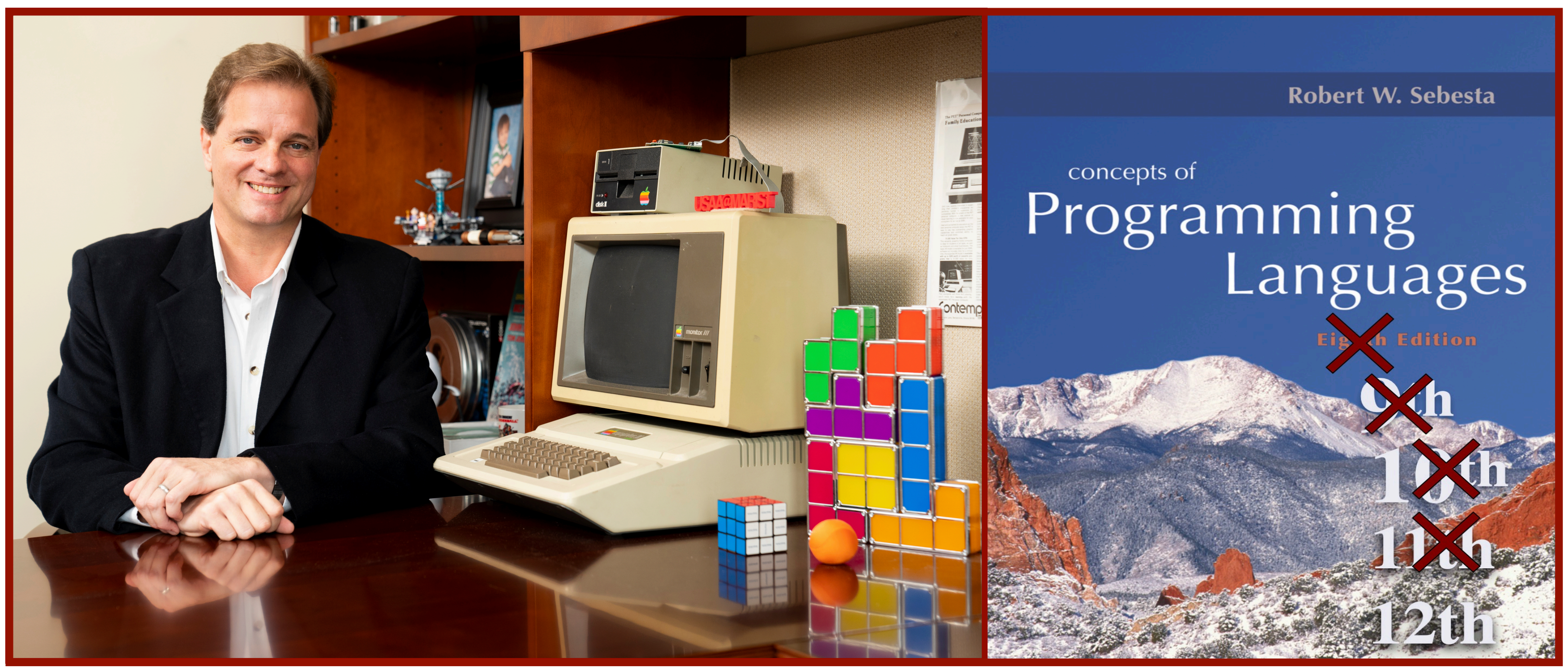
# Scope and Type




Alan G. Labouseur, Ph.D.
Alan.Labouseur@Marist.edu

# Scope and Type

There are many aspects of data types to consider :

- Static and Dynamic types
- Expressing Type Systems
- Checking Scope and Type in a Program

# Identifiers, Variables, and Scope

Identifiers =? Names

- Used for namespaces, classes, methods, variables, etc.

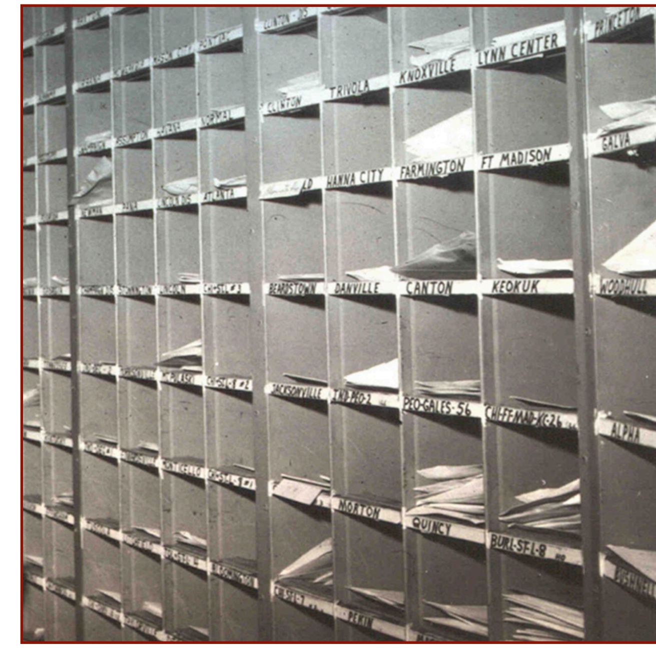
Design issues for names

- Are names case sensitive?
- Are special words *reserved* words or *key*words? (What's the difference)
- How many characters can be in an identifier names?
  - If they're too short they cannot be meaningful.
  - If they're too long they might get unwieldy.
  - Examples
    - FORTRAN I: maximum 6
    - COBOL: maximum 30
    - FORTRAN 90 and C89: maximum 31
    - C99: maximum 63
    - C#, Ada, Java: no limit (in theory, not in practice)

# Identifiers, Variables, and Scope

A variable is an abstraction of a memory cell.

- Think of a post office model.

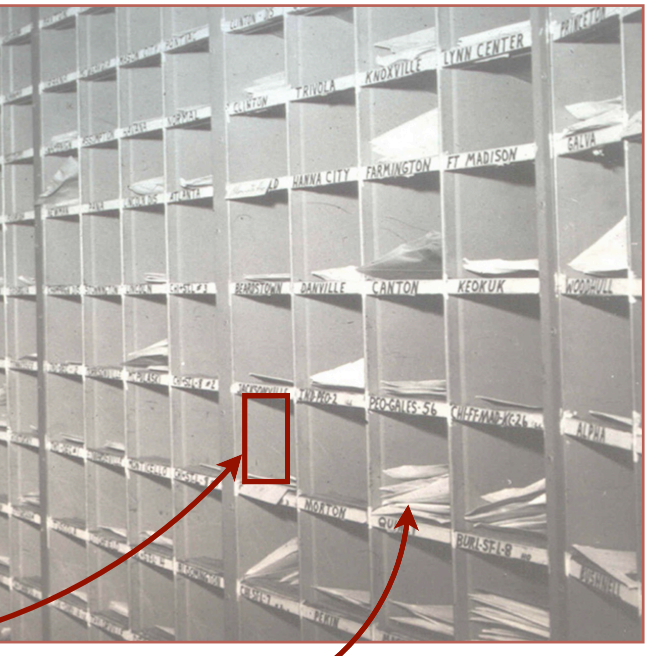



Variables are characterized by attributes

- name
- address
- value
- type
- scope
- lifetime
- visibility
- category

  … and more

# Identifiers, Variables, and Scope



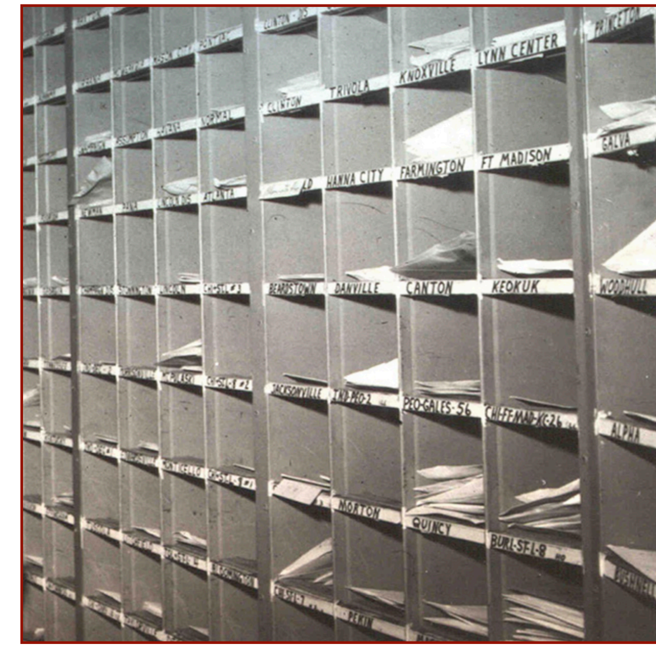
A variable is an abstraction of a memory cell.

- Think of a post office model.

Variables are characterized by attributes

- name — believe it or not, not all variables have names
- address — location in memory
- value — contents of the location in memory
- type — range of values and set of operations defined for them
- scope — range of statements in a program over which the var is “alive”
- lifetime — amount of time a variable is bound to a given memory location
- visibility — public, protected, private, internal, etc.
- category — const, iterator, etc.

# Identifiers, Variables, and Scope




A variable is an abstraction of a memory cell.

- Think of a post office model.

Variables are characterized by attributes

- name — believe it or not, not all variables have names
- address — location in memory
- value — contents of the location in memory
- type — range of values and set of operations defined for them
- scope — range of statements in a program over which the var is “alive”
- lifetime — amount of time a variable is bound to a given memory location
- visibility — public, protected, private, internal, etc.
- category — const, iterator, etc.

Binding?

# Identifiers, Variables, and Scope

Binding

- A binding is an association, such as between an attribute and an entity, or between an operation and a symbol
- Binding time is the time at which a binding takes place.
- What are the choices? When can binding take place?

# Identifiers, Variables, and Scope

Binding

- A binding is an association, such as between an attribute and an entity, or between an operation and a symbol
- Binding time is the time at which a binding takes place.
- What are the choices? When can binding take place?
  - **Language design time** - bind operator symbols to operations
  - **Language implementation time** - bind floating point type to a representation (BCD, two's compliment, whatever)
  - **OS Installation time** - .Net pre-compiles CLR and DLLs / JVM
  - **Compile time** - bind a variable to a type (C and Java, and our language)
  - **Load time** - bind a C or C++ static variable to a memory cell, for example)
  - **Runtime** - bind a non-static local variable to a memory cell

# Identifiers, Variables, and Scope

Binding

- A binding is an association, such as between an attribute and an entity, or between an operation and a symbol
- Binding time is the time at which a binding takes place.
- What are the choices? When can binding take place?
  - **Language design time** - bind operator symbols to operations
  - **Language implementation time** - bind floating point type to a representation (BCD, two's compliment, whatever)
  - **OS Installation time** - .Net pre-compiles CLR and DLLs / JVM
  - **Compile time** - bind a variable to a type (C and Java, and our language)
  - **Load time** - bind a C or C++ static variable to a memory cell, for example)
  - **Runtime** - bind a non-static local variable to a memory cell

Early

- A binding is **static** if it first occurs before run time and remains unchanged throughout program execution.
- A binding is **dynamic** if it first occurs during execution or can change during execution of the program.

# Identifiers, Variables, and Scope

Binding

- A binding is an association, such as between an attribute and an entity, or between an operation and a symbol
- Binding time is the time at which a binding takes place.
- What are the choices? When can binding take place?
  - **Language design time** - bind operator symbols to operations
  - **Language implementation time** - bind floating point type to a representation (BCD, two's compliment, whatever)
  - **OS Installation time** - .Net pre-compiles CLR and DLLs / JVM
  - **Compile time** - bind a variable to a type (C and Java, and our language)
  - **Load time** - bind a C or C++ static variable to a memory cell, for example)
  - **Runtime** - bind a non-static local variable to a memory cell

Late

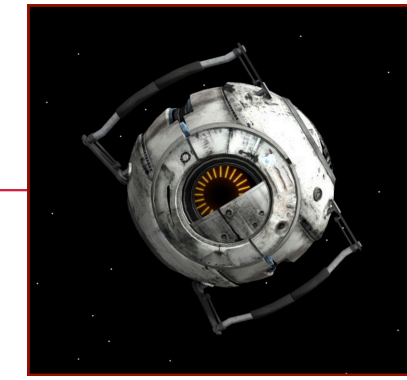
- A binding is **static** if it first occurs before run time and remains unchanged throughout program execution.
- A binding is **dynamic** if it first occurs during execution or can change during execution of the program.

# Static and Dynamic Scope

What's the output of this code?

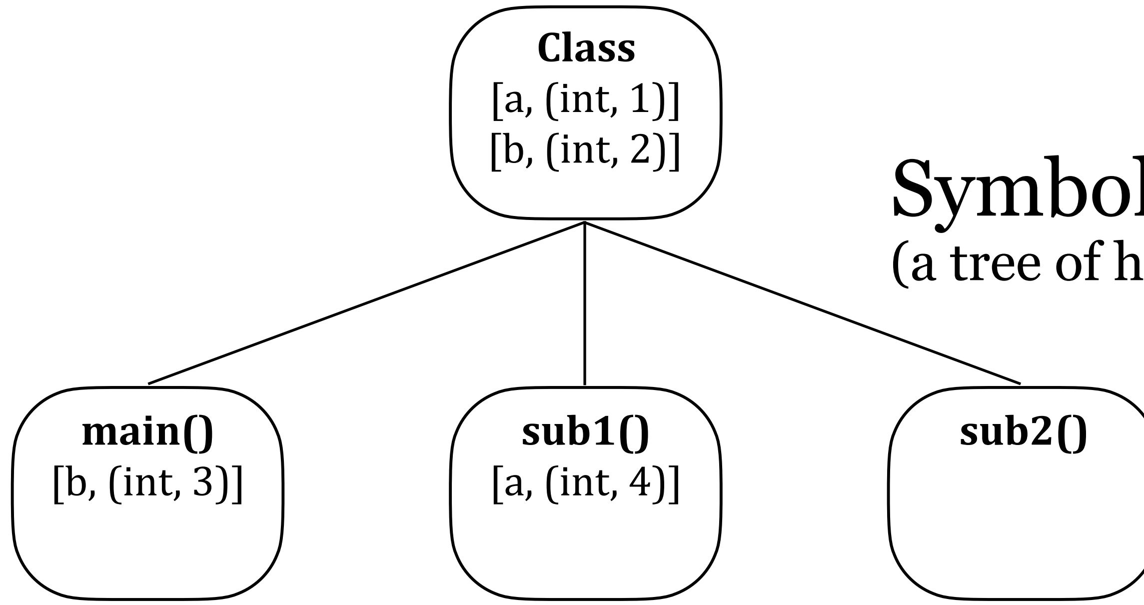
```
Class ScopeMan {
    int a := 1;
    int b := 2;

    main() {
        int b := 3;
        print(a,b);
        sub1();
    }

    sub1() {
        int a := 4;
        print(a,b);
        sub2();
    }

    sub2() {
        print(a,b);
    }
}
```

# Static and Dynamic Scope



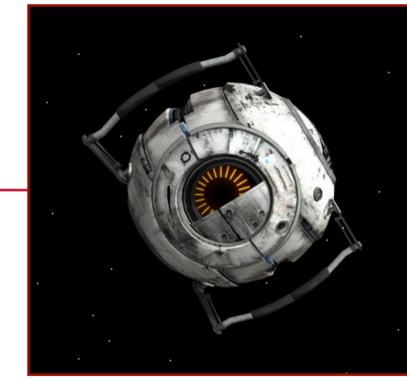
## Static Scope

```
Class ScopeMan {
   int a := 1;
   int b := 2;

   main() {
      int b := 3;
      print(a,b);
      sub1();
   }

   sub1() {
      int a := 4;
      print(a,b);
      sub2();
   }

   sub2() {
      print(a,b);
   }
}
```



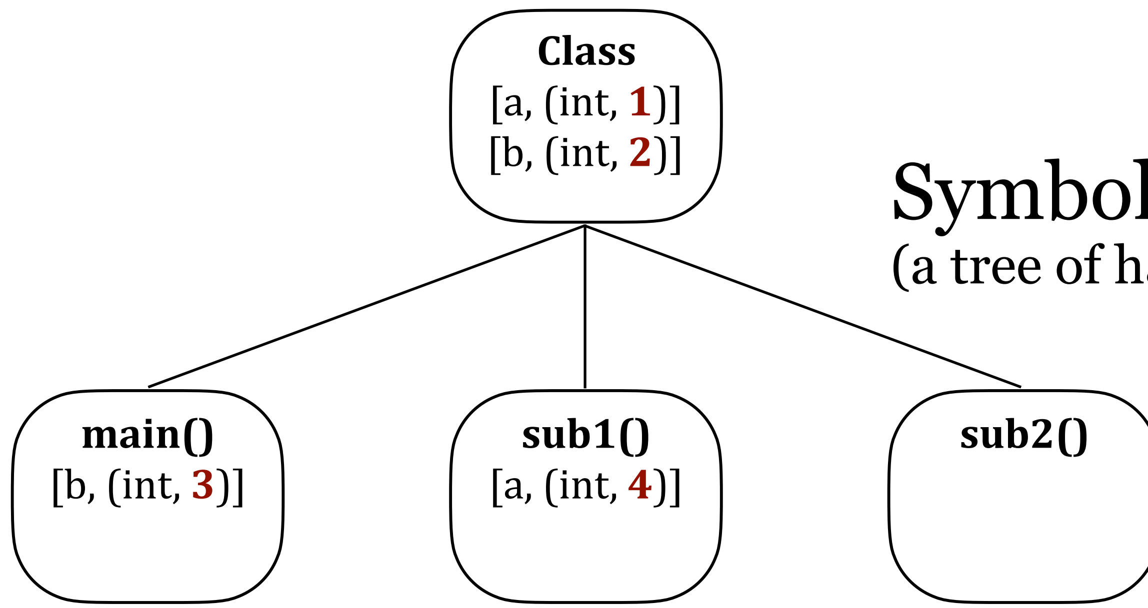

Symbol Table
(a tree of hash tables)

# Static and Dynamic Scope



## Static Scope

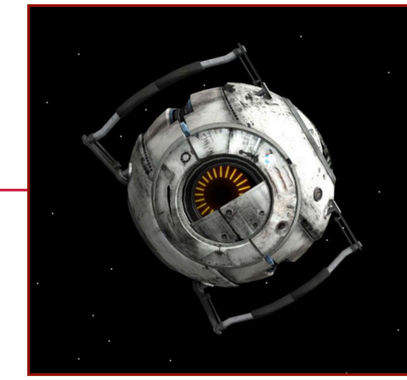
```
Class ScopeMan {
   int a := 1;
   int b := 2;

   main() {
      int b := 3;
      print(a,b);
      sub1();
   }

   sub1() {
      int a := 4;
      print(a,b);
      sub2();
   }

   sub2() {
      print(a,b);
   }
}
```




Symbol Table
(a tree of hash tables)

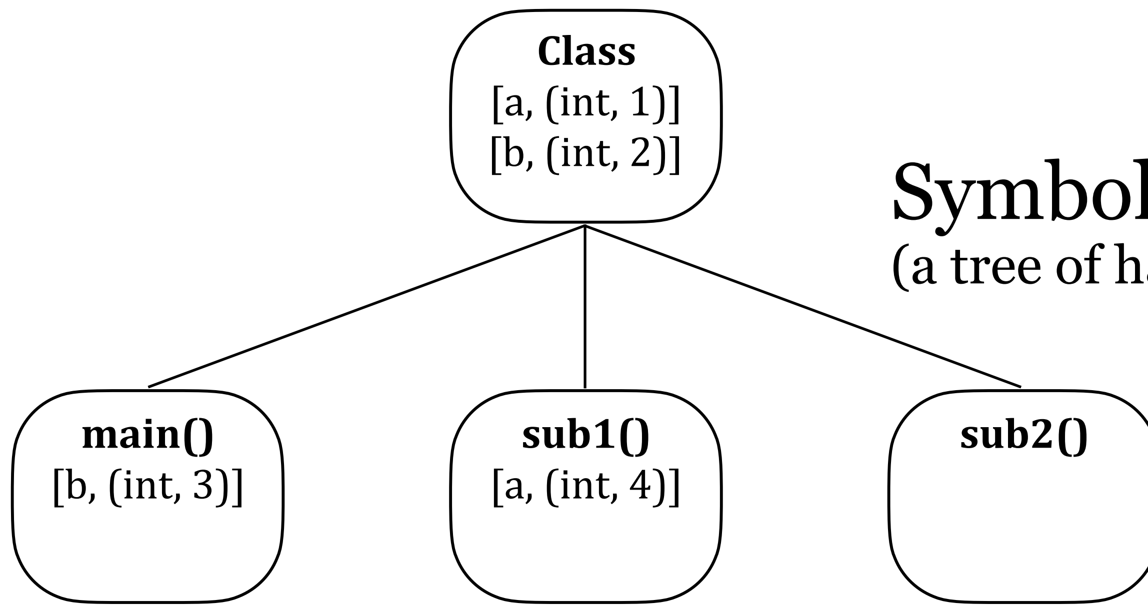
*Note*: We don't actually store the value of the ids in the symbol table, as they will be stored in memory. They're present in this depiction of a symbol table only so that we can easily see what the output should be. We'll have other attributes to store in the hash table along with the ids later on.

