The University for world-class professionals

Manchester Metropolitan University

Compilers: Semantic Analysis and Intermediate Representations

Dr. Paris Yiapanis
room: John Dalton E151
email: p.yiapanis@mmu.ac.uk

Where we are…

- ~~Admin and overview~~
- ~~Lexical analysis~~
- ~~Parsing~~
- **Semantic analysis**
- Machine-independent optimisation
- Code generation
- Hardware architectures
- Machine-dependent optimisation
- Review

Objectives

- Revisit symbol tables
- Examine type checking
- Contrast different ways of handling types

Symbol Tables

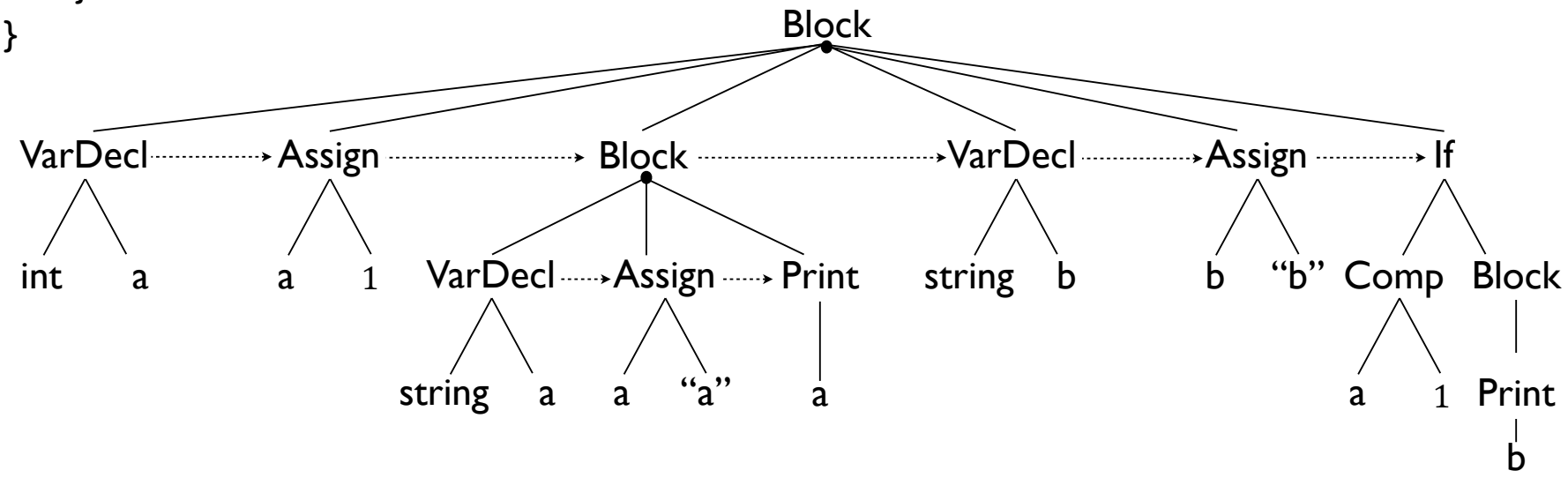
- Symbols are program entities like:
  - object fields
  - local variables
  - methods
  - classes
- A *symbol table* is a data structure that tracks symbols

Symbol tracking & basic semantic checking using a Symbol Table

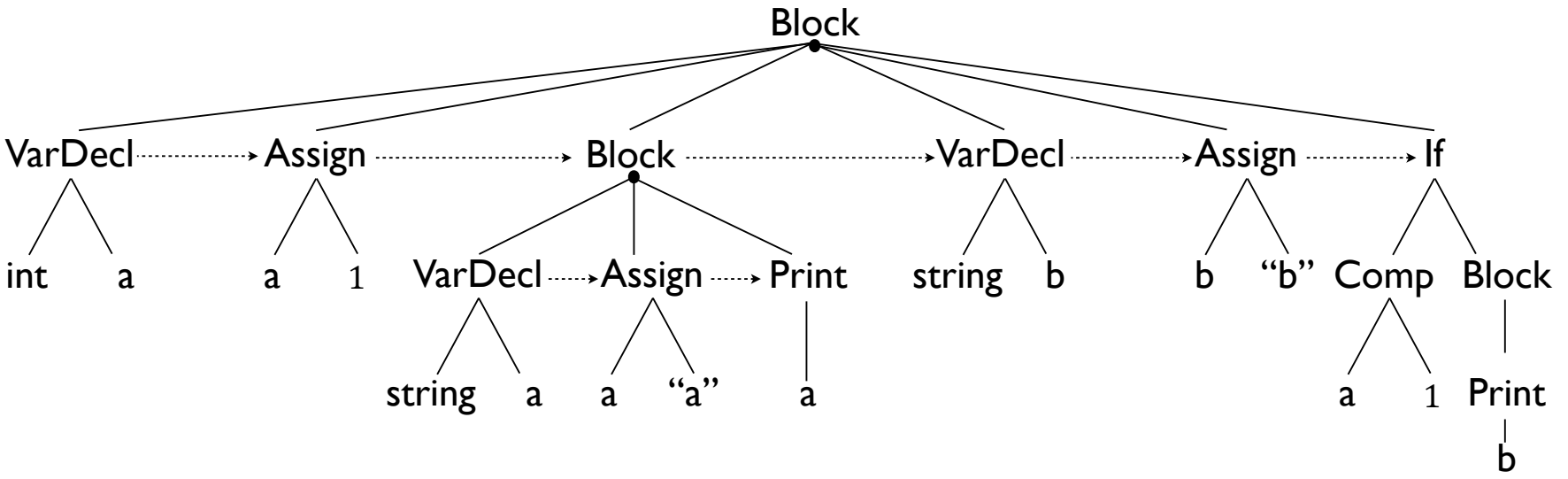
Source Code

```
{
    int a
    a = 1
    {
        string a
        a = "a"
        print(a)
    }
    string b
    b = "b"
    if (a == 1) {
        print(b)
    }
}
```

