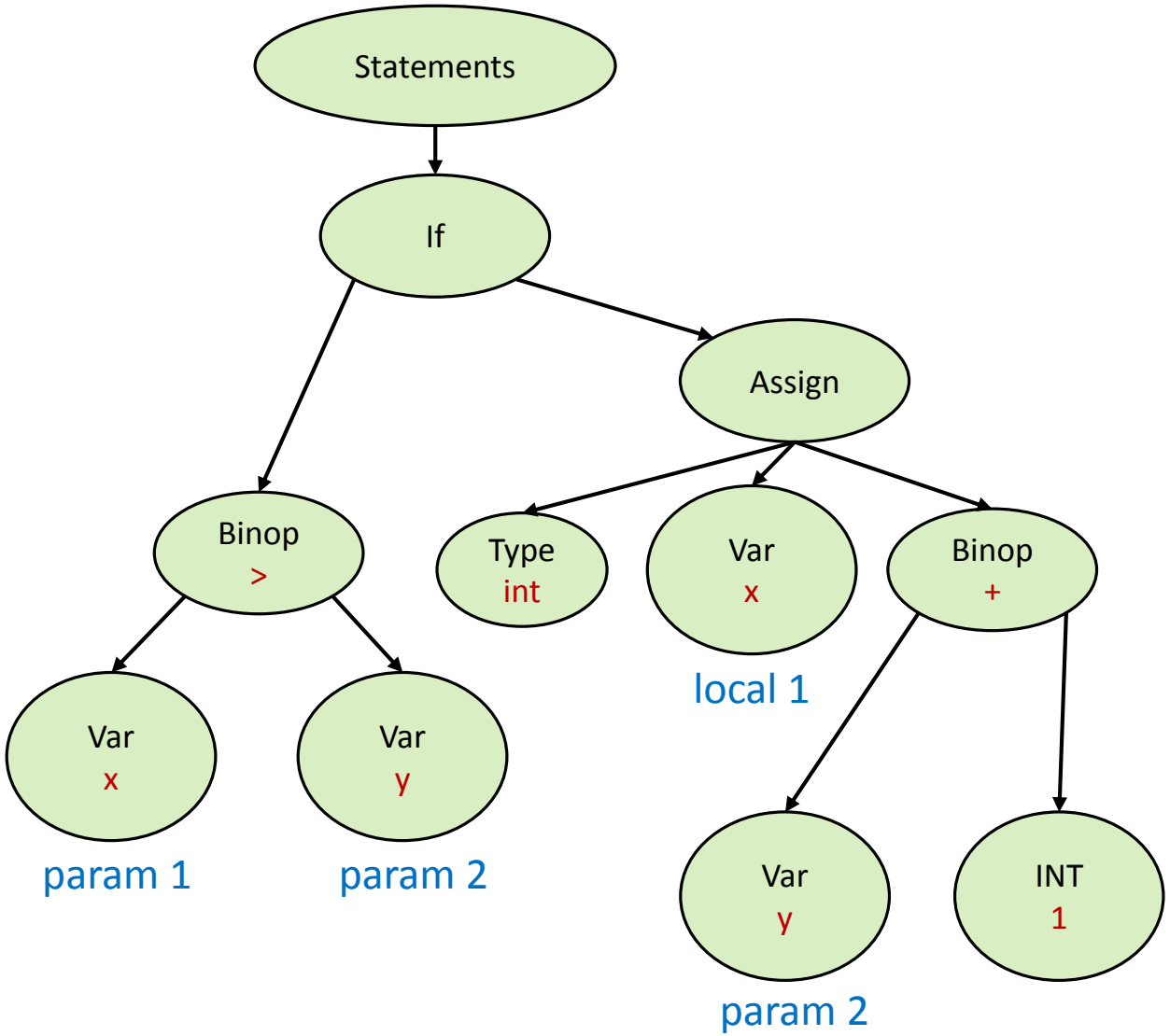




Variable Offsets

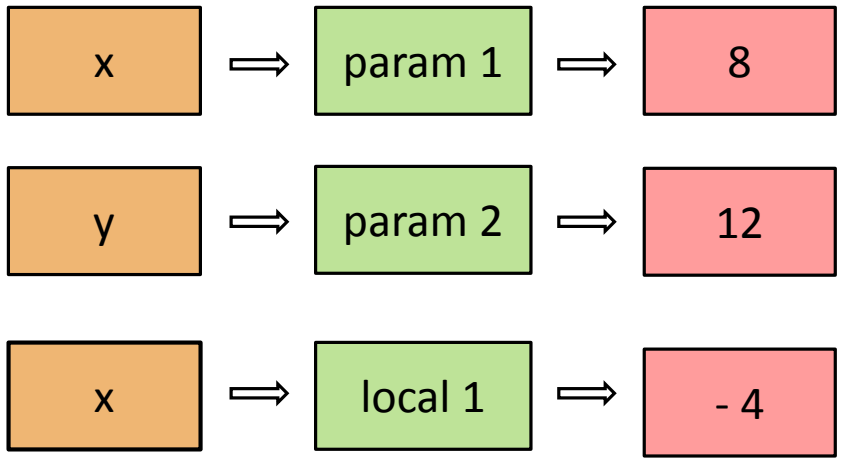
```
void f(int x, int y){
  if (x > y) {
    int x = y + 1;
  }
}
```

