# 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
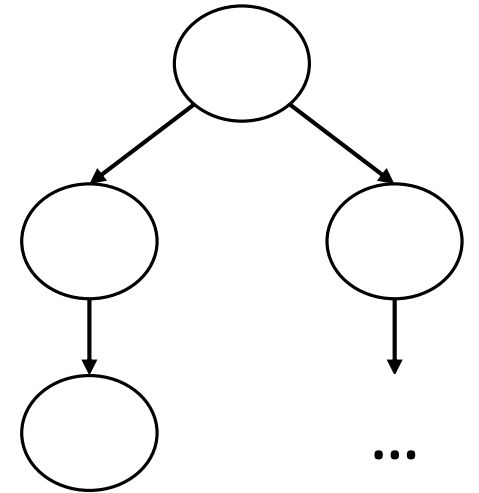
$visit(node)$:
    // do something with node

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

# Visitor Design Pattern: Example

Printing the AST

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

# Symbol Table

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

# Symbol Table

main scope

top of stack

| ID | Type | Kind |
|------|-----------|----------|
| main | int, void | function |
| msg | string | variable |
| | | |

| ID | Type | Kind |
|------|------|----------|
| x | int | variable |
| y | int | variable |
| | | |

. . .

| ID | Type | Kind |
|--------|--------|----------|
| result | int | variable |
| y | string | variable |
| | | |

$scope_1$

$scope_2$

$scope_n$

# Symbol Table Operations

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

# Symbol Table: Insert

Example:
- Insert(z, int, variable)

main scope

| ID | Type | Kind |
|----|------|------|
| x | int | variable |
| y | int | variable |
| | | |

$scope_1$

| ID | Type | Kind |
|----|------|------|
| tmp | int | variable |
| | | |

$scope_2$

# Symbol Table: Insert

Example:
- Insert(z, int, variable)

main scope

| ID | Type | Kind |
|----|------|------|
| x | int | variable |
| y | int | variable |
| | | |

| ID | Type | Kind |
|----|------|------|
| tmp | int | variable |
| z | Int | variable |

$scope_1$                              $scope_2$

# 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 |
| | | |

| ID | Type | Kind |
|----|------|------|
| tmp | int | variable |
| | | |

$$scope_1 \qquad\qquad\qquad scope_2$$

# 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 |
|    |      |      |

| ID  | Type | Kind |
|-----|------|------|
| tmp | int  | variable |
|     |      |      |

| ID | Type | Kind |
|----|------|------|
|    |      |      |
|    |      |      |

$scope_1$      $scope_2$      $scope_3$

# 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";
  }
}
```
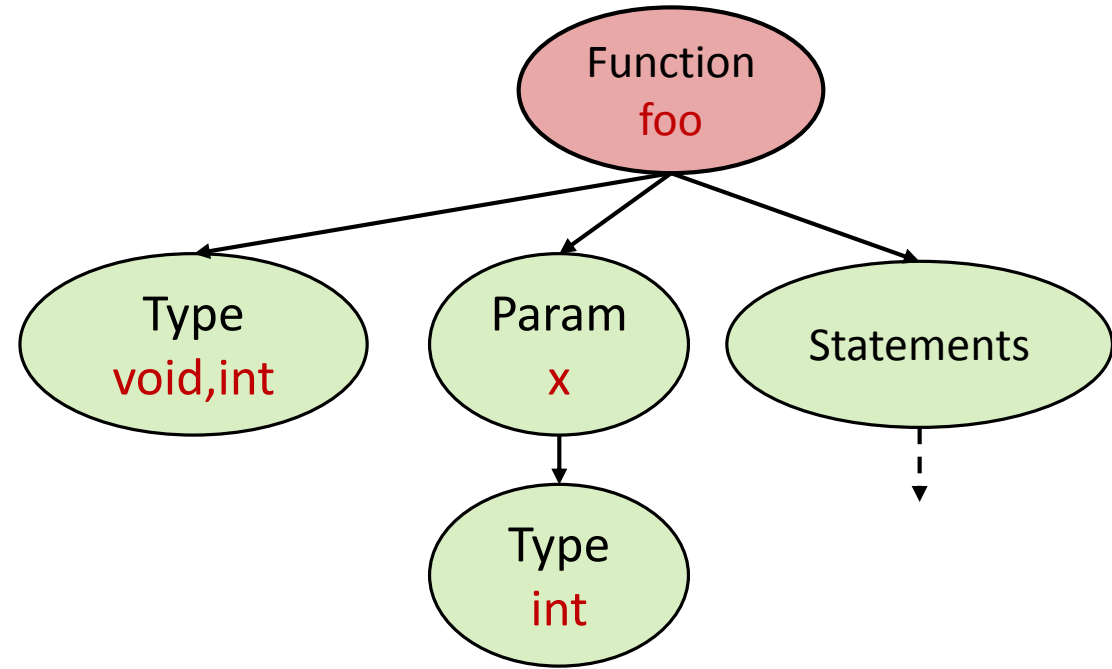
| ID  | Type     | Kind     |
|-----|----------|----------|
| foo | void,int | function |

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

Function
foo

Type
void,int

Param
x

Statements

Type
int

| 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 |
|----|------|------|
| | | |

Function
foo

Statements

If

Var
x

Assign

```
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";
  }
}
```
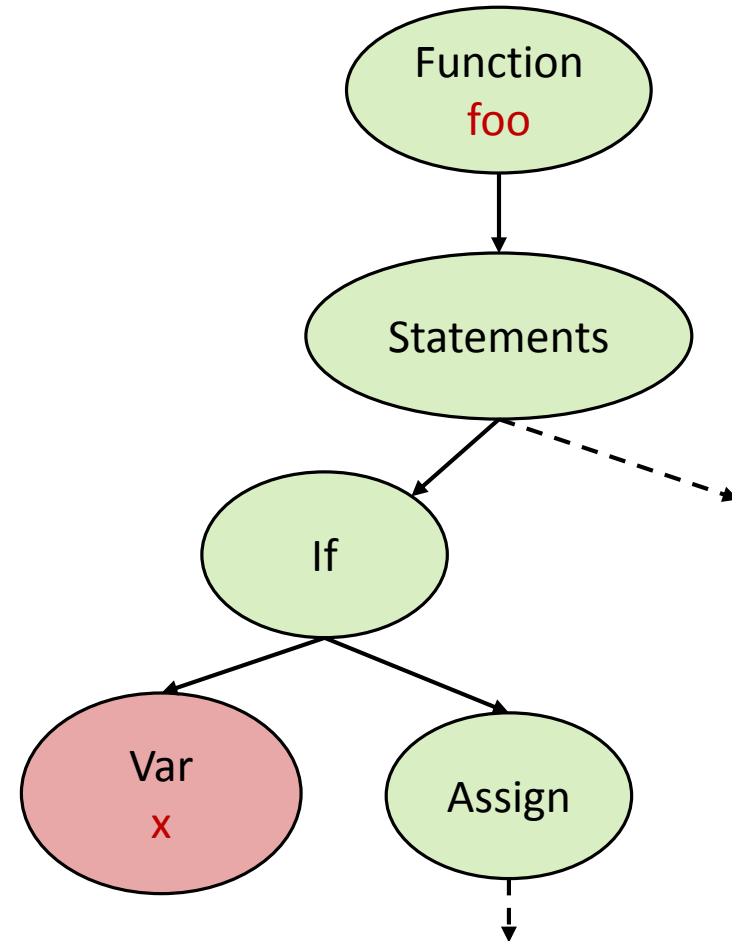
| ID | Type | Kind |
|----|------|------|
| foo | void,int | function |

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

| ID | Type | Kind |
|----|------|------|
| a | string | variable |
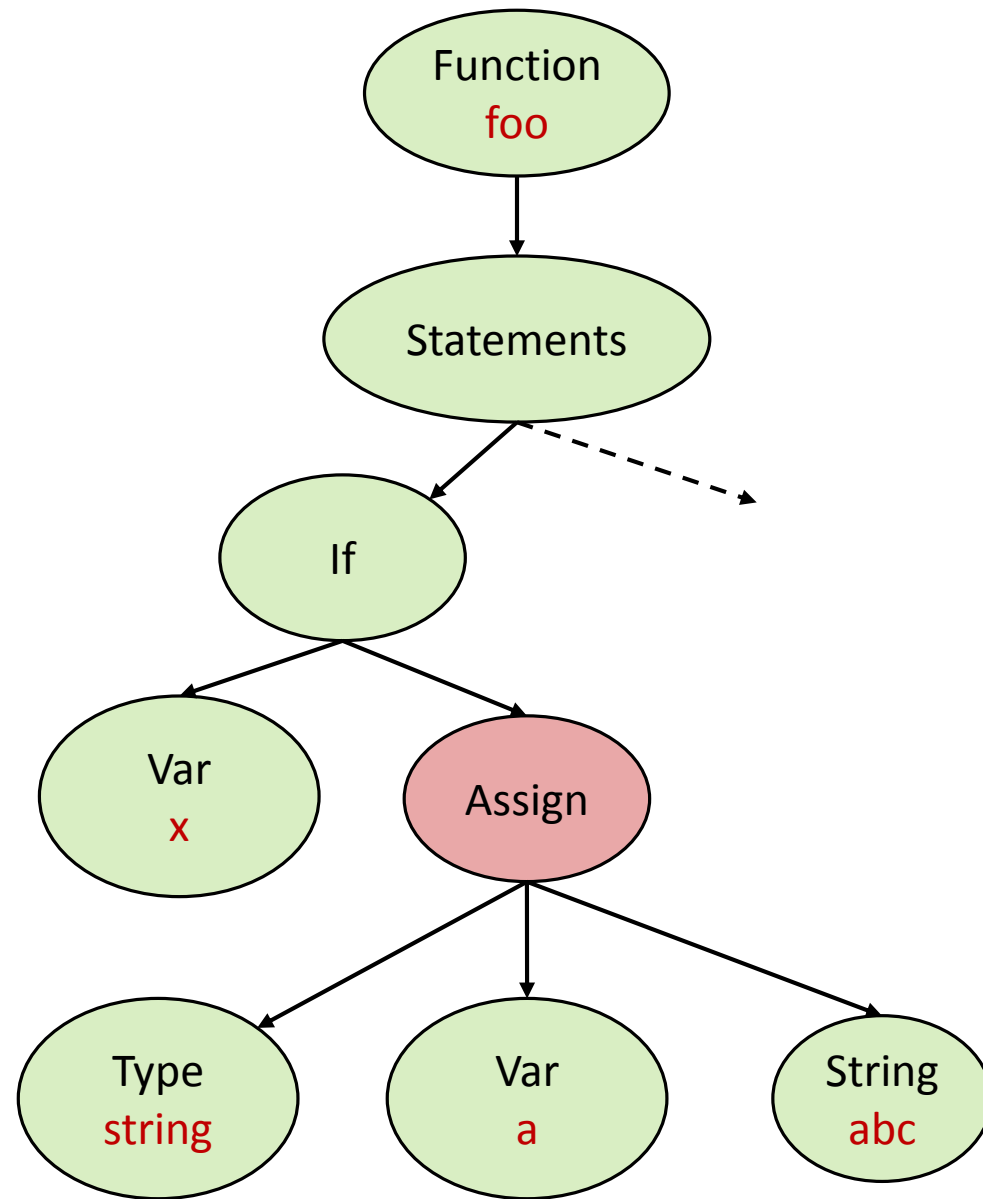
```
void foo(int x) {
    int a = 1;
    int b = 2;
    if (x) {
        string a = "abc";
    }
}
```

| 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";
    }
}
```

Function
foo

Statements
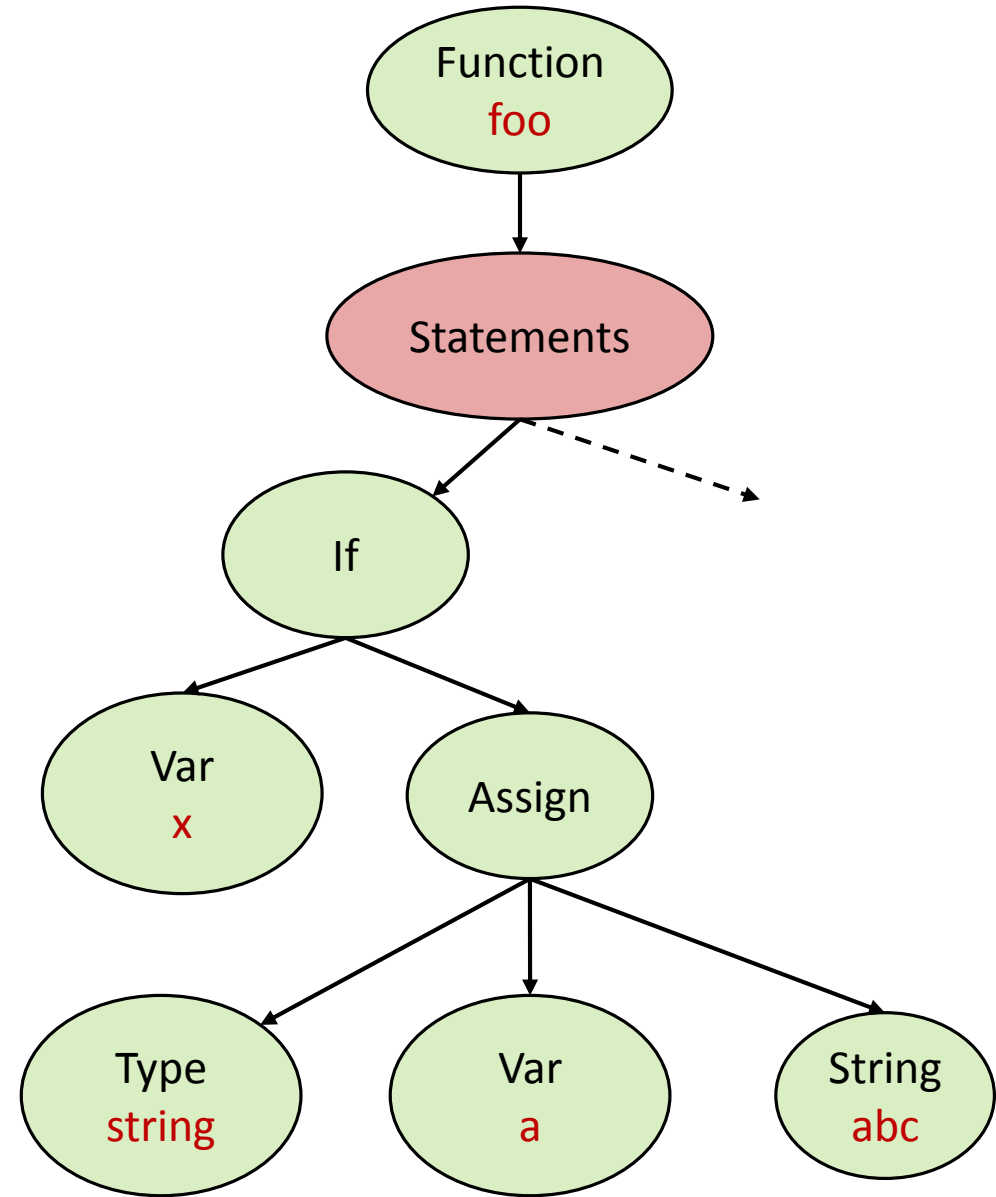
| ID  | Type     | Kind     |
|-----|----------|----------|
| foo | void,int | function |
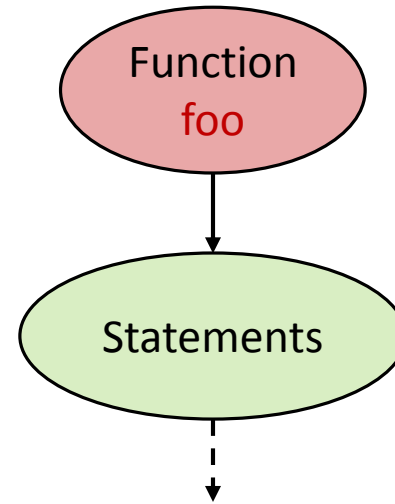
```
void foo(int x) {
   int a = 1;
   int b = 2;
   if (x) {
      string a = "abc";
   }
}
```

# Type Checking

Goals:
- Type correctness of expressions
- Compute type of expressions

Performed using:
- AST visitor
- Symbol table

# Type Checking

Basic algorithm:

$visit(node):$
       $t_1 = visit(node.child_1)$

       $...$

       $t_n = visit(node.child_n)$

       $return\ compute\_type(t_1, ..., t_n)$

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

**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 |

Assign

Type
int

Var
a

Binop
-

Binop
+

INT
7

int

Call
foo

Var
b

```
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 |

Assign

Type
int

Var
a

Binop
-

int — Binop
+

INT
7

int — Call
foo

int — Var
b

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

| ID | Type | Kind |
|----|------|------|
| foo | int,void | function |
| b | int | variable |

Assign

Type
int

Var
a

Binop
-

int

Binop
+

int

INT
7

int

Call
foo

int

Var
b