# Static and Dynamic Scope



## Static Scope

```
Class ScopeMan {
   int a := 1;
   int b := 2;

   main() {
      int b := 3;
      print(a,b);
      sub1();
   }

   sub1() {
      int a := 4;
      print(a,b);
      sub2();
   }

   sub2() {
      print(a,b);
   }
}
```



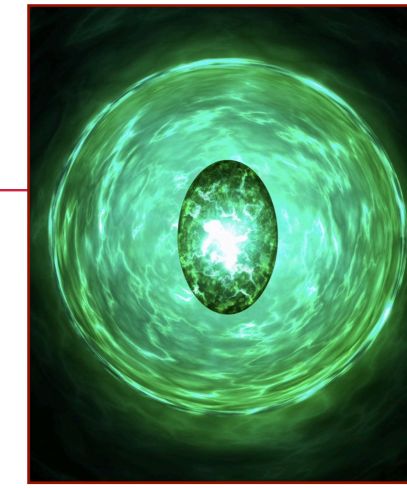

Symbol Table
(a tree of hash tables)

```
> run
1 3
4 2
1 2
```

Static scope is . . .

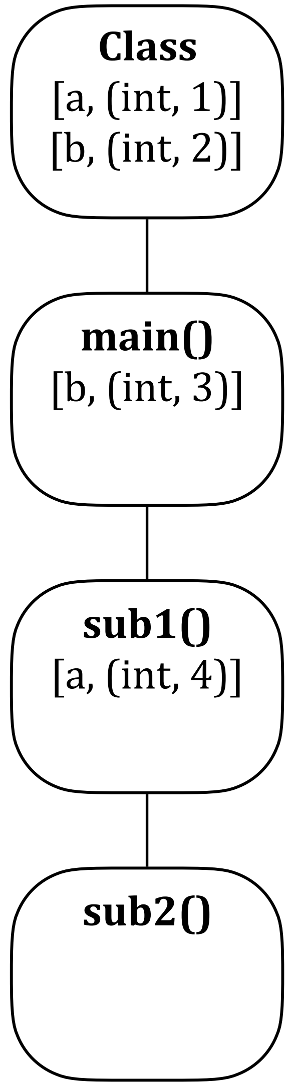
- Early binding
- Compile time
- about Space,
  - the shape of the code
  - the spacial relationships of code modules to each other at compile time.

# Static and Dynamic Scope



## Dynamic Scope

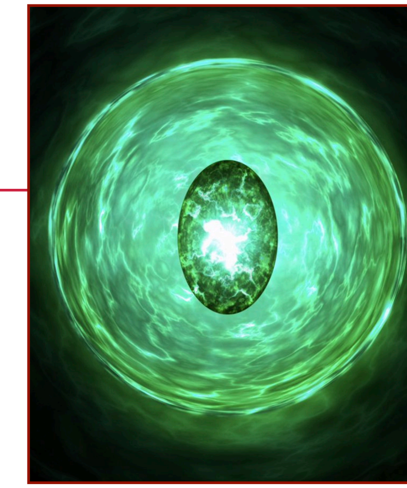
```
Class ScopeMan {
   int a := 1;
   int b := 2;

   main() {
      int b := 3;
      print(a,b);
      sub1();
   }

   sub1() {
      int a := 4;
      print(a,b);
      sub2();
   }

   sub2() {
      print(a,b);
   }
}
```




Symbol Table
(a tree of hash tables)

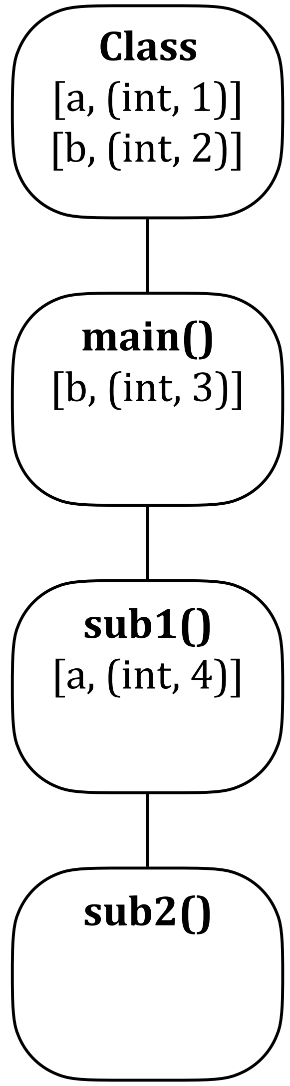
*Yes, this is a tree. It's also a list. And it looks like a stack. But it's a tree. And a graph. Let's just think of it as a tree.*

# Static and Dynamic Scope



## Dynamic Scope

```
Class ScopeMan {
   int a := 1;
   int b := 2;

   main() {
      int b := 3;
      print(a,b);
      sub1();
   }

   sub1() {
      int a := 4;
      print(a,b);
      sub2();
   }

   sub2() {
      print(a,b);
   }
}
```



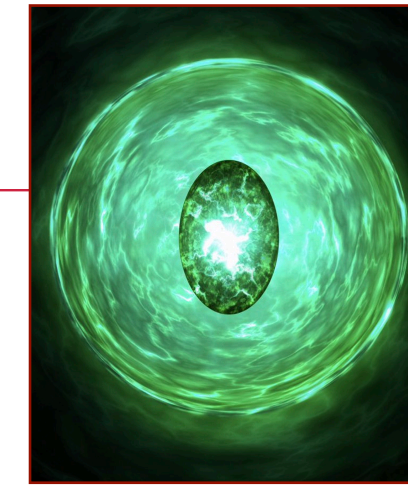

```
> run
1 3
4 3
4 3
```

Symbol Table
(a tree of hash tables)

Dynamic scope is . . .

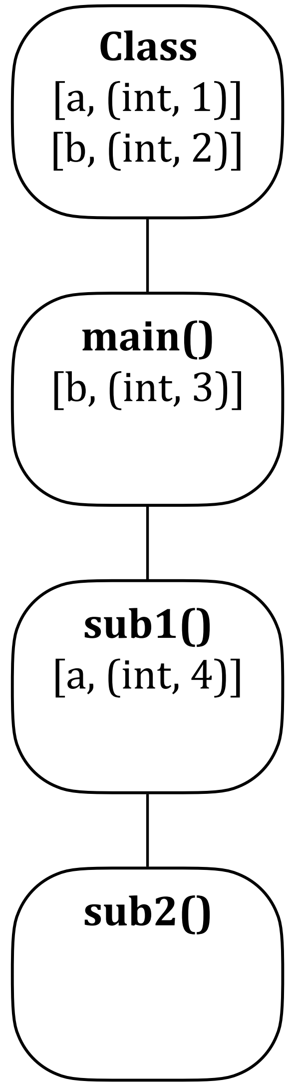
- Late binding
- Run time
- about Time,
  - the call stack
  - the execution order of code modules at run time.

# Static and Dynamic Scope



## Dynamic Scope

```
Class ScopeMan {
   int a := 1;
   int b := 2;

   main() {
      int b := 3;
      print(a,b);
      sub1();
   }

   sub1() {
      int a := 4;
      print(a,b);
      sub2();
   }

   sub2() {
      print(a,b);
   }
}
```




```
> run
1 3
4 3
4 3
```

Symbol Table
(a tree of hash tables)

*Yes, this is a tree. It's also a list. And it looks like a stack. But it's a tree.*

Dynamic scope is . . .

- Late binding
- Run time
- about Time,
  - the call stack
  - the execution order of code modules at run time.

The Basics

**What is a Type**?

- A set of values
- A set of operations on those values

**Type errors** happen when we try to perform operations on values that do not support them.

**Type expressions** are textual representations of type

- primitive/prime: *int*, *boolean*, *real*, *date*, *time*, *char*, *pointer*, …
- composite: *timestamp*, *latitude*, *longitude*, *student-id*, …

What about *string*? Or *String*?

# Type Systems

## The Basics

**What is a Type**?

- A set of values
- A set of operations on those values

**Type errors** happen when we try to perform operations on values that do not support them.

**Type expressions** are textual representations of type

- primitive/prime: *int*, *boolean*, *real*, *date*, *time*, *char*, *pointer*, …
- composite: *timestamp*, *latitude*, *longitude*, *student-id*, string, …

**Type systems** consist of rules governing what operations are permitted on what values.

- strong type systems prevent type errors at runtime.
- weak type systems ~~allow~~ encourage type errors at runtime.
- Type systems can be documented and reasoned about using inferences rules.

# Type Systems

## Specifying a Type System with Inference Rules

$$\frac{\textit{preconditions}}{\textit{postconditions}}$$

We can use inference rules from mathematics and Axiomatic Semantics. Why?

- It's fun.
- It's accurate. We need a rigorous definition of types and type systems so that we can enforce them in the compiler.
- It gives flexibility in implementation because it's not tied to any grammar.
- It allows for formal verification of program properties.
- It's what used in the computer science literature.

# Inference Rules

An inference rule is written

$$\frac{f_1,\ f_2,\ \dots f_n}{f_0}$$

It expresses that **if** $f_1$, $f_2$, ... $f_n$ are theorems — that is, they are proven well-formed formulae (WFF) — **then** we can infer that $f_0$ is another theorem.

That's nice, but how do we know?
How can we actually prove things?

Let's look at famous inference rule: Modus Ponens.

# A Famous Inference Rule

Modus Ponens

$$\frac{p,\ p \Rightarrow q}{q}$$

Modus Ponens ("the mode that affirms") can be read:

**if**
    we have $p$ (meaning, $p$ is true) **and**
    $p$ implies $q$
**then**
    we can infer that $q$ is true.
**end if**

The implication/conditional operator ($\Rightarrow$) is like a contract:
if $p$ then $q$.

# Inference Rule vs. Propositional Connective

Modus Ponens

$$\frac{p,\ p \Rightarrow q}{q}$$

Inference Rule

Modus Ponens ("the mode that affirms") can be read:

**if**

we have *p* (meaning, *p* is true) **and**

*p* implies *q*

**then**

we can infer that *q* is true.

**end if**

Propositional Connective

The implication/conditional operator ($\Rightarrow$) is like a contract:
if *p* then *q*.

Let's review this in Propositional logic.

# Propositional Logic

## Truth Tables

| p | q |
|---|---|
| 0 | 0 |
| 0 | 1 |
| 1 | 0 |
| 1 | 1 |

Propositional logic has only false and true, no variables.
It also has logical operators like and, or, and not.
(propositional connectors)

# Propositional Logic

## Truth Tables

| p | q | p∧q |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Propositional logic has only false and true, no variables.
It also has logical operators like **and**, or, and not.

# Propositional Logic

## Truth Tables

| p | q | p∧q | p∨q |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 |

Propositional logic has only false and true, no variables.
It also has logical operators like and, **or**, and not.

# Propositional Logic

## Truth Tables

| p | q | p∧q | p∨q | ¬p |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 |

Propositional logic has only false and true, no variables.
It also has logical operators like and, or, and **not**.

Do we need more? (Do we even need all of these?)

# Propositional Logic

## Truth Tables

| p | q | p∧q | p∨q | ¬p |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 |

Propositional logic has only false and true, no variables.
It also has logical operators like and, or, and not.

Do we need more? No. (Do we even need all of these? No.)

$$p \lor q = \lnot(\lnot p \land \lnot q)$$

# Propositional Logic

## Truth Tables

| $p$ | $q$ | $p \land q$ | $p \lor q$ | $\neg p$ | $p \Rightarrow q$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | |
| 0 | 1 | 0 | 1 | 1 | |
| 1 | 0 | 0 | 1 | 0 | |
| 1 | 1 | 1 | 1 | 0 | |

Propositional logic has only false and true, no variables.
It also has logical operators like and, or, and not.

Implication is like a contract:
"if $p$ then $q$" or "$p \Rightarrow q$".

# Propositional Logic

## Truth Tables

| p | q | p∧q | p∨q | ¬p | p⇒q |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | |
| 1 | 1 | 1 | 1 | 0 | |

These are vacuously true because *p* is false and false can imply anything because it's an invalid premise.

Also, we take "if *p* then *q*" to be false **only** when *p* is true and *q* is false.

Propositional logic has only false and true, no variables.
It also has logical operators like and, or, and not.

Implication is like a contract:
"if *p* then *q*" or "$p \Rightarrow q$".

# Propositional Logic

## Truth Tables

| p | q | p∧q | p∨q | ¬p | p⇒q |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | |

This is false because *p* is true and *q* is false, and "true implies false" is false.

Propositional logic has only false and true, no variables.
It also has logical operators like and, or, and not.

Implication is like a contract:
"if *p* then *q*" or "$p \Rightarrow q$".

# Propositional Logic

## Truth Tables

| p | q | p∧q | p∨q | ¬p | p⇒q |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 |

This is true because *p* is true and *q* is true, and "true implies true" is true.

Propositional logic has only false and true, no variables.
It also has logical operators like and, or, and not.

Implication is like a contract:
"if *p* then *q*" or "$p \Rightarrow q$".

# Propositional Logic

## Truth Tables

| p | q | p∧q | p∨q | ¬p | p⇒q | ¬p∨q |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | |
| 0 | 1 | 0 | 1 | 1 | 1 | |
| 1 | 0 | 0 | 1 | 0 | 0 | |
| 1 | 1 | 1 | 1 | 0 | 1 | |

Propositional logic has only false and true, no variables.
It also has logical operators like and, or, and not.

Implication is like a contract:
"if $p$ then $q$" or "$p \Rightarrow q$".
Implication can also be written as $\neg p \vee q$.

# Propositional Logic

## Truth Tables

These two columns are the same.
Both are implication.

| p | q | p∧q | p∨q | ¬p | p⇒q | ¬p∨q |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 |

Propositional logic has only false and true, no variables.
It also has logical operators like and, or, and not.

Implication is like a contract:
"if $p$ then $q$" or "$p \Rightarrow q$".
Implication can also be written as $\neg p \vee q$.

# Propositional Logic

## Truth Tables

| p | q | p∧q | p∨q | ¬p | p⇒q | ¬p∨q | p∧q⇒(p⇒q) |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 | |

Propositional logic has only false and true, no variables.
It also has logical operators like and, or, and not.

Implication is like a contract:
"if $p$ then $q$" or "$p \Rightarrow q$".
Implication can also be written as $\neg p \vee q$.

A tautology is logical statement that is always true regardless of the truth values of its components. Here's one: $p \wedge q \Rightarrow (p \Rightarrow q)$

# Propositional Logic

## Truth Tables

| p | q | p∧q | p∨q | ¬p | p⇒q | ¬p∨q | p∧q⇒(p⇒q) |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |

Tautology

Propositional logic has only false and true, no variables.
It also has logical operators like and, or, and not.

Implication is like a contract:
"if $p$ then $q$" or "$p \Rightarrow q$".
Implication can also be written as $\neg p \vee q$.

A tautology is logical statement that is always true regardless of the truth values of its components. Here's one: $p \wedge q \Rightarrow (p \Rightarrow q)$

# Propositional Logic

## Truth Tables

| $p$ | $q$ | $p \wedge q$ | $p \vee q$ | $\neg p$ | $p \Rightarrow q$ | $\neg p \vee q$ | $p \wedge q \Rightarrow (p \Rightarrow q)$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |

What's the opposite of a tautology, where the statement is always false?

Propositional logic has only false and true, no variables.
It also has logical operators like and, or, and not.

Implication is like a contract:
"if $p$ then $q$" or "$p \Rightarrow q$".
Implication can also be written as $\neg p \vee q$.

A tautology is logical statement that is always true regardless of the truth values of its components. Here's one: $p \wedge q \Rightarrow (p \Rightarrow q)$

# Propositional Logic

## Truth Tables

| p | q | p∧q | p∨q | ¬p | p⇒q | ¬p∨q | p∧q⇒(p⇒q) |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |

What's the opposite of a tautology, where the statement is always false?
A contradiction.

Propositional logic has only false and true, no variables.
It also has logical operators like and, or, and not.

Implication is like a contract:
"if $p$ then $q$" or "$p \Rightarrow q$".
Implication can also be written as $\neg p \vee q$.

A tautology is logical statement that is always true regardless of the truth values of its components. Here's one: $p \wedge q \Rightarrow (p \Rightarrow q)$

# Propositional Logic

## Truth Tables

A contradiction

| p | q | p∧q | p∨q | ¬p | p⇒q | ¬p∨q | p∧q⇒(p⇒q) | p∧¬p |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

Propositional logic has only false and true, no
It also has logical operators like and, or, and n

Contradictions cannot exist.

Implication is like a contract:
"if *p* then *q*" or "$p \Rightarrow q$".
Implication can also be written as $\neg p \vee q$.

