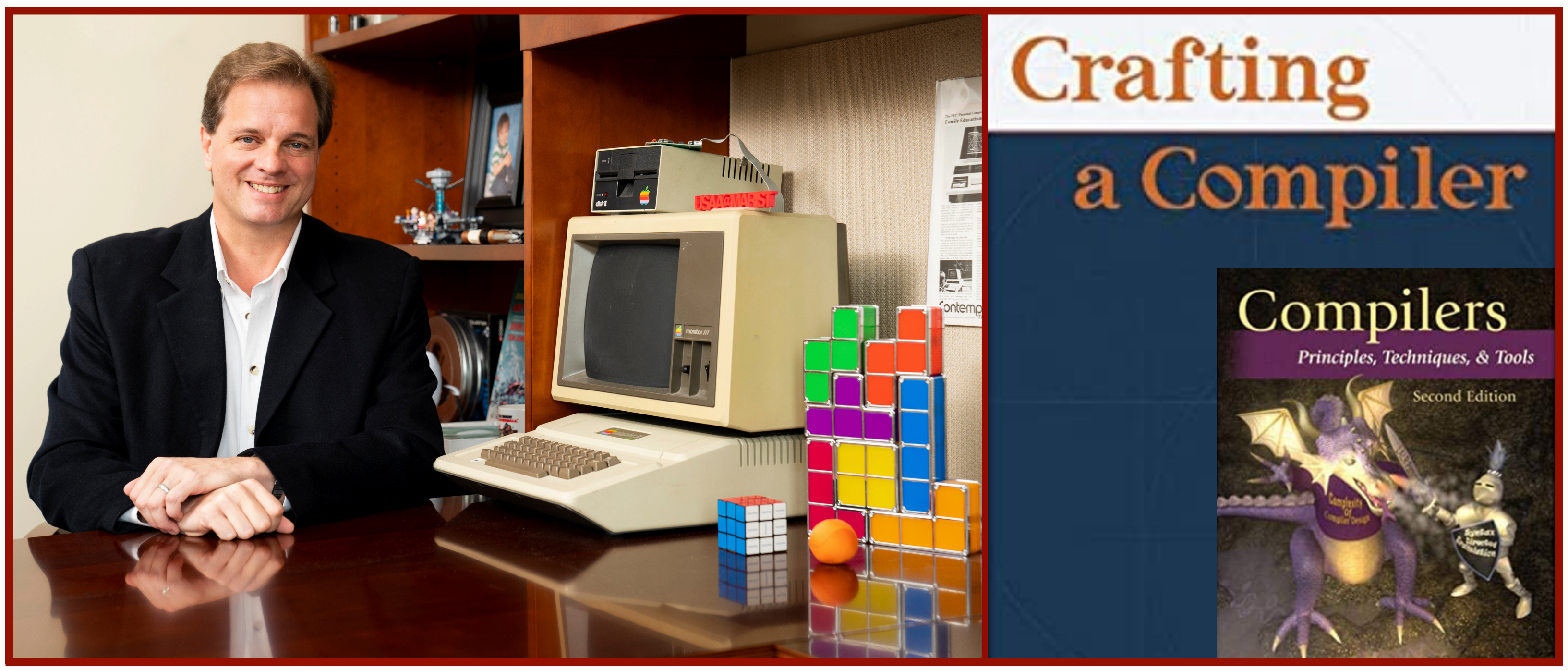
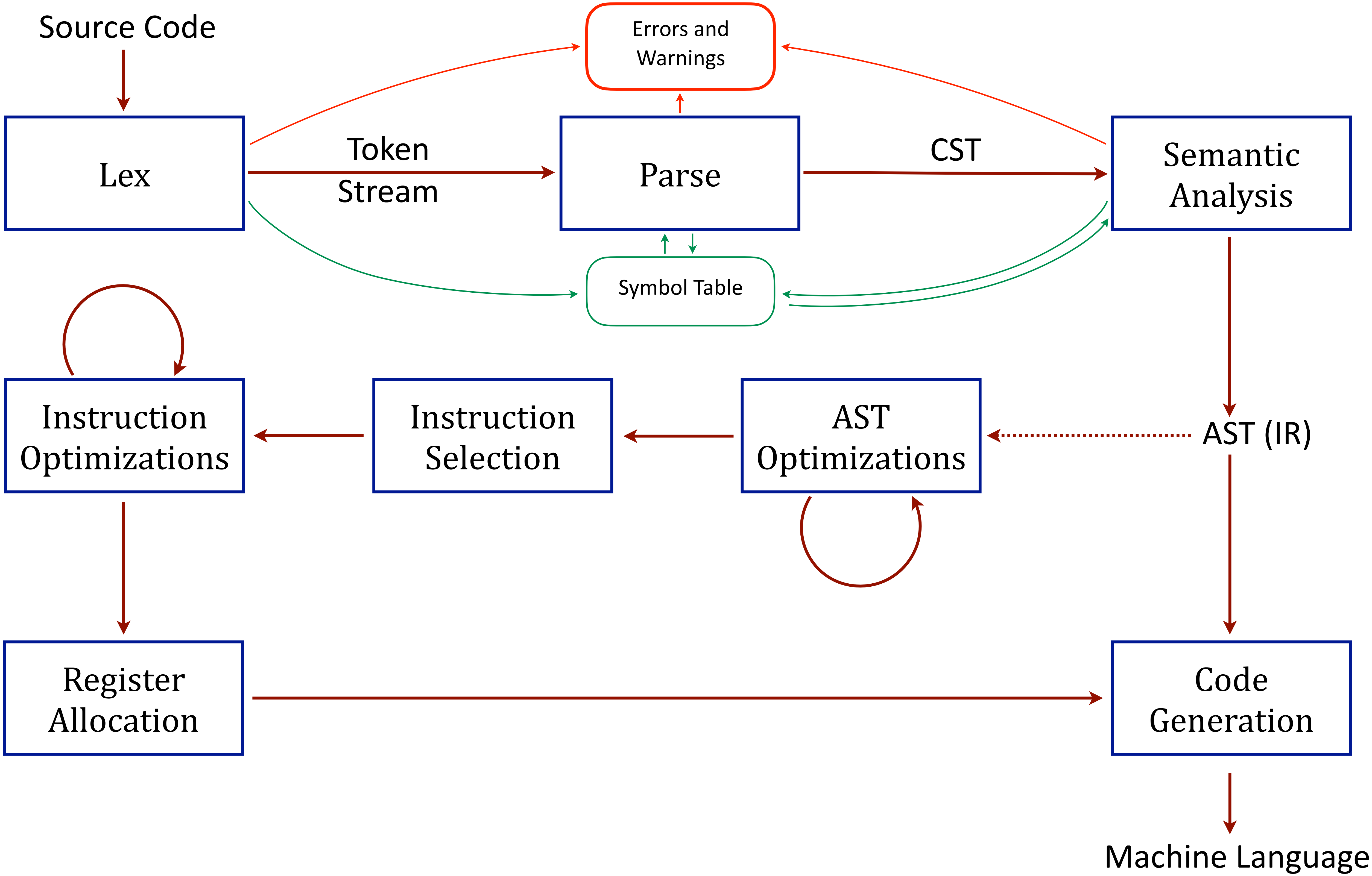
Semantic Analysis

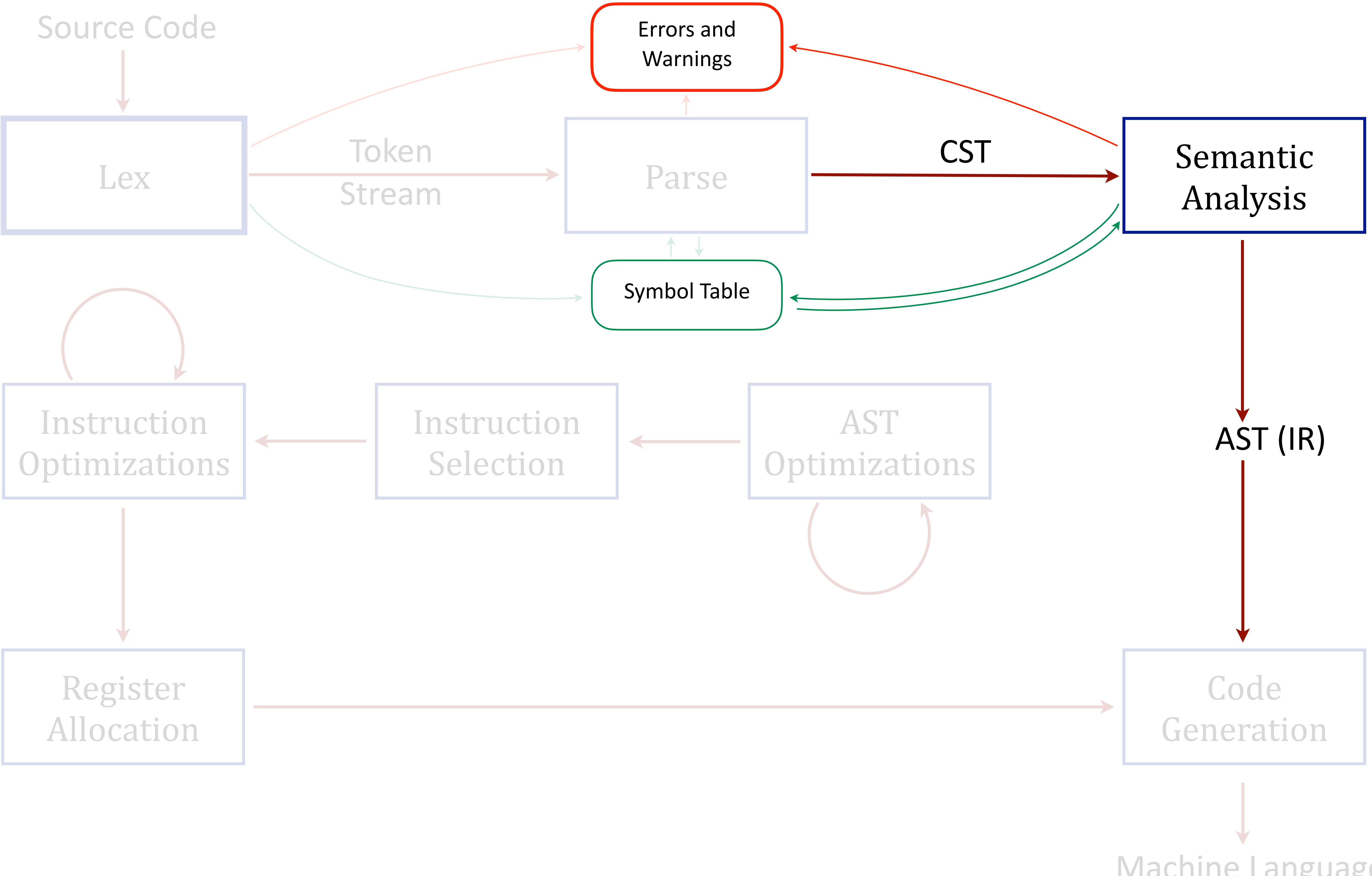


Alan G. Labouseur, Ph.D.
Alan.Labouseur@Marist.edu

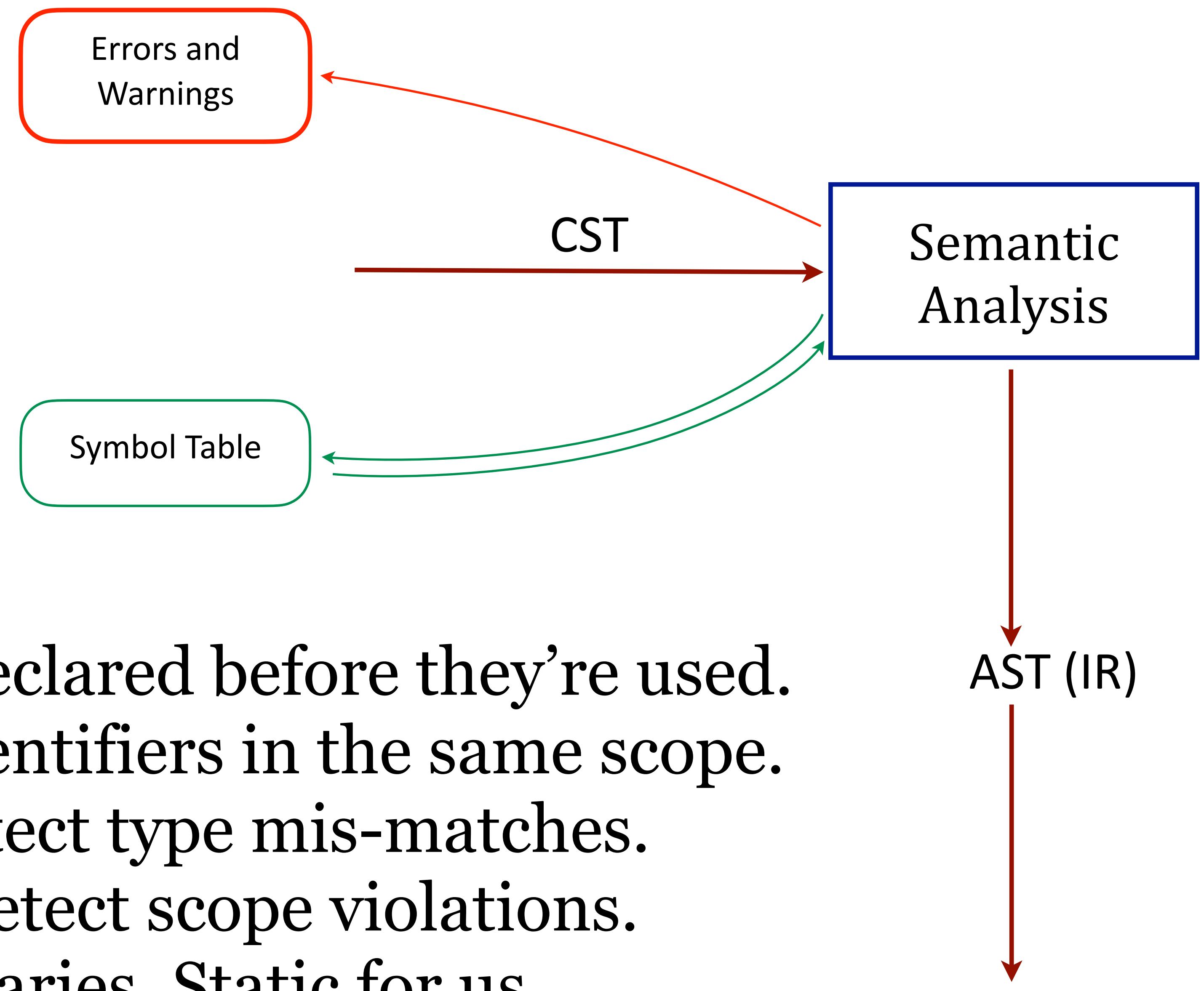
Compiler — High Level View



Compiler — High Level View



High Level View: Semantic Analysis



Semantic Analysis

- Verify all identifiers are declared before they're used.
- Detect redeclaration of identifiers in the same scope.
- Enforce type rules and detect type mis-matches.
- Enforce scope rules and detect scope violations.
  - Static or Dynamic? It varies. Static for us.
- Produce a valid Abstract Syntax Tree and Symbol Table for Code Generation.
- Focus on meaning: scope and type.
- But first, a few words about identifiers, variables, and scope.

Identifiers, Variables, and Scope

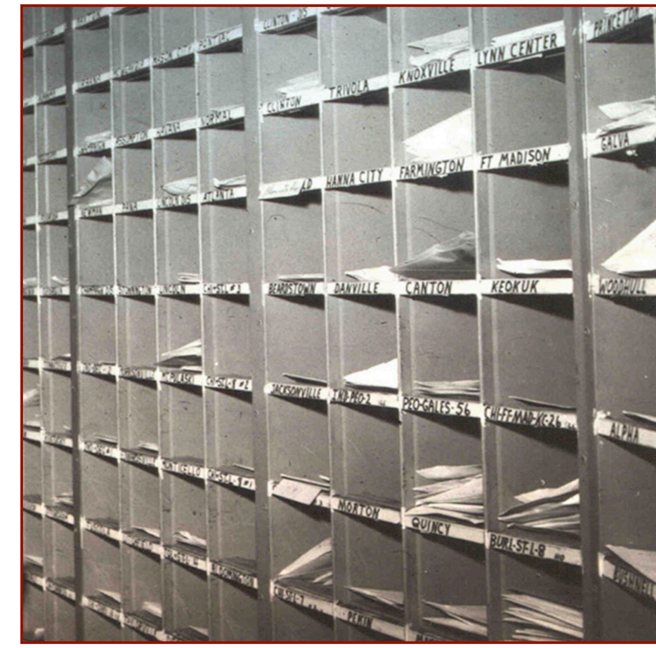
Identifiers =? Names

- 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)
    - Our grammar: 1

Identifiers, Variables, and Scope



A variable is an abstraction of a memory cell.

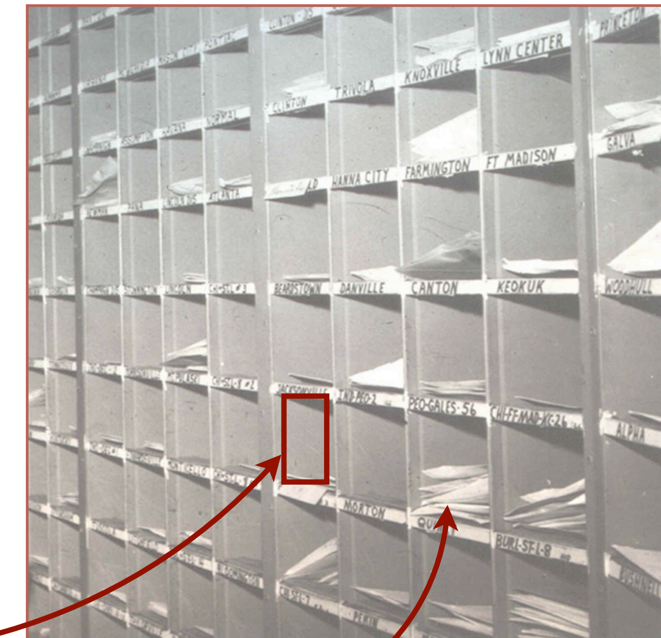
- Think of a post office model.

Variables are characterized by attributes

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

  … and more

Identifiers, Variables, and Scope



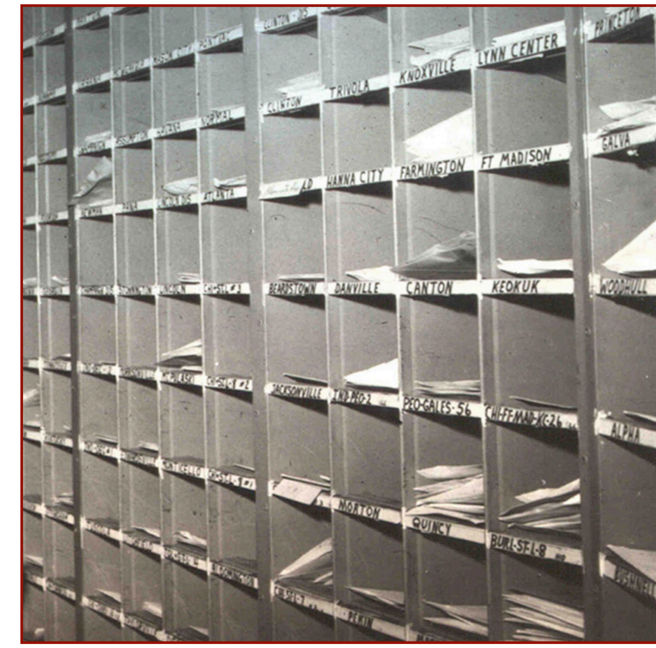
A variable is an abstraction of a memory cell.

- 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

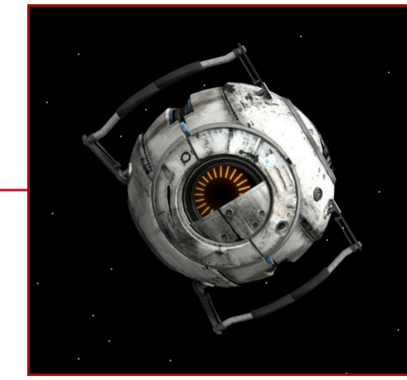
- A binding is **static** if it first occurs before run time and remains unchanged throughout program execution.
- A binding is **dynamic** if it first occurs during execution or can change during execution of the program.

Static and Dynamic Scope

What's the output of this code?

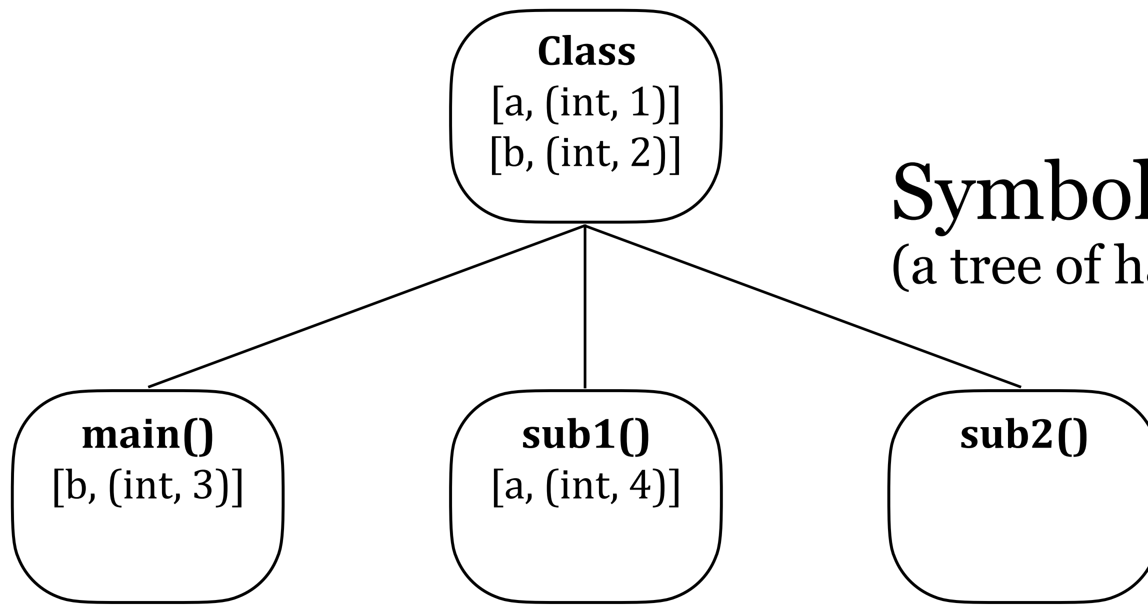
```
Class ScopeMan {
    int a := 1;
    int b := 2;

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

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

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

Static and Dynamic Scope



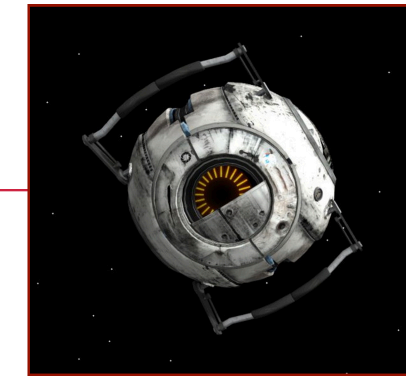
Static Scope

```
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);
   }
}
```



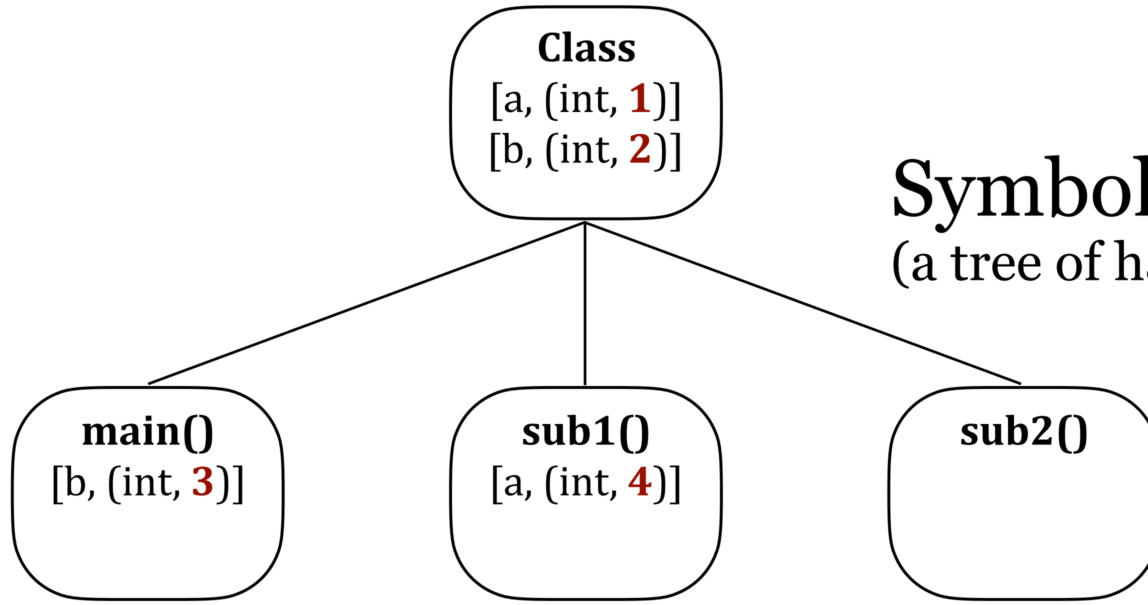
Symbol Table
(a tree of hash tables)

Static and Dynamic Scope



Static Scope

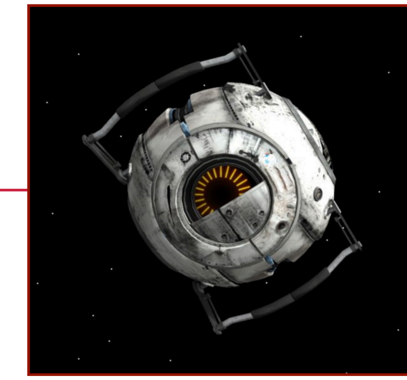
```
Class ScopeMan {
   int a := 1;
   int b := 2;

   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)

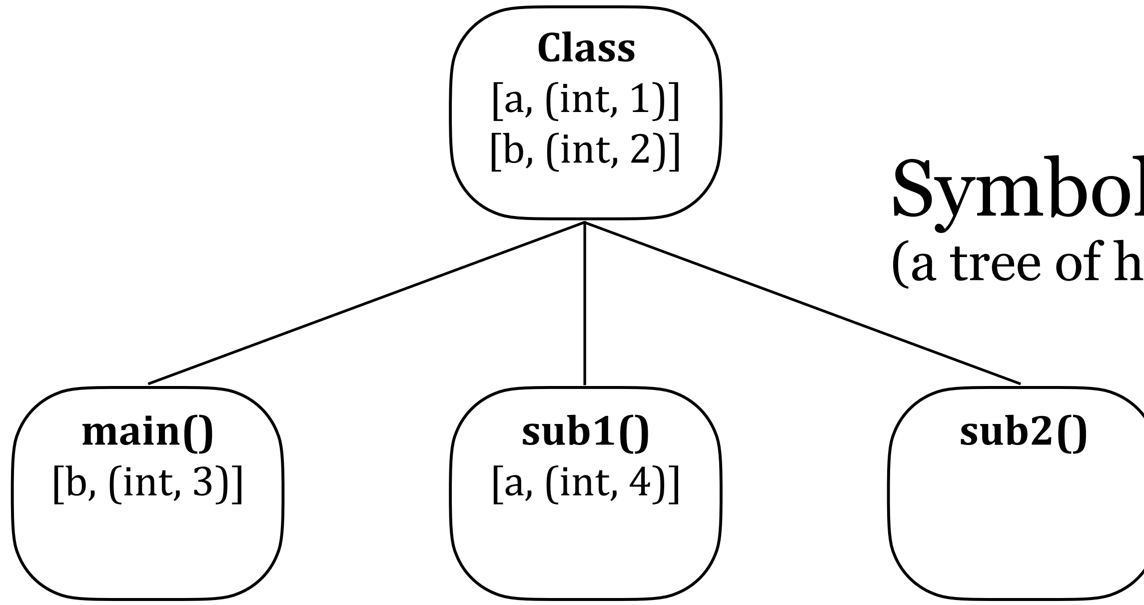
Note: We don't actually store the value of the ids in the symbol table, as they will be stored in memory. They're present in this depiction of a symbol table only so that we can easily see what the output should be. We'll have other attributes to store in the hash table along with the ids later on.