AST



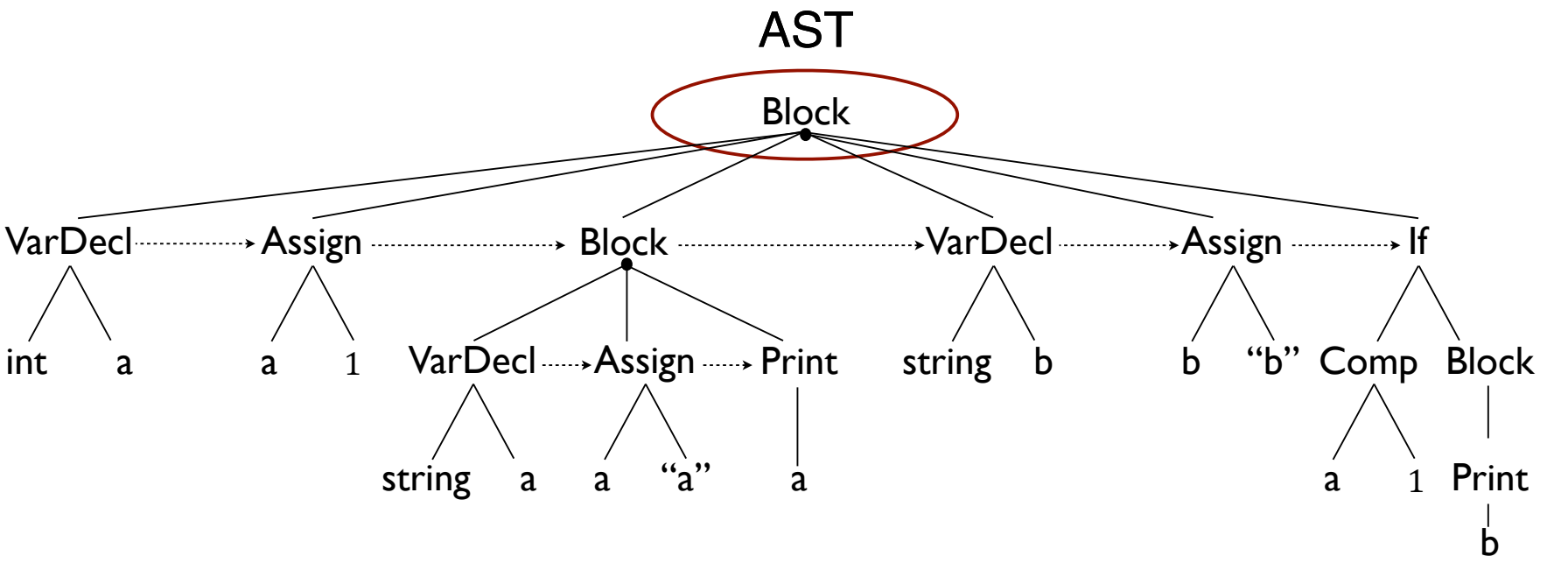
AST



Source Code

```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```

Symbol Table

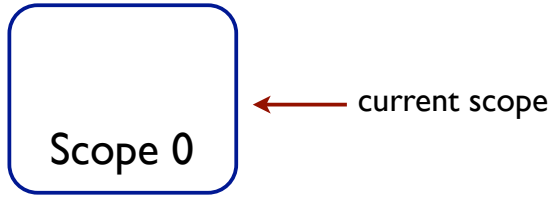


Initialize Scope 0

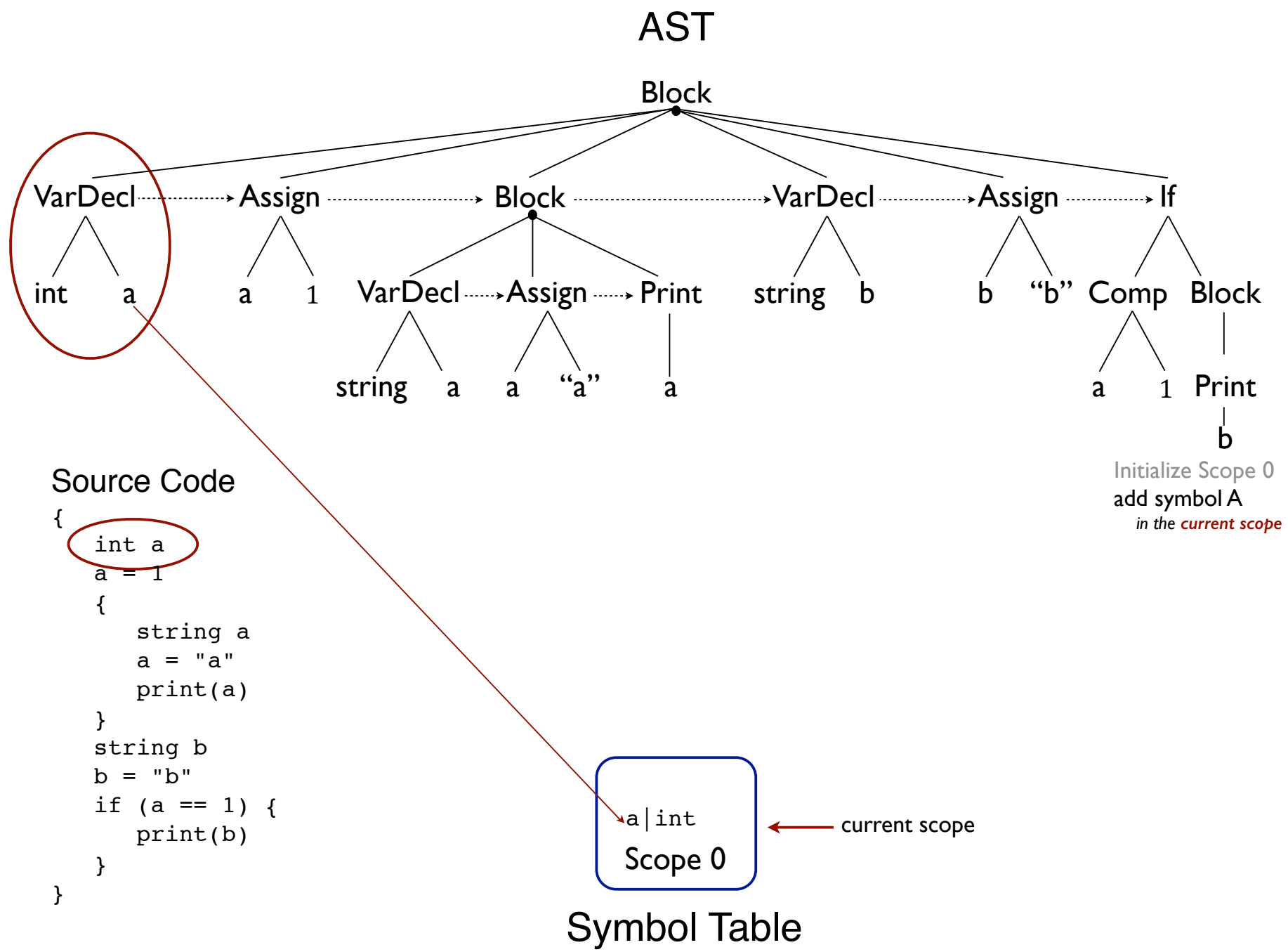
Set the ***current scope*** *pointer.*

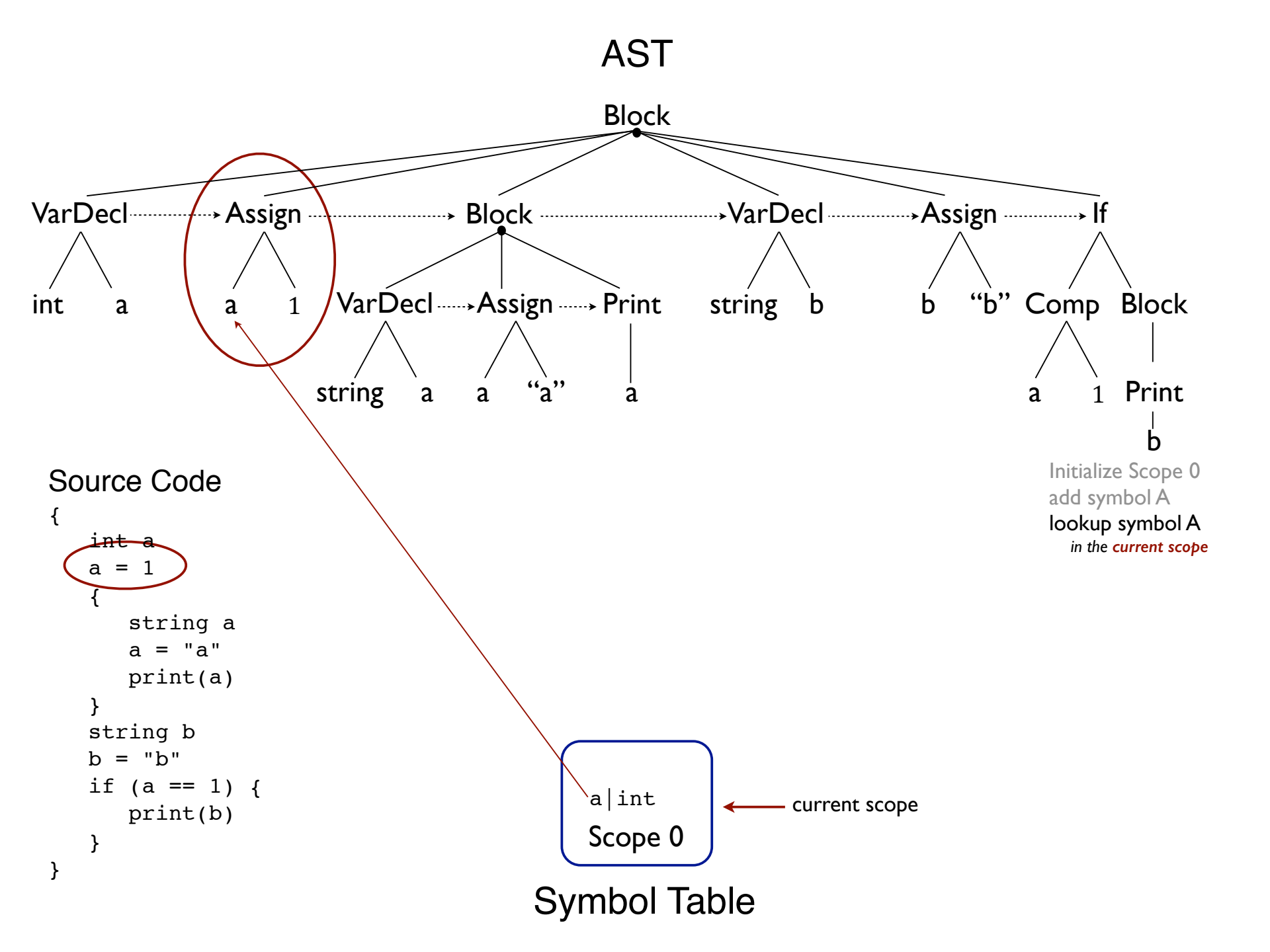
Source Code

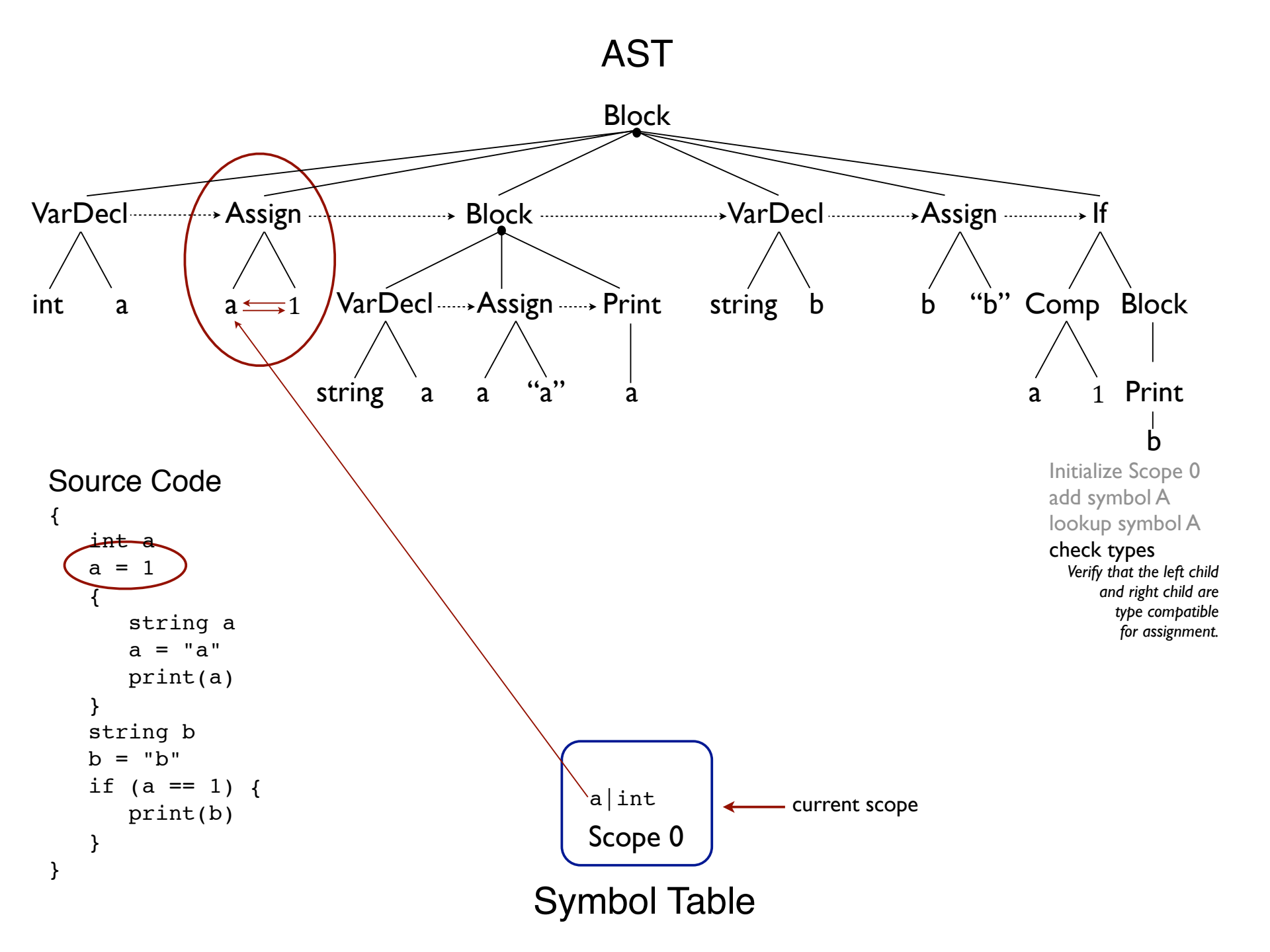
```
{
    int a
    a = 1
    {
       string a
       a = "a"
       print(a)
    }
    string b
    b = "b"
    if (a == 1) {
       print(b)
    }
}
```



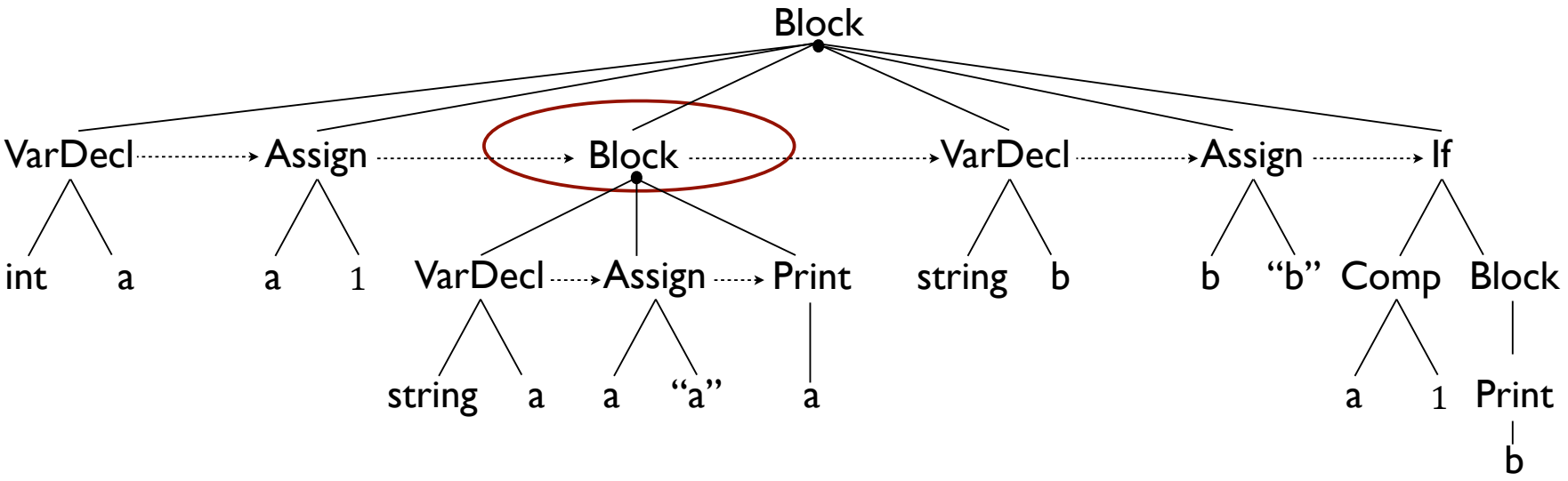
Symbol Table


AST
Block
VarDecl
Assign
Block
VarDecl
Assign
If
int
a
a
1
VarDecl
Assign
Print
string
b
b
“b”
Comp
Block
string
a
a
“a”
a
a
1
Print
b
Initialize Scope 0
add symbol A
in the current scope
Source Code
{
int a
a = 1
{
string a
a = "a"
print(a)
}
string b
b = "b"
if (a == 1) {
print(b)
}
}
a | int
Scope 0
current scope
Symbol Table


AST
Block
VarDecl
Assign
Block
VarDecl
Assign
If
int
a
a
1
VarDecl
Assign
Print
string
b
b
“b”
Comp
Block
string
a
a
“a”
a
a
1
Print
b
Initialize Scope 0
add symbol A
lookup symbol A
in the current scope
Source Code
{
int a
a = 1
{
string a
a = "a"
print(a)
}
string b
b = "b"
if (a == 1) {
print(b)
}
}
a | int
Scope 0
current scope
Symbol Table

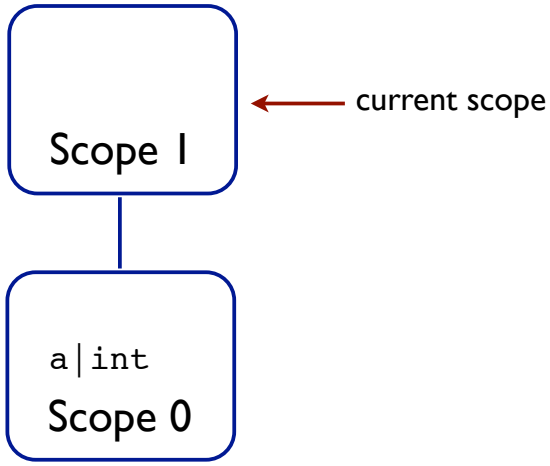

AST
Block
VarDecl
Assign
Block
VarDecl
Assign
If
int
a
a
1
VarDecl
Assign
Print
string
b
b
"b"
Comp
Block
string
a
a
"a"
a
a
1
Print
b
Initialize Scope 0
add symbol A
lookup symbol A
check types
Verify that the left child and right child are type compatible for assignment.
Source Code
{
int a
a = 1
{
string a
a = "a"
print(a)
}
string b
b = "b"
if (a == 1) {
print(b)
}
}
a | int
Scope 0
current scope
Symbol Table

AST



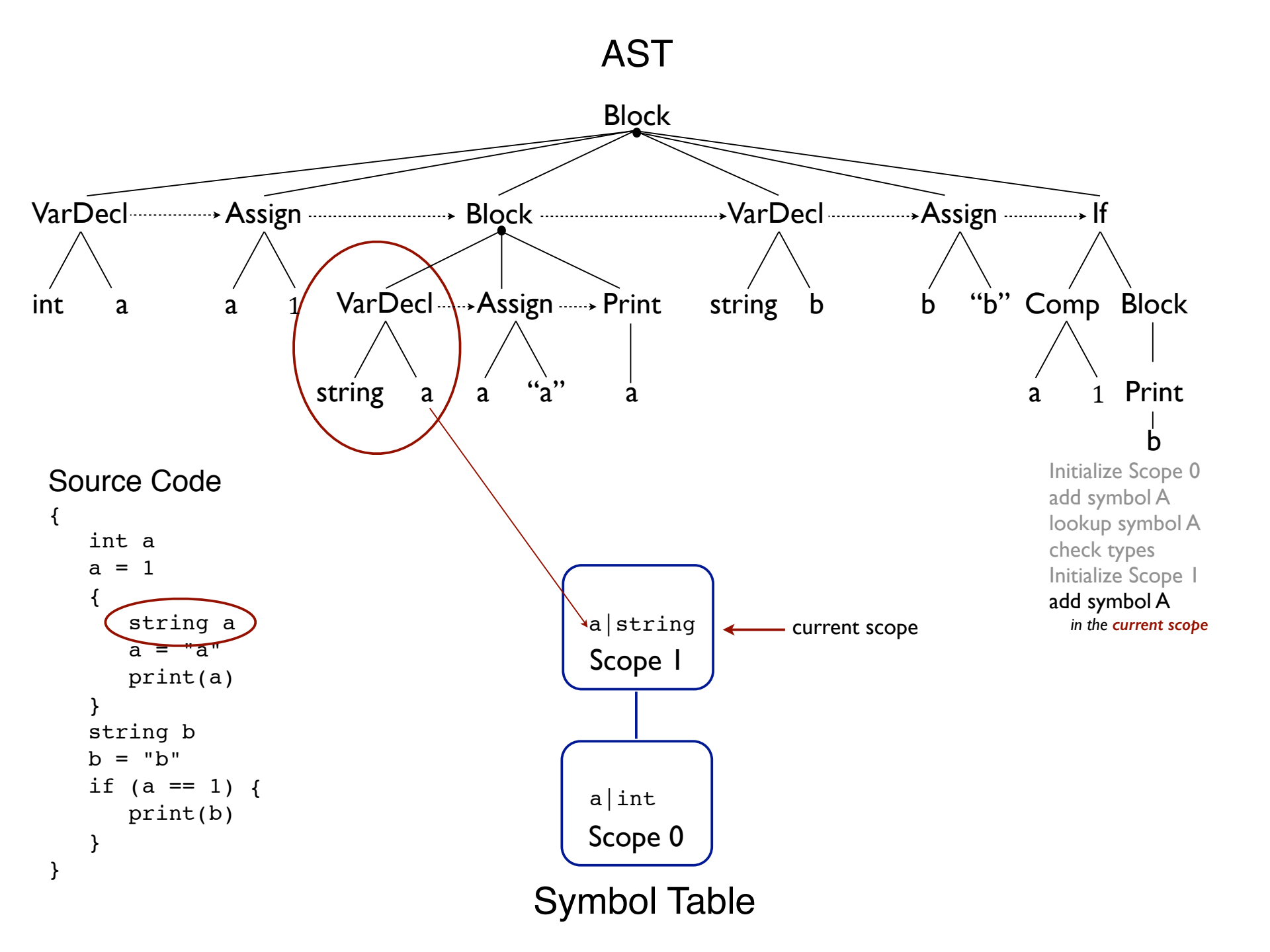
Source Code