A tautology is logical statement that is always true regardless of the truth values of its components. Here's one: $p \wedge q \Rightarrow (p \Rightarrow q)$

# Back to that Famous Inference Rule

## Propositional Logic for Modus Ponens

| p | q | p∧q | p∨q | ¬p | p⇒q | ¬p∨q | p∧q⇒(p⇒q) |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |

## Modus Ponens

$$\frac{p,\ p \Rightarrow q}{q}$$

can be written "if $p$ and $p \Rightarrow q$ then $q$", which can be written

$$(\,p \;\wedge\; (p \Rightarrow q)\,) \;\Rightarrow q$$

# A Famous Inference Rule

Propositional Logic for Modus Ponens

| p | q | p∧q | p∨q | ¬p | p⇒q | ¬p∨q | p∧q⇒(p⇒q) | p∧(p⇒q) |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |

Modus Ponens

$$\frac{p,\ p \Rightarrow q}{q}$$

can be written "if $p$ and $p \Rightarrow q$ then $q$", which can be written

$$(\ \underline{p \ \wedge\ (p \Rightarrow q)}\ ) \Rightarrow q$$

# A Famous Inference Rule

## Propositional Logic for Modus Ponens

| p | q | p∧q | p∨q | ¬p | p⇒q | ¬p∨q | p∧q⇒(p⇒q) | p∧(p⇒q) | (p∧(p⇒q))⇒q |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |

## Modus Ponens

$$\frac{p,\ p \Rightarrow q}{q}$$

can be written "if $p$ and $p \Rightarrow q$ then $q$", which can be written

$$(p \wedge (p \Rightarrow q)) \Rightarrow q$$

# A Famous Inference Rule

## Propositional Logic for Modus Ponens

| p | q | p∧q | p∨q | ¬p | p⇒q | ¬p∨q | p∧q⇒(p⇒q) | p∧(p⇒q) | (p∧(p⇒q))⇒q |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |

## Modus Ponens

$$\frac{p,\ p \Rightarrow q}{q}$$

can be written "if $p$ and $p \Rightarrow q$ then $q$", which can be written

$$(p \wedge (p \Rightarrow q)) \Rightarrow q$$

Tautology. Woot!

# Inference Rules for Type Systems

With Modus Ponens proved and used as the basis for inference rules, we need to move from Propositional logic to Predicate logic.

The complexity of reasoning about type systems cannot be handled with truth tables because we need to accommodate ideas like *any*, *all*, or *some*. Also, we need variables and functions. This leads us to . . .

First Order Logic

- variables
- domains
- named constants
- relations (>, <, etc.)
- functions (math operations)
- logical operators
- quantifiers (for-all “∀” and there-exists “∃”)

Now we can reason about type systems.

## Primitives / Literals / Intrinsic Types

### Boolean literals

$$\frac{}{\vdash \mathit{true}\text{: boolean}} \qquad \frac{}{\vdash \mathit{false}\text{: boolean}}$$

An empty pre-condition means "under any circumstances".

### String literals

$$\frac{s \text{ is a string literal or constant}}{\vdash s\text{: string}}$$

### Integer literals

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

# Type Systems

## Addition

Boolean literals

We cannot add Booleans because no inference rules are given to support that.

String literals

$$\frac{\vdash e_1 : \text{string} \qquad \vdash e_2 : \text{string}}{\vdash e_1 + e_2 : \text{string}}$$

This would be better labeled as *concatenation*, since it's not really addition. Maybe we should choose a different operator, like " • ".

Integer literals

$$\frac{\vdash e_1 : \text{integer} \qquad \vdash e_2 : \text{integer}}{\vdash e_1 + e_2 : \text{integer}}$$

Something is missing here . . .

# Type Systems

## Assignment

$$\frac{\vdash e_1 : T \quad \vdash e_2 : T \quad T \text{ is a primitive type}}{\vdash e_1 = e_2 : T}$$

## Comparisons

$$\frac{\vdash e_1 : T \quad \vdash e_2 : T \quad T \text{ is a primitive type}}{\vdash e_1 == e_2 : \text{boolean}}$$

$$\frac{\vdash e_1 : T \quad \vdash e_2 : T \quad T \text{ is a primitive type}}{\vdash e_1 > e_2 : \text{boolean}}$$

$$\frac{\vdash e_1 : T \quad \vdash e_2 : T \quad T \text{ is a primitive type}}{\vdash e_1 \mathrel{!=} e_2 : \text{boolean}}$$

$$\frac{\vdash e_1 : T \quad \vdash e_2 : T \quad T \text{ is a primitive type}}{\vdash e_1 < e_2 : \text{boolean}}$$

I've got a bad feeling about this . . .

# Type Systems

## Example

```
string x = “I have a”;
string y = “bad feeling”;

int aboutThis(int x) {
   return x + y;
}

main() {
   int z;
   z = aboutThis(42);
   print(z);
}
```

# Type Systems

## Example

```
string x = “I have a”;
string y = “bad feeling”;

int aboutThis(int x) {
   return x + y;
}

main() {
   int z;
   z = aboutThis(42);
   print(z);
}
```

**Things we know**

x: string
y: string
x: int
z: int

# Type Systems

## Example

```
string x = "I have a";
string y = "bad feeling";

int aboutThis(int x) {
   return x + y;
}

main() {
   int z;
   z = aboutThis(42);
   print(z);
}
```

**Things we know**
x: string
y: string
x: int
z: int

**Things we wonder about**
Is x + y a legal operation under our type rules?

$$\frac{\vdash e_1 : \text{string} \quad \vdash e_2 : \text{string}}{\vdash e_1 + e_2 : \text{string}}$$

$$\frac{\vdash e_1 : \text{integer} \quad \vdash e_2 : \text{integer}}{\vdash e_1 + e_2 : \text{integer}}$$

# Type Systems

## Example

```
string x = "I have a";
string y = "bad feeling";

int aboutThis(int x) {
   return x + y;
}

main() {
   int z;
   z = aboutThis(42);
   print(z);
}
```

**Things we know**
x: string
y: string
x: int
z: int

**Things we wonder about**
Is x + y a legal operation under our type rules?
Which x is that?

$$\frac{\vdash e_1 : \text{string} \quad \vdash e_2 : \text{string}}{\vdash e_1 + e_2 : \text{string}}$$

$$\frac{\vdash e_1 : \text{integer} \quad \vdash e_2 : \text{integer}}{\vdash e_1 + e_2 : \text{integer}}$$

# Type Systems

## Example - No Context

```
string x = "I have a";
string y = "bad feeling";

int aboutThis(int x) {
   return x + y;
}

main() {
   int z;
   z = aboutThis(42);
   print(z);
}
```

**Things we know**
x: string
y: string
x: int
z: int

**Things we wonder about**
Is x + y a legal operation under our type rules?

$$\frac{\vdash e_1 : \text{string} \qquad \vdash e_2 : \text{string}}{\vdash e_1 + e_2 : \text{string}} \qquad \frac{\vdash e_1 : \text{integer} \qquad \vdash e_2 : \text{integer}}{\vdash e_1 + e_2 : \text{integer}}$$

The problem is that our type rules lack context. We need to strengthen them to specify under what circumstances they apply. In other words, we need **scope**.

# Type Systems

## Addition with Scope Context

**Boolean literals**

We cannot add Booleans because no inference rules are given to support that.

**String literals**

$$\frac{S \vdash e_1 : \text{string} \qquad S \vdash e_2 : \text{string}}{S \vdash e_1 \cdot e_2 : \text{string}}$$

$e_1$ is a string in scope S
$e_2$ is a string in scope S
$e_1 \cdot e_2$ results in a string in scope S

**Integer literals**

$$\frac{S \vdash e_1 : \text{integer} \qquad S \vdash e_2 : \text{integer}}{S \vdash e_1 + e_2 : \text{integer}}$$

$e_1$ is an integer in scope S
$e_2$ is an integer in scope S
$e_1 + e_2$ results in an integer in scope S

This is better . . .

# Type Systems

**Assignment with Scope Context**

$$\frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 = e_2 : T}$$

**Comparisons with Scope Context**

$$\frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 == e_2 : \text{boolean}}$$

$$\frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 > e_2 : \text{boolean}}$$

$$\frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 \mathrel{!=} e_2 : \text{boolean}}$$

$$\frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 < e_2 : \text{boolean}}$$

I’ve got a good feeling about this.

# Type Systems

Addition
with Scope Context ...

$$\frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 + e_2 : T}$$

## ... and an implementation in Prolog

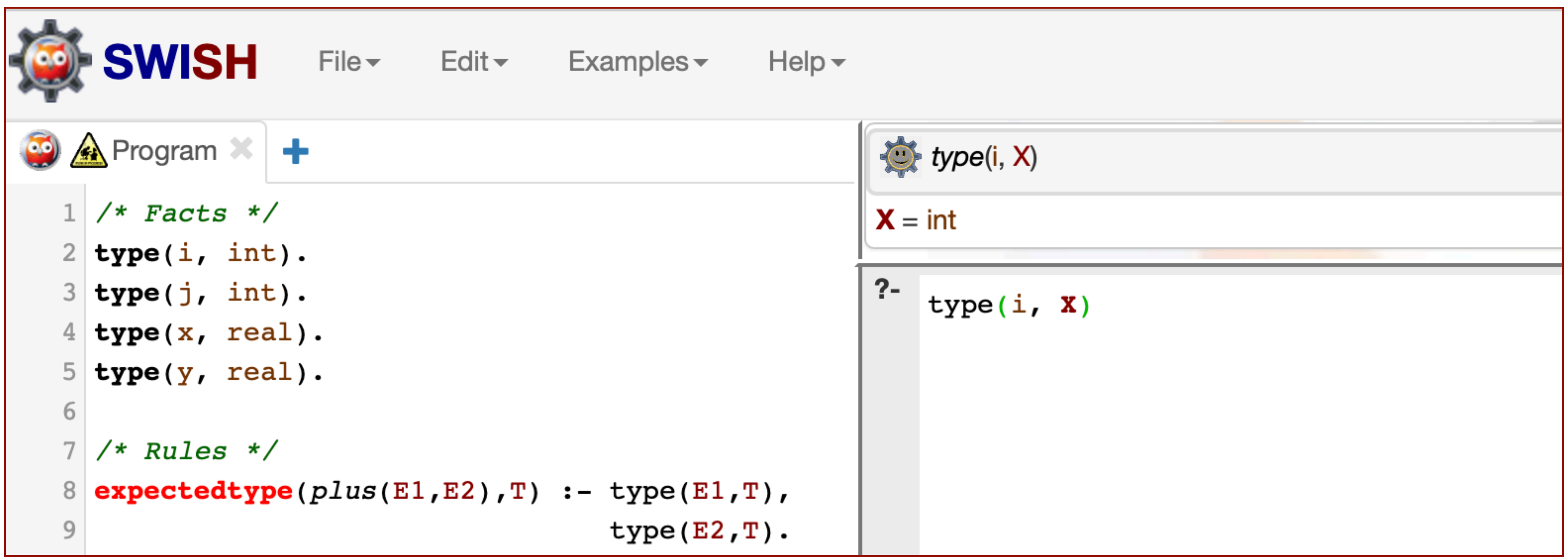
```
/* Symbol Table facts */
type(i, int).
type(j, int).
type(x, real).
type(y, real).

/* Type System rules */
expectedtype(plus(E1,E2),T) :- type(E1,T),
                               type(E2,T).

/* Type inference and checking queries */

expectedtype(plus(i,j),X)    /* int */

expectedtype(plus(x,y),X)    /* real */

expectedtype(plus(i,y),X)    /* false - Type error. (No unifying match.) */
```

# Type Systems

Addition
with Scope Context …

$$\frac{S \vdash e_1 : T \qquad S \vdash e_2 : T \qquad T \text{ is a primitive type}}{S \vdash e_1 + e_2 : T}$$

… and an implementation in Prolog

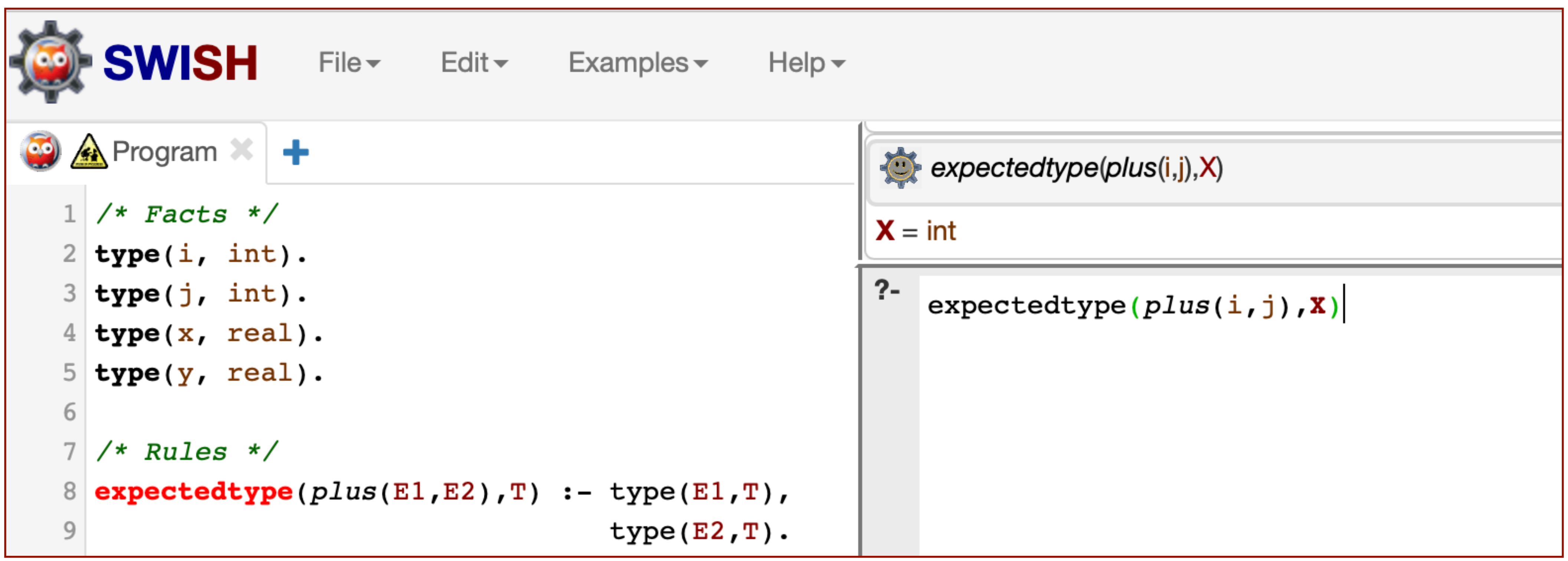


# Type Systems

Addition
with Scope Context …

$$\frac{S \vdash e_1 : T \qquad S \vdash e_2 : T \qquad T \text{ is a primitive type}}{S \vdash e_1 + e_2 : T}$$

… and an implementation in Prolog

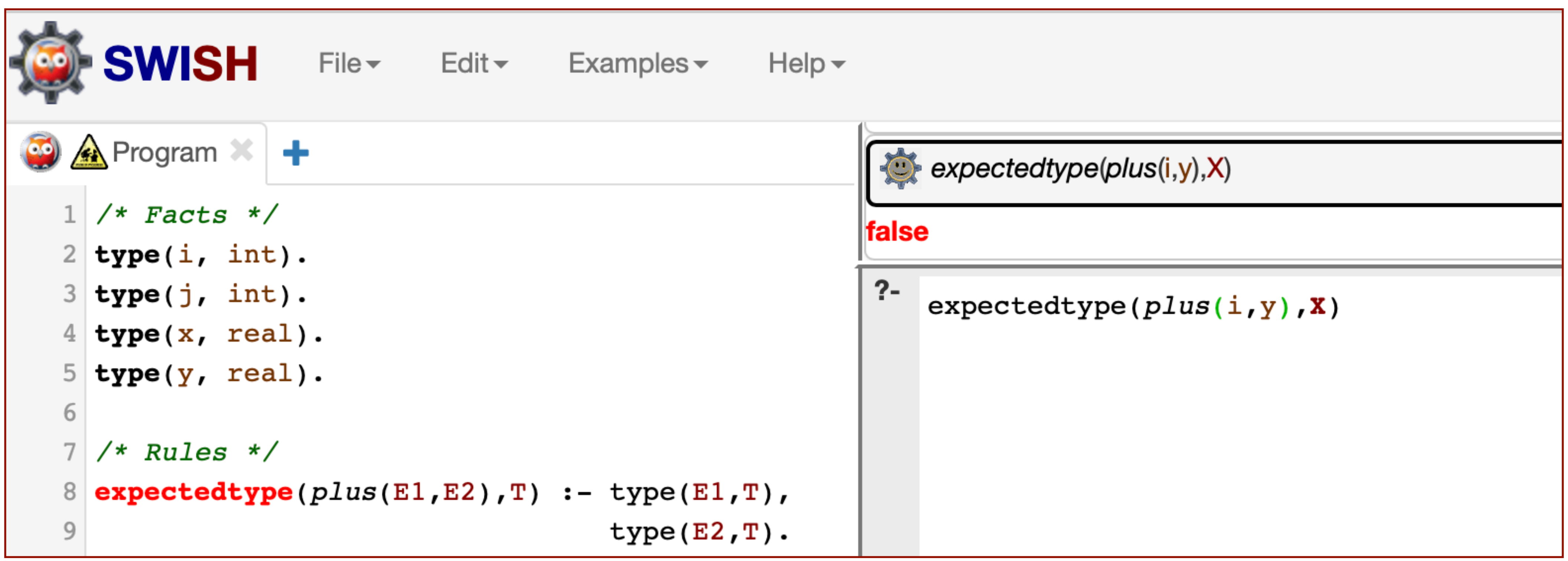


# Type Systems

Addition
with Scope Context …

$$\frac{S \vdash e_1 : T \qquad S \vdash e_2 : T \qquad T \text{ is a primitive type}}{S \vdash e_1 + e_2 : T}$$

… and an implementation in Prolog



# Type Systems

## Type Equivalence and Compatibility

What does it mean to say that two variable/values are equivalent?

$$1 \stackrel{?}{=} 1.0$$

$$1.0 \stackrel{?}{=} 1.000$$

$$\text{"c"} \stackrel{?}{=} \text{'c'}$$

There are two approaches:

**Name Equivalence**
Types are equivalent if they have the same name.
I.e., they are the same if the programmer says they are the same.
Restrictive, but easier to implement than structural equivalence.

**Structural Equivalence**
Types are equivalent if they have the same structure.
I.e., they are the same if they are built the same: **same parts** in the **same order**.
Flexible, but harder to implement than name equivalence.

## Type Equivalence and Compatibility

### Name Equivalence
Types are equivalent if they have the same name.

*first* and *last* are the same type.
*head* and *tail* are the same type.
*first* and *head* are different types.

```
type link = ↑cell;

var  first : link;
     last  : link;
     head  : ↑cell;
     tail  : ↑cell;
```

### Structural Equivalence
Types are equivalent if they have the same structure.

*first*, *last*, *head*, and *tail* are all the same type.

## Type Equivalence and Compatibility

### Name Equivalence

Types are equivalent if they have the same name.

*MyRec* and *YourRec* are different types.
*a1*, *a2*, and *a3* are all different types.

```
val MyRec   = { a=1, b=2 };
val YourRec = { a=1, b=2 };

var a1 = array[1..10] of int;
var a2 = array[1..2*5] of int;
var a3 = array[0..9] of int;
```

### Structural Equivalence

Types are equivalent if they have the same structure.

*MyRec* and *YourRec* are the same type.
*a1*, *a2*, and *a3* are all the same type.