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);
   }
}
```



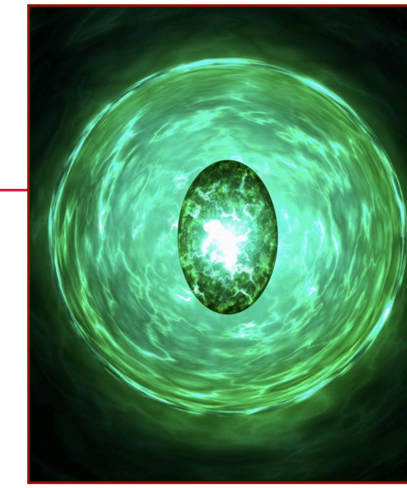
Symbol Table
(a tree of hash tables)

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

Static scope is . . .

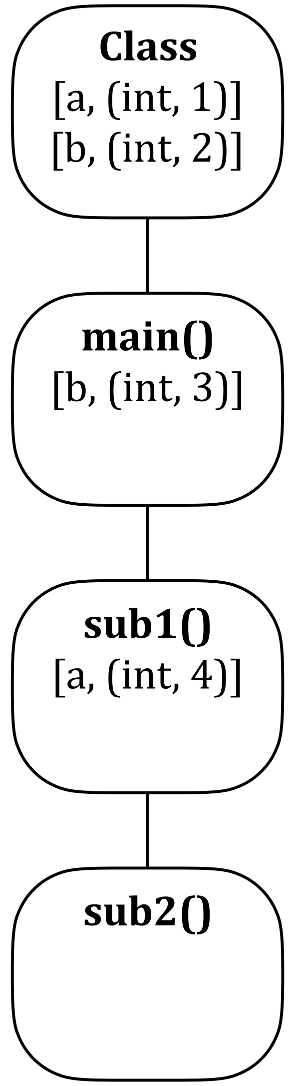
- Early binding
- Compile time
- about Space,
  - the shape of the code
  - the spacial relationships of code modules to each other at compile time.

Static and Dynamic Scope



Dynamic Scope

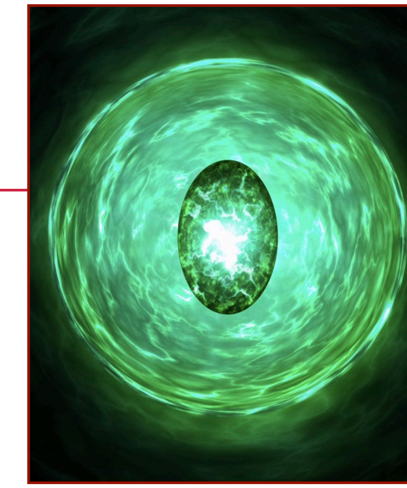
```
Class ScopeMan {
   int a := 1;
   int b := 2;

   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)

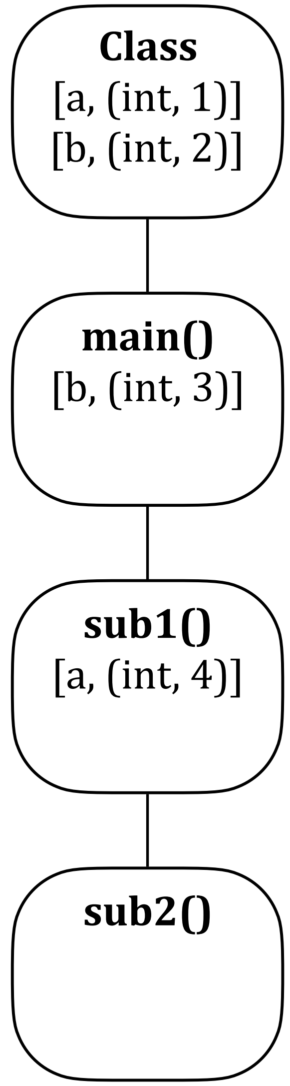
Yes, this is a tree. It's also a list. And it looks like a stack. But it's a tree. And a graph. Let's just think of it as a tree.

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.

Abstract Syntax Trees

An Abstract Syntax Tree (AST) is like a Concrete Syntax Tree (CST) but with just the “good stuff” in there.

A CST is concrete because it contains all of the concrete details of the parse/derivation. Once parsing is complete and error-free, we know that the syntax is right so we don’t need to carry along all the concrete details that prove its correctness in the parse tree. Instead we take only what we need for Semantic Analysis, optimizations, and Code Generation and put that “good stuff” in an AST.

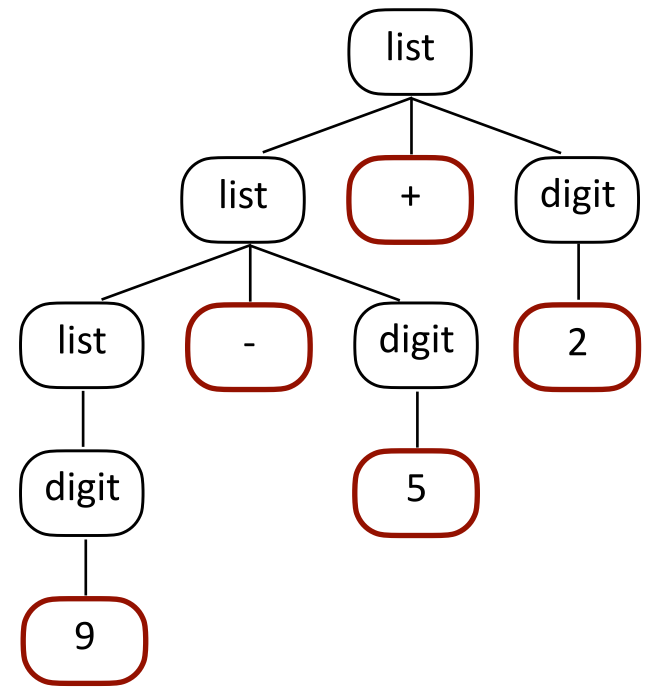
The AST is our Intermediate Representation (IR). It’s what we use for Semantic Analysis, and what we pass on (along with the Symbol Table) to the back end of our compiler.

Abstract Syntax Trees

Recall our simple grammar and the Concrete Syntax Tree representing the derivation of 9 - 5 + 2 in that grammar:

```
1. <list>  ::== <list> + <digit>
2. <list>  ::== <list> - <digit>
3. <list>  ::== <digit>
4. <digit> ::== 0|1|2|3|4|5|6|7|8|9
```

```
S. <list>
1. <list>                + <digit>
4. <list>                +   2
2. <list>  - <digit>     +   2
4. <list>  -   5         +   2
3. <digit> -   5         +   2
4.    9    -   5         +   2
```



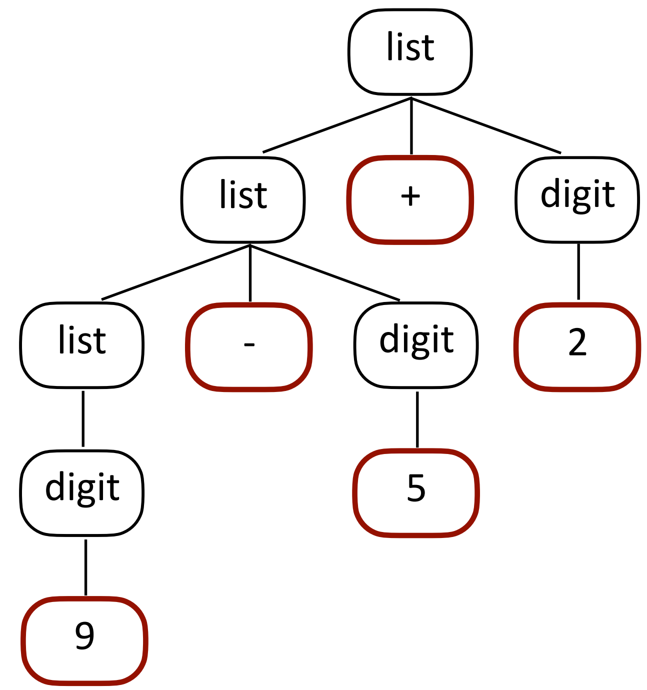
CST

Abstract Syntax Trees

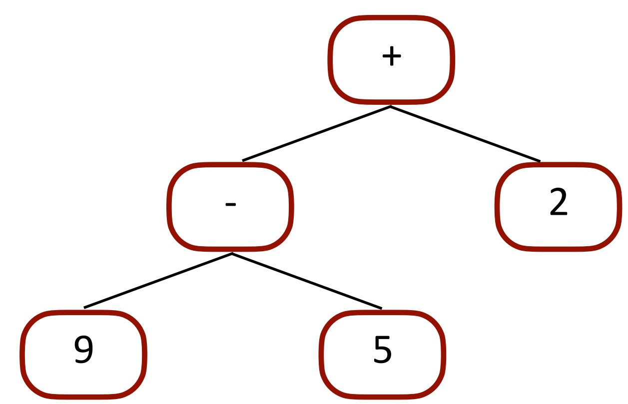
Recall our simple grammar and the Concrete Syntax Tree representing the derivation of 9 - 5 + 2 in that grammar:

```
1. <list>  ::== <list> + <digit>
2. <list>  ::== <list> - <digit>
3. <list>  ::== <digit>
4. <digit> ::== 0|1|2|3|4|5|6|7|8|9
```

```
S. <list>
1. <list>                + <digit>
4. <list>                +   2
2. <list>  -  <digit>    +   2
4. <list>  -    5        +   2
3. <digit> -    5        +   2
4.    9    -    5        +   2
```



CST



AST

Just the good parts.

Abstract Syntax Trees

Consider a program in our grammar:

```
{
   int a
   a = 1
   print(a)
}
```

```
Lexical Analysis...
LEXER -> | TLbrace [ { ] on line 1 col 0
LEXER -> | TType [ int ] on line 2 col 3
LEXER -> | TId [ a ] on line 2 col 7
LEXER -> | TId [ a ] on line 3 col 3
LEXER -> | TAssign [ = ] on line 3 col 5
LEXER -> | TDigit [ 1 ] on line 3 col 7
LEXER -> | TPrint [ print ] on line 4 col 3
LEXER -> | TLparen [ ( ] on line 4 col 8
LEXER -> | TId [ a ] on line 4 col 9
LEXER -> | TRparen [ ) ] on line 4 col 10
LEXER -> | TRbrace [ } ] on line 5 col 0
LEXER -> | WARNING: No EOP [$] detected at end-of-file. Adding to end-of-file...

Parsing...
PARSER -> | VALID - Expecting [Program], found [Block] on line 1 col 0
PARSER -> | VALID - Expecting [TLbrace], found [{] on line 1 col 0
PARSER -> | VALID - Expecting [StatementList], found [Statement] on line 2 col 3
PARSER -> | VALID - Expecting [Statement], found [VarDecl] on line 2 col 3
PARSER -> | VALID - Expecting [VarDecl], found [Type] on line 2 col 3
PARSER -> | VALID - Expecting [TType], found [int] on line 2 col 3
PARSER -> | VALID - Expecting [Id], found [Id] on line 2 col 7
PARSER -> | VALID - Expecting [TId], found [a] on line 2 col 7
PARSER -> | VALID - Expecting [StatementList], found [Statement] on line 3 col 3
PARSER -> | VALID - Expecting [Statement], found [AssignmentStatement] on line 3 col 3
PARSER -> | VALID - Expecting [AssignmentStatement], found [Id] on line 3 col 3
PARSER -> | VALID - Expecting [TId], found [a] on line 3 col 3
PARSER -> | VALID - Expecting [TAssign], found [=] on line 3 col 5
PARSER -> | VALID - Expecting [Expression], found [IntegerExpression] on line 3 col 7
PARSER -> | VALID - Expecting [IntegerExpression], found [Digit] on line 3 col 7
PARSER -> | VALID - Expecting [TDigit], found [1] on line 3 col 7
PARSER -> | VALID - Expecting [StatementList], found [Statement] on line 4 col 3
PARSER -> | VALID - Expecting [Statement], found [PrintStatement] on line 4 col 3
PARSER -> | VALID - Expecting [TPrint], found [print] on line 4 col 3
PARSER -> | VALID - Expecting [TLparen], found [(] on line 4 col 8
PARSER -> | VALID - Expecting [Expression], found [Id] on line 4 col 9
PARSER -> | VALID - Expecting [TId], found [a] on line 4 col 9
PARSER -> | VALID - Expecting [TRparen], found [)] on line 4 col 10
PARSER -> | VALID - Found ε on line 5 col 0
PARSER -> | VALID - Expecting [TRbrace], found [}] on line 5 col 0
PARSER -> | VALID - Expecting [TEop], found [$] on line 6 col 0
```

Abstract Syntax Trees

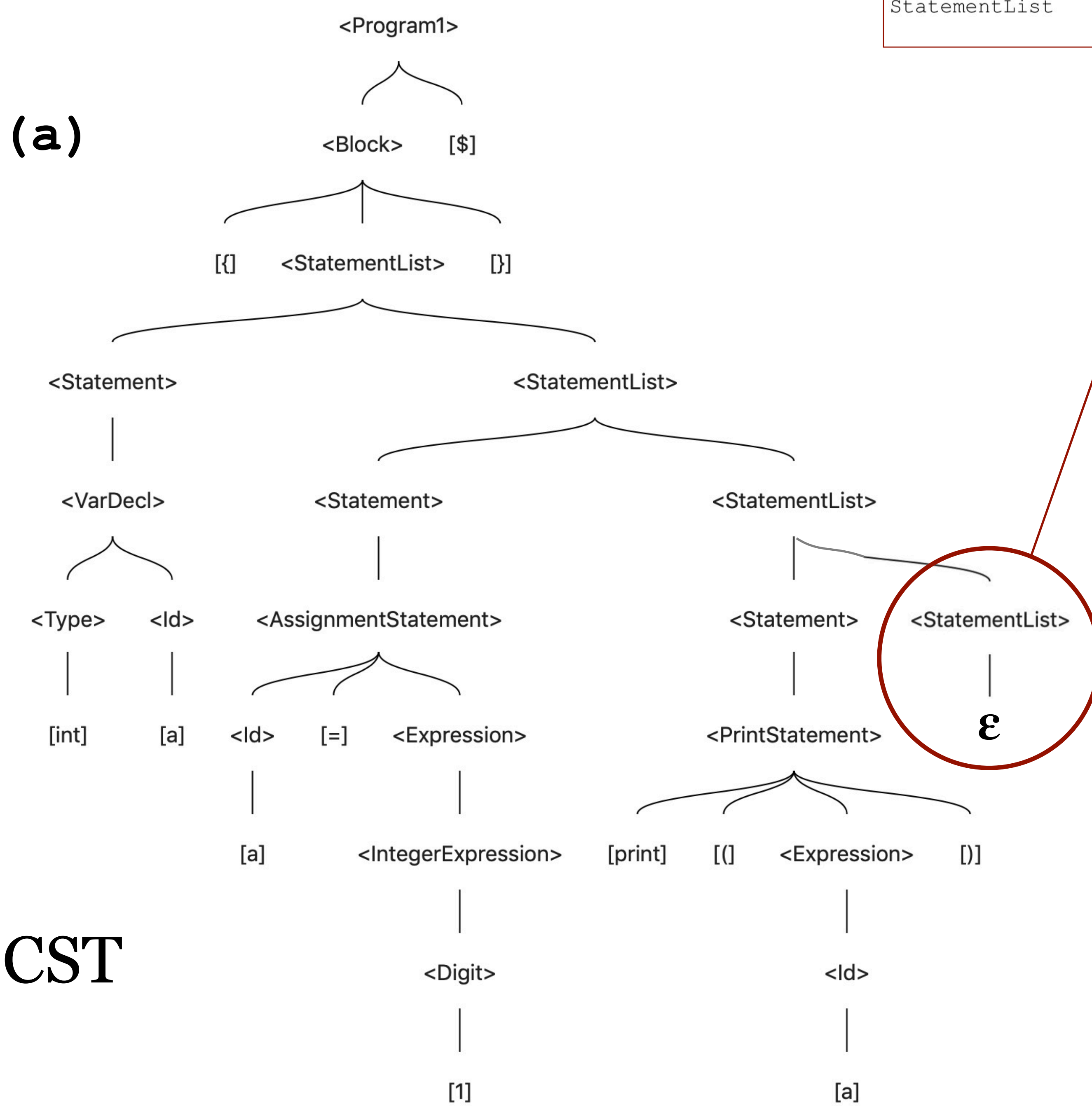
CST for a program in our grammar:

```
{
   int a
   a = 1
   print(a)
}
```

Our Language Grammar

```
Program          ::== Block $
Block            ::== { StatementList  }
StatementList    ::== Statement StatementList
                 ::== ε
```

Curly braces denote scope.



CST

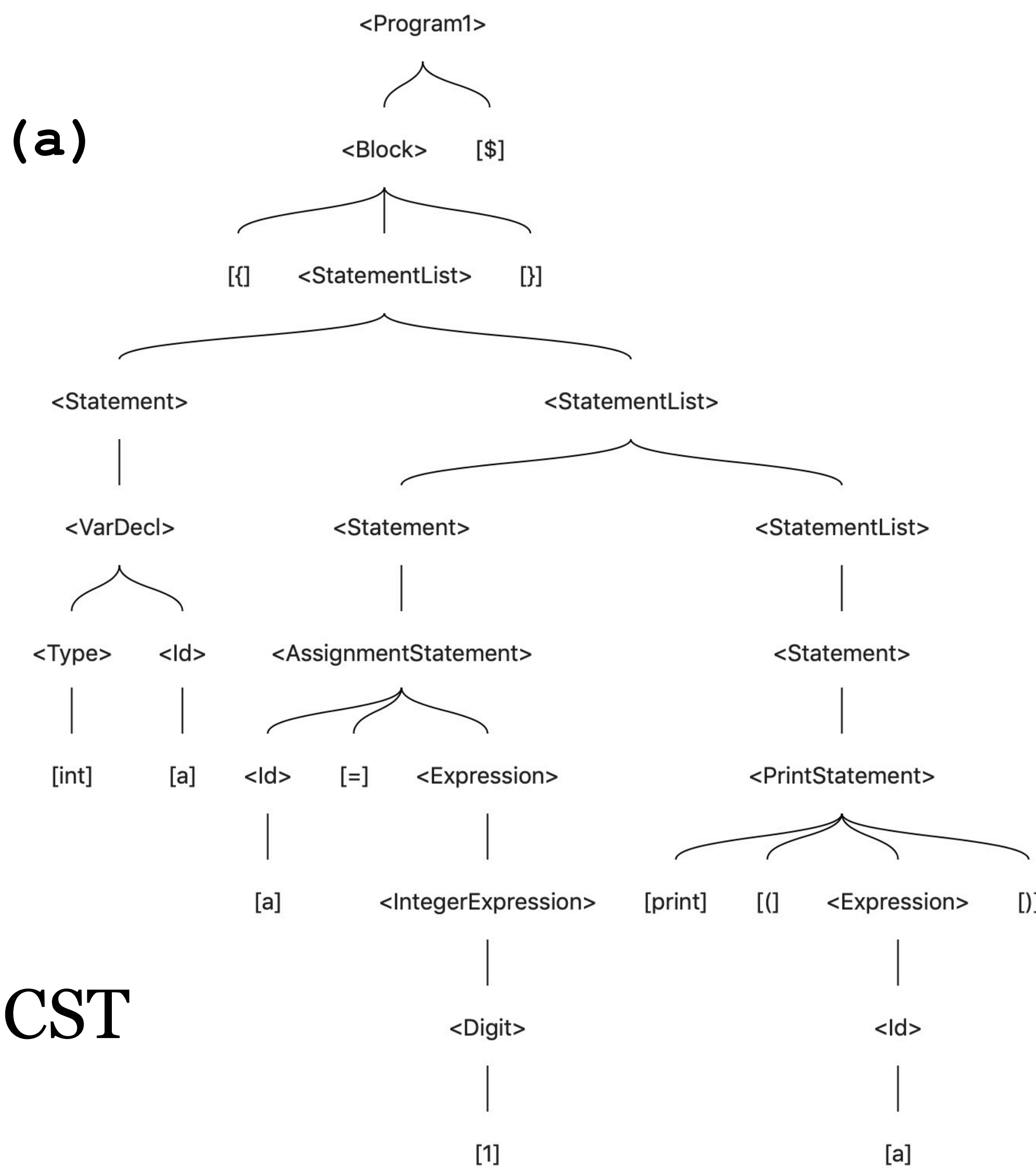
Note the epsilon production of StatementList that stops the recursion.

This does not have to be part of the CST. But it does illustrate how ε productions work, so I kinda like it.

Abstract Syntax Trees

CST for a program in our grammar:

```
{
   int a
   a = 1
   print(a)
}
```



CST

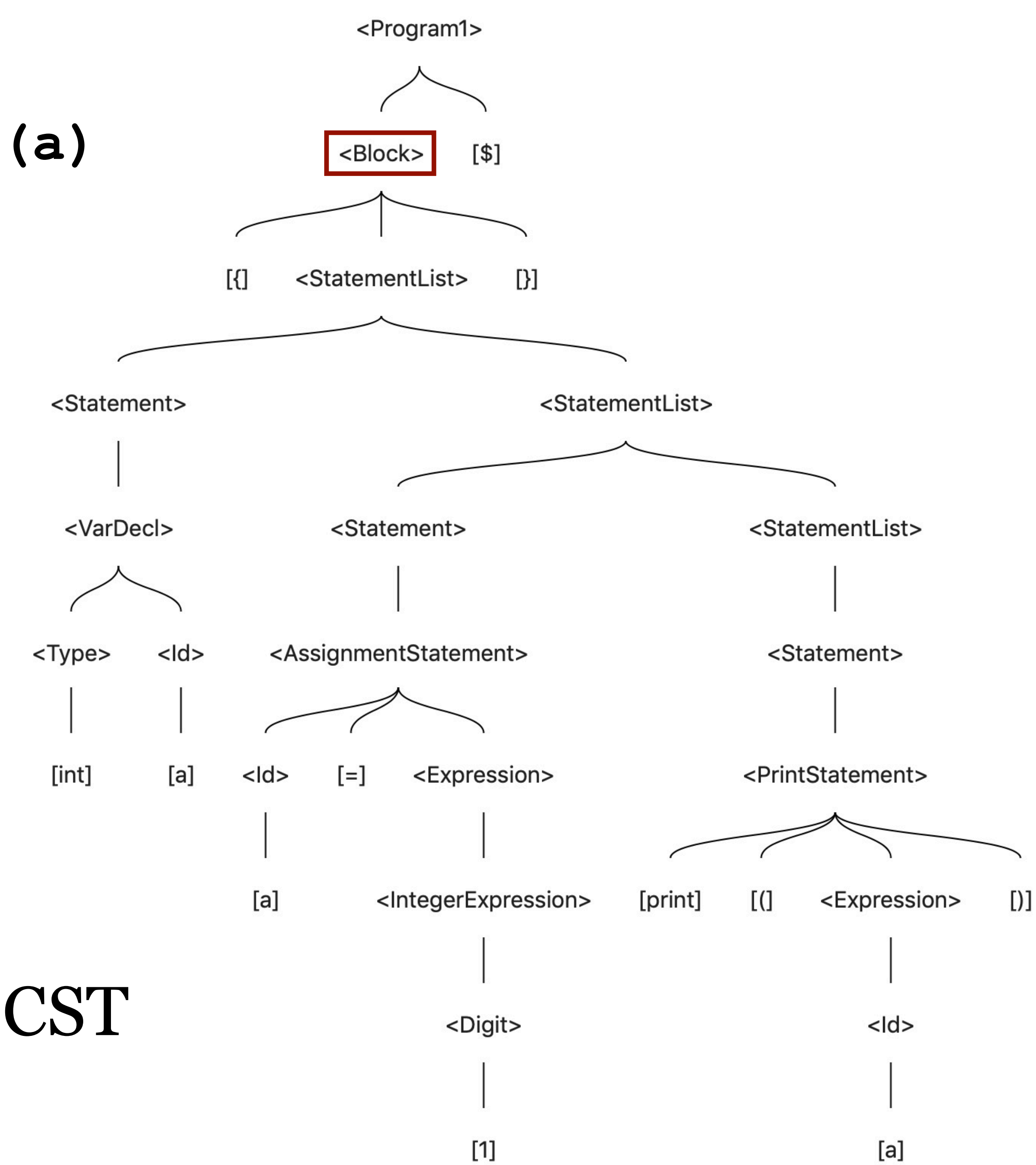
Now that we know everything parses, what are the essential elements of this CST that we need in the AST to express the meaning of the program?

In other words, what are the good parts?

Abstract Syntax Trees

CST for a program in our grammar:

```
{
   int a
   a = 1
   print(a)
}
```



CST

The good parts:

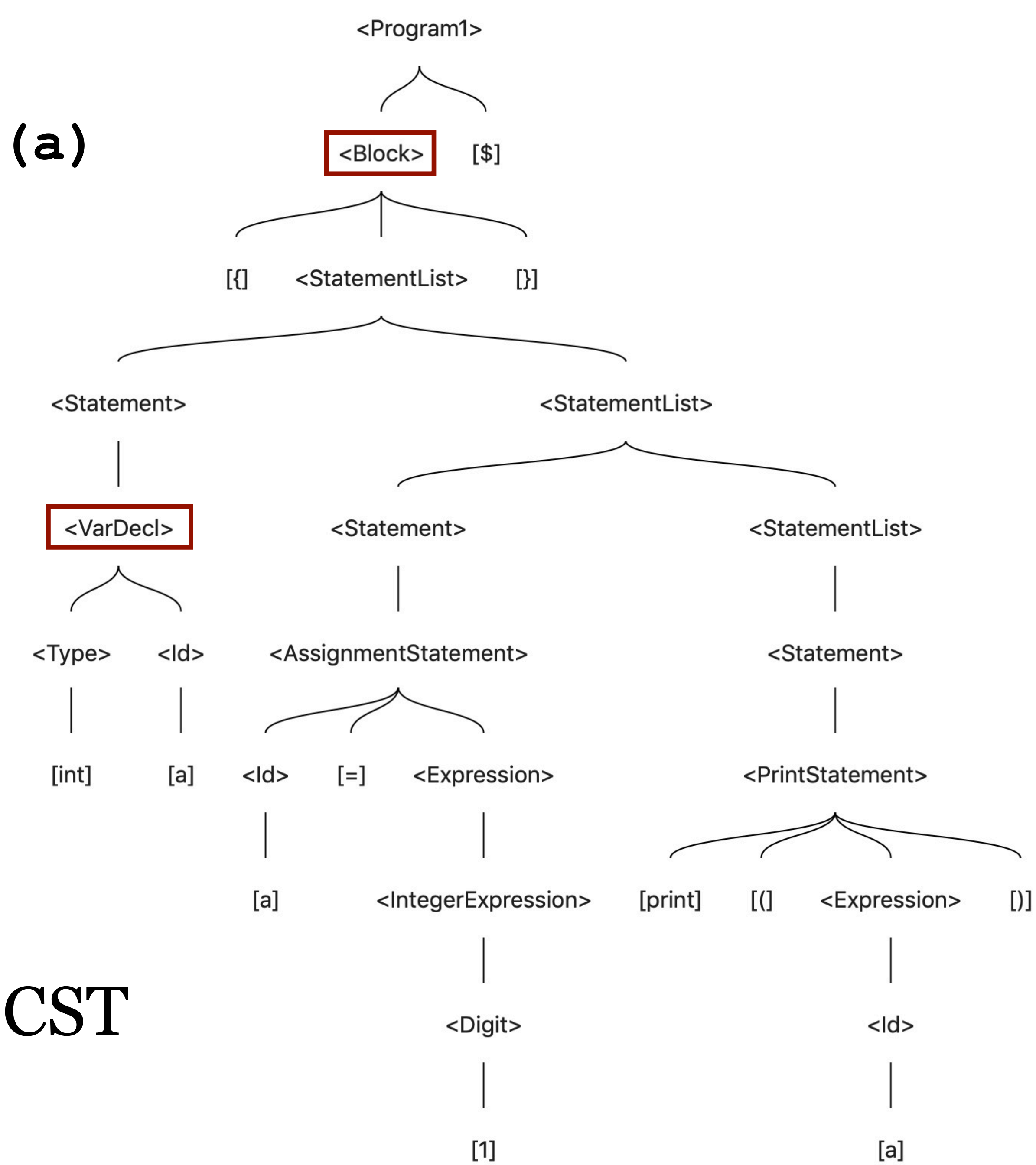
Block -
We definitely want to know about all the blocks.