```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```

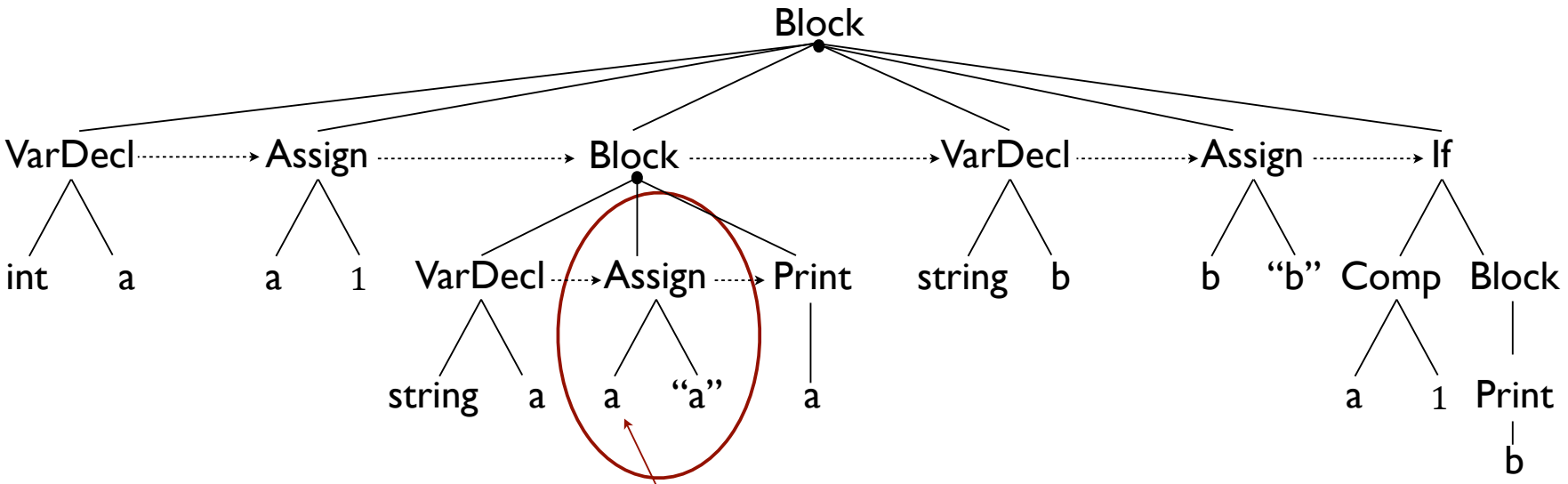


Initialize Scope 0
add symbol A
lookup symbol A
check types
Initialize Scope 1
*Move the **current scope** pointer to this child.*

Symbol Table

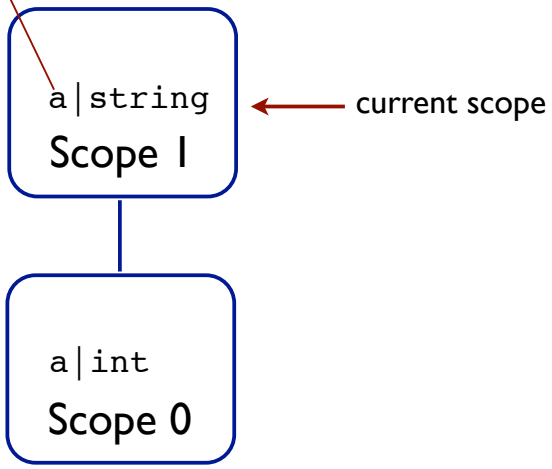

AST
Block
VarDecl
Assign
Block
VarDecl
Assign
If
int
a
a
1
VarDecl
Assign
Print
string
b
b
“b”
Comp
Block
string
a
a
“a”
a
a
1
Print
b
Source Code
{
int a
a = 1
{
string a
a = "a"
print(a)
}
string b
b = "b"
if (a == 1) {
print(b)
}
}
a|string
Scope 1
current scope
a|int
Scope 0
Symbol Table
Initialize Scope 0
add symbol A
lookup symbol A
check types
Initialize Scope 1
add symbol A
in the current scope