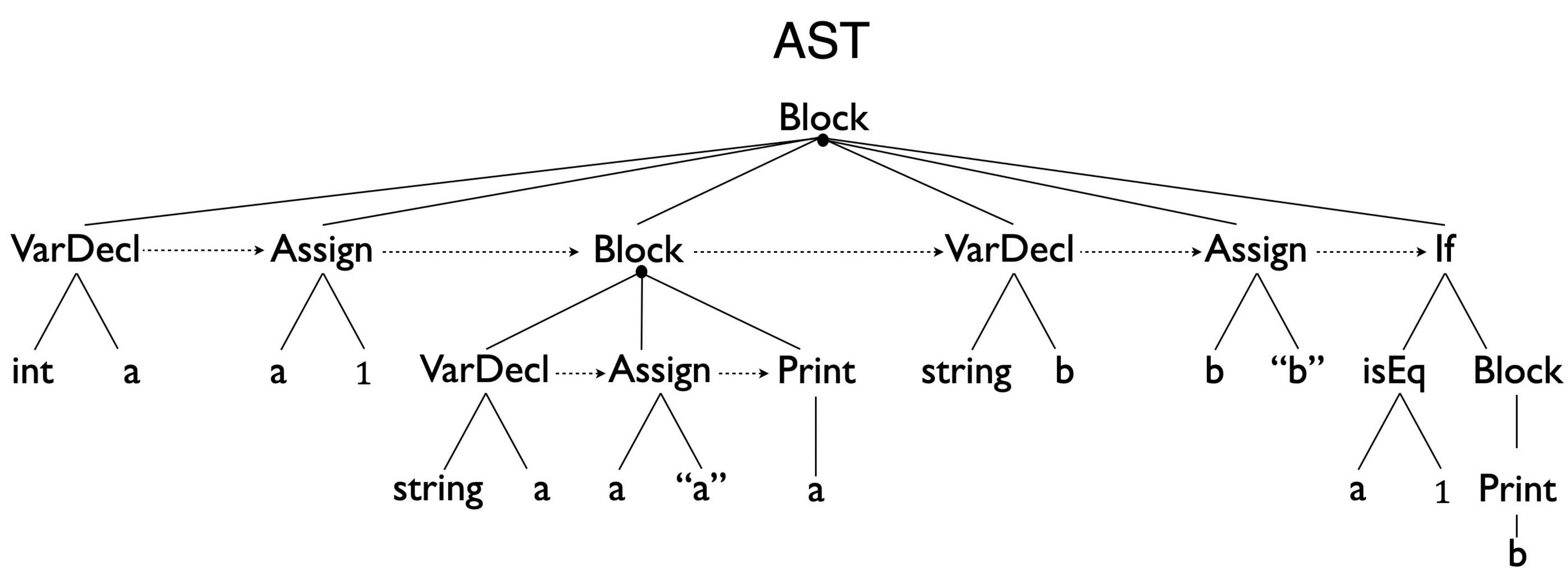
# Semantic Analysis

Semantic Analysis is the compiler phase that checks scope and type.

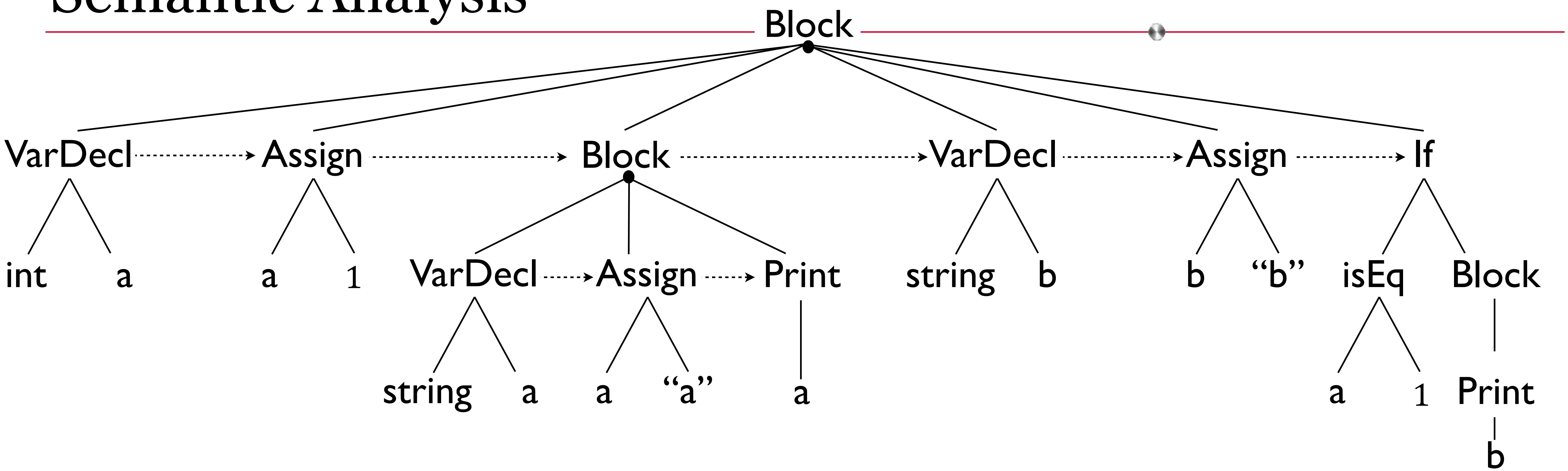
A depth-first, in-order Abstract Syntax Tree (AST) traversal will allow us to …

- build the symbol table (a tree of hash tables)
- check scope
- check type

… in a single pass. It's very cool. Let's do it!