```
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;
}
```



Function
main

Statements

Assign

Type
int

Var
j
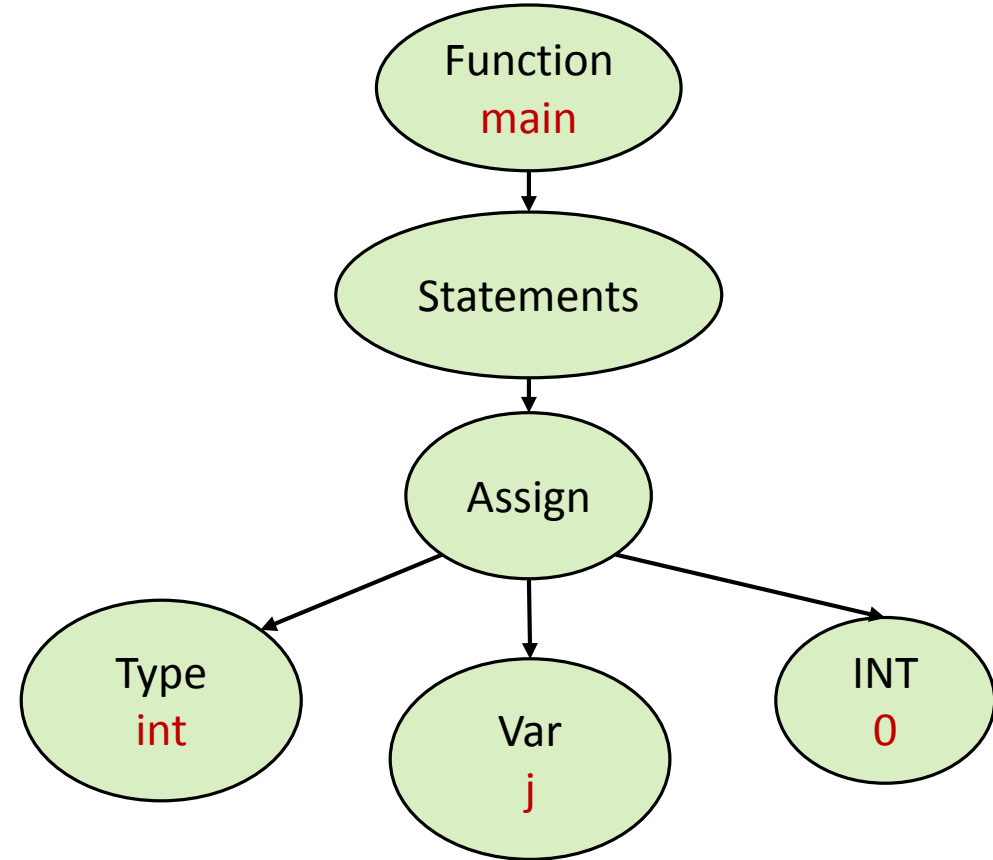
INT
0
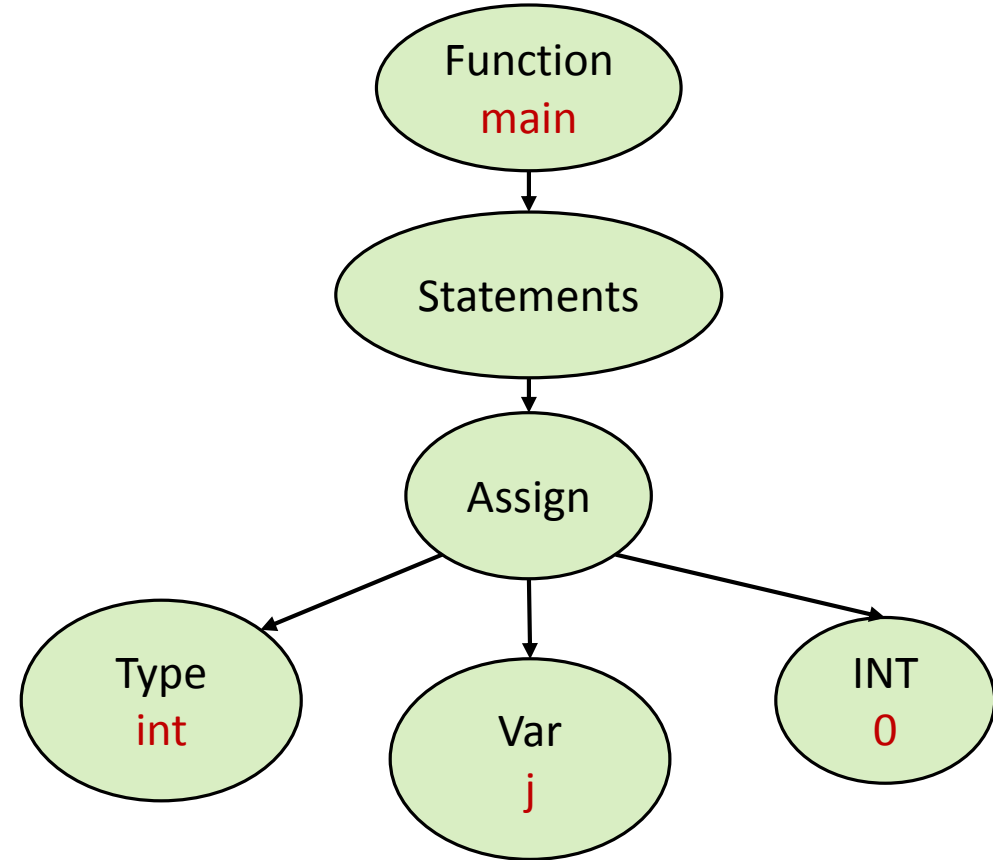
## Valid

# Assignments

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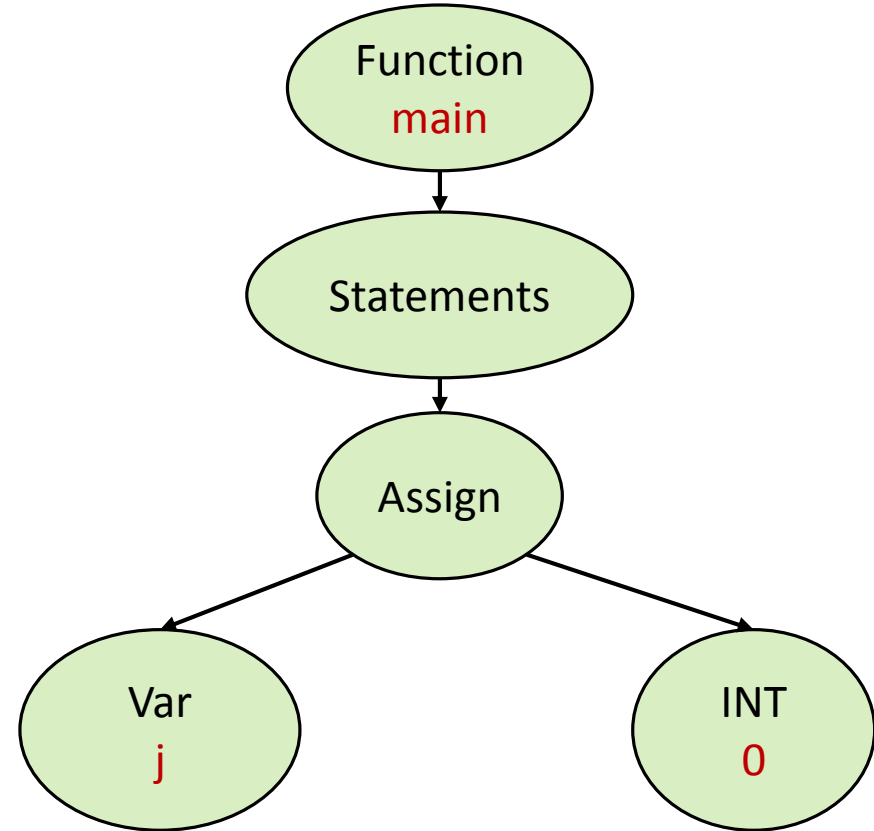
# Assignments

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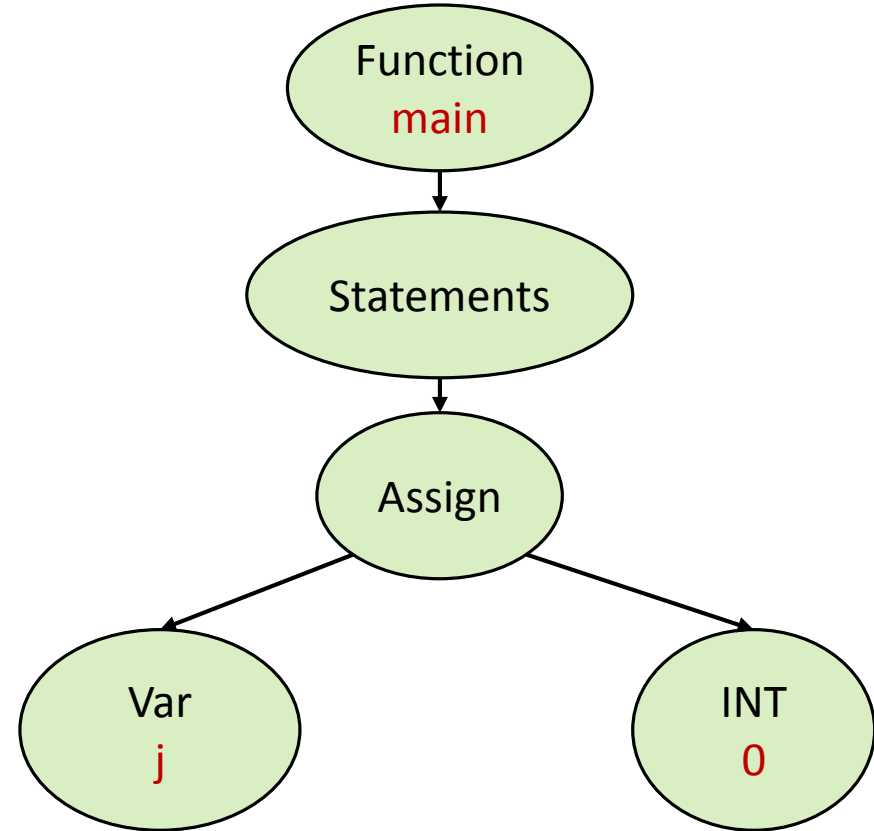


Invalid

# Assignments

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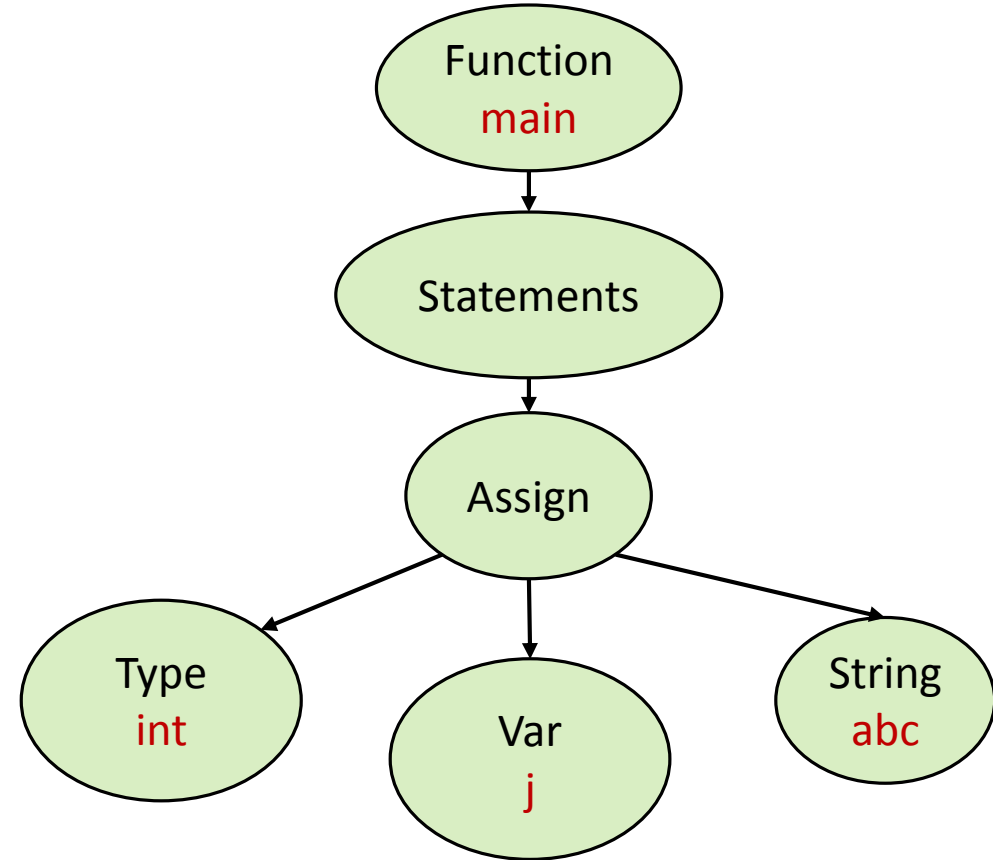
# Assignments

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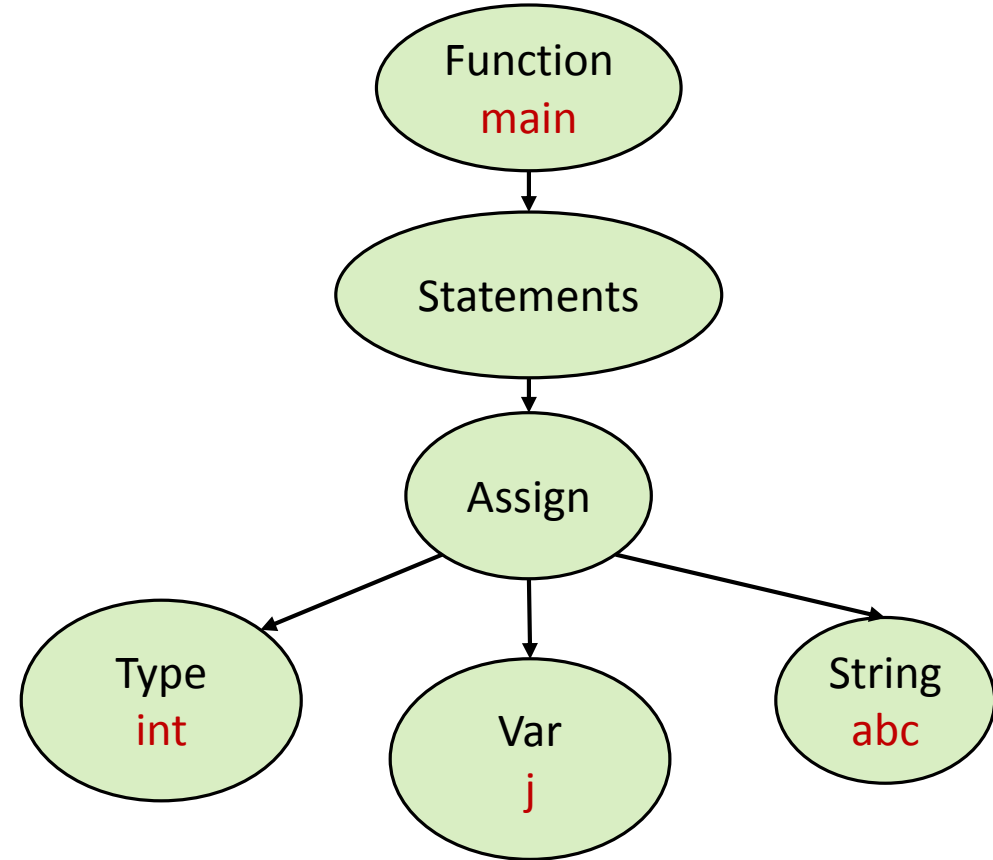


Invalid

# Assignments

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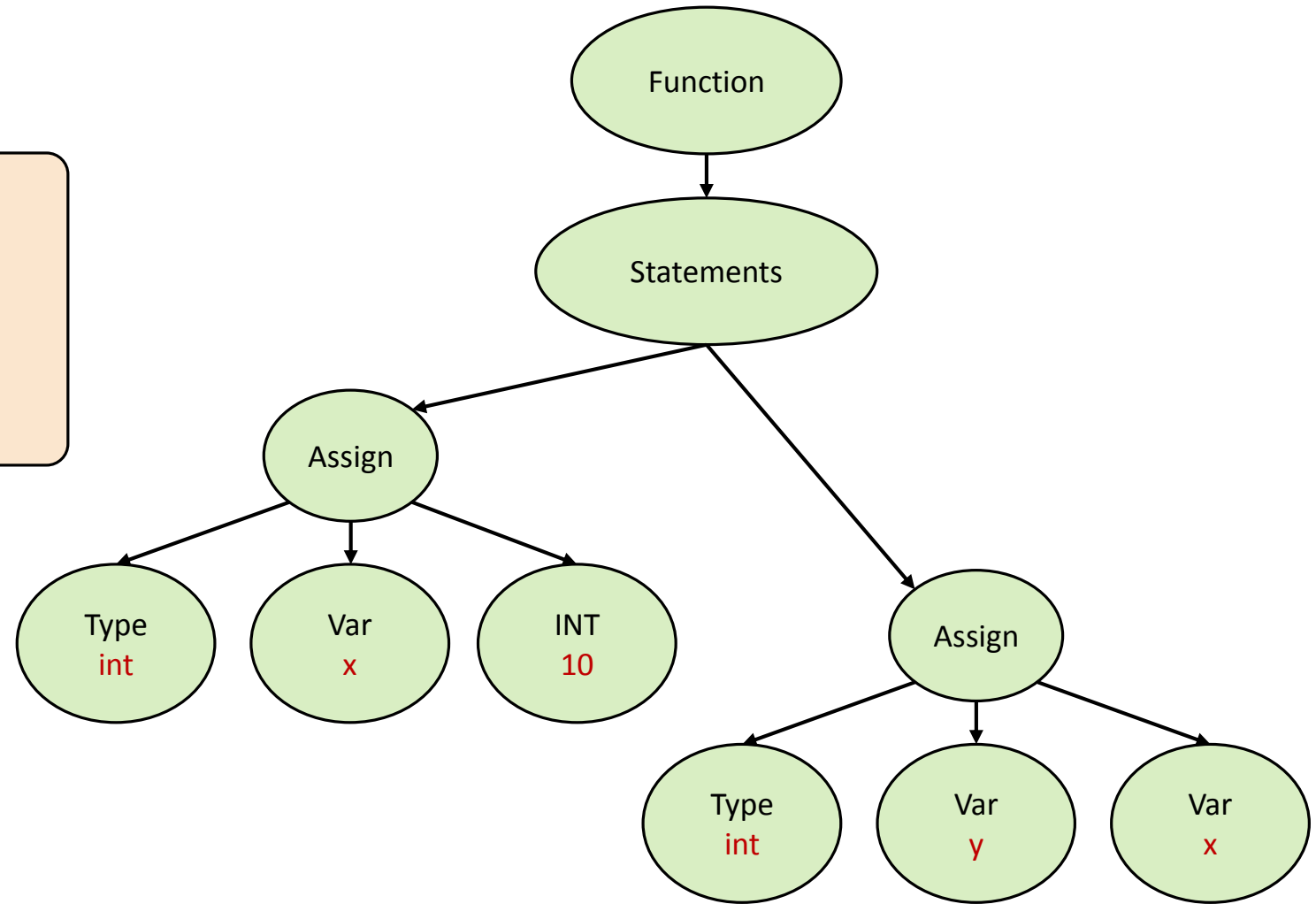
# Assignments

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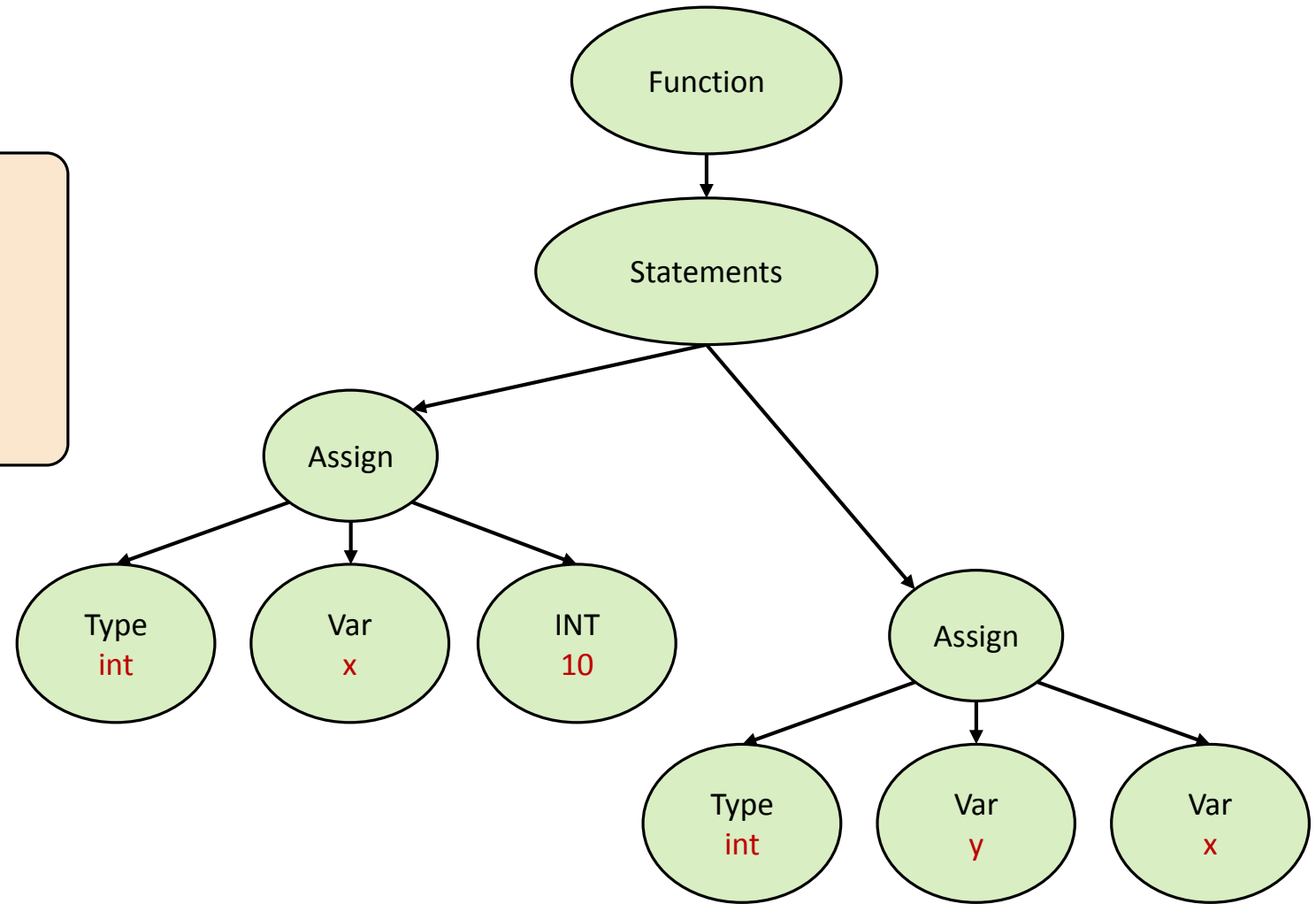
Valid

# Assignments

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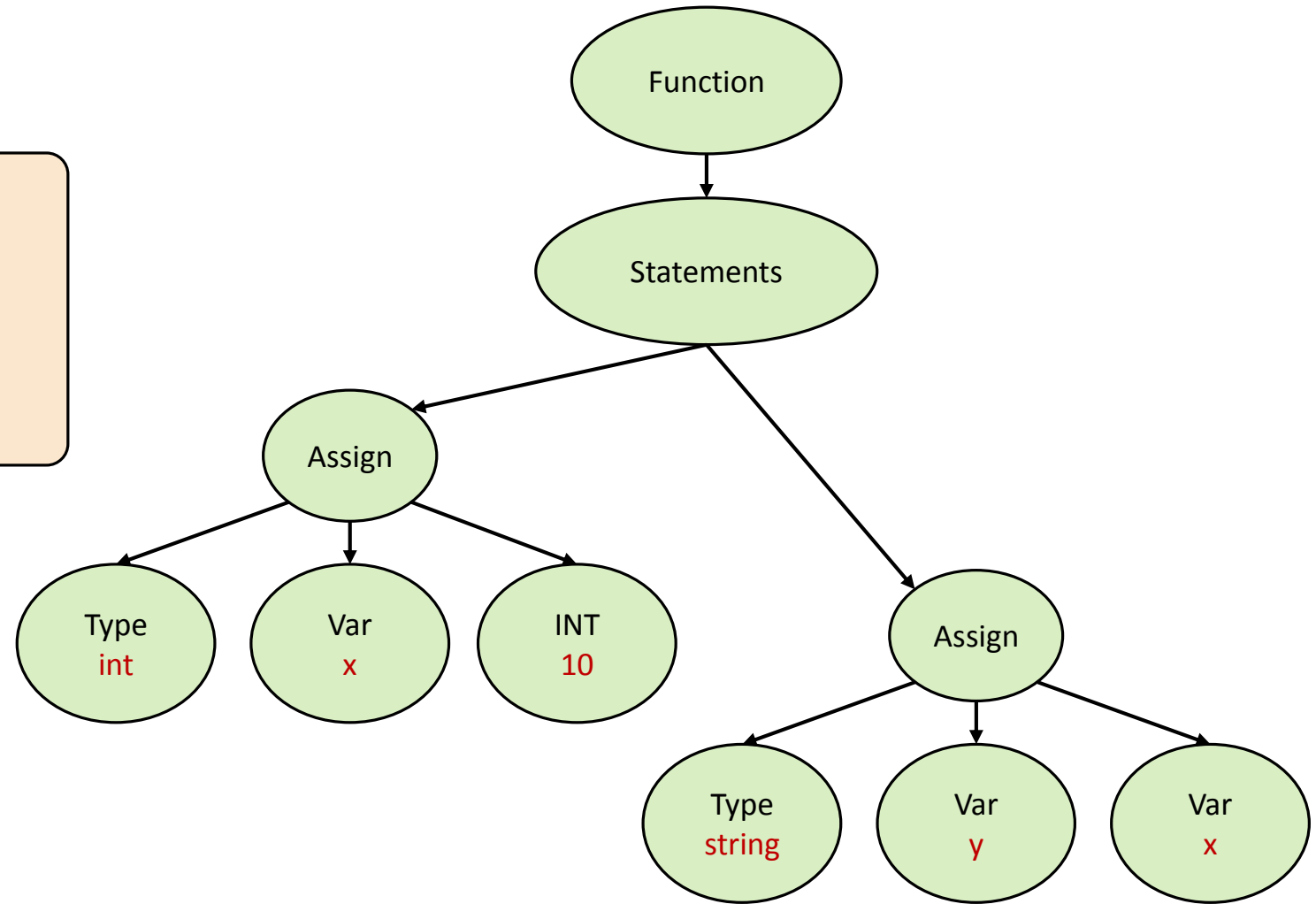
# Assignments

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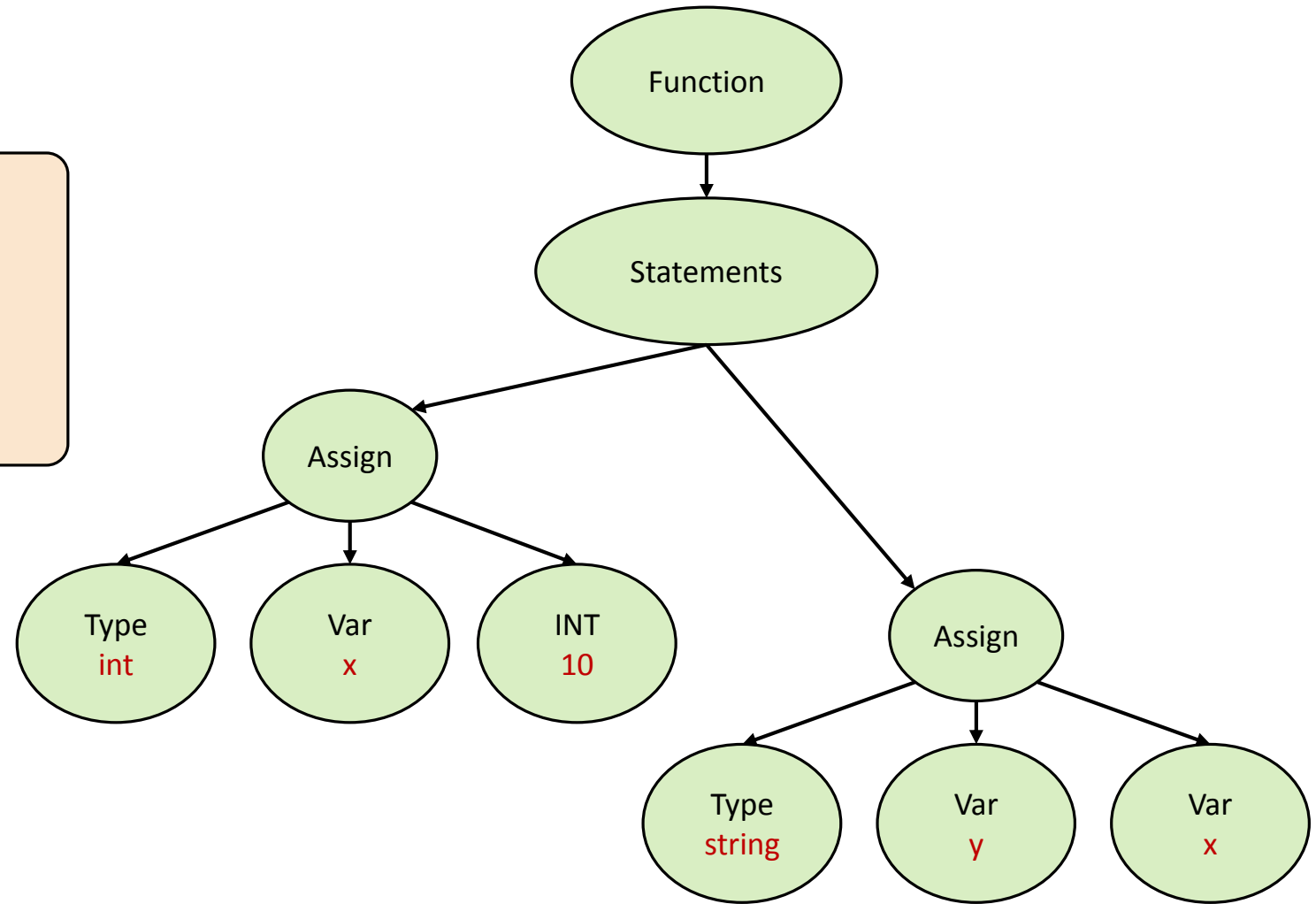
Invalid

# Binary Operations

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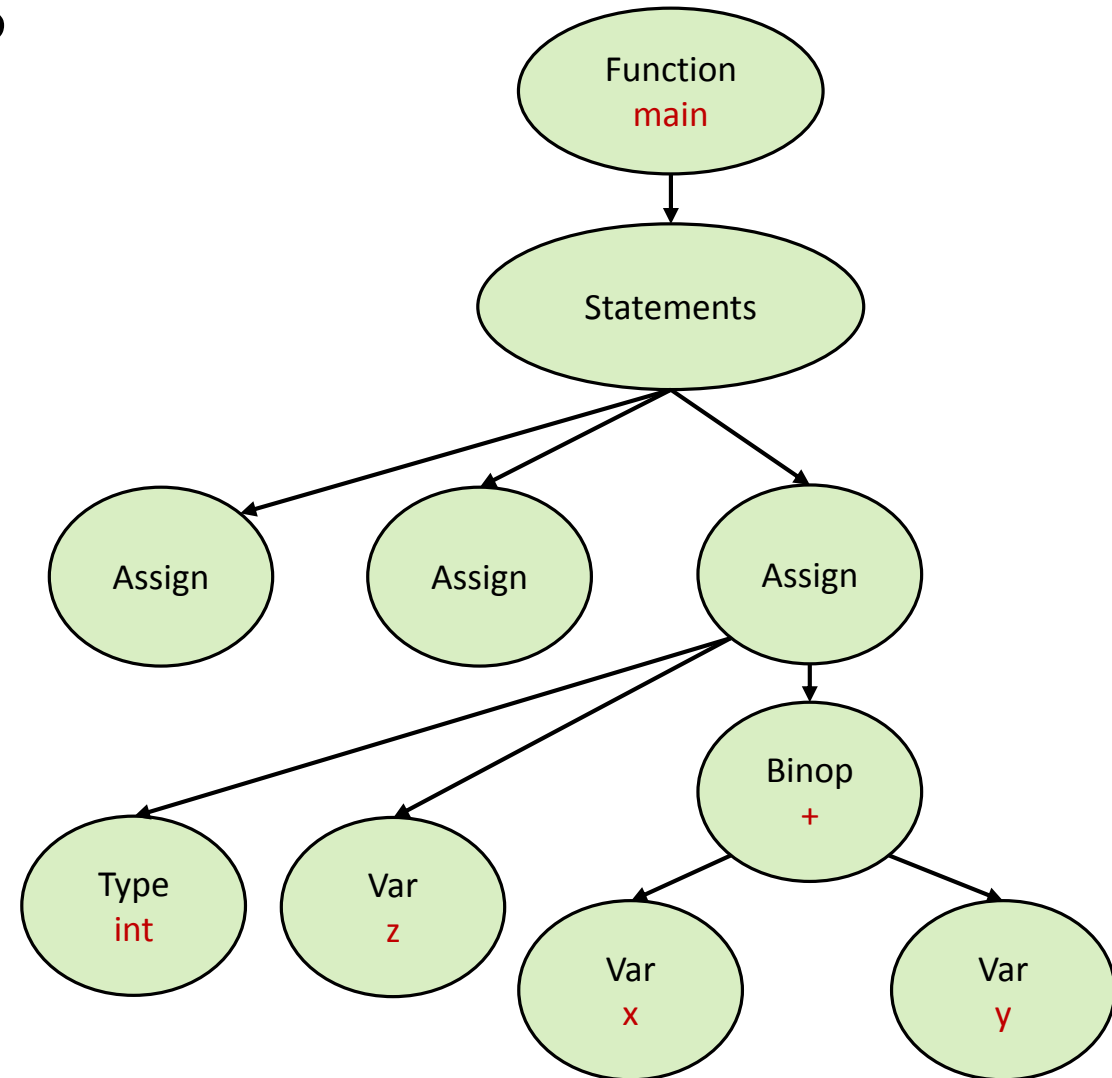
# Binary Operations

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

Valid
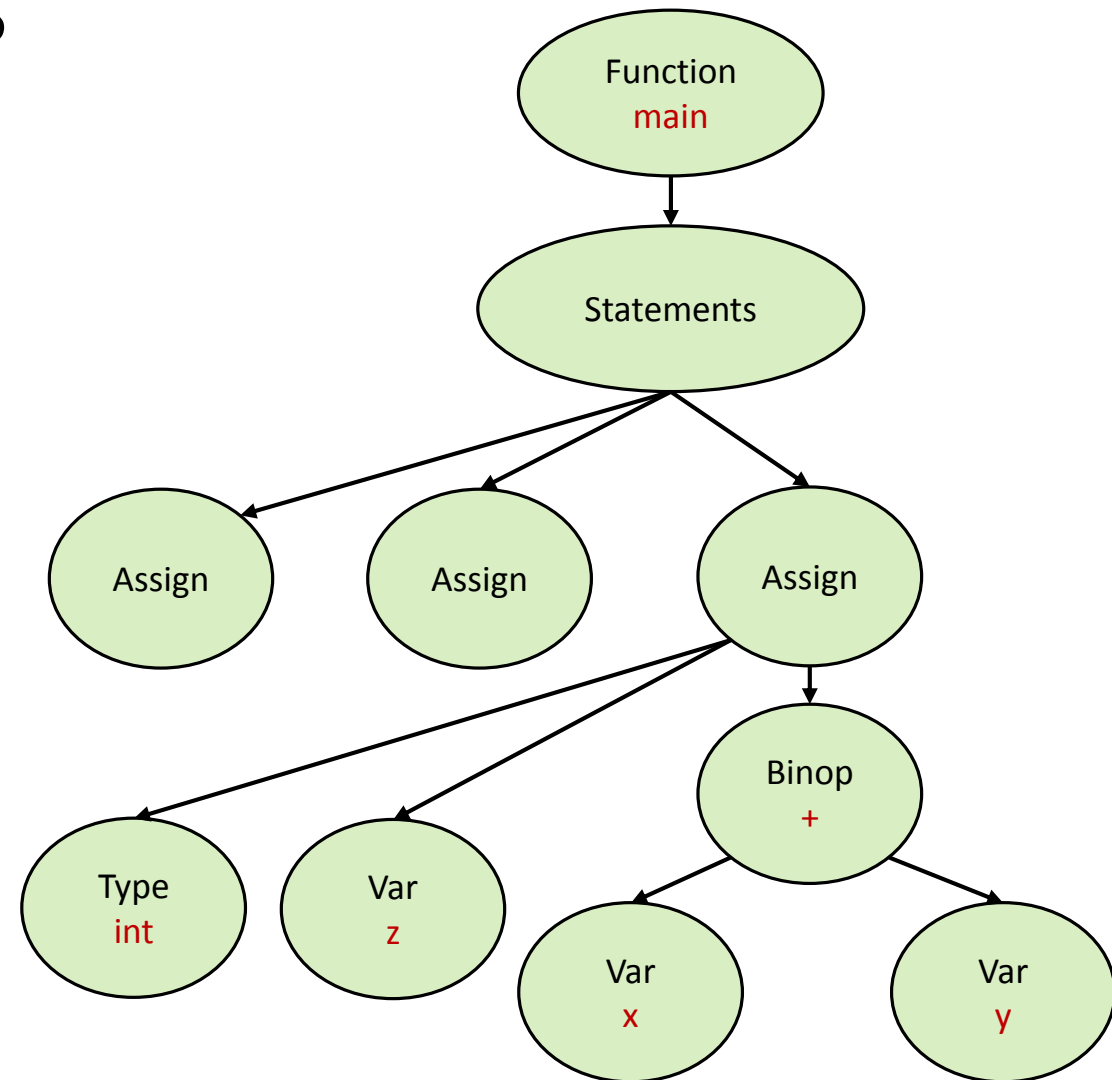
# Binary Operations

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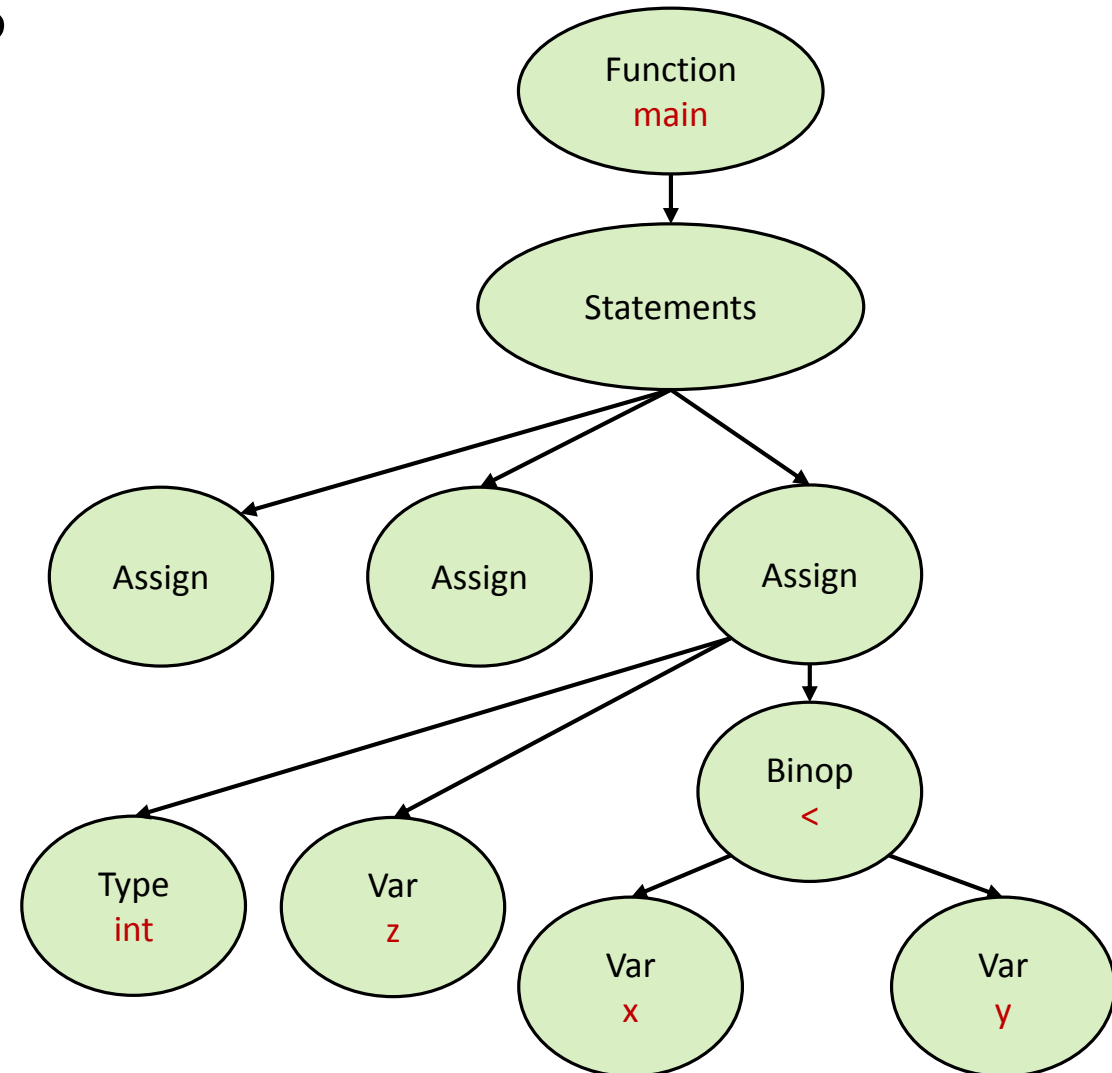
# Binary Operations

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

Invalid
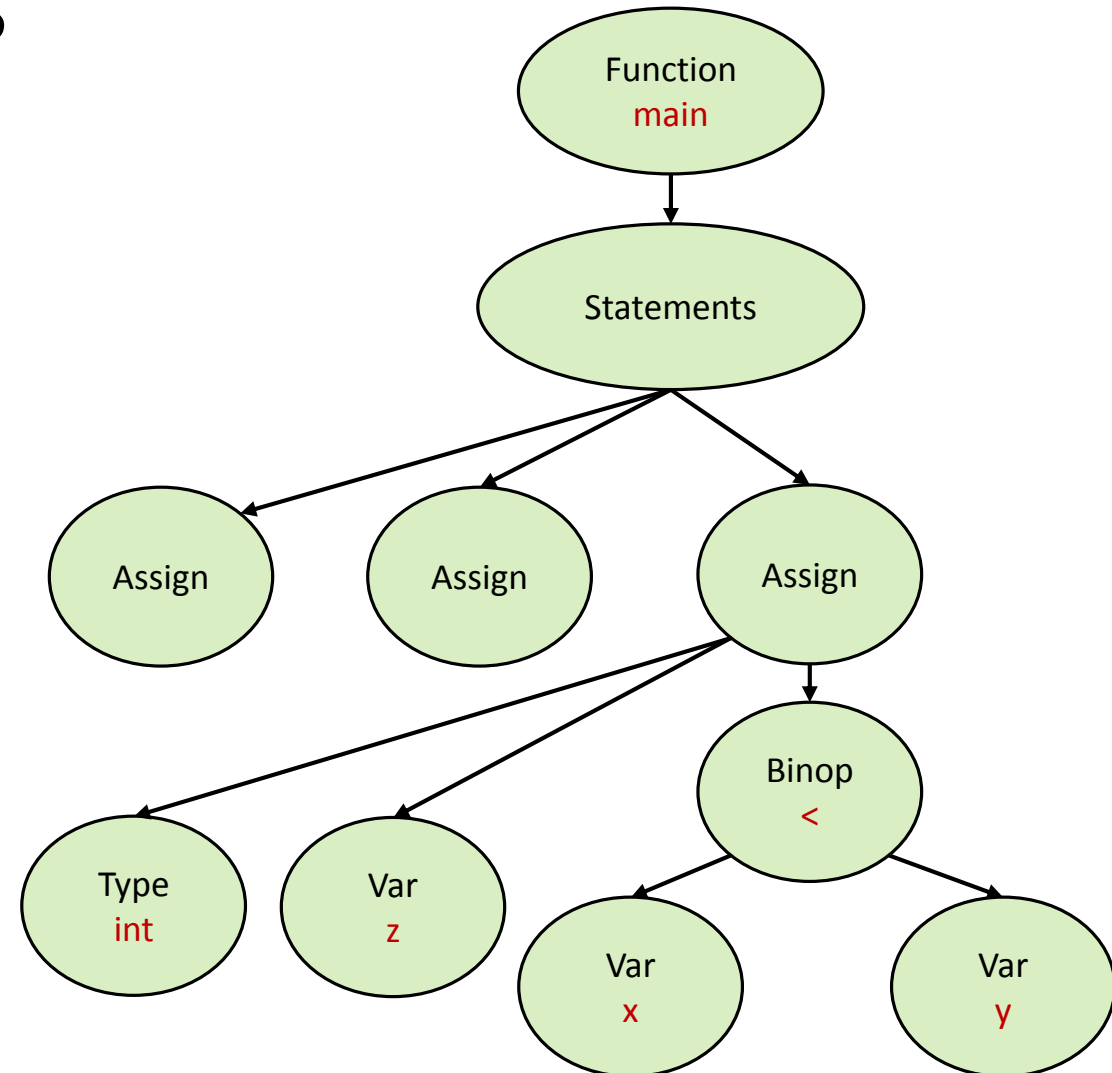
# Binary Operations

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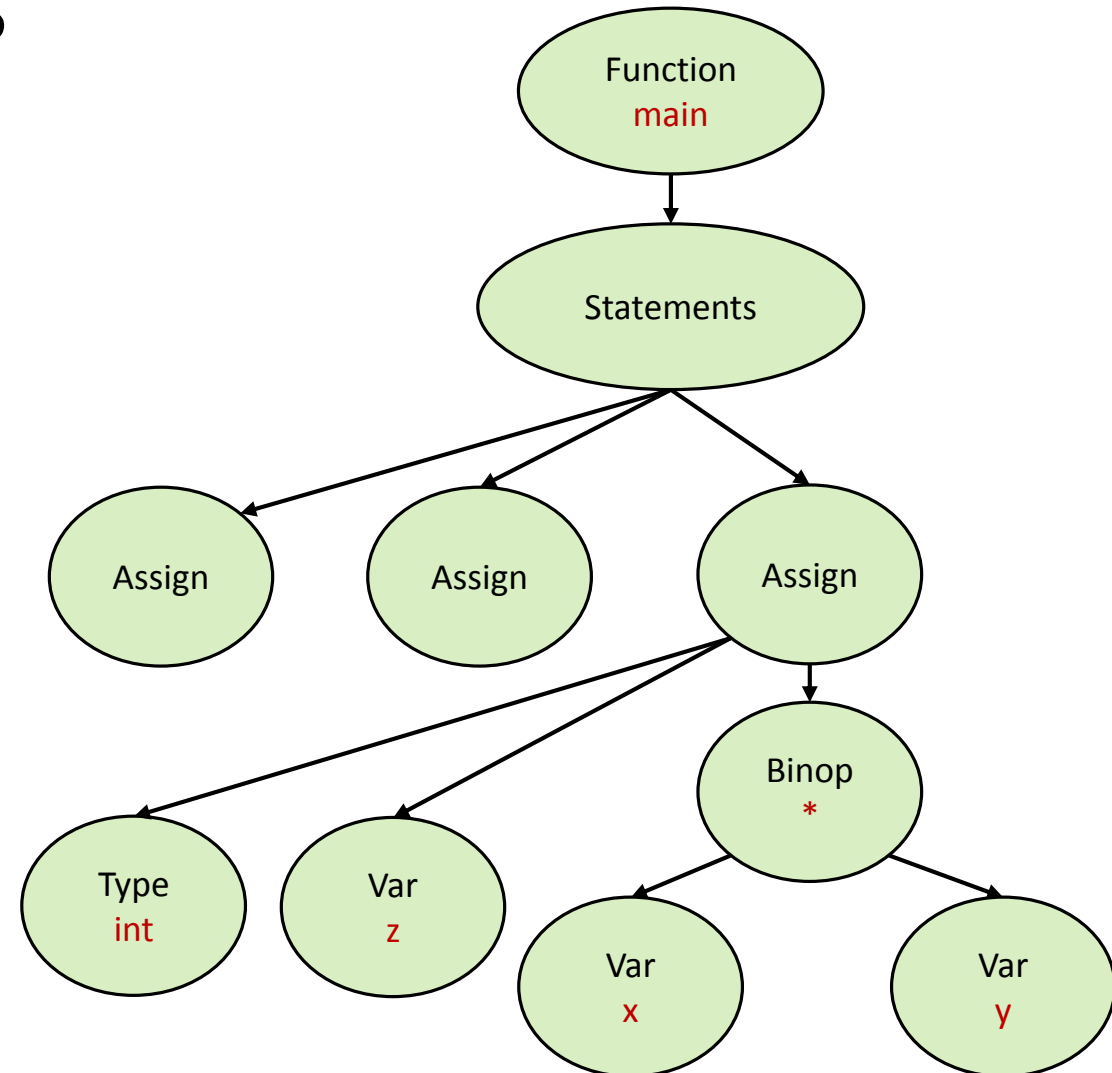
# Binary Operations

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



Invalid
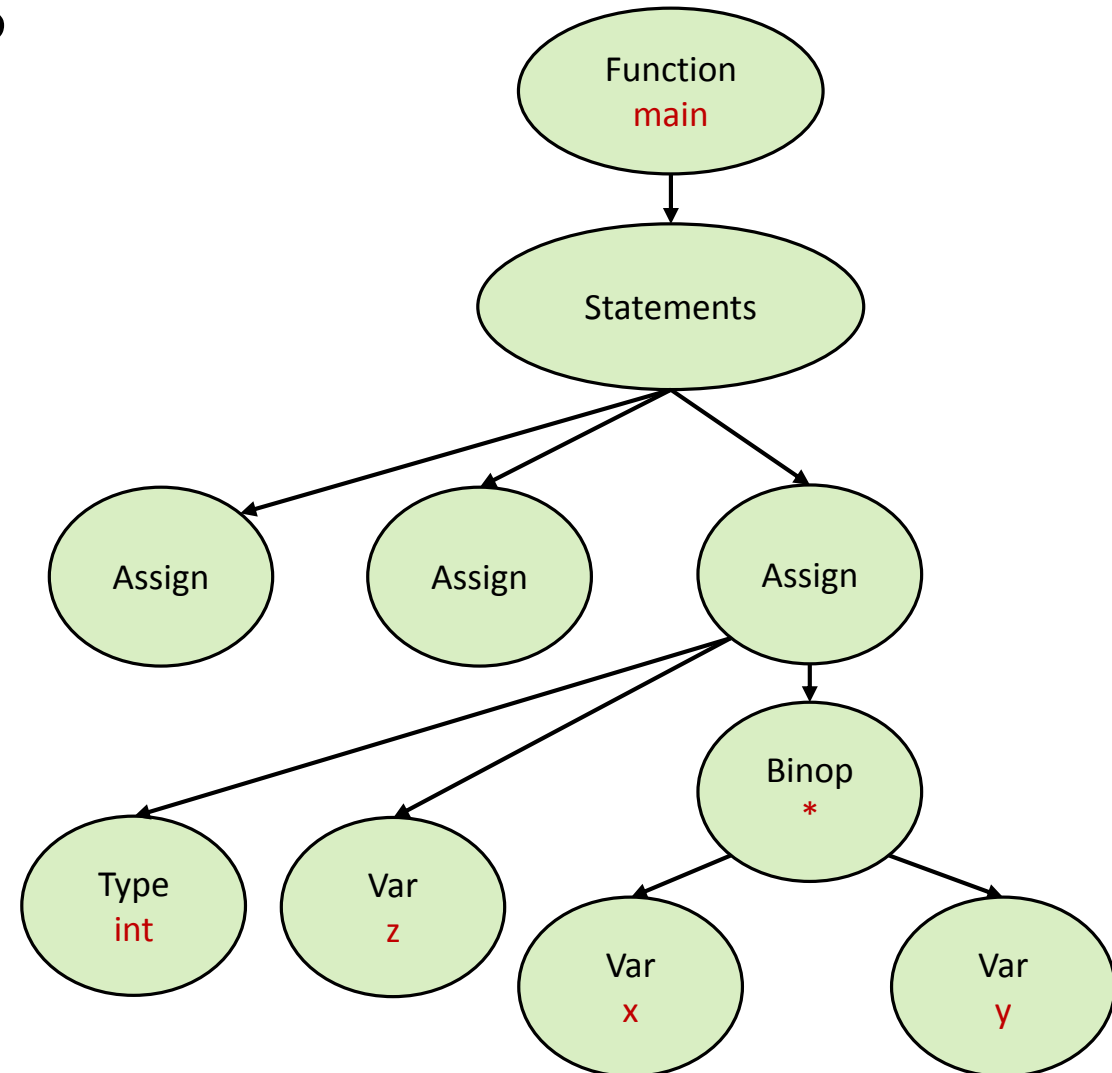
# Binary Operations

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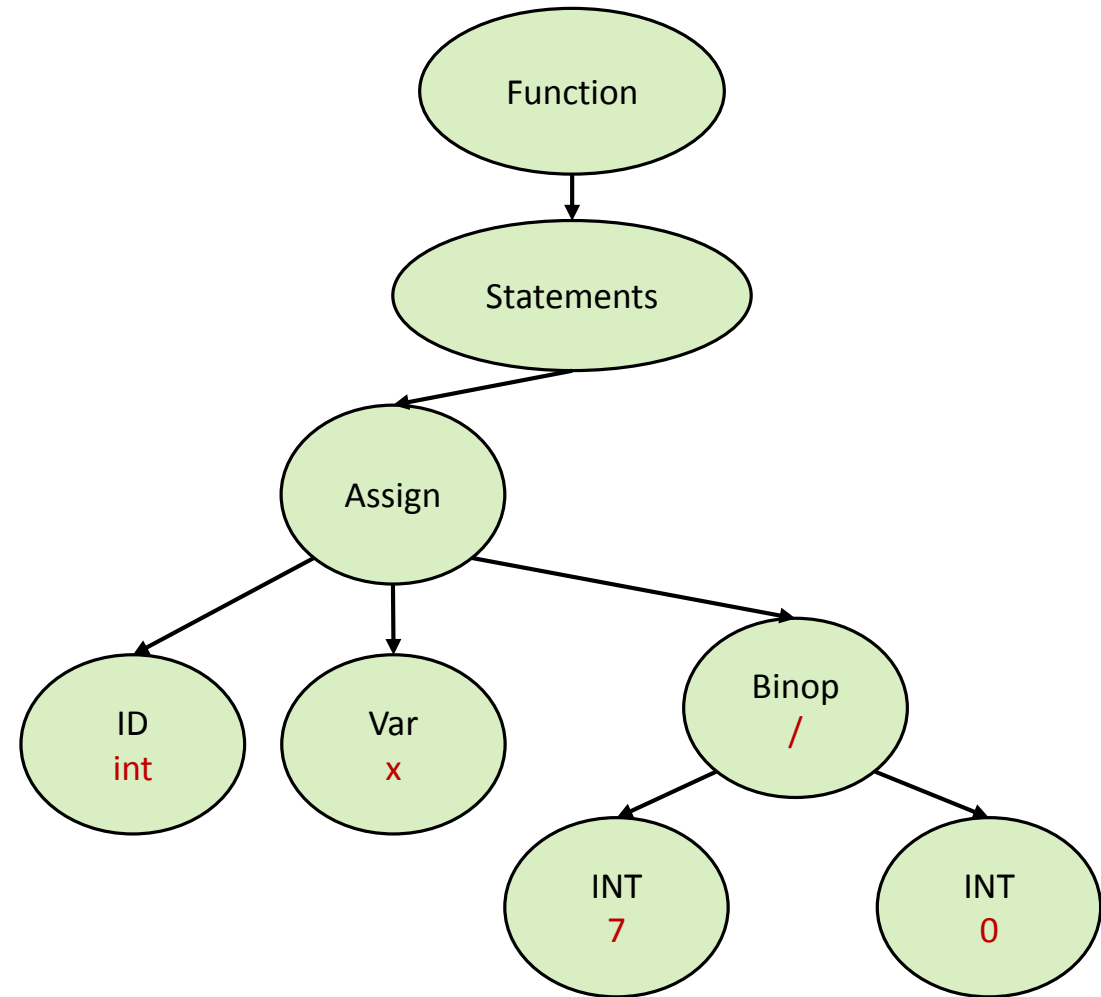
# Binary Operations

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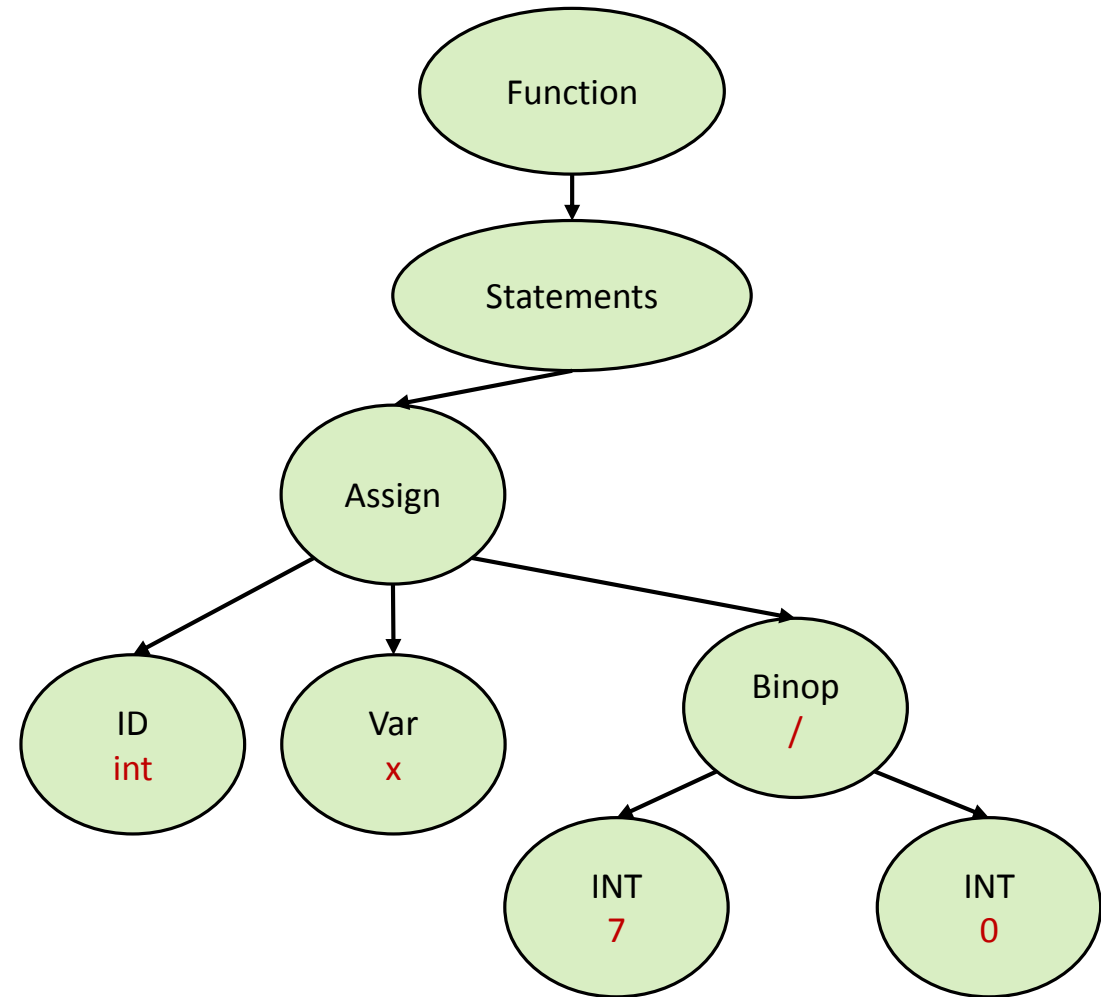
Invalid

# Binary Operations

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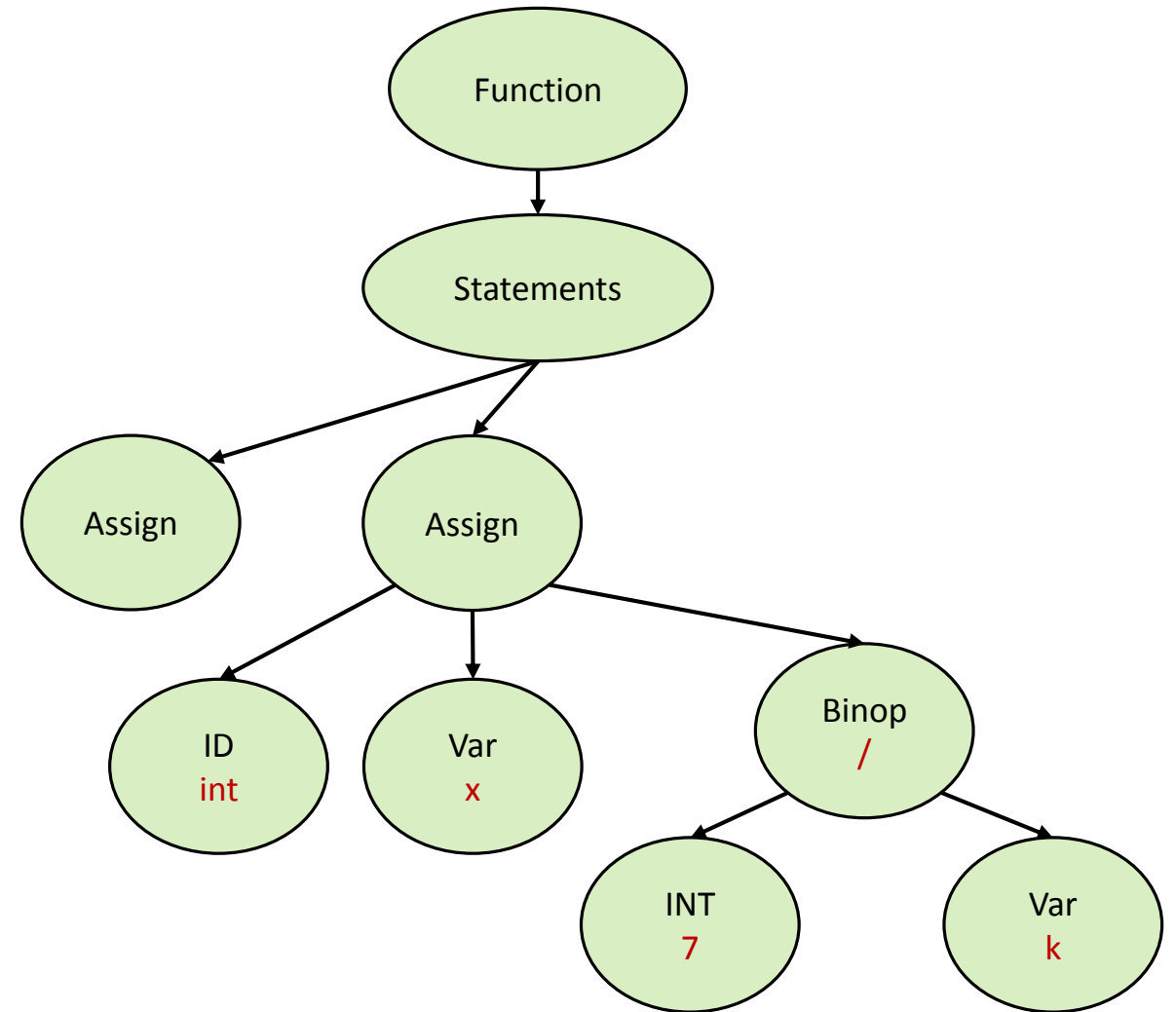
# Binary Operations

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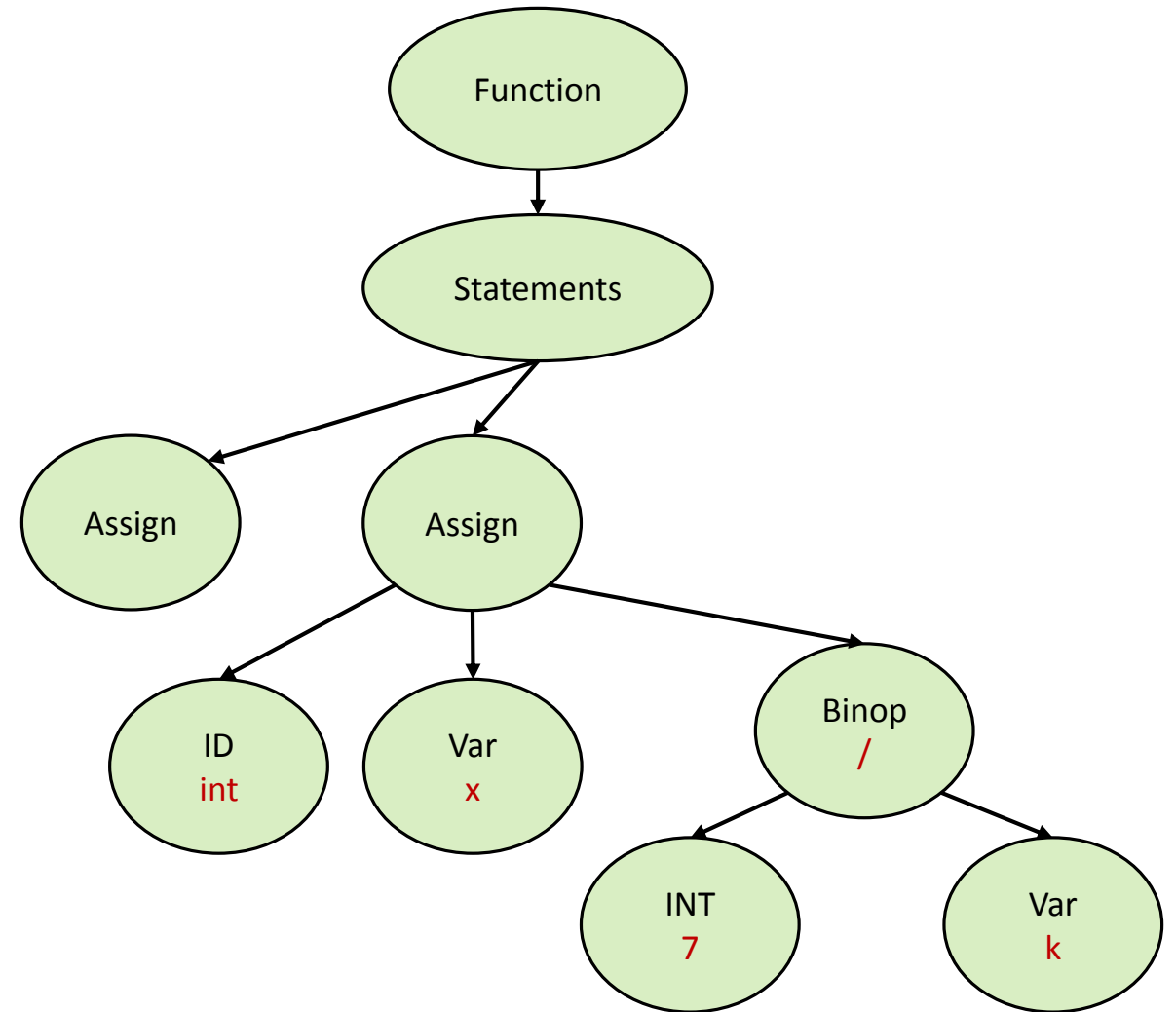


Depends

# If, While, …

```
void main() {
  int x = 1;
  if (x + 1) {
    int z = 2;
  }
}
```
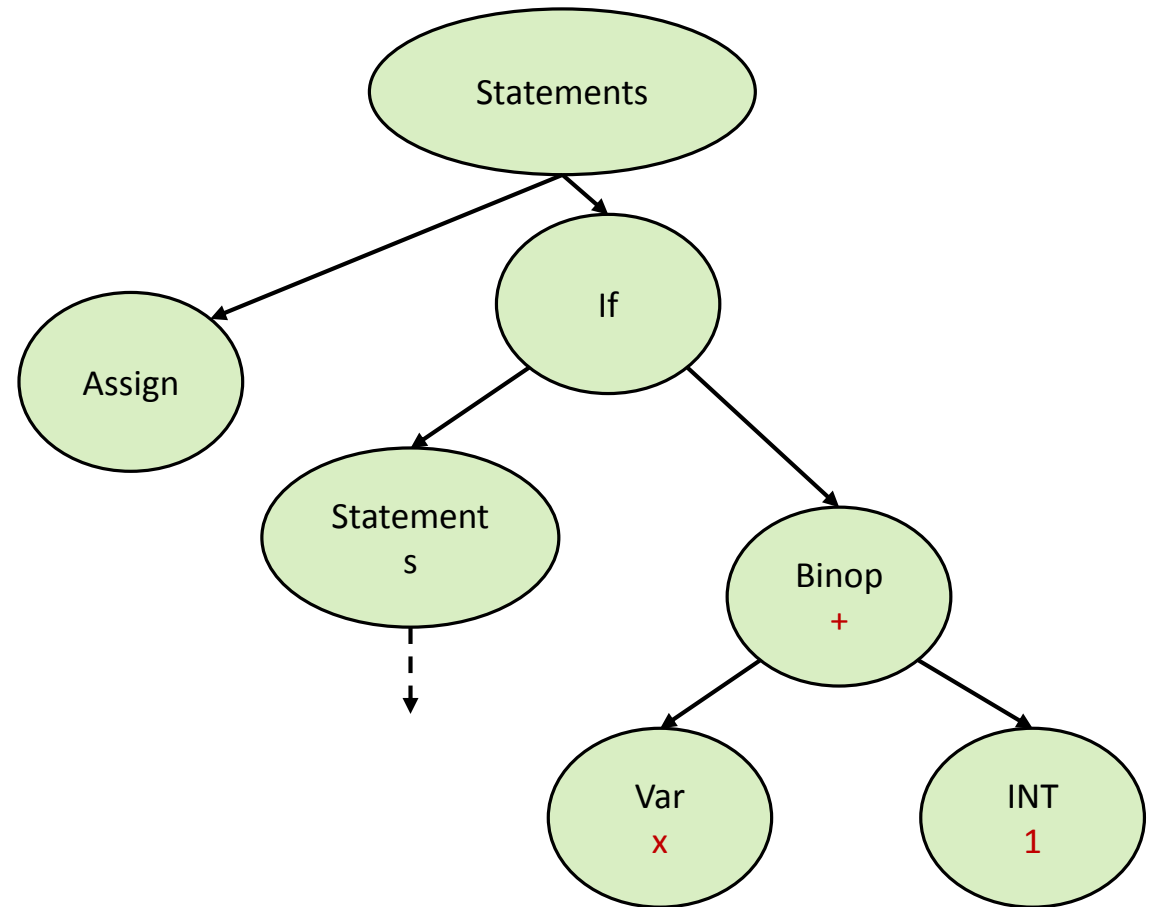
# If, While, …

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

**Valid**
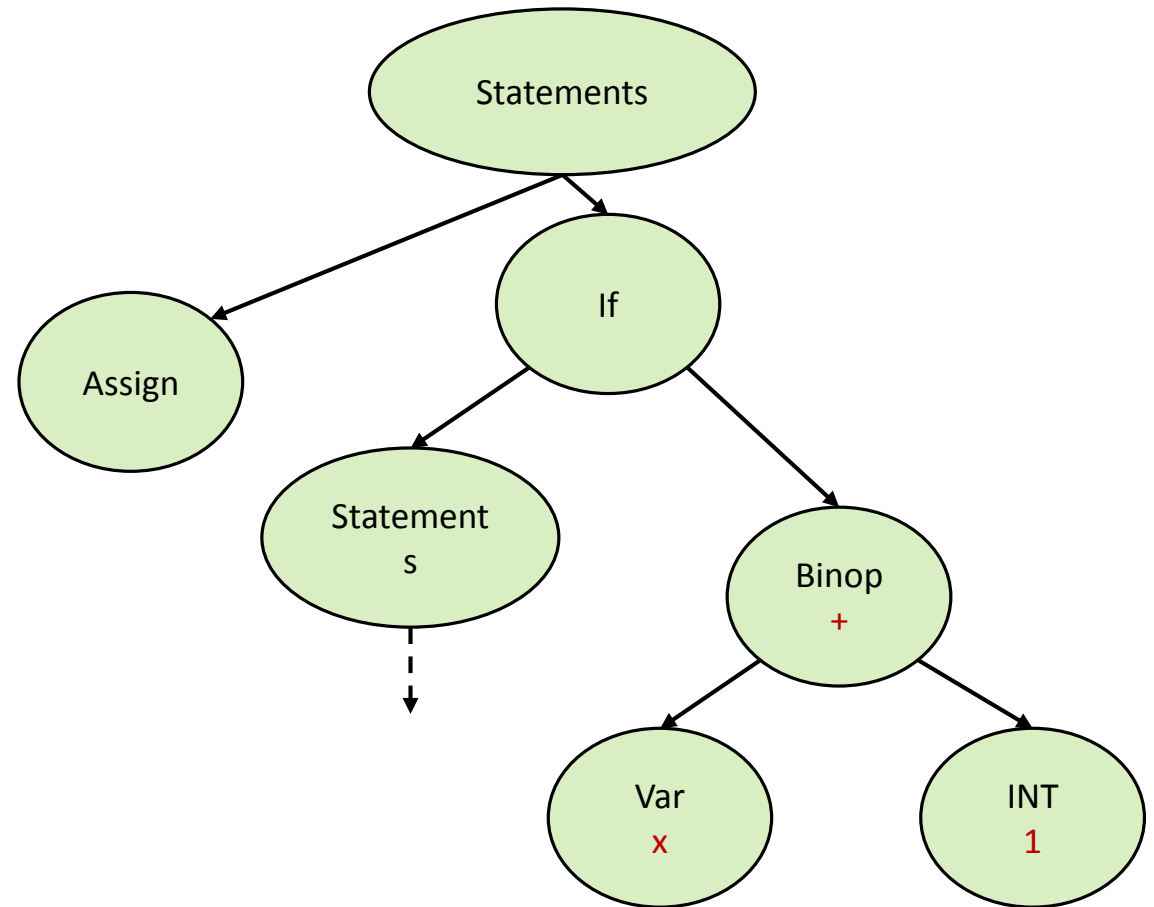
# If, While, …