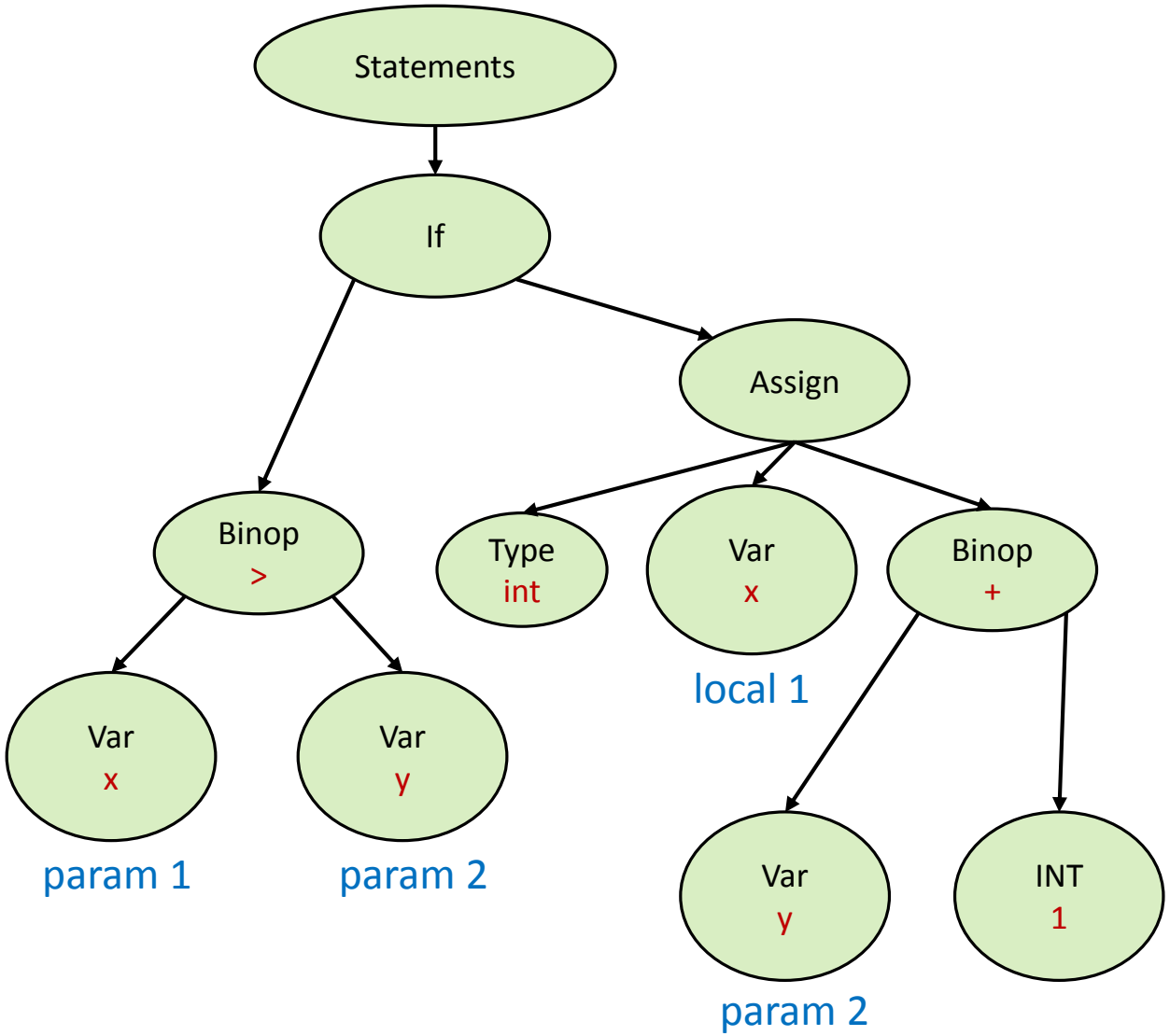




Variable Offsets

```
void f(int x, int y){
  if (x > y) {
    int x = y + 1;
  }
}
```





Storing Values

In our language, values can be stored in 32-bit registers:

- type **int** (4 bytes)
- type **string** (4 bytes pointer)