# Semantic Analysis



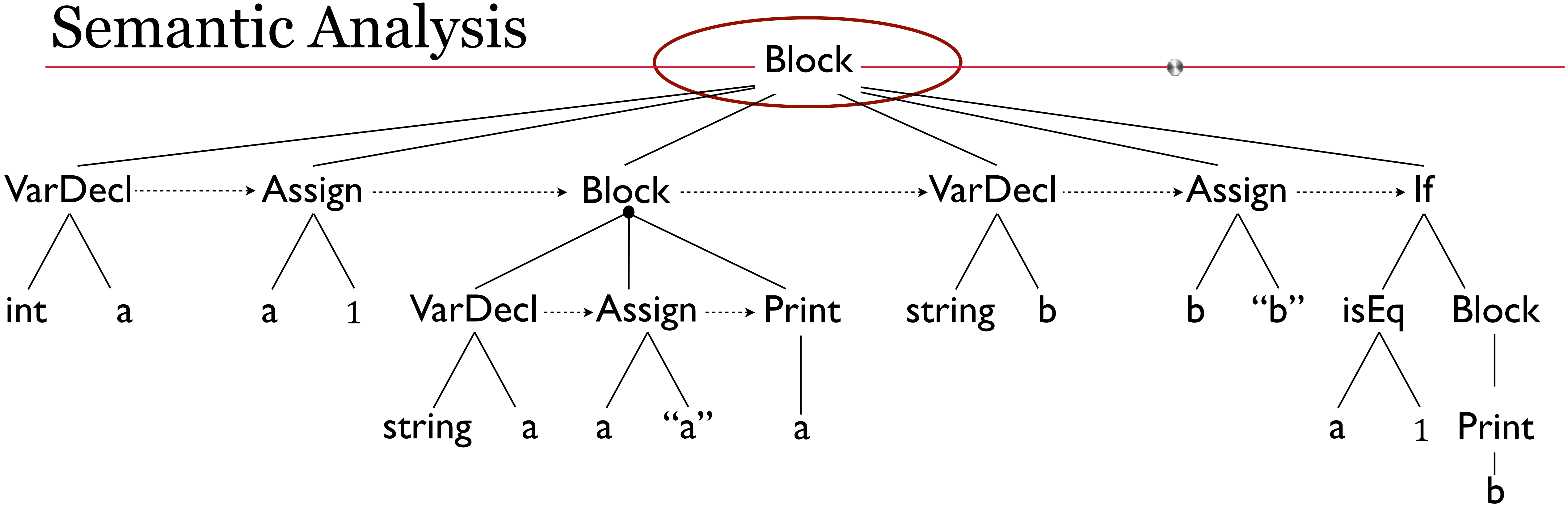

Source Code

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

Symbol Table

# Semantic Analysis

AST

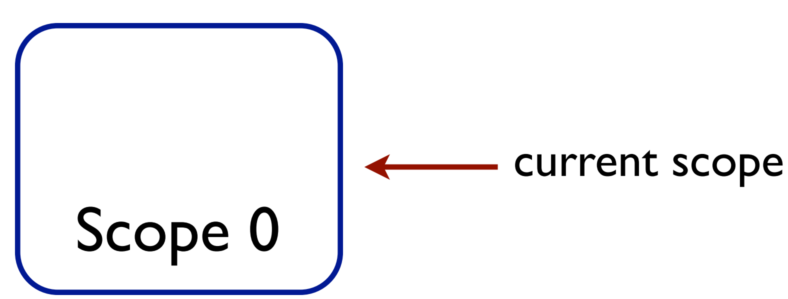



Initialize Scope 0

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

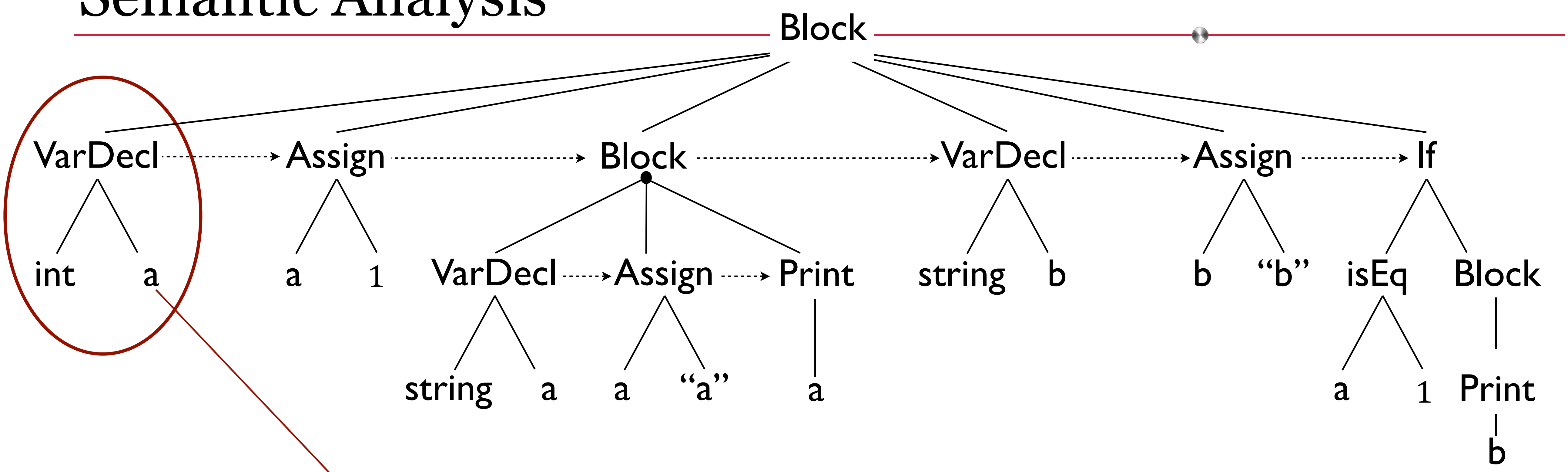
Source Code

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




Symbol Table

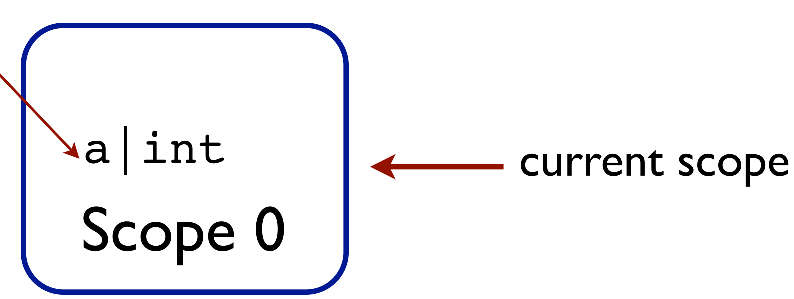
# Semantic Analysis




Initialize Scope 0
add symbol a
*in the* ***current scope***

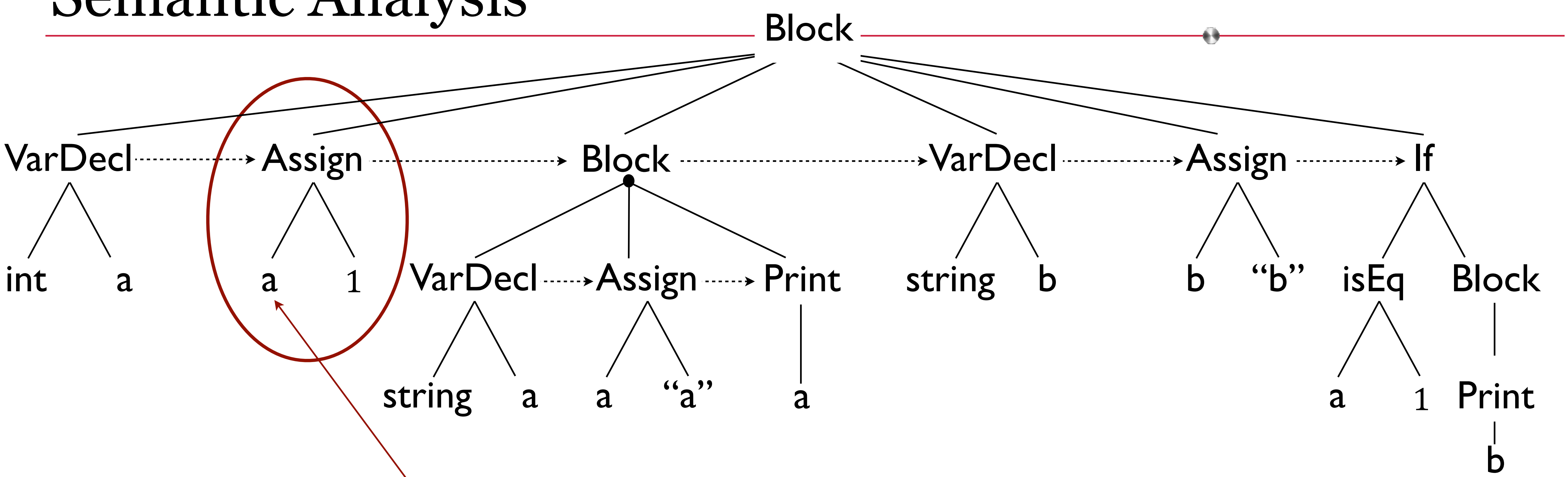
Source Code

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




Symbol Table

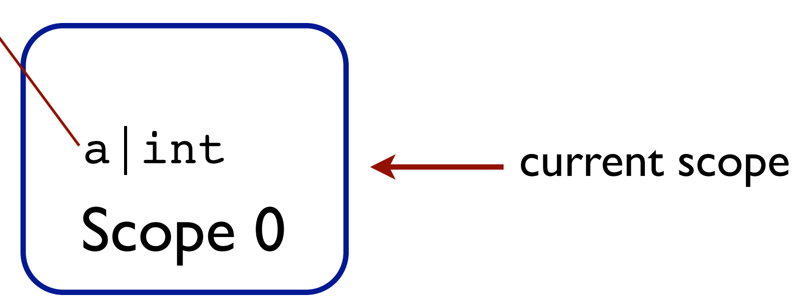
# Semantic Analysis




Initialize Scope 0
add symbol a
lookup symbol a
*in the* ***current scope***

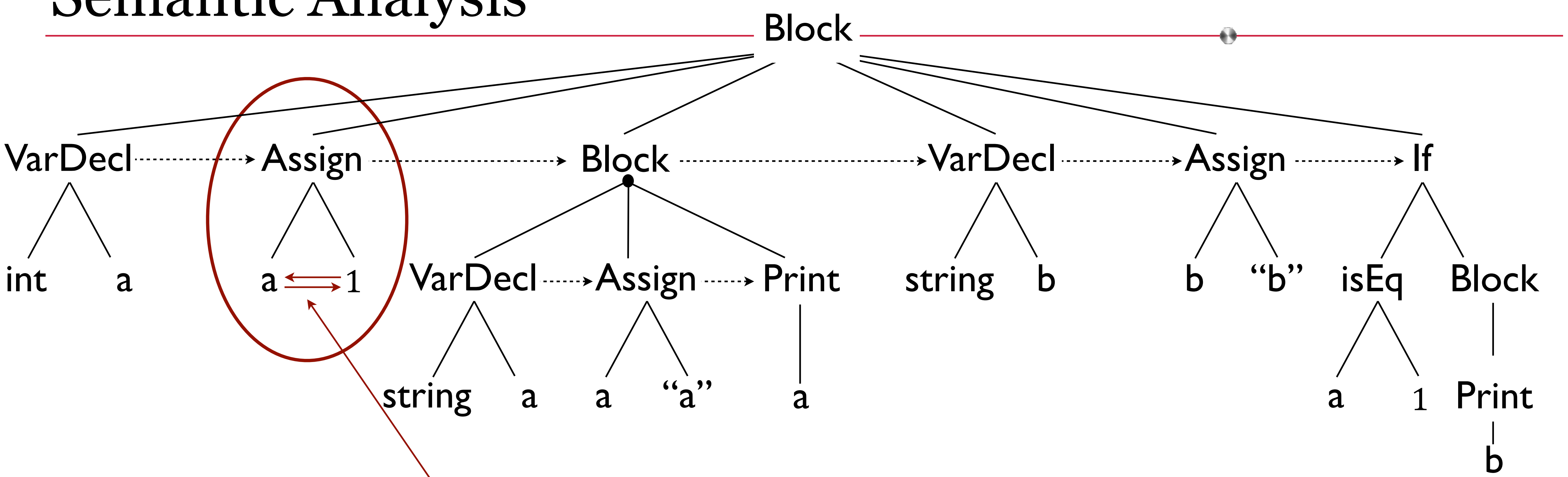
Source Code

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




Symbol Table

# Semantic Analysis

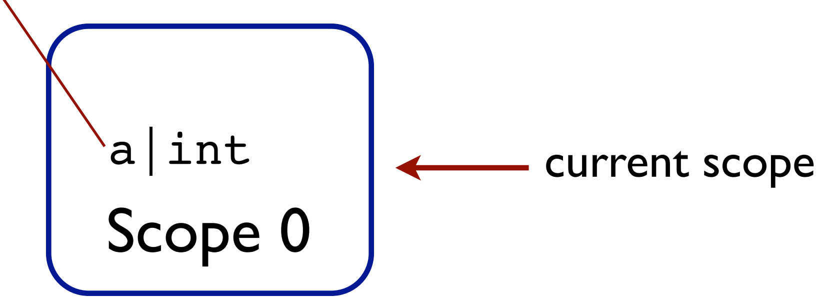



Initialize Scope 0
add symbol a
lookup symbol a
check types

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

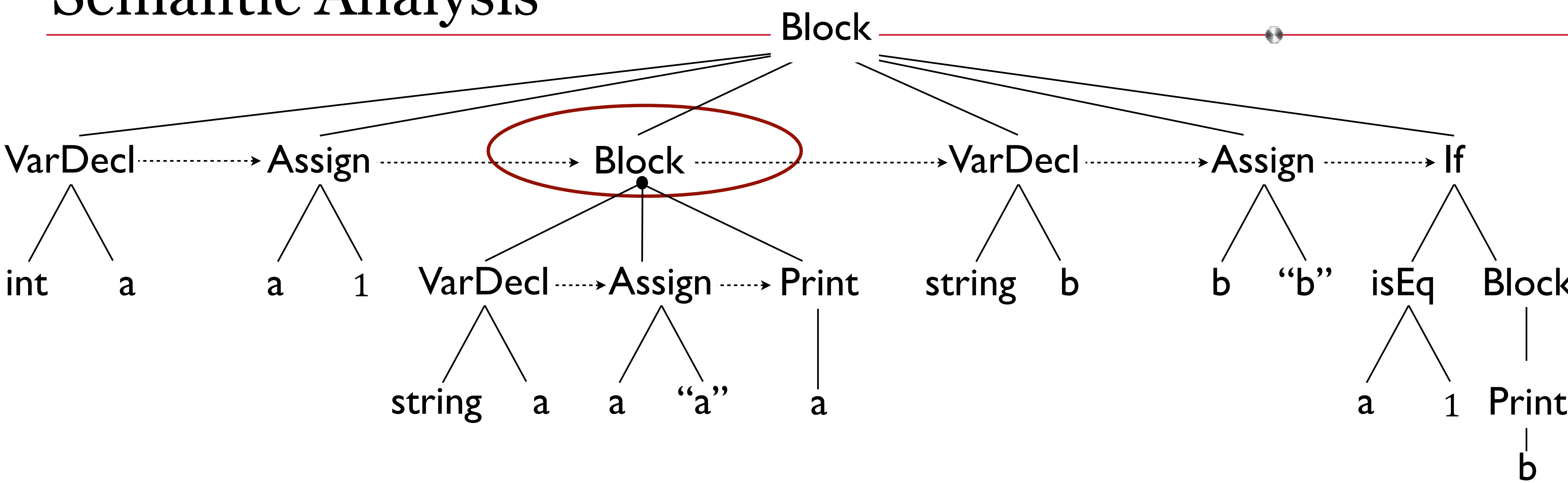
## Source Code

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




Symbol Table

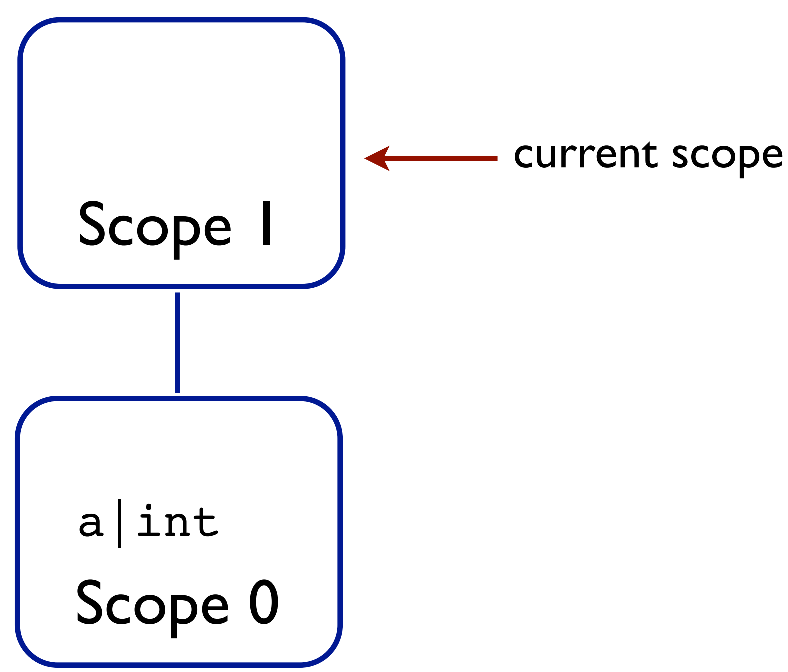
# Semantic Analysis




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

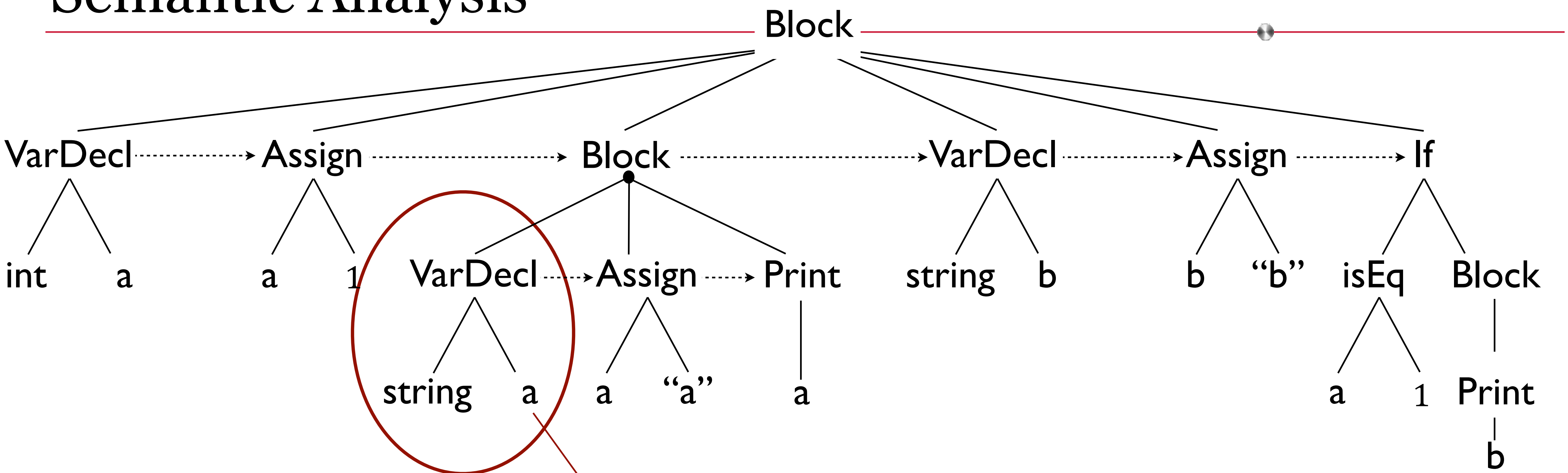
## Source Code

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




Symbol Table

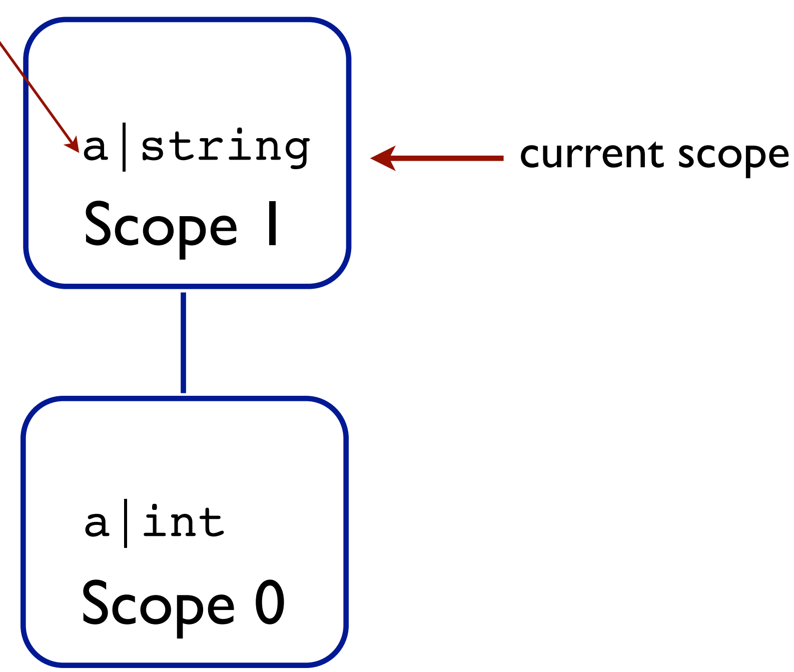
# Semantic Analysis




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

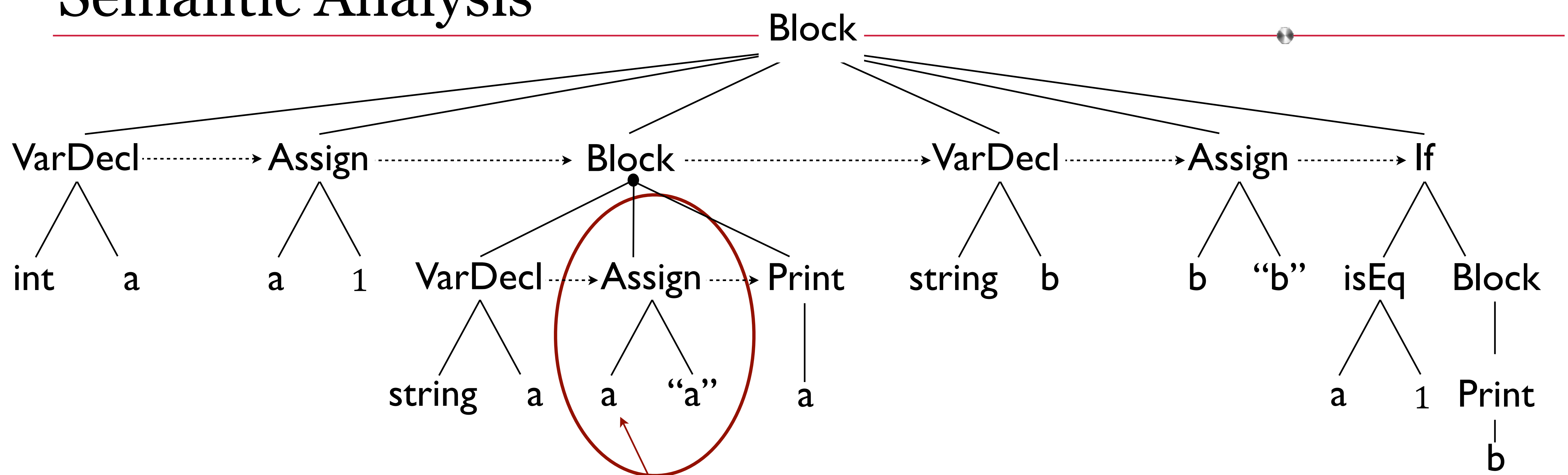
## Source Code

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



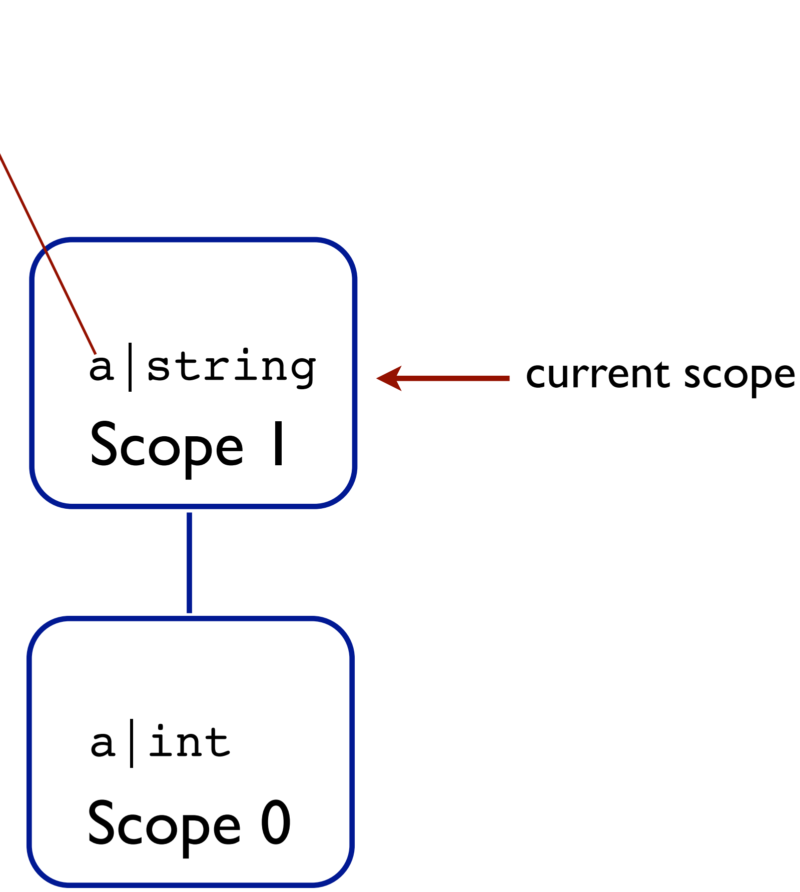

Symbol Table

# Semantic Analysis




Source Code

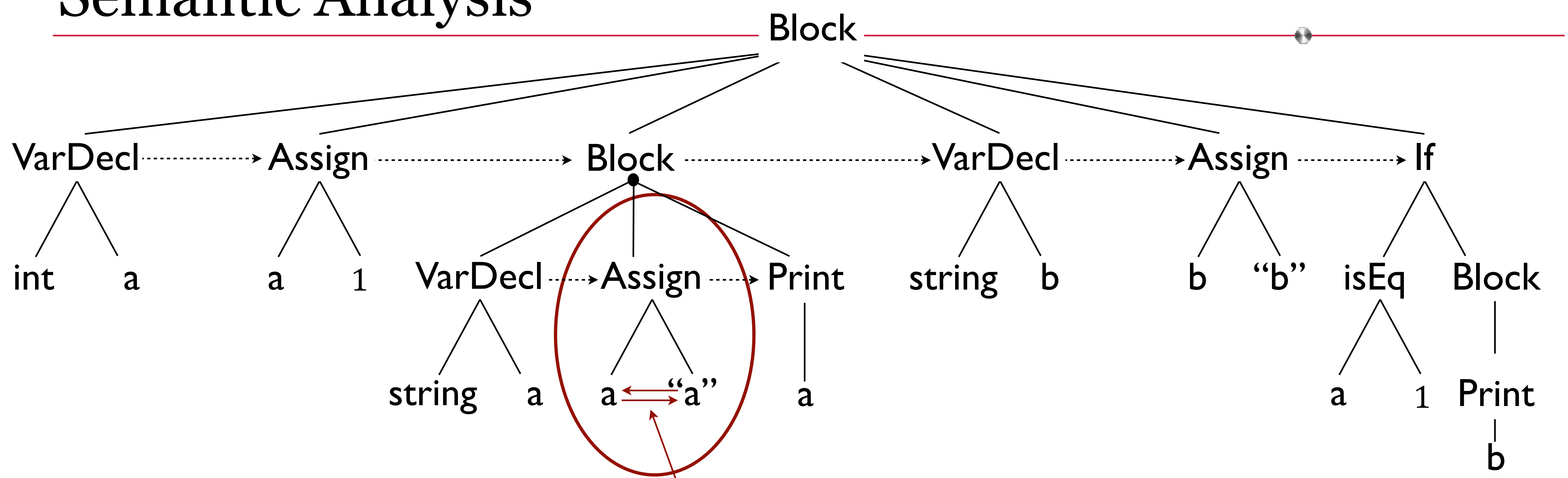
```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```




Symbol Table

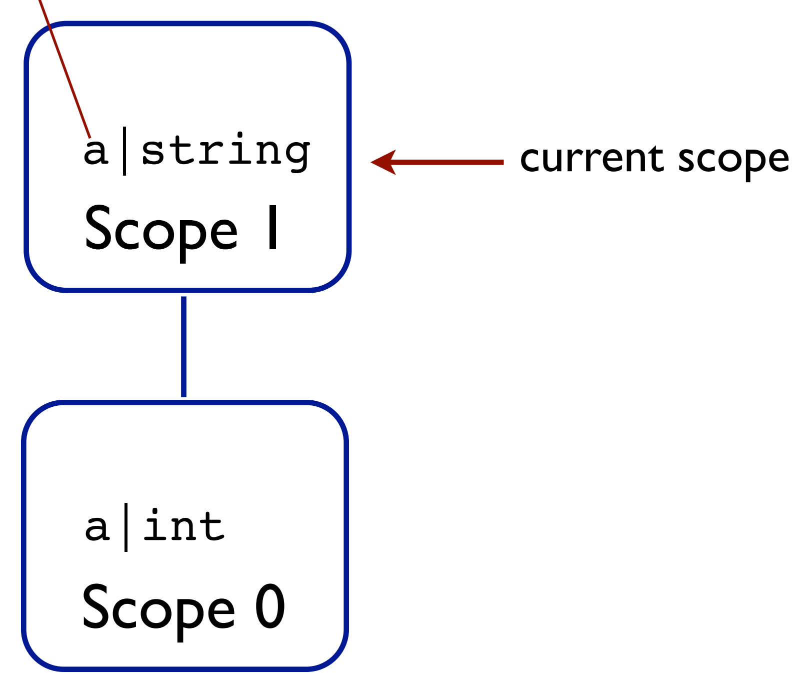
Initialize Scope 0
add symbol a
lookup symbol a
check types
Initialize Scope 1
add symbol a
lookup symbol a
*in the **current scope***

# Semantic Analysis




Source Code

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

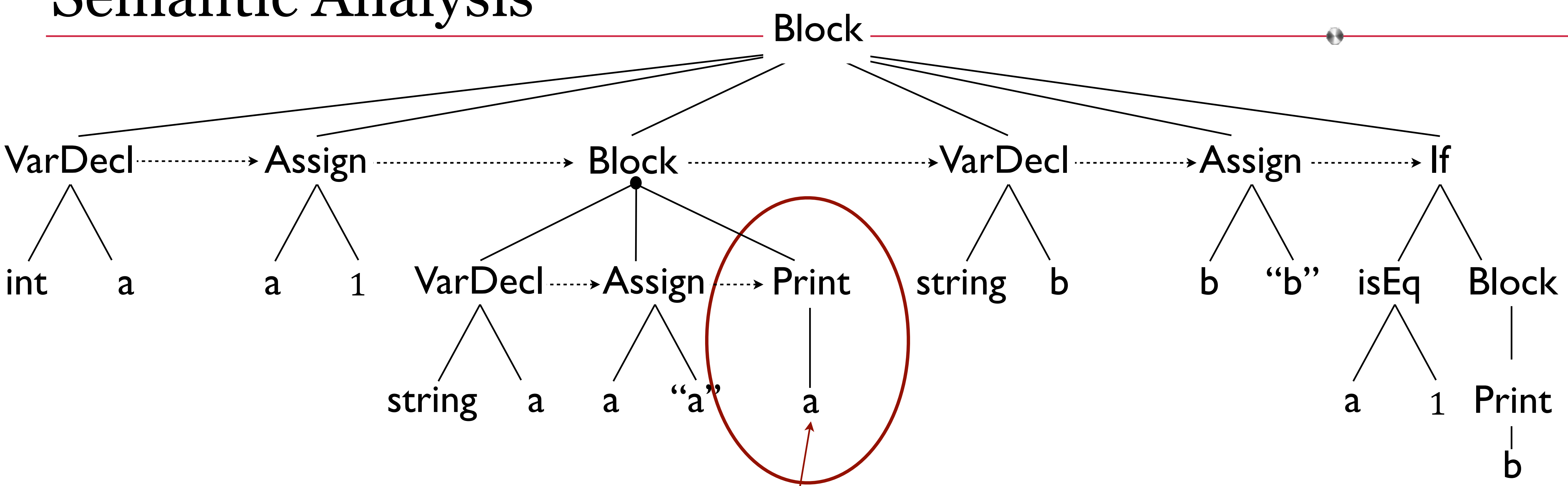



Symbol Table

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

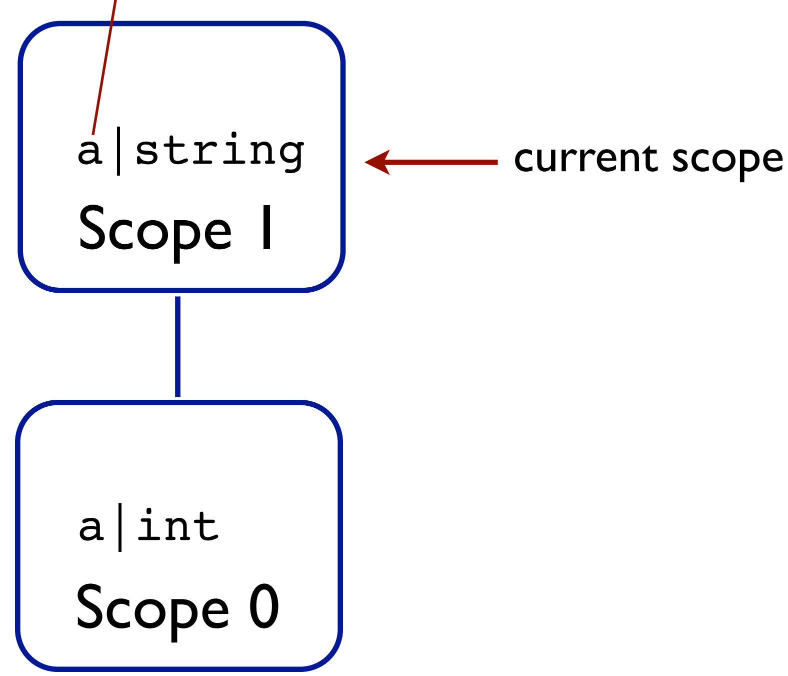
*Verify that the left child and right child are type compatible for assignment.*

# Semantic Analysis




Source Code

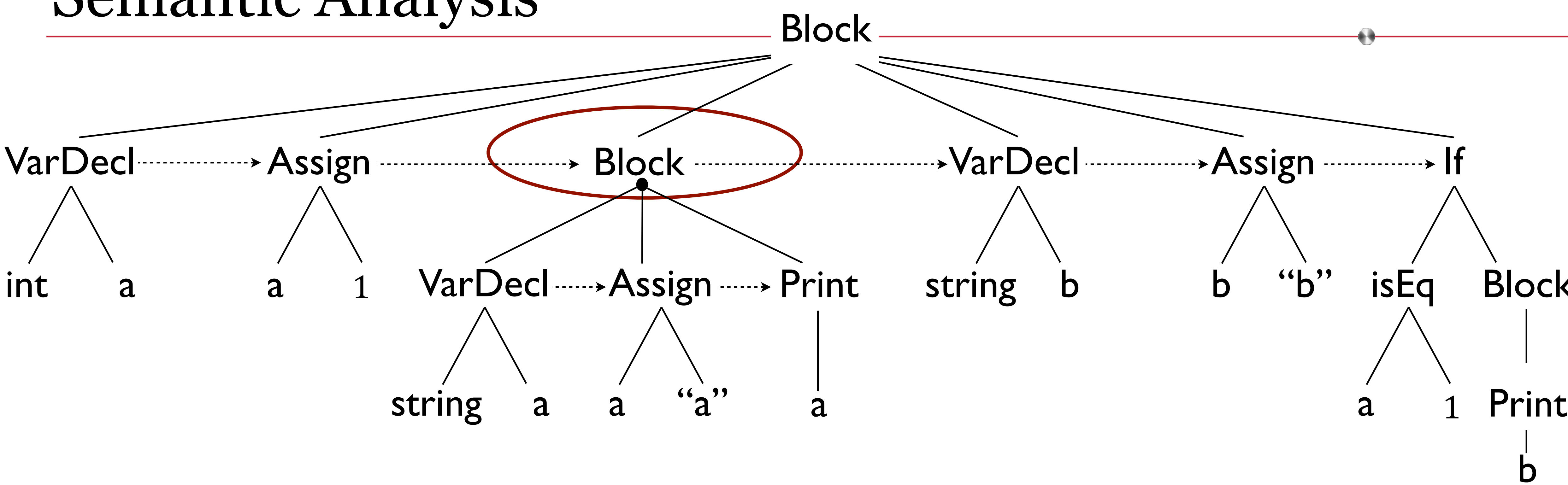
```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```