```
void main() {
    string x = "A";
    while (x + "B") {
        int z = 2;
    }
}
```
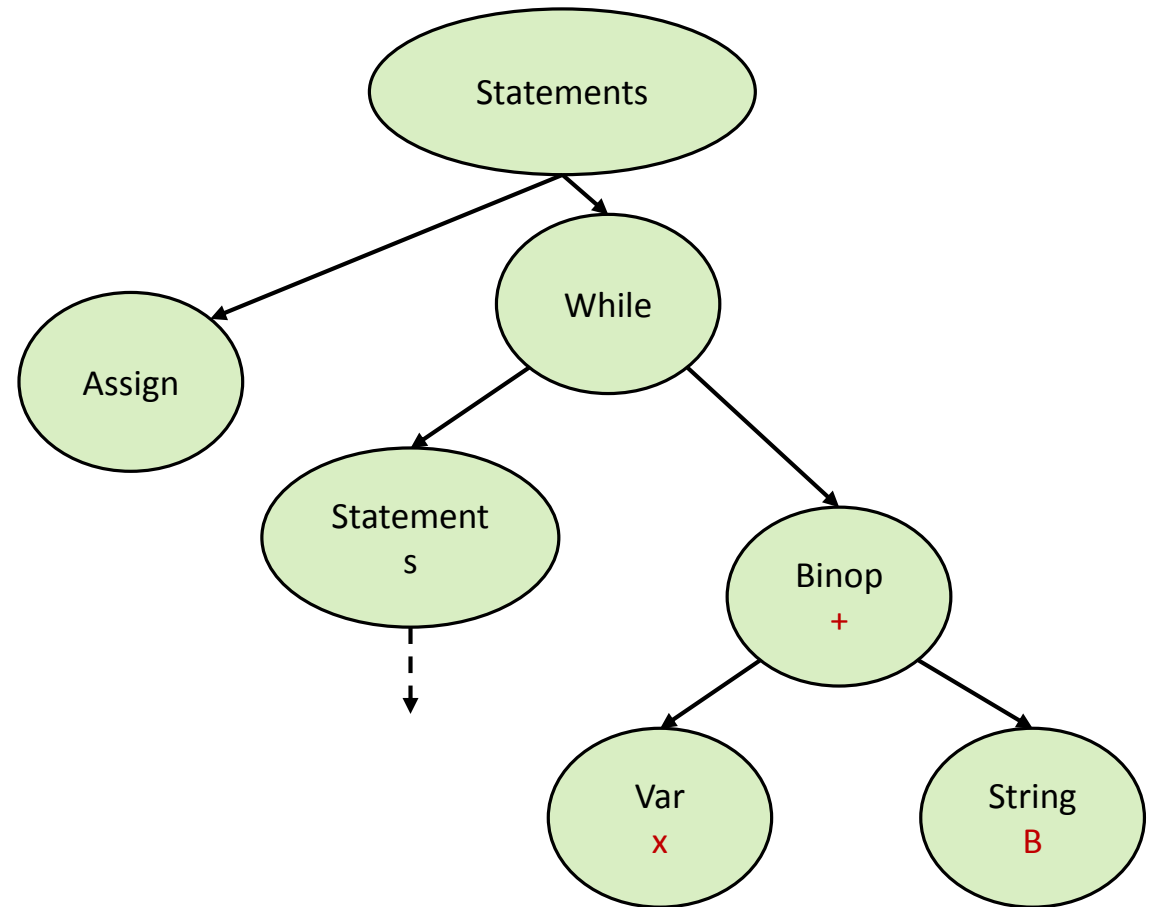
# If, While, …

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

Invalid
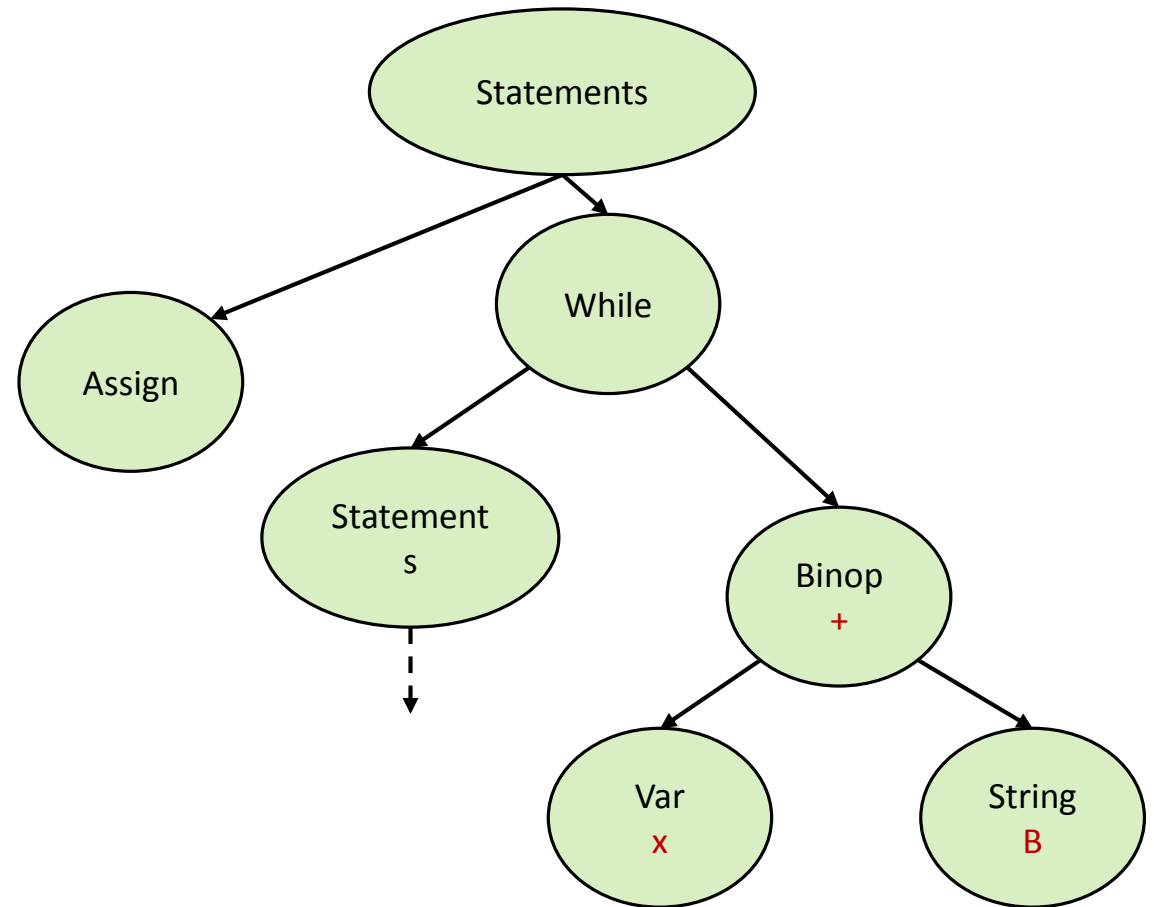
# Return Statement

```
int main() {
    return 17;
}
```
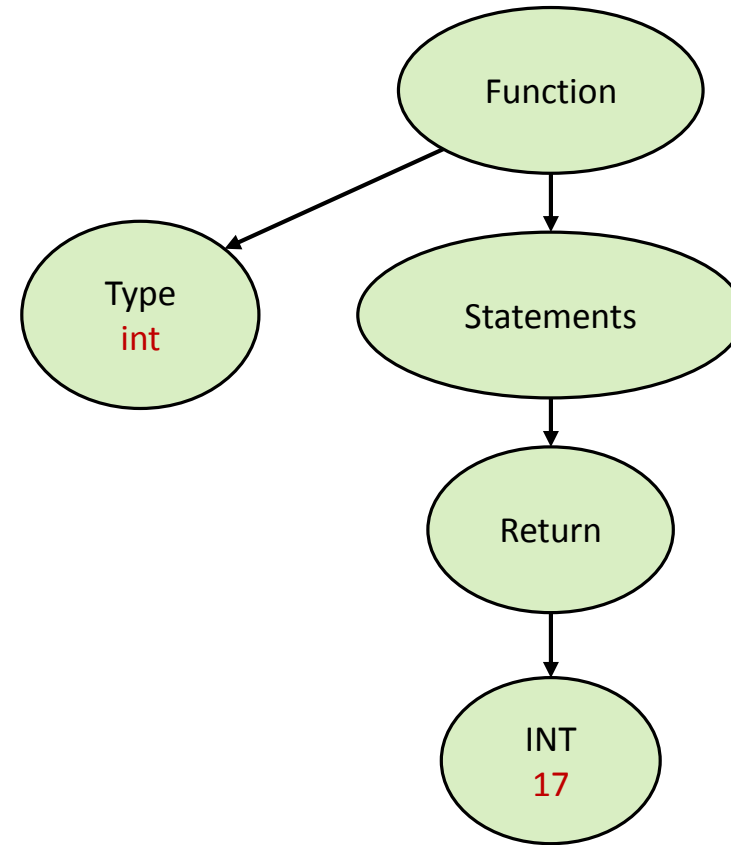
# Return Statement

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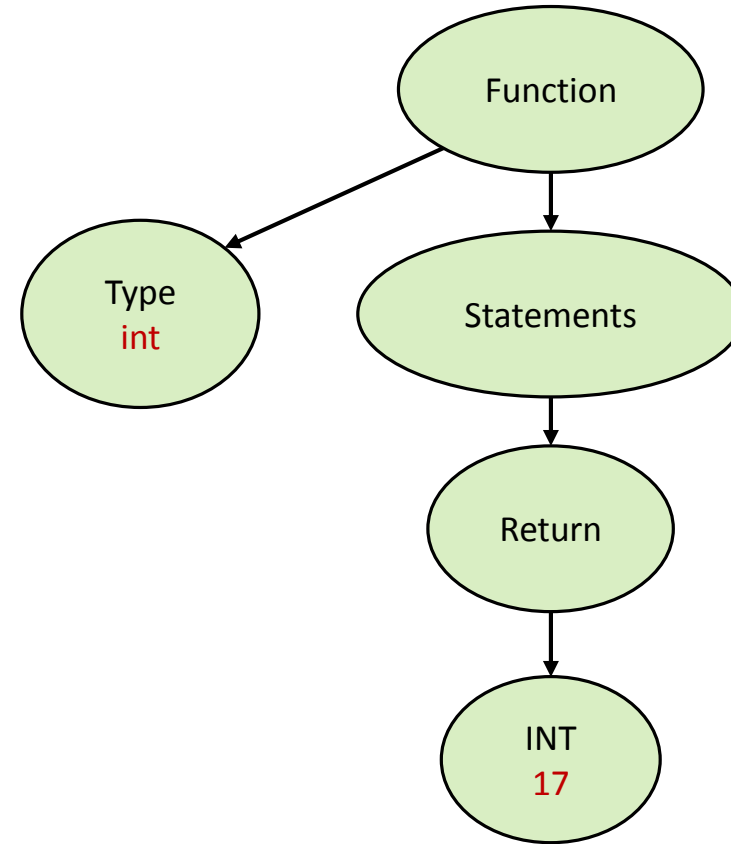


Valid

# Return Statement

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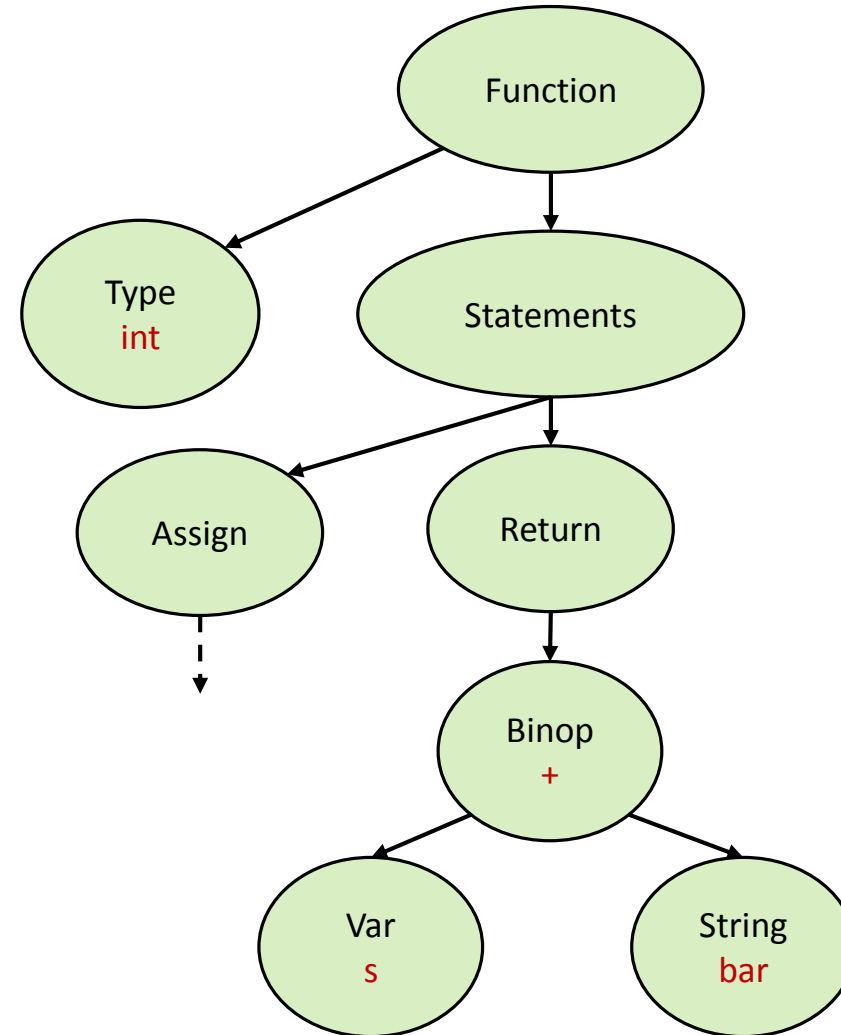
# Return Statement

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

Invalid
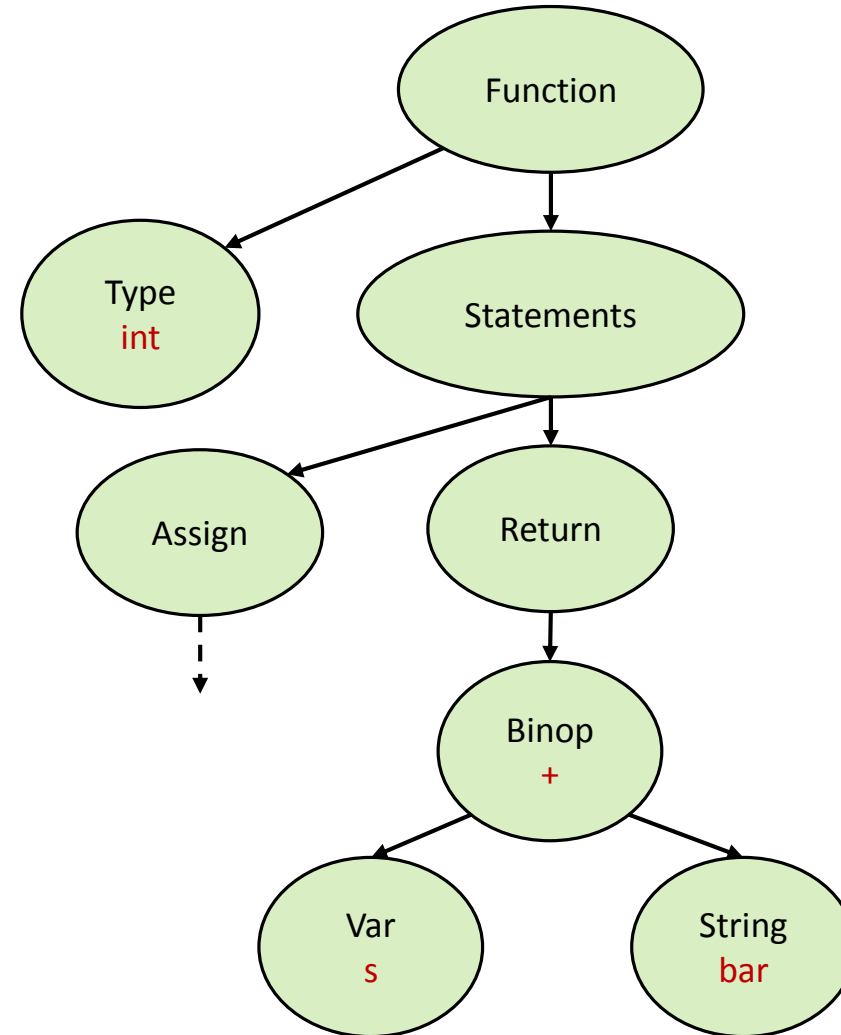
# Return Statement

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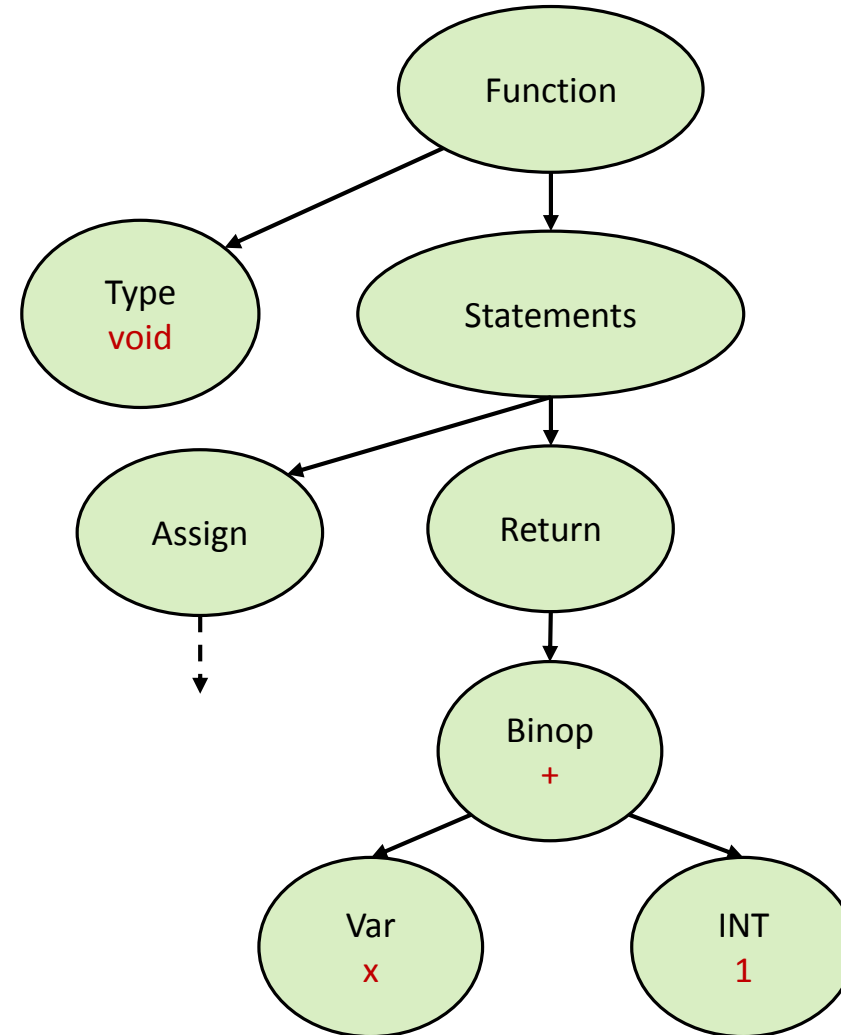
# Return Statement

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

Invalid
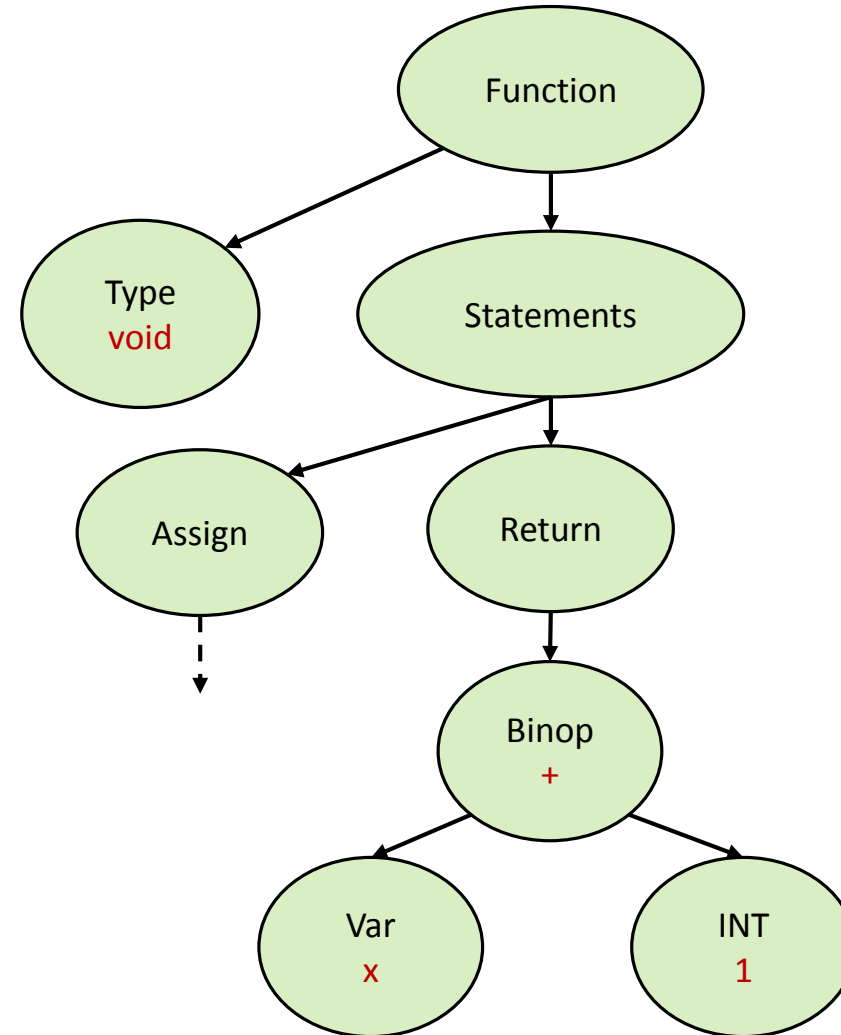
# Return Statement

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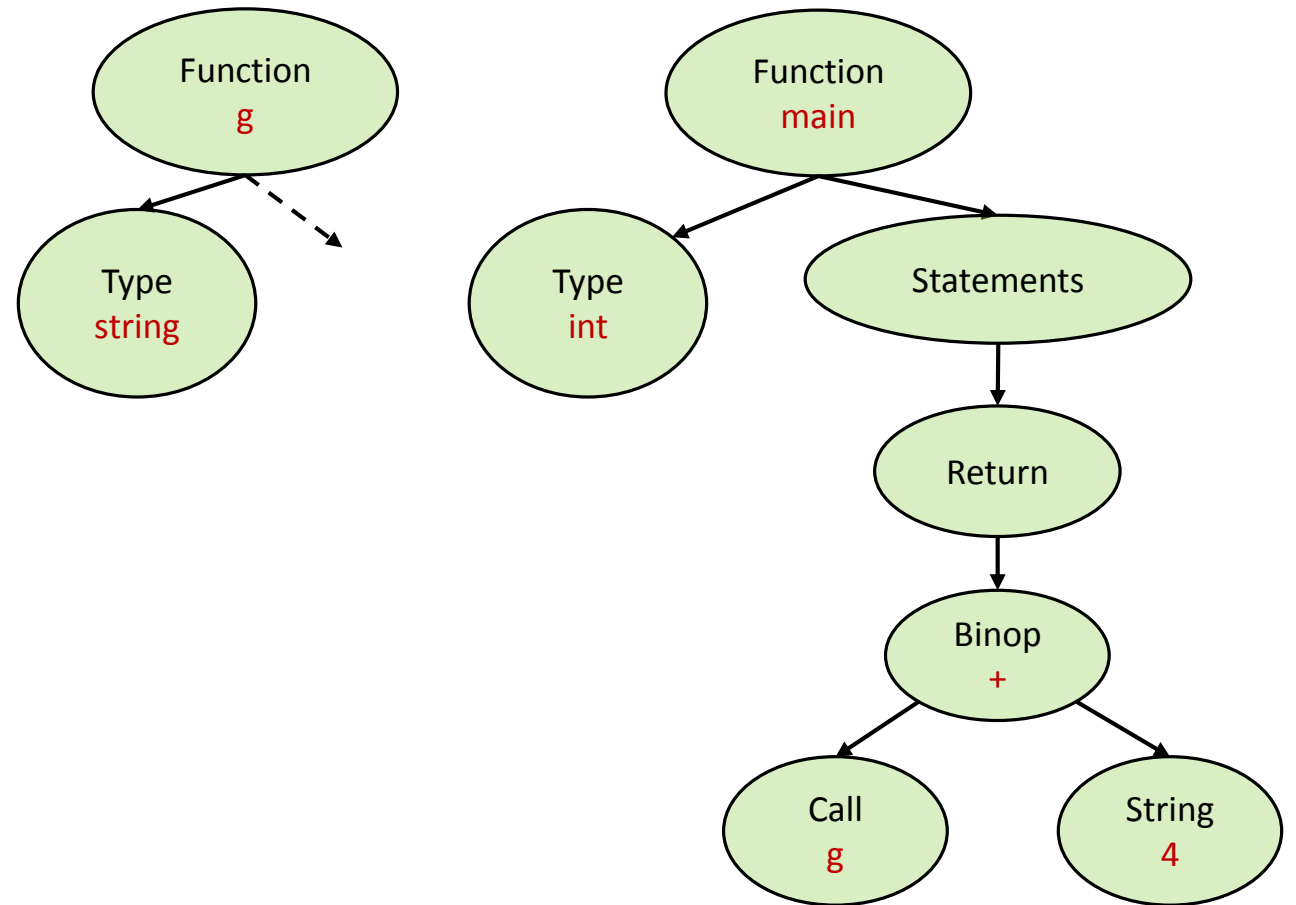
# Return Statement

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



Invalid
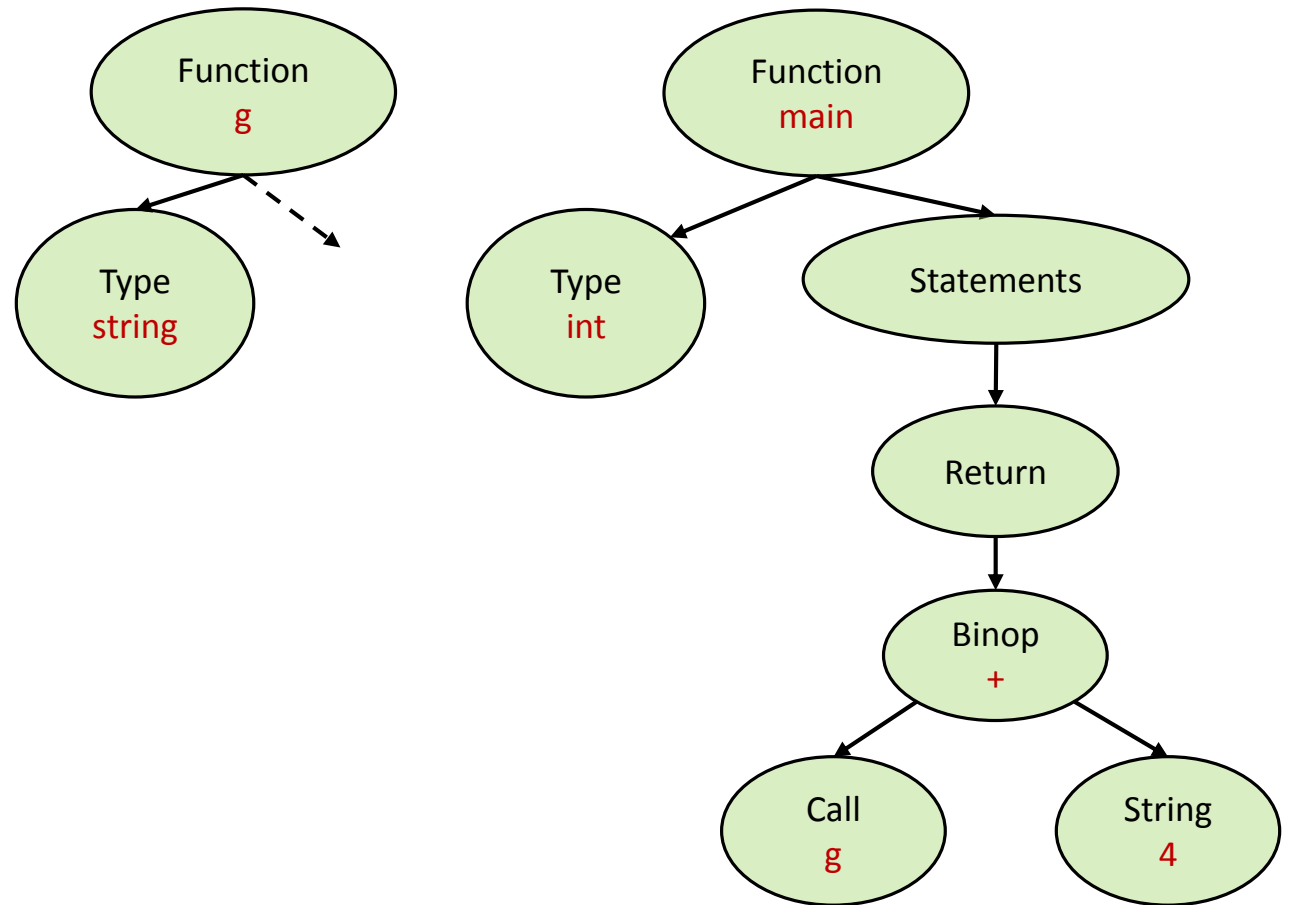
# Function Calls

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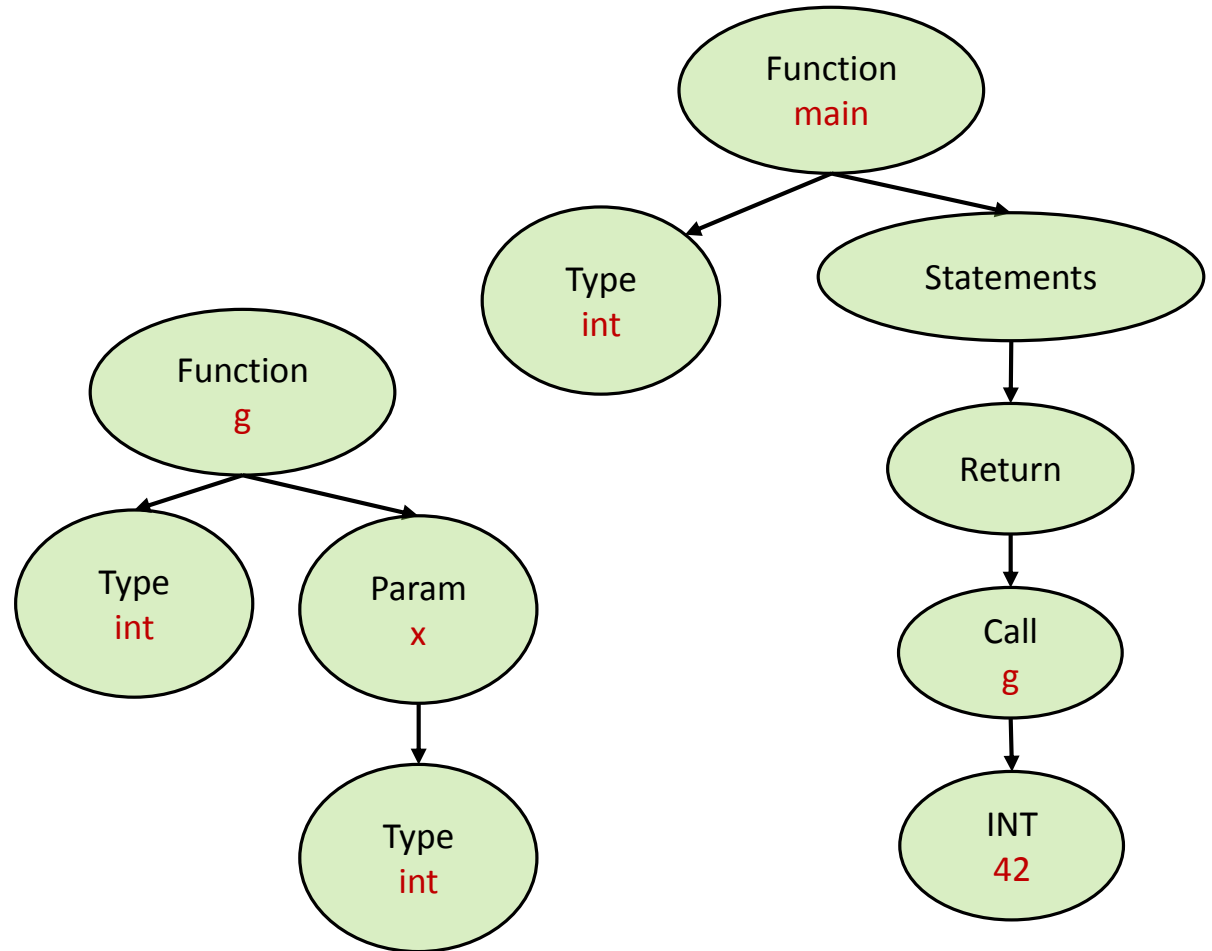
# Function Calls

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

Valid
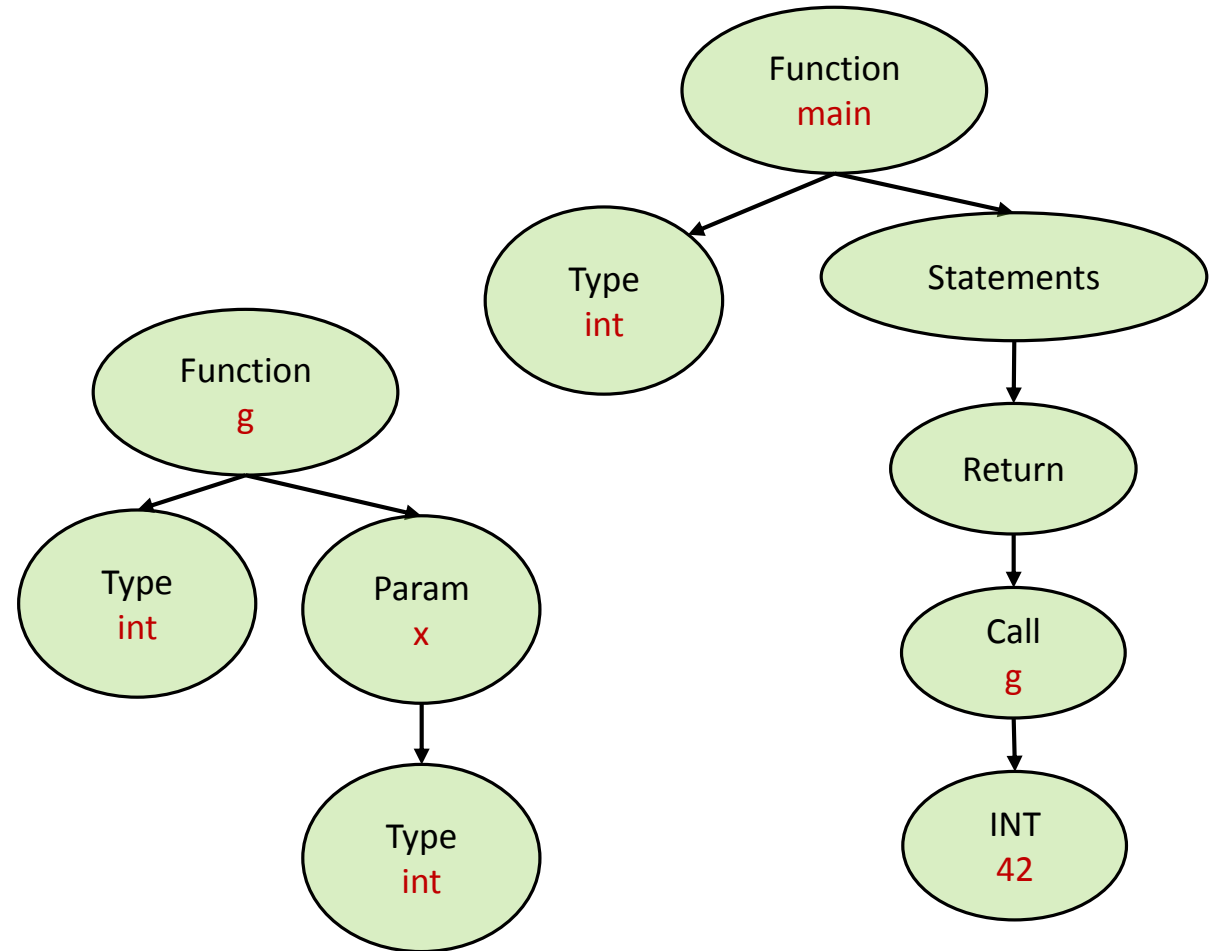
# Function Calls

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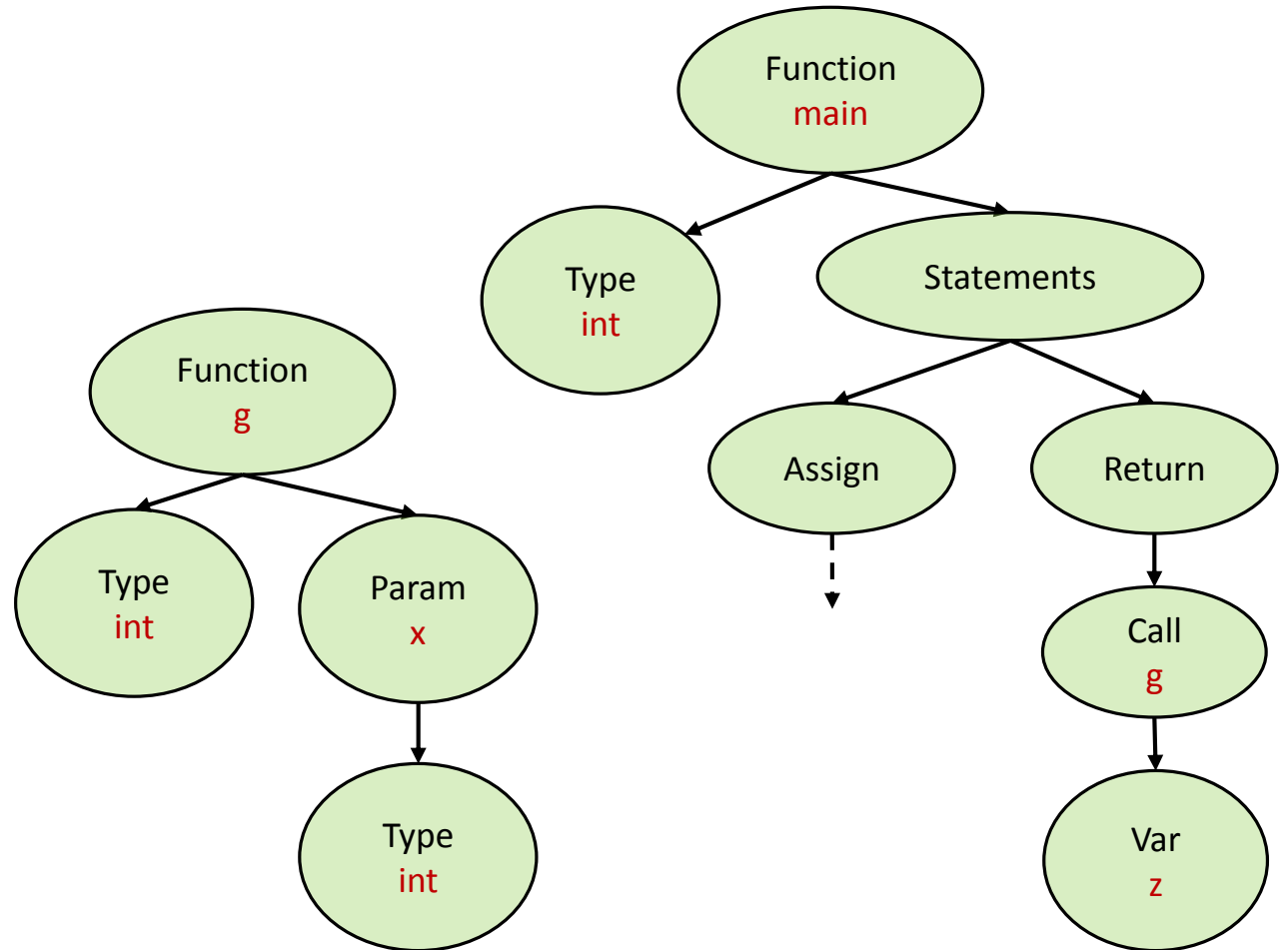
# Function Calls

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

**Invalid**
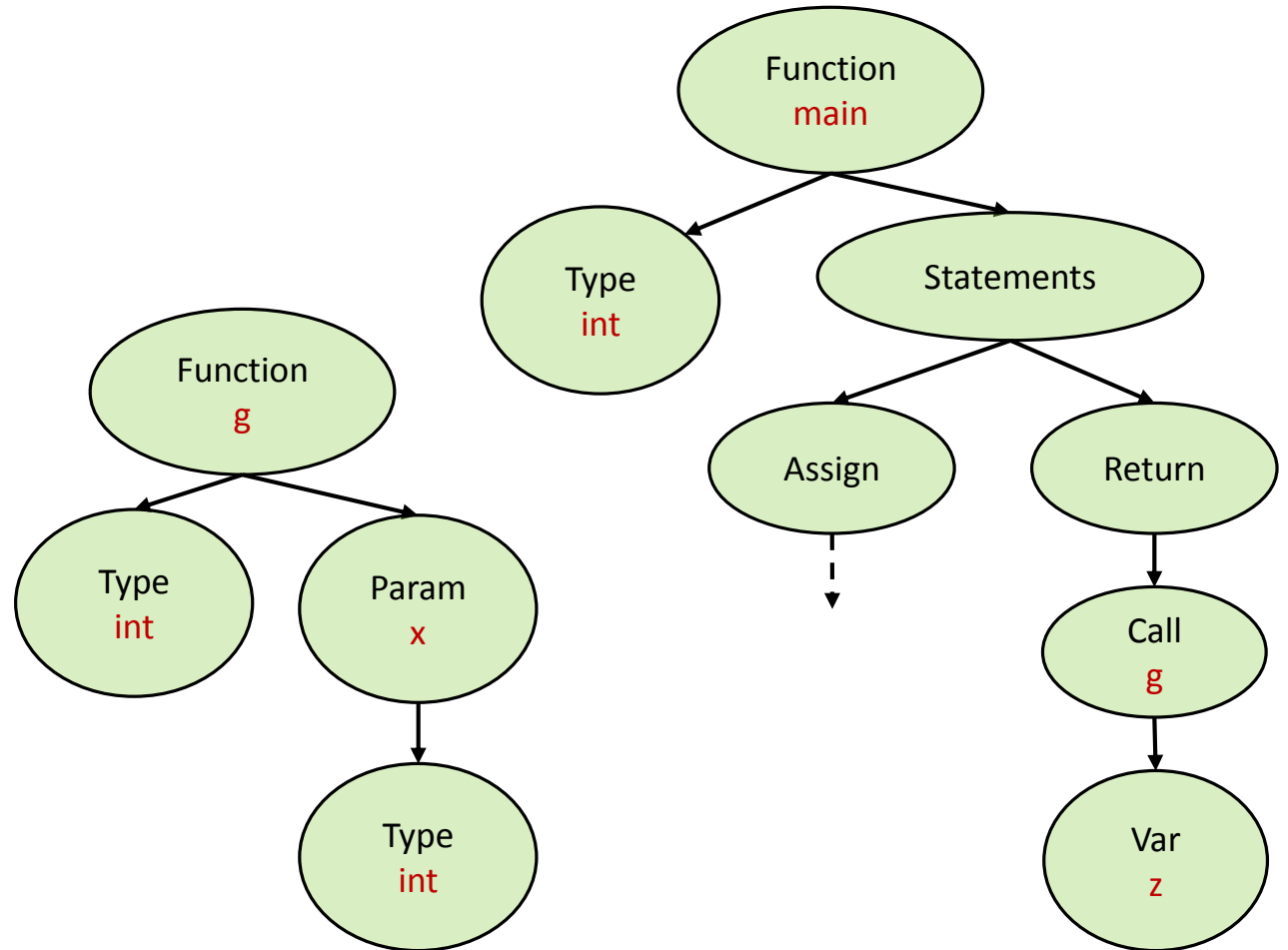
# Function Calls

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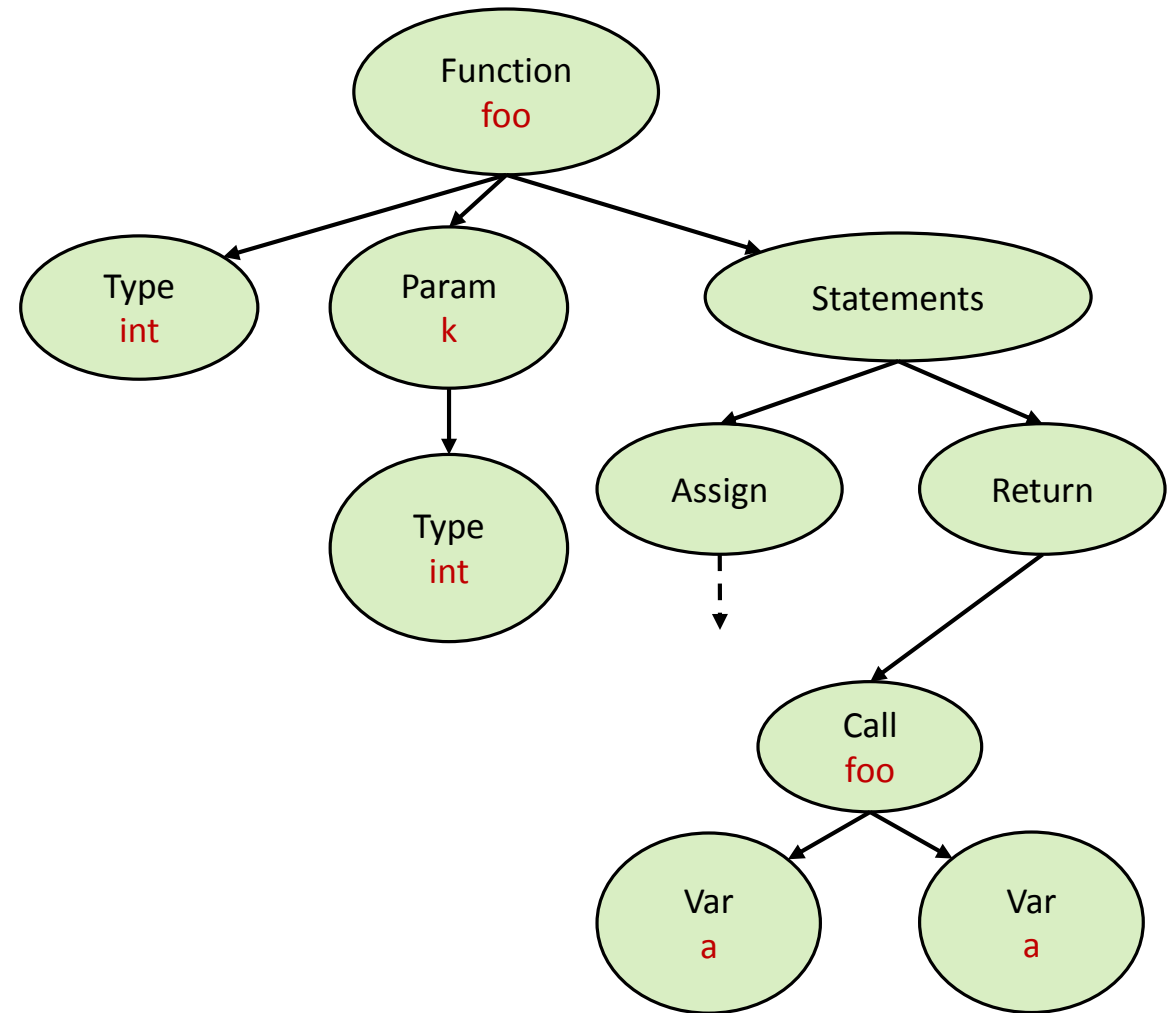
# Function Calls

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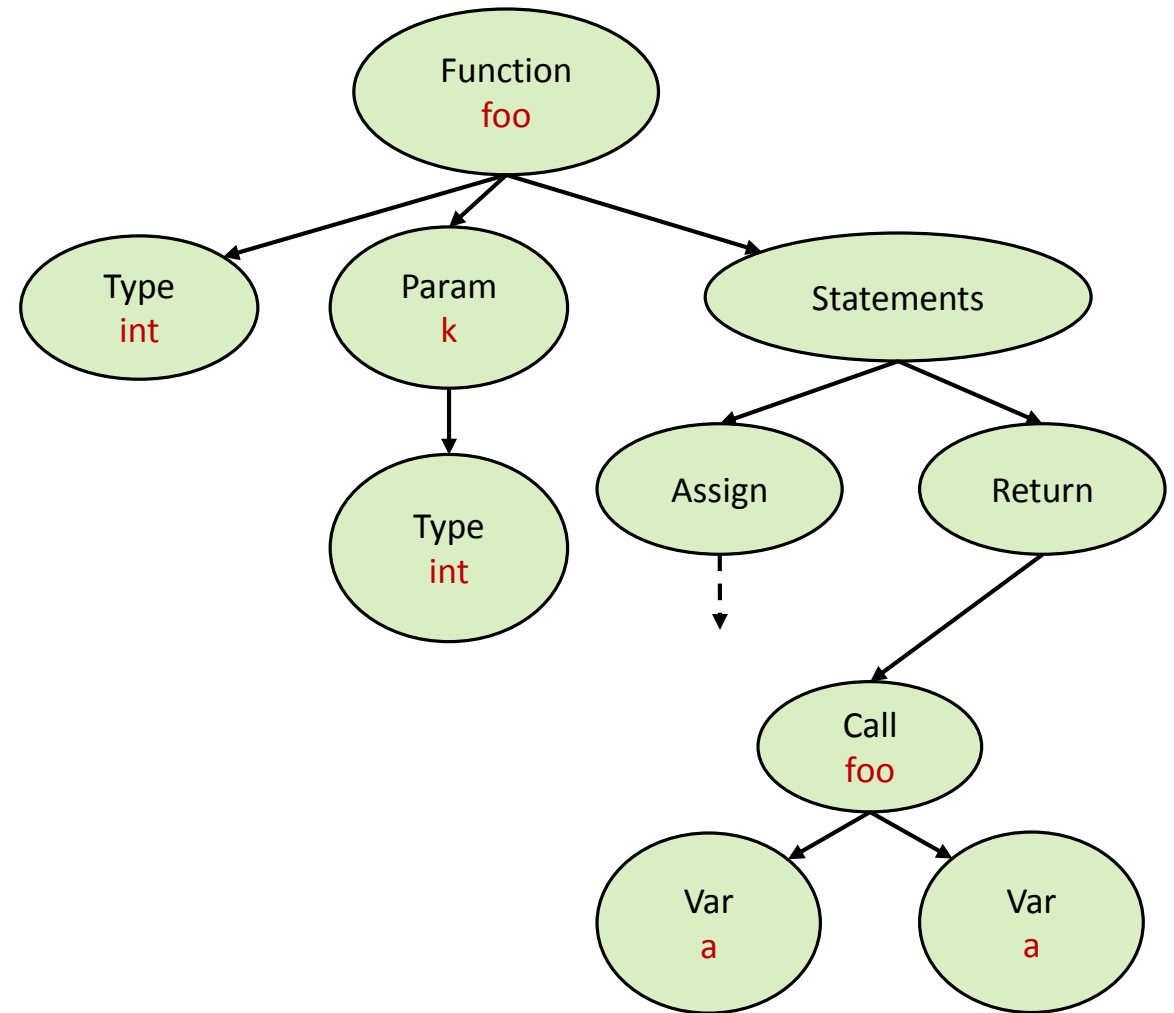
**Invalid**

# Function Calls

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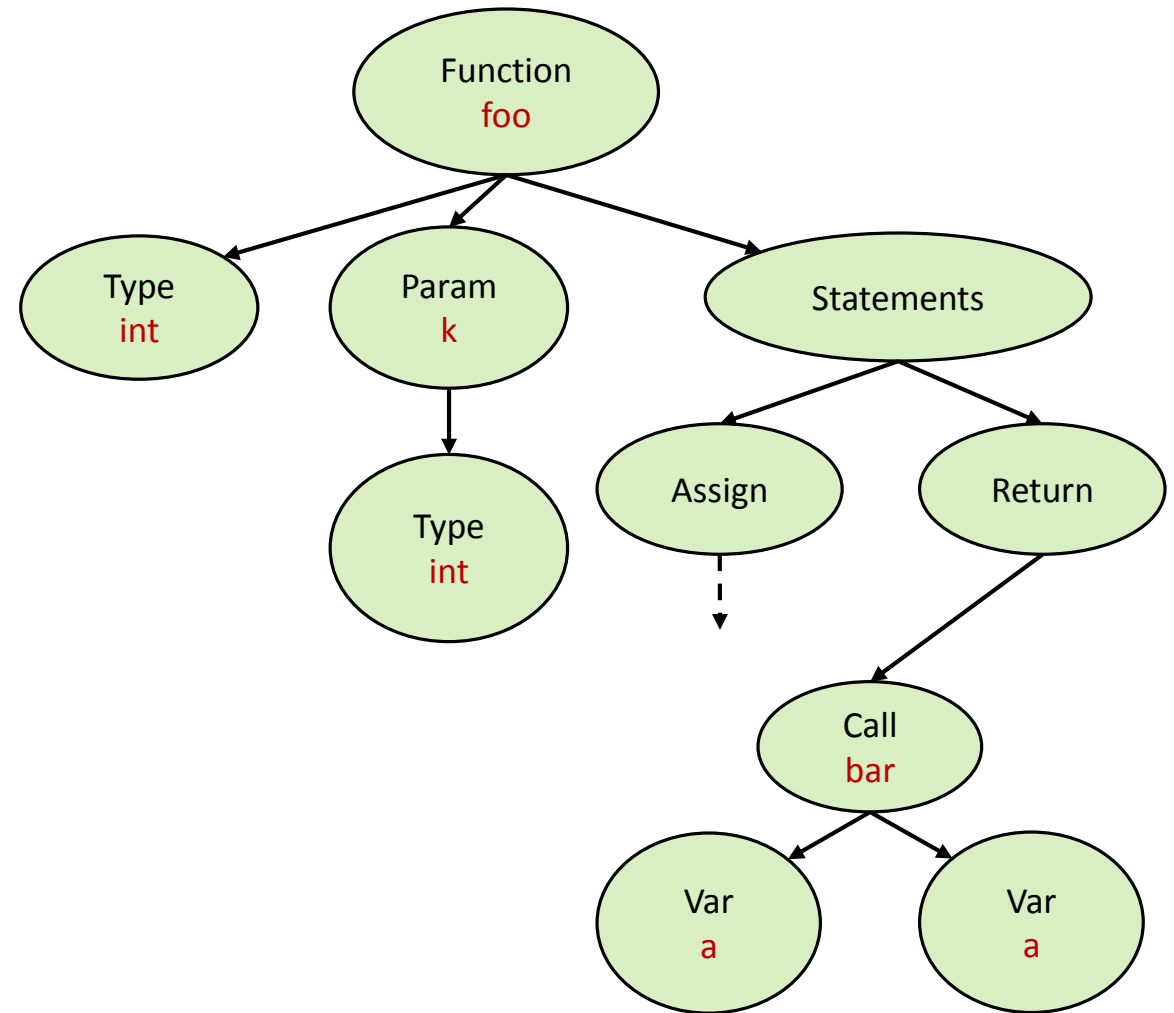
# Function Calls

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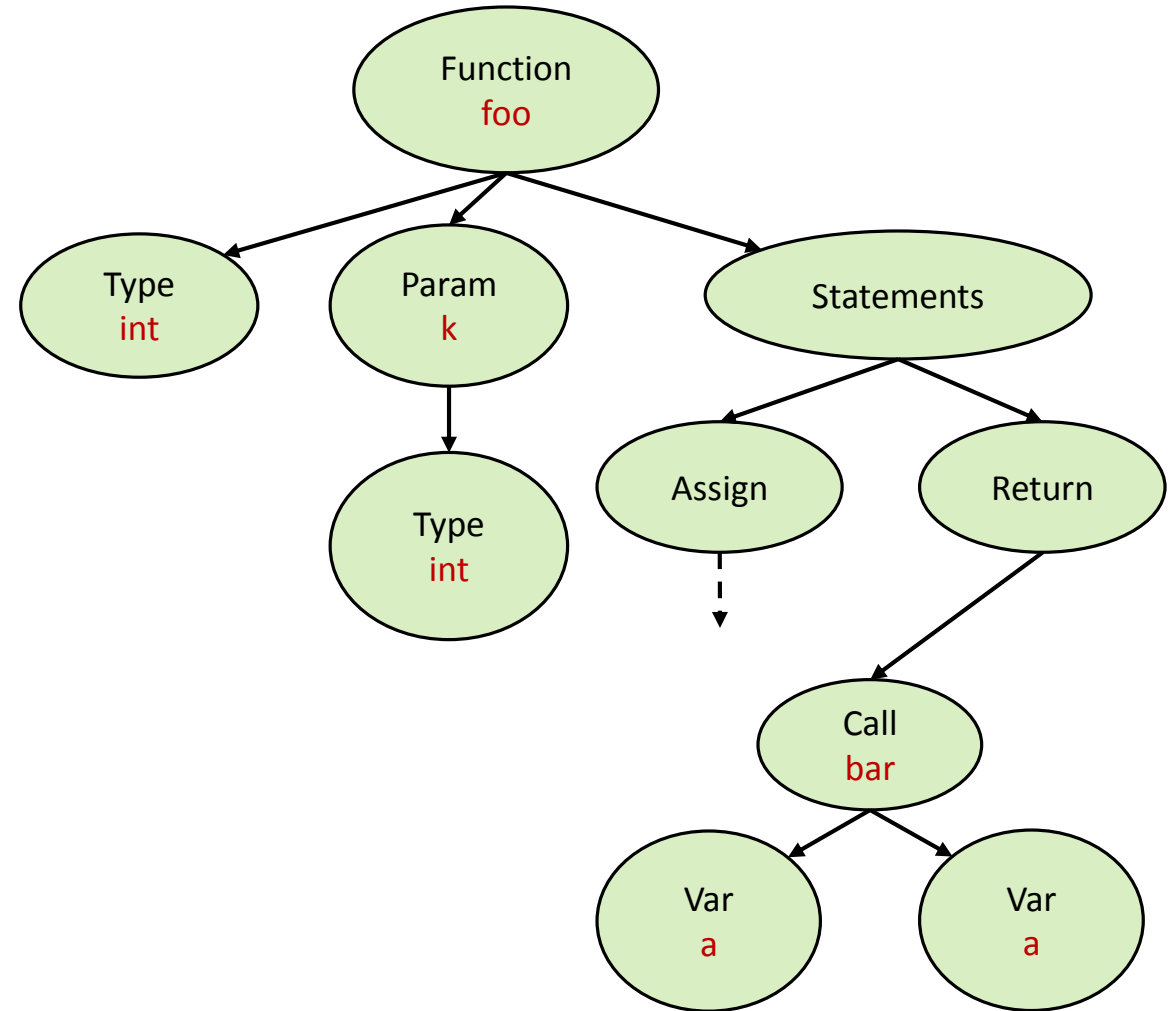
Invalid

# Function Calls

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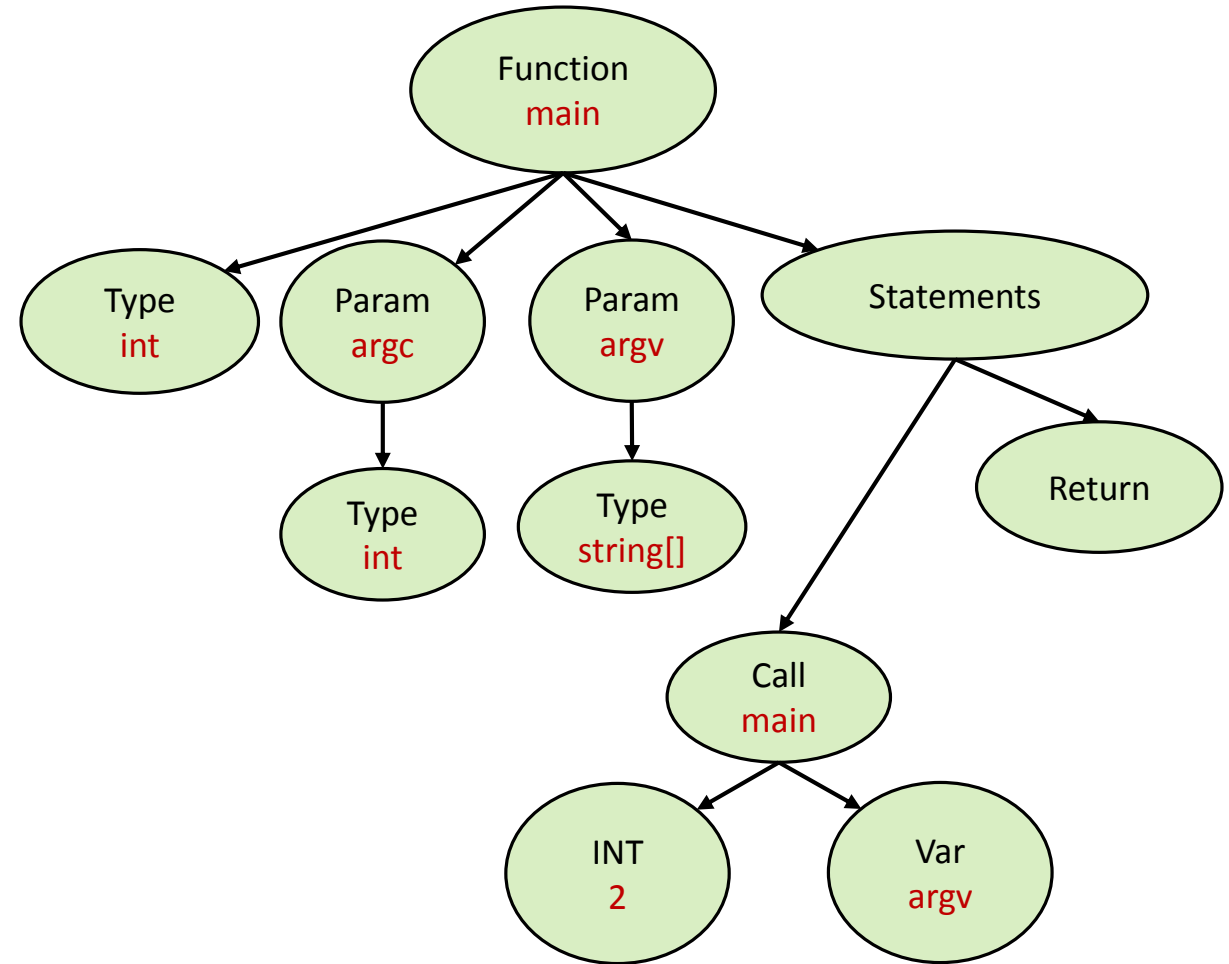
# Function Calls

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

Valid
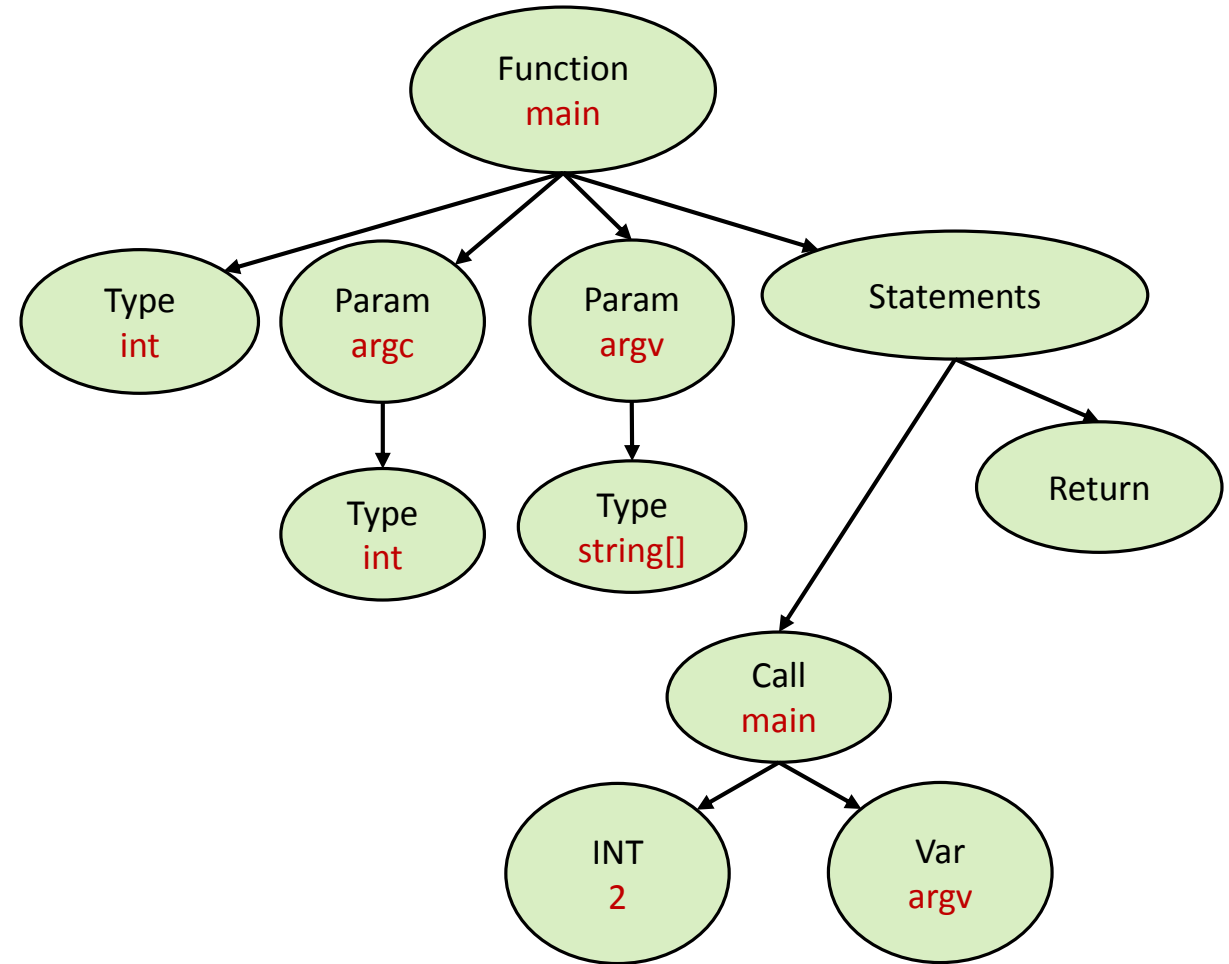
# Arrays