We don't care about the concrete details of blocks, so we can skip the braces.

Abstract Syntax Trees

CST for a program in our grammar:

```
{
   int a
   a = 1
   print(a)
}
```



CST

The good parts:

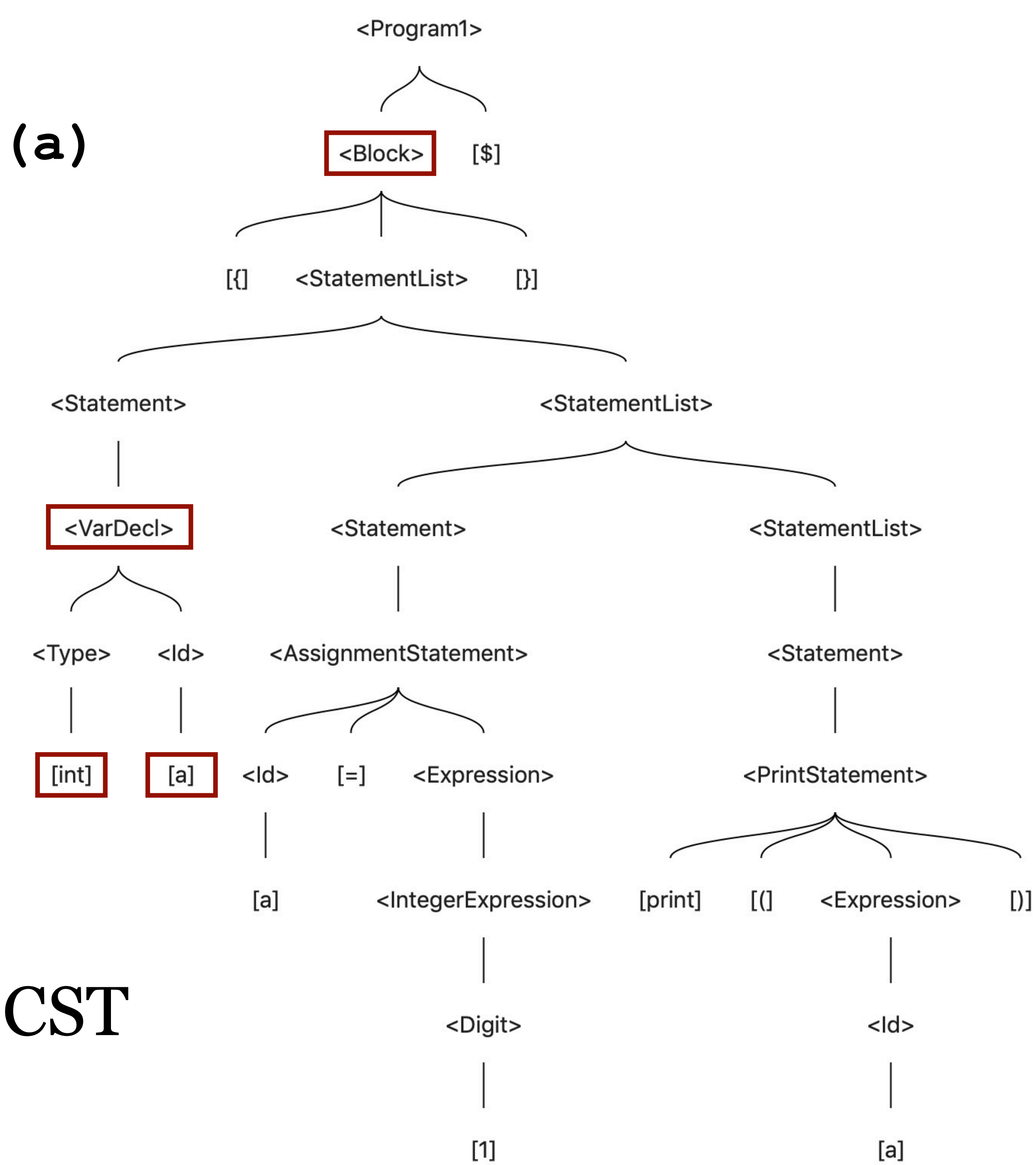
Block
VarDecl -
It's a kind of statement. We definitely want to know about all the statements.

What's essential about variable declarations specifically?

Abstract Syntax Trees

CST for a program in our grammar:

```
{
   int a
   a = 1
   print(a)
}
```



CST

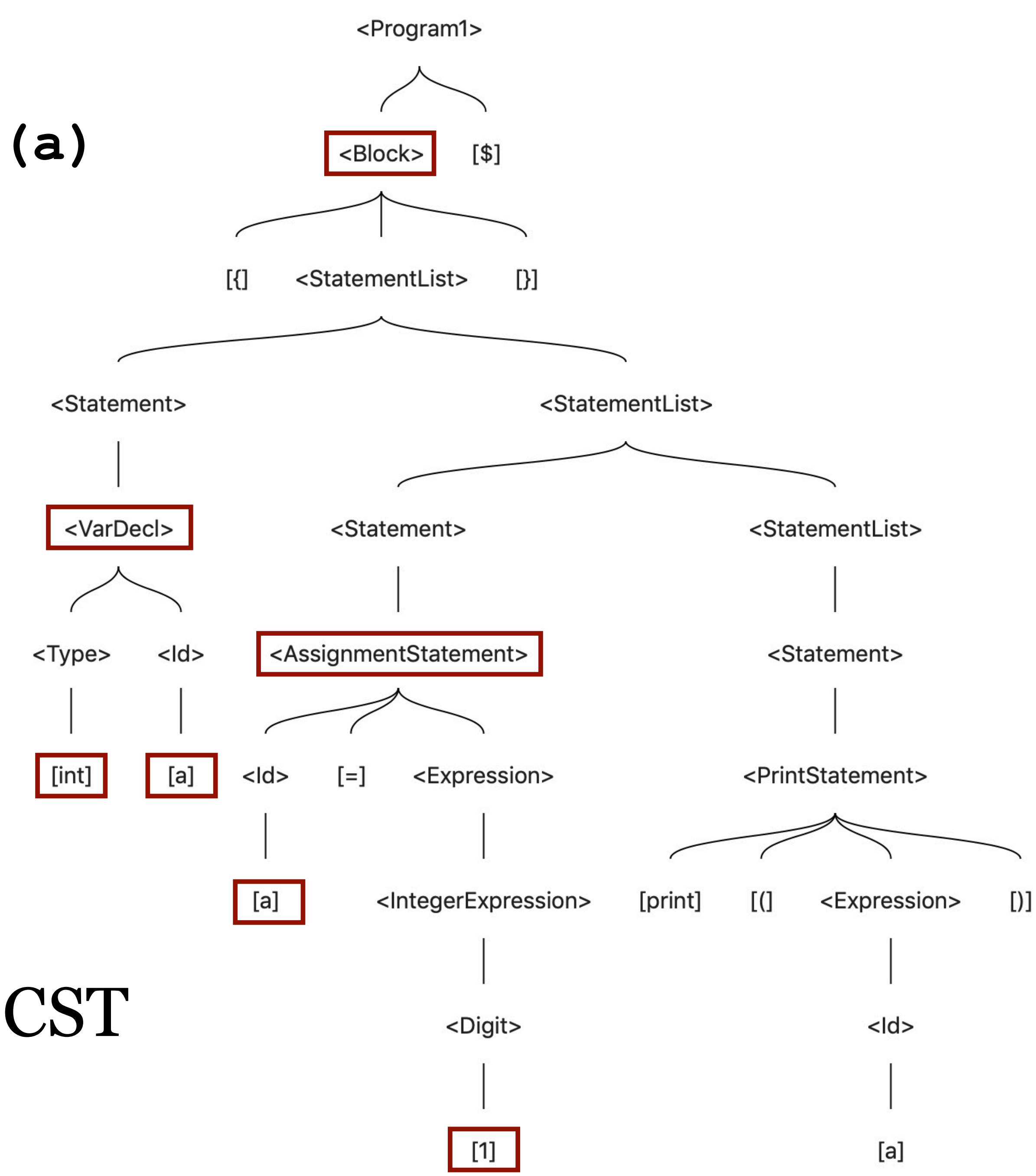
The good parts:

Block
VarDecl -
Type and id are essential about variable declarations. Everything else in the CST about VarDecl are just concrete details of the proof of its conformity to our grammar. We don't need those in an AST.

Abstract Syntax Trees

CST for a program in our grammar:

```
{
   int a
   a = 1
   print(a)
}
```



CST

The good parts:

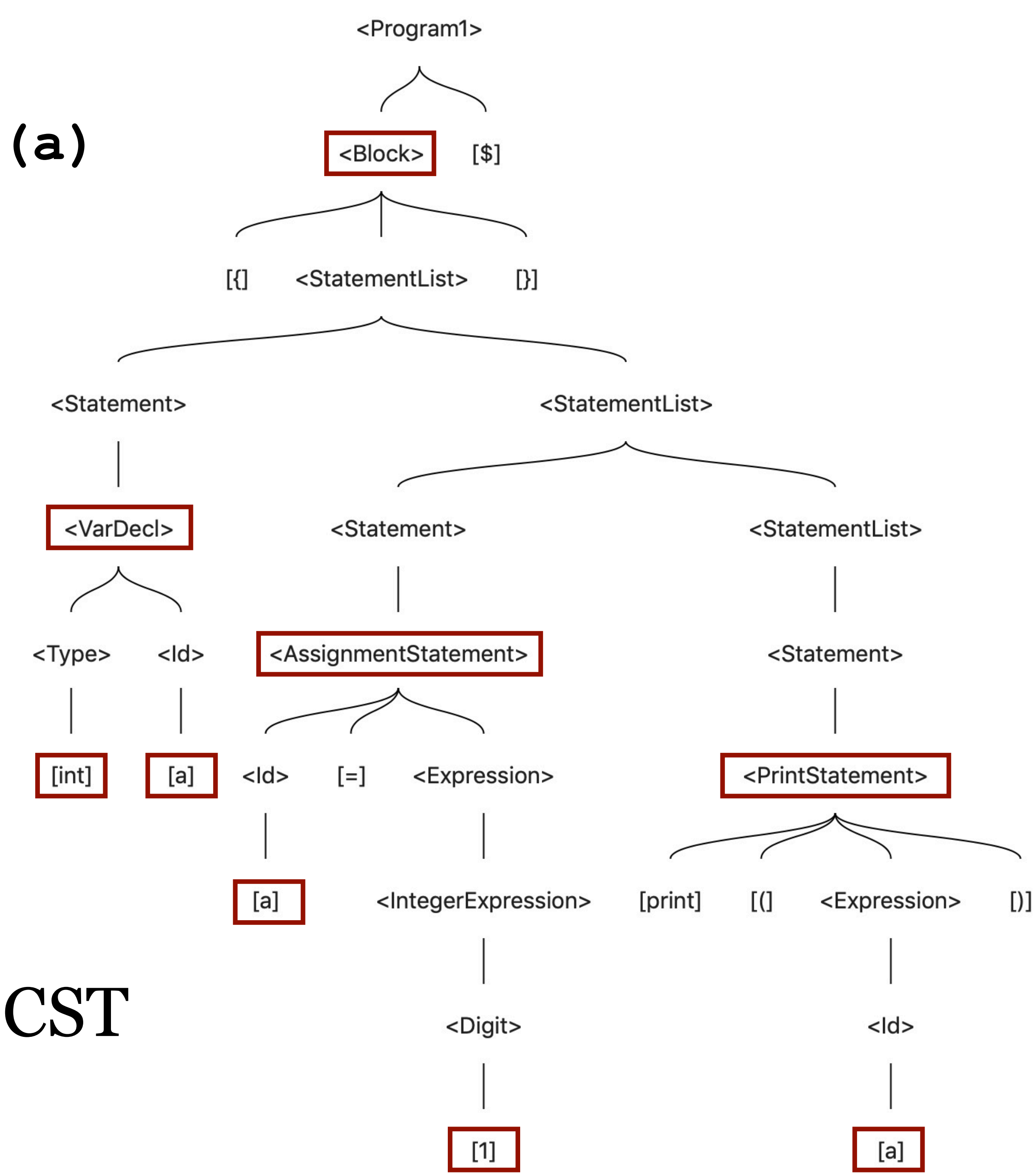
Block
VarDecl
AssignmentStatement - Another kind of statement. Need it.

The id and the result of the expression are essential. Everything else is just proof of syntax, even the "=" symbol.

Abstract Syntax Trees

CST for a program in our grammar:

```
{
    int a
    a = 1
    print(a)
}
```



CST

The good parts:

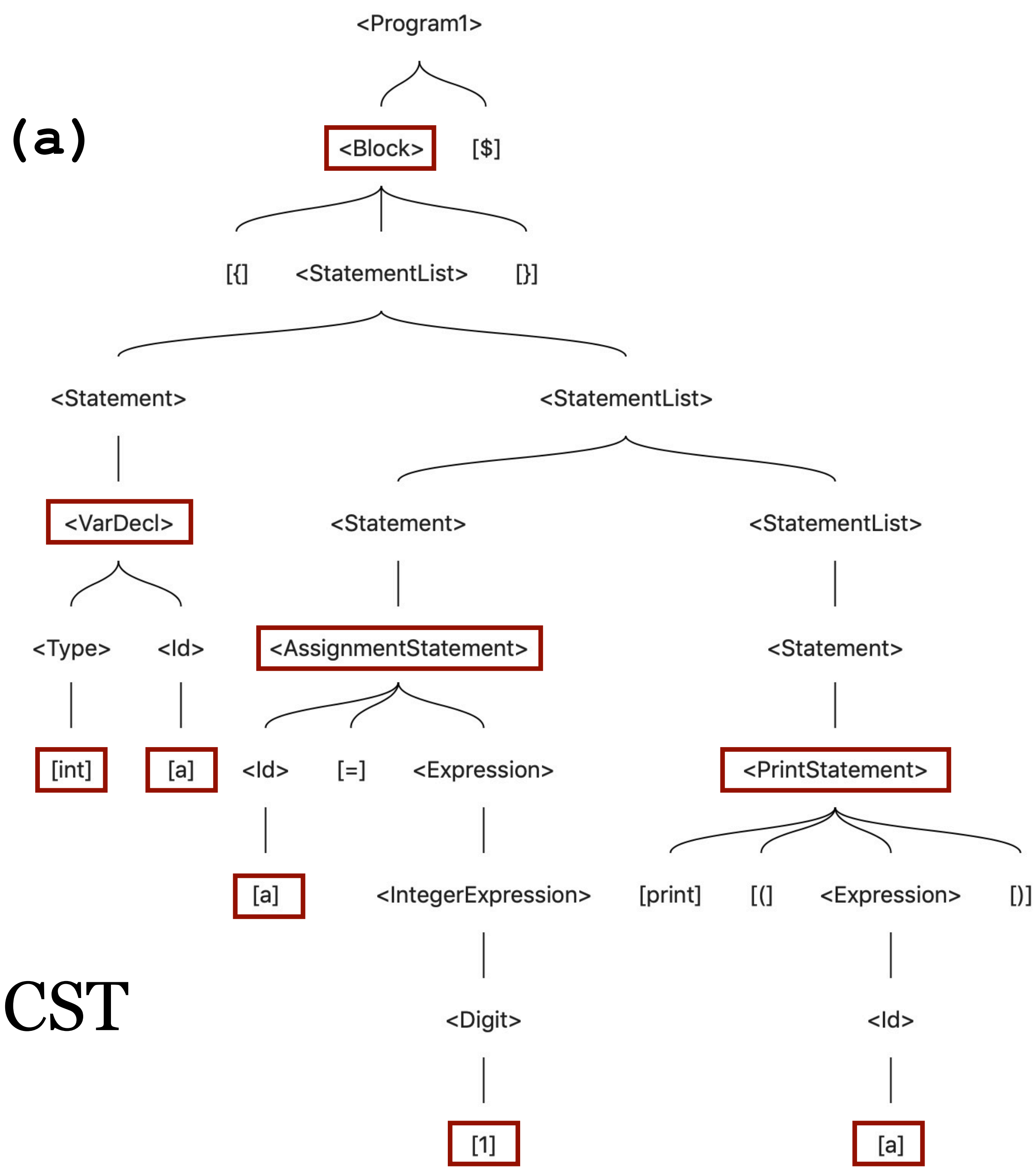
Block
VarDecl
AssignmentStatement
PrintStatement - Another kind of statement. Need it.

The only essential thing here is what we want to print.

Abstract Syntax Trees

CST for a program in our grammar:

```
{
   int a
   a = 1
   print(a)
}
```



CST

The good parts:

Block
VarDecl
AssignmentStatement
PrintStatement

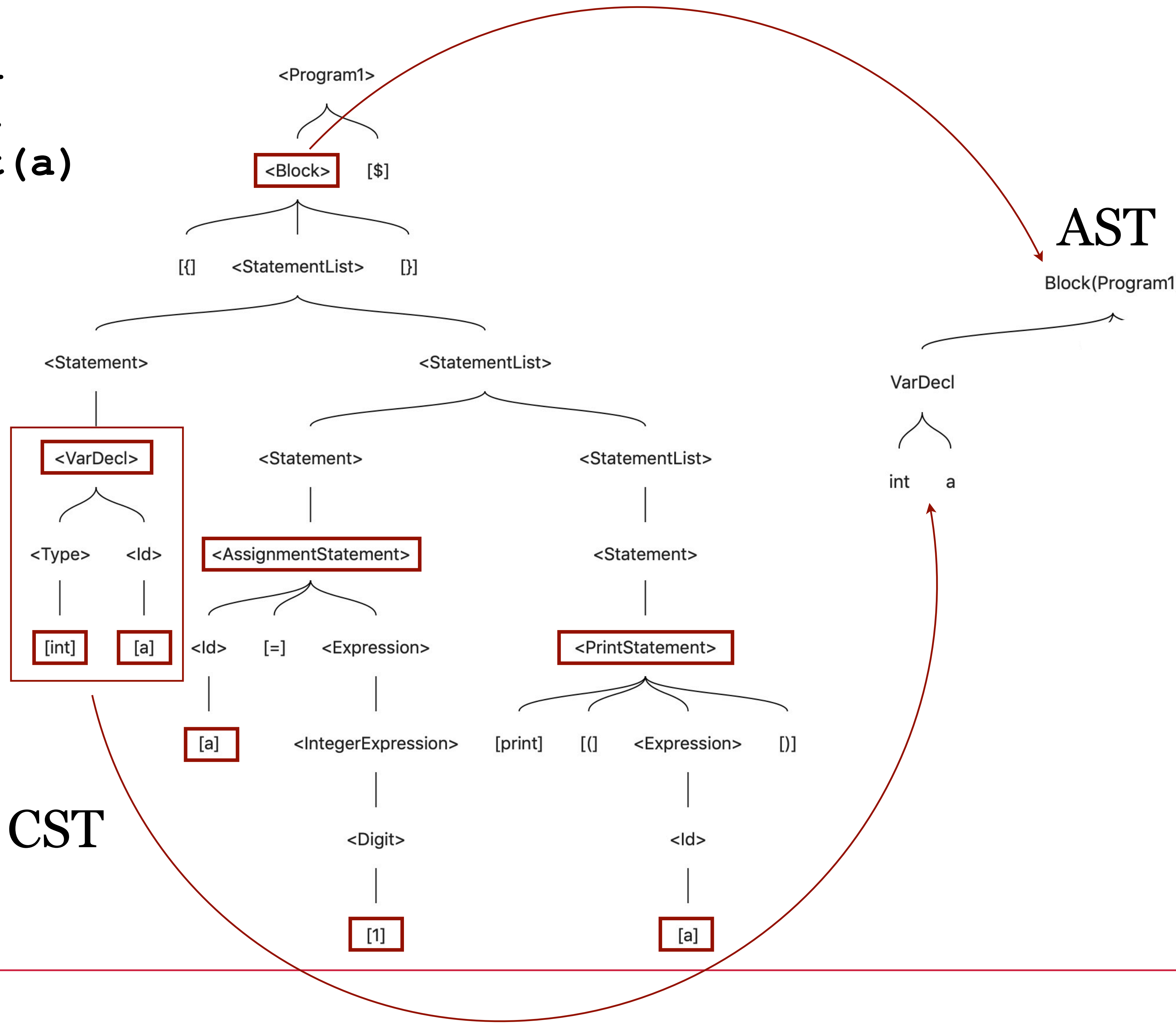
These are all we need in our AST for this program.

Make a new tree of just those things.

Abstract Syntax Trees

Building an AST for a program in our language:

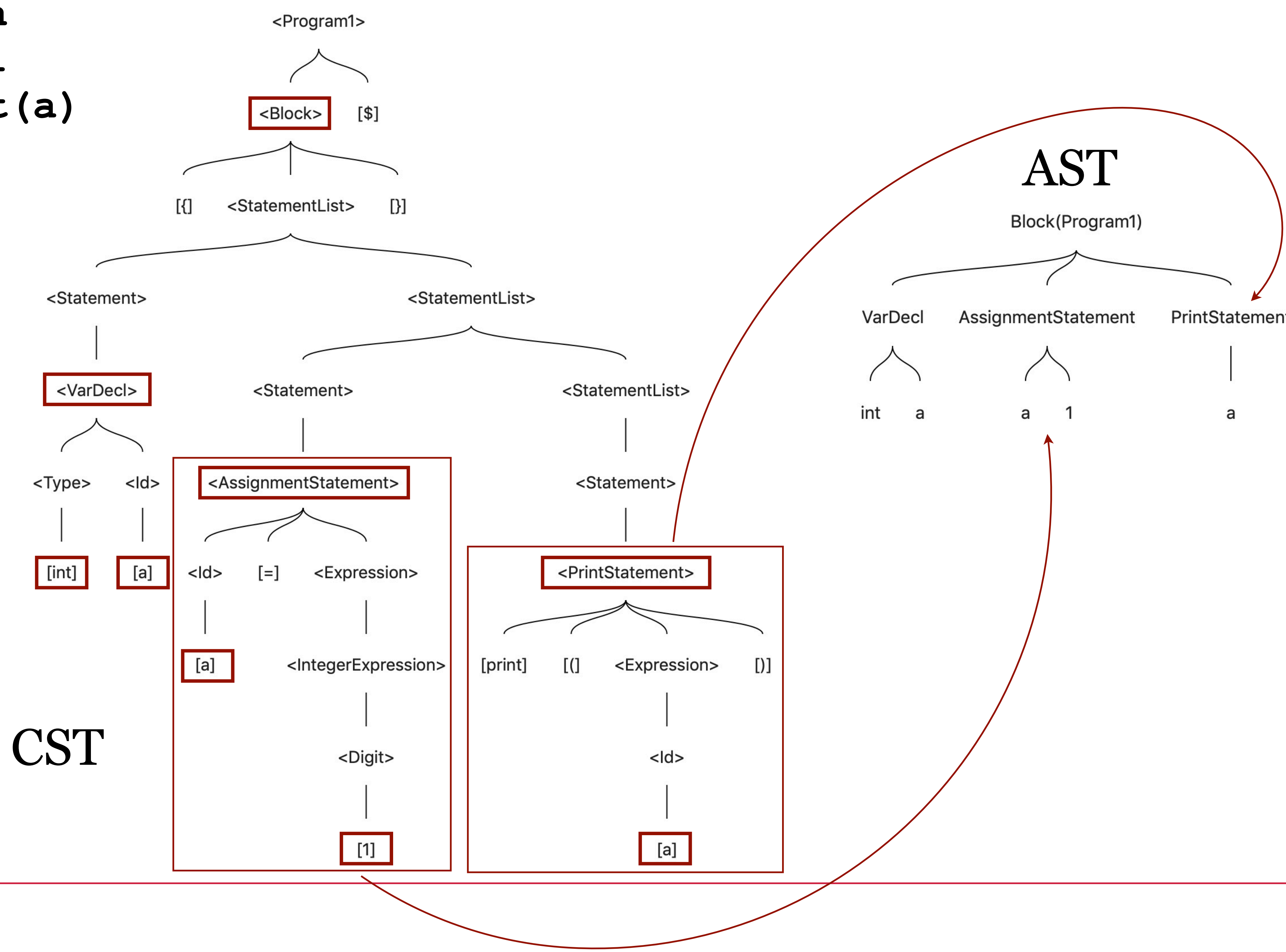
```
{
   int a
   a = 1
   print(a)
}
```



Abstract Syntax Trees

Building an AST for a program in our language:

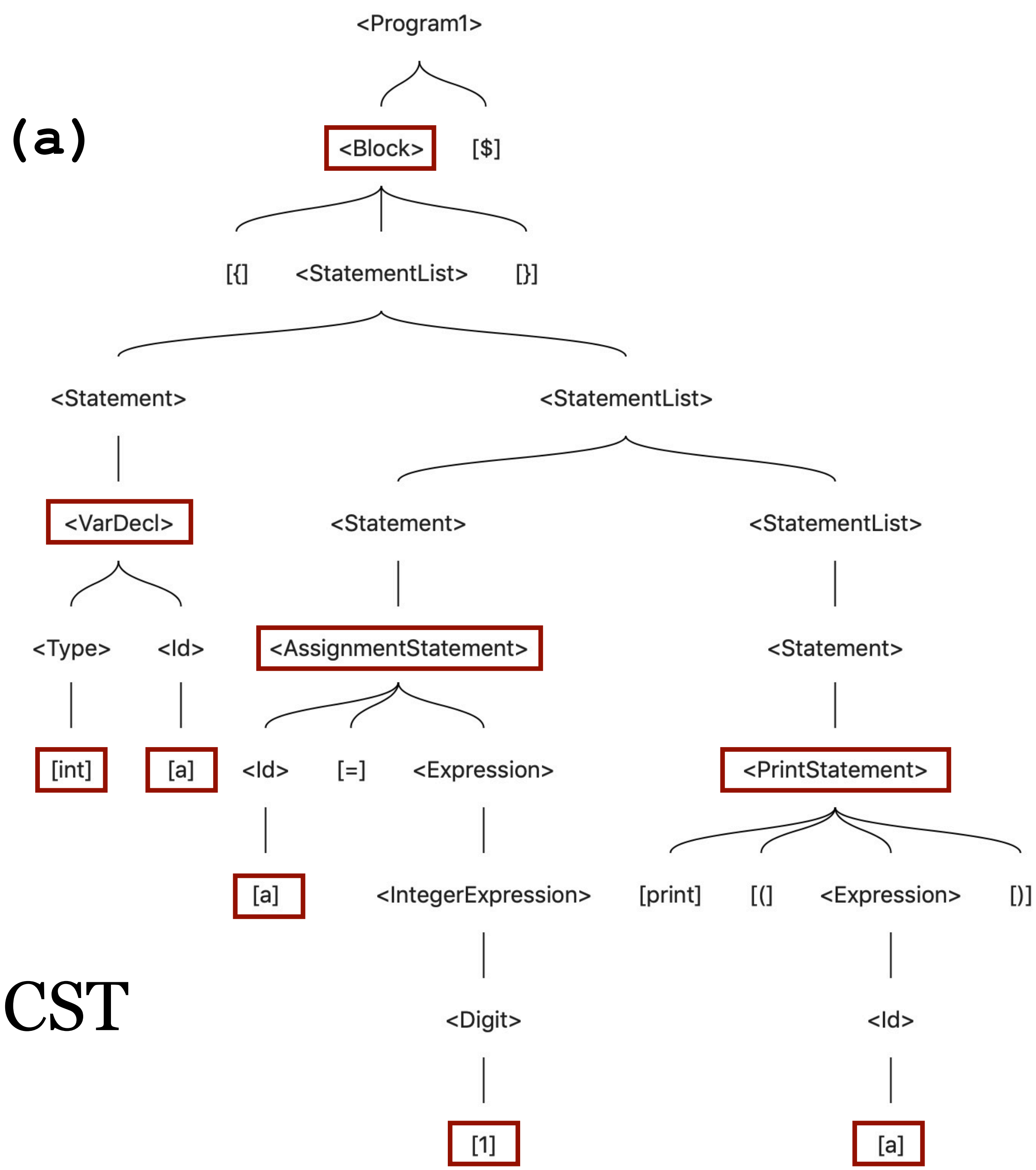
```
{
   int a
   a = 1
   print(a)
}
```