- arrays (4 bytes pointer)
- classes (4 bytes pointer)

Field Offsets

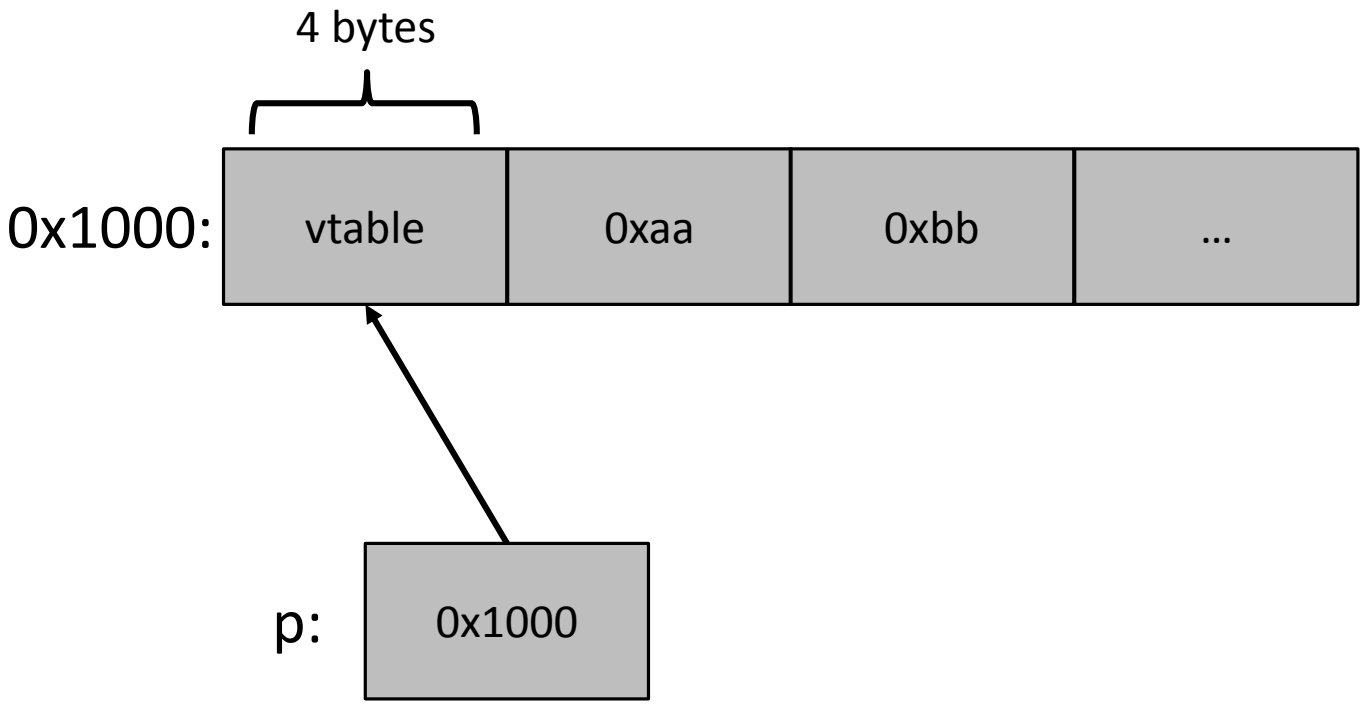
How does an object of this class look in memory?

```
class Point {
  int x;
  int y;
}
void f(Point p) {
  p.x = 0xaa;
  p.y = 0xbb;
}
```