```
void foo(void) {
    int[] arr = new int["z"];
}
```
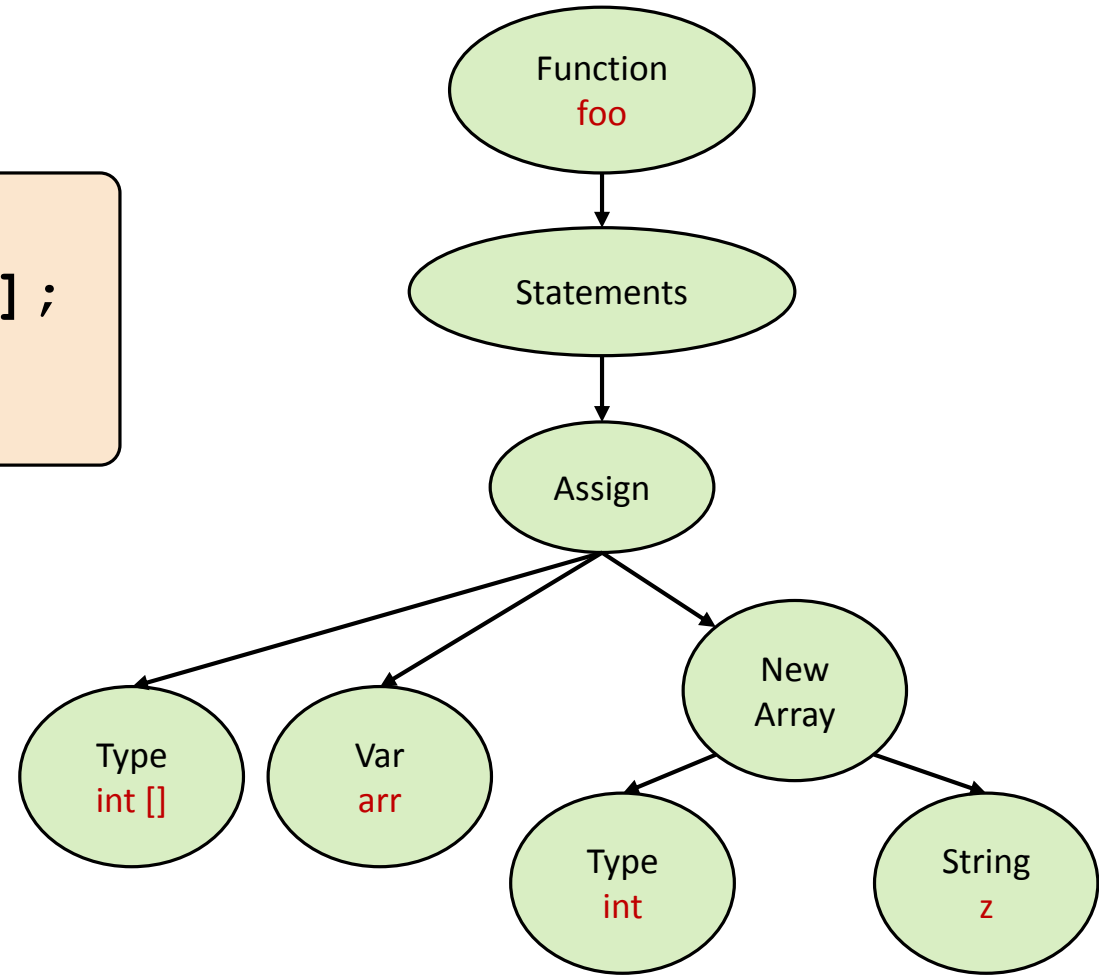
# Arrays

```
void foo(void) {
   int[] arr = new int["z"];
}
```
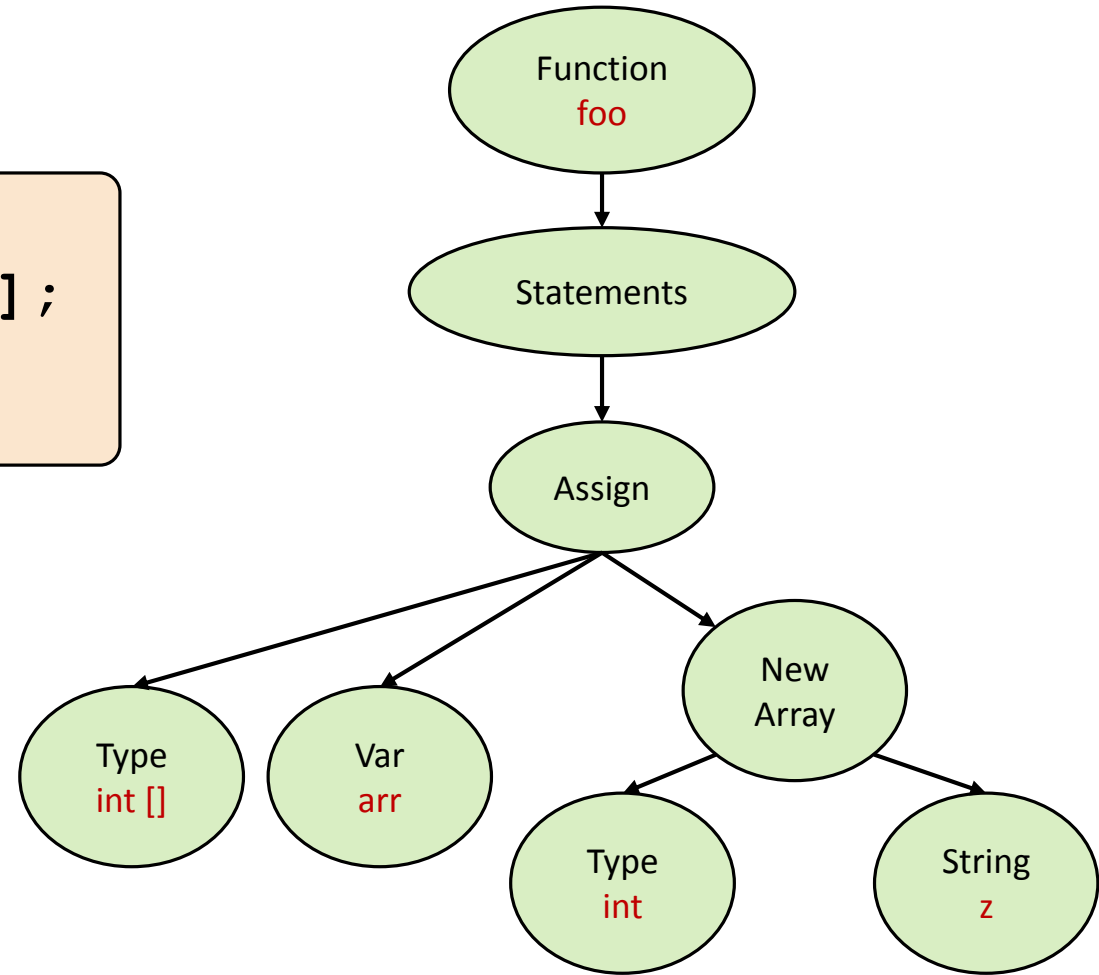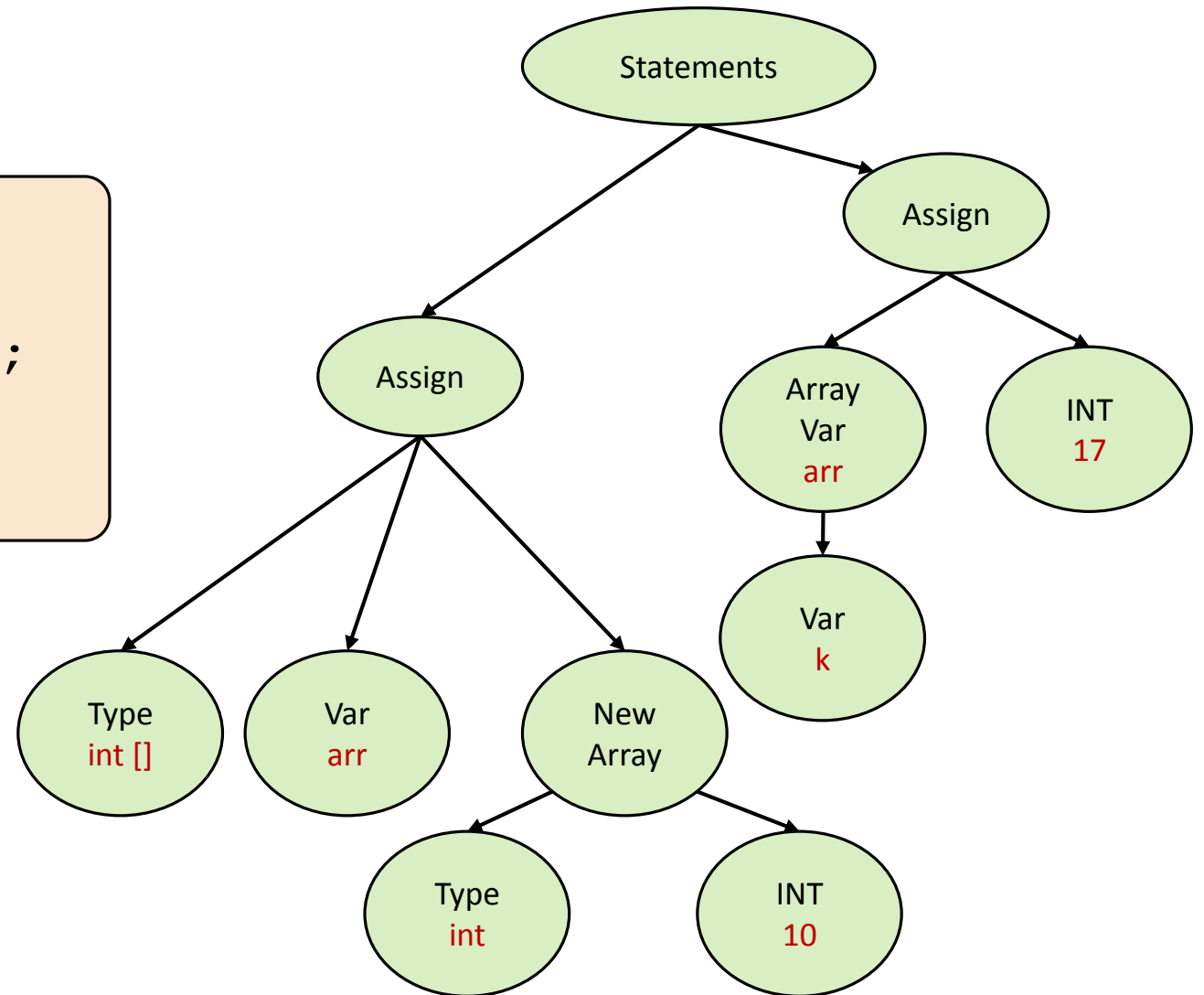
**Invalid**

# Arrays

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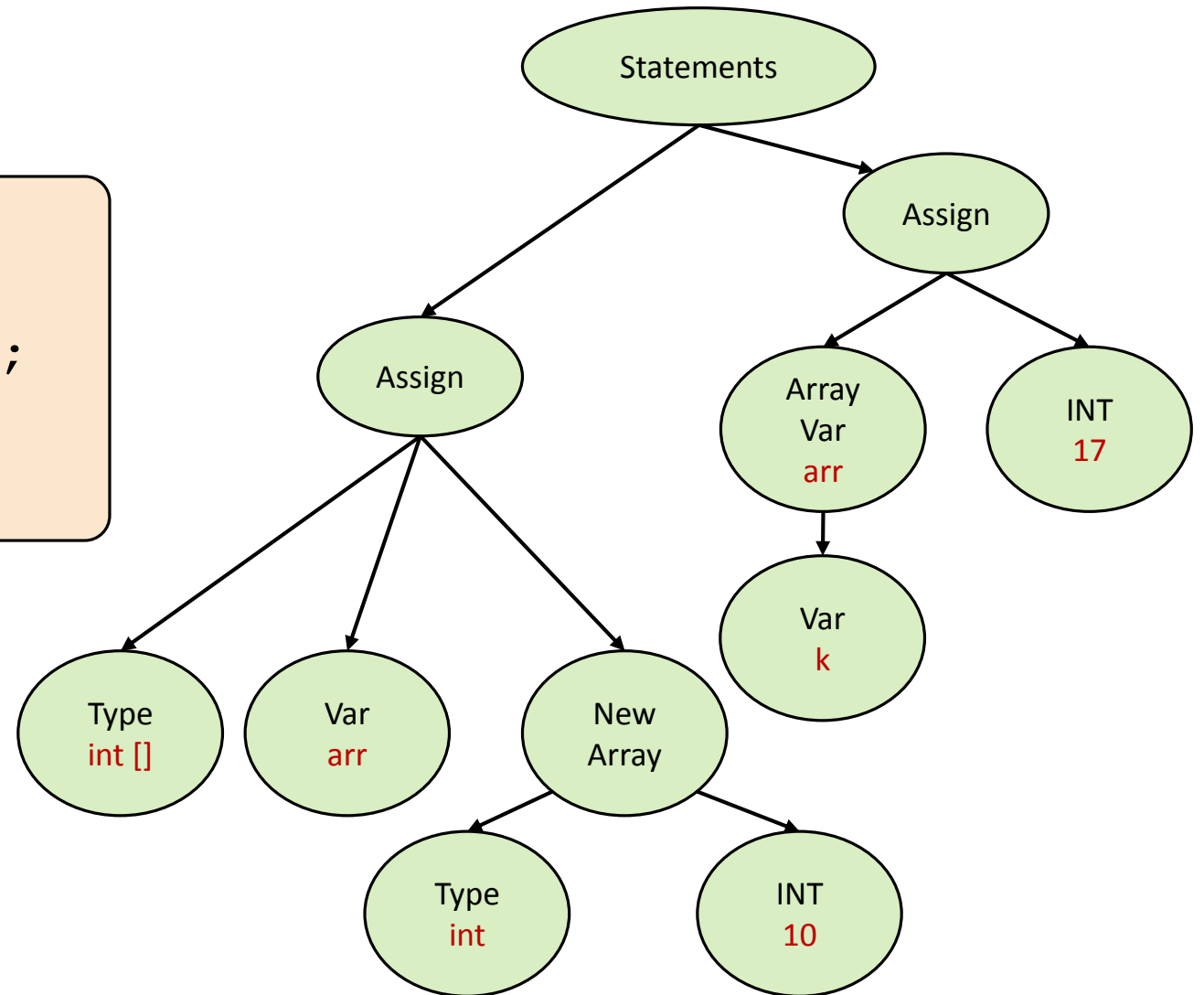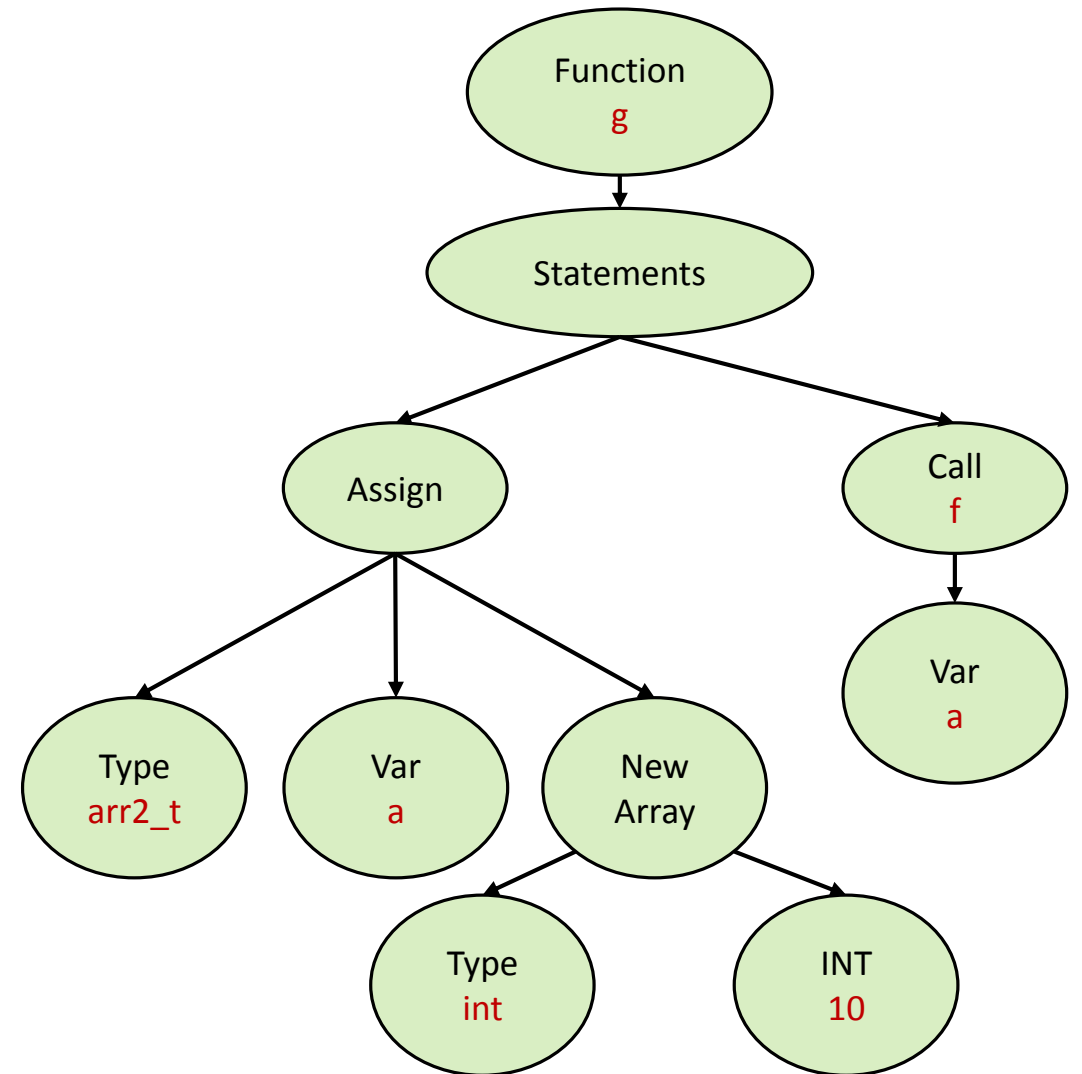
# Arrays

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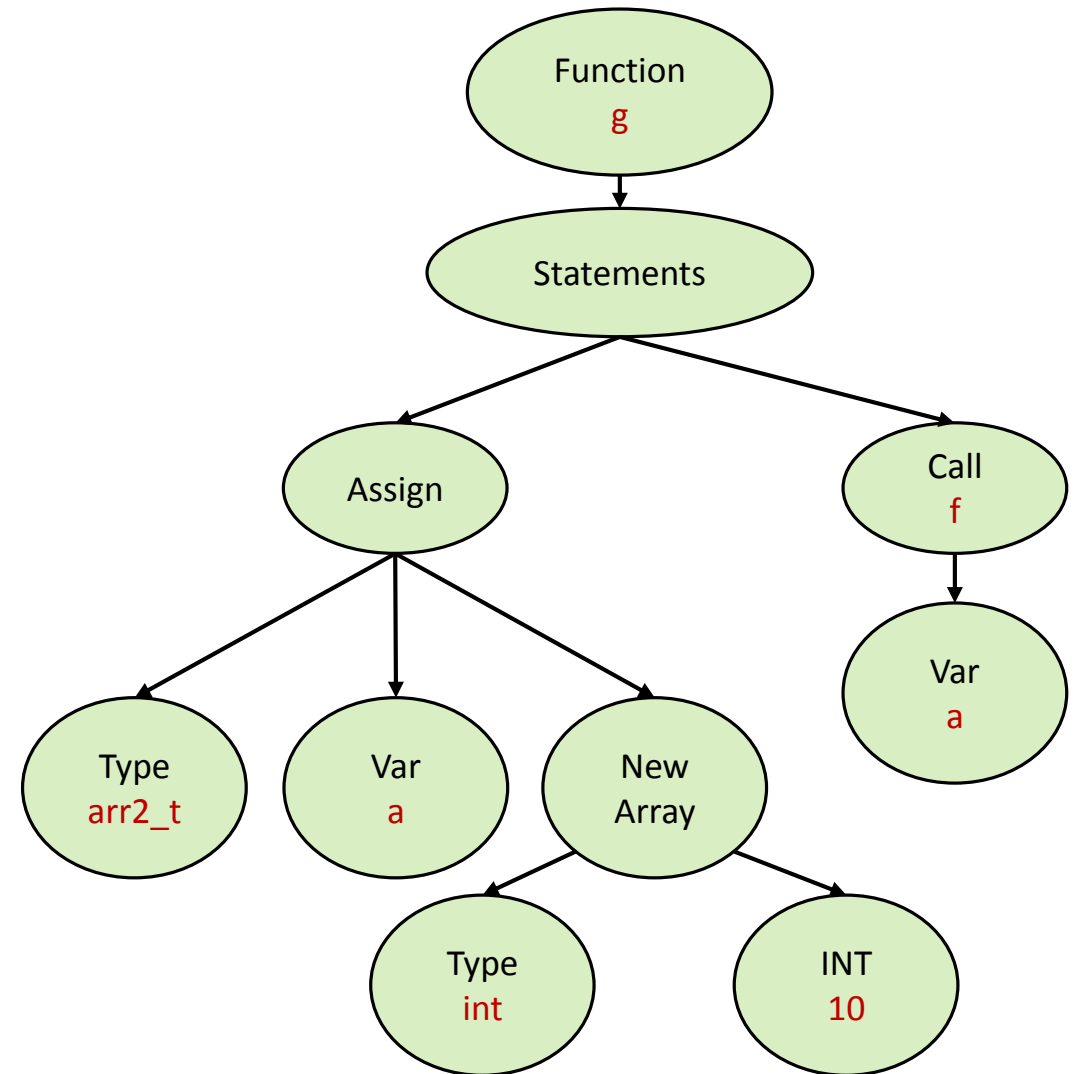
Valid

# Arrays

```
typedef int arr1_t[];
typedef int arr2_t[];
void f(arr1_t a) { }
void g() {
    arr2_t a = new int[10];
    f(a);
}
```
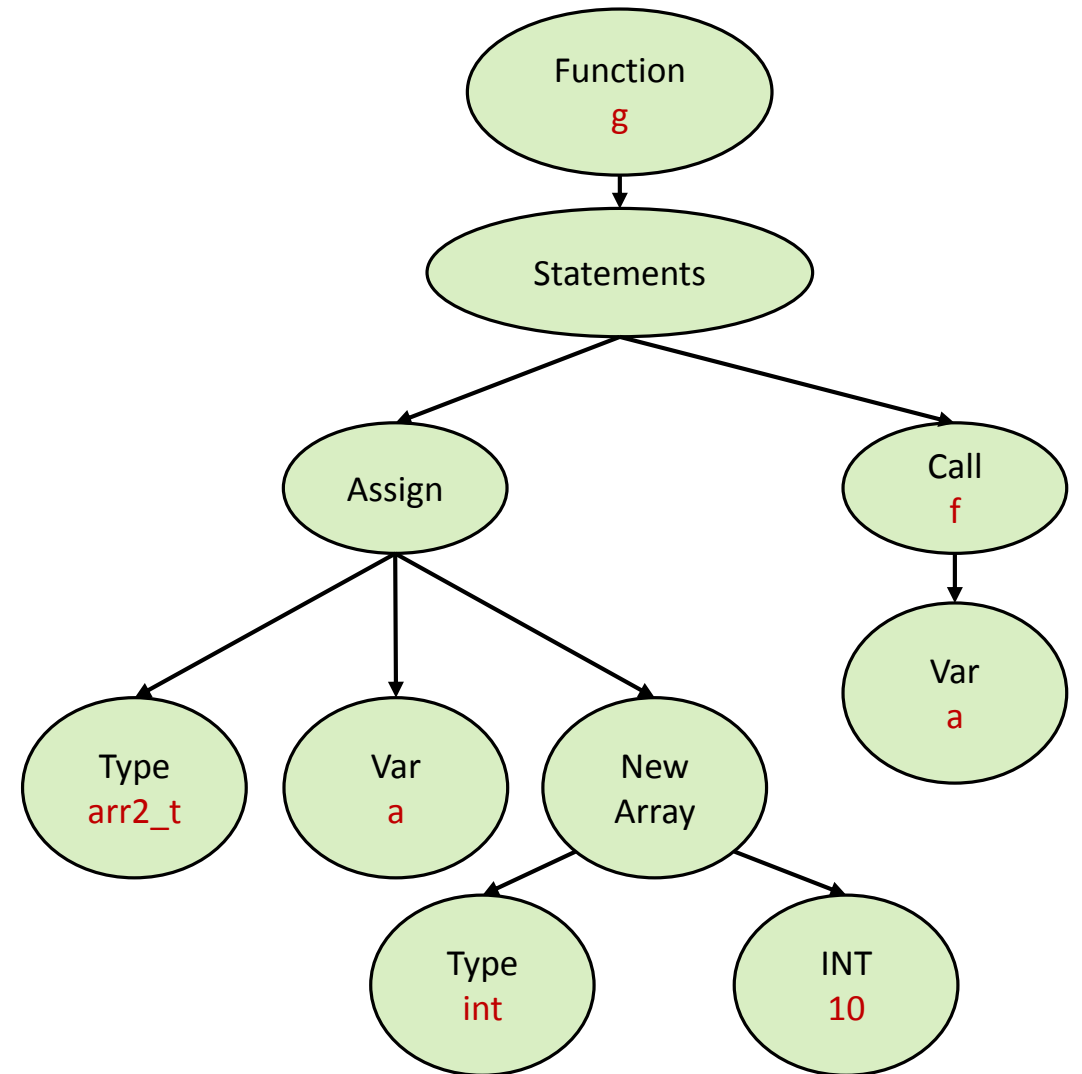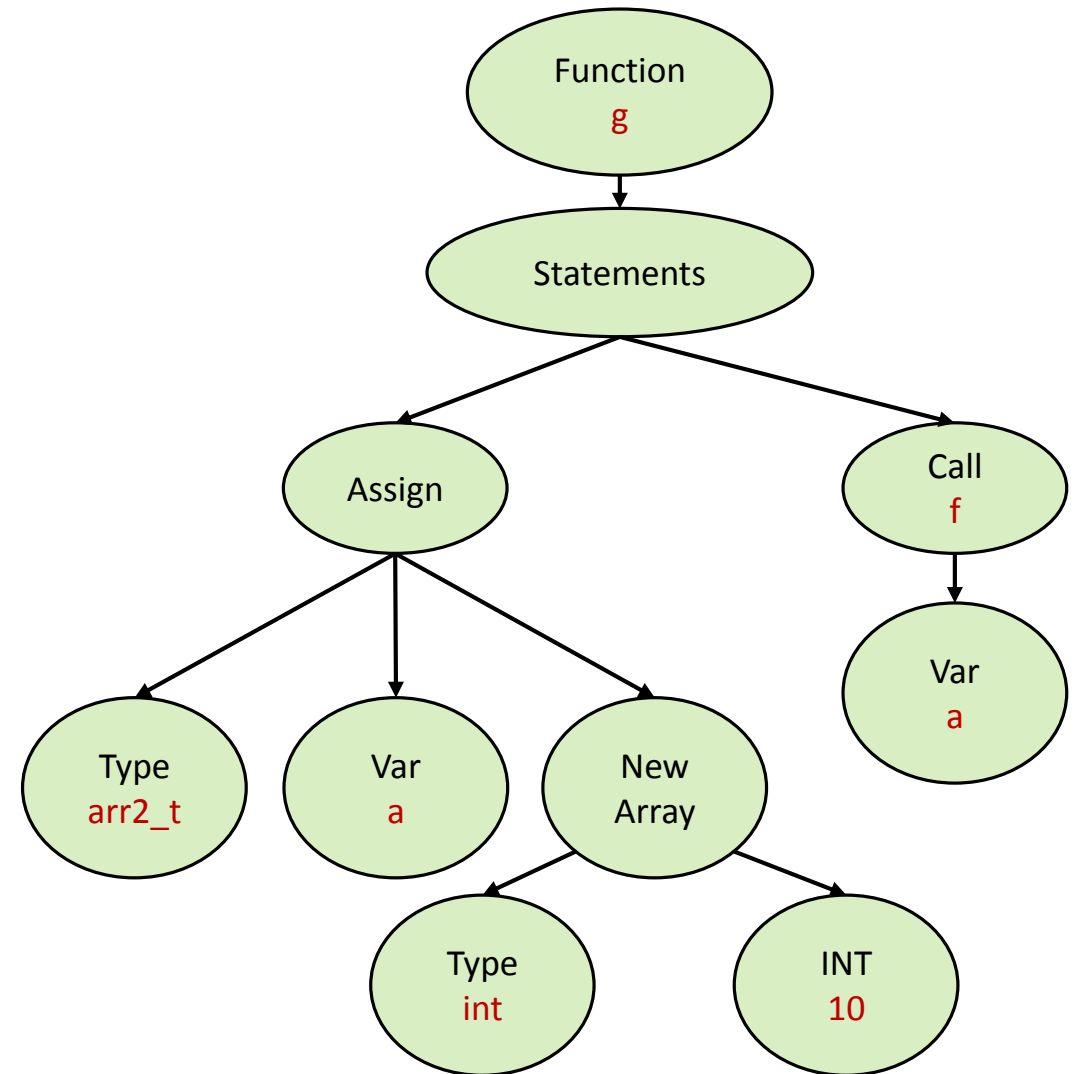
# Arrays