AST



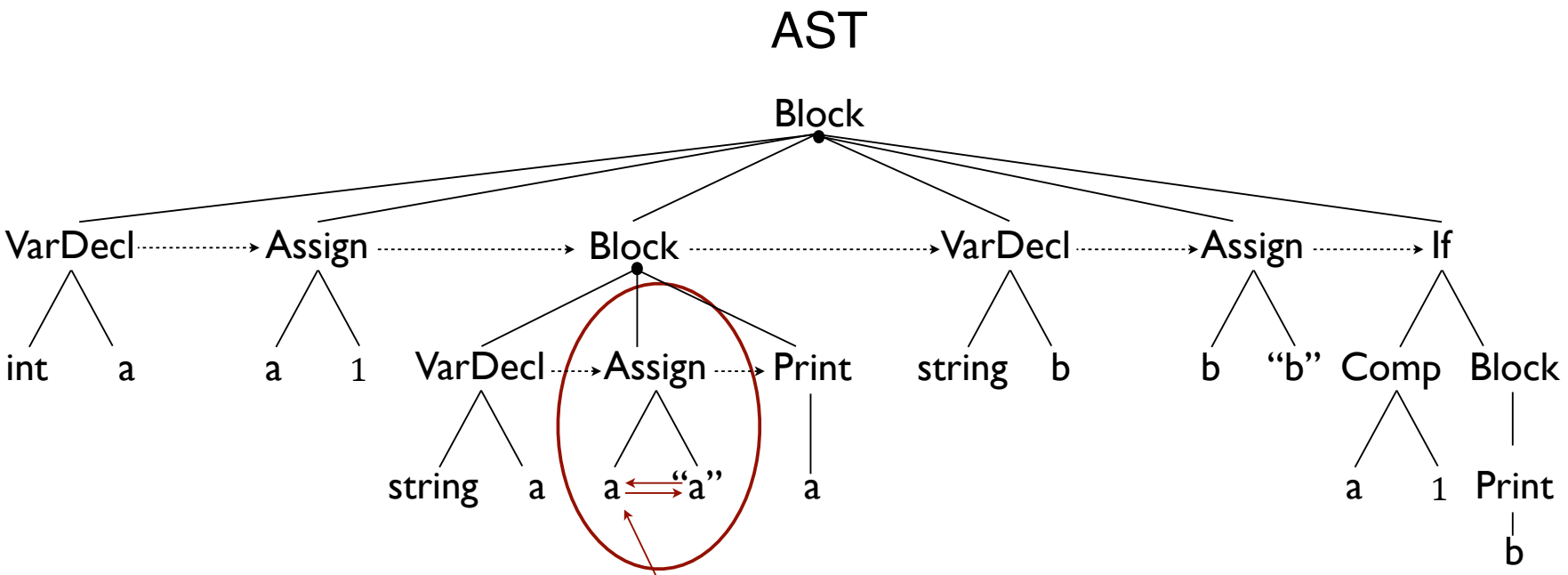
Source Code

```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```



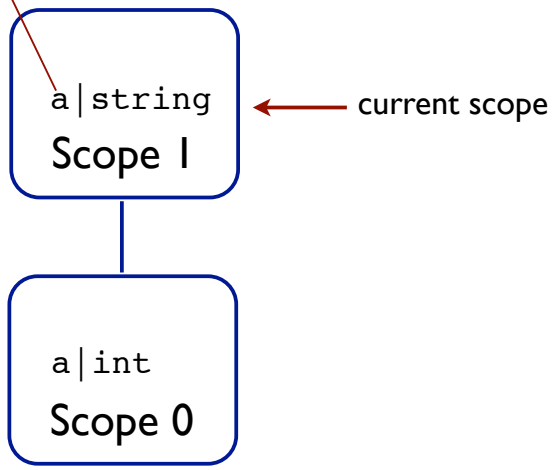
Initialize Scope 0
add symbol A
lookup symbol A
check types
Initialize Scope 1
add symbol A
lookup symbol A
*in the **current scope***

Symbol Table



Source Code

```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```

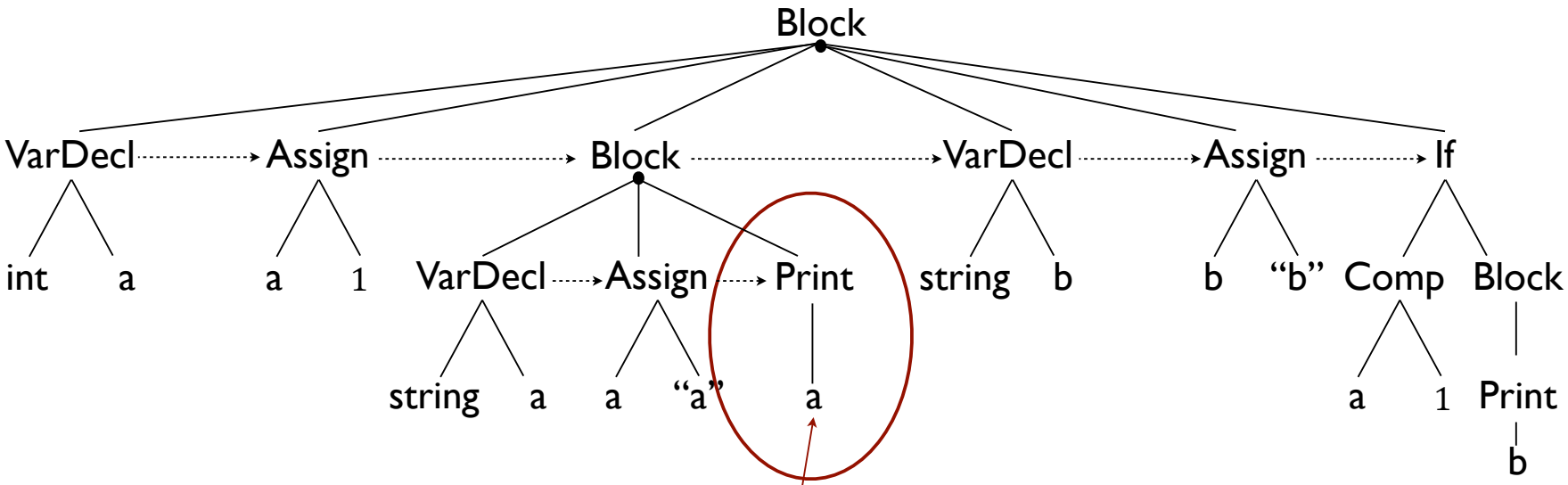


Symbol Table

Initialize Scope 0
add symbol A
lookup symbol A
check types
Initialize Scope 1
add symbol A
lookup symbol A
check types

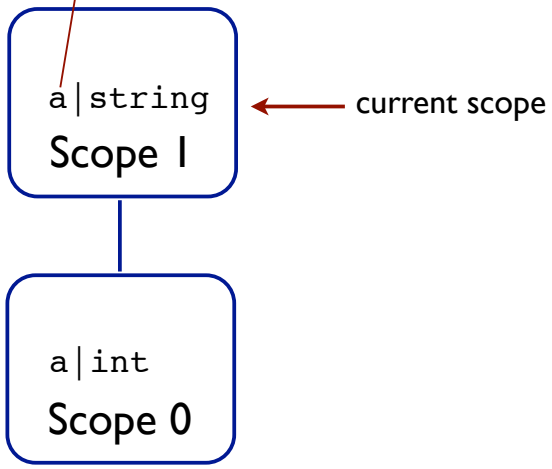
Verify that the left child and right child are type compatible for assignment.

AST



Source Code

```
{
    int a
    a = 1
    {
        string a
        a = "a"
        print(a)
    }
    string b
    b = "b"
    if (a == 1) {
        print(b)
    }
}
```

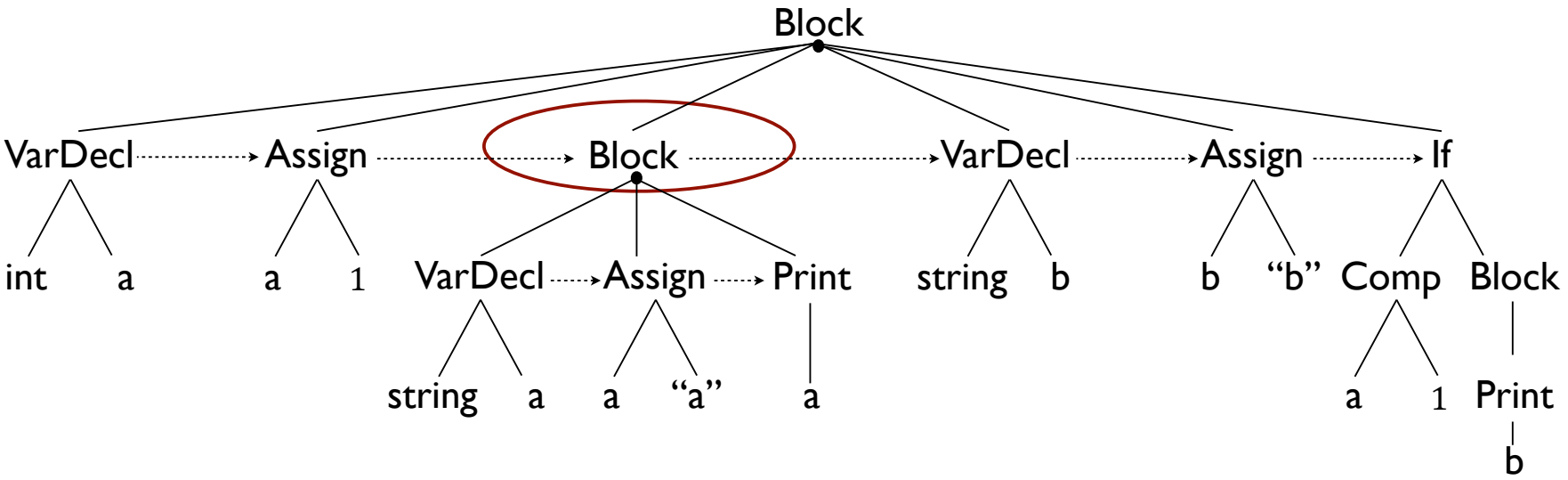


Symbol Table

Initialize Scope 0
add symbol A
lookup symbol A
check types
Initialize Scope 1
add symbol A
lookup symbol A
check types
lookup symbol A

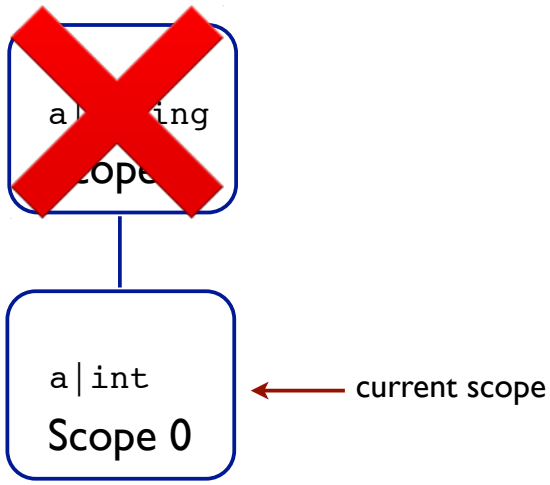
in the current scope.
Print can take any type, so there's no need to type check here.
We must still check the scope, of course!