Symbol Table

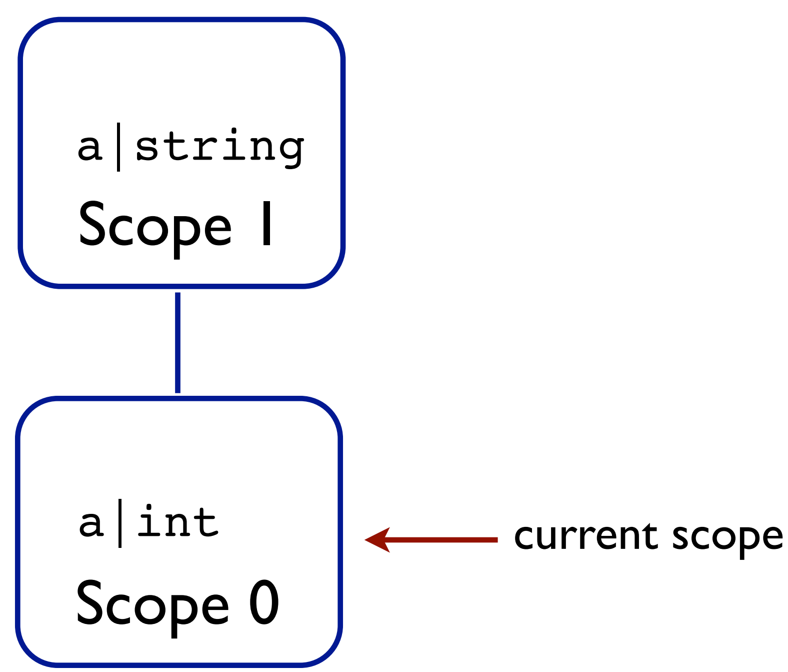
Initialize Scope 0
add symbol a
lookup symbol a
check types
Initialize Scope 1
add symbol a
lookup symbol a
check types
lookup symbol a
*in the **current scope**. Print can take any type, so there's no need to type check here. We must still check the scope, of course!*

# Semantic Analysis




## Source Code

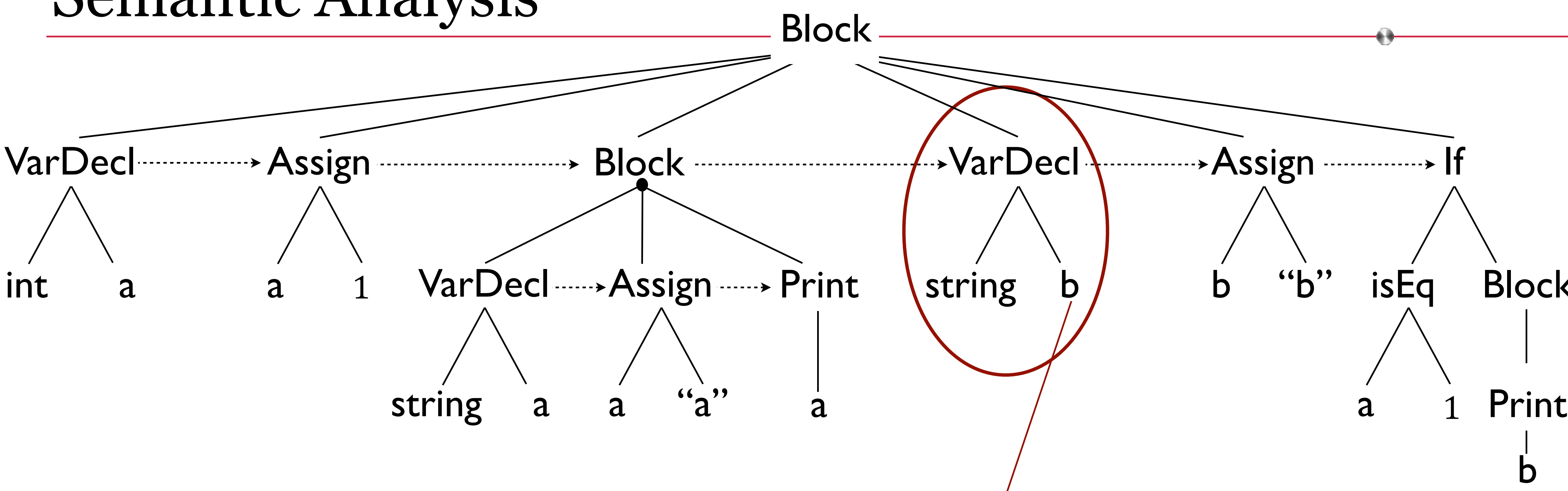
```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```




Symbol Table

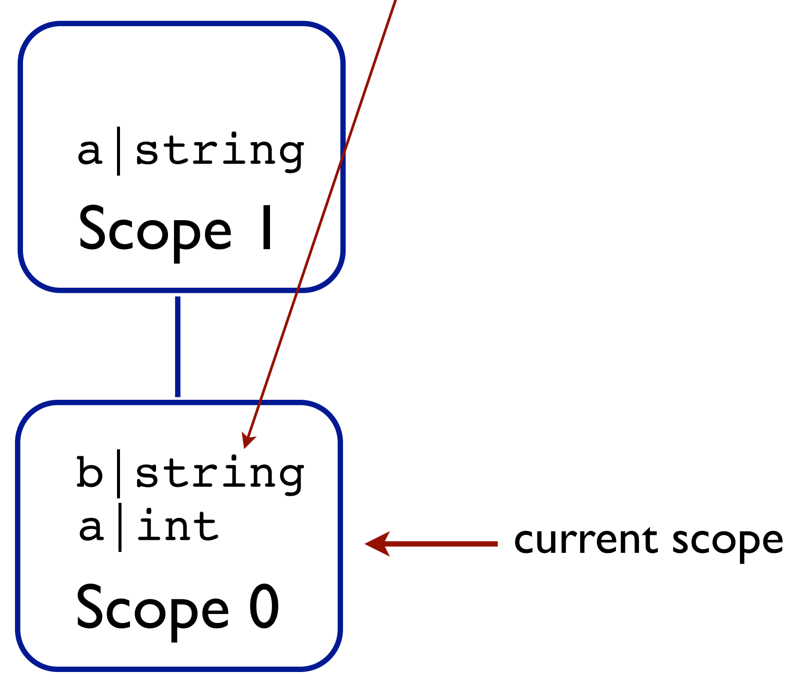
Initialize Scope 0
add symbol a
lookup symbol a
check types
Initialize Scope 1
add symbol a
lookup symbol a
check types
lookup symbol a
Close Scope 1
*Move the **current scope** pointer to its parent.*

# Semantic Analysis




Source Code

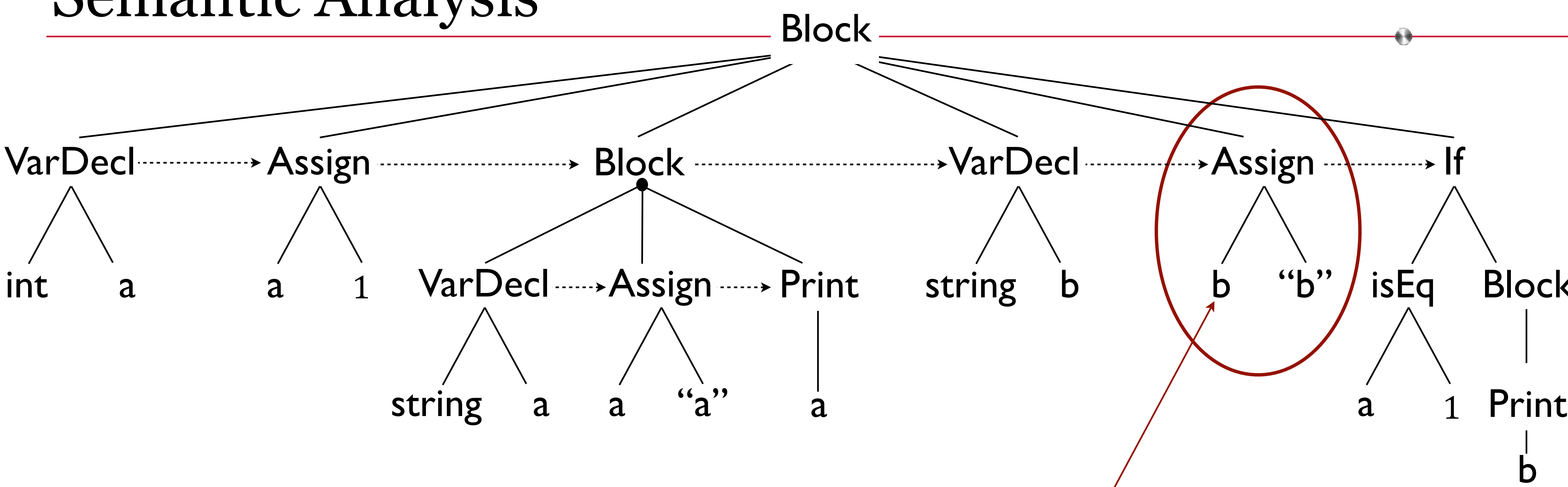
```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```




Symbol Table

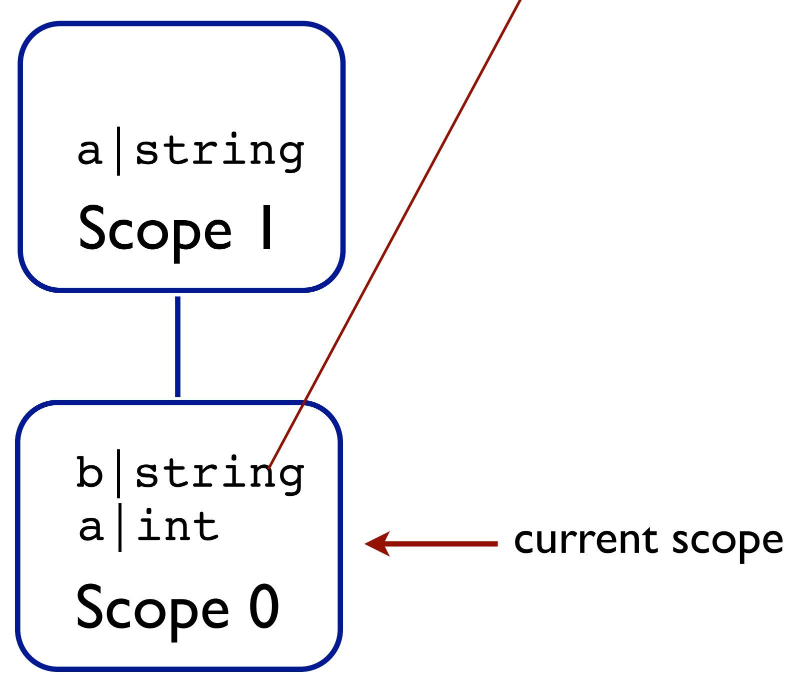
Initialize Scope 0
add symbol a
lookup symbol a
check types
Initialize Scope 1
add symbol a
lookup symbol a
check types
lookup symbol a
Close Scope 1
add symbol b
*in the* ***current scope***

# Semantic Analysis




Source Code

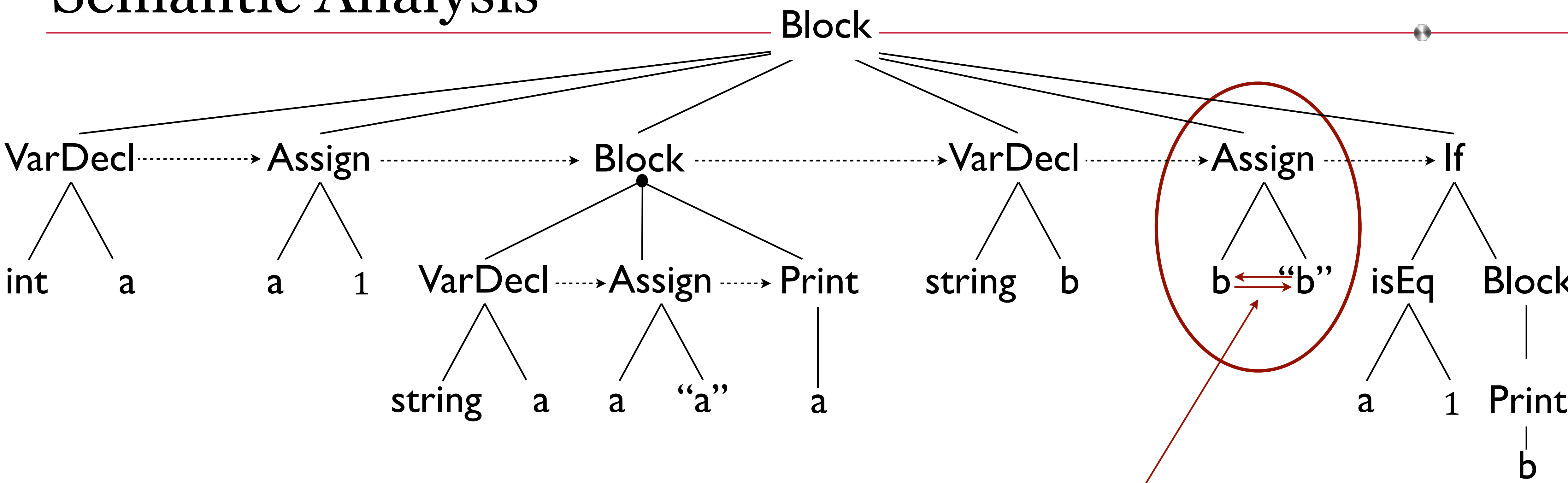
```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```




Symbol Table

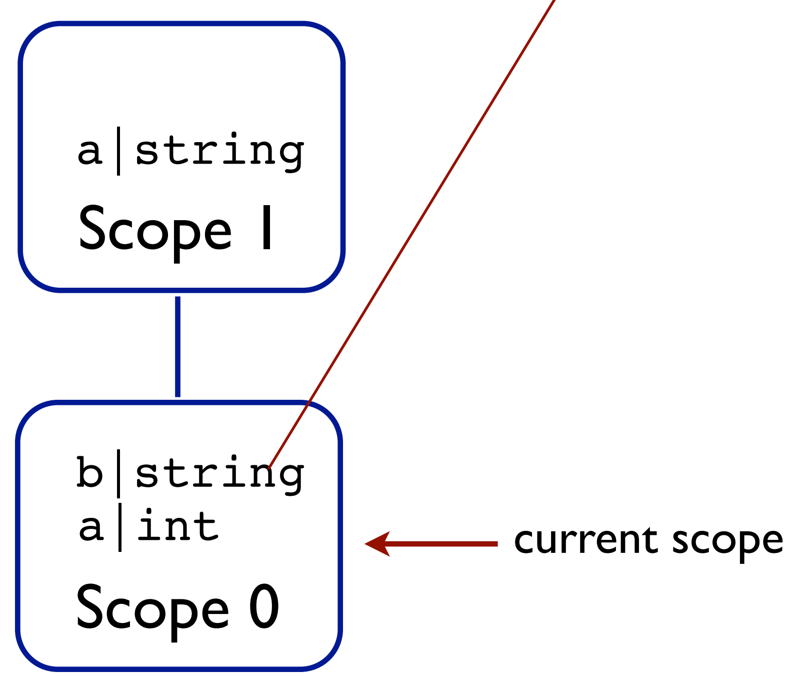
Initialize Scope 0
add symbol a
lookup symbol a
check types
Initialize Scope 1
add symbol a
lookup symbol a
check types
lookup symbol a
Close Scope 1
add symbol b
**lookup symbol b**
*in the* ***current scope***

# Semantic Analysis




Source Code

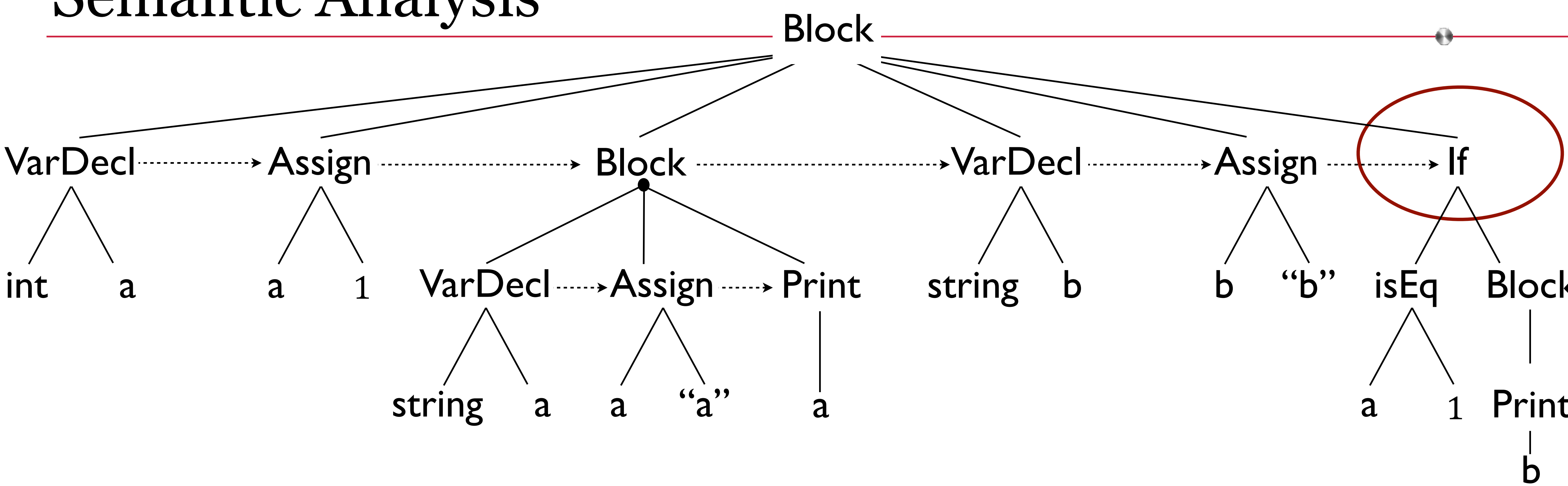
```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```