```
typedef int arr1_t[];
typedef int arr2_t[];
void f(arr1_t a) { }
void g() {
    arr2_t a = new int[10];
    f(a);
}
```

Invalid

# Arrays

```
typedef int arr1_t[];
typedef int arr2_t[];
void f(arr1_t a) { }
void g() {
    arr2_t a = new int[10];
    f(a);
}
```
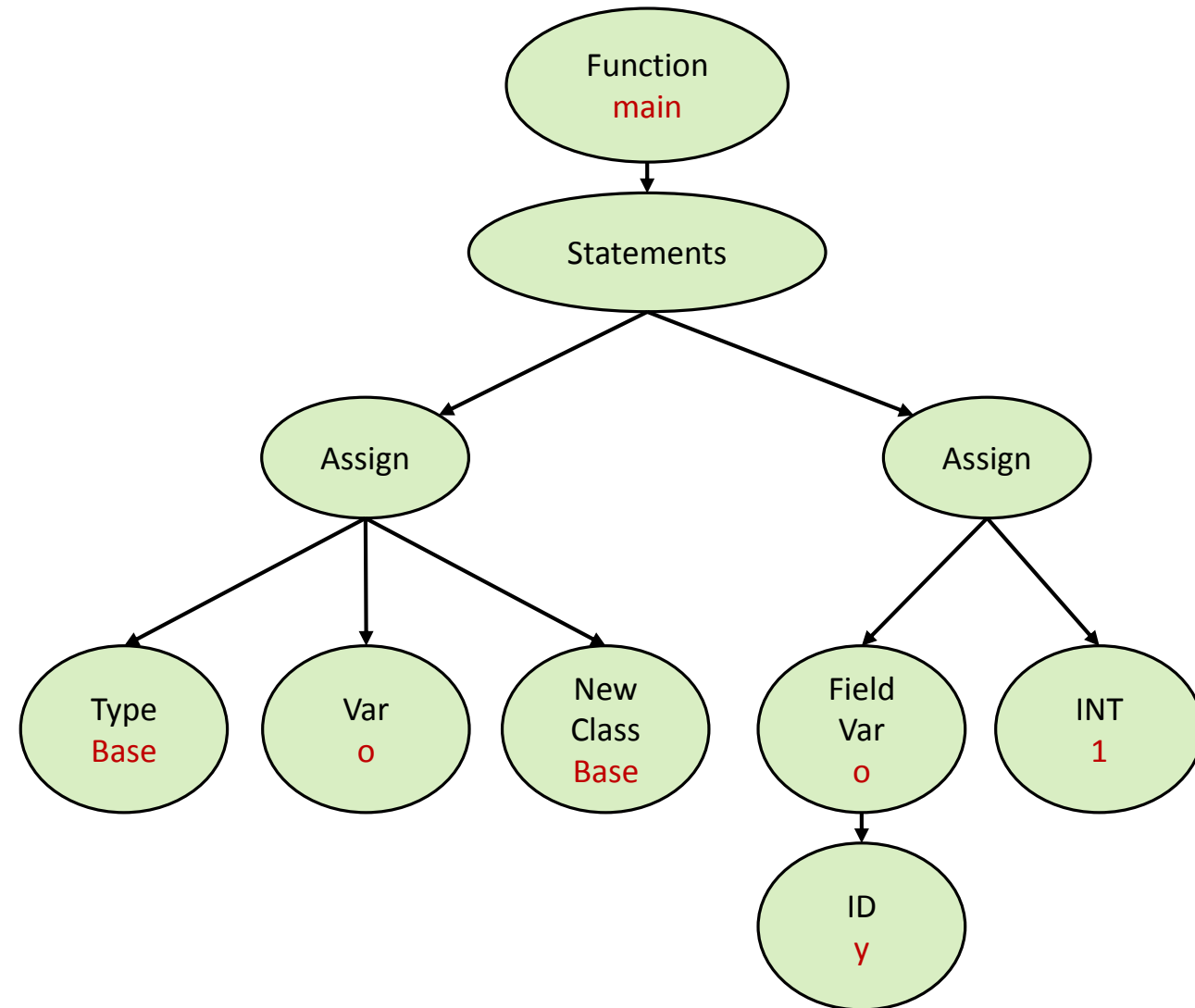
# Arrays

```
typedef int arr1_t[];
typedef int arr2_t[];
void f(arr1_t a) { }
void g() {
    arr2_t a = new int[10];
    f(a);
}
```
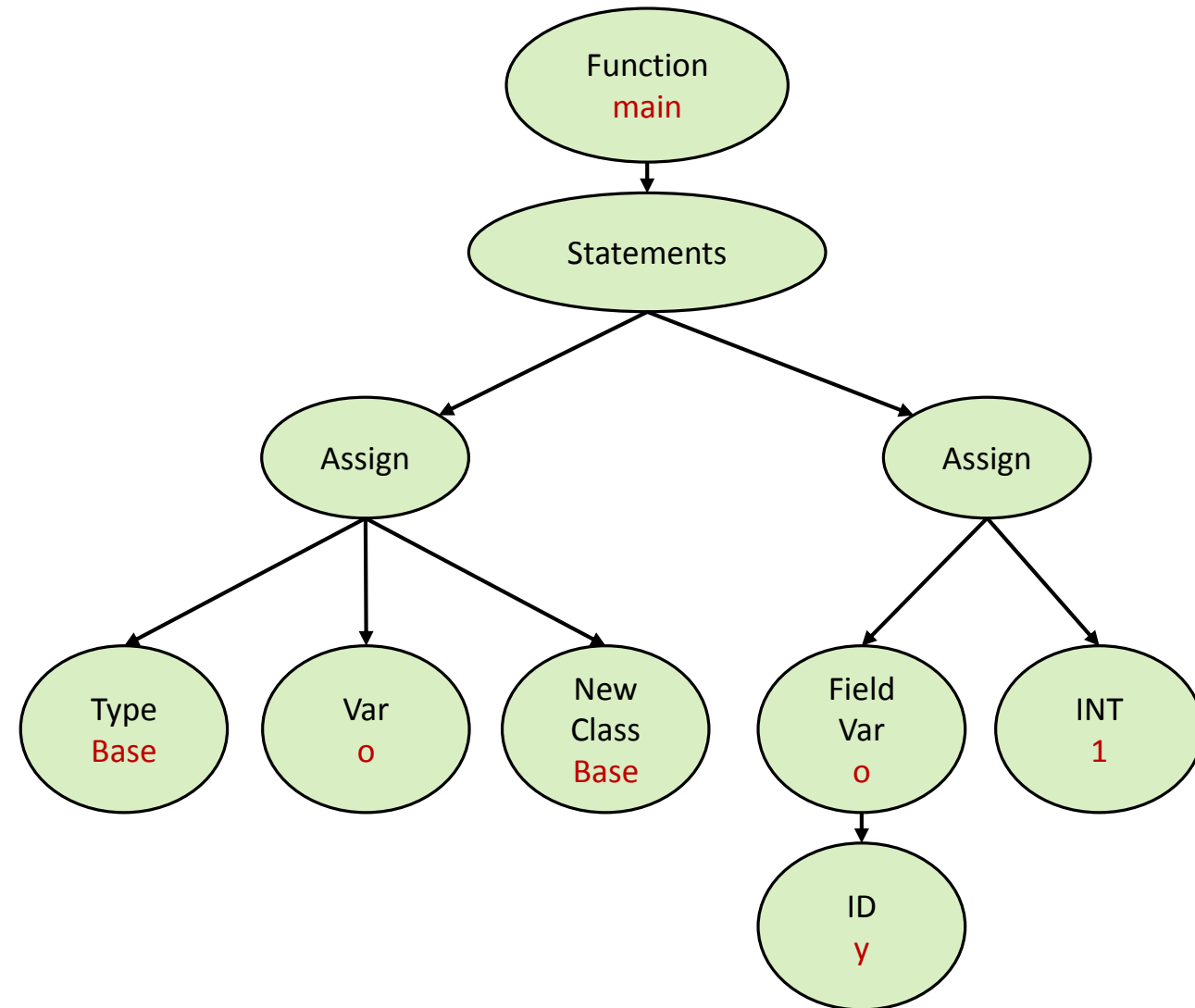
**Invalid**

# Classes

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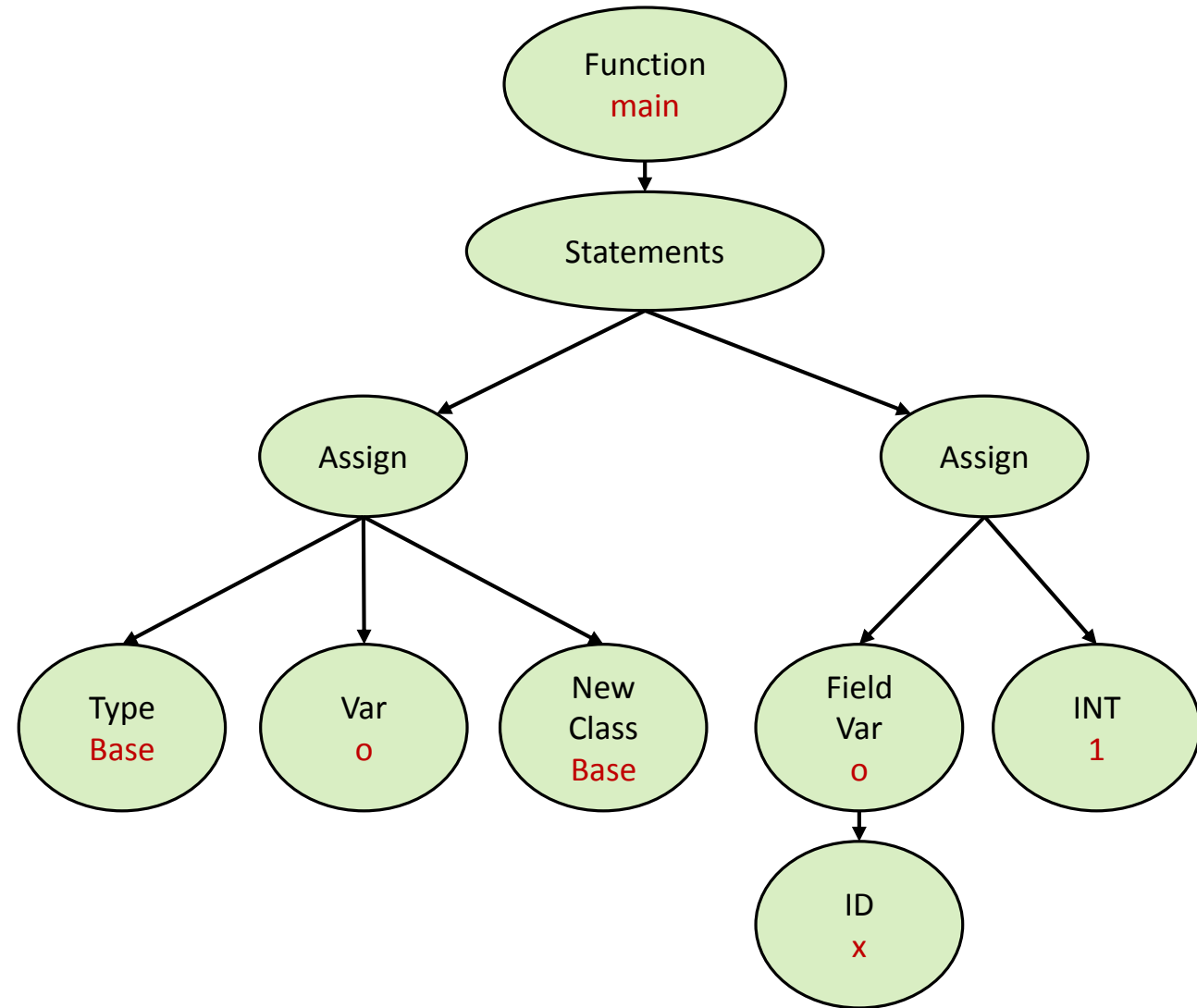
# Classes

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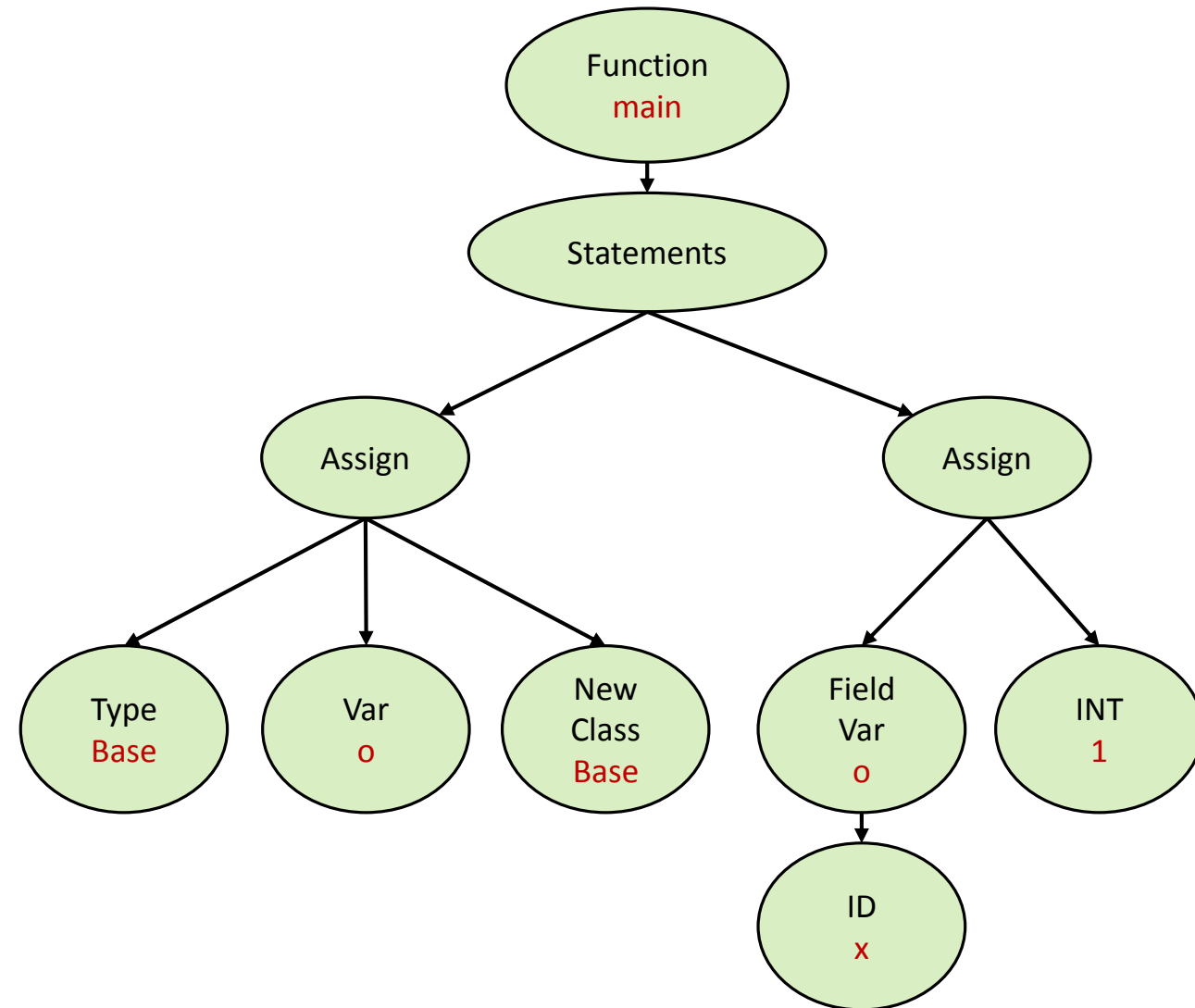
Invalid

# Classes

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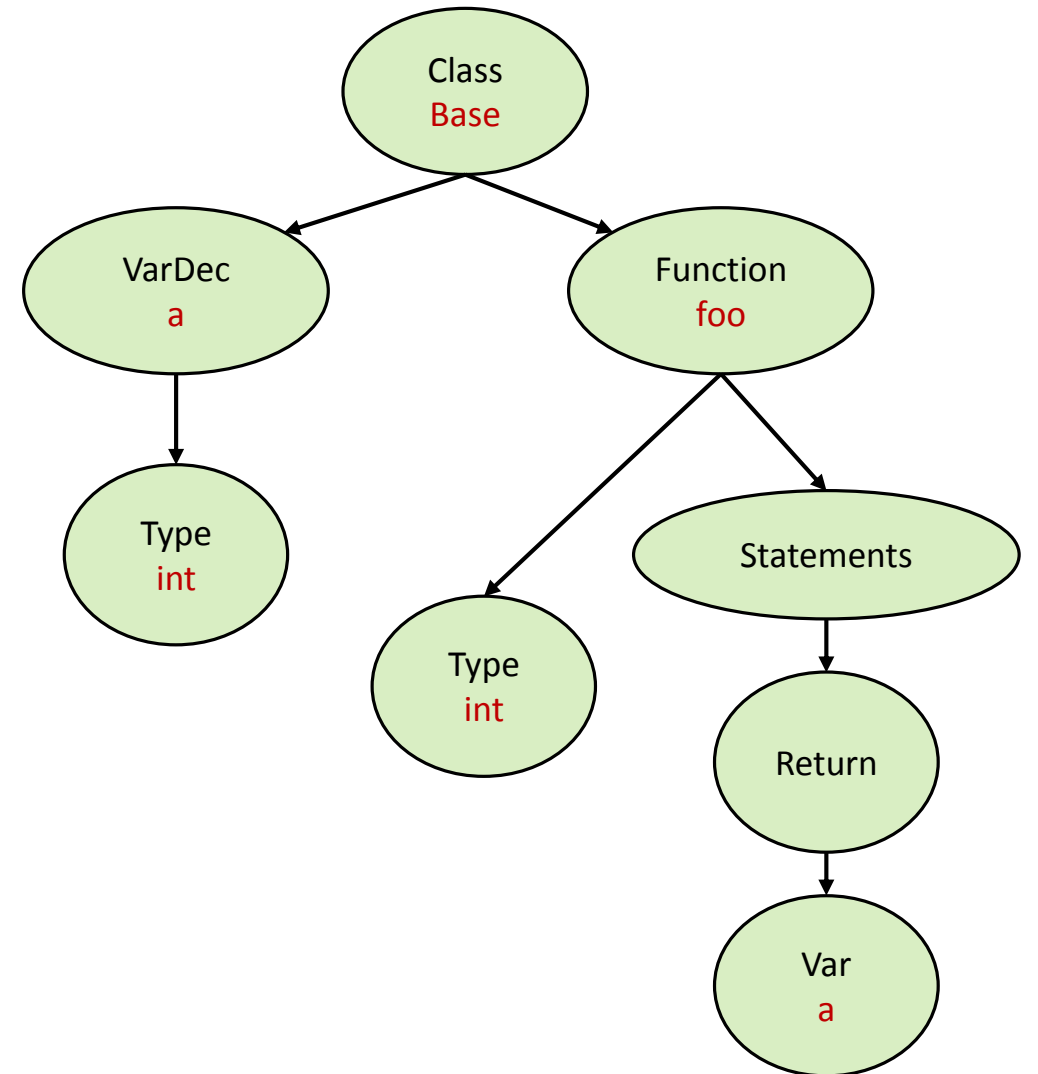
# Classes

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

Valid

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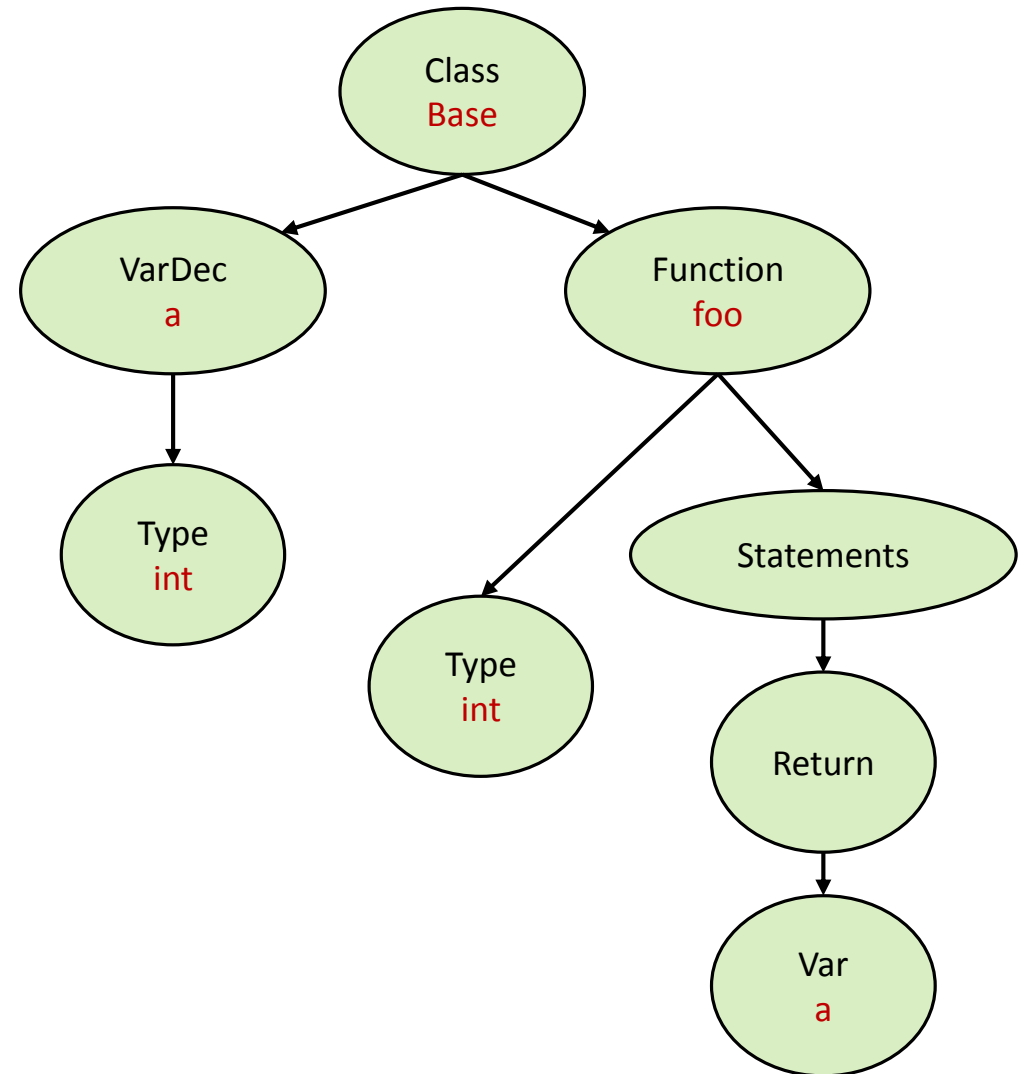
```
class Base {
  int a;
  int foo() {
    return a;
  }
}
```
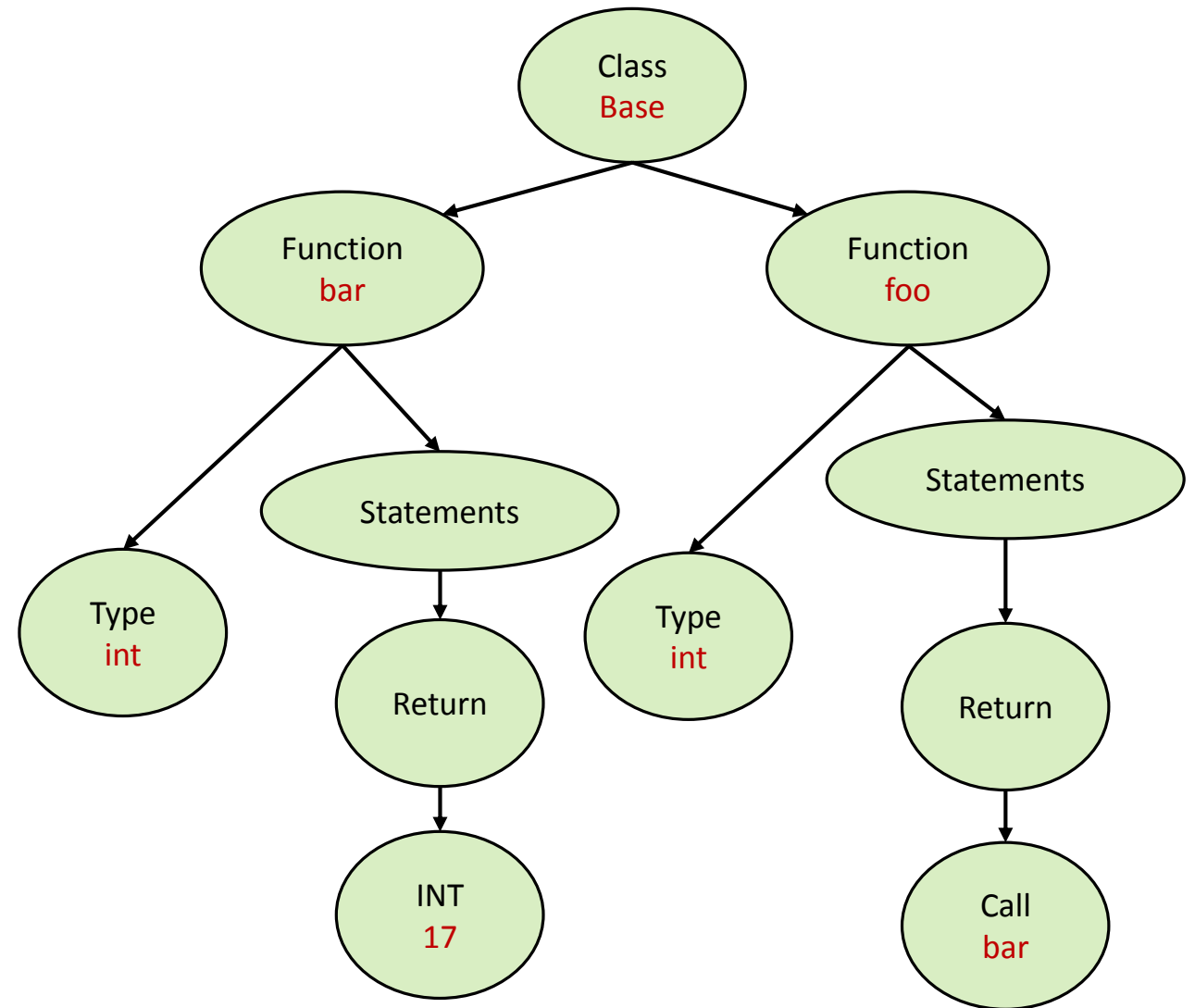
# Classes

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

Valid

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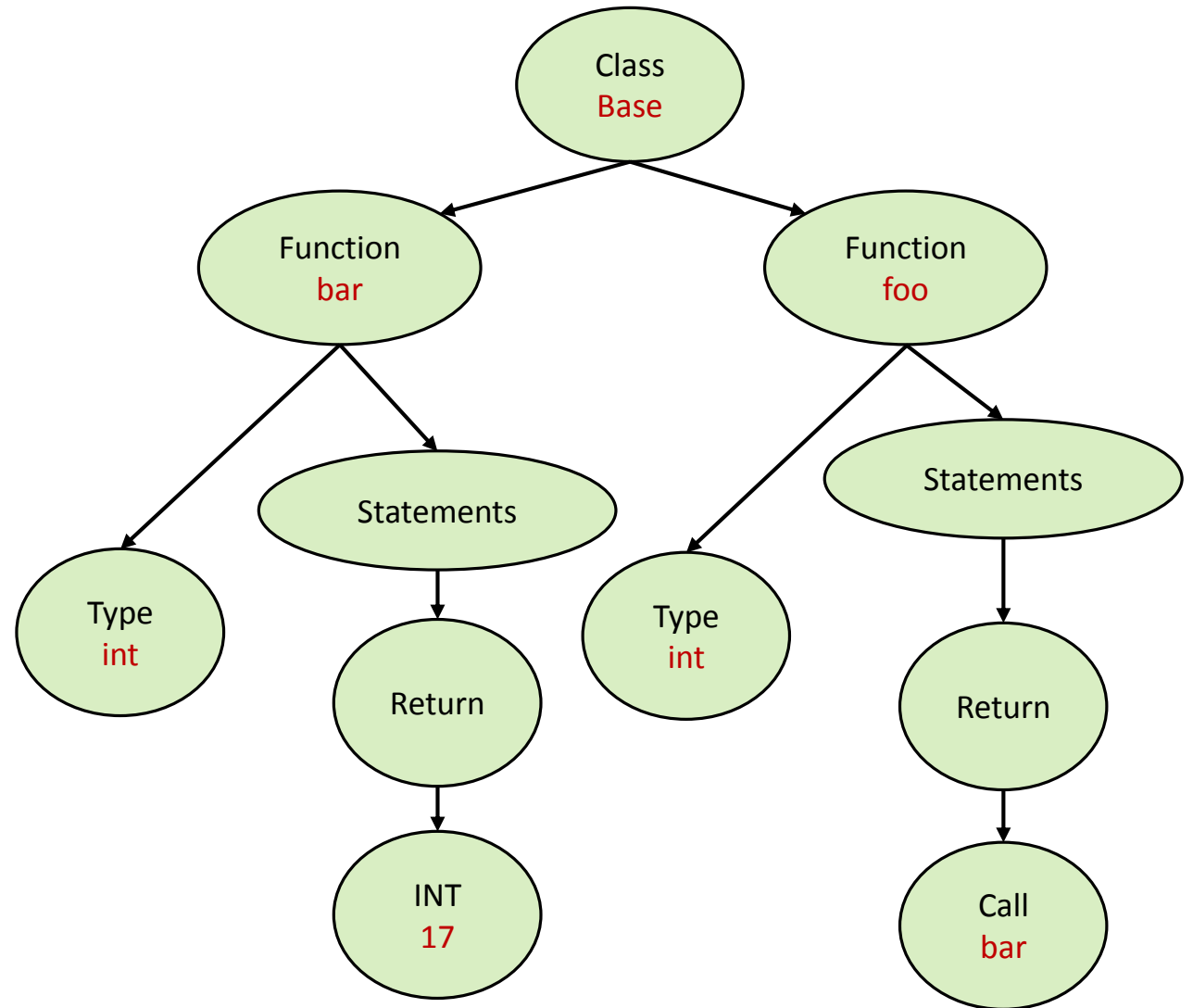
```
class Base {
  int bar() {
    return 17;
  }
  int foo() {
    return bar();
  }
}
```
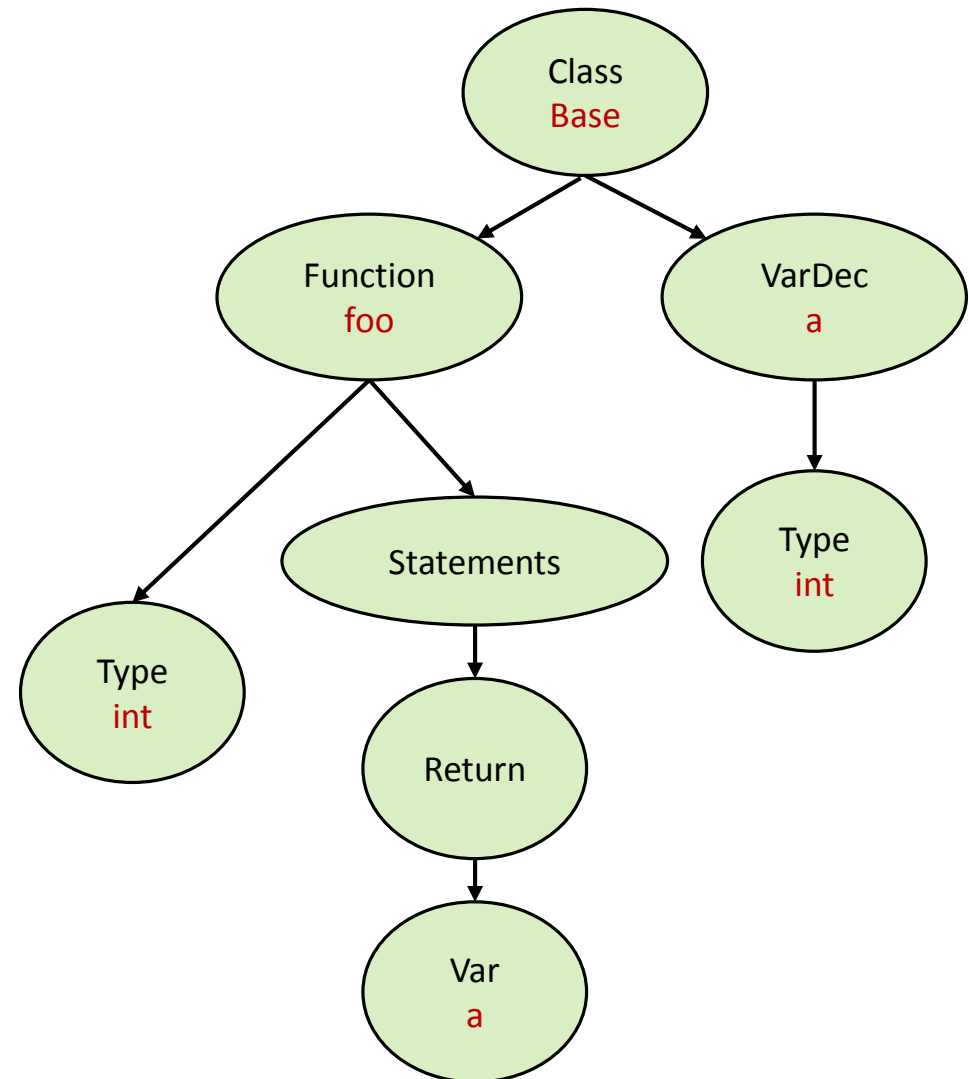
# Classes

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

Valid

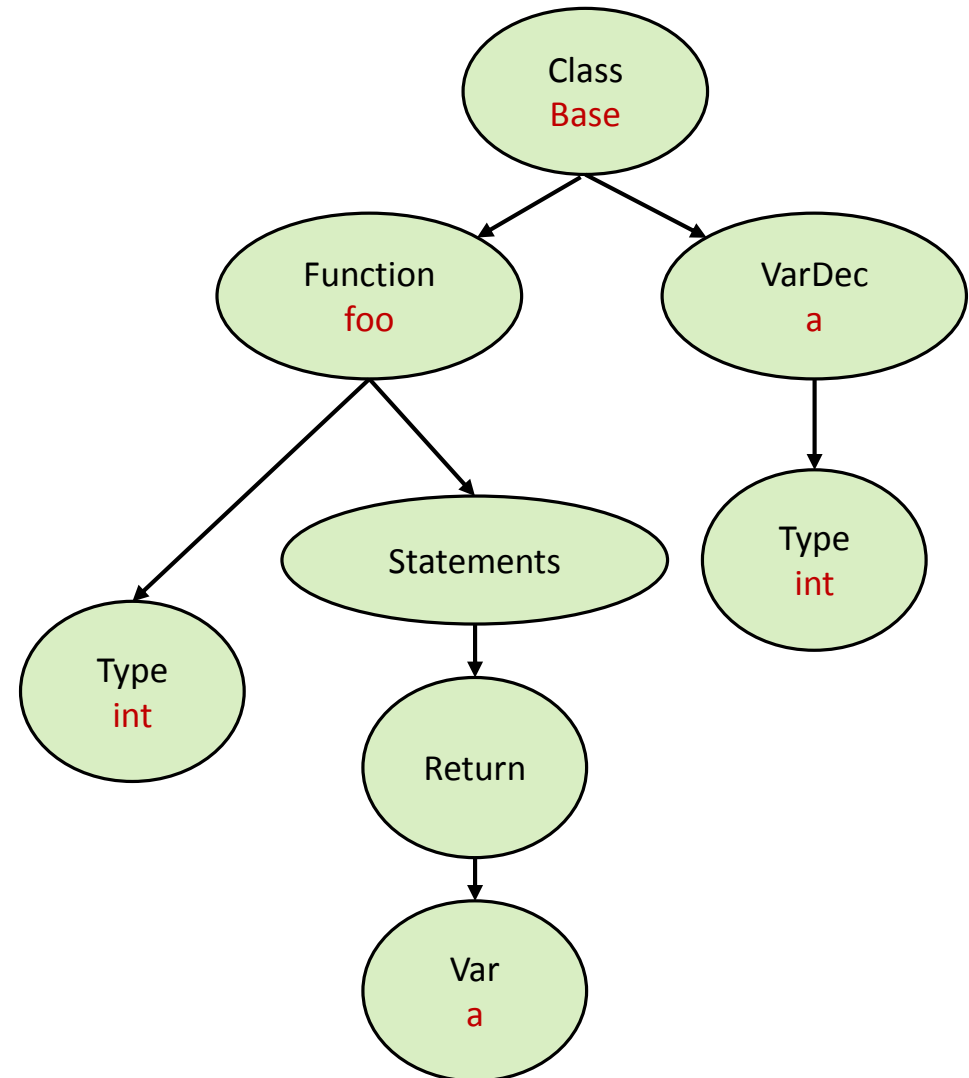# Classes

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

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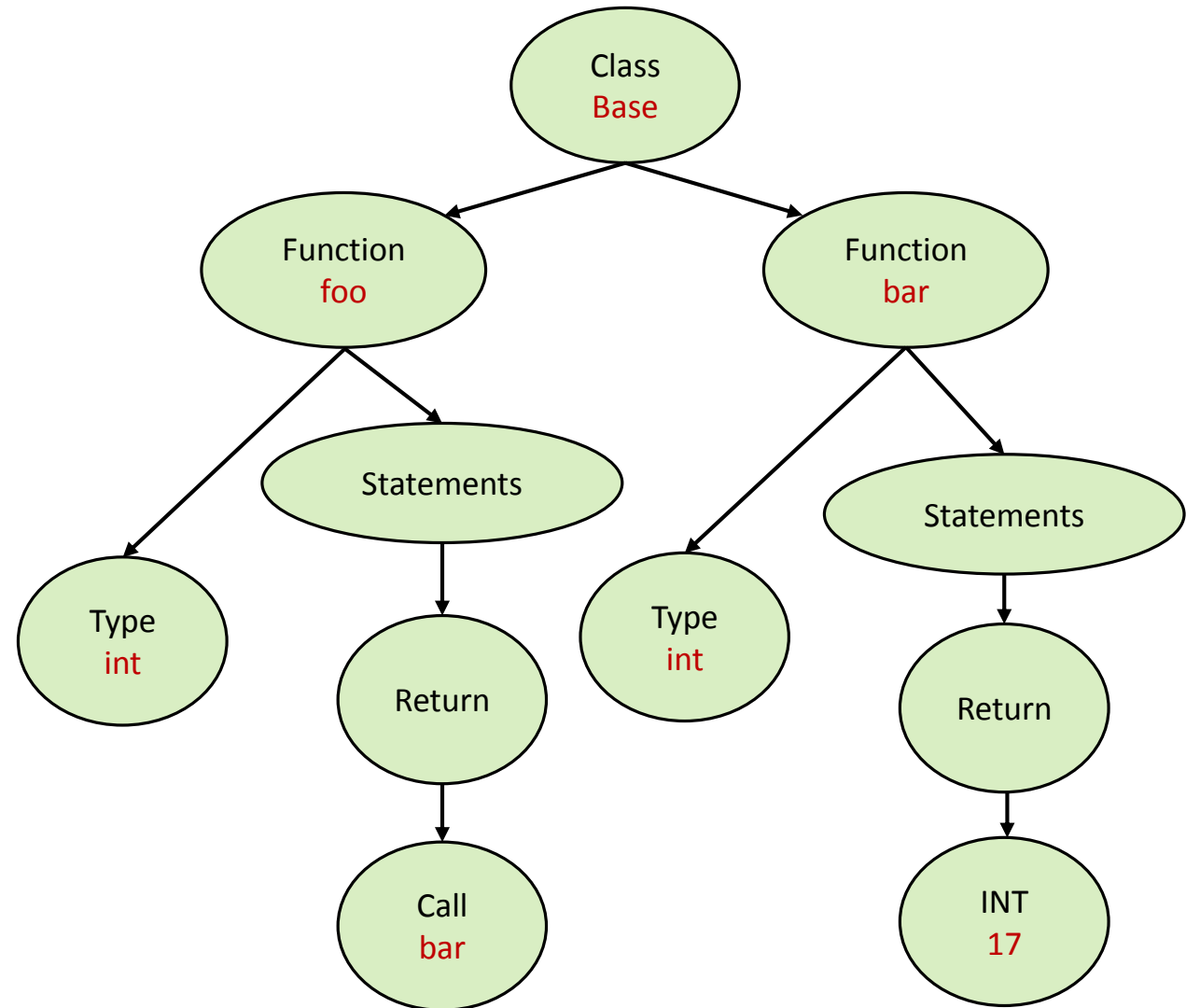
```
class Base {
  int foo() {
    return a;
  }
  int a;
}
```
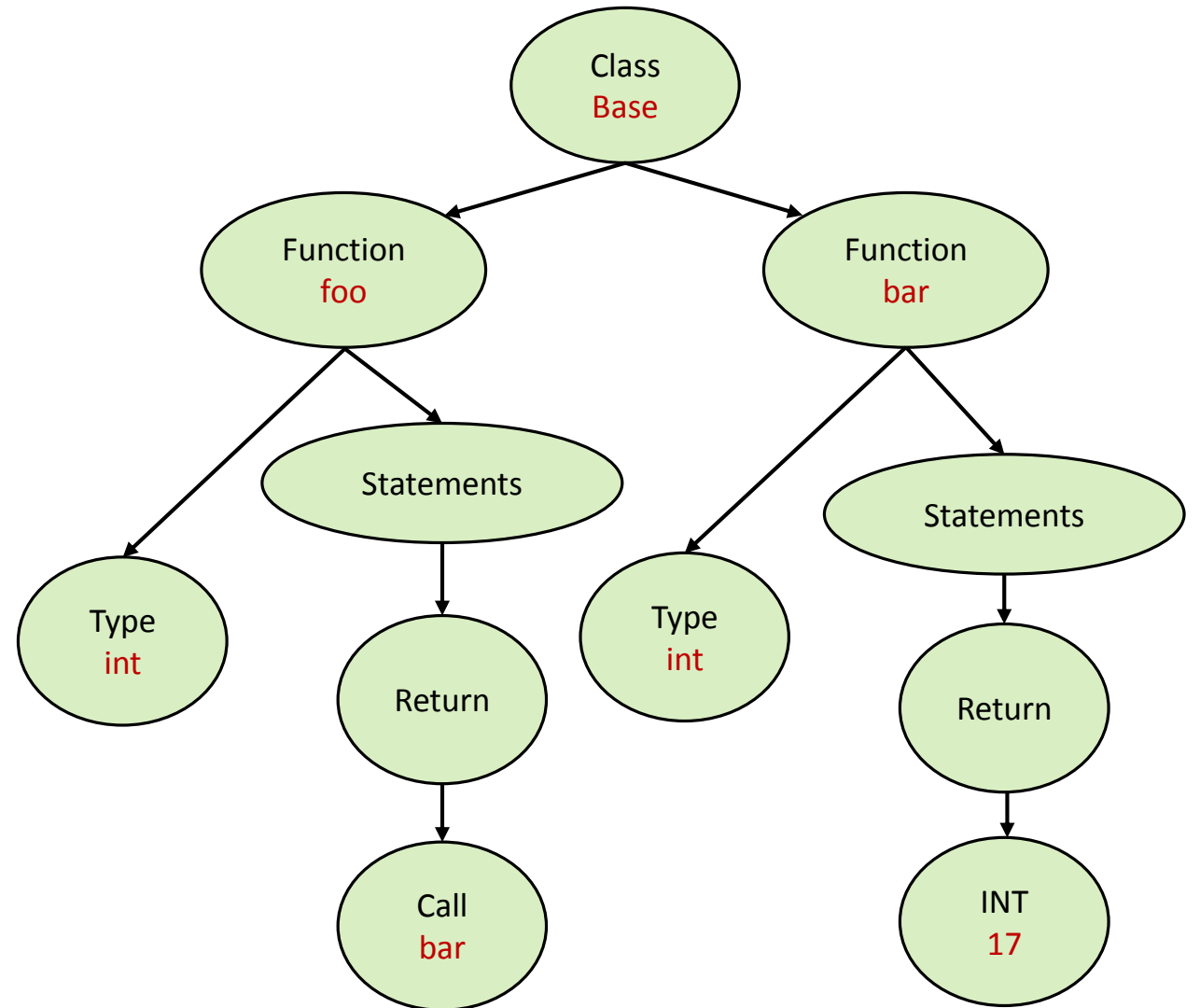
Invalid

# Classes

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

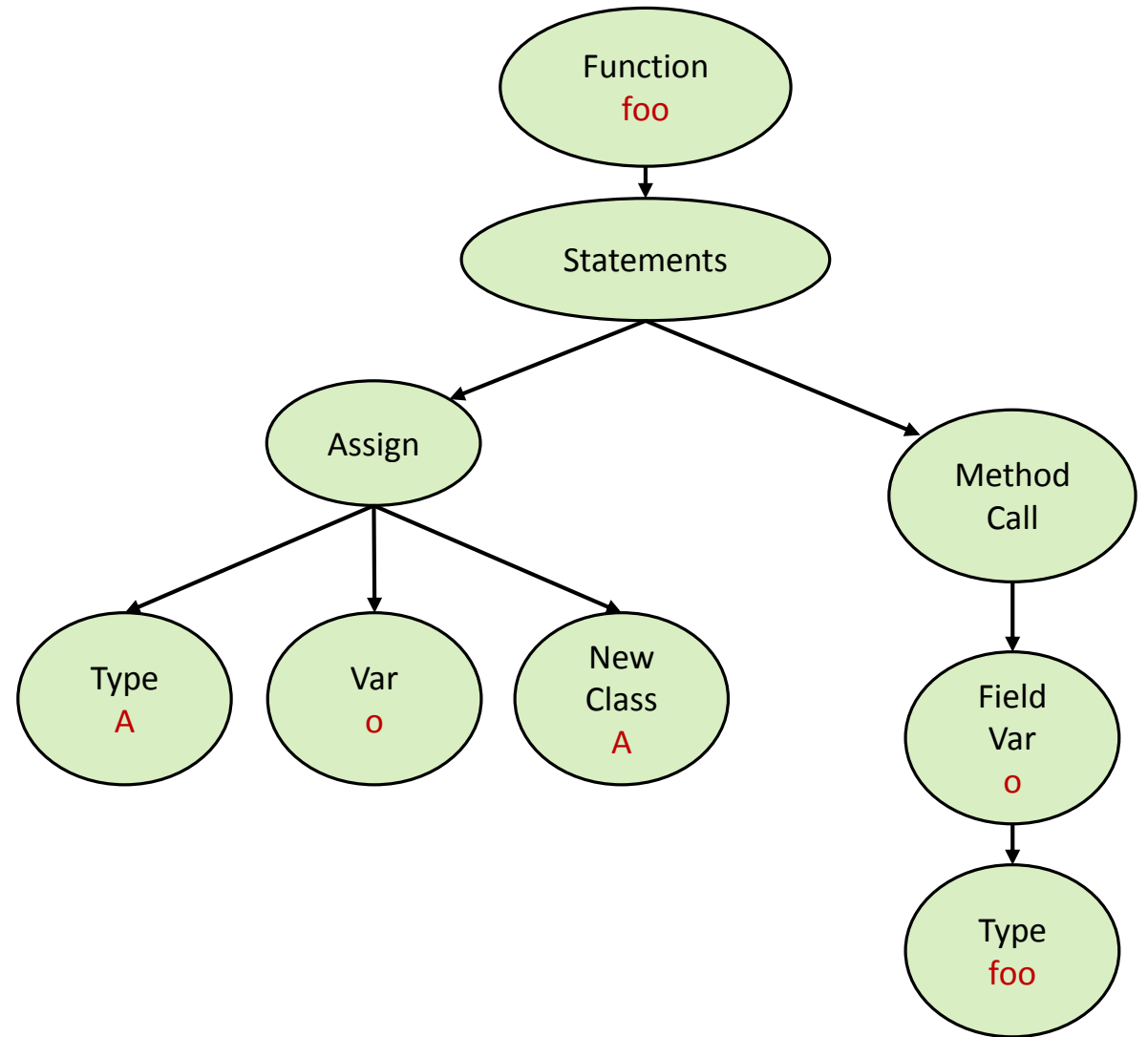# Classes

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

Invalid

# Classes

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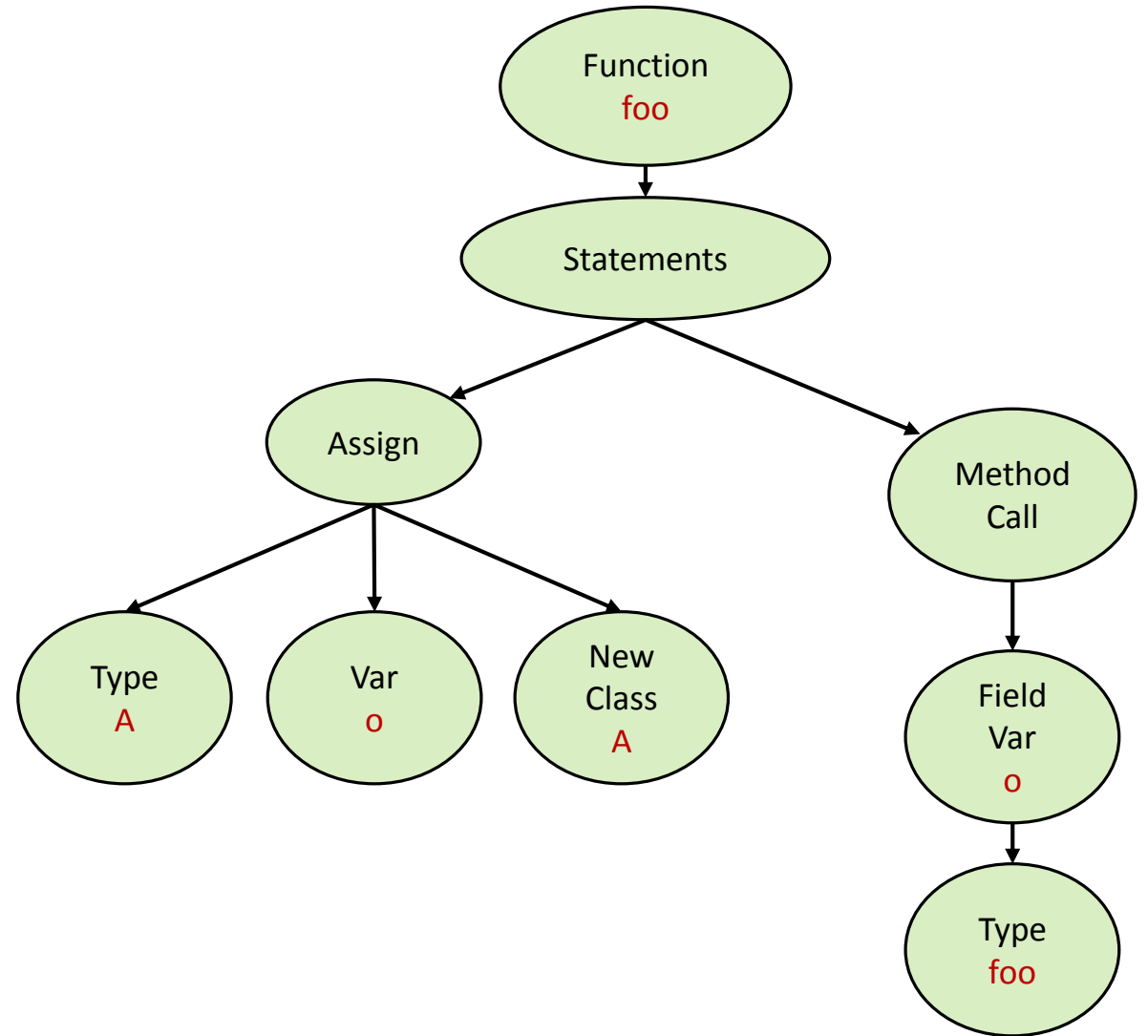
# Classes

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

**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;
}
```
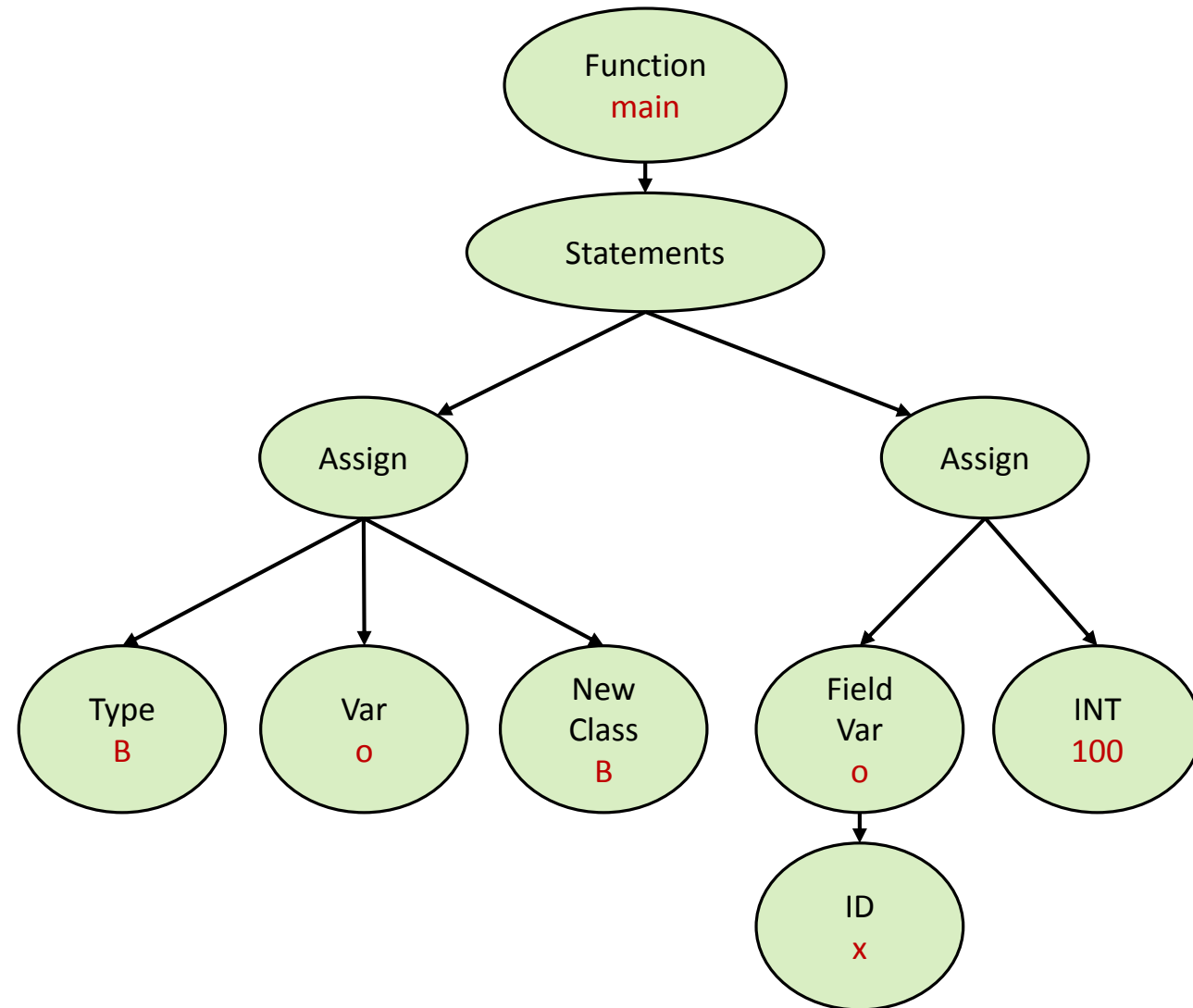
Valid