AST



Source Code

```
{
    int a
    a = 1
    {
        string a
        a = "a"
        print(a)
    }
    string b
    b = "b"
    if (a == 1) {
        print(b)
    }
}
```

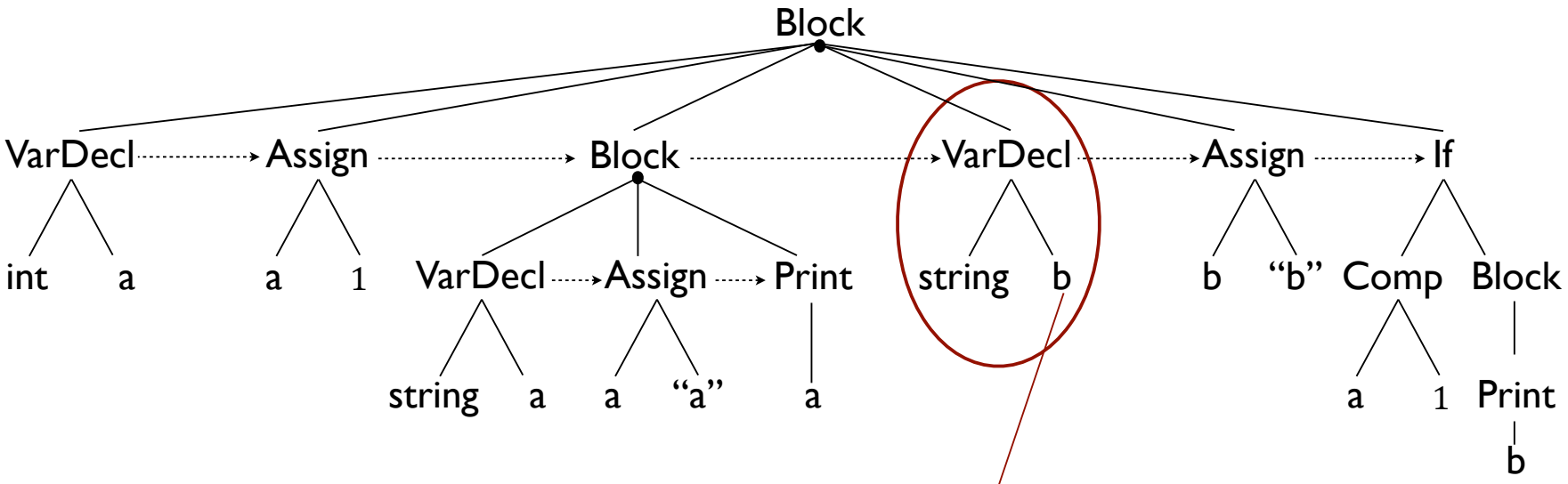


Initialize Scope 0
add symbol A
lookup symbol A
check types
Initialize Scope 1
add symbol A
lookup symbol A
check types
lookup symbol A
Close Scope 1

*Move the **current scope** pointer to its parent.*

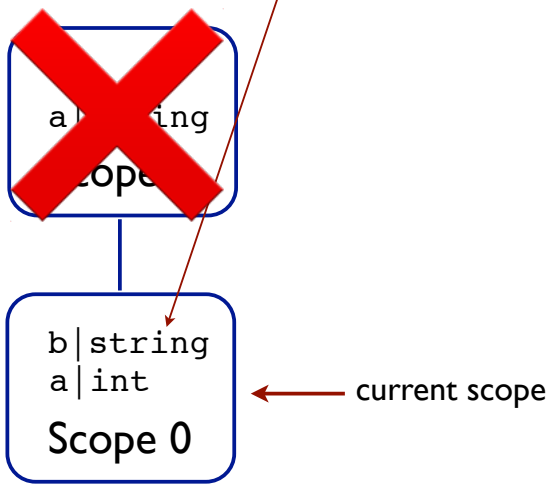
Symbol Table

AST



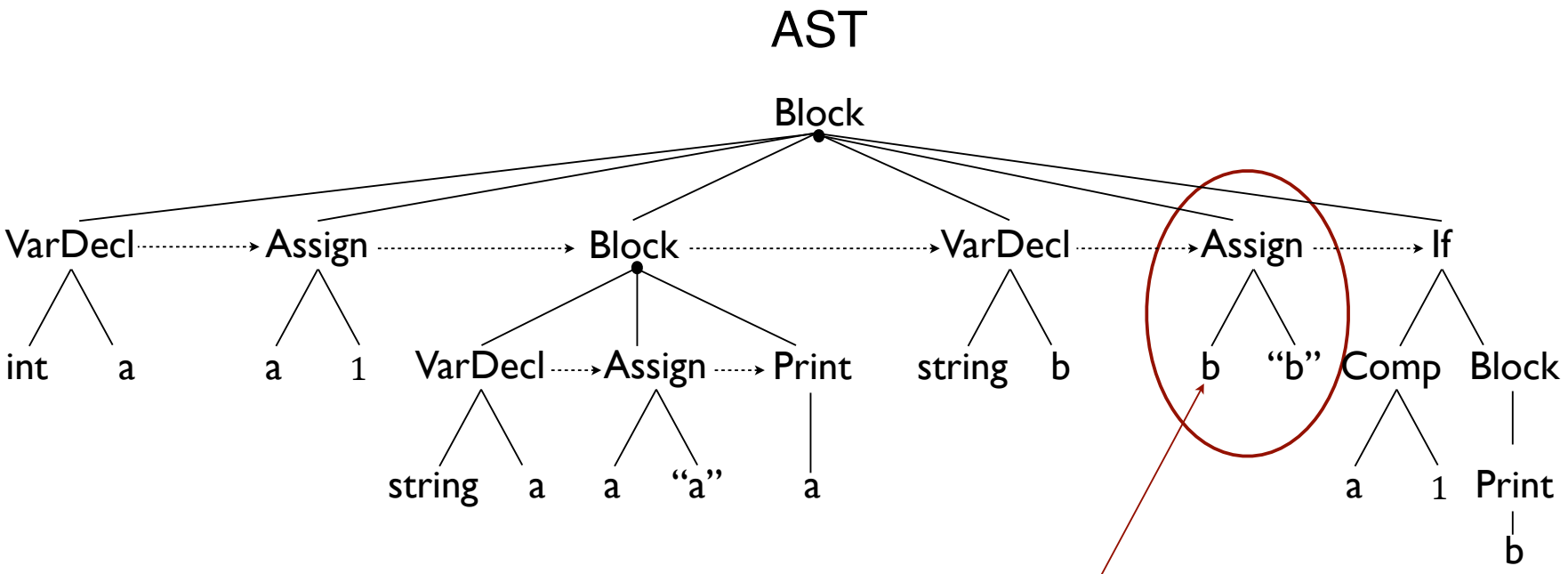
Source Code

```
{
    int a
    a = 1
    {
        string a
        a = "a"
        print(a)
    }
    string b
    b = "b"
    if (a == 1) {
        print(b)
    }
}
```



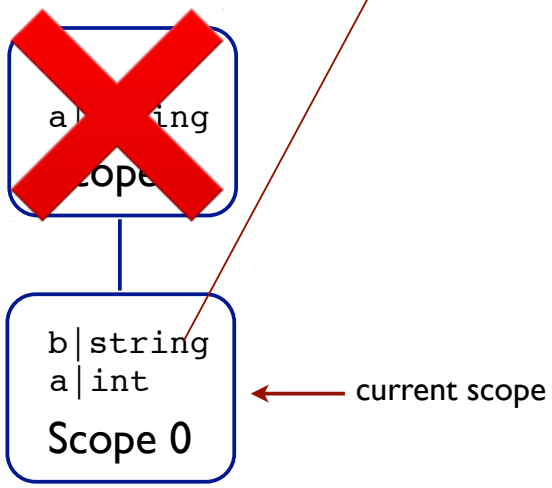
Initialize Scope 0
add symbol A
lookup symbol A
check types
Initialize Scope 1
add symbol A
lookup symbol A
check types
lookup symbol A
Close Scope 1
add symbol B
in the current scope

Symbol Table



Source Code

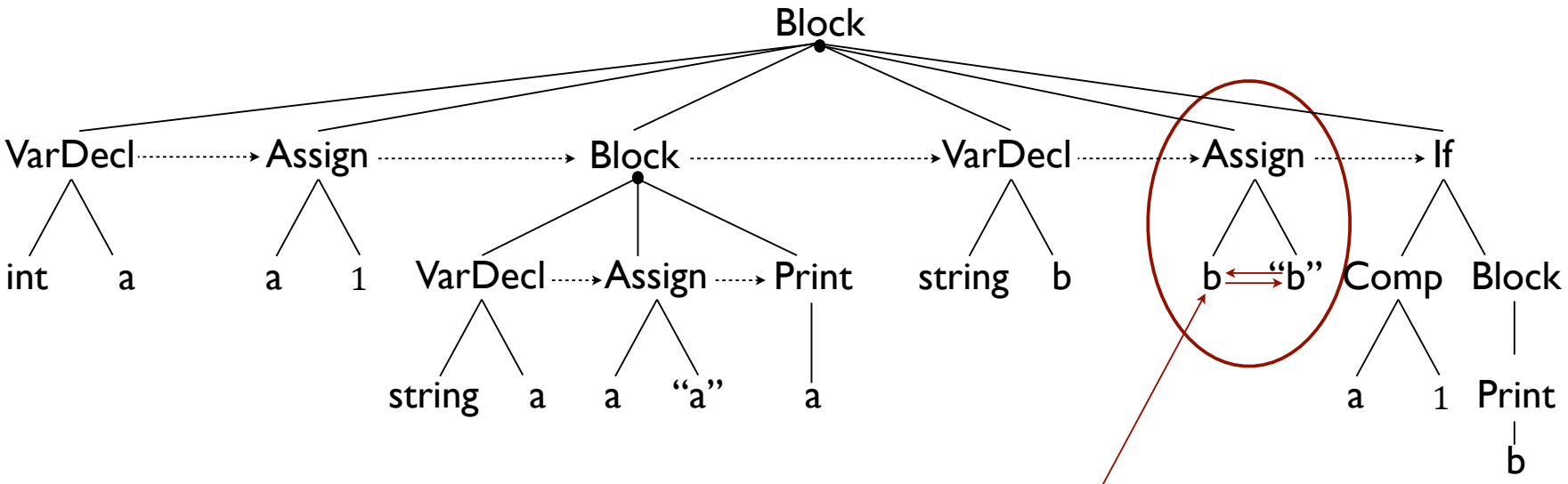
```
{
    int a
    a = 1
    {
        string a
        a = "a"
        print(a)
    }
    string b
    b = "b"
    if (a == 1) {
        print(b)
    }
}
```



Symbol Table

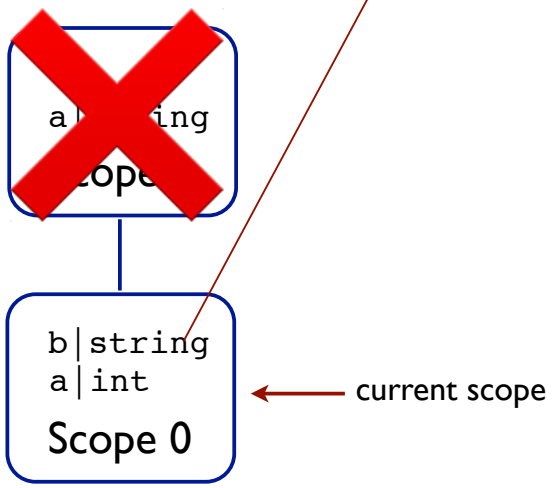
Initialize Scope 0
add symbol A
lookup symbol A
check types
Initialize Scope 1
add symbol A
lookup symbol A
check types
lookup symbol A
Close Scope 1
add symbol B
lookup symbol B
in the current scope

AST



Source Code

```
{
    int a
    a = 1
    {
        string a
        a = "a"
        print(a)
    }
    string b
    b = "b"
    if (a == 1) {
        print(b)
    }
}
```