Field Offsets

How does an object of this class look in memory?

```
class Point {
  int x;
  int y;
}
void f(Point p) {
  p.x = 0xaa;
  p.y = 0xbb;
}
```



Field Offsets

How does an object of this class look in memory?

```
class Point {
  int x;
  int y;
}
void f(Point p) {
  p.x = 0xaa;
  p.y = 0xbb;
}
```

```
f:
<prologue>
lw $t0, 8($fp)
li $t1, 0xaa
sw $t1, 4($t0)
li $t1, 0xbb
sw $t1, 8($t0)
<epilogue>
```

Field Offsets

How does an object of this class look in memory?

```
class Point {
  int x;
  int y;
}
void f(Point p) {
  p.x = 0xaa;
  p.y = 0xbb;
}
```

```
f:
<prologue>
lw $t0, 8($fp)
li $t1, 0xaa
sw $t1, 4($t0)
li $t1, 0xbb
sw $t1, 8($t0)
<epilogue>
```

Field Offsets

How does an object of this class look in memory?

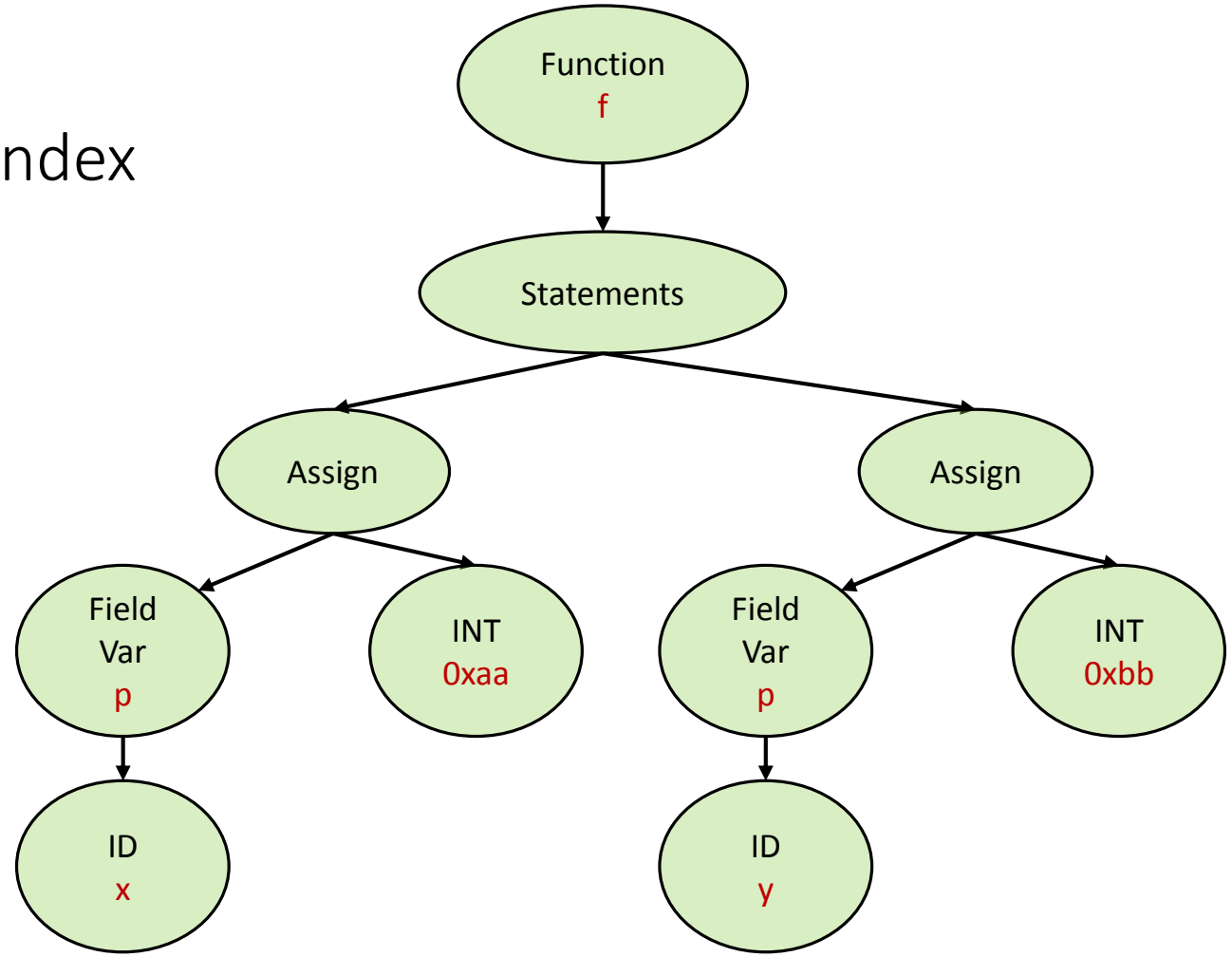
```
class Point {
  int x;
  int y;
}
void f(Point p) {
  p.x = 0xaa;
  p.y = 0xbb;
}
```

```
f:
<prologue>
lw $t0, 8($fp)
li $t1, 0xaa
sw $t1, 4($t0)
li $t1, 0xbb
sw $t1, 8($t0)
<epilogue>
```