Abstract Syntax Trees

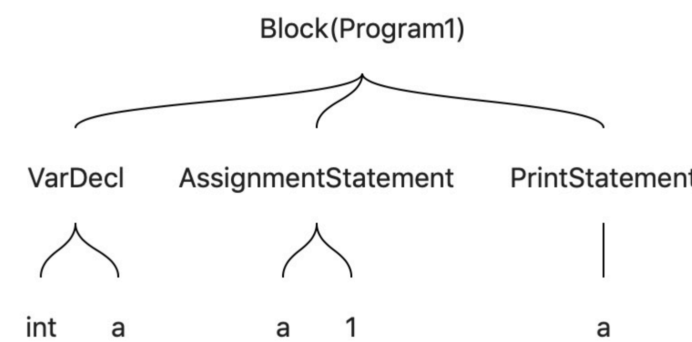
CST and AST for a program in our language:

```
{
   int a
   a = 1
   print(a)
}
```



CST

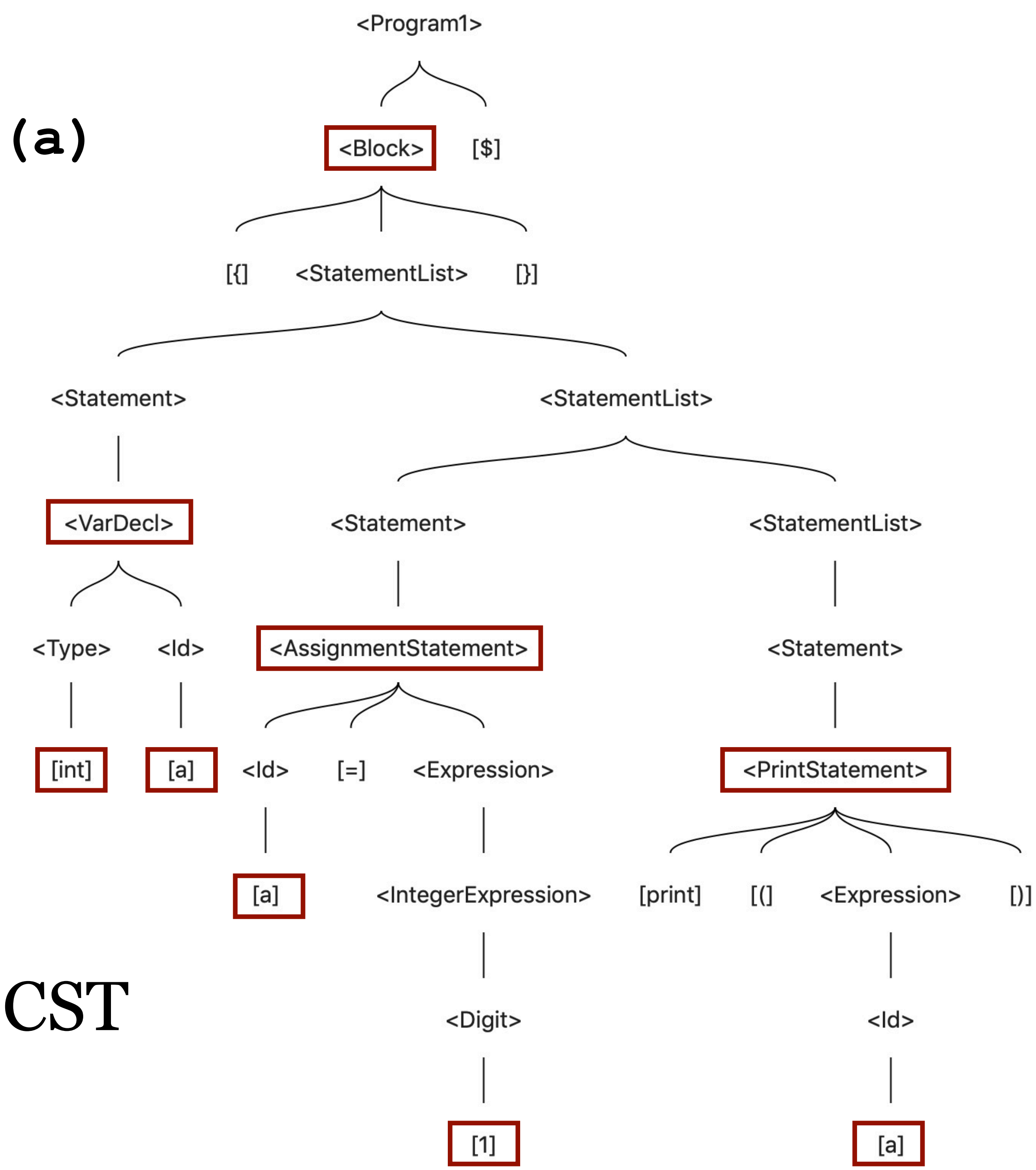
AST



Abstract Syntax Trees

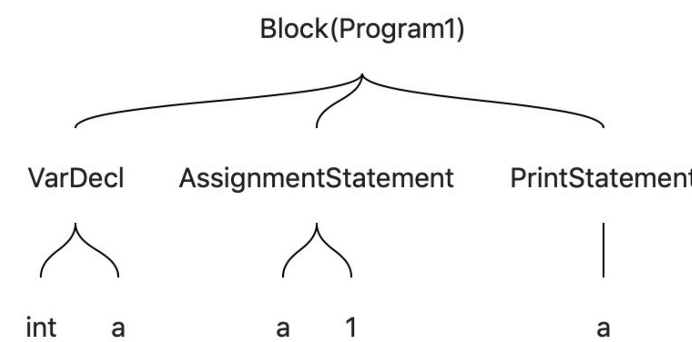
CST and AST for a program in our language:

```
{
   int a
   a = 1
   print(a)
}
```



CST

AST



Q: How do we identify the good parts?

Semantic Analysis

Turning a CST into an AST

Q: How do we identify the good parts?
A: It depends on the grammar and the rules of the language. For this reason, there is no general procedure for turning a CST into an AST.

We have to analyze our grammar and the rules for type and scope in our language and decide for ourselves what elements of the CST are necessary elements of the AST. We do this by identifying key elements of the grammar and developing AST subtree patterns for them.

We've done this for **block** and three more elements already:

- VarDecl
- AssignmentStatement
- PrintStatement

Semantic Analysis

Turning a CST into an AST

Our AST Subtree Patterns so far:

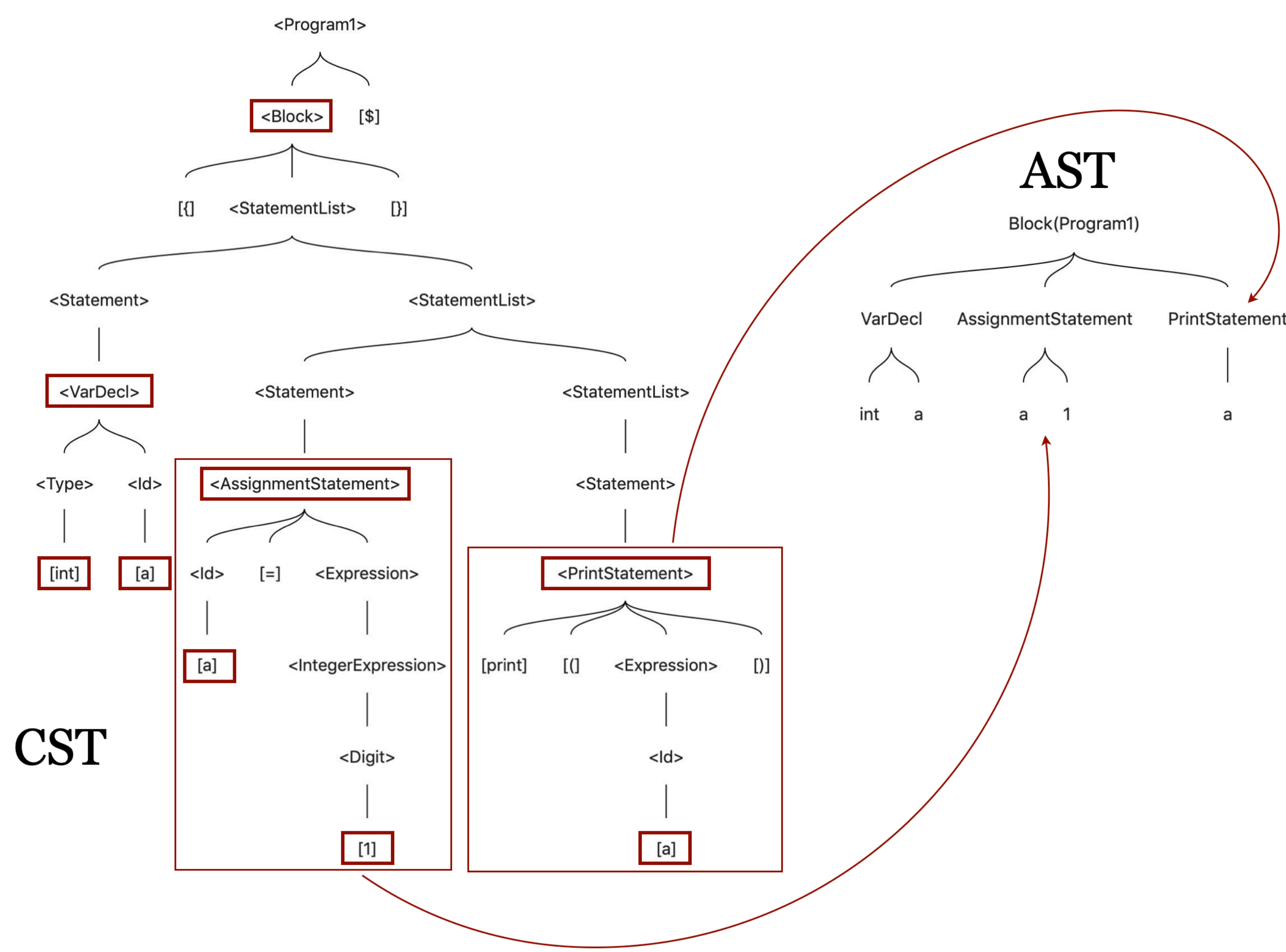
- **VarDecl** - Construct a subtree in the AST rooted with VarDecl and add two children: the type and the id.
- **AssignmentStatement** - Construct a subtree in the AST rooted with AssignmentStatement and add two children: the id and the result of the expression being assigned.
- **PrintStatement** - Construct a subtree in the AST rooted with PrintStatement and add one child: the result of the expression being printed.

Semantic Analysis

Turning a CST into an AST

Recipe:

(Option 1) Traverse the CST (depth-first, in-order) looking for key elements, building the AST as you go according to the AST subtree patterns.



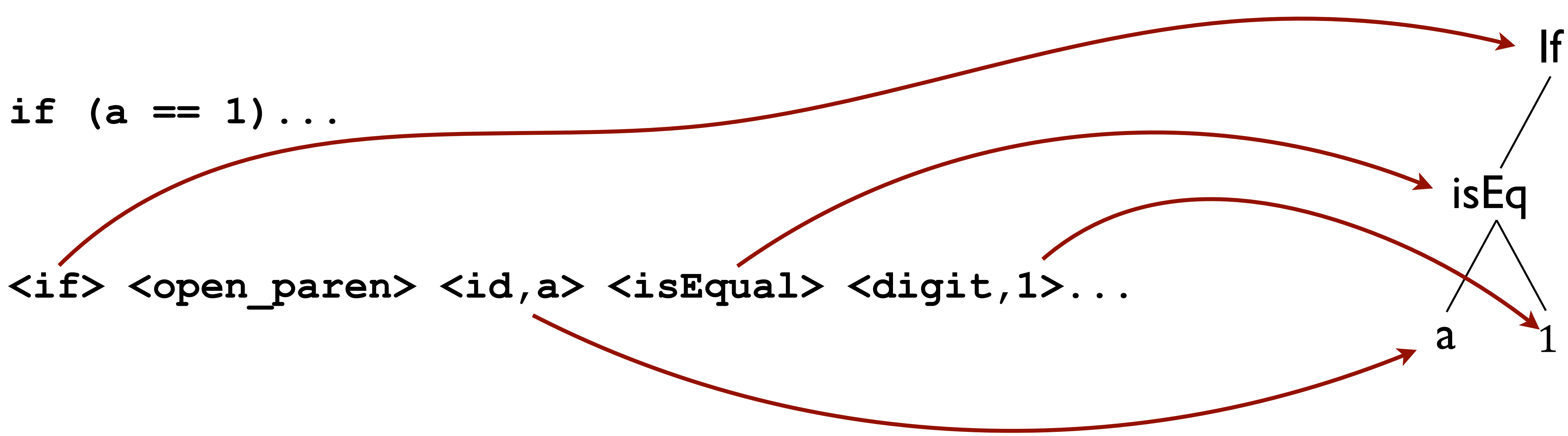
Semantic Analysis

Turning a CST into an AST

Recipe:

(Option 2) Re-“parse” the token stream, looking for key elements, building the AST as you go according to the AST subtree patterns.

- This is not a real parse, because you already know that everything is syntactically correct and where it’s supposed to be. It’s just another pass over the tokens to build the AST, which **may** be easier than traversing the CST.
- You will need to address the tokens out of order from time to time, but you know that everything you need will be where you expect it to be, because if that were not the case we would not have made it out of parse.



Semantic Analysis

Turning a CST into an AST

Our AST Subtree Patterns so far:

- **Block** - ?
- **VarDecl** - Construct a subtree in the AST rooted with VarDecl and add two children: the type and the id.
- **AssignmentStatement** - Construct a subtree in the AST rooted with AssignmentStatement and add two children: the id and the result of the expression being assigned.
- **PrintStatement** - Construct a subtree in the AST rooted with PrintStatement and add one child: the result of the expression being printed.
- **WhileStatement** - ?
- **IfStatement** - ?
- Others? Yes, several.

Our Language Grammar

```
Program              ::== Block $
Block                ::== { StatementList }
StatementList        ::== Statement StatementList
                     ::== ε
Statement            ::== PrintStatement
                     ::== AssignmentStatement
                     ::== VarDecl
                     ::== WhileStatement
                     ::== IfStatement
                     ::== Block
PrintStatement       ::== print ( Expr )
AssignmentStatement  ::== Id = Expr
VarDecl              ::== type Id
WhileStatement       ::== while BooleanExpr Block
IfStatement          ::== if BooleanExpr Block

Expr                 ::== IntExpr
                     ::== StringExpr
                     ::== BooleanExpr
                     ::== Id
IntExpr              ::== digit intop Expr
                     ::== digit
StringExpr           ::== " CharList "
BooleanExpr          ::== ( Expr boolop Expr )
                     ::== boolval
Id                   ::== char
CharList             ::== char CharList
                     ::== space CharList
                     ::== ε

type                 ::== int | string | boolean
char                 ::== a | b | c ... z
space                ::== the space character
digit                ::== 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

boolop               ::== == | !=
boolval              ::== false | true
intop                ::== +

Comments are bounded by /* and */ and ignored by the lexer.
```

Curly braces denote scope.

= is assignment.

== is test for equality.

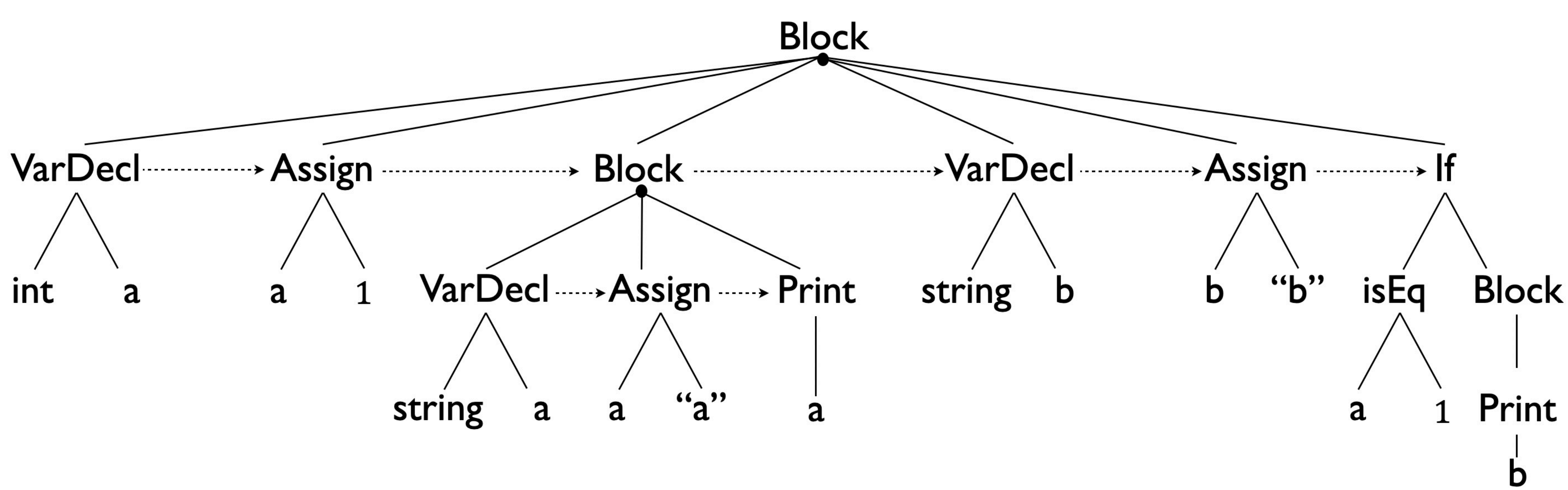
Semantic Analysis

Another example in our language

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

What other “good parts” do you see in this AST?

AST

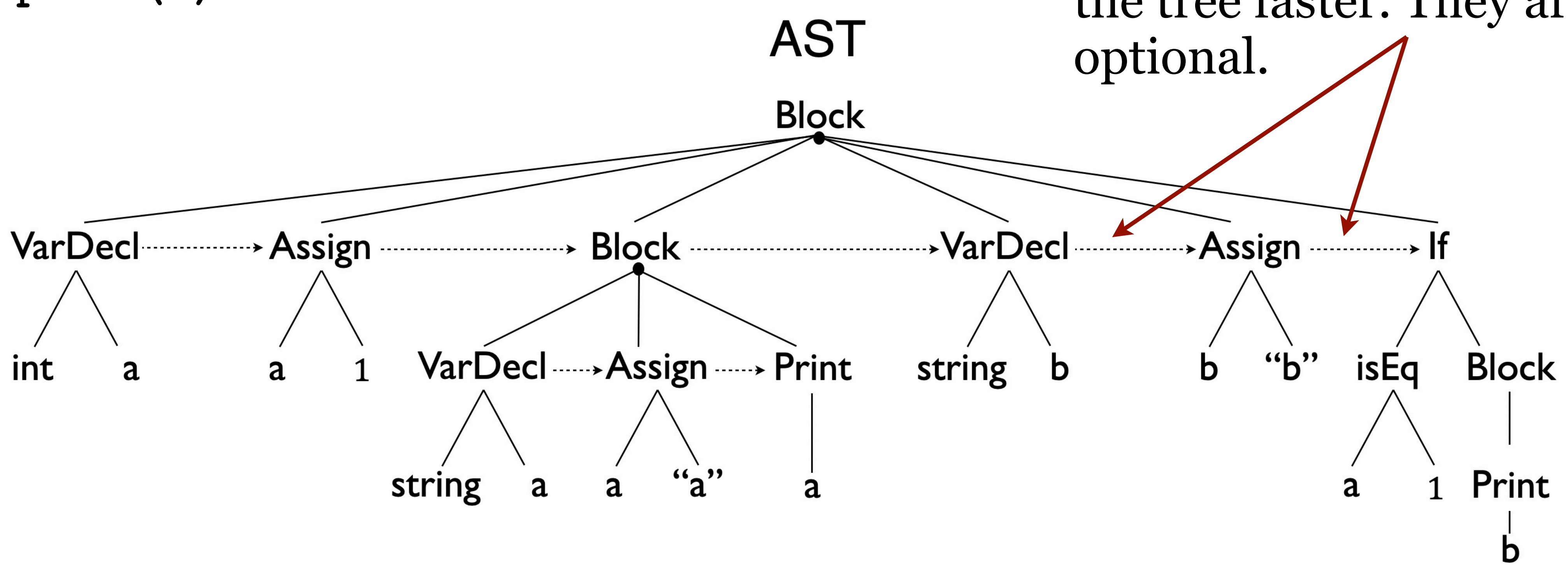


Semantic Analysis

Another example in our language

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

The dotted pointers connect sequential statements in a block. This can help the back end of the compiler traverse the tree faster. They are optional.



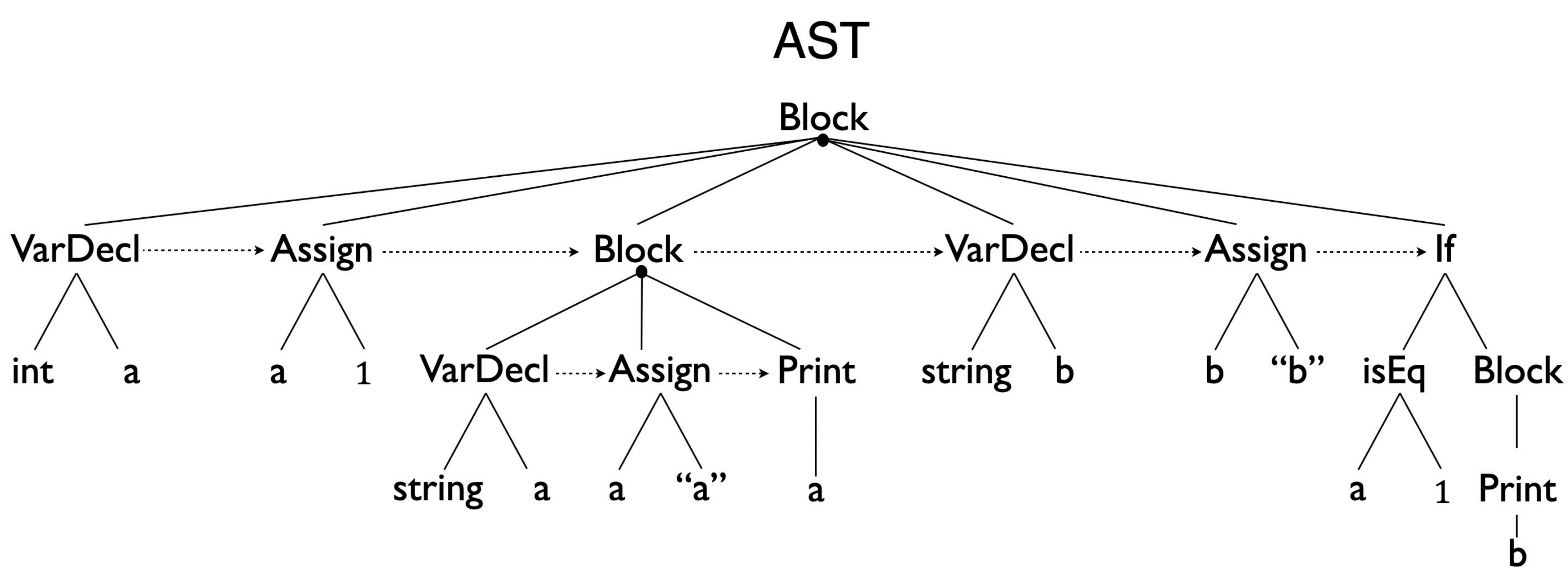
Semantic Analysis

We are ready to check scope and type.