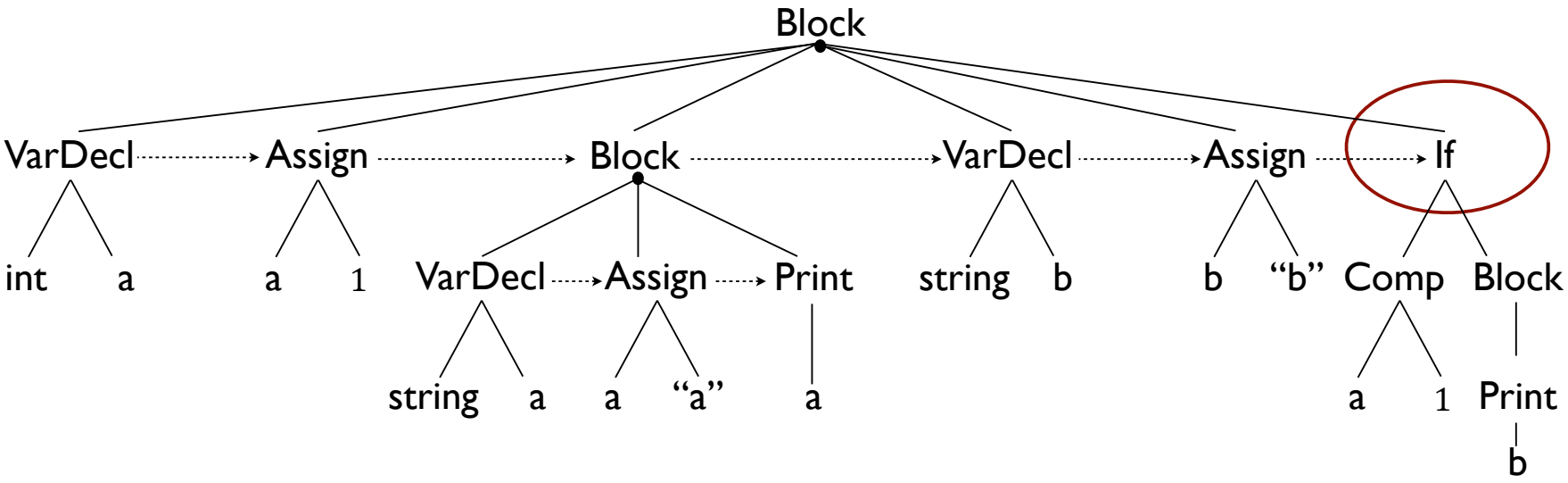


Initialize Scope 0
add symbol A
lookup symbol A
check types
Initialize Scope 1
add symbol A
lookup symbol A
check types
lookup symbol A
Close Scope 1
add symbol B
lookup symbol B
check types

Verify that the left child and right child are type compatible for assignment.

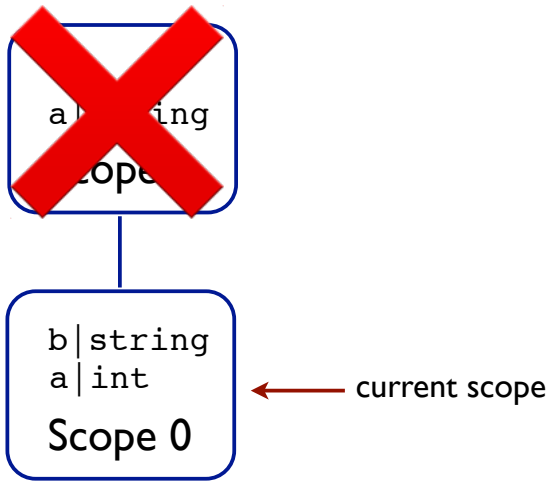
Symbol Table

AST



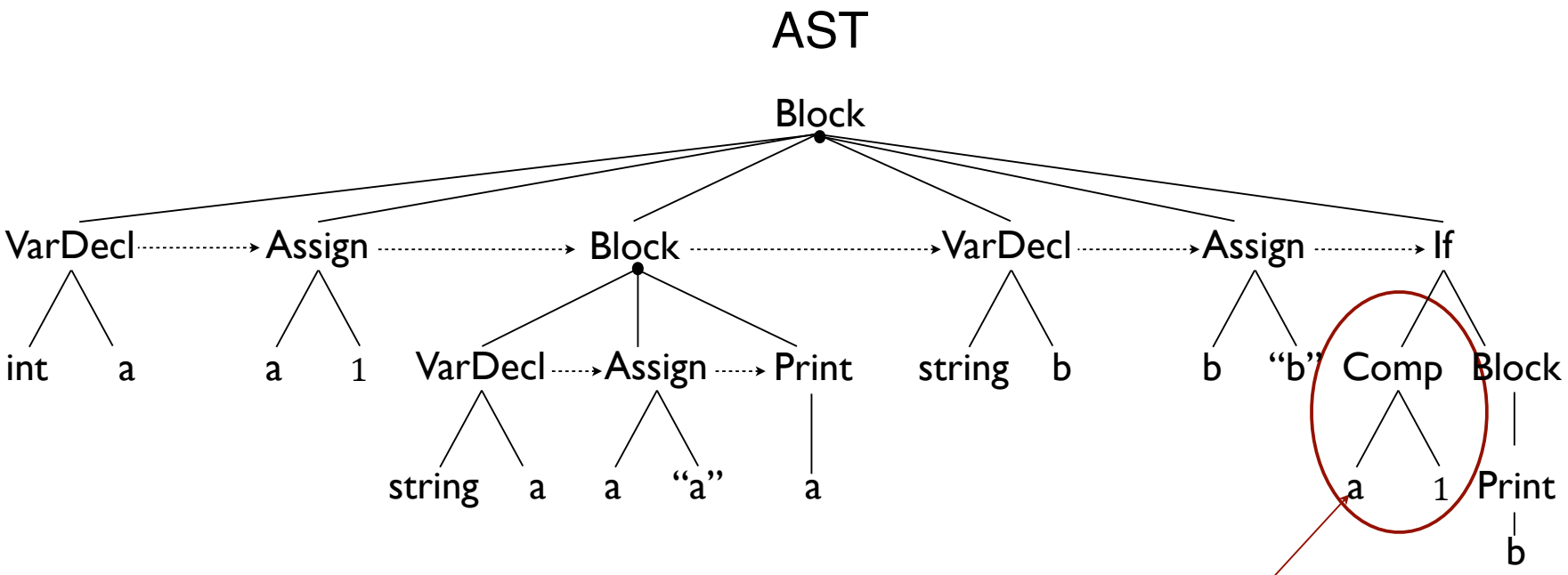
Source Code

```
{
    int a
    a = 1
    {
        string a
        a = "a"
        print(a)
    }
    string b
    b = "b"
    if (a == 1) {
        print(b)
    }
}
```



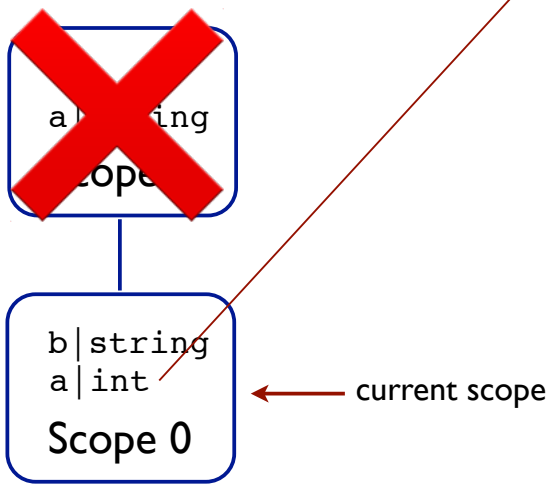
Initialize Scope 0
add symbol A
lookup symbol A
check types
Initialize Scope 1
add symbol A
lookup symbol A
check types
lookup symbol A
Close Scope 1
add symbol B
lookup symbol B
check types

Symbol Table



Source Code

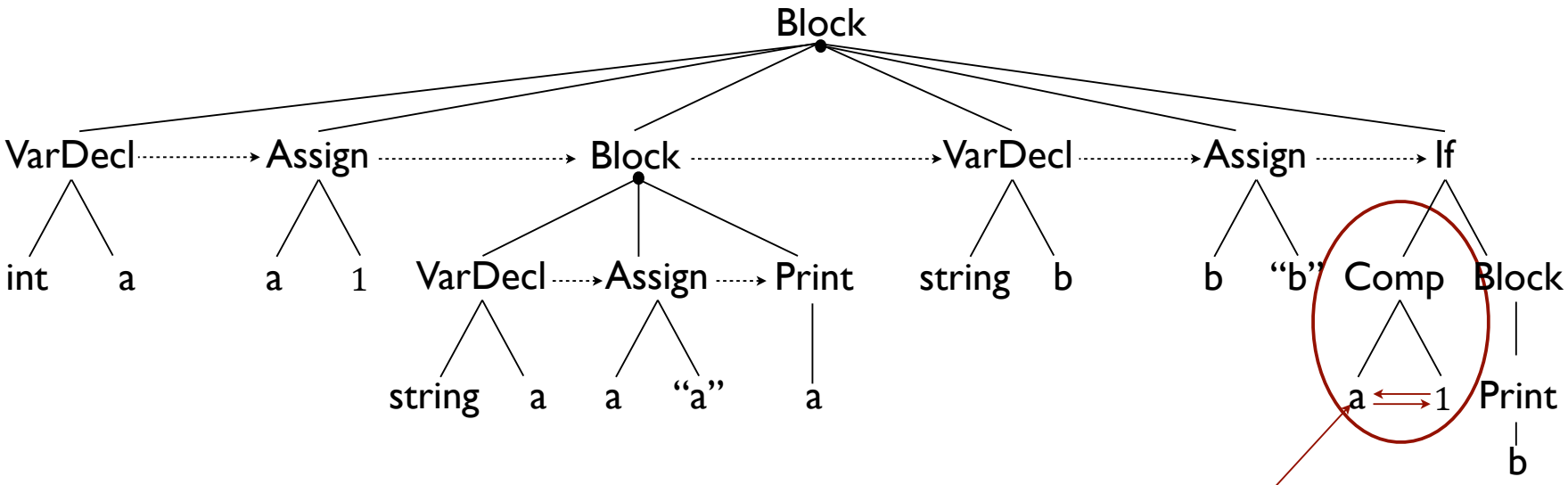
```
{
    int a
    a = 1
    {
        string a
        a = "a"
        print(a)
    }
    string b
    b = "b"
    if (a == 1) {
        print(b)
    }
}
```



Initialize Scope 0
add symbol A
lookup symbol A
check types
Initialize Scope 1
add symbol A
lookup symbol A
check types
lookup symbol A
Close Scope 1
add symbol B
lookup symbol B
check types
lookup symbol A
in the current scope

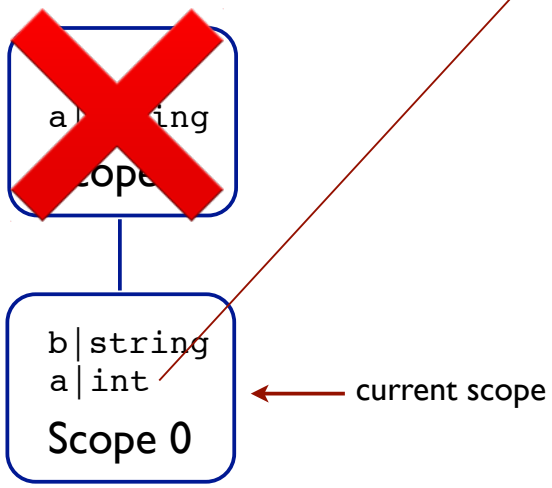
Symbol Table

AST



Source Code

```
{
    int a
    a = 1
    {
        string a
        a = "a"
        print(a)
    }
    string b
    b = "b"
    if (a == 1) {
        print(b)
    }
}
```