Field Offsets

Each class field should have an index

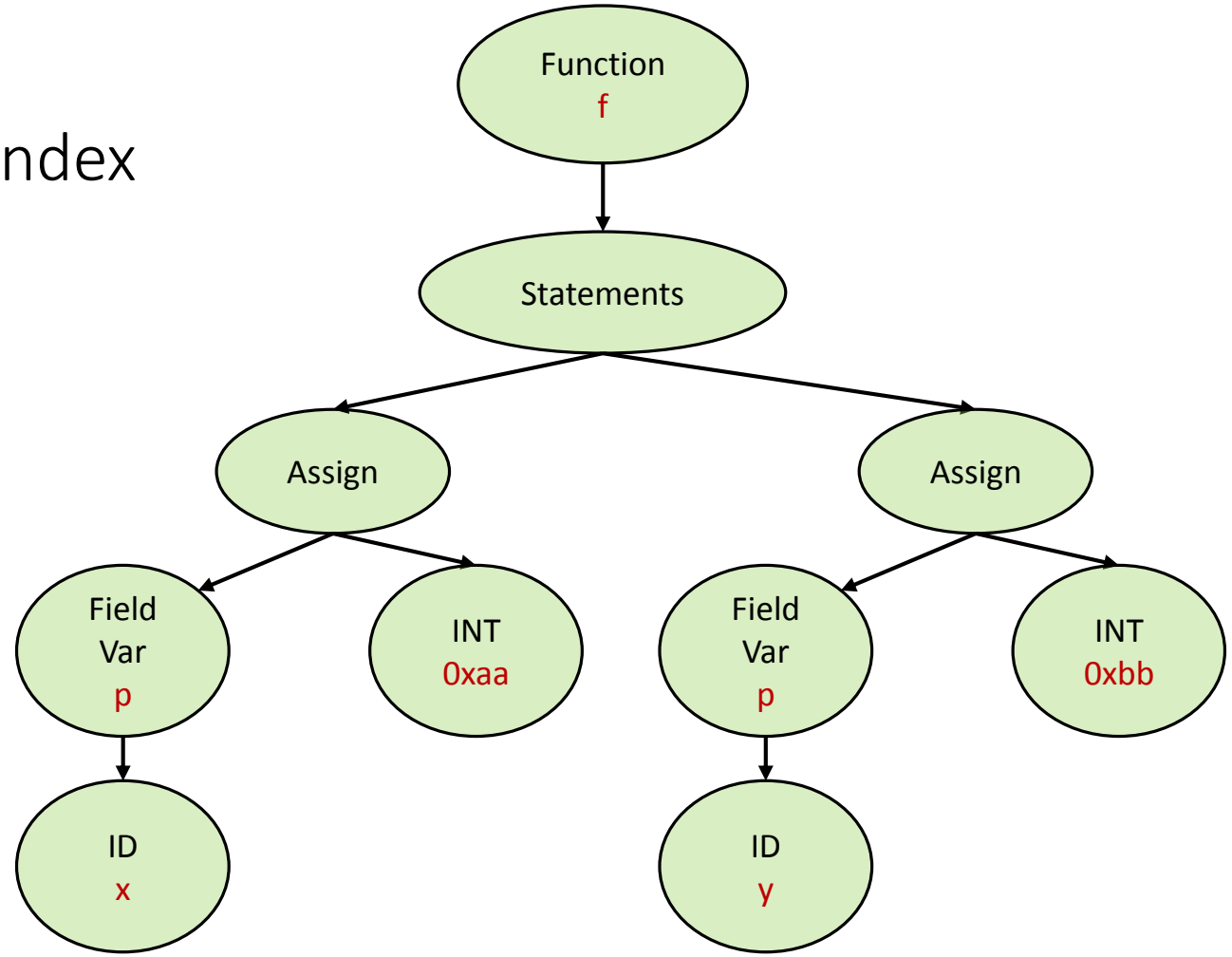
```
class Point {
  int x;
  int y;
}
void f(Point p) {
  p.x = 0xaa;
  p.y = 0xbb;
}
```



Field Offsets

Each class field should have an index

```
class Point {
  int x;
  int y;
}
void f(Point p) {
  p.x = 0xaa;
  p.y = 0xbb;
}
```