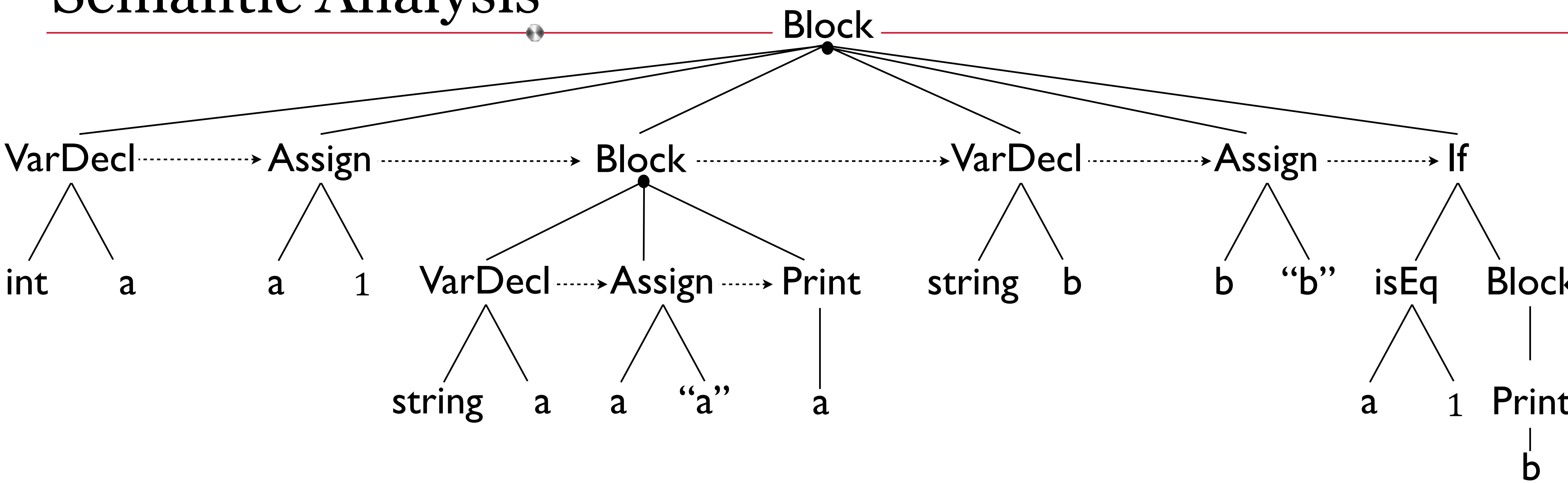
A depth-first, in-order AST traversal will allow us to …

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

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



Semantic Analysis



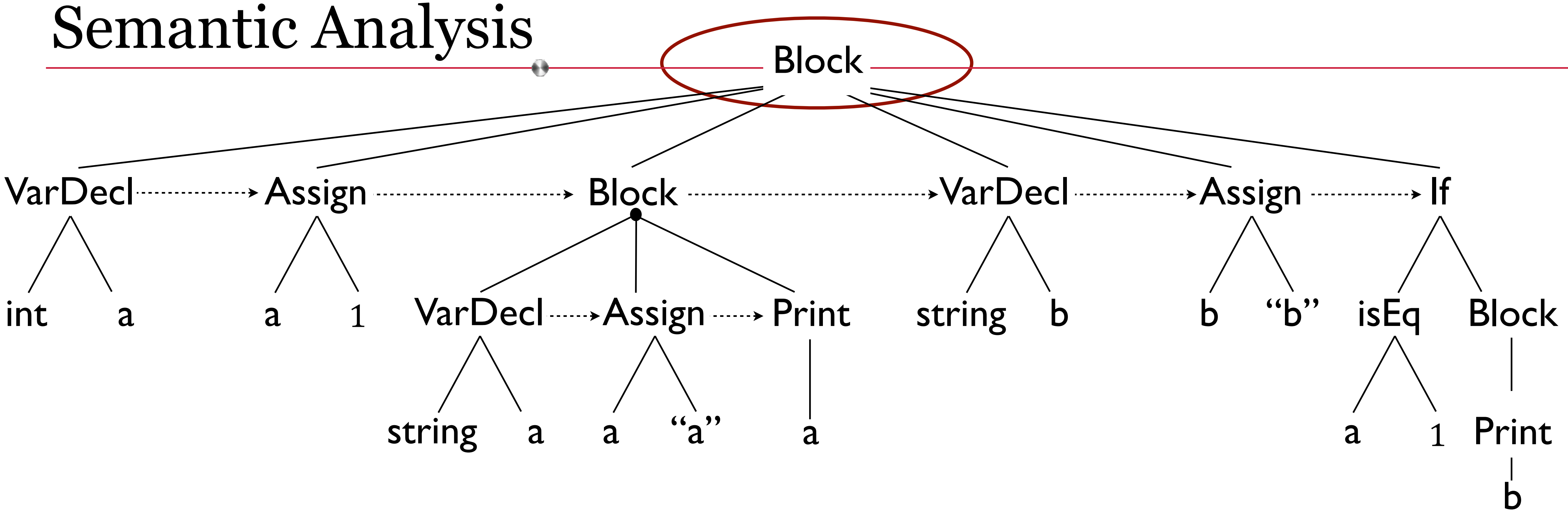
Source Code

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

Symbol Table

Semantic Analysis

AST

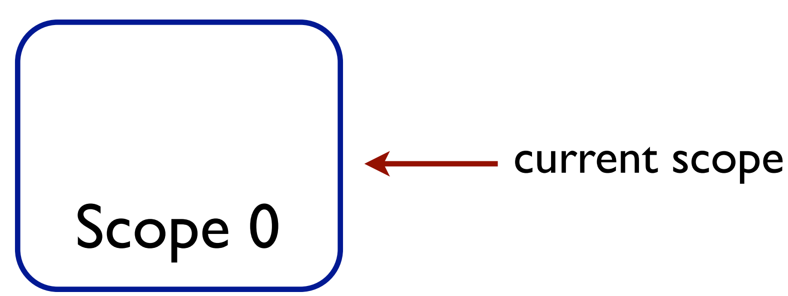


Initialize Scope 0

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

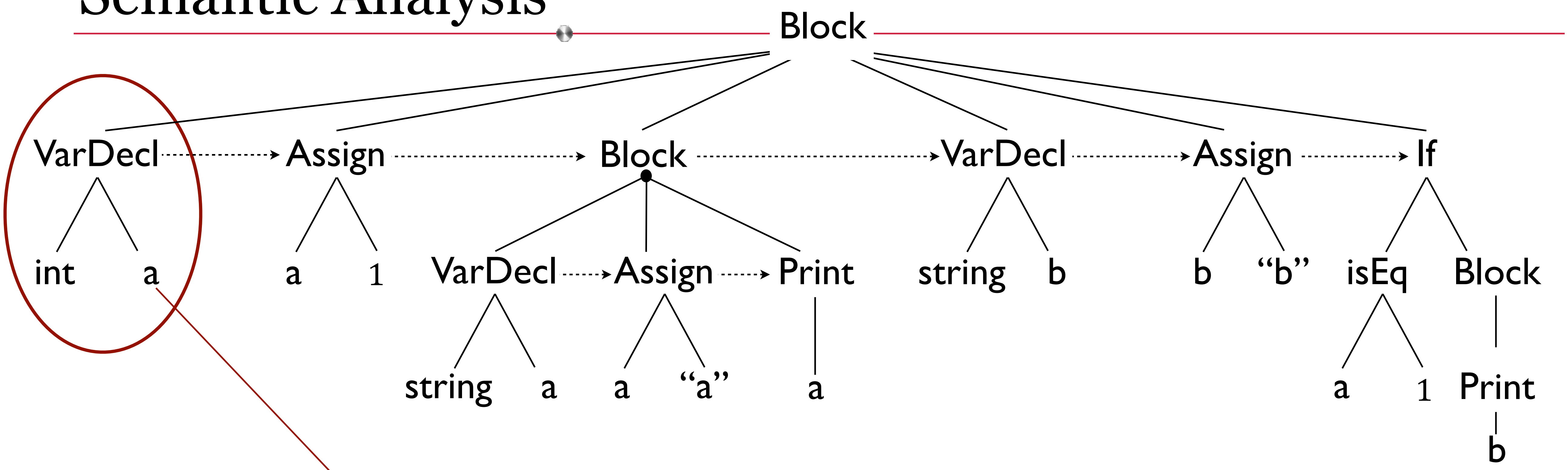
Source Code

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



Symbol Table

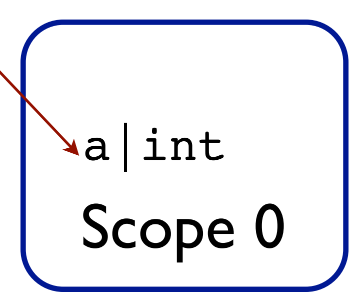
Semantic Analysis



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

Source Code

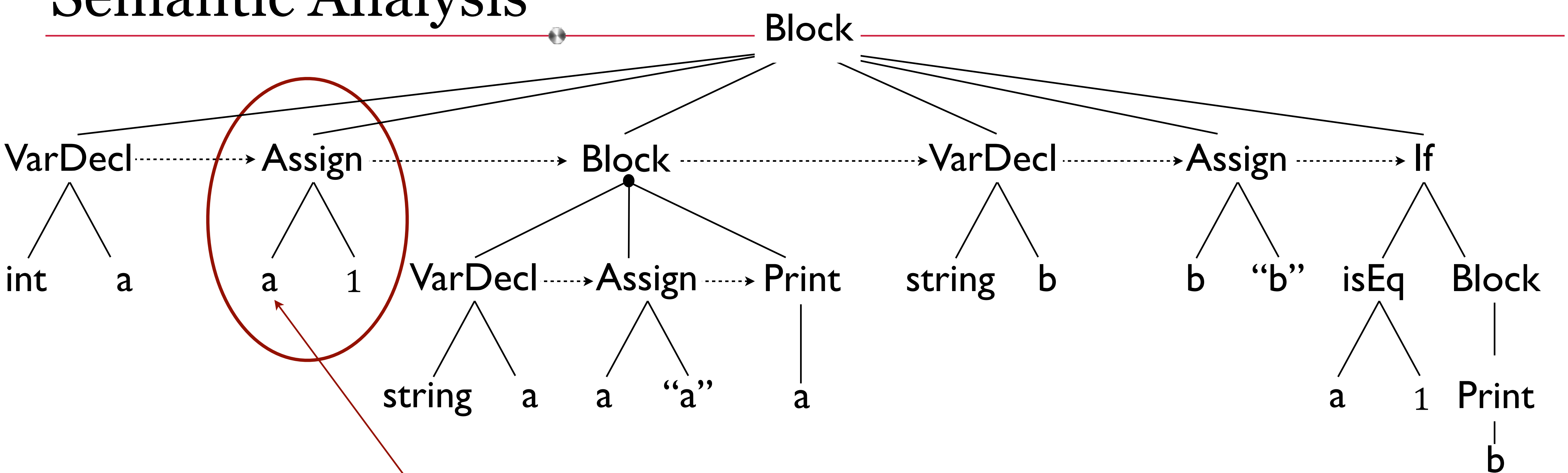
```
{
  int a
  a = 1
  {
     string a
     a = "a"
     print(a)
  }
  string b
  b = "b"
  if (a == 1) {
     print(b)
  }
}
```



current scope

Symbol Table

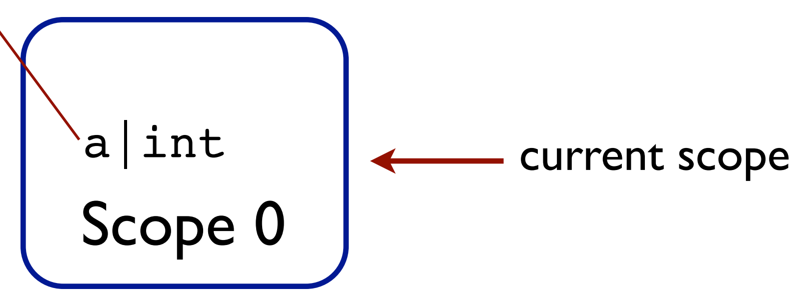
Semantic Analysis



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

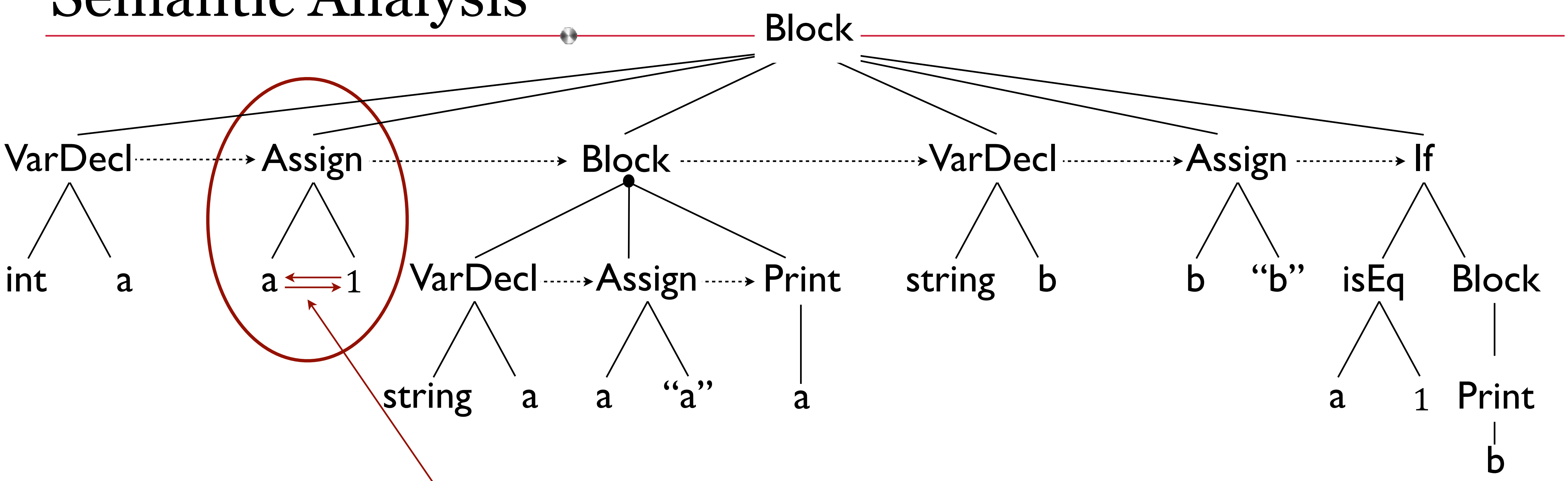
Source Code

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



Symbol Table

Semantic Analysis

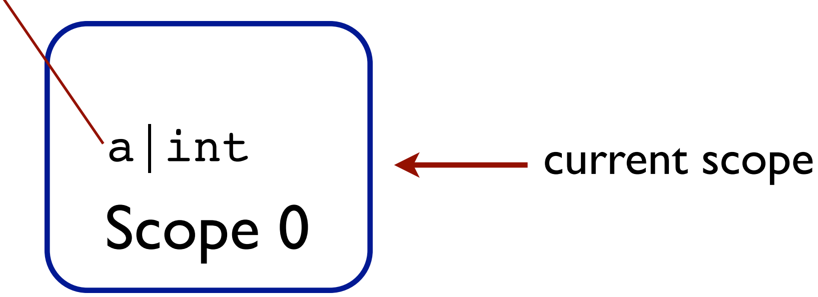


Initialize Scope 0
add symbol a
lookup symbol a
check types

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

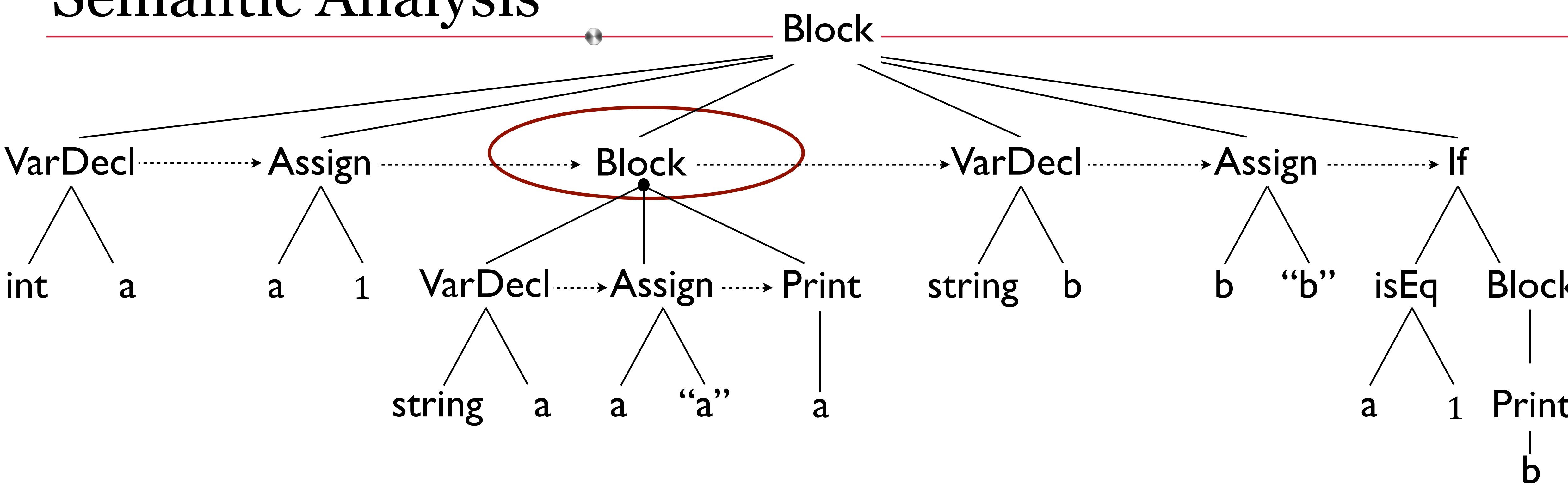
Source Code

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



Symbol Table

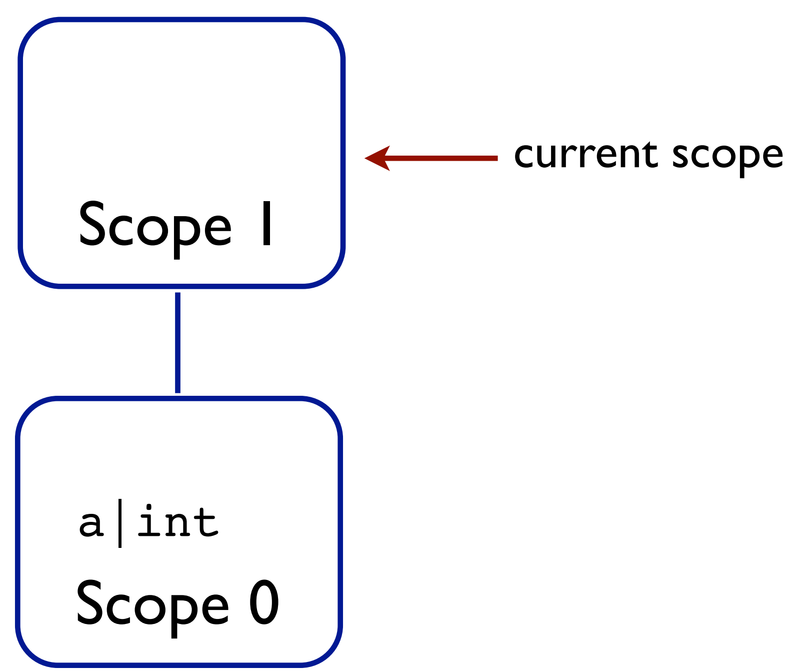
Semantic Analysis



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

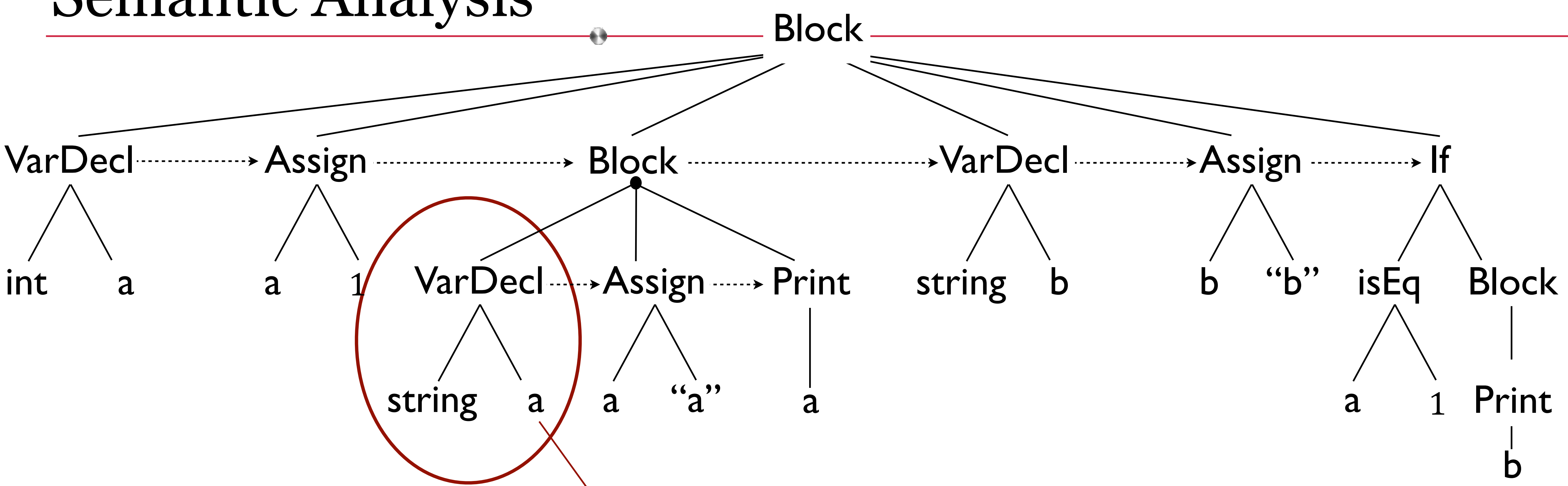
Source Code

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



Symbol Table

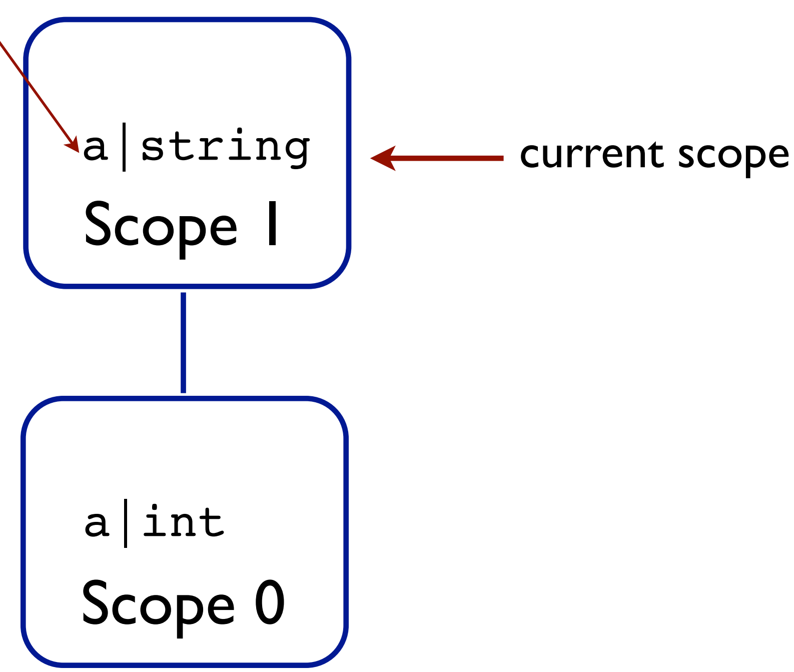
Semantic Analysis



Source Code

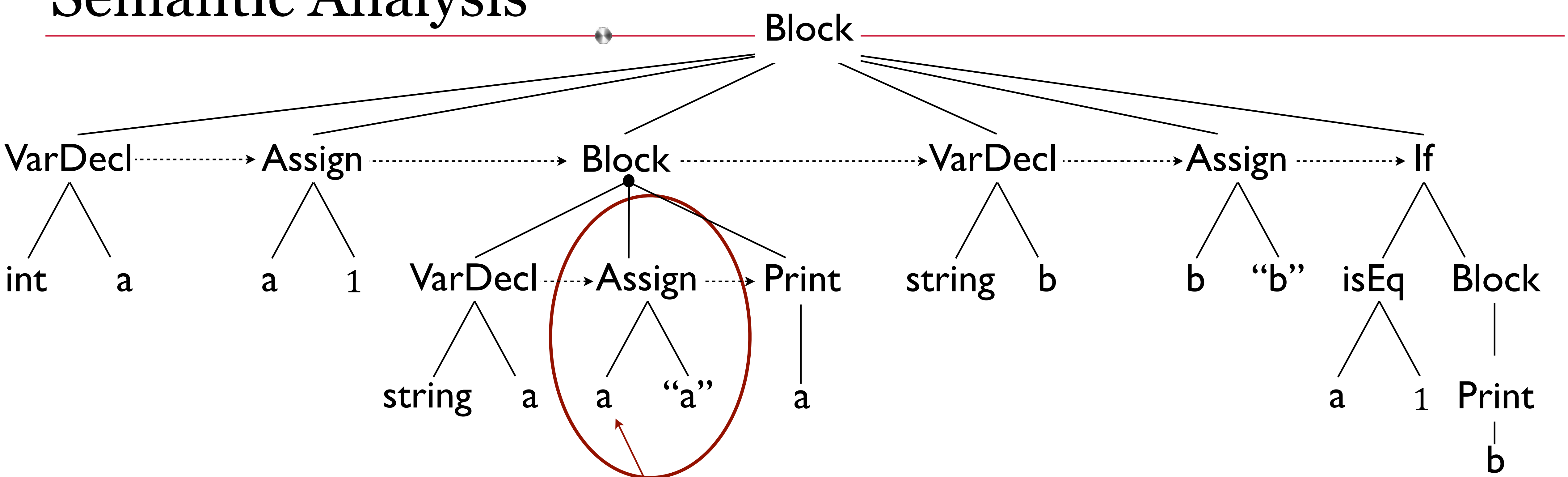
```
{
    int a
    a = 1
    {
      string a
      a = "a"
      print(a)
    }
    string b
    b = "b"
    if (a == 1) {
      print(b)
    }
}
```

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



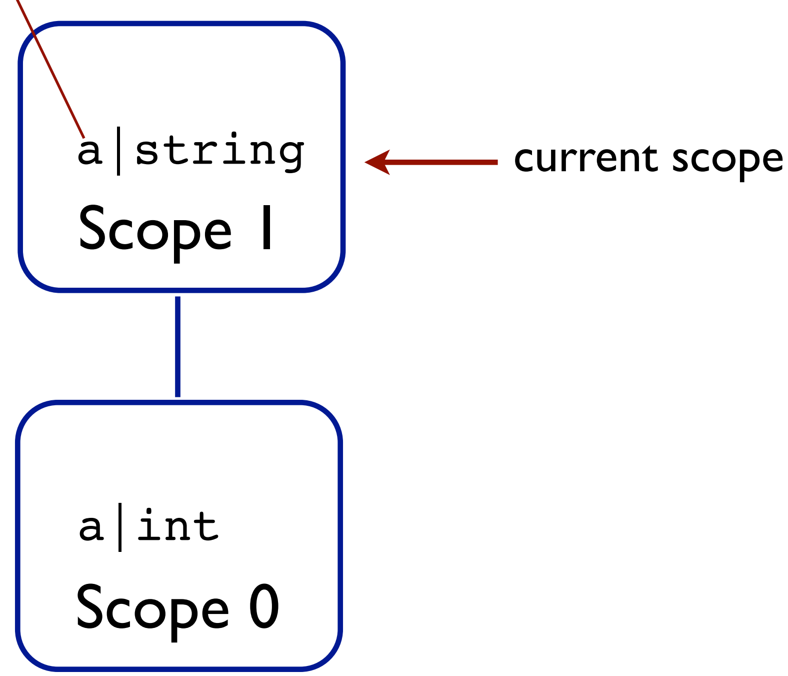
Symbol Table

Semantic Analysis



Source Code

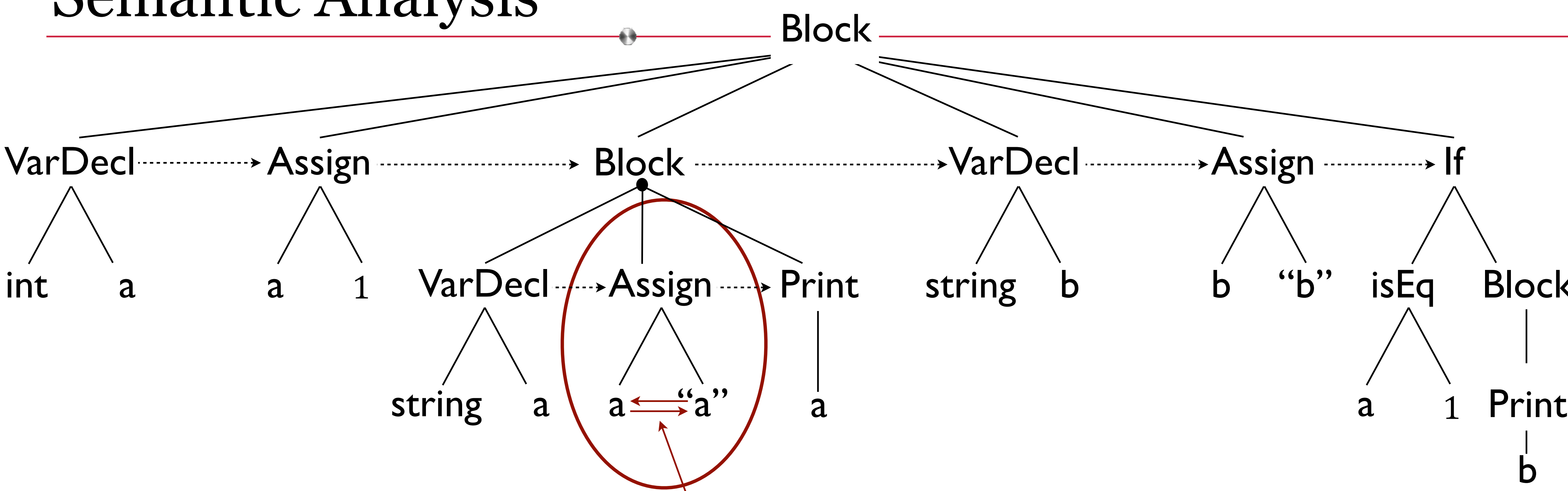
```
{
    int a
    a = 1
    {
        string a
        a = "a"
        print(a)
    }
    string b
    b = "b"
    if (a == 1) {
        print(b)
    }
}
```



Symbol Table

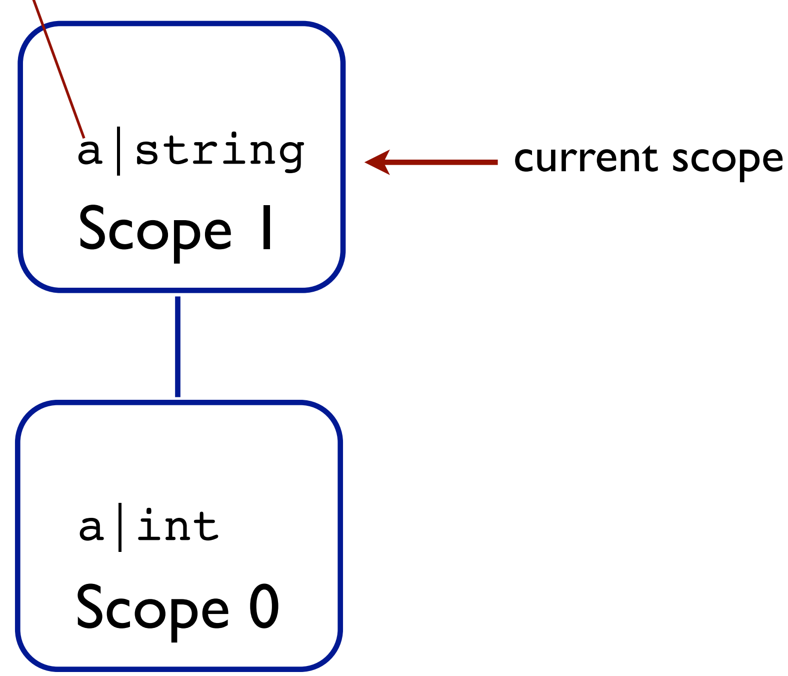
Initialize Scope 0
add symbol a
lookup symbol a
check types
Initialize Scope 1
add symbol a
lookup symbol a
in the ***current scope***

Semantic Analysis



Source Code

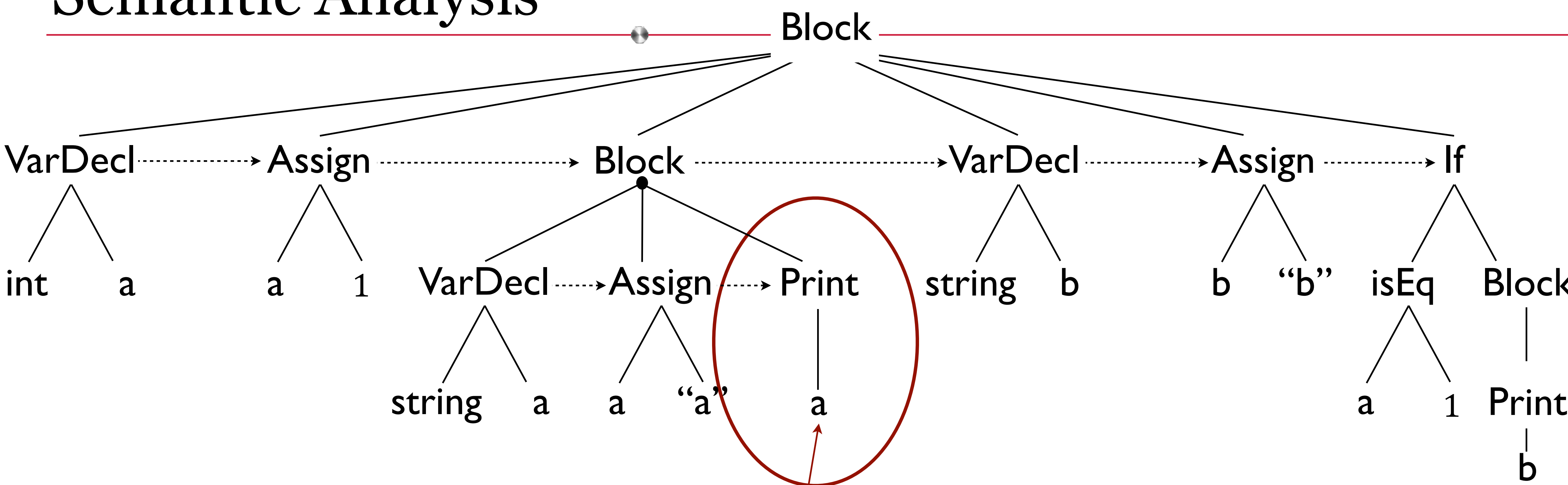
```
{
    int a
    a = 1
    {
        string a
        a = "a"
        print(a)
    }
    string b
    b = "b"
    if (a == 1) {
        print(b)
    }
}
```



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.