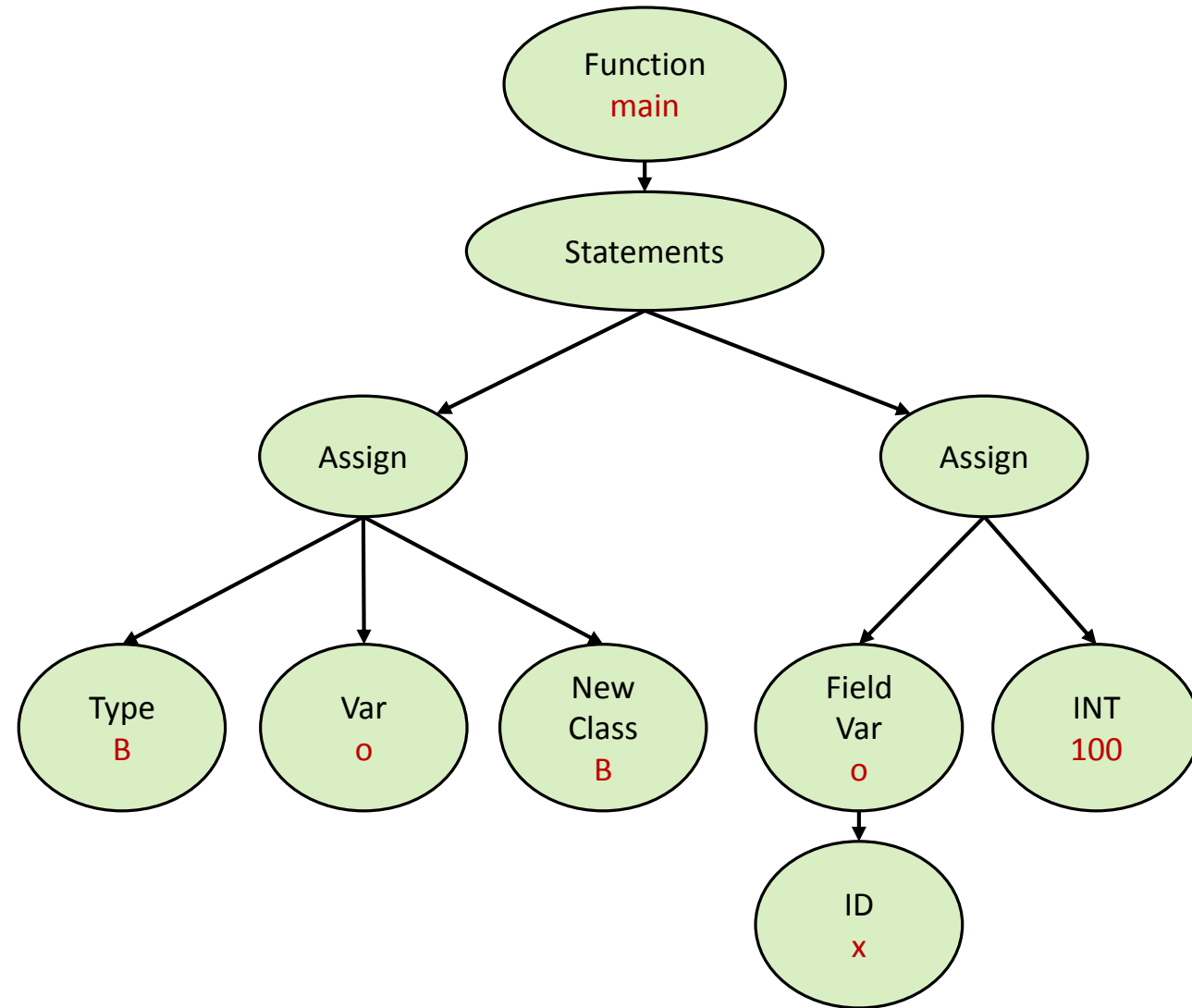# Inheritance

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

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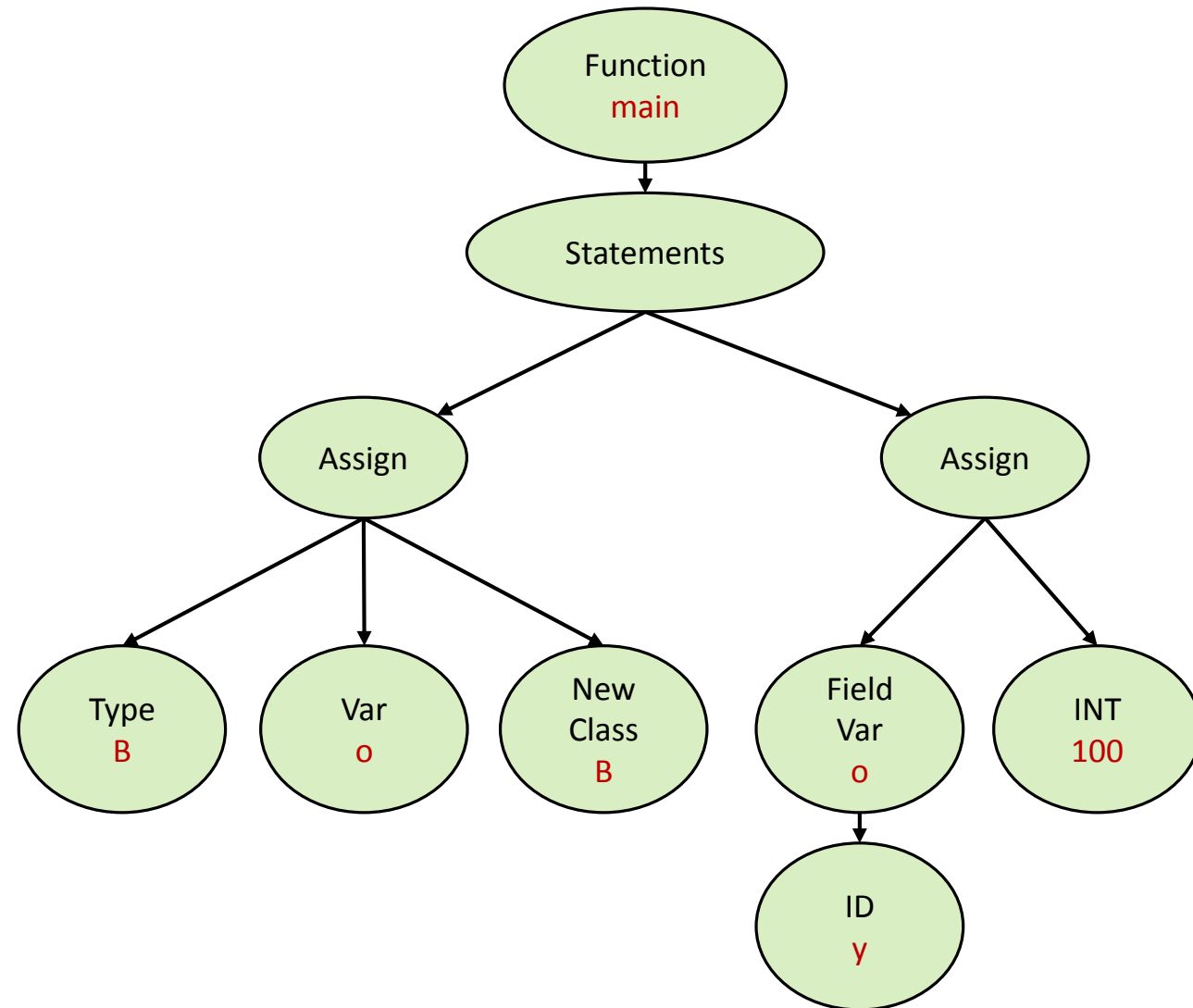
```
class A {
    int x;
}
class B extends A { }
void main() {
    B o = new B;
    o.y = 100;
}
```

## Invalid
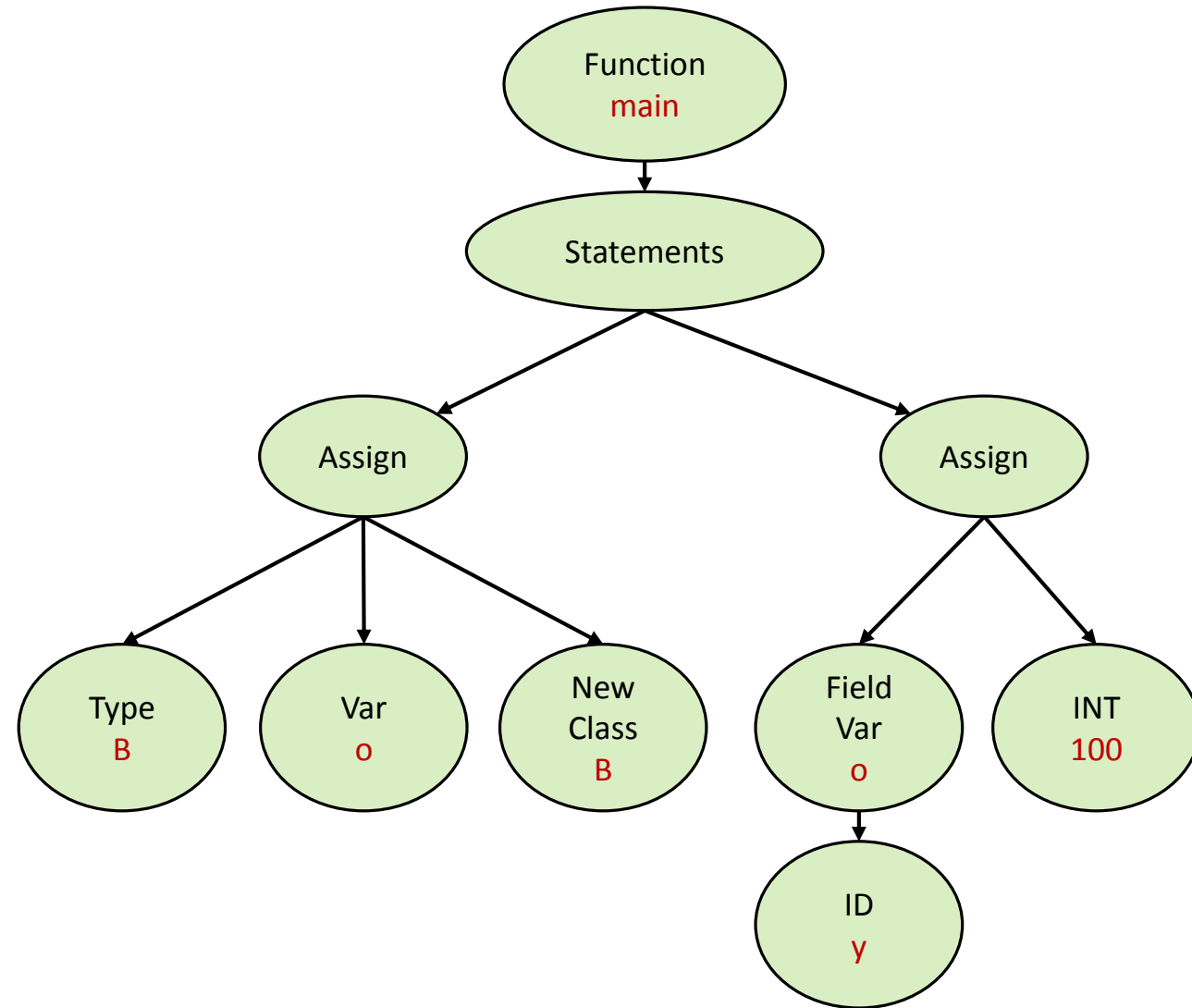
# Inheritance

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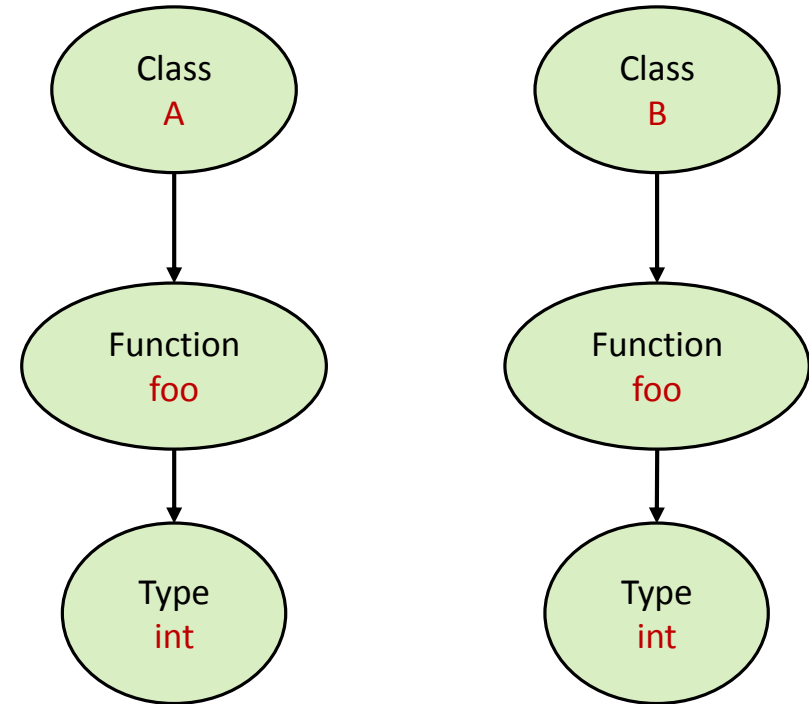
# Inheritance

```
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;
  }
}
```
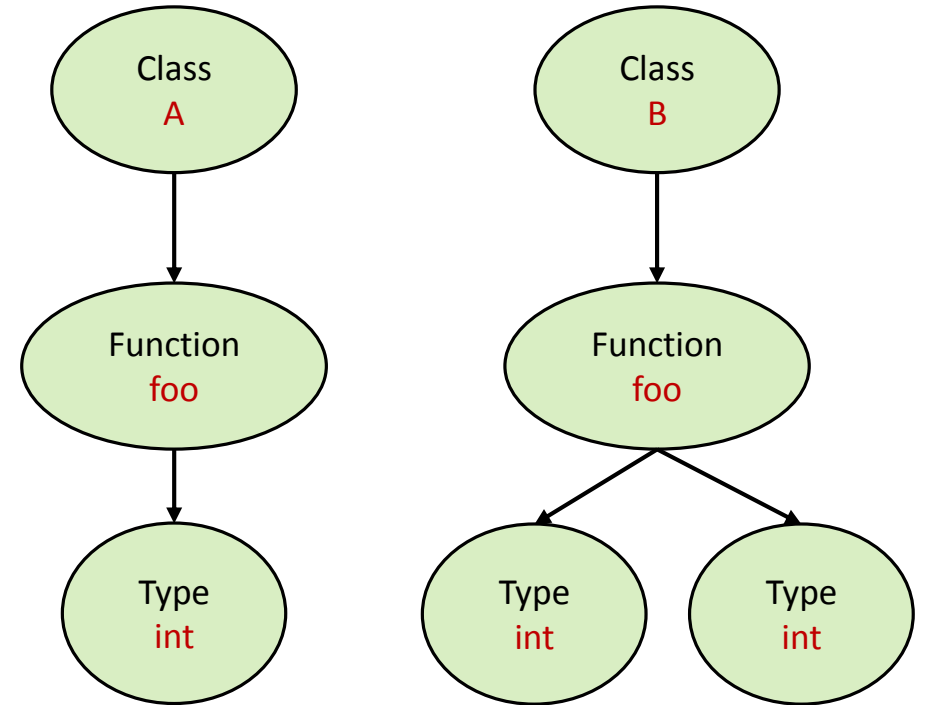
# Inheritance

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

Invalid
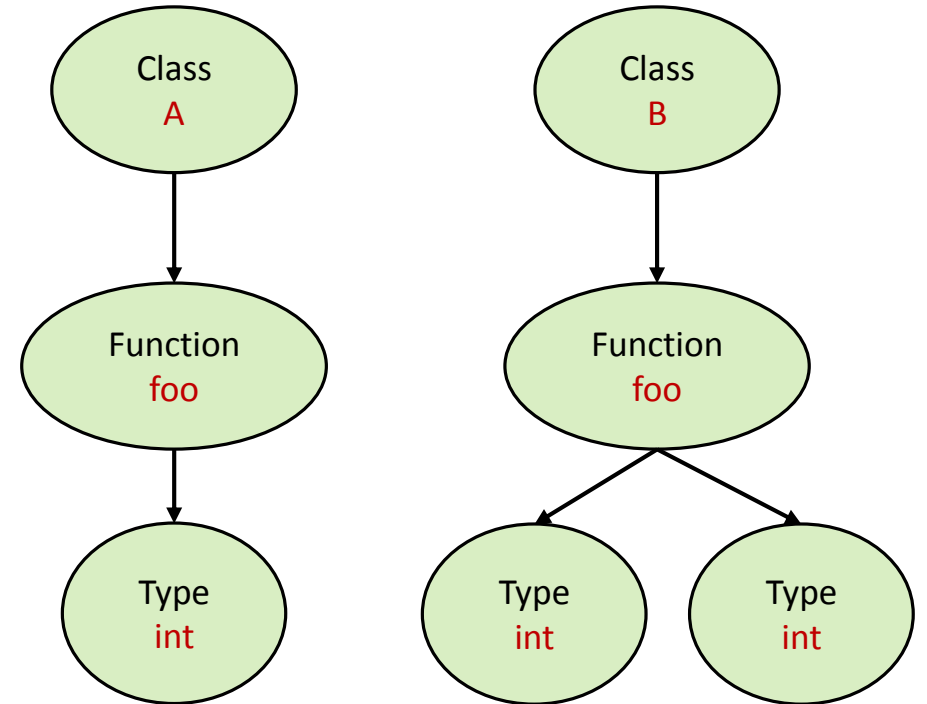
# Inheritance

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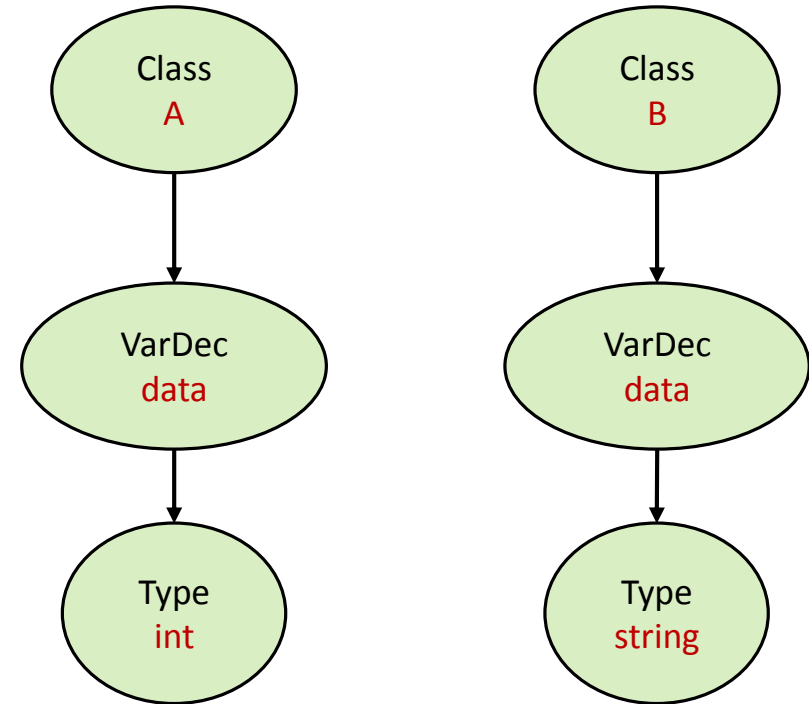
# Inheritance

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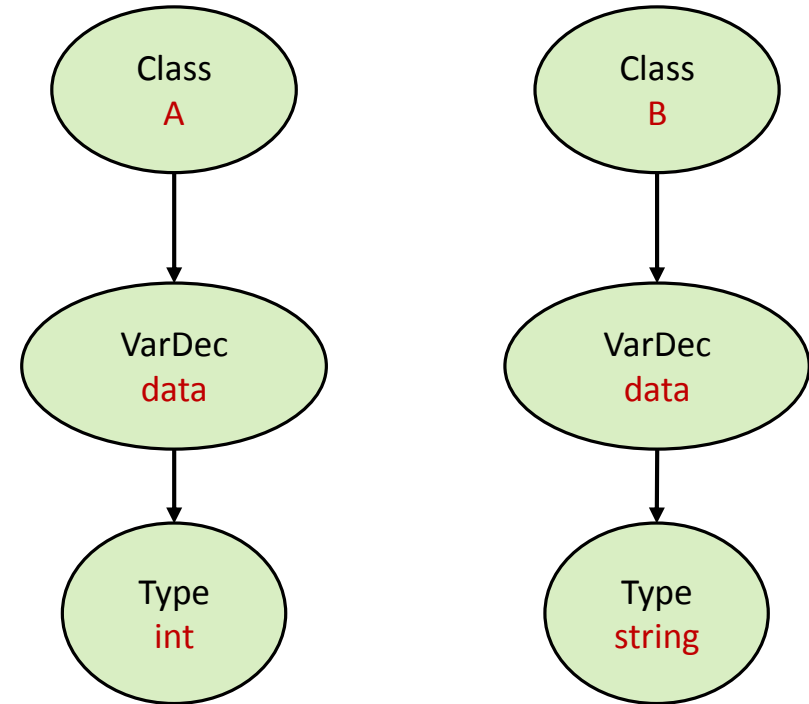


Invalid

# Inheritance

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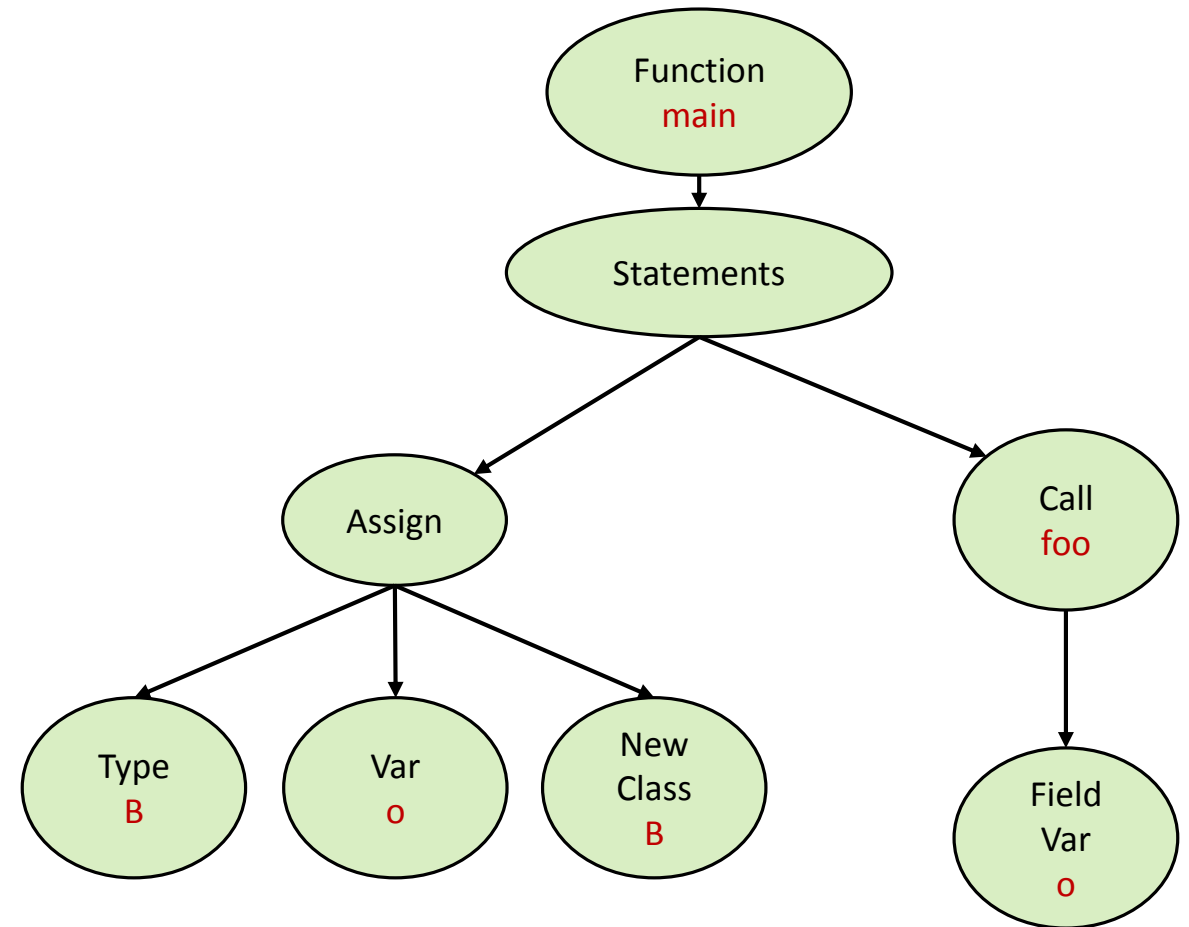
# Inheritance
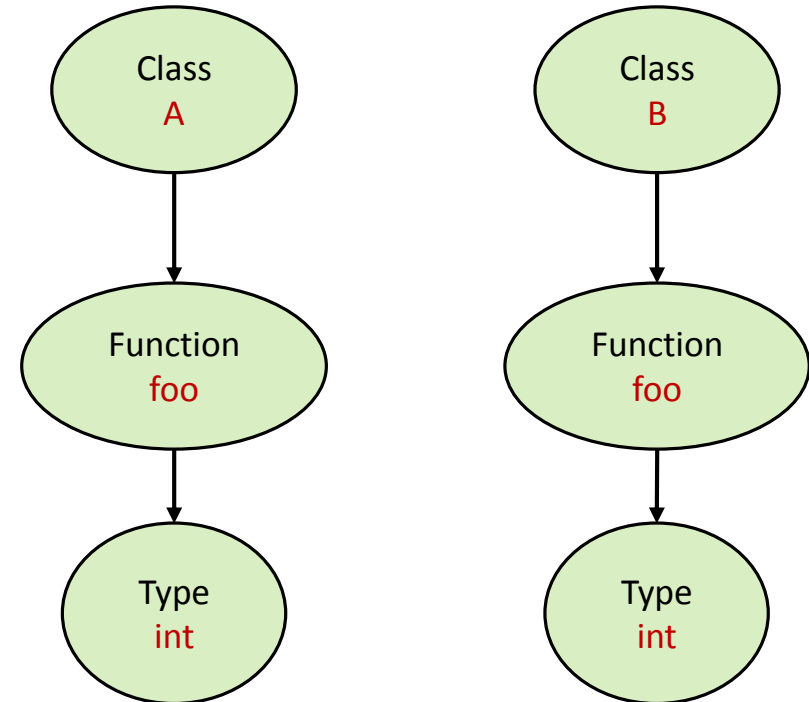
```
class A { }
class B extends A { }
void foo(A a) { }
void main() {
    B o = new B;
    foo(o);
}
```



**Valid**
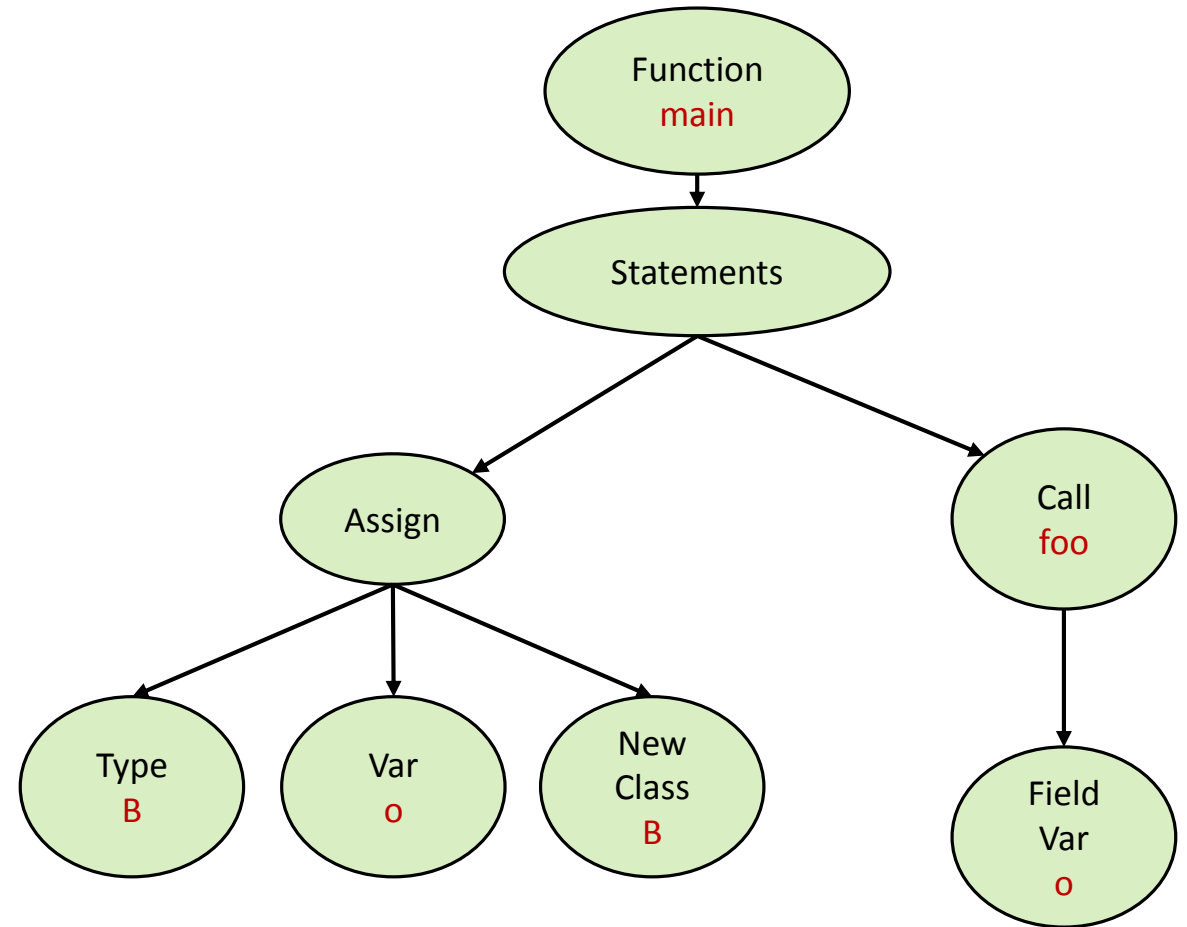
# Inheritance

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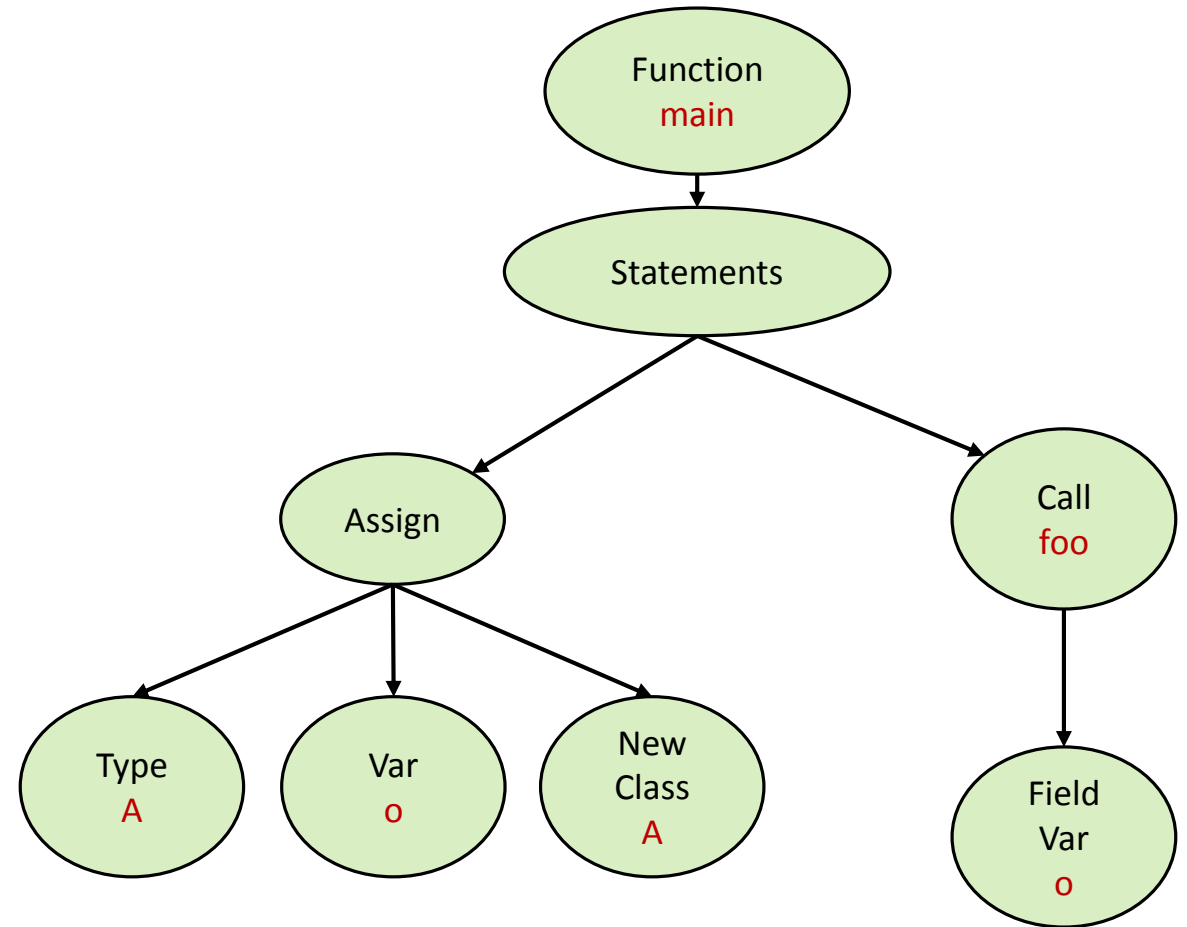
# Inheritance

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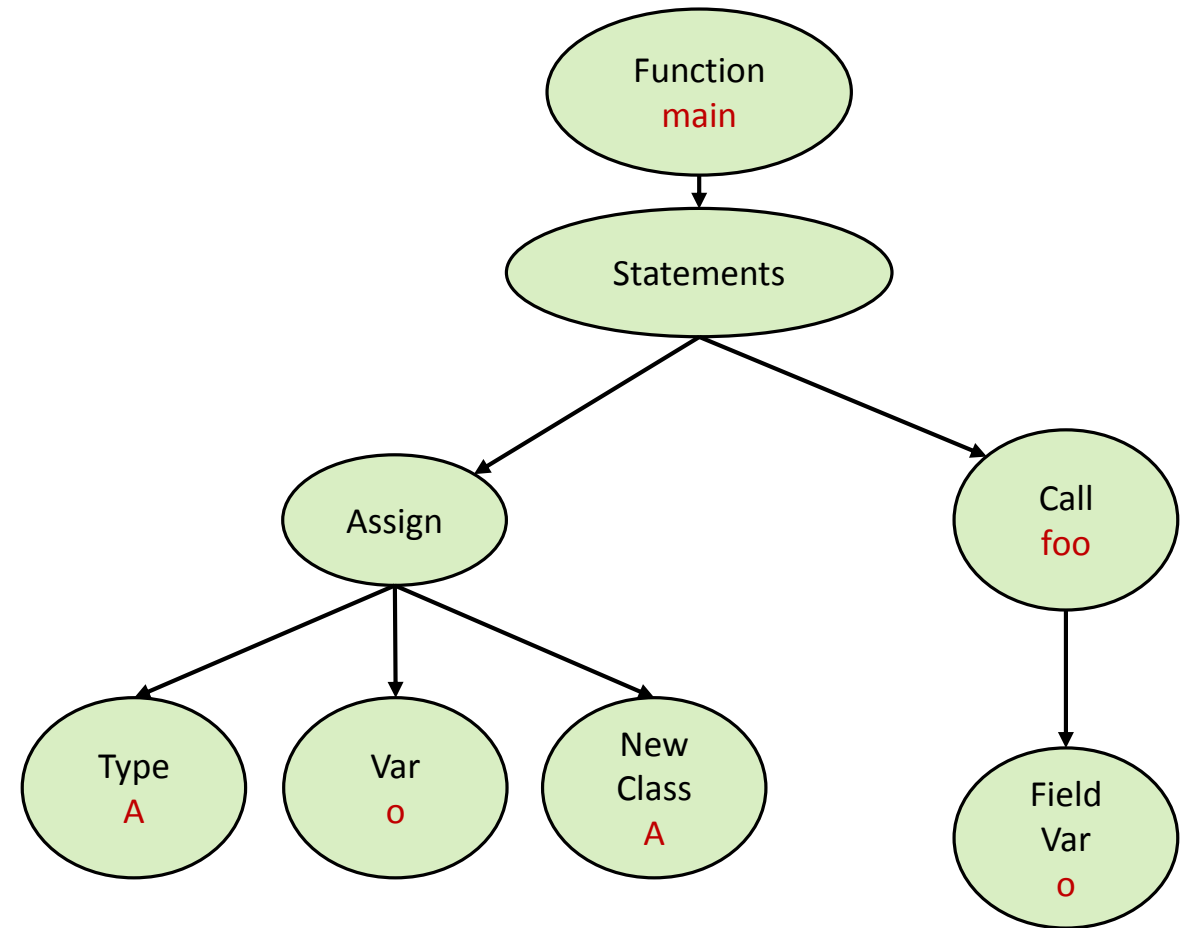
**Invalid**