ID	Type	Kind
…	…	…

$scope_1$

ID	Type	Kind
p	Point	variable

$scope_2$

Field Offsets

Each class field should have an index

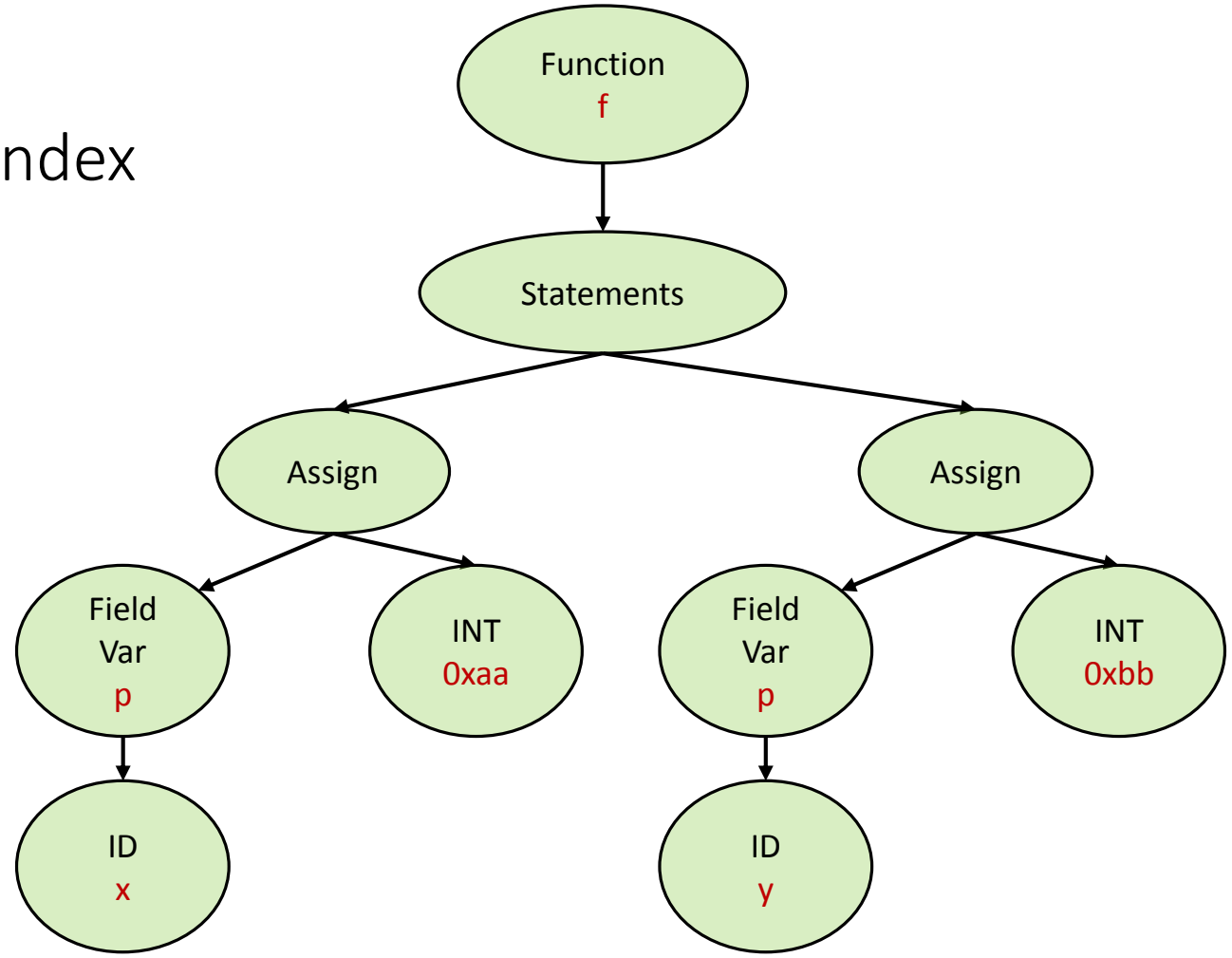
```
class Point {
  int x;
  int y;
}
void f(Point p) {
  p.x = 0xaa;
  p.y = 0xbb;
}
```

ID	Type	Kind
…	…	…

$scope_1$

ID	Type	Kind
p	Point	variable

$scope_2$



Type Sizes

```
class Point {
  int x;
  int y;
}
void f() {
  Point p = new Point;
}
```

```
f:
<prologue>
li $t0, 12
subu $sp, $sp, 4
sw $t0, 0($sp)
jal malloc
addu $sp, $sp, 4
sw $v0, -4($fp)
<epilogue>
```

Type Sizes

```
class Point {
  int x;
  int y;
}
void f() {
  Point p = new Point;
}
```

```
f:
<prologue>
li $t0, 12
subu $sp, $sp, 4
sw $t0, 0($sp)
jal malloc
addu $sp, $sp, 4
sw $v0, -4($fp)
<epilogue>
```

Type Sizes

```
class A {
  int a;
  string name;
}
class B extends A {
  A object;
}
void f() {
  B b = new B;
}
```

```
f:
<prologue>
li $t0, ?
subu $sp, $sp, 4
sw $t0, 0($sp)
jal malloc
addu $sp, $sp, 4
sw $v0, -4($fp)
<epilogue>
```

Type Sizes

```
class A {
  int a;
  string name;
}
class B extends A {
  A object;
}
void f() {
  B b = new B;
}
```

```
f:
<prologue>
li $t0, 16
subu $sp, $sp, 4
sw $t0, 0($sp)
jal malloc
addu $sp, $sp, 4
sw $v0, -4($fp)
<epilogue>
```