Symbol Table

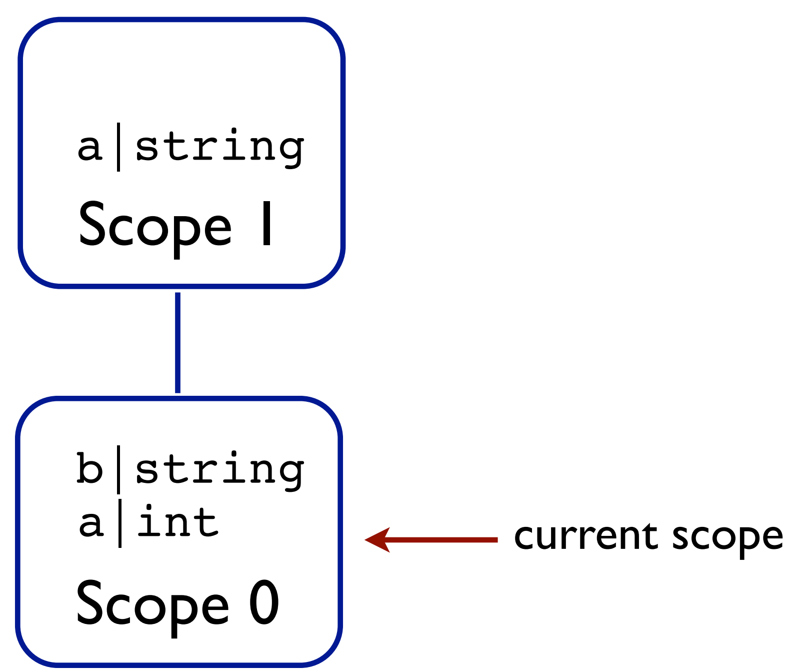
Initialize Scope 0
add symbol a
lookup symbol a
check types
Initialize Scope 1
add symbol a
lookup symbol a
check types
lookup symbol a
Close Scope 1
add symbol b
lookup symbol b
**check types**
*Verify that the left child and right child are type compatible for assignment.*

# Semantic Analysis




Source Code

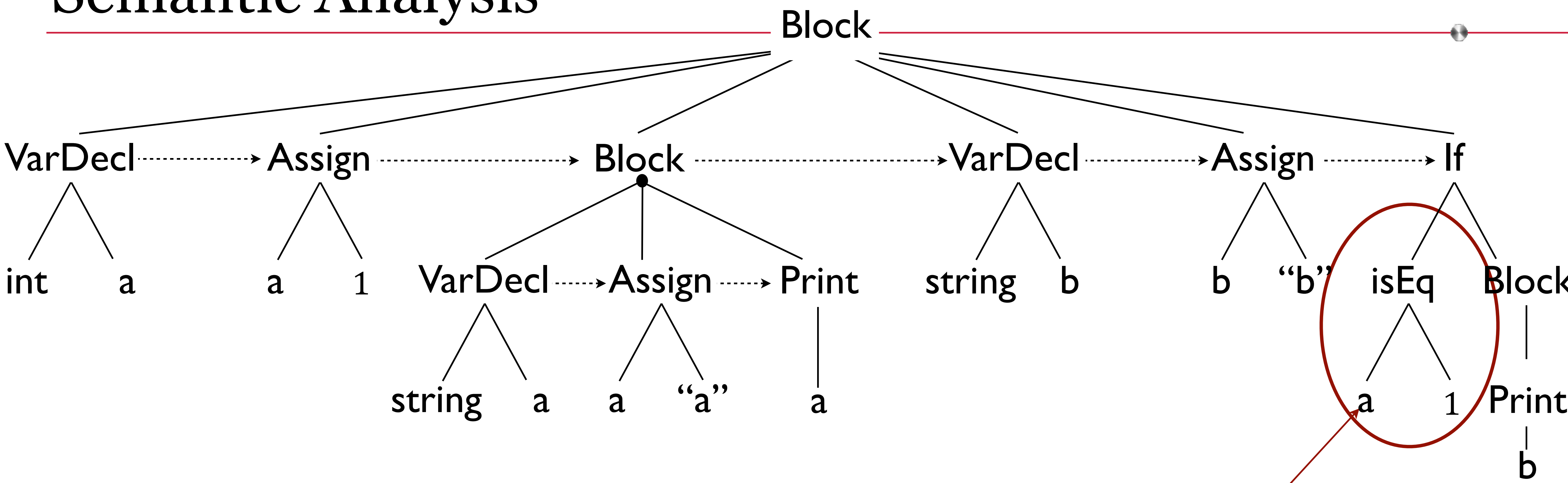
```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```




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

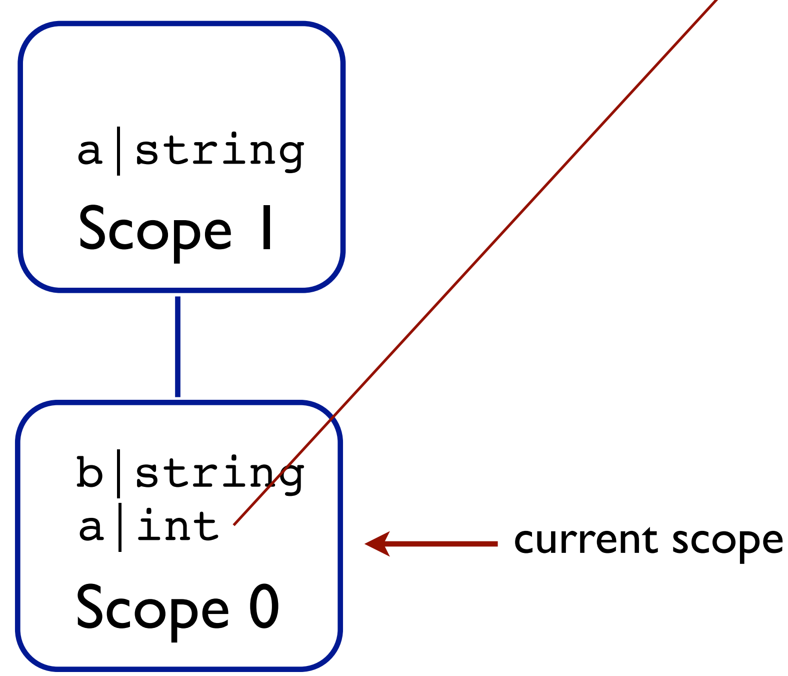
# Semantic Analysis




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***

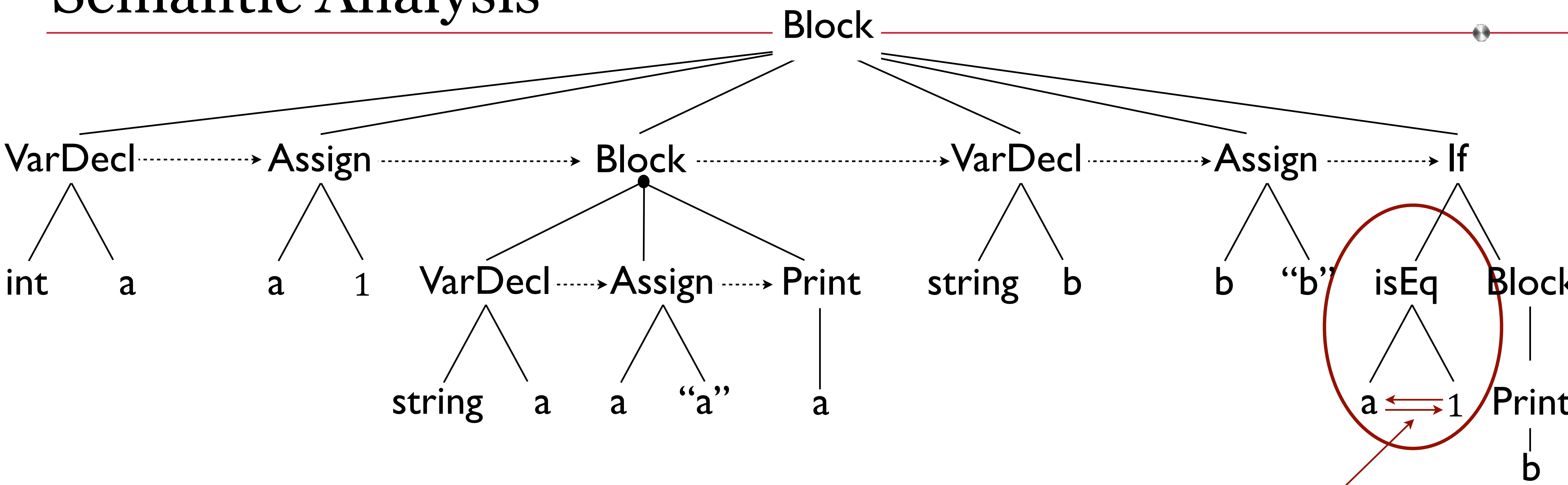
## Source Code

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



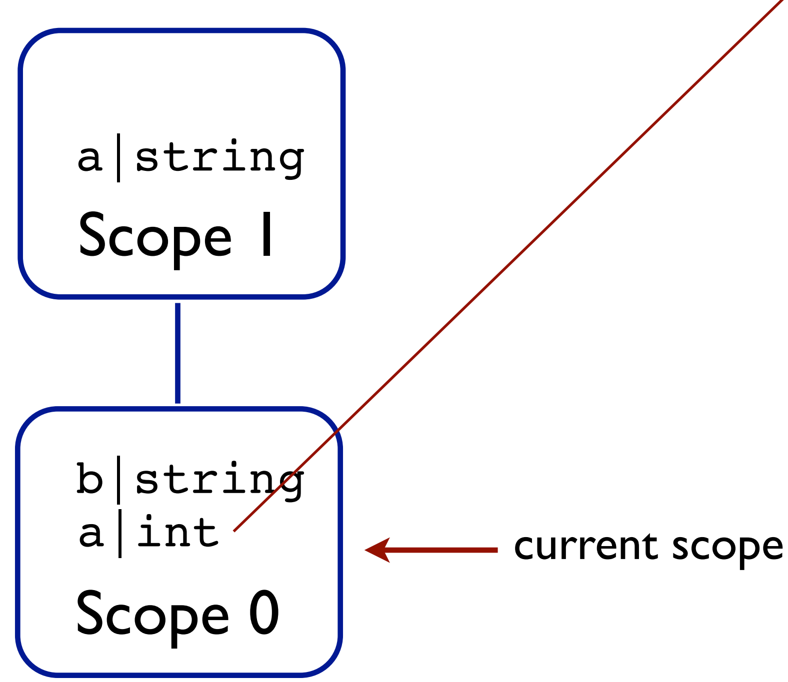

Symbol Table

# Semantic Analysis




## Source Code

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

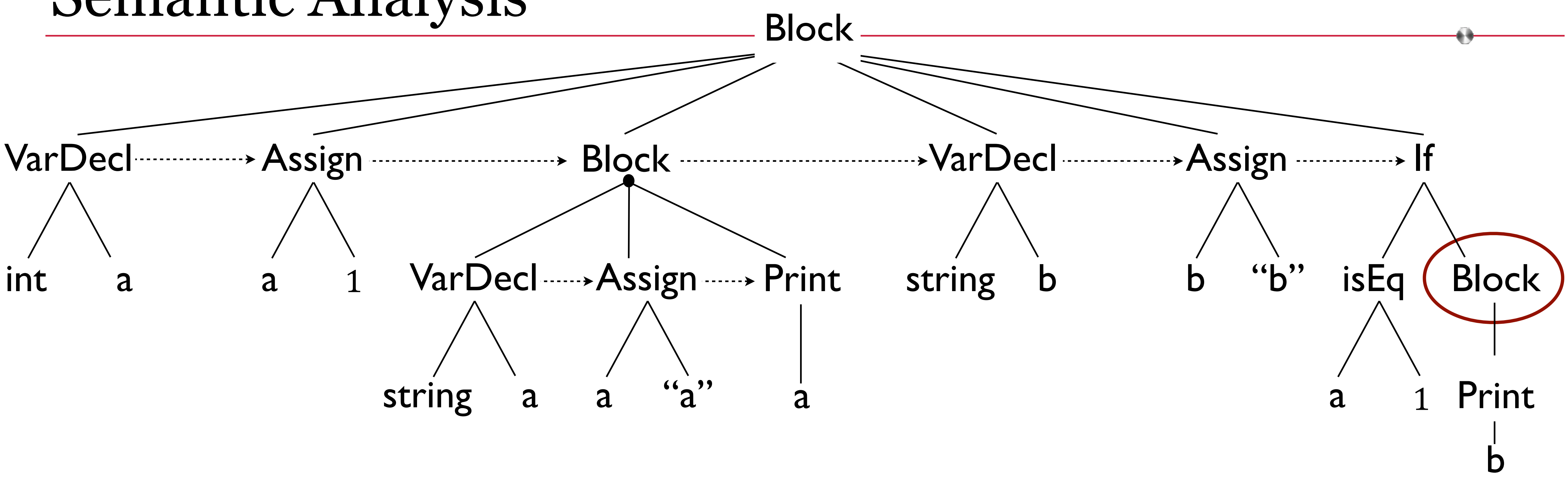



Initialize Scope 0
add symbol a
lookup symbol a
check types
Initialize Scope 1
add symbol a
lookup symbol a
check types
lookup symbol a
Close Scope 1
add symbol b
lookup symbol b
check types
lookup symbol a
check types

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

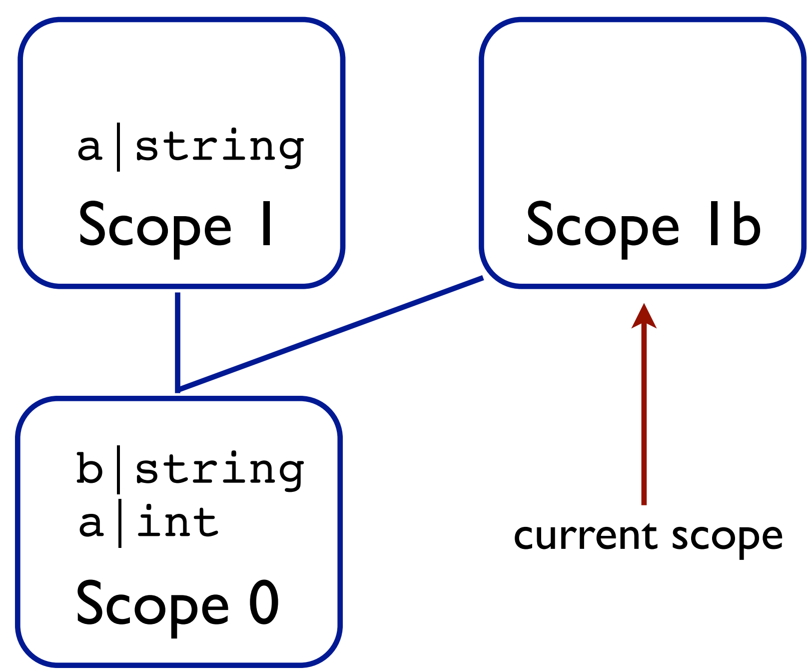
Symbol Table

# Semantic Analysis




## Source Code

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

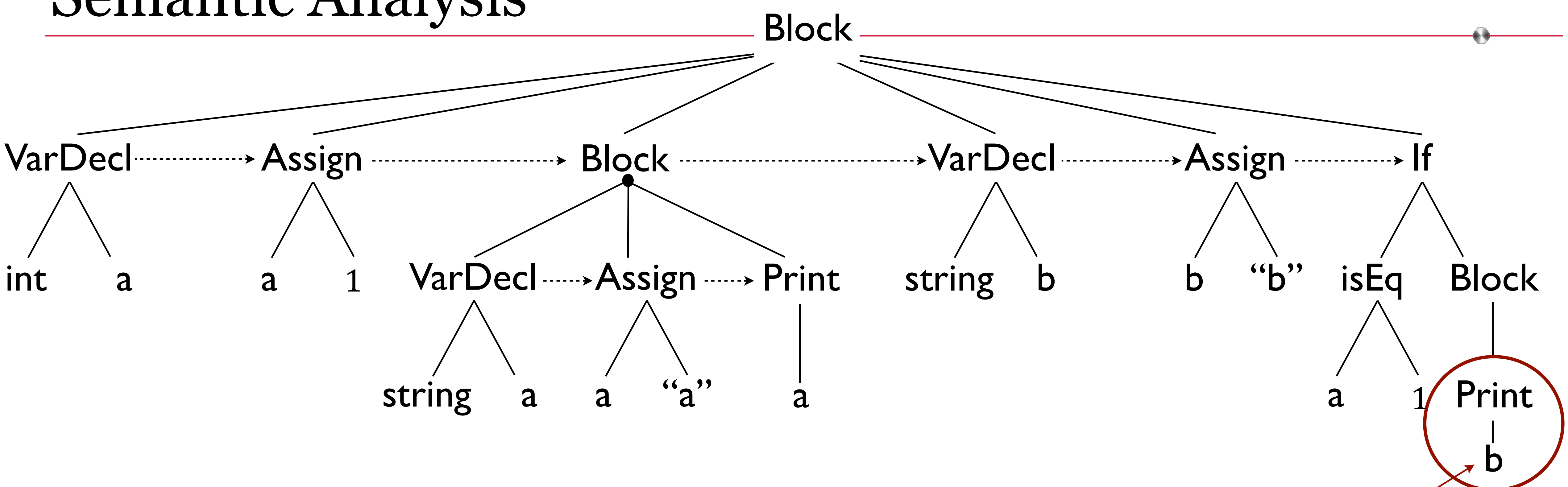



Initialize Scope 0
add symbol a
lookup symbol a
check types
Initialize Scope 1
add symbol a
lookup symbol a
check types
lookup symbol a
Close Scope 1
add symbol b
lookup symbol b
check types
lookup symbol a
check types
Initialize Scope 1b

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

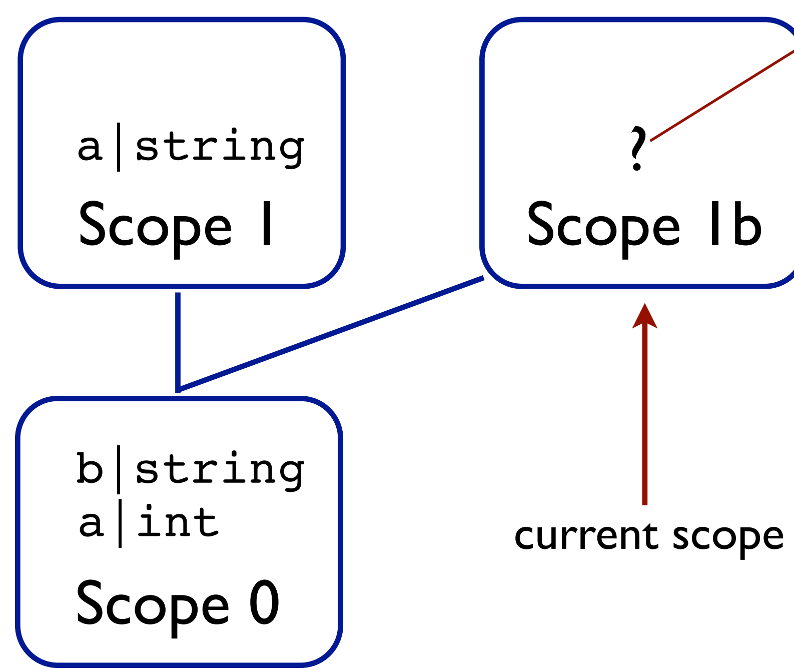
## Symbol Table

# Semantic Analysis




## Source Code

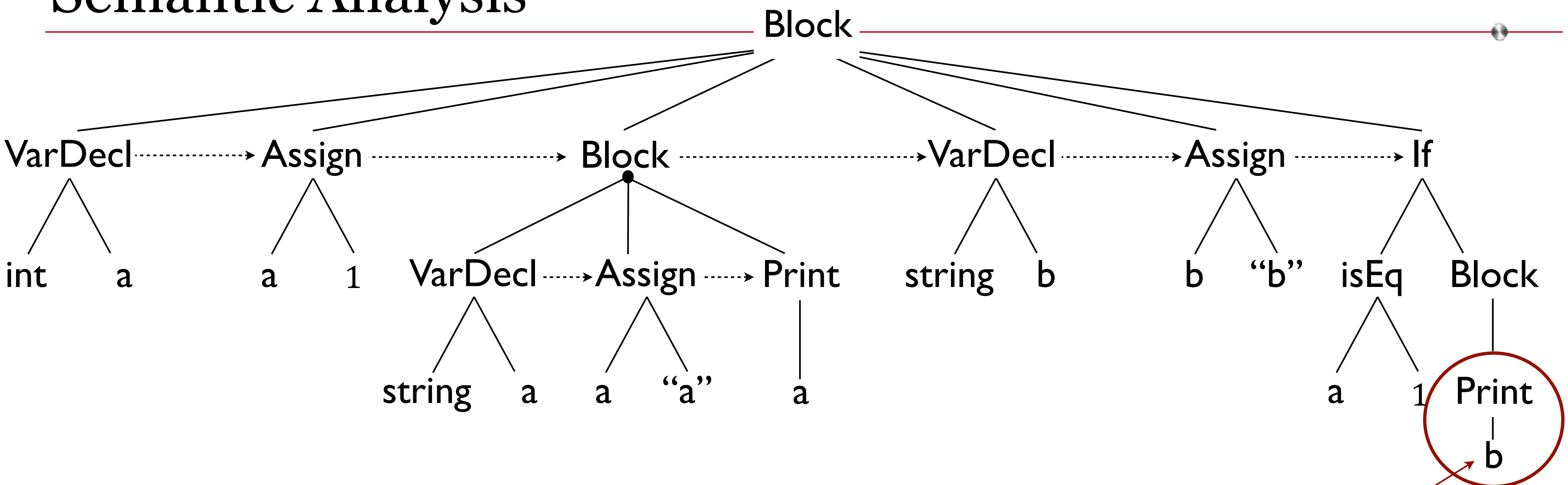
```
{
    int a
    a = 1
    {
        string a
        a = "a"
        print(a)
    }
    string b
    b = "b"
    if (a == 1) {
        print(b)
    }
}
```




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
check types
Initialize Scope 1b
lookup symbol b
*in the* ***current scope***.

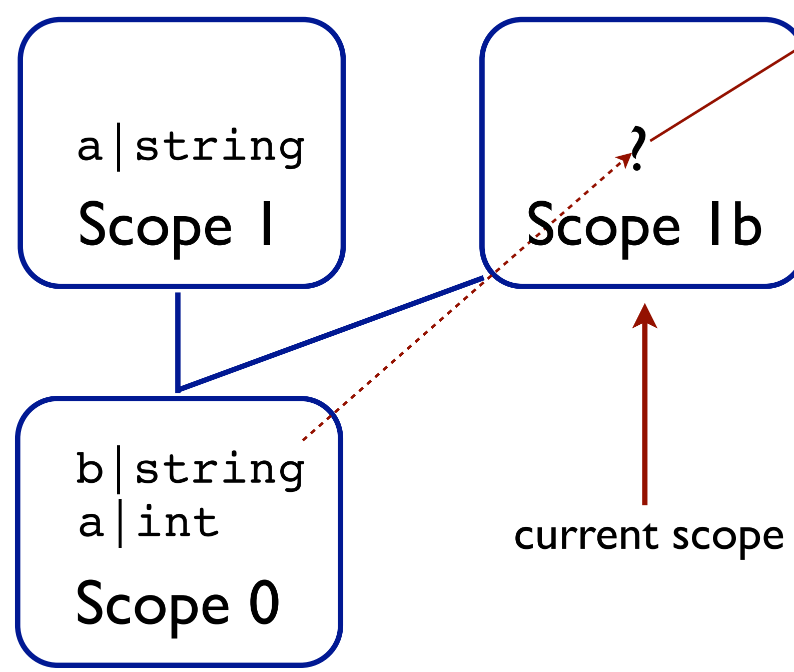
Symbol Table

# Semantic Analysis




## Source Code

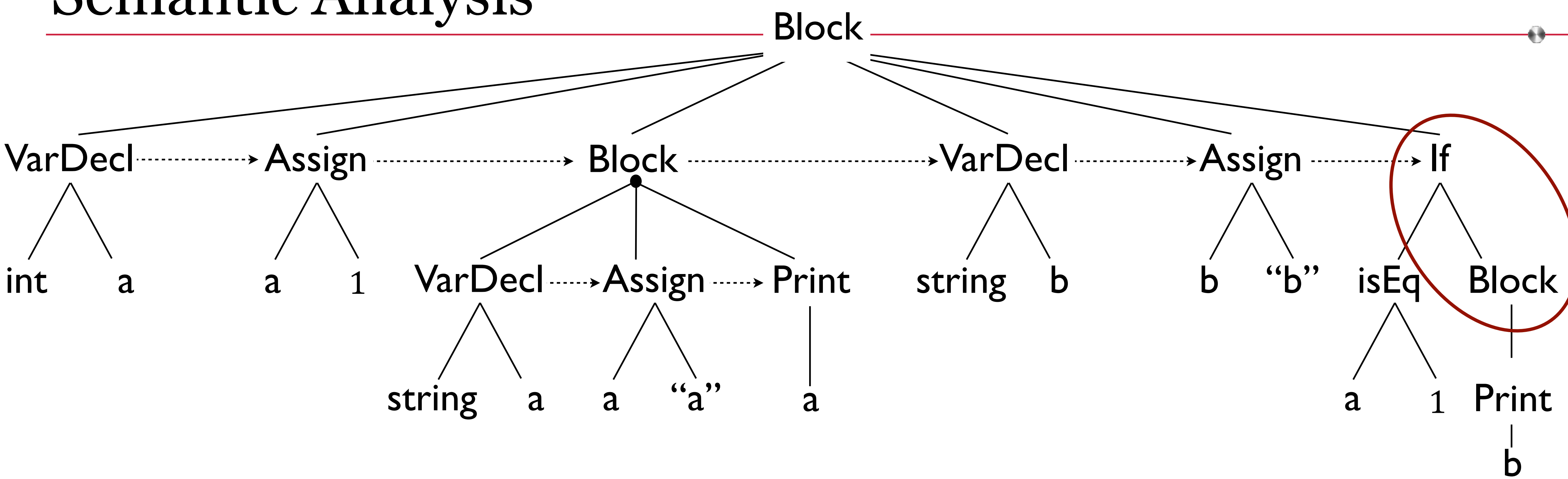
```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```