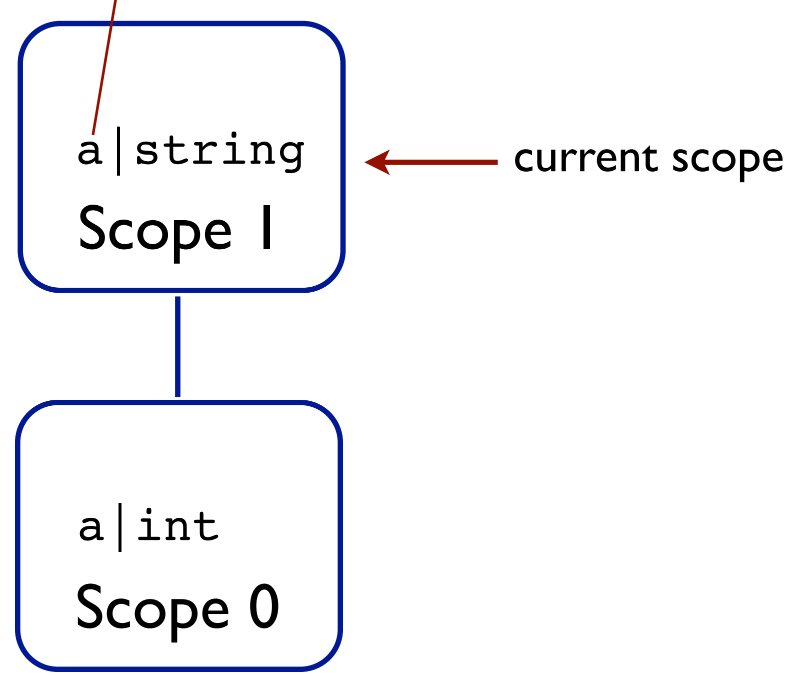
Symbol Table

Semantic Analysis



Source Code

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

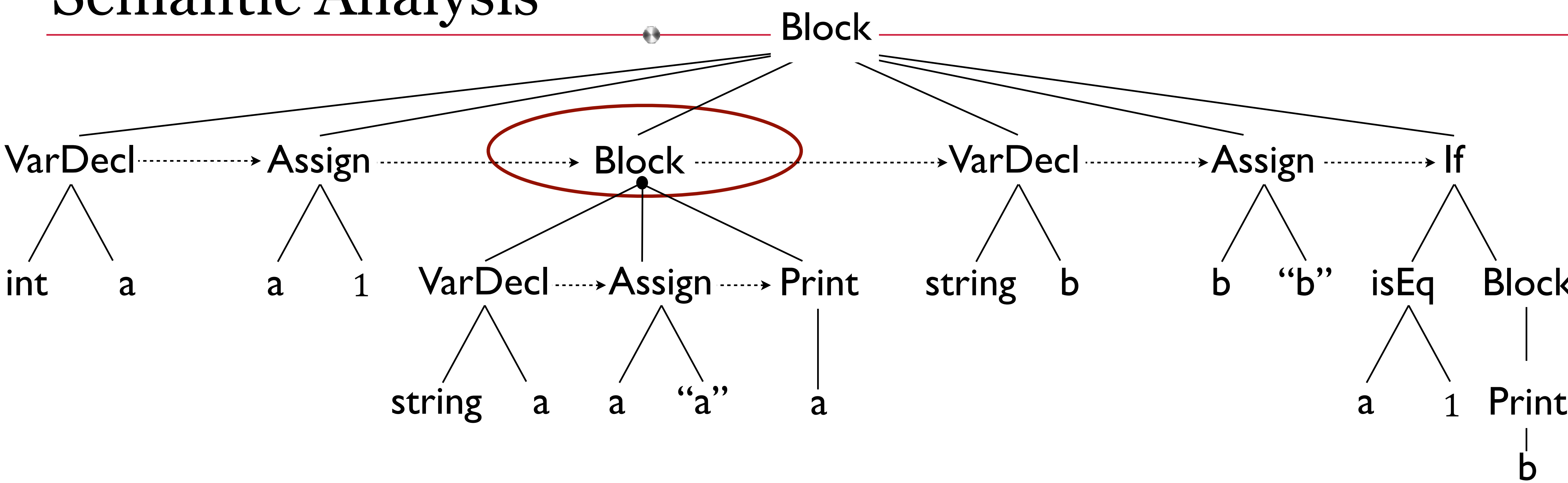


Symbol Table

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

*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!*

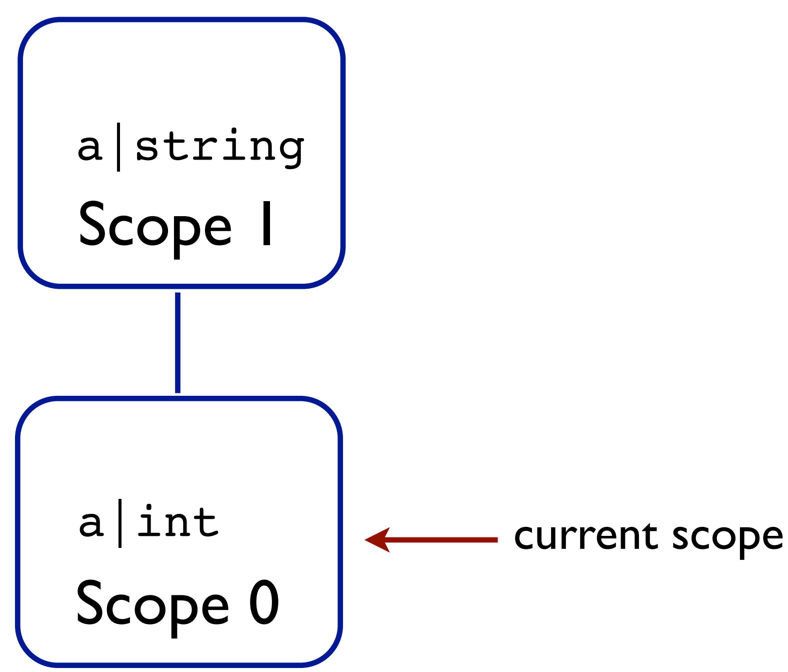
Semantic Analysis



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.*

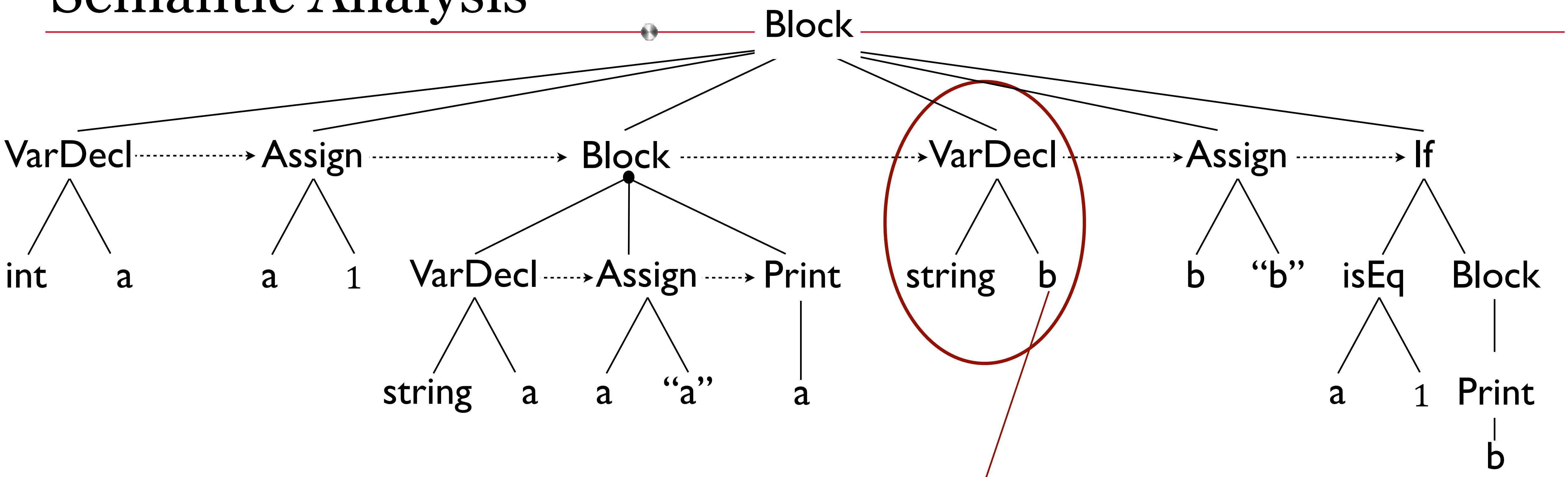
Source Code

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



Symbol Table

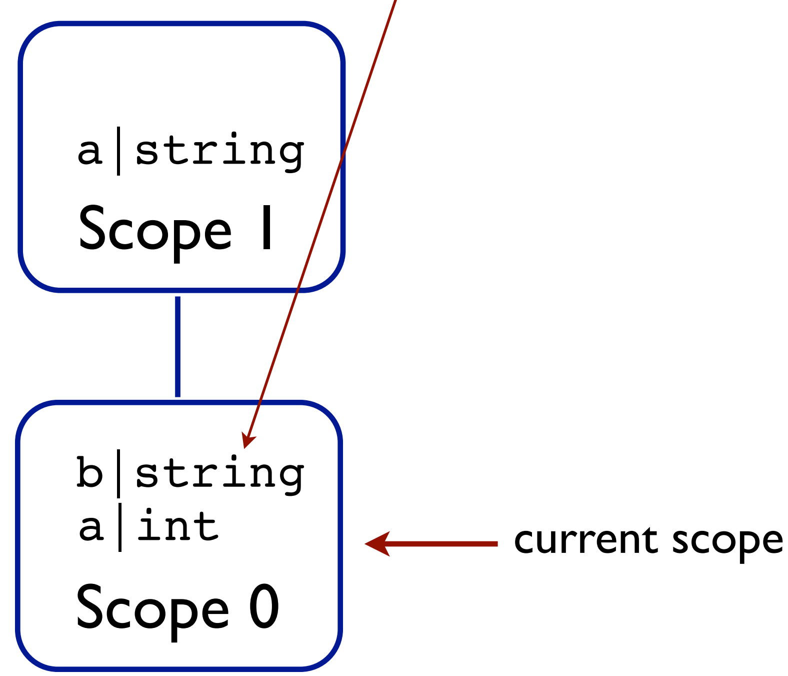
Semantic Analysis



Source Code

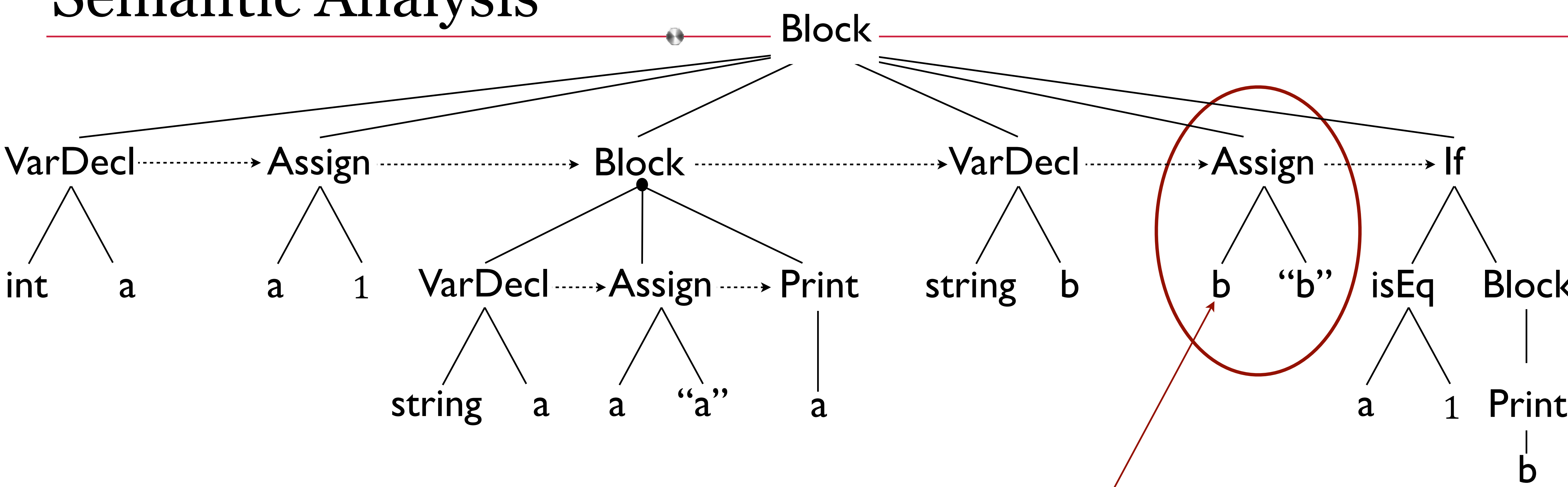
```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```

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
in the ***current scope***



Symbol Table

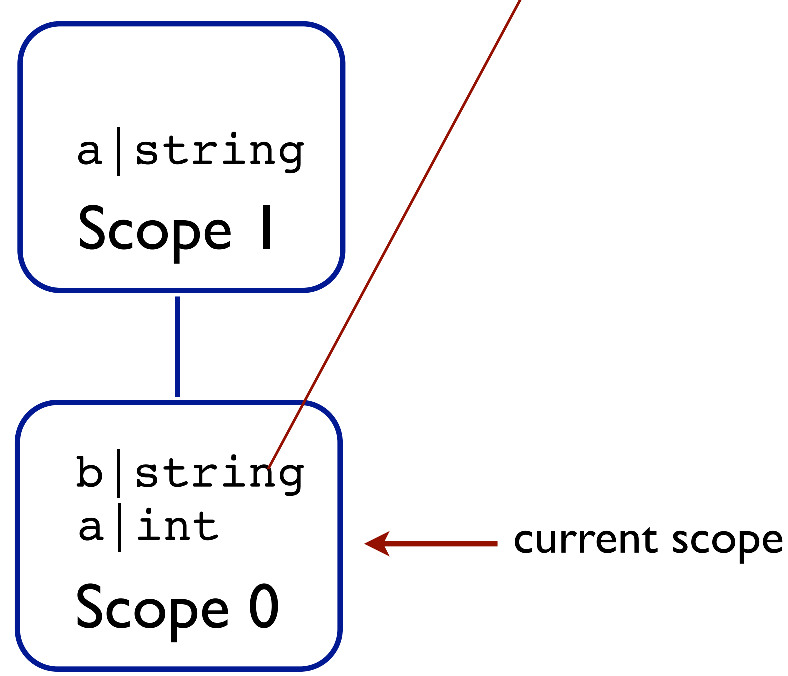
Semantic Analysis



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
in the ***current scope***

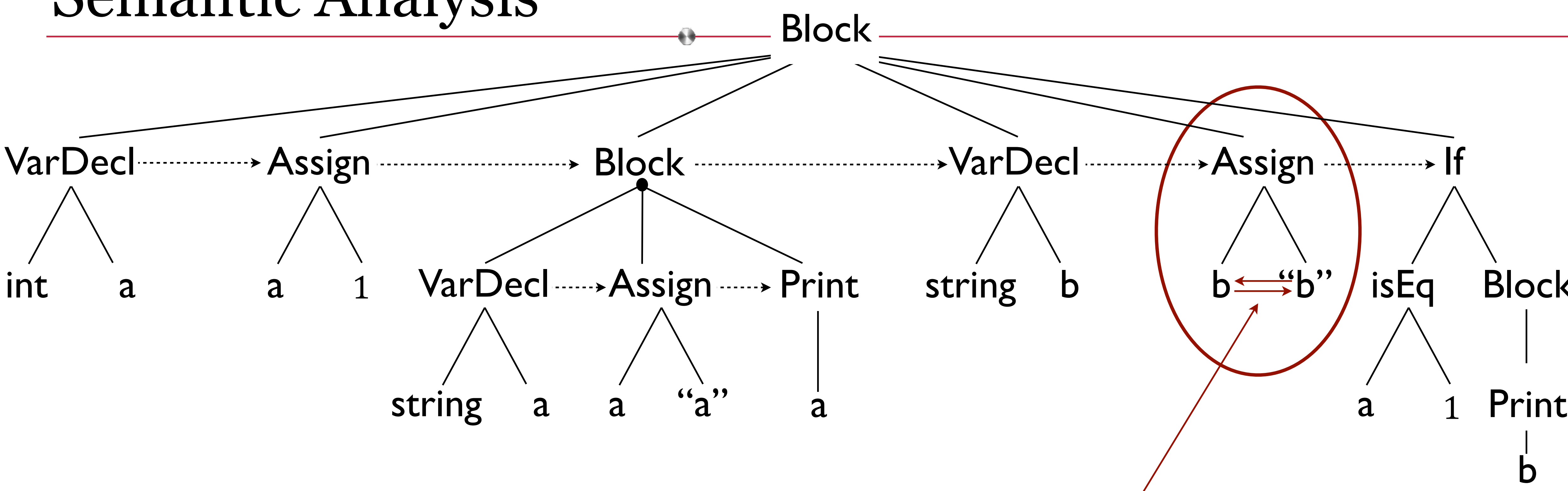
Source Code

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



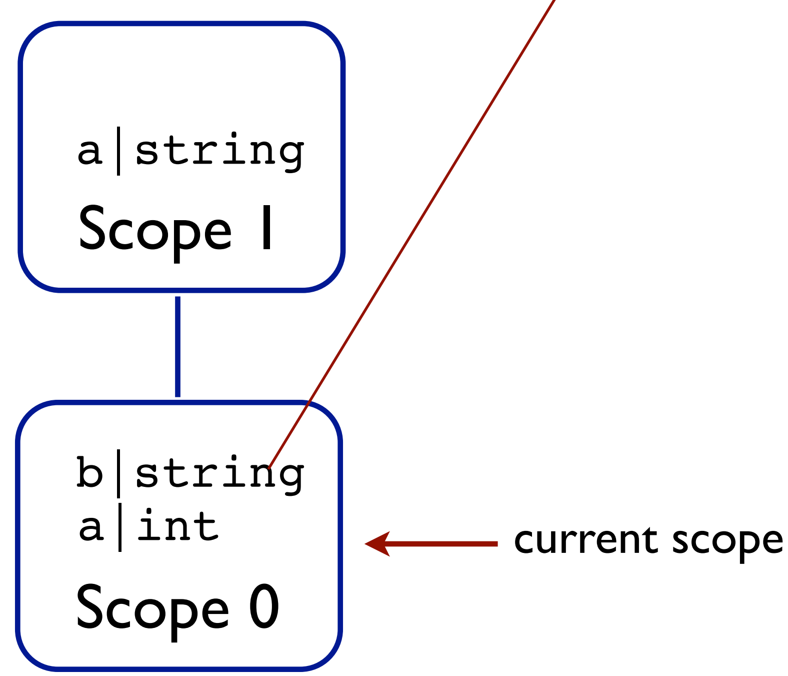
Symbol Table

Semantic Analysis



Source Code

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

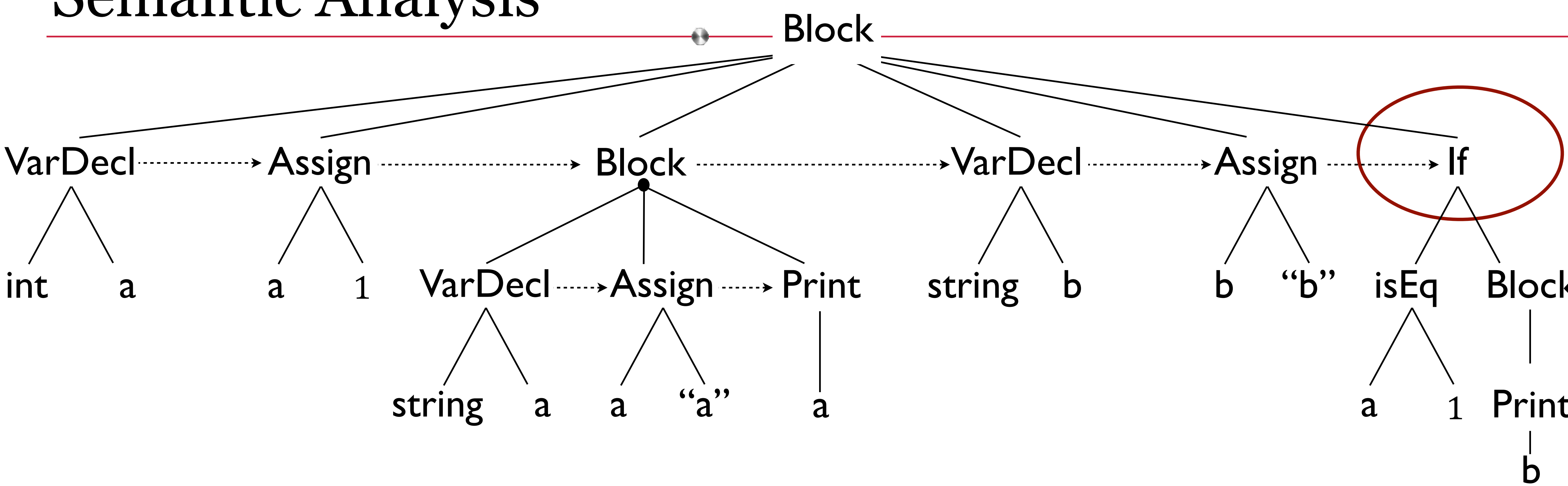


Symbol Table

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

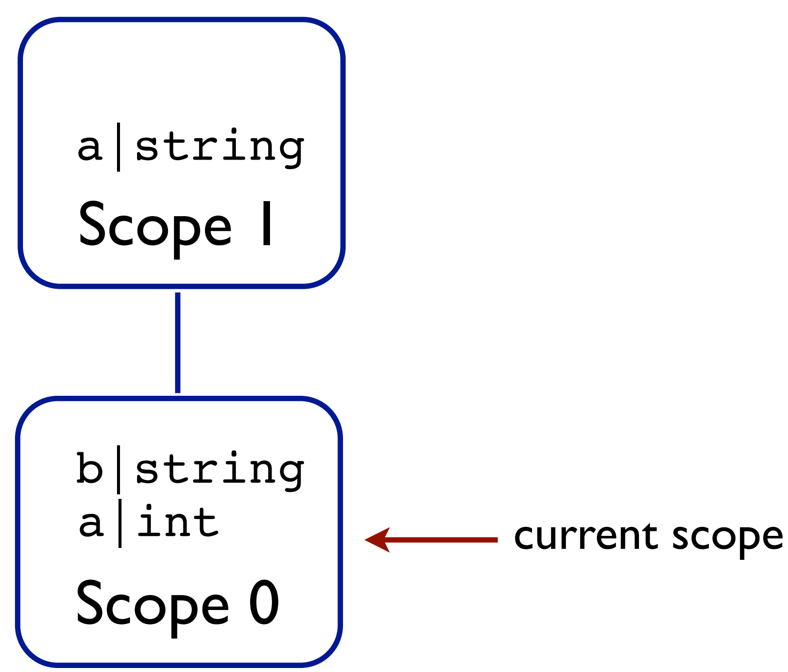
Verify that the left child and right child are type compatible for assignment.

Semantic Analysis



Source Code

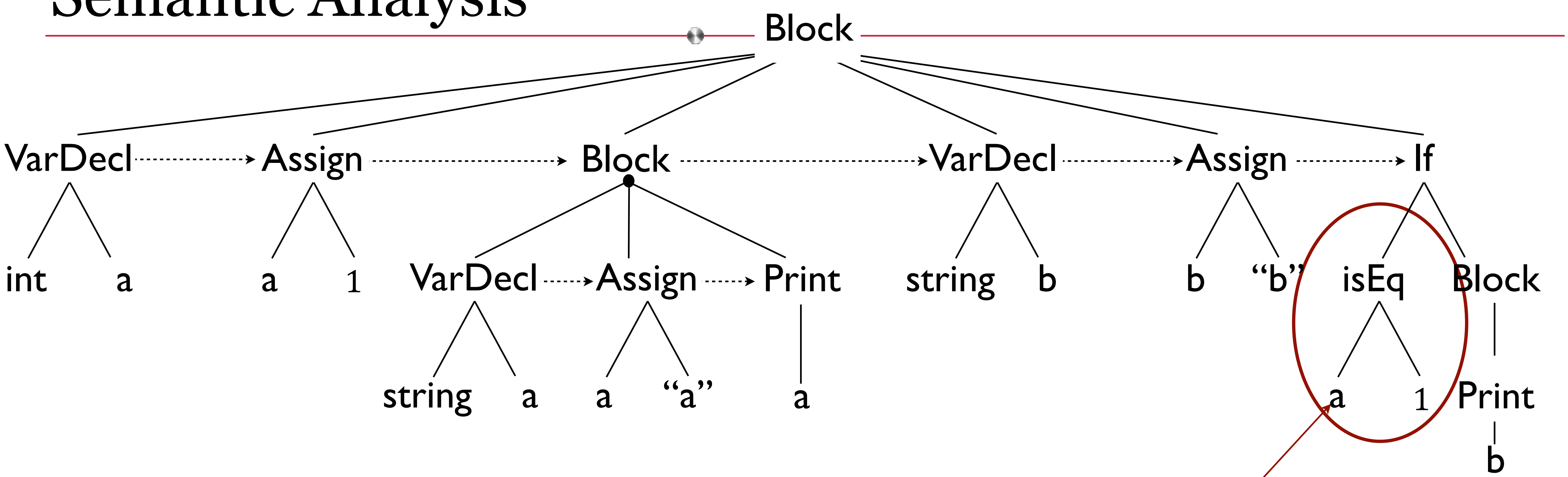
```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```