# Null

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

Function
foo

Type
arr_t

Function
main

Statements

Call
foo

Null

# Null

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

Function
foo

Type
arr_t

Function
main

Statements
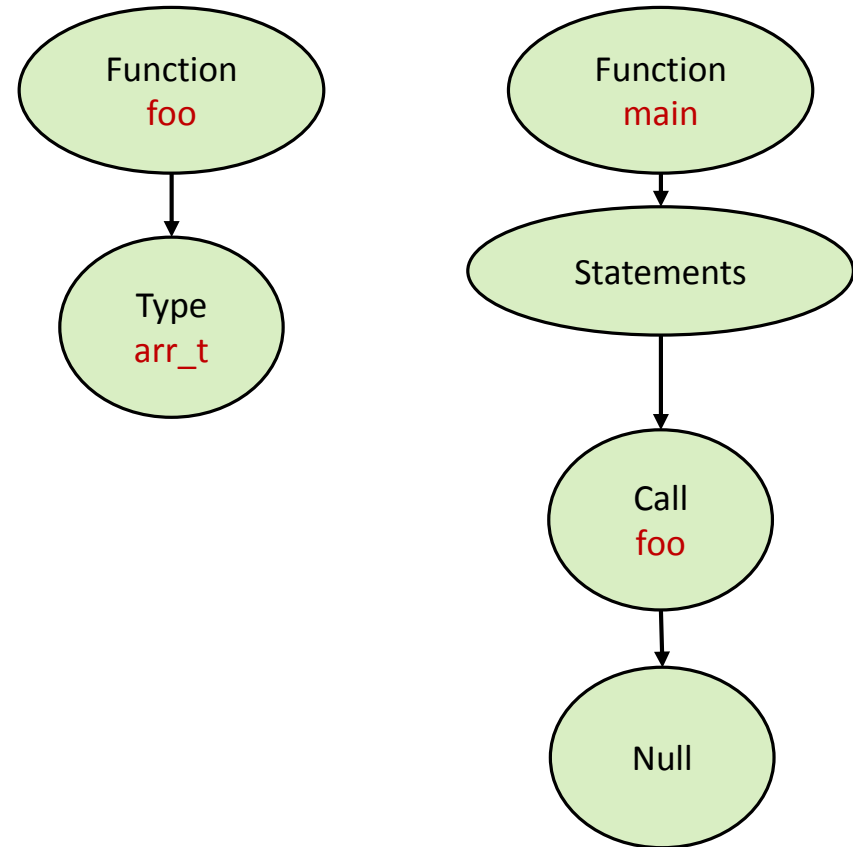
Call
foo

Null

**Valid**

# Null

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

Function
foo

Type
A

Function
main

Statements
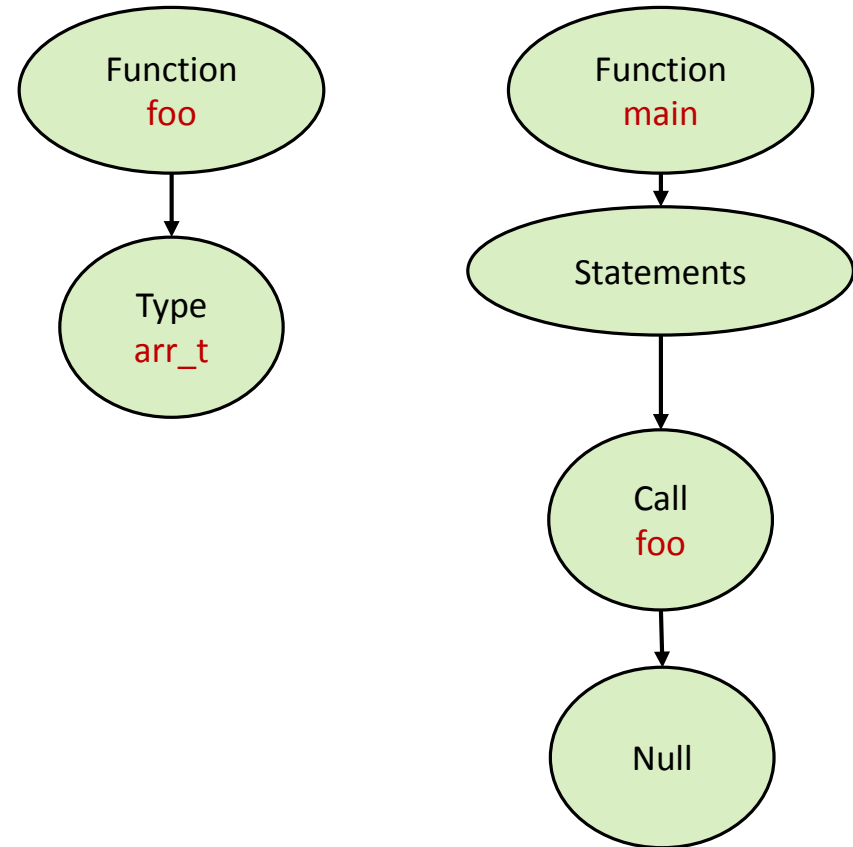
Call
foo

Null

# Null

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

Function
foo

Type
A

Function
main

Statements
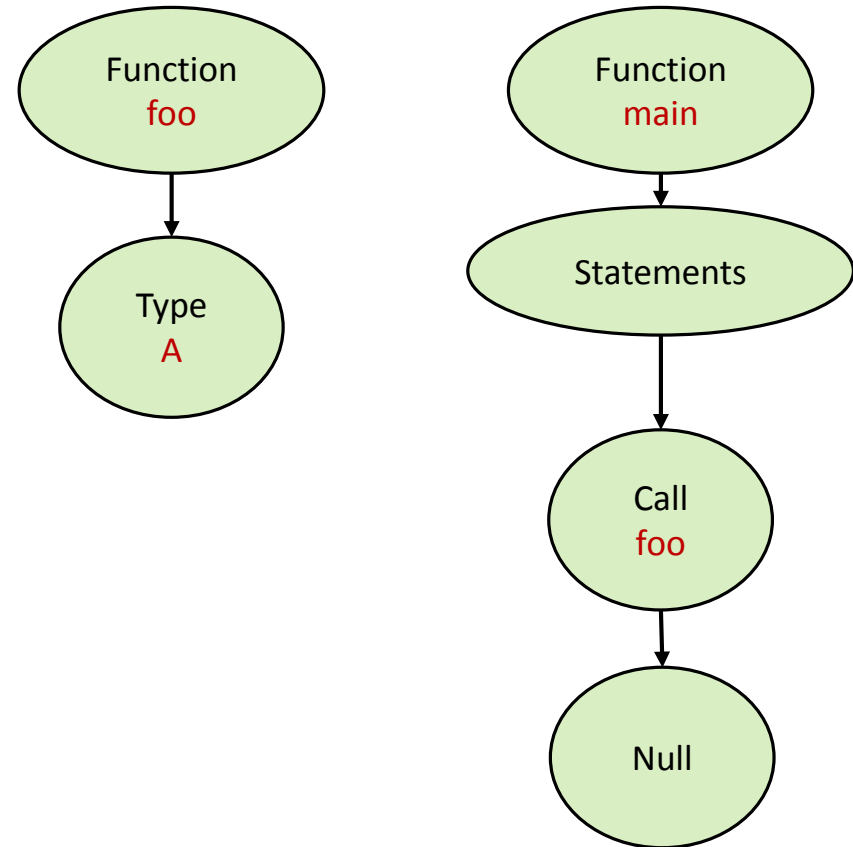
Call
foo

Null
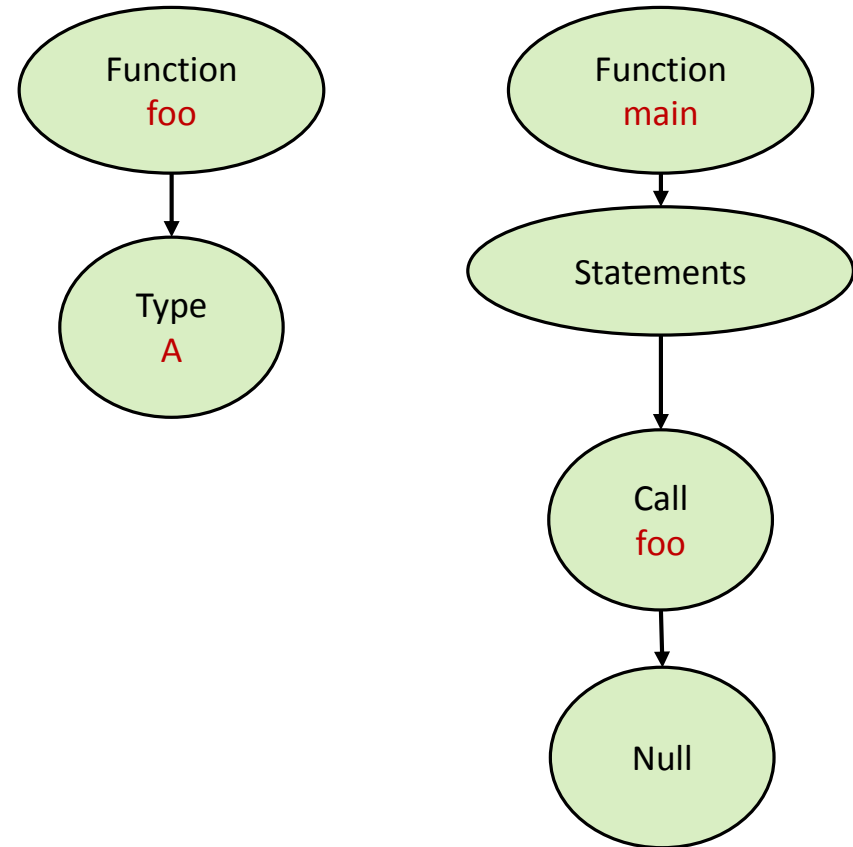
Valid

# Scopes

```
void foo(void) {
  string a = "A";
  if (7) {
    int a = 1;
    int b = a + 1;
  }
}
```
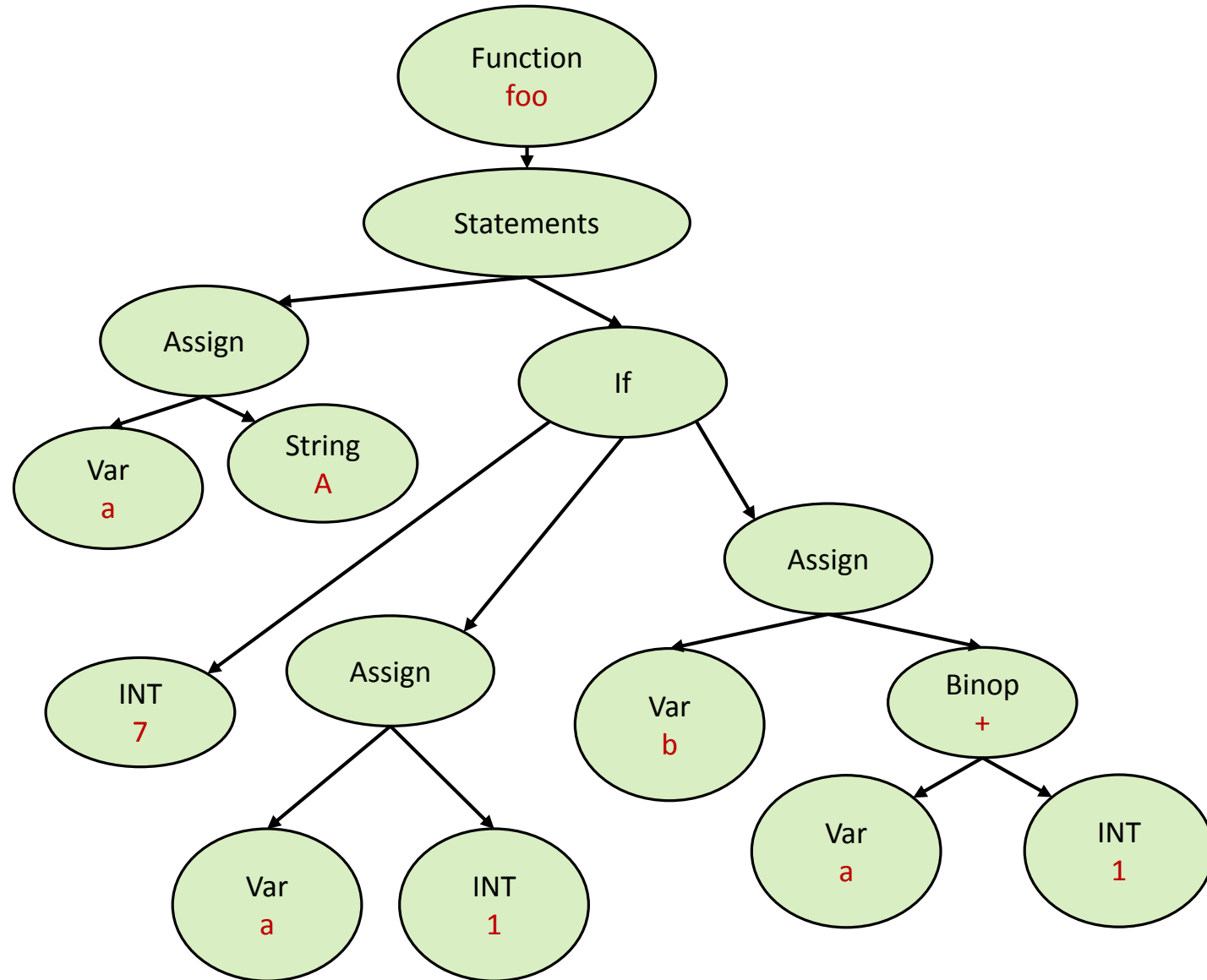
# Scopes

```
void foo(void) {
    string a = "A";
    if (7) {
        int a = 1;
        int b = a + 1;
    }
}
```

Valid
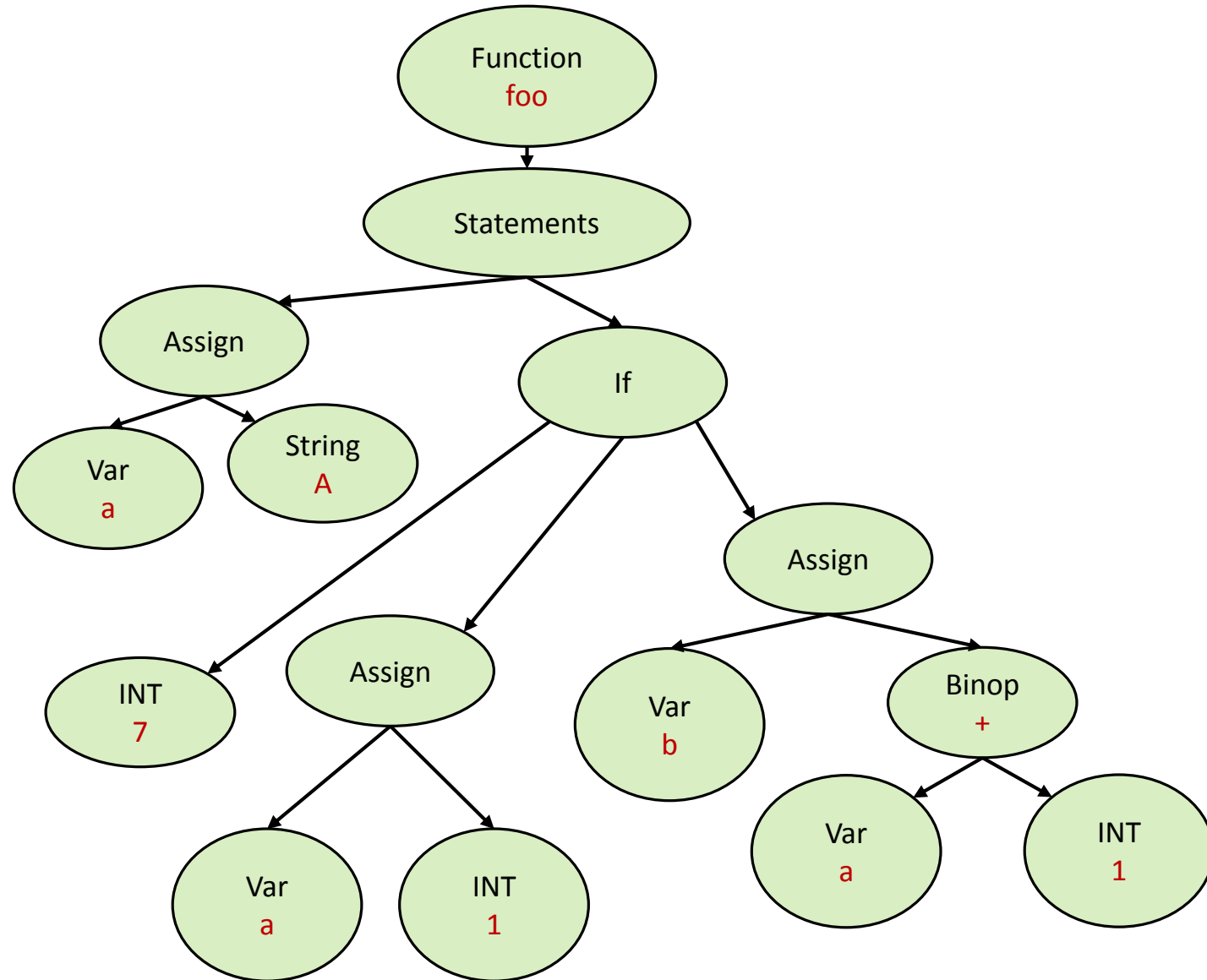
# Scopes

```
void foo(void) {
    int a = 1;
    if (7) {
        string a = "A";
        int b = a + 1;
    }
}
```
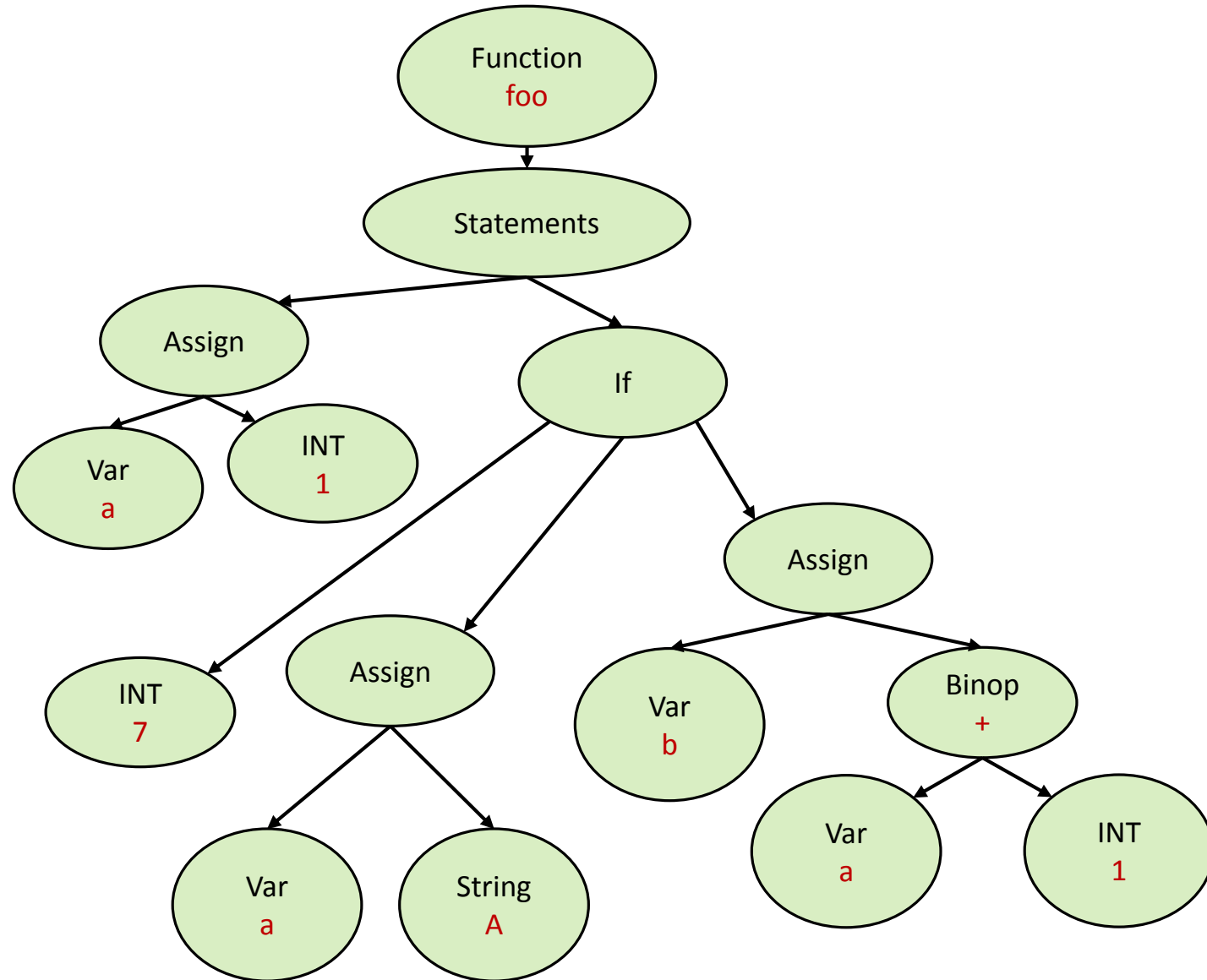
# Scopes

```
void foo(void) {
    int a = 1;
    if (7) {
        string a = "A";
        int b = a + 1;
    }
}
```

Invalid
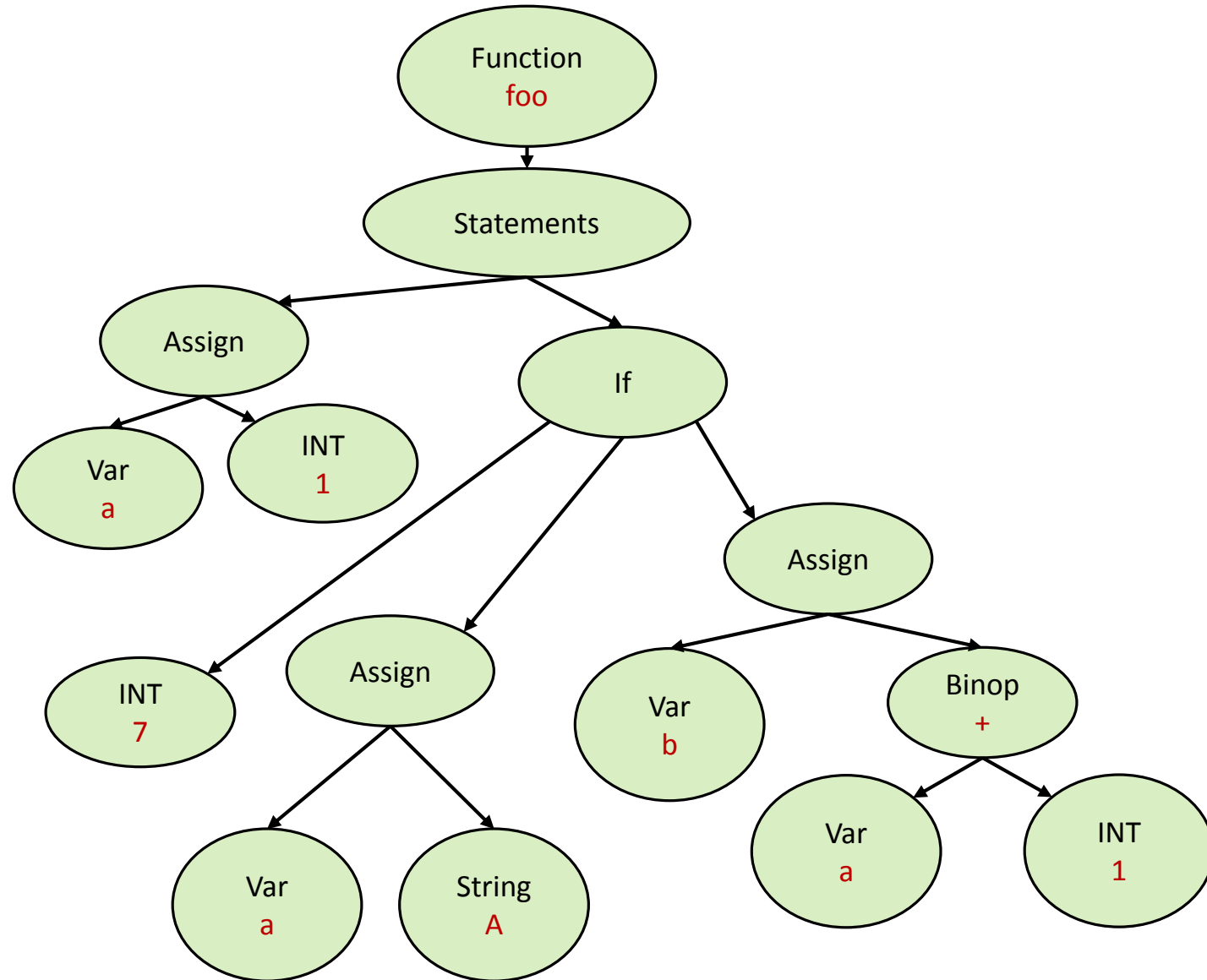
# Scopes

```
int z = 1000;
void foo(int z) {
  if (7) {
    int b = z + 1;
  }
}
```
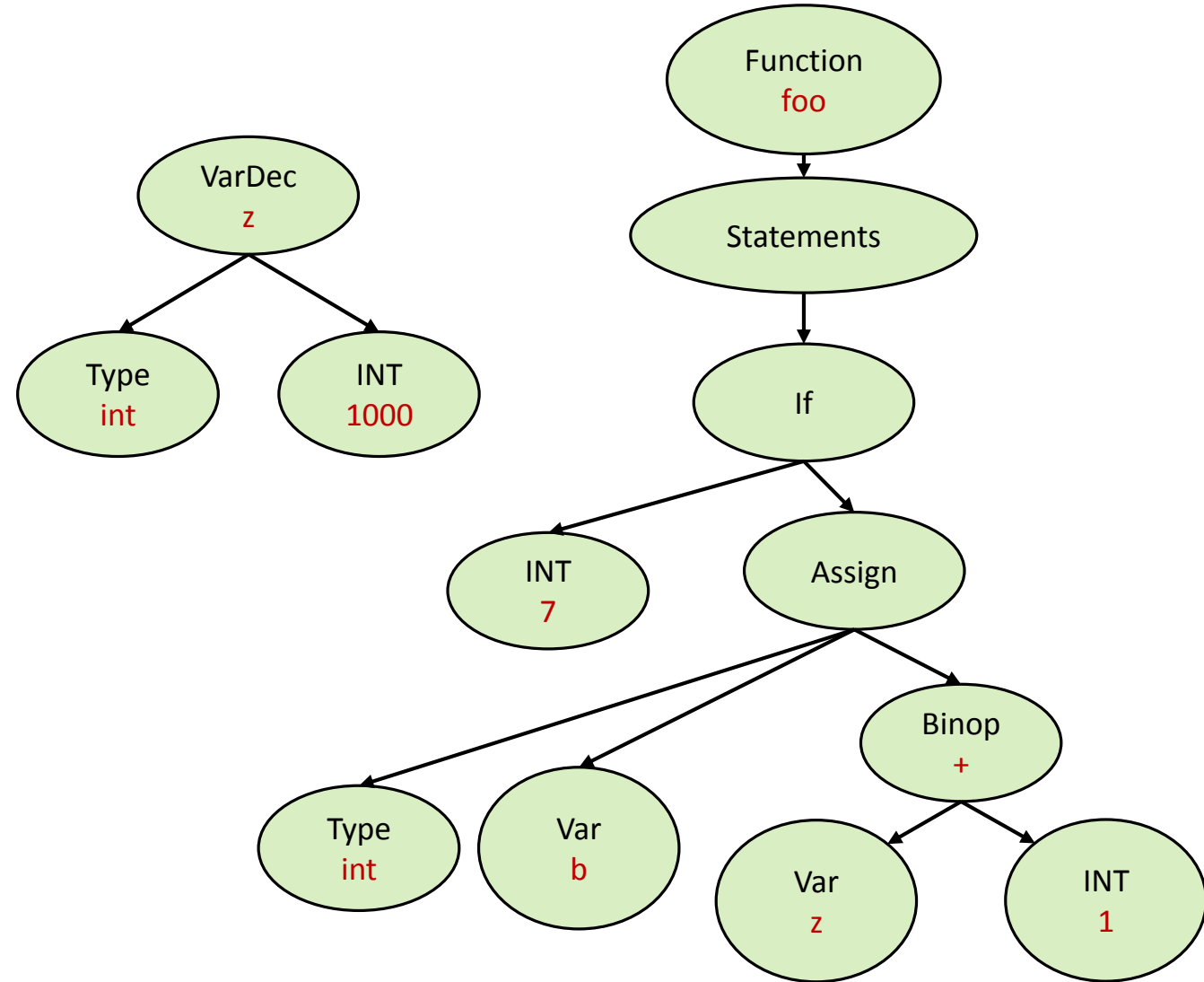
# Scopes

```
int z = 1000;
void foo(int z) {
  if (7) {
    int b = z + 1;
  }
}
```

Valid
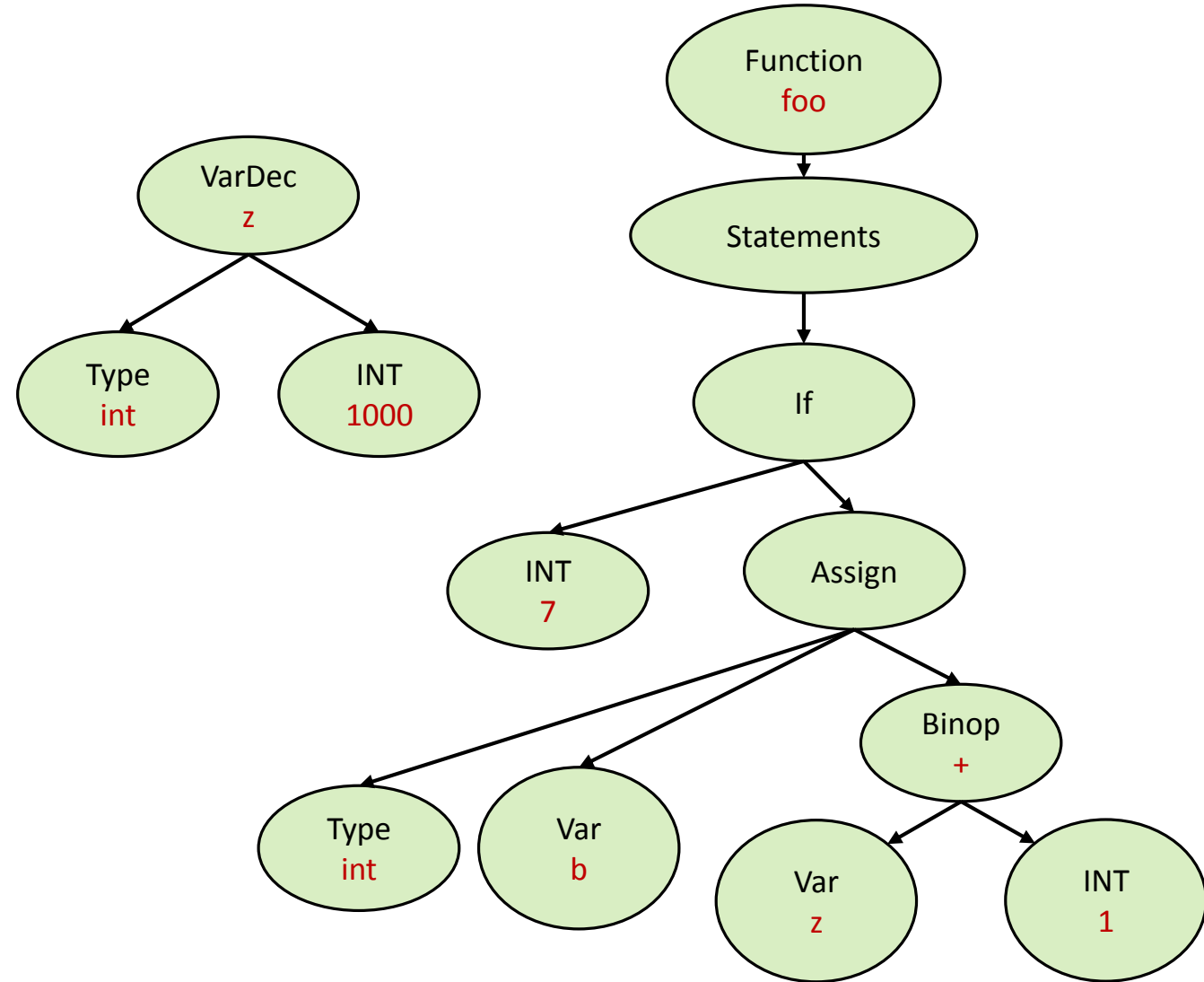
# Scopes

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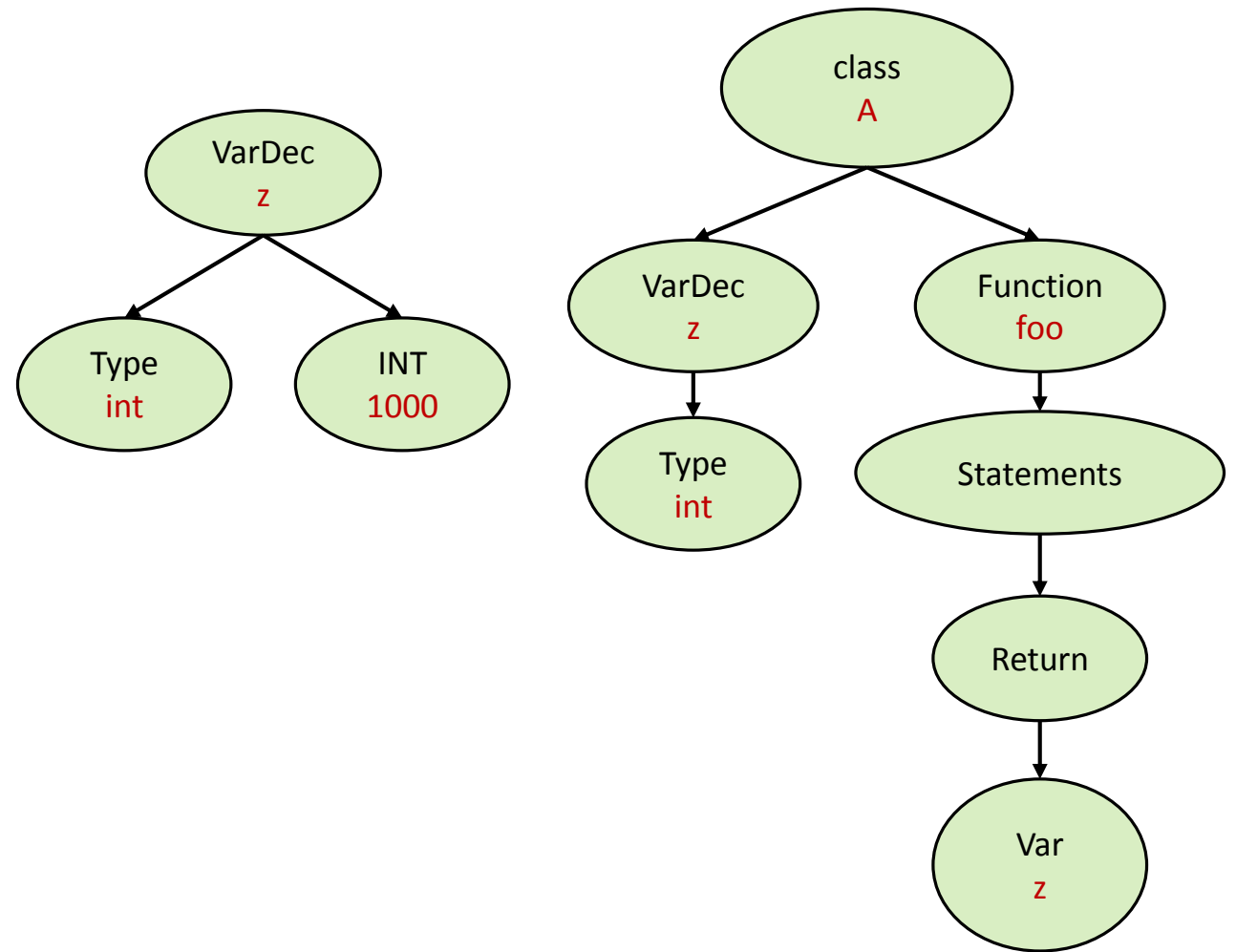
# Scopes

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



Valid
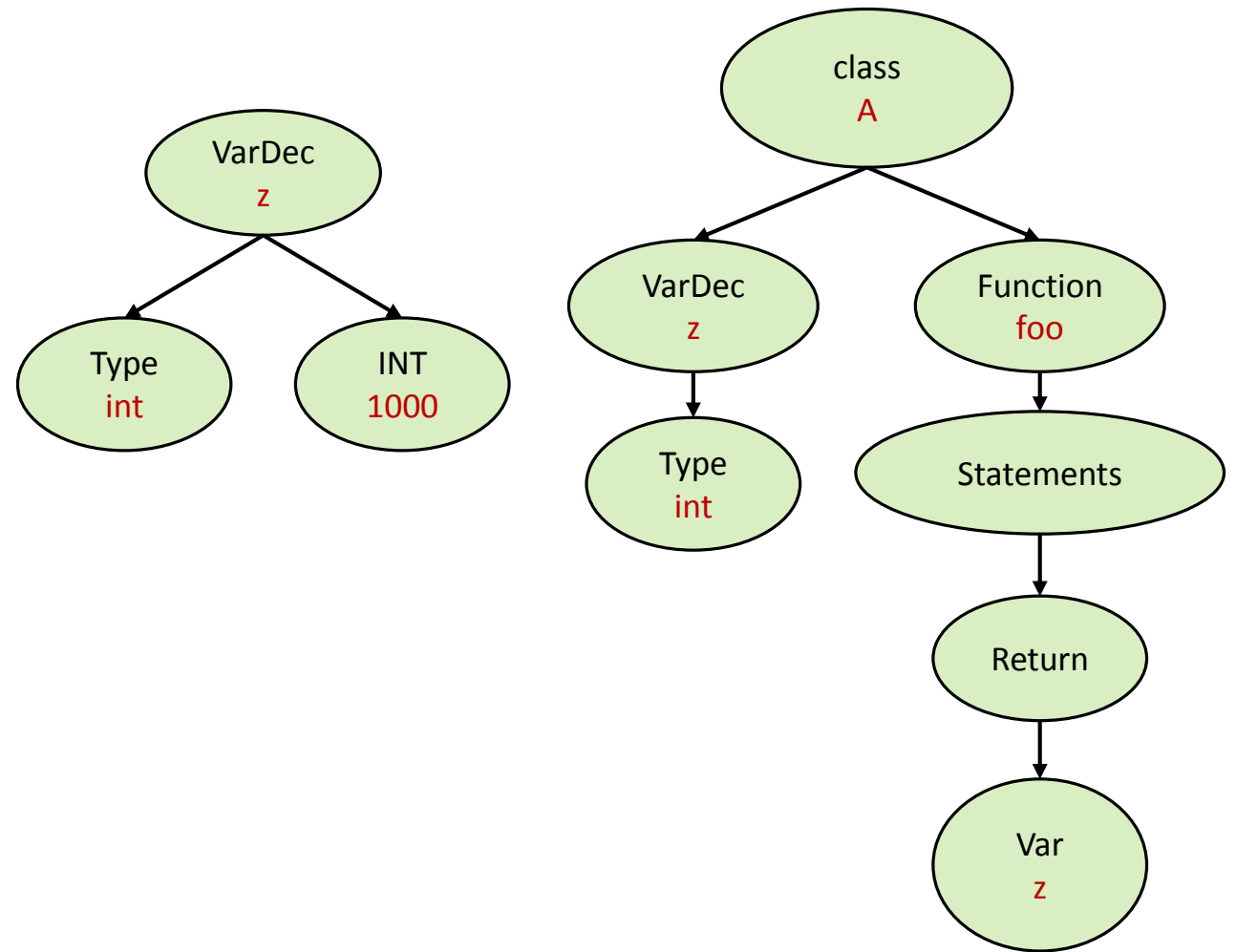
# Scopes

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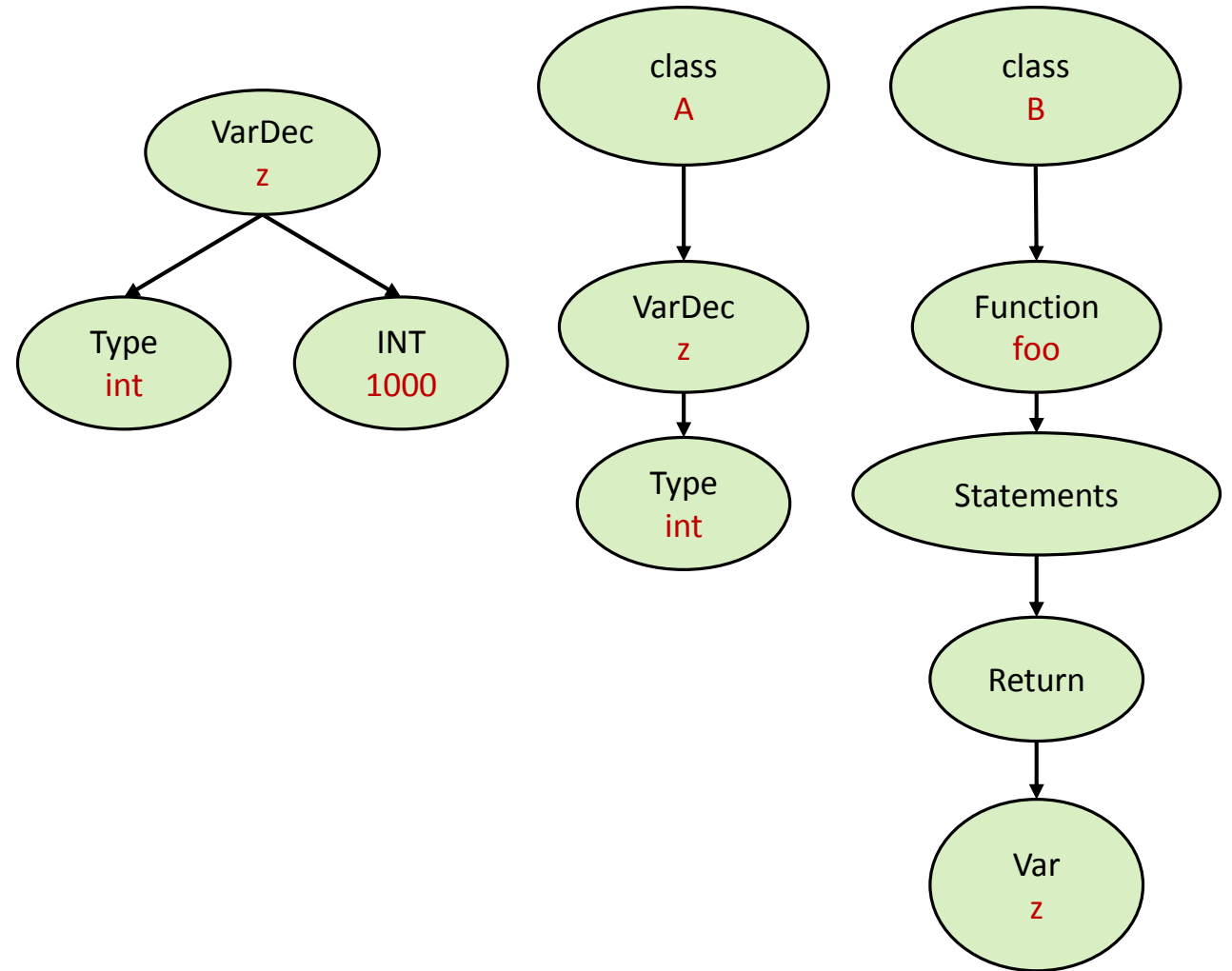
# Scopes