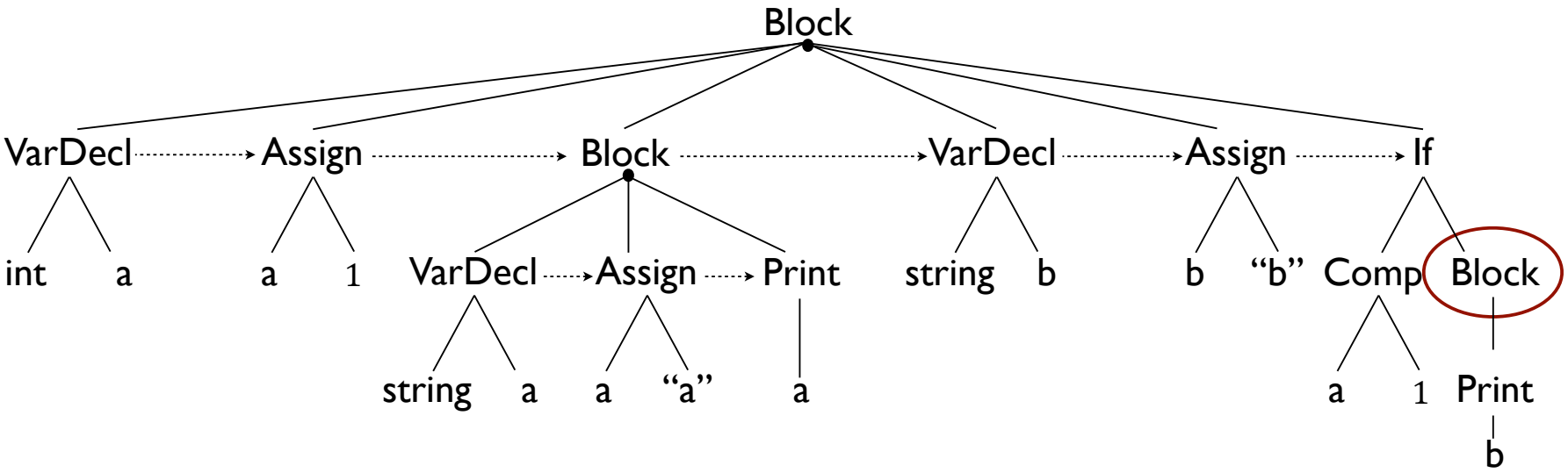


Symbol Table

Initialize Scope 0
add symbol A
lookup symbol A
check types
Initialize Scope 1
add symbol A
lookup symbol A
check types
lookup symbol A
Close Scope 1
add symbol B
lookup symbol B
check types
lookup symbol A
check types

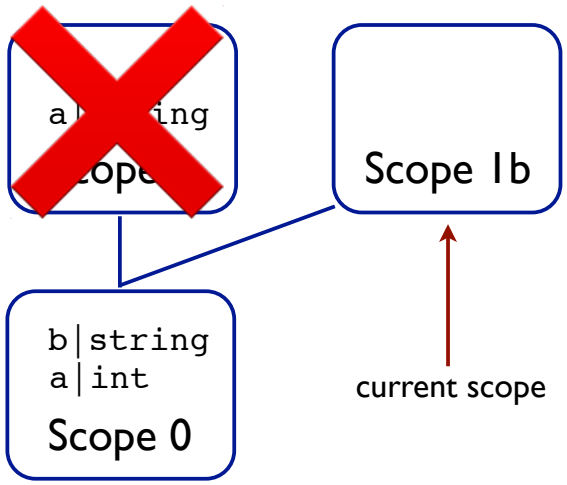
Verify that the left child and right child are ***type comparable***

AST



Source Code

```
{
    int a
    a = 1
    {
        string a
        a = "a"
        print(a)
    }
    string b
    b = "b"
    if (a == 1) {
        print(b)
    }
}
```

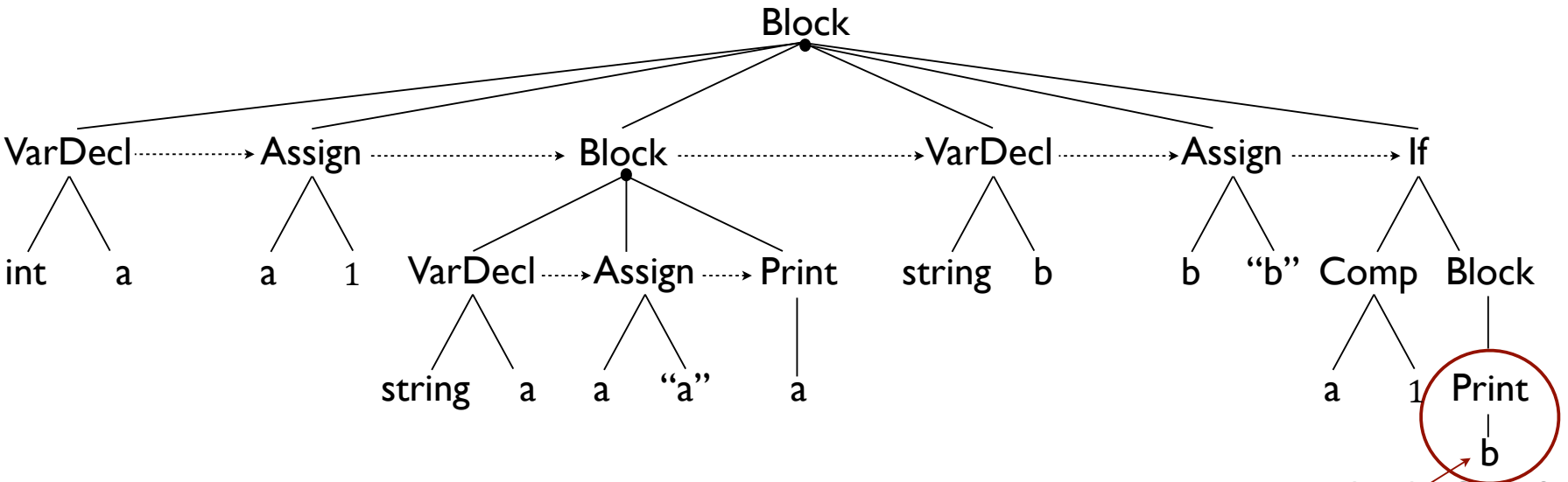


Symbol Table

Initialize Scope 0
add symbol A
lookup symbol A
check types
Initialize Scope 1
add symbol A
lookup symbol A
check types
lookup symbol A
Close Scope 1
add symbol B
lookup symbol B
check types
lookup symbol A
check types
Initialize Scope 1b

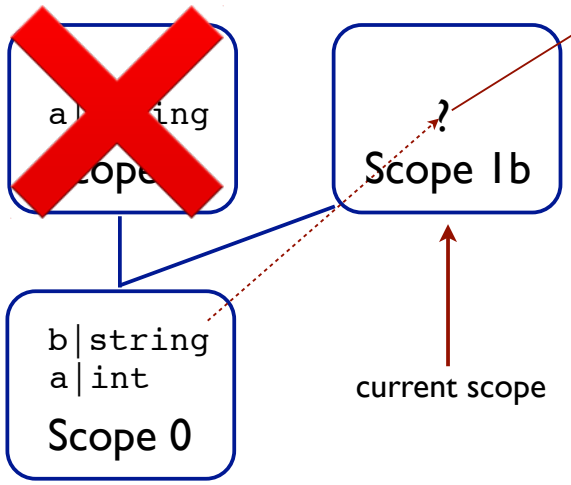
*Move the **current scope** pointer to this child.*

AST



Source Code

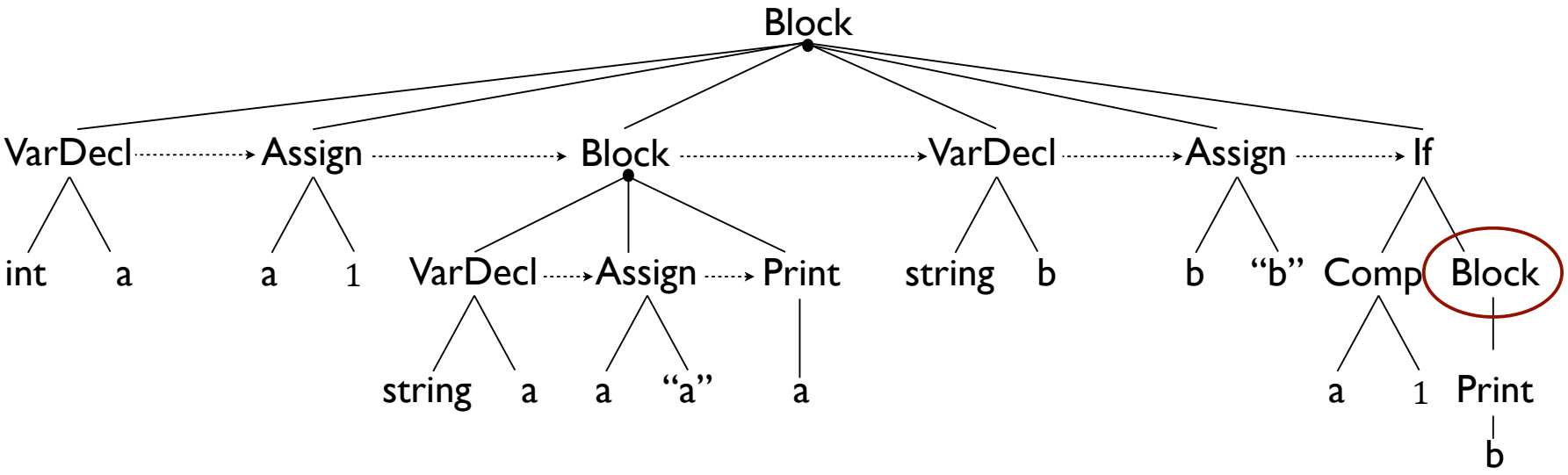
```
{
    int a
    a = 1
    {
        string a
        a = "a"
        print(a)
    }
    string b
    b = "b"
    if (a == 1) {
        print(b)
    }
}
```



Symbol Table

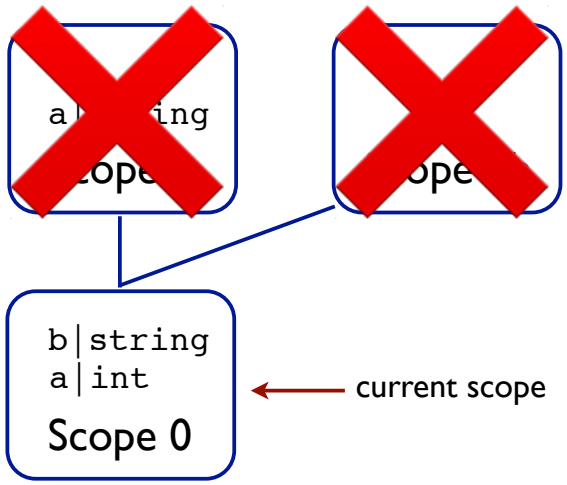
Initialize Scope 0
add symbol A
lookup symbol A
check types
Initialize Scope 1
add symbol A
lookup symbol A
check types
lookup symbol A
Close Scope 1
add symbol B
lookup symbol B
check types
lookup symbol A
check types
Initialize Scope 1b
lookup symbol B
*in the **current scope**.*
Print can take any type.