Symbol Table

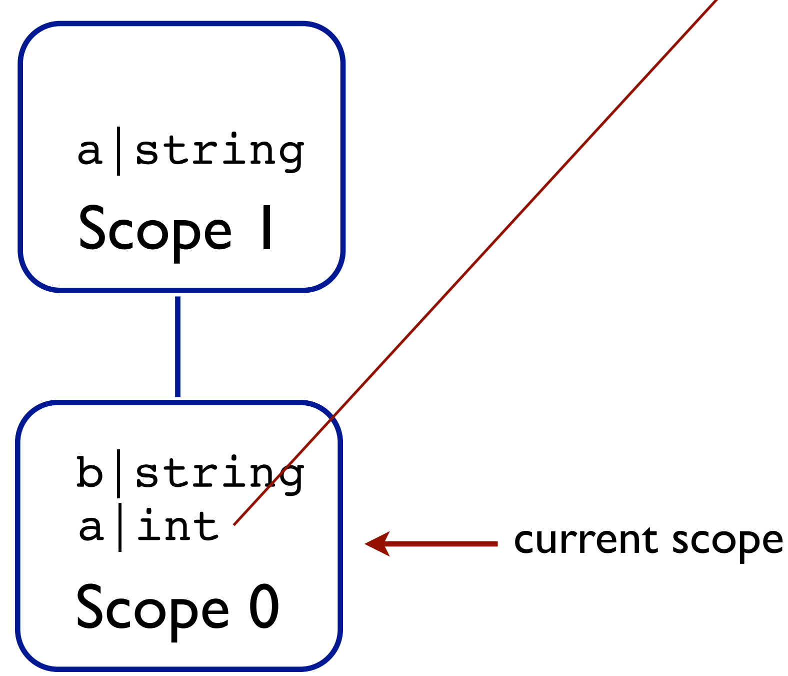
Initialize Scope 0
add symbol a
lookup symbol a
check types
Initialize Scope 1
add symbol a
lookup symbol a
check types
lookup symbol a
Close Scope 1
add symbol b
lookup symbol b
check types

Semantic Analysis



Source Code

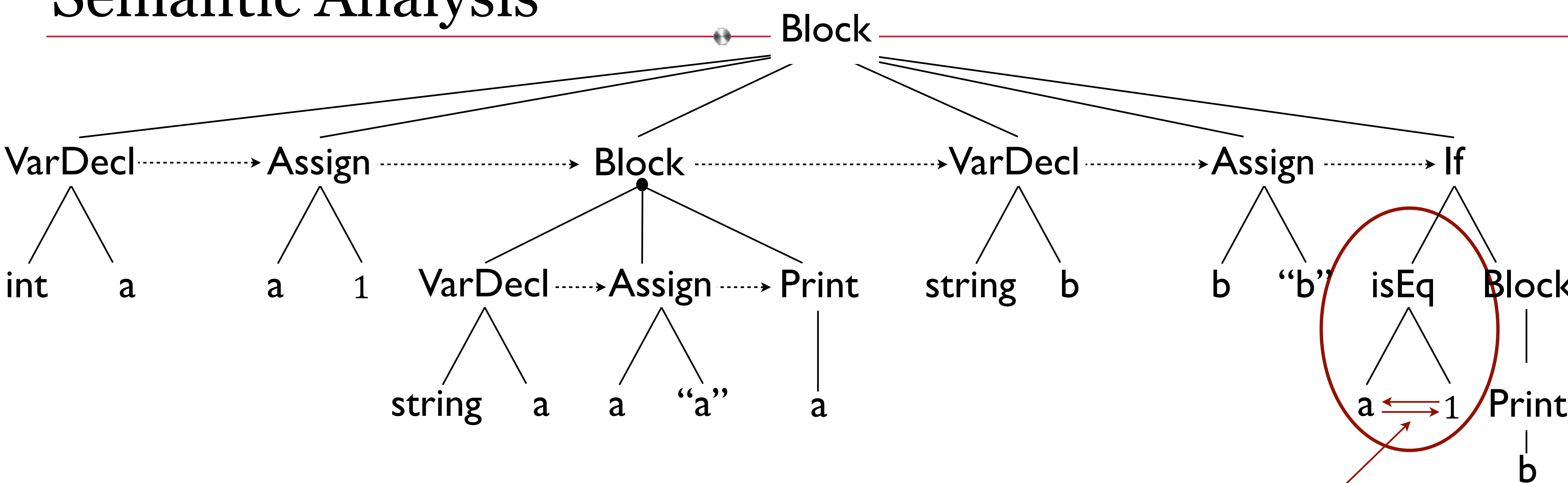
```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```



Symbol Table

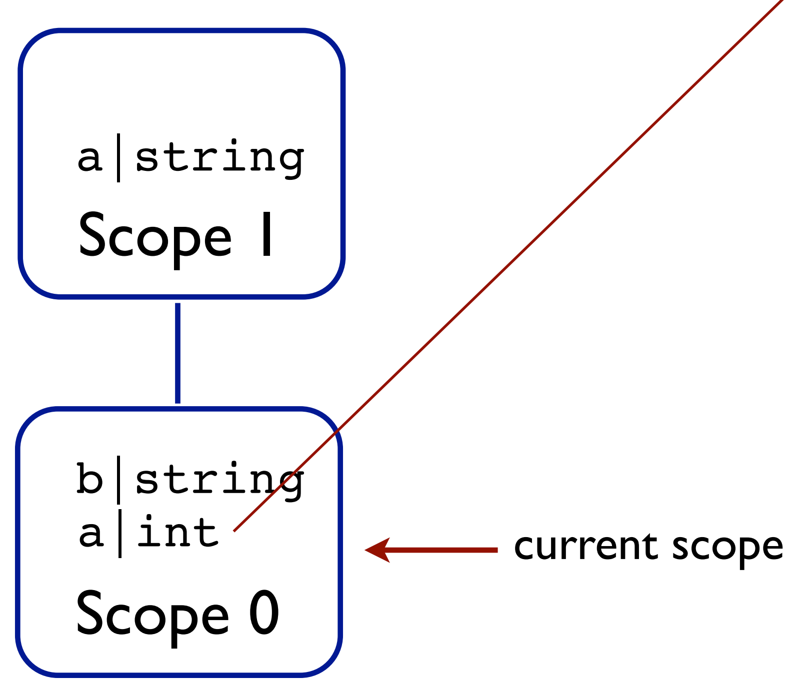
Initialize Scope 0
add symbol a
lookup symbol a
check types
Initialize Scope 1
add symbol a
lookup symbol a
check types
lookup symbol a
Close Scope 1
add symbol b
lookup symbol b
check types
lookup symbol a
in the ***current scope***

Semantic Analysis



Source Code

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

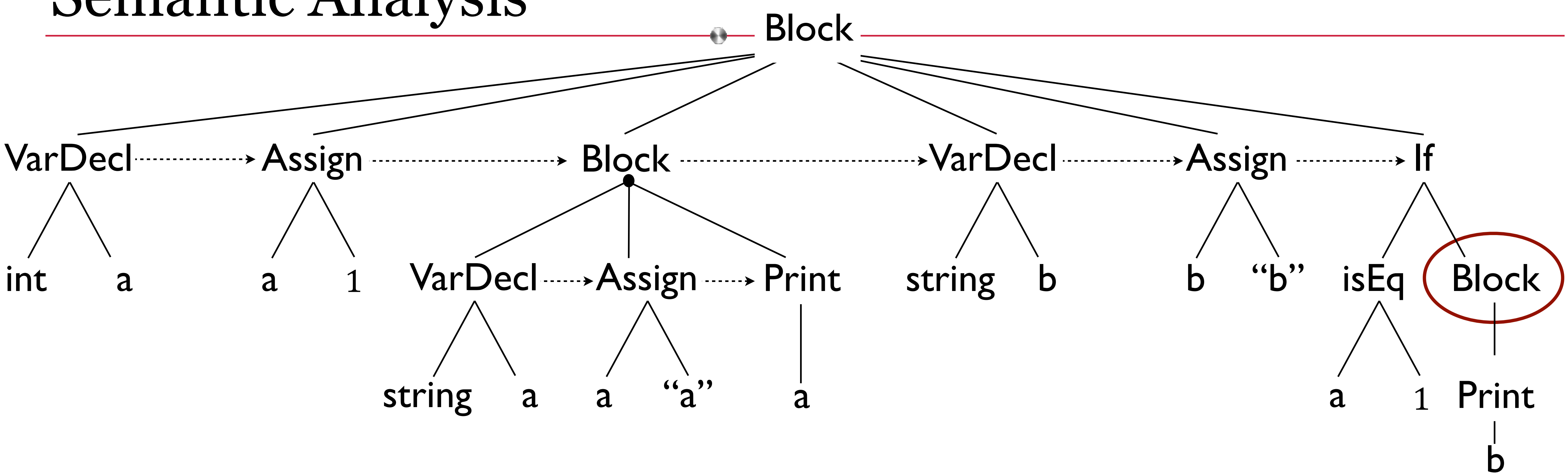


Symbol Table

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
lookup symbol a
check types

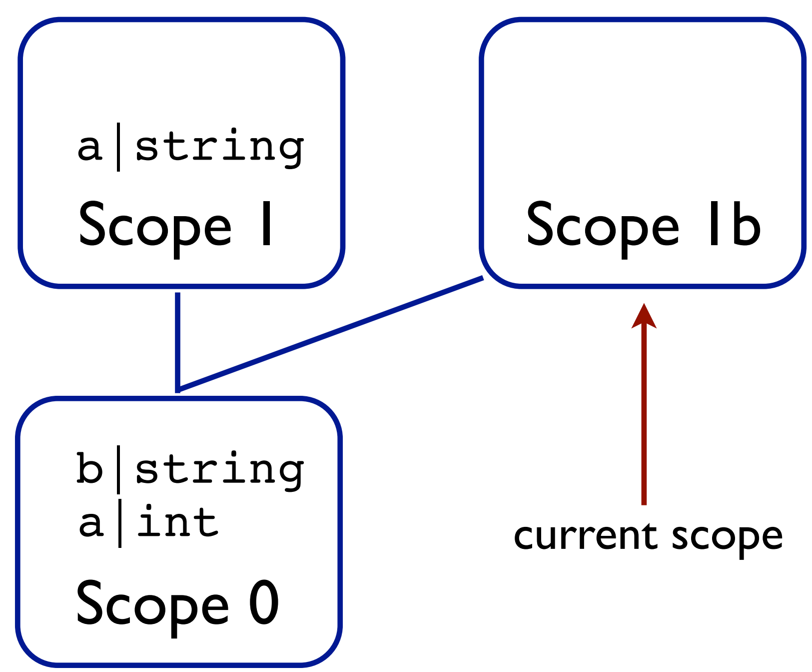
Verify that the left child and right child are ***type comparable***

Semantic Analysis



Source Code

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

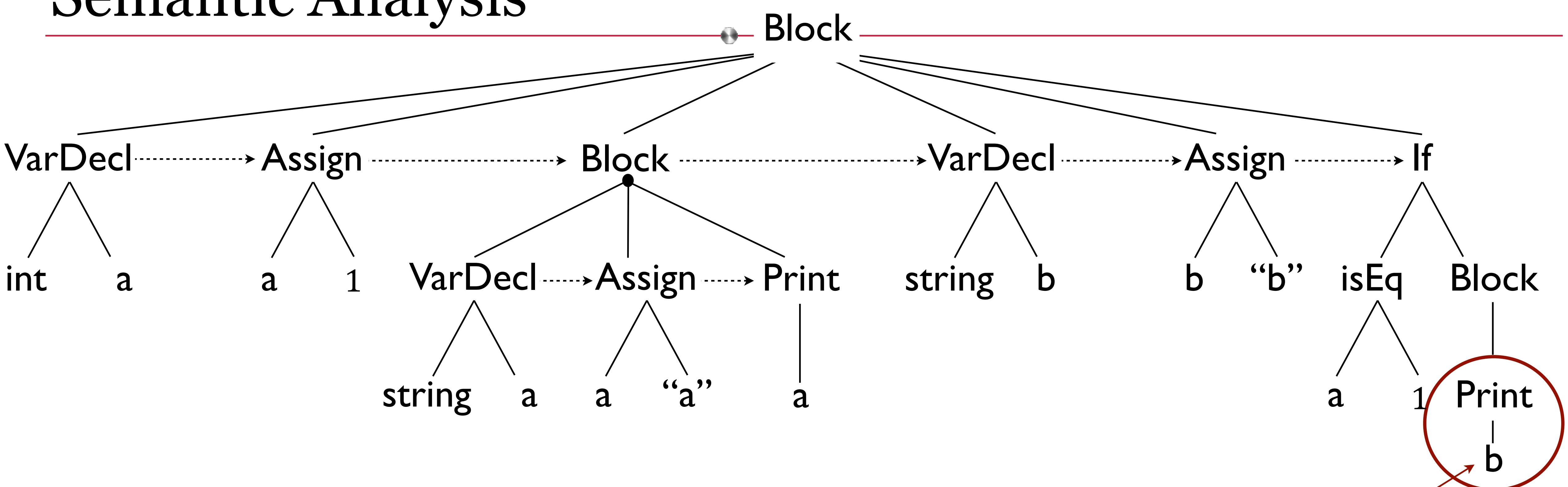


Initialize Scope 0
add symbol a
lookup symbol a
check types
Initialize Scope 1
add symbol a
lookup symbol a
check types
lookup symbol a
Close Scope 1
add symbol b
lookup symbol b
check types
lookup symbol a
check types
Initialize Scope 1b

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

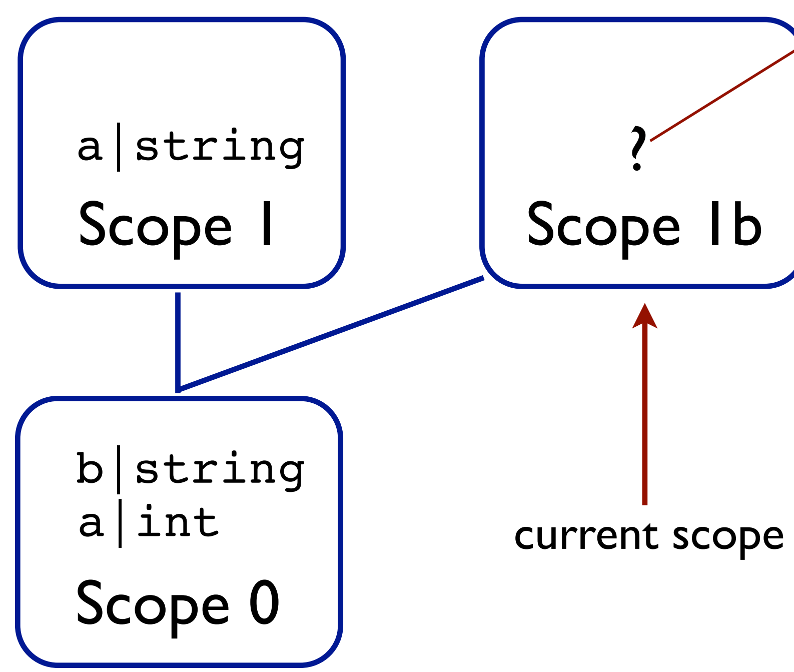
Symbol Table

Semantic Analysis



Source Code

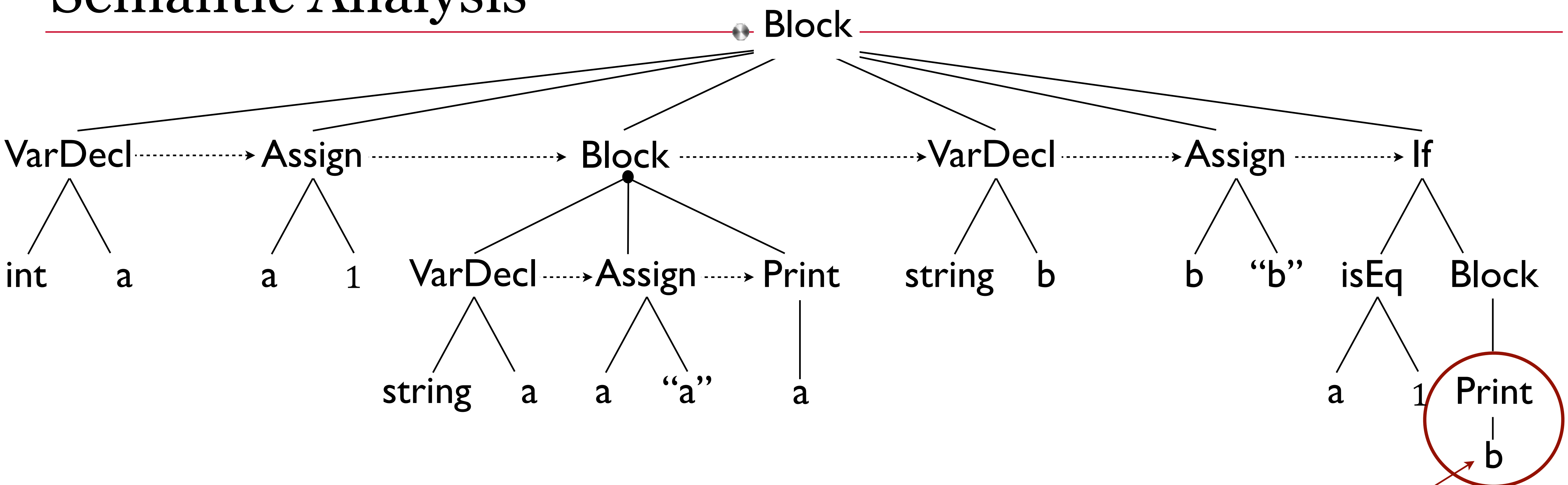
```
{
    int a
    a = 1
    {
        string a
        a = "a"
        print(a)
    }
    string b
    b = "b"
    if (a == 1) {
        print(b)
    }
}
```



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***.

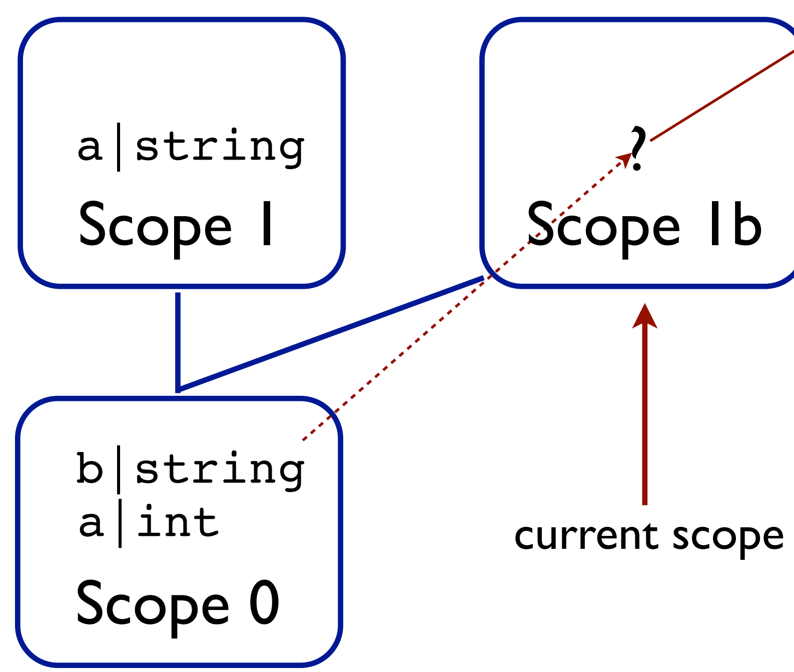
Symbol Table

Semantic Analysis



Source Code

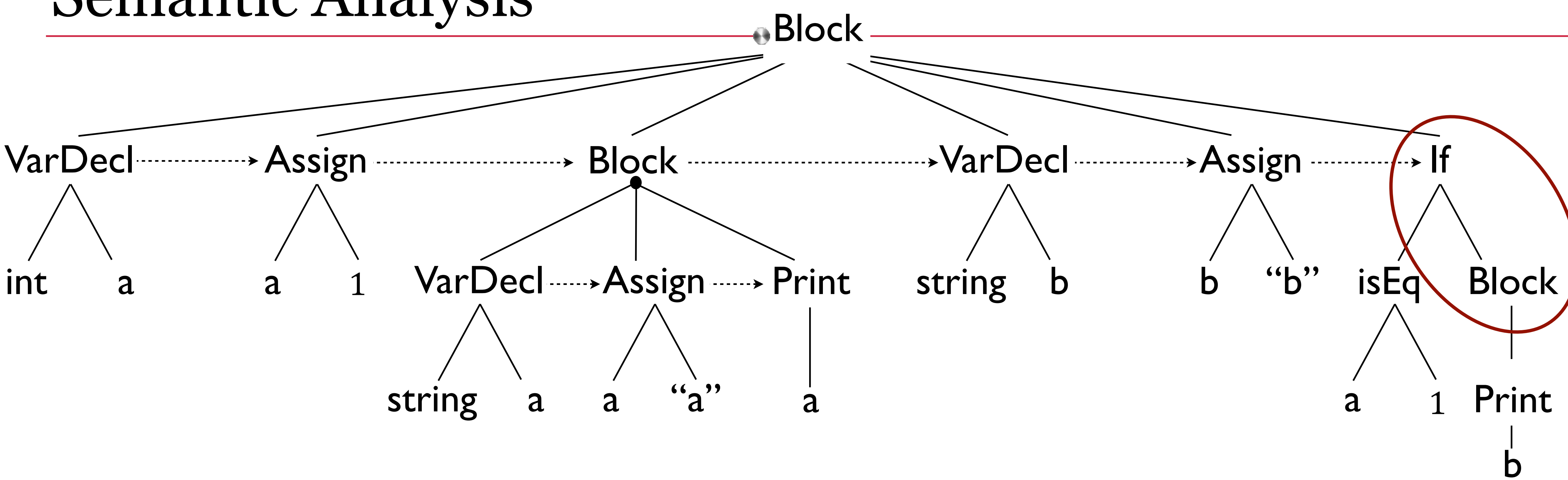
```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```



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.