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

Valid
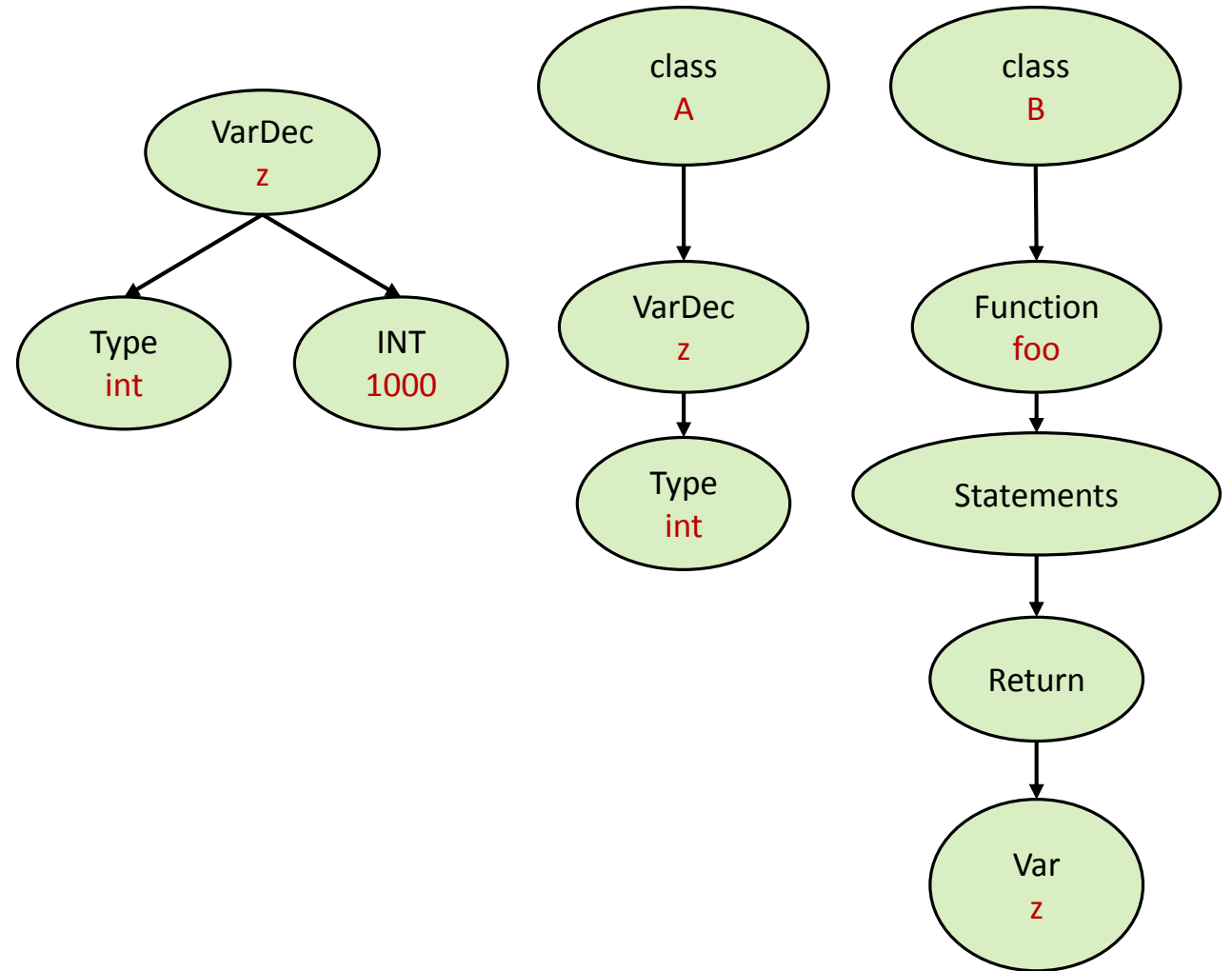
# Library Functions

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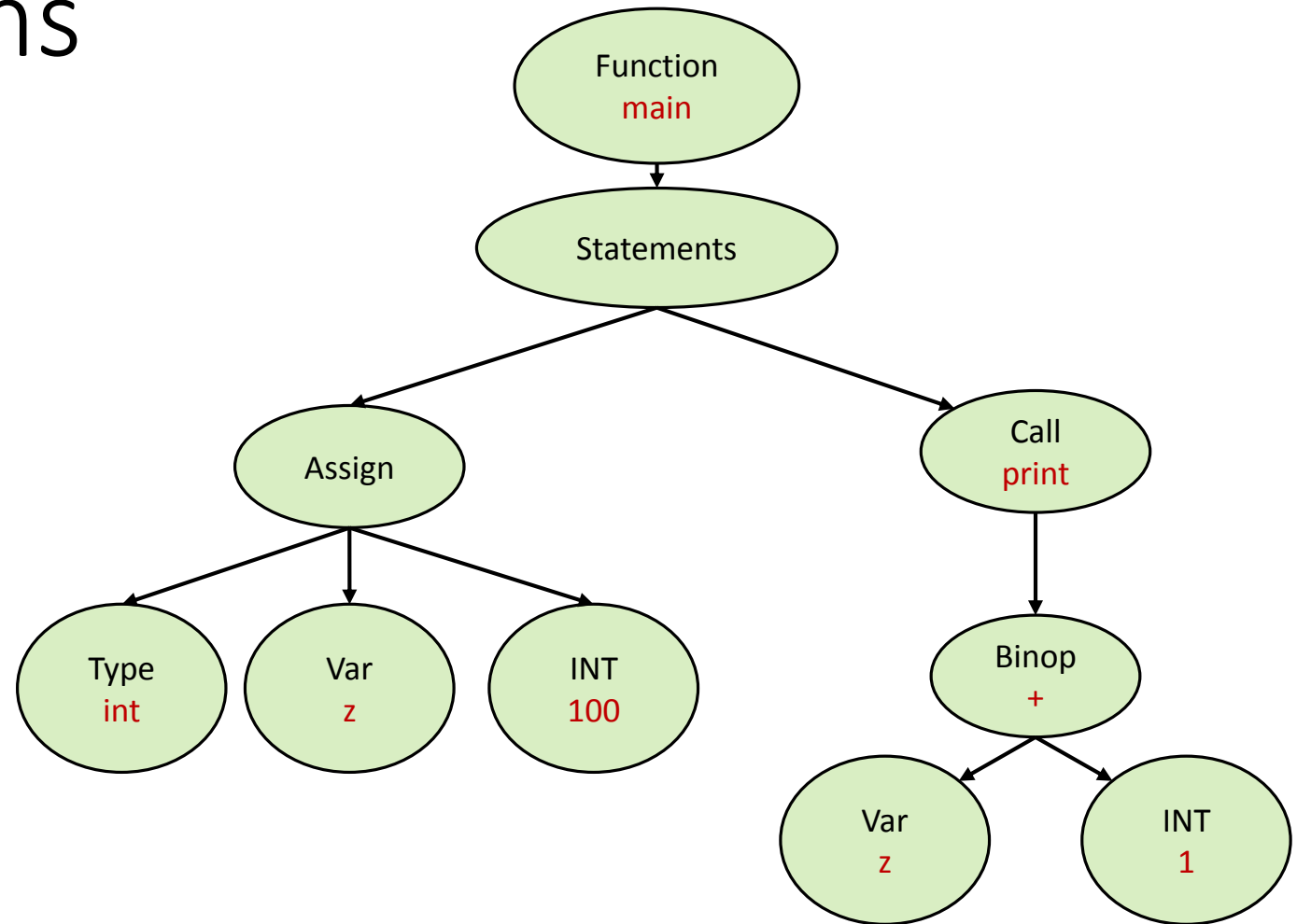
# Library Functions

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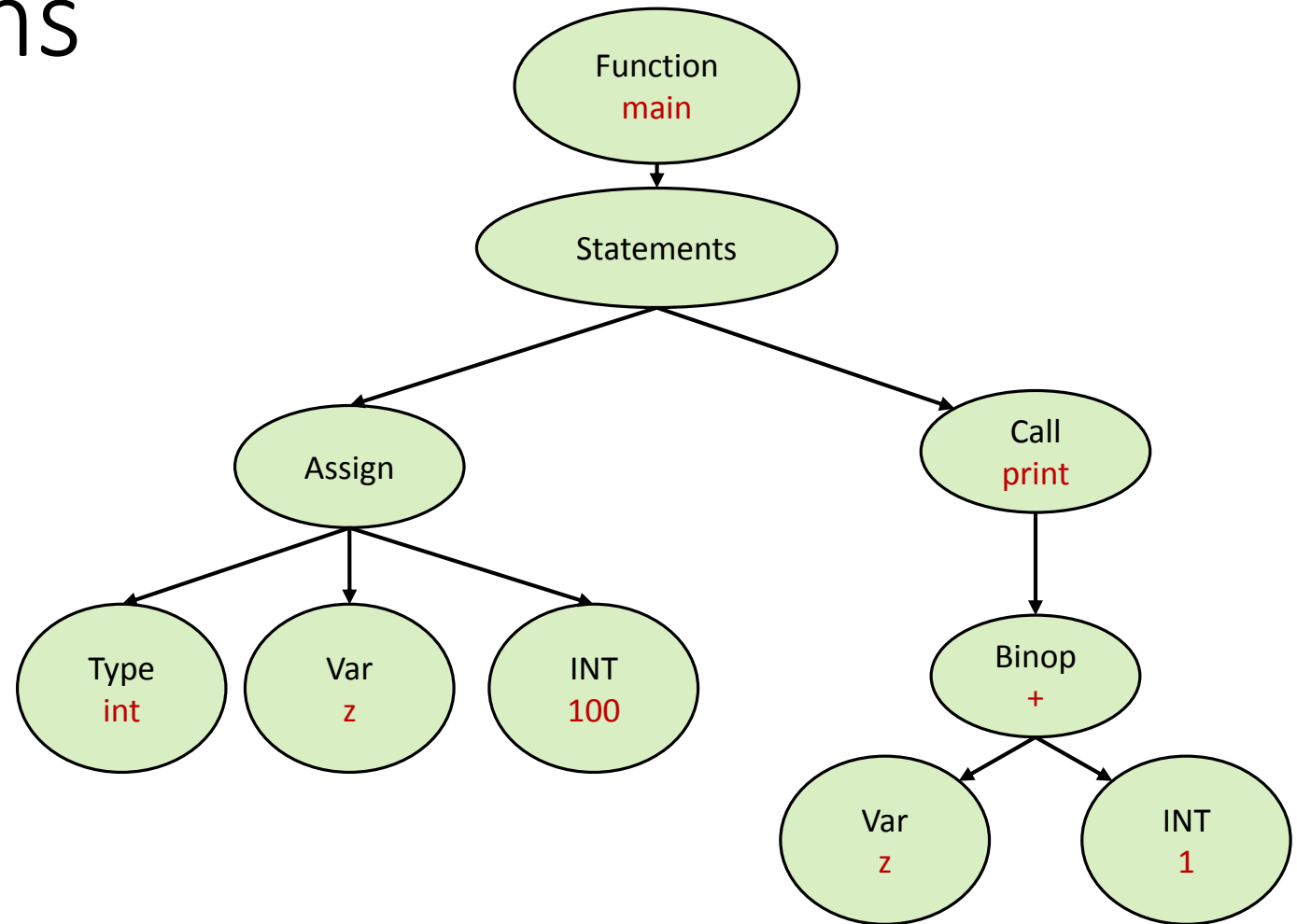


**Valid**

# 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
  - In the skeleton it's called *semantMe*
- 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
    }
    …
  }
}
```

# 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

While analyzing the AST, we can extend it with useful information:
- Variable offsets
- Parameter offsets
- Class layout
- Type size

# Stack
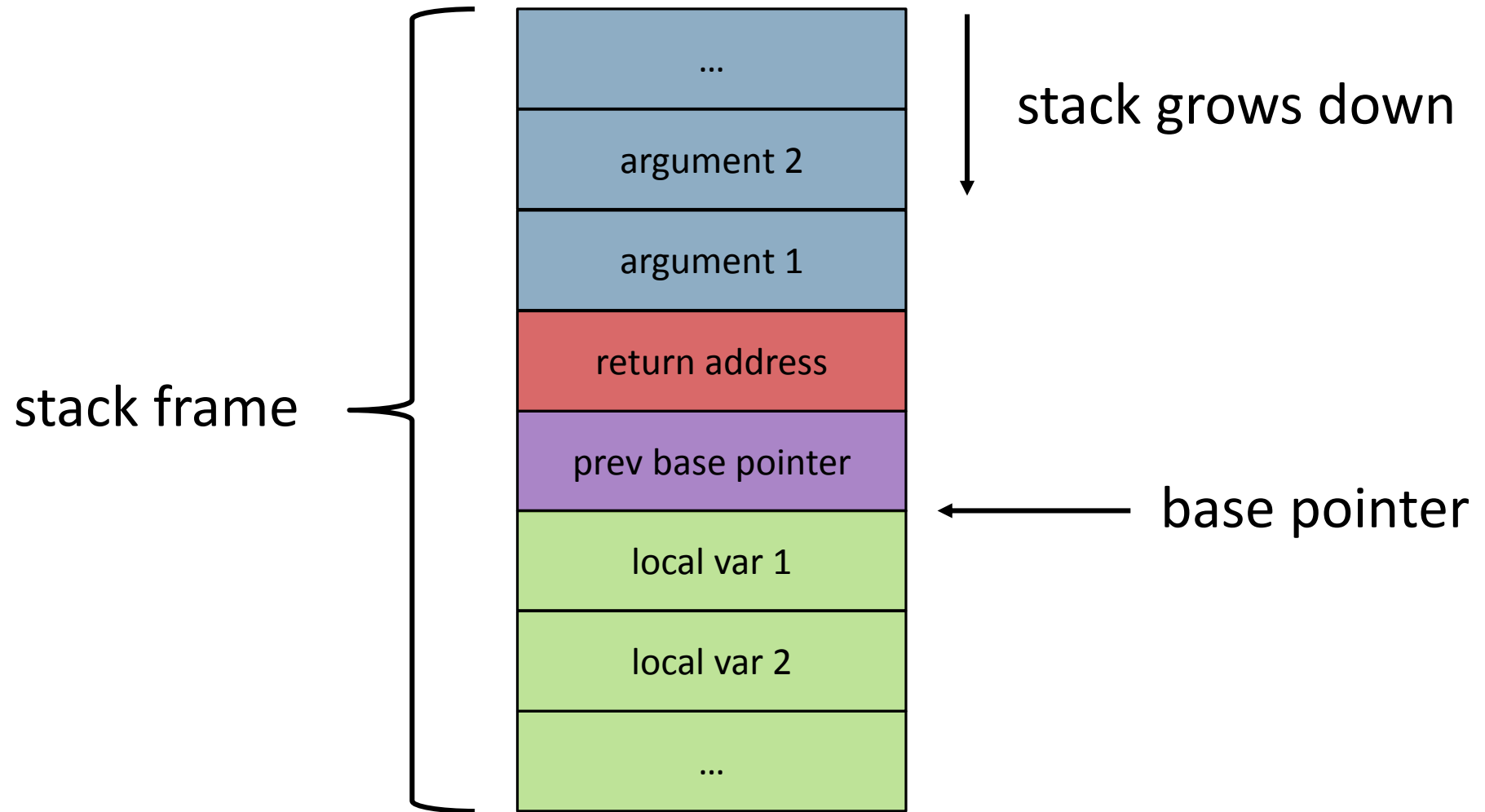
# Stack

```
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

argument 2

```
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

| |
|---|
| argument 2 |
| argument 1 |

```
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

| |
|---|
| argument 2 |
| argument 1 |

```
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
...
```
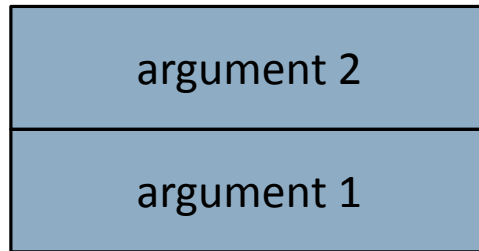
# Stack



```
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
...
```
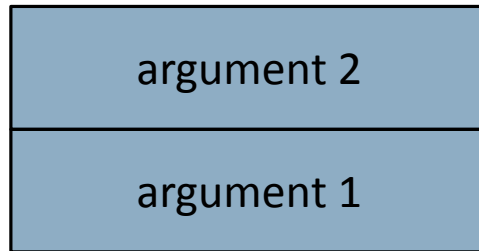
# Stack



```
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
...
```
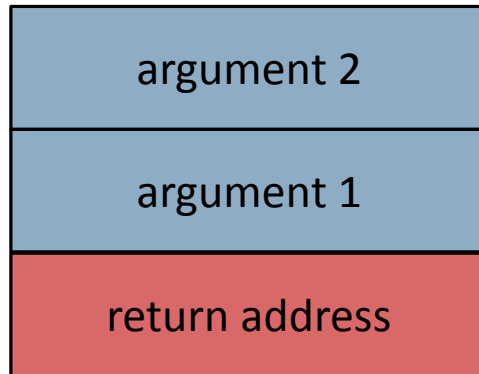
# Stack



```
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
...
```
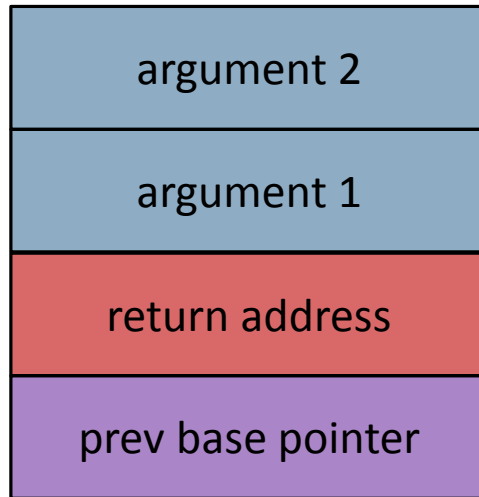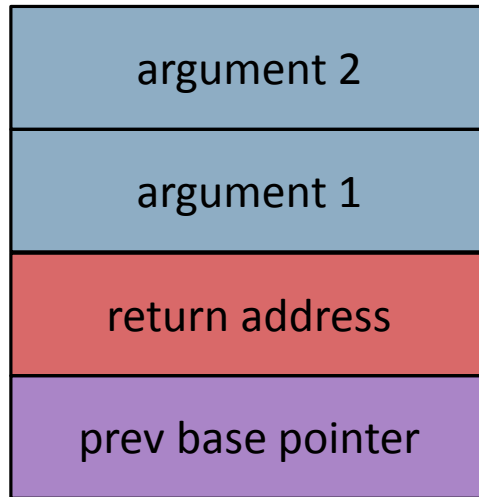
# Stack



```
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
...
```
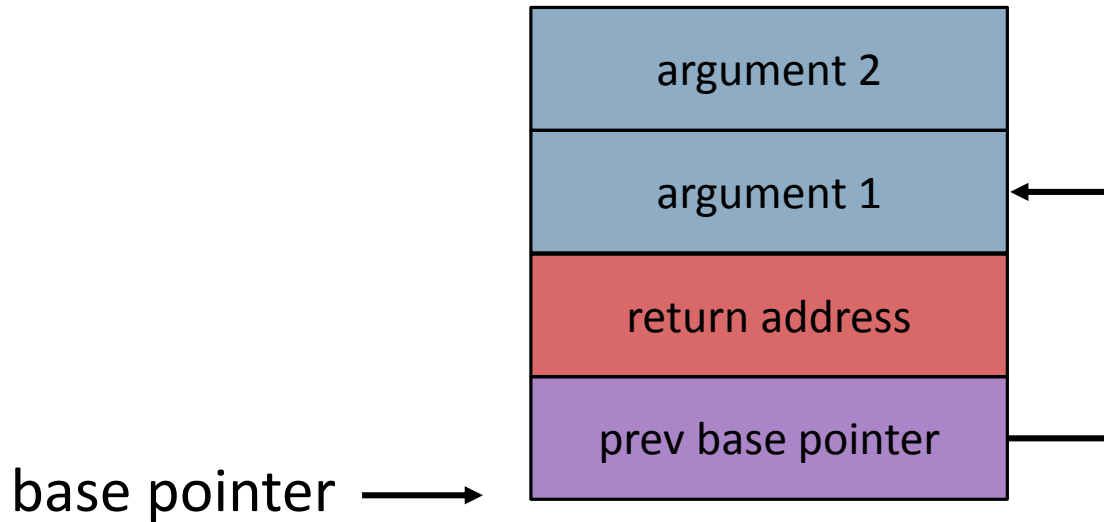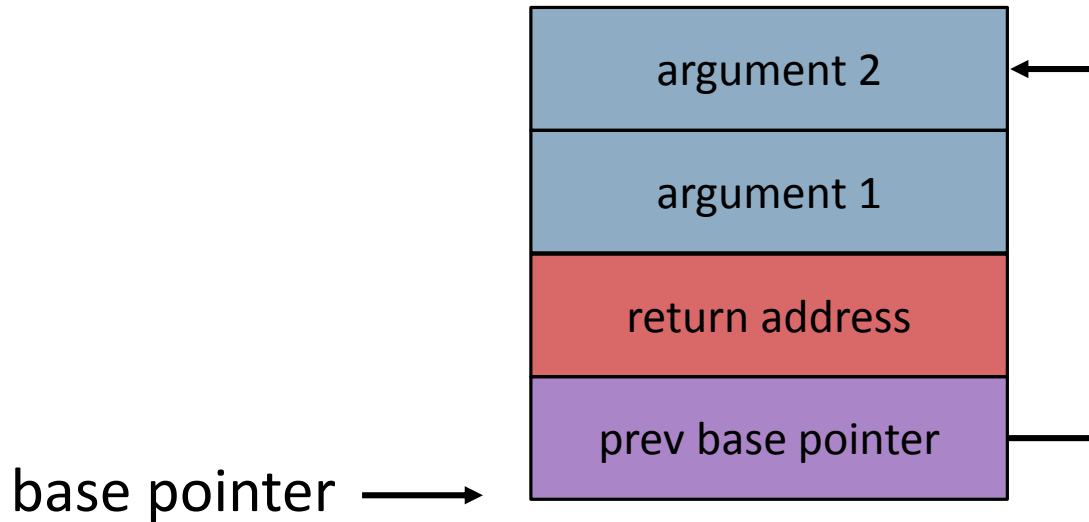
# Stack



```
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
...
```
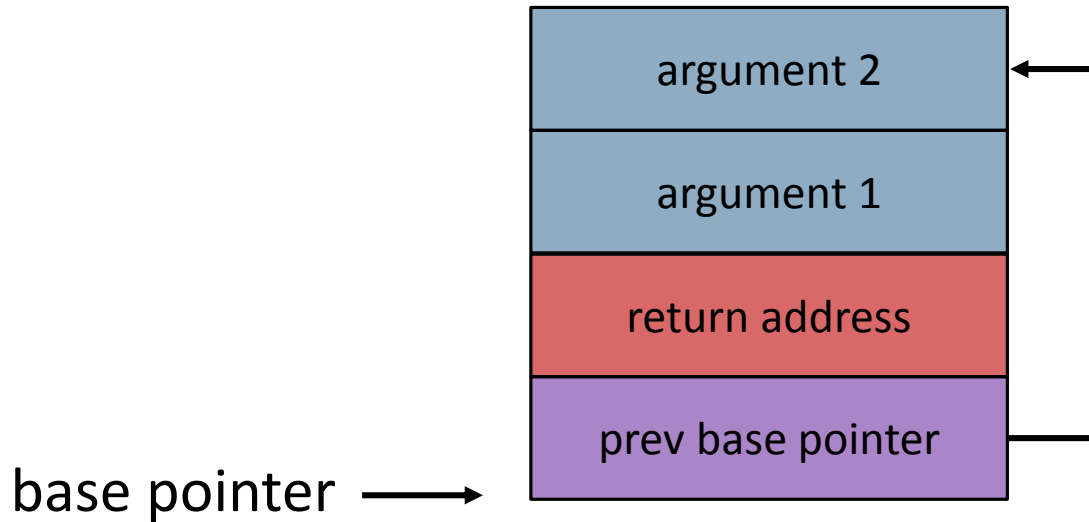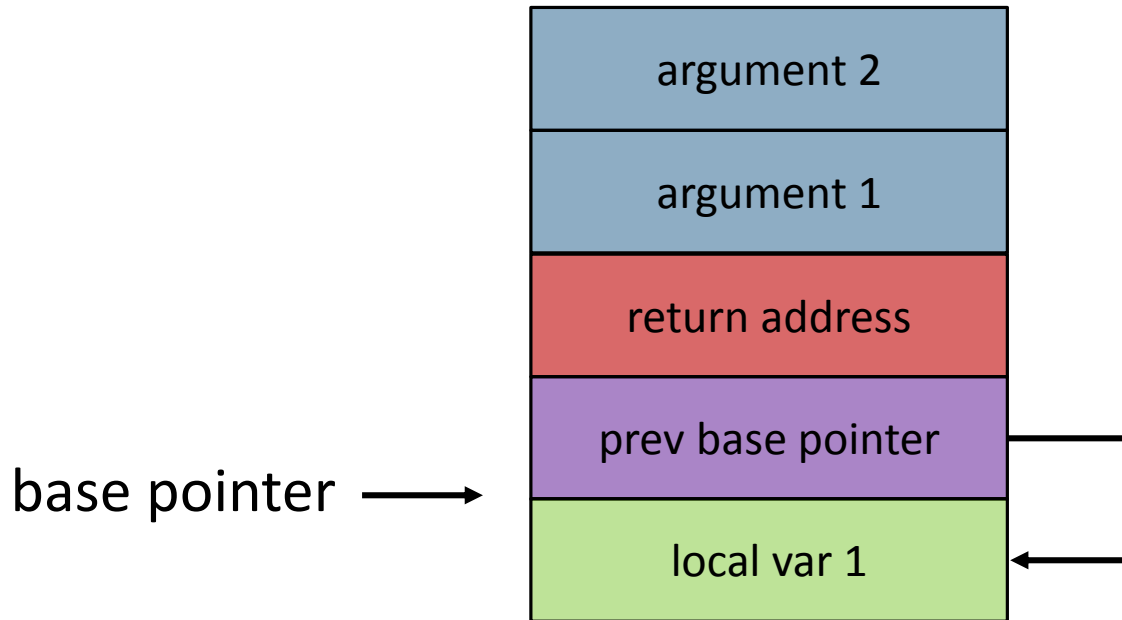
# Stack



```
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** from the **stack base pointer**

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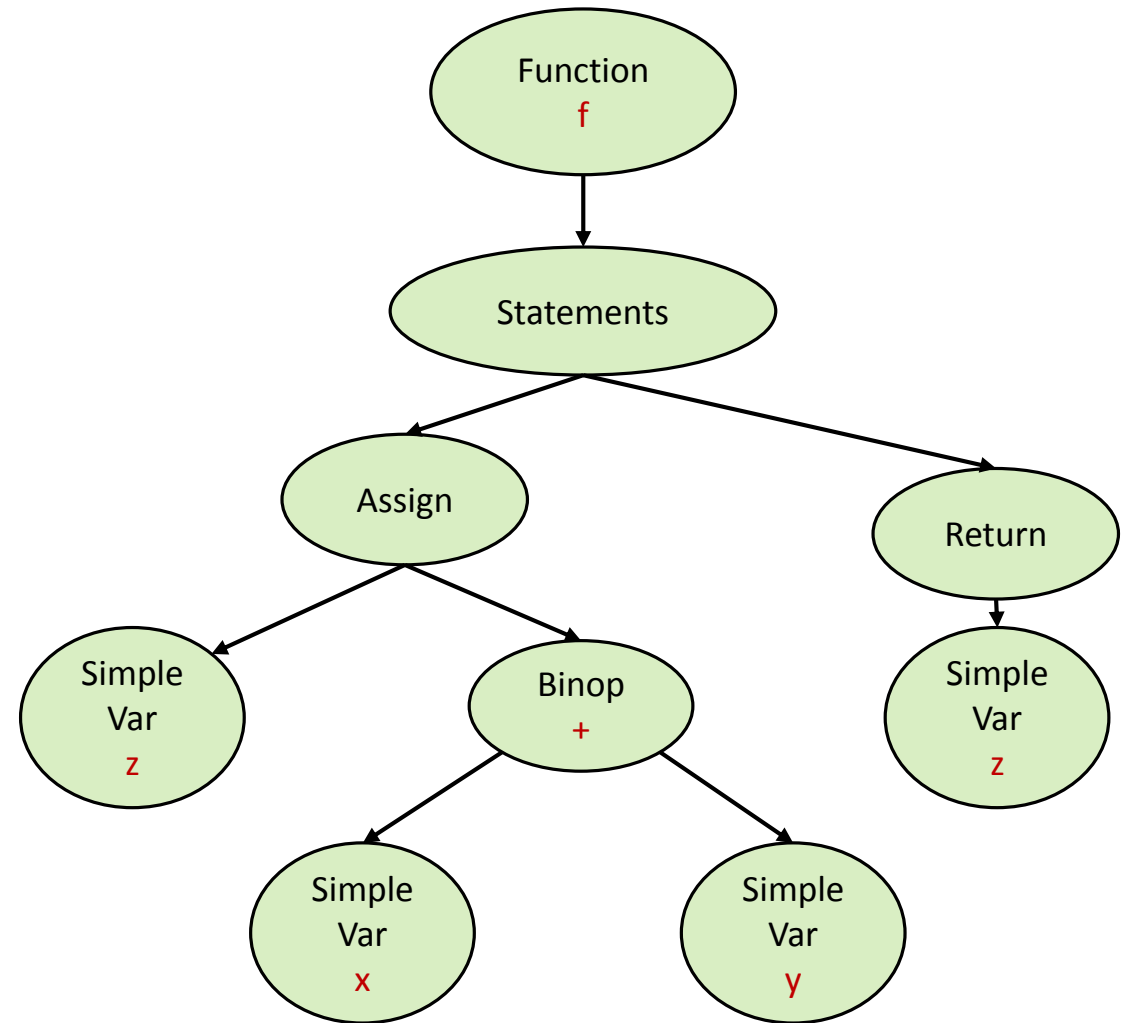
```
int f(int x, int y){
    int z = x + y;
    return z;
}
```

# Variable Offsets

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

# Variable Offsets

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

| x | ⟹ | param 1 | ⟹ | 0x08 |
| y | ⟹ | param 2 | ⟹ | 0x0c |
| z | ⟹ | local 1 | ⟹ | - 0x04 |

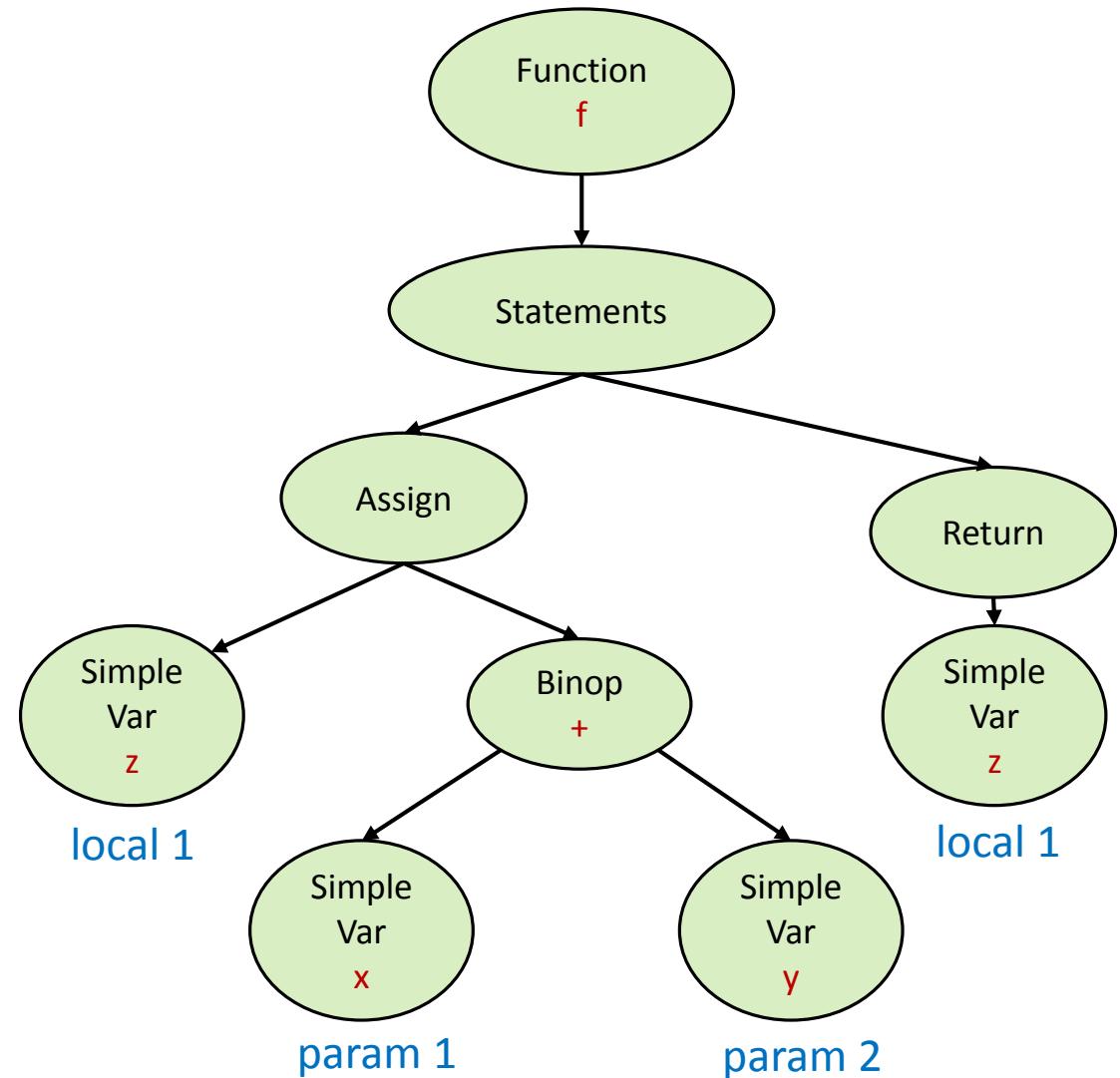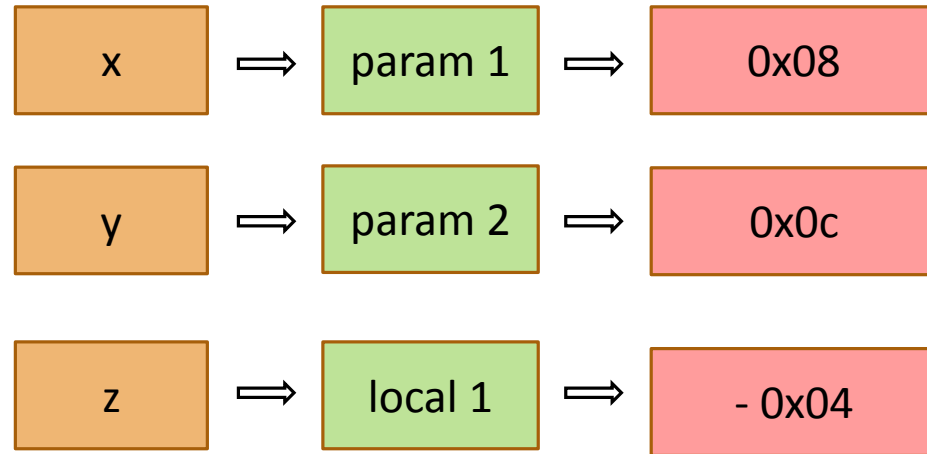# Variable Offsets

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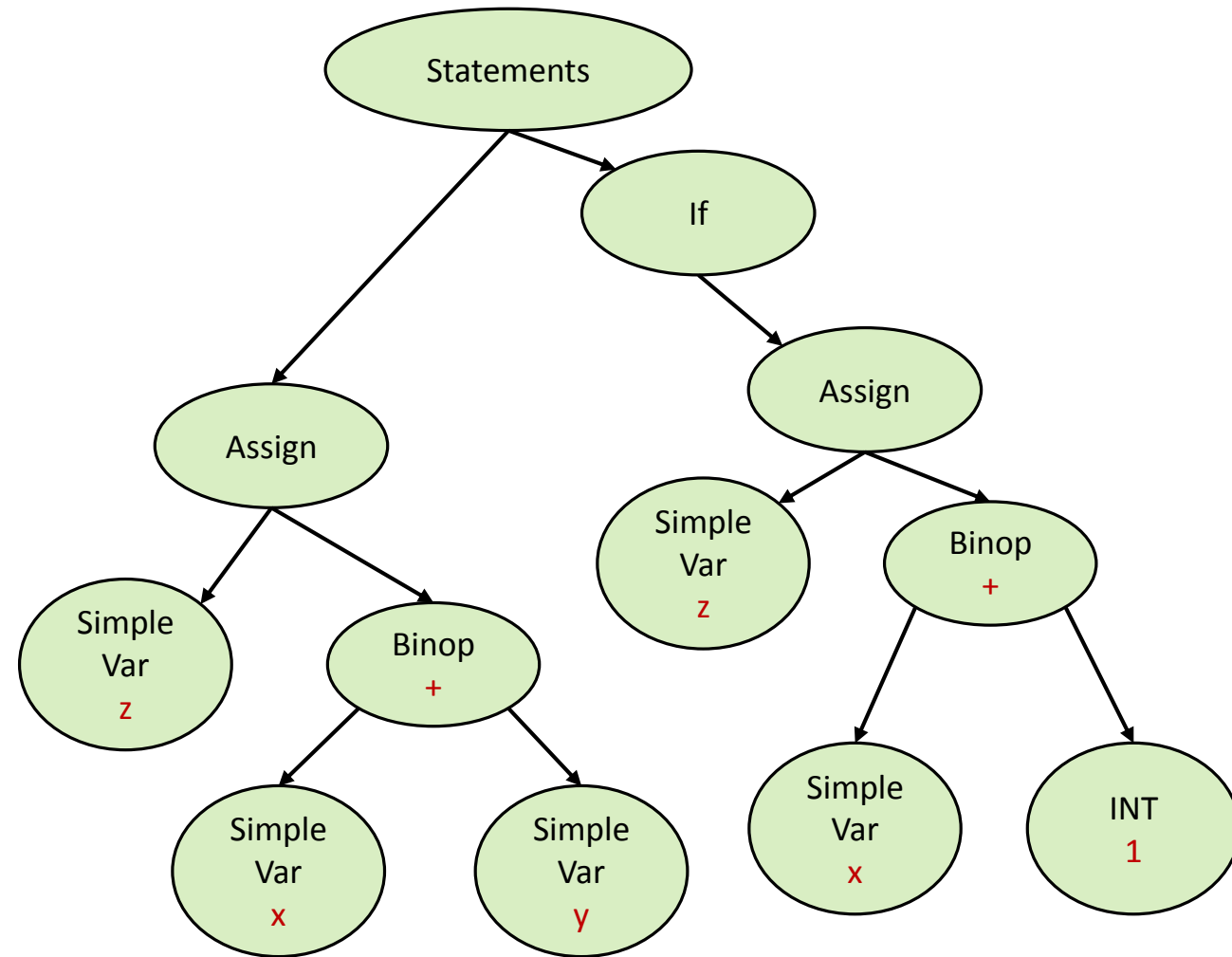
# Variable Offsets

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

# 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;
}
```

4 bytes

0x1000:

| ... | 0xaa | 0xbb | ... |

p: | 0x1004 |

# Field Offsets

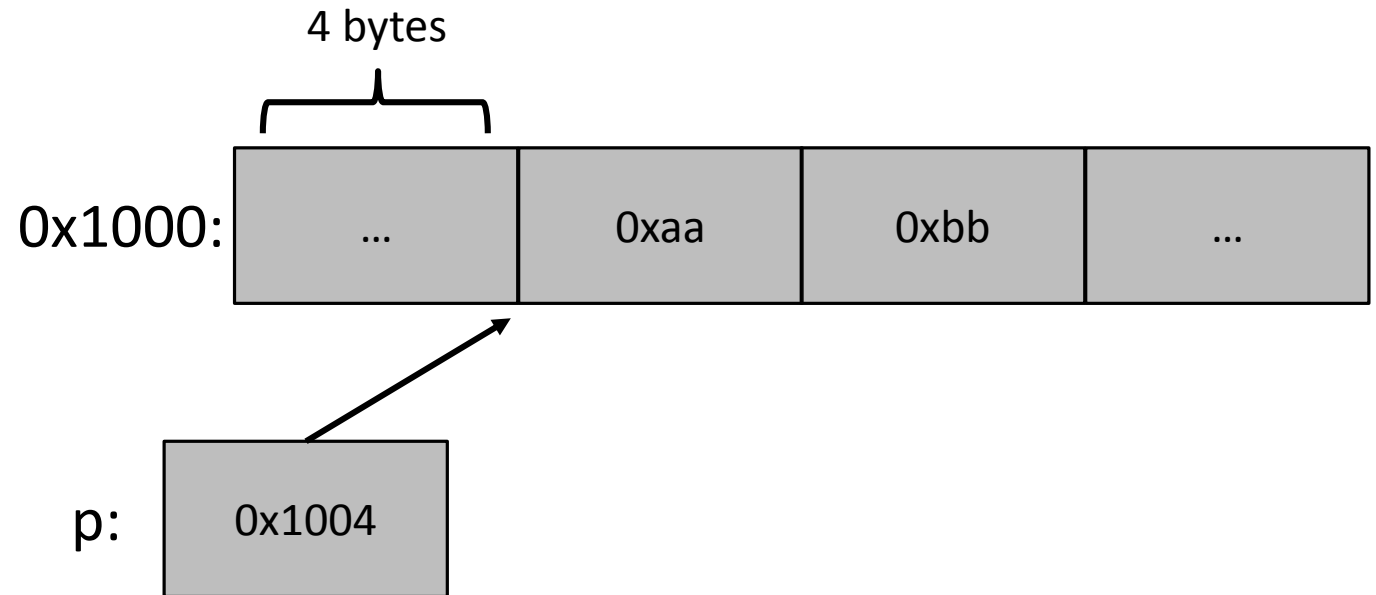How does an object of this class look in memory?

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

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

# Field Offsets

How does an object of this class look in memory?
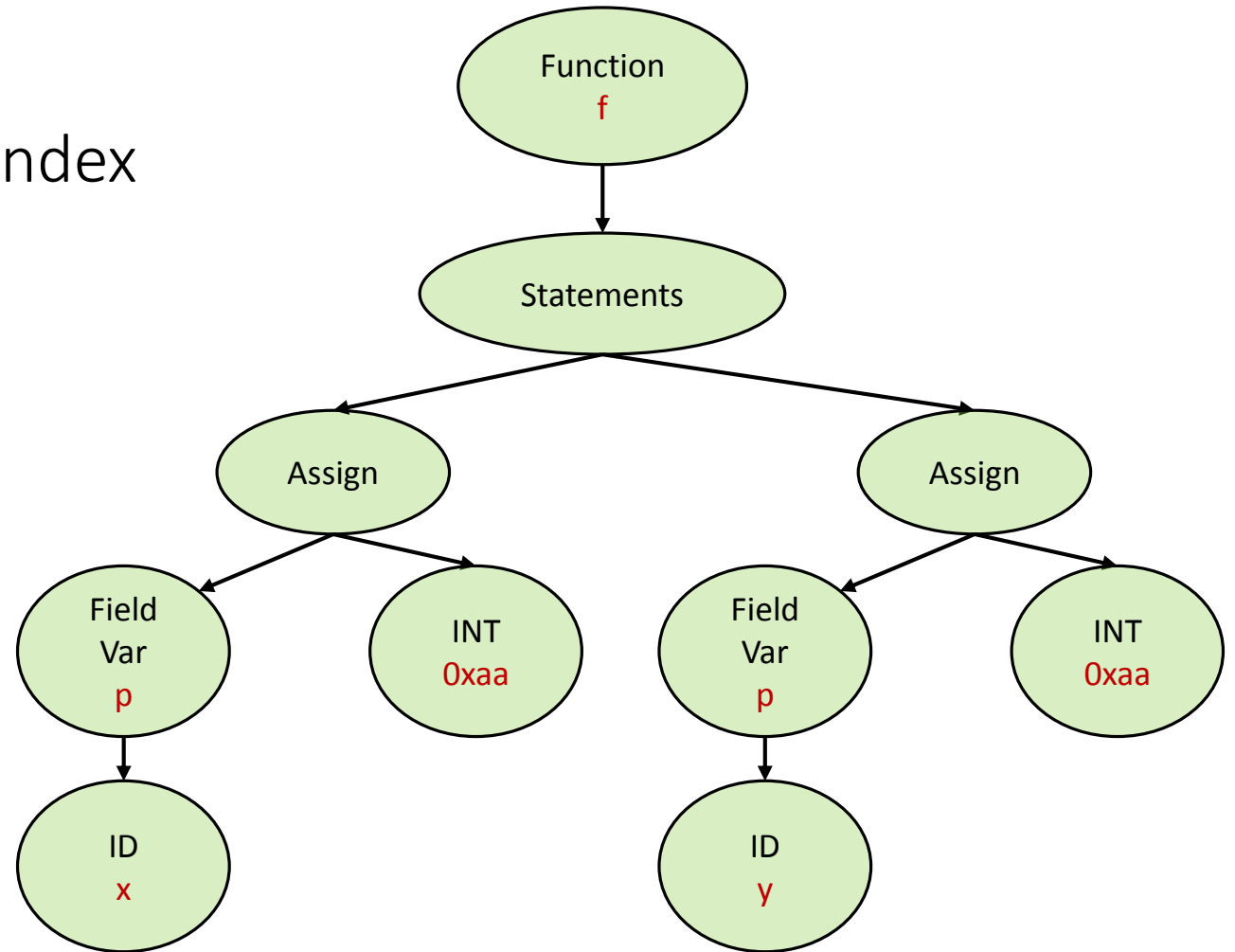
```
class Point {
  int x;
  int y;
}
void f(Point p) {
  p.x = 0xaa;
  p.y = 0xbb;
}
```

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

# Field Offsets

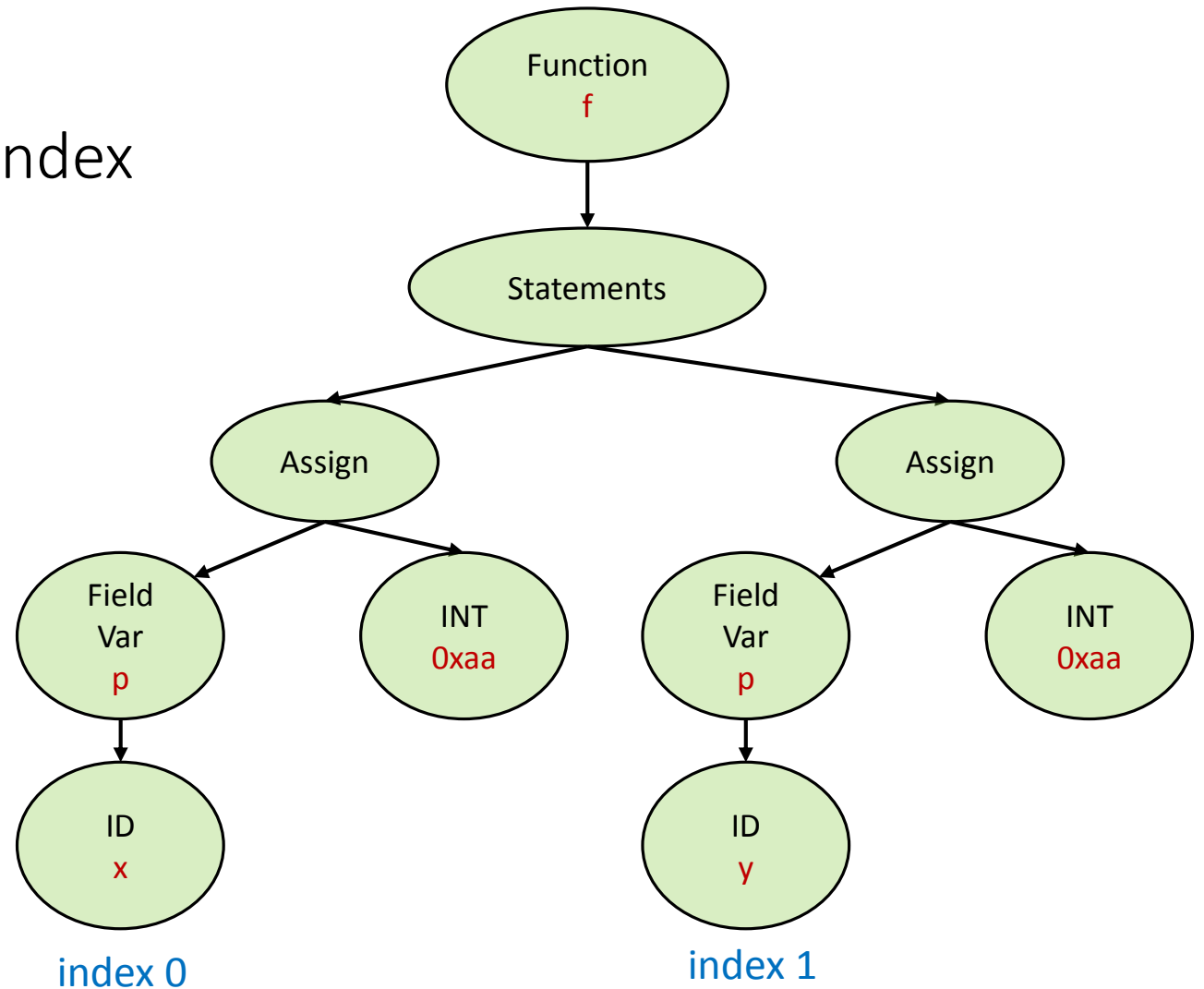Each class field should have an index

```
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;
}
```

# Type Sizes

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

```
f:
<prologue>
li $t0, 8
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, 8
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;
    string name;
}
void f() {
    Point p = new Point;
}
```

```
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 Point {
    int x;
    int y;
    string name;
}
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>
```