AST



Source Code

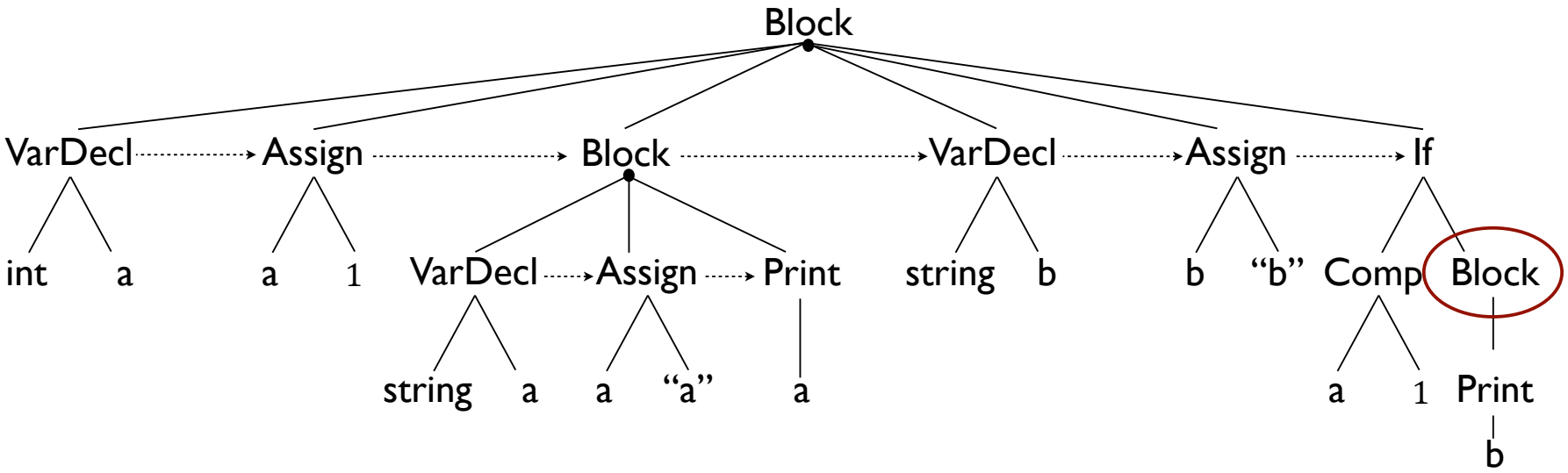
```
{
    int a
    a = 1
    {
        string a
        a = "a"
        print(a)
    }
    string b
    b = "b"
    if (a == 1) {
        print(b)
    }
}
```



Symbol Table

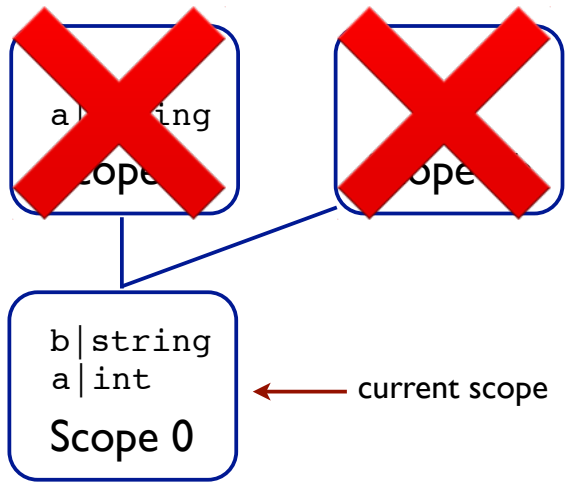
Initialize Scope 0
add symbol A
lookup symbol A
check types
Initialize Scope 1
add symbol A
lookup symbol A
check types
lookup symbol A
Close Scope 1
add symbol B
lookup symbol B
check types
lookup symbol A
check types
Initialize Scope 1b
lookup symbol B
Close Scope 1b

AST



Source Code

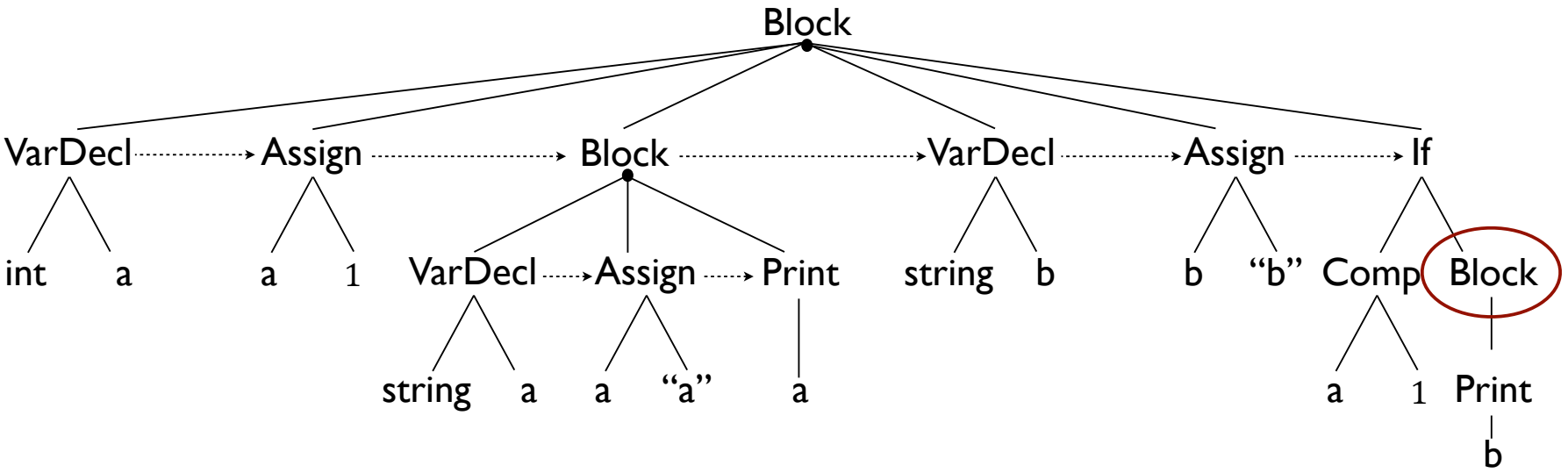
```
{
    int a
    a = 1
    {
        string a
        a = "a"
        print(a)
    }
    string b
    b = "b"
    if (a == 1) {
        print(b)
    }
}
```



Symbol Table

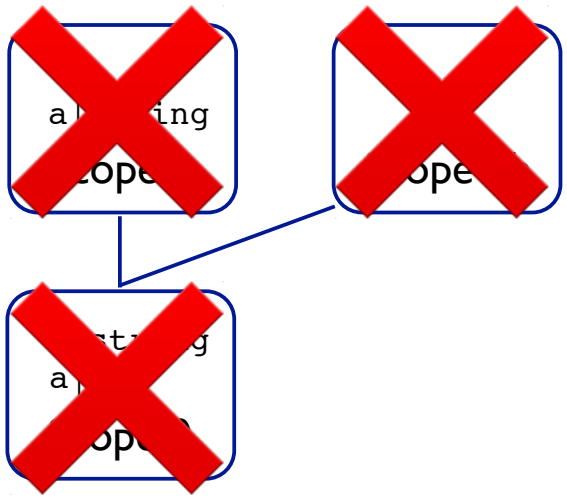
Initialize Scope 0
add symbol A
lookup symbol A
check types
Initialize Scope 1
add symbol A
lookup symbol A
check types
lookup symbol A
Close Scope 1
add symbol B
lookup symbol B
check types
lookup symbol A
check types
Initialize Scope 1b
lookup symbol B
Close Scope 1b
Close Scope 0

AST



Source Code

```
{
    int a
    a = 1
    {
        string a
        a = "a"
        print(a)
    }
    string b
    b = "b"
    if (a == 1) {
        print(b)
    }
}
```



Symbol Table

Initialize Scope 0
add symbol A
lookup symbol A
check types
Initialize Scope 1
add symbol A
lookup symbol A
check types
lookup symbol A
Close Scope 1
add symbol B
lookup symbol B
check types
lookup symbol A
check types
Initialize Scope 1b
lookup symbol B
Close Scope 1b
Close Scope 0

Type Checking

Type Checking

- Most extensive of the semantic checks
- Examples:
  - Method arguments matches the number of formals and the corresponding types are equivalent
  - If method is called as an expression, should return a type
  - Identifiers in an expression should be “evaluatable”
  - LHS of an assignment should be “assignable”
  - In an expression all the types of variables, method return types and operators should be “compatible”

What is a Type?

- A type is:
  - A set of values.
  - A set of operations on those values
- e.g.
  - int: +, -, >=, <
  - String: concat, isNull?
- **Type errors** arise when operations are performed on values that do not support that operation.

- Consider the assembly language fragment

```
add $r1, $r2, $r3
```

- What are the types of $r1, $r2, $r3?

Types

- Certain operations are legal for values of each type
  - It doesn't make sense to add a function pointer and an integer in C
  - It does make sense to add two integers
  - But both have the same assembly language implementation

Types of Type-Checking

- Static type checking:
  - Analyze the program during compile-time to prove the absence of type errors
  - Never let bad things happen at runtime
  - Restricts the kind of programs your can write
  - e.g. Java, C
- Dynamic type checking:
  - Check operations at runtime before performing them
  - But, usually less efficient (due to runtime checks)
  - Less restrictive than static type checking
  - e.g. Lisp, Python
- No type checking:
  - e.g. machine code

Categories of Types

- Base types
  - int, float, double, char, boolean, etc
- Compound types
  - arrays, pointers, records, structs, unions, classes, etc
- Complex types
  - lists, stacks, queues, trees, heaps, tables
  - AKA *abstract data types*
  - language may or may not support them

So How *Do* You Perform Type Checking?

- Essentially a process called *type inference*.
  - User declares types for identifiers
  - Complier infers types for expressions
- For example, if **a** and **b** are integers and there is an expression of the form

  a + b

  the type of the expression must be integer.
- What is the type of the expression

  a + 5.0 ?

Involves *type conversion*

Type conversion

- Explicit conversion (*aka* Casting)
  - e.g. (double) a + 5.0
- Implicit conversion (*aka* Coercion)
  - widening or narrowing conversions
  - compiler finds a type error and changes the type of the variable to an appropriate type

Examples:

```
int a = 2;
double b = 5.0;

a = b + 2;
b = a + 7;
b = a / 4;
b = a / 4.0;
```

Run the code in Java and print out the result after each statement to find the answer.

Type Equivalence

- How do you decide if two types are equal?
  - Example:

```
int A[128];
foo(A);

foo(int B[128]) { ... }
```

- Two different type entries in different symbol tables
  - But they should be the same

Name versus Structural Equivalence

- For name equivalence, two types must have the same name to be considered equivalent
- For structural equivalence, if the type expression of two types have the same construction, then they are equivalent
- “Same construction”
  - Equivalent base types

Structural Equivalence Example

```
type student = record
    name, address : string
    age : integer

type school = record
    name, address : string
    age : integer

x : student;
y : school;
```

x and y are structurally equivalent but not name equivalent.
Name equivalence is based on the assumption that if the programmer goes to the effort of writing two type definitions, then those definitions are probably meant to represent different types.

Summary

- Explain the use and (conceptual) construction of symbol tables

  You should be implementing these ideas in your compiler
- Static type checking versus dynamic type checking
- How to perform type checking and type conversions
- Structural versus name equivalence of types

Where we are…

- ~~Admin and overview~~
- ~~Lexical analysis~~
- ~~Parsing~~
- ~~Semantic analysis~~
- Machine-independent optimisation
- Code generation
- Hardware architectures
- Machine-dependent optimisation
- Review