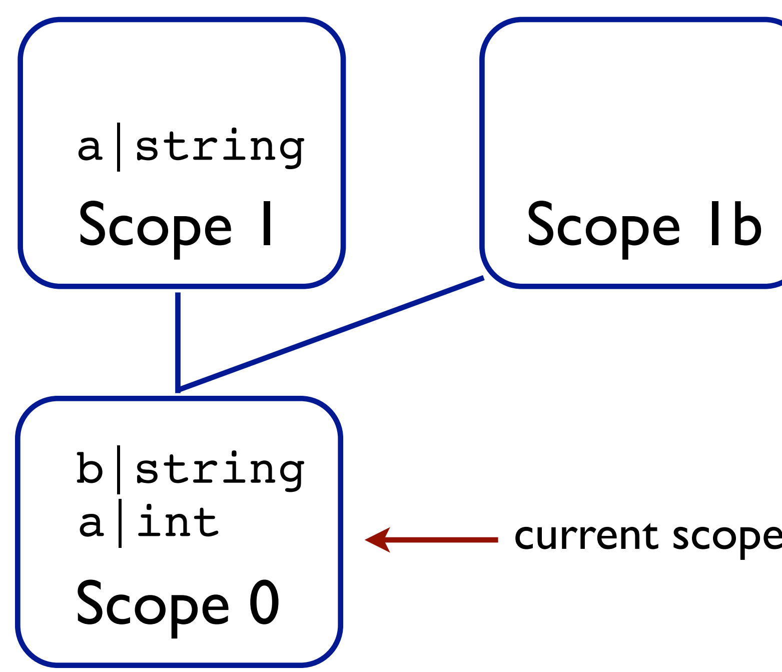
Symbol Table

Semantic Analysis



Source Code

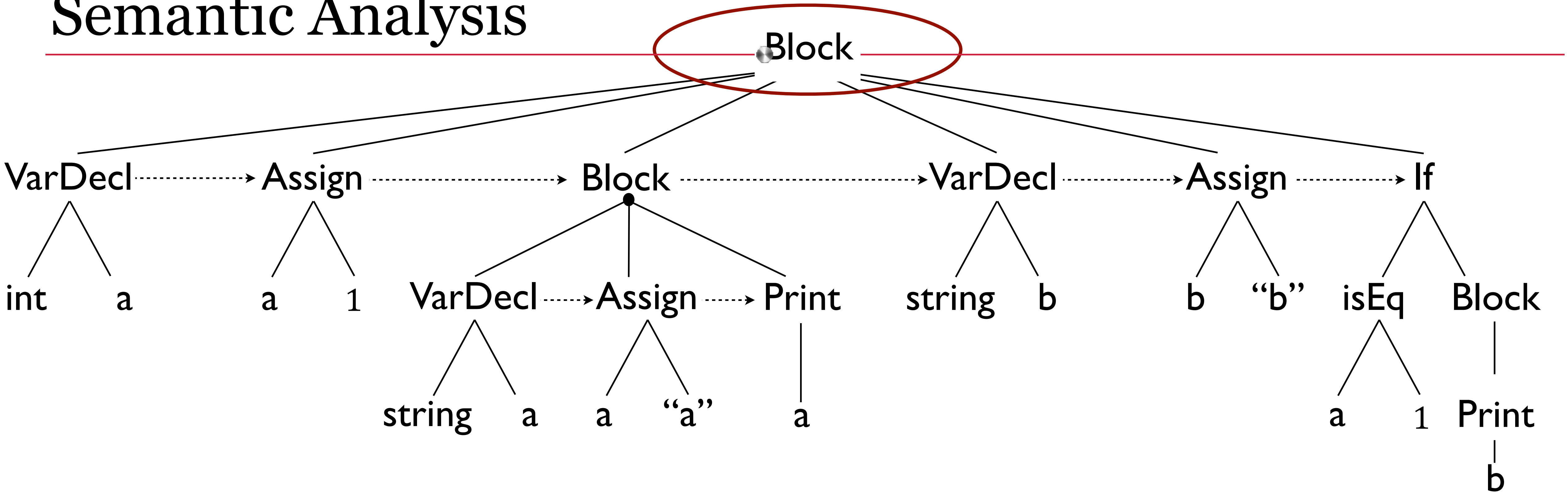
```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```



Symbol Table

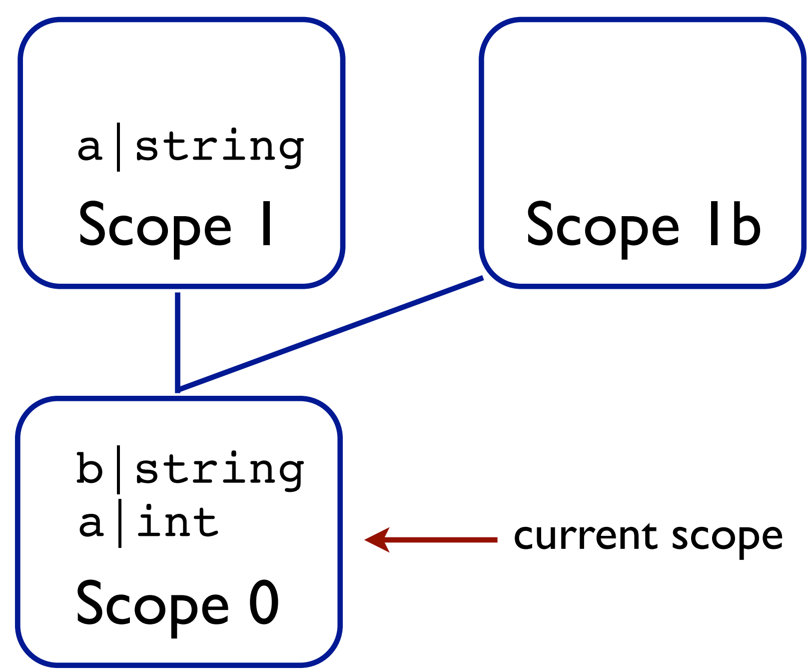
Initialize Scope 0
add symbol a
lookup symbol a
check types
Initialize Scope 1
add symbol a
lookup symbol a
check types
lookup symbol a
Close Scope 1
add symbol b
lookup symbol b
check types
lookup symbol a
check types
Initialize Scope 1b
lookup symbol b
Close Scope 1b

Semantic Analysis



Source Code

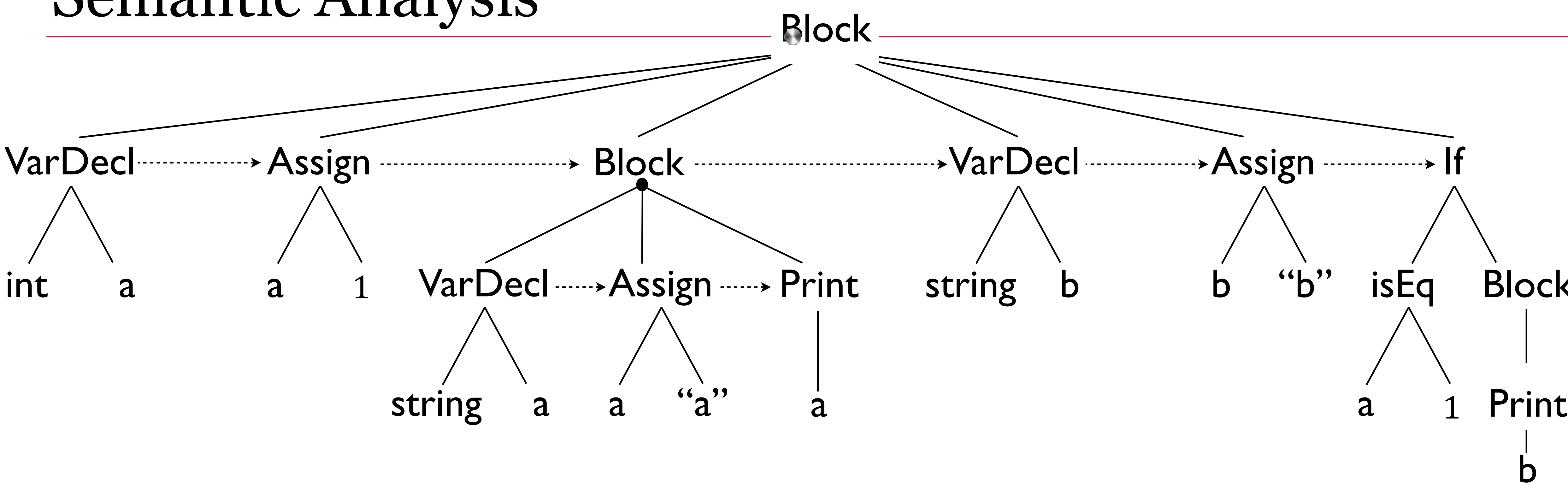
```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```



Symbol Table

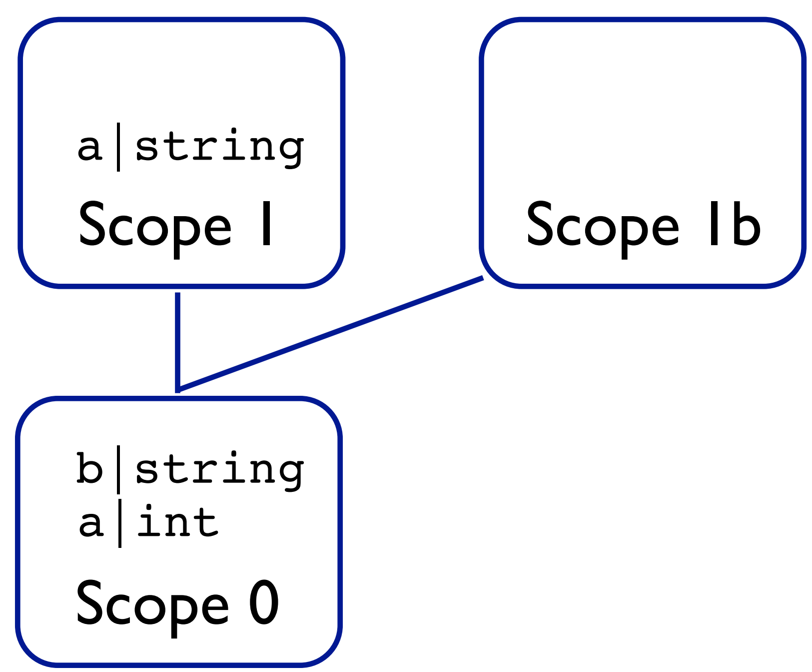
Initialize Scope 0
add symbol a
lookup symbol a
check types
Initialize Scope 1
add symbol a
lookup symbol a
check types
lookup symbol a
Close Scope 1
add symbol b
lookup symbol b
check types
lookup symbol a
check types
Initialize Scope 1b
lookup symbol b
Close Scope 1b
Close Scope 0

Semantic Analysis



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

We are ready for Code Generation.



Semantic Analysis

Yet another example in our language

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

There's an error here.
Where is it?

Semantic Analysis

Yet another example in our language

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

Type-mismatch error.

Semantic Analysis

Yet another example in our language

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

Type-mismatch error.

Wait. Is this okay?

Semantic Analysis

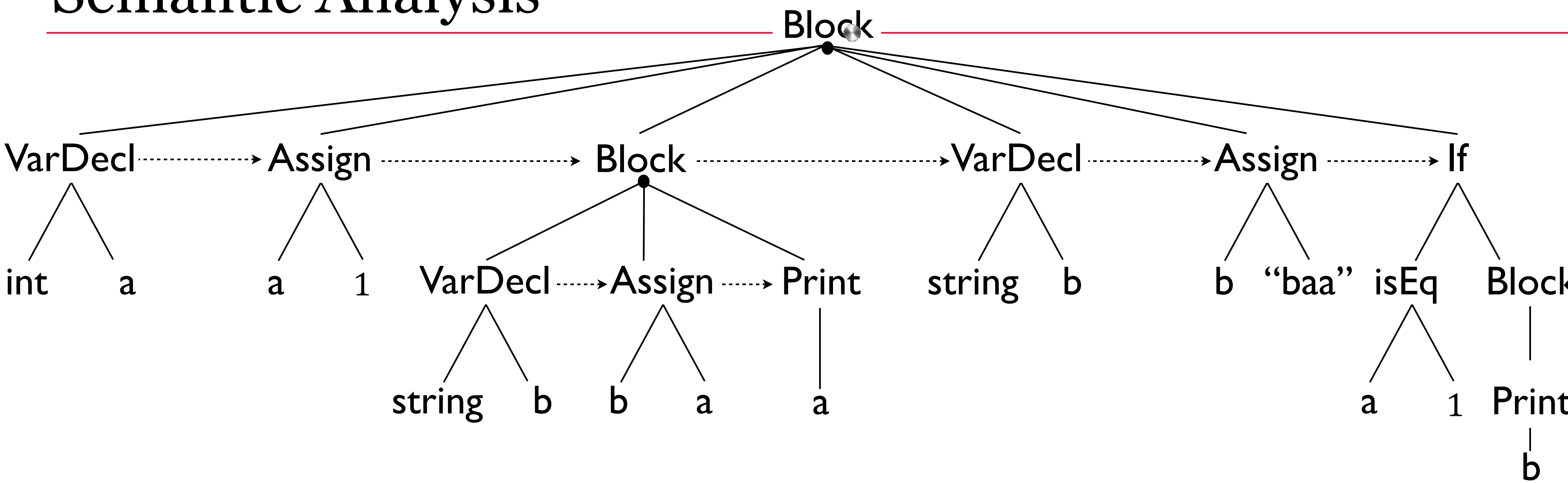
Yet another example in our language

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

Type-mismatch error.

Wait. Is this okay? Yes. (Sheep noise.)

Semantic Analysis



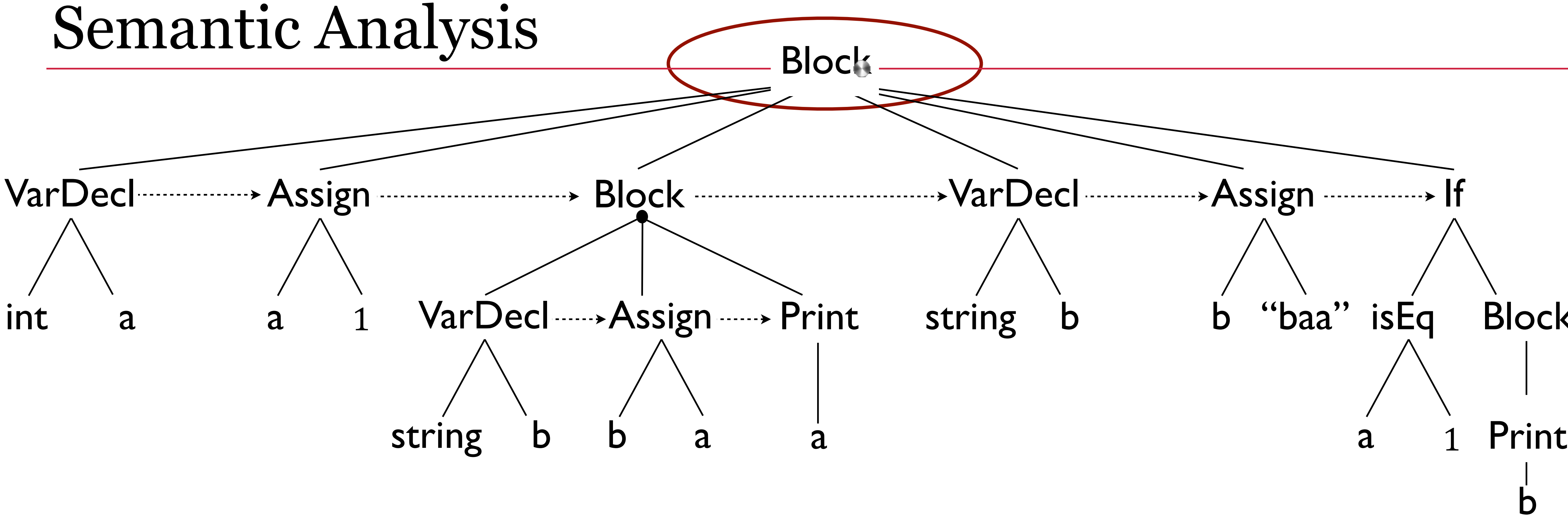
Source Code

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

Symbol Table

Semantic Analysis

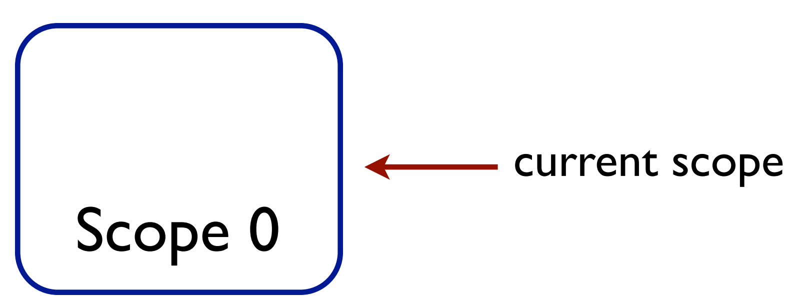
AST



Initialize Scope 0
Set the ***current scope*** *pointer.*

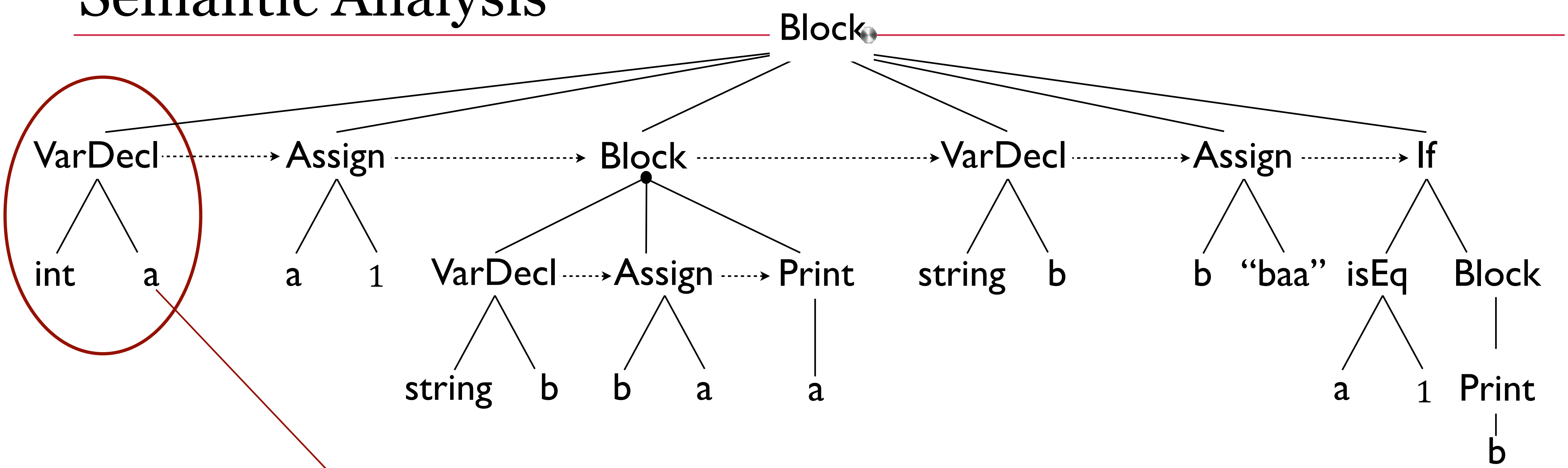
Source Code

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



Symbol Table

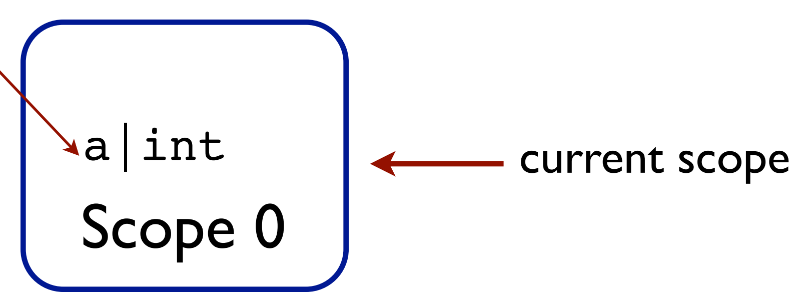
Semantic Analysis



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

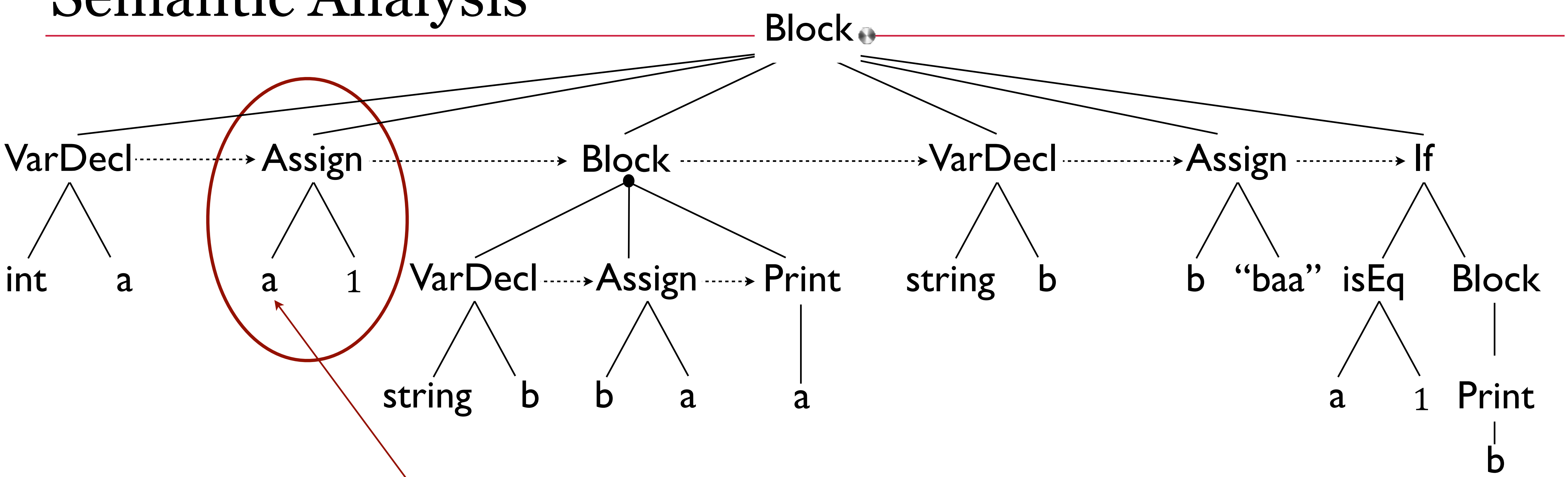
Source Code

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



Symbol Table

Semantic Analysis



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

Source Code

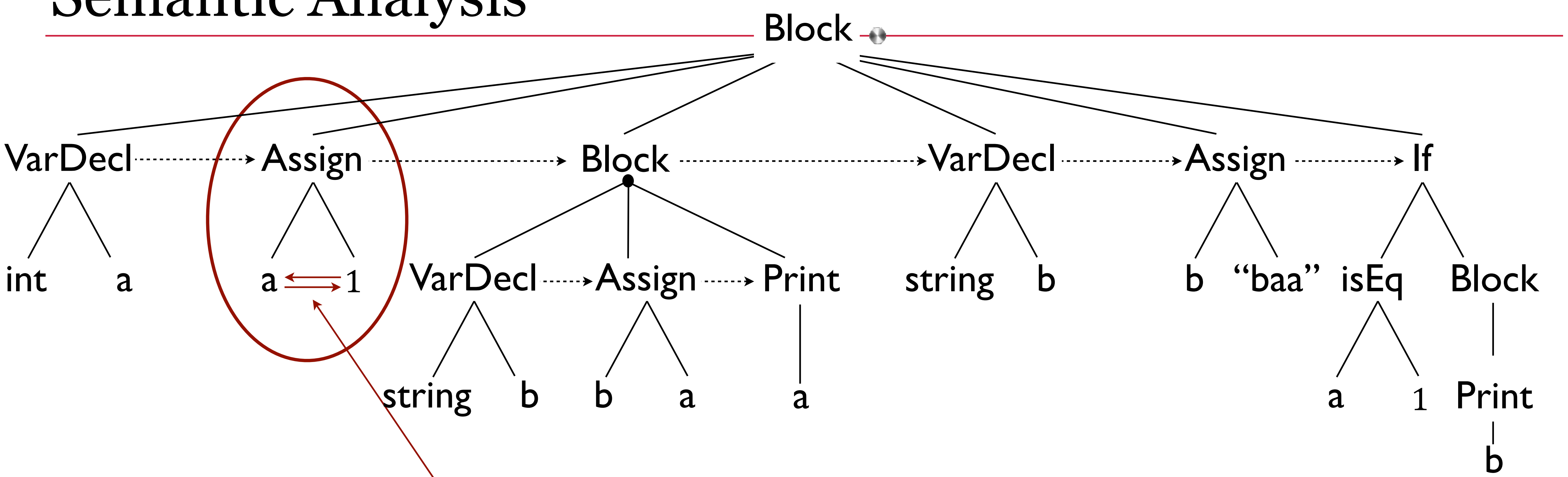
```
{
   int a
   a = 1
   {
      string b
      b = a
      print(a)
   }
   string b
   b = “baa"
   if (a == 1) {
      print(b)
   }
}
```

a | int
Scope 0

current scope

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

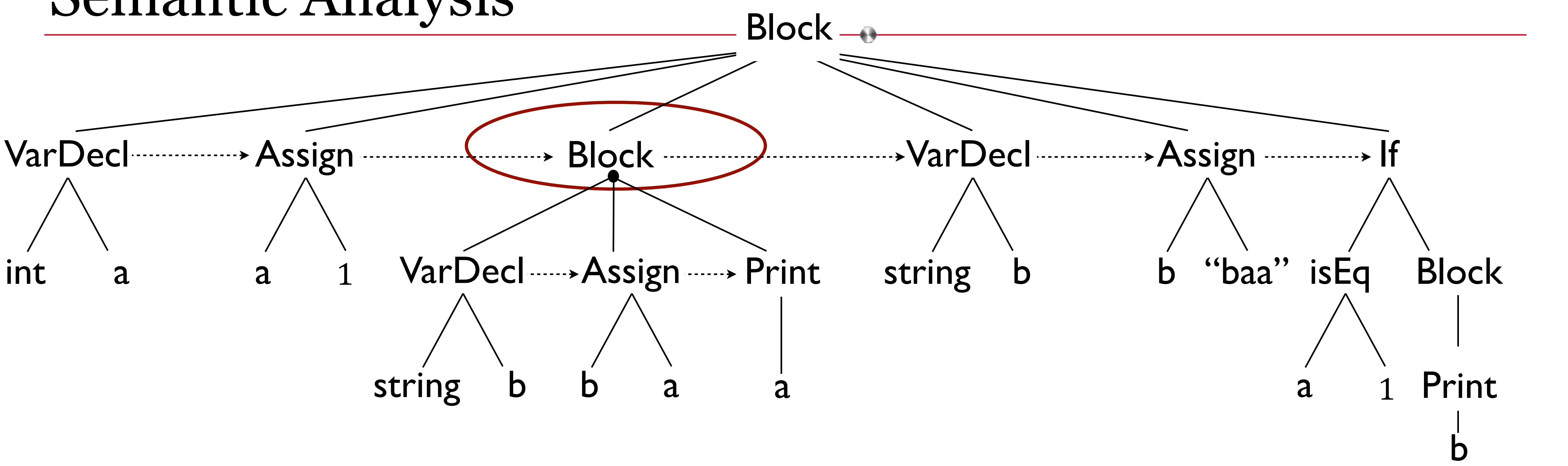
```
{
   int a
   a = 1
   {
      string b
      b = a
      print(a)
   }
   string b
   b = "baa"
   if (a == 1) {
      print(b)
   }
}
```

a | int
Scope 0

current scope

Symbol Table

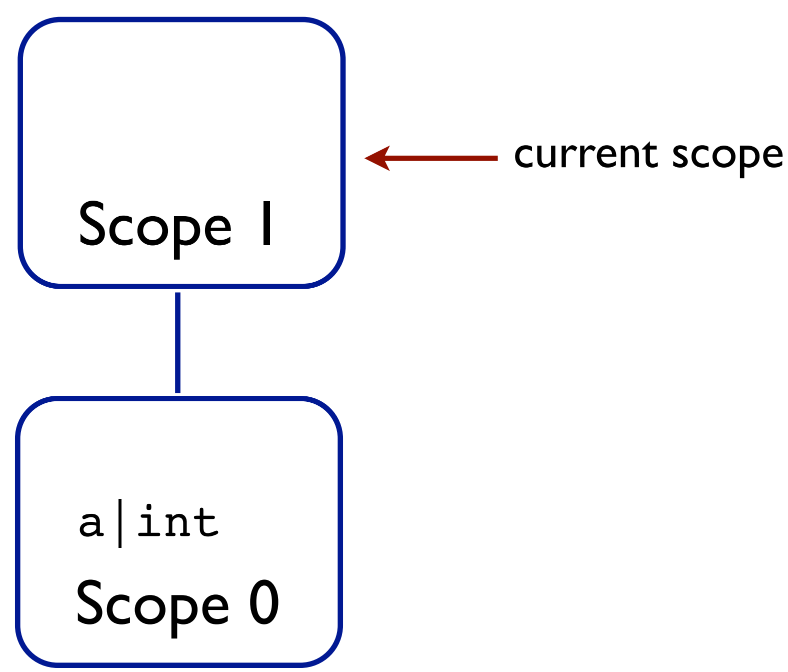
Semantic Analysis



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

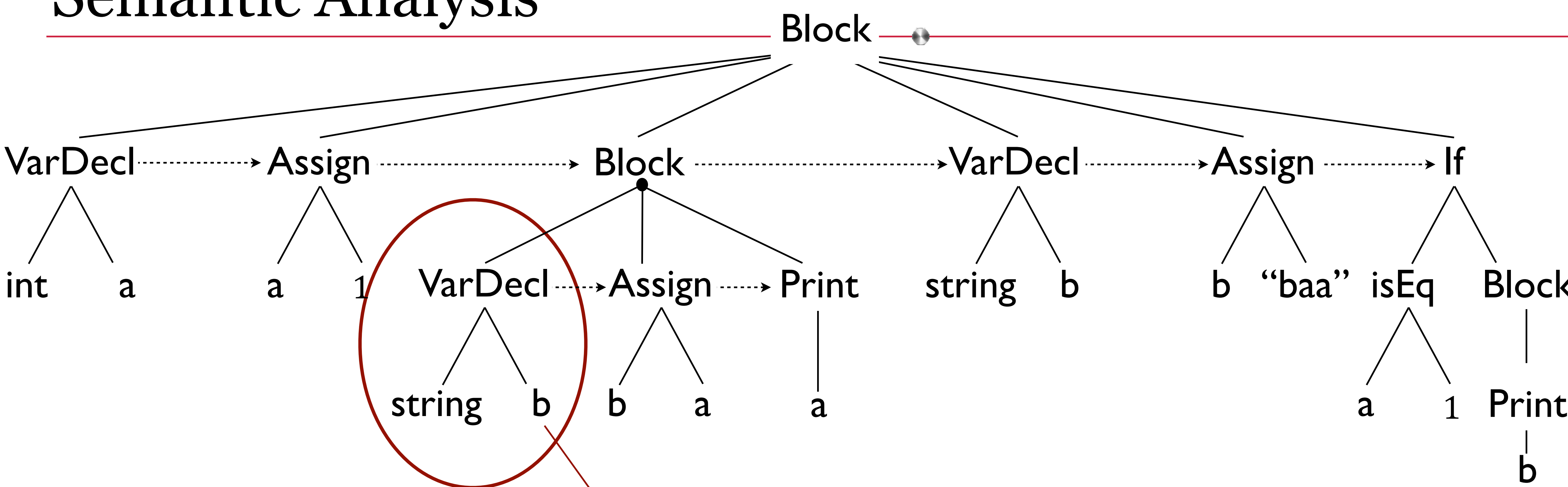
Source Code

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



Symbol Table