Symbol Table

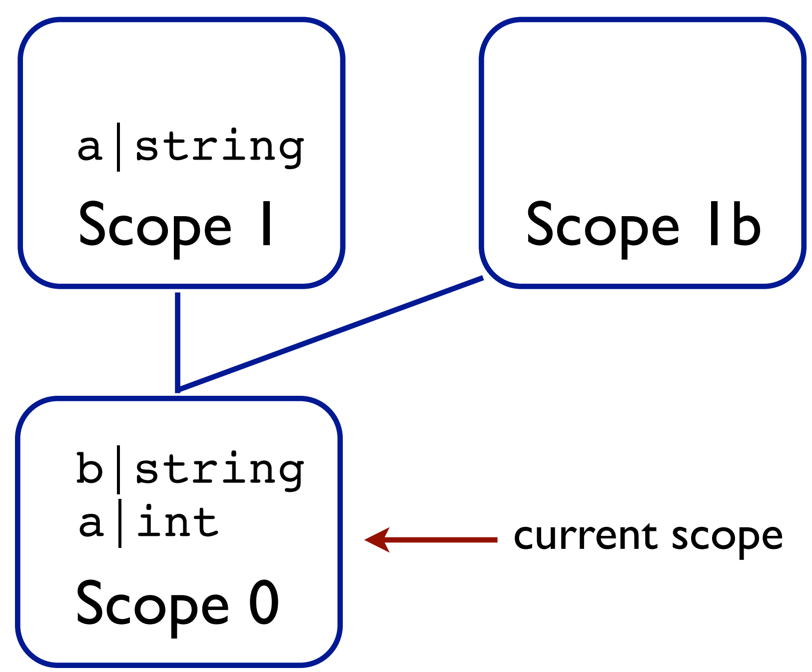
Initialize Scope 0
add symbol a
lookup symbol a
check types
Initialize Scope 1
add symbol a
lookup symbol a
check types
lookup symbol a
Close Scope 1
add symbol b
lookup symbol b
check types
lookup symbol a
check types
Initialize Scope 1b
lookup symbol b
*in the **parent scope**.*
*Print can take any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

# Semantic Analysis




Source Code

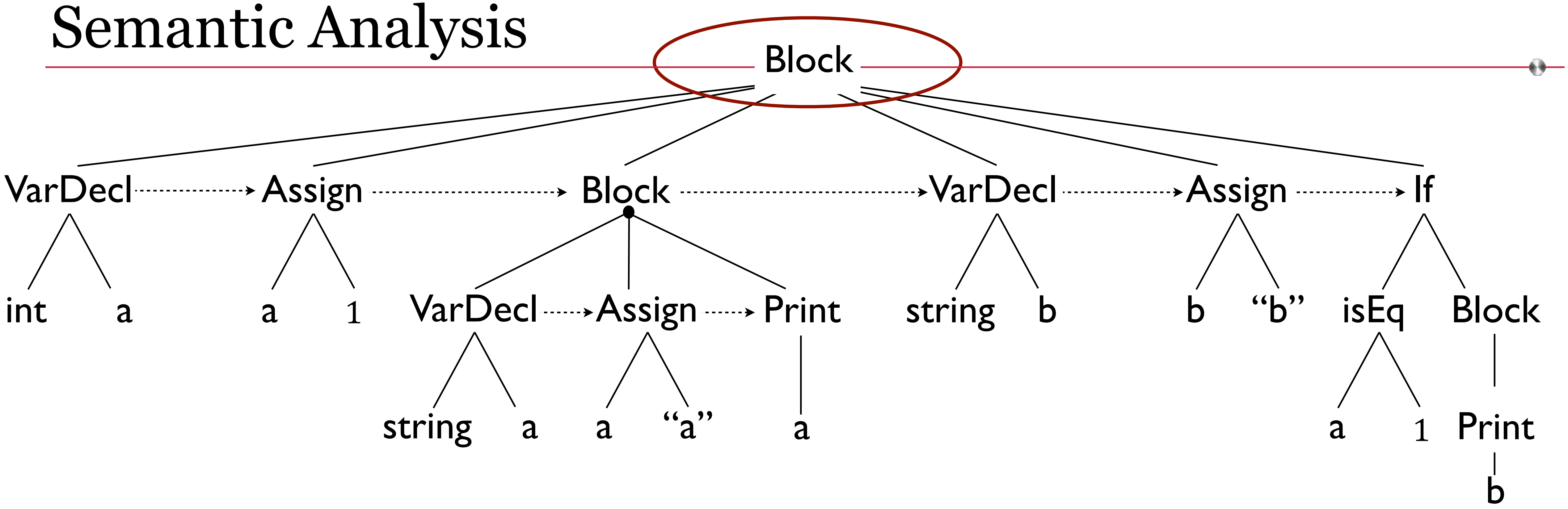
```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```




Symbol Table

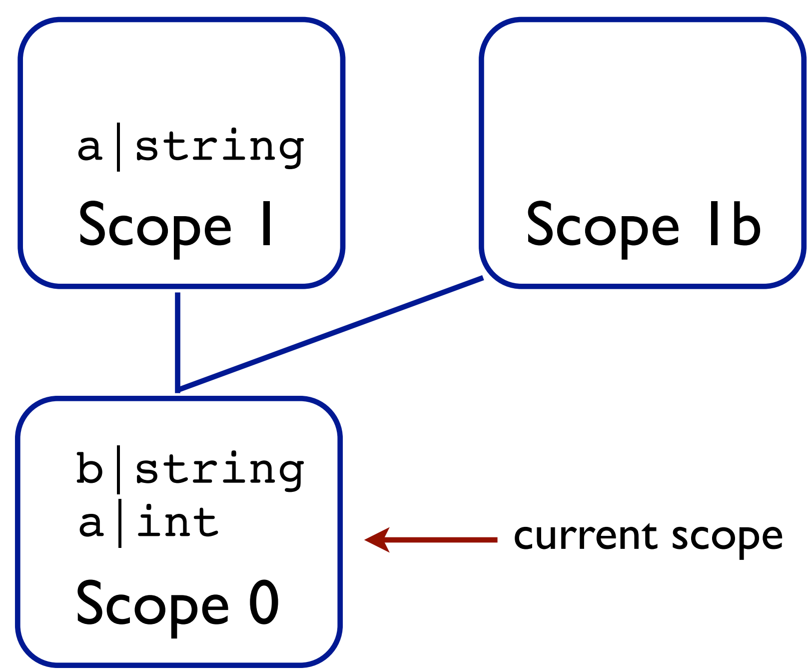
Initialize Scope 0
add symbol a
lookup symbol a
check types
Initialize Scope 1
add symbol a
lookup symbol a
check types
lookup symbol a
Close Scope 1
add symbol b
lookup symbol b
check types
lookup symbol a
check types
Initialize Scope 1b
lookup symbol b
Close Scope 1b

# Semantic Analysis




Source Code

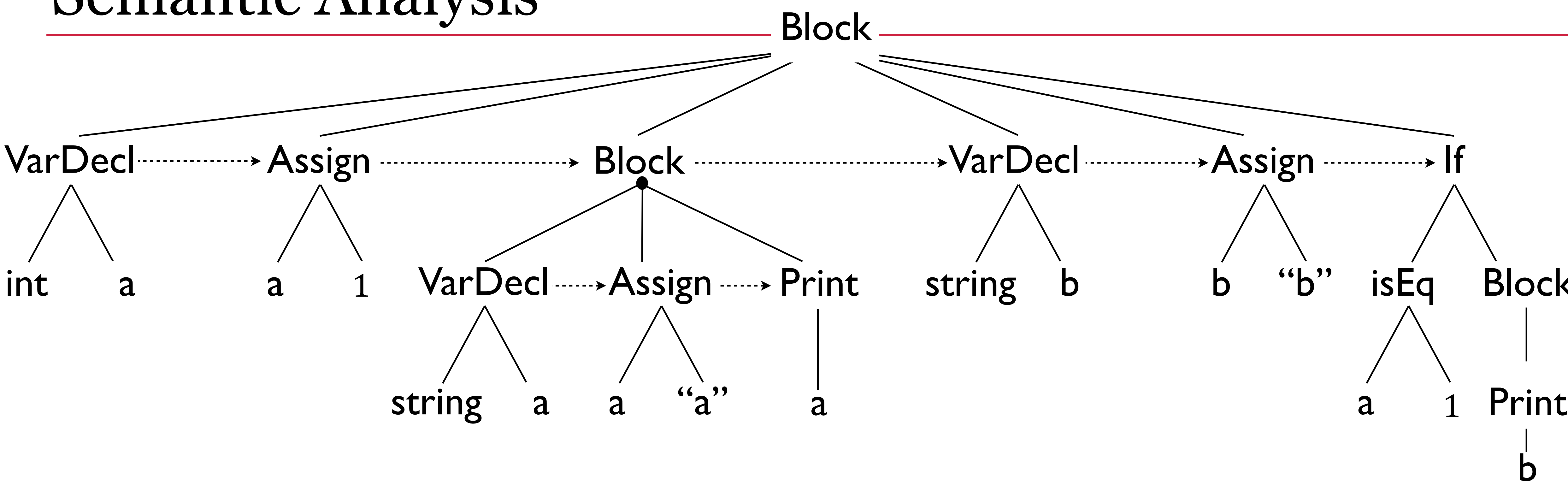
```
{
    int a
    a = 1
    {
        string a
        a = "a"
        print(a)
    }
    string b
    b = "b"
    if (a == 1) {
        print(b)
    }
}
```




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

# Semantic Analysis




Now we have a lexically, syntactically, and semantically correct AST and a complete symbol table to go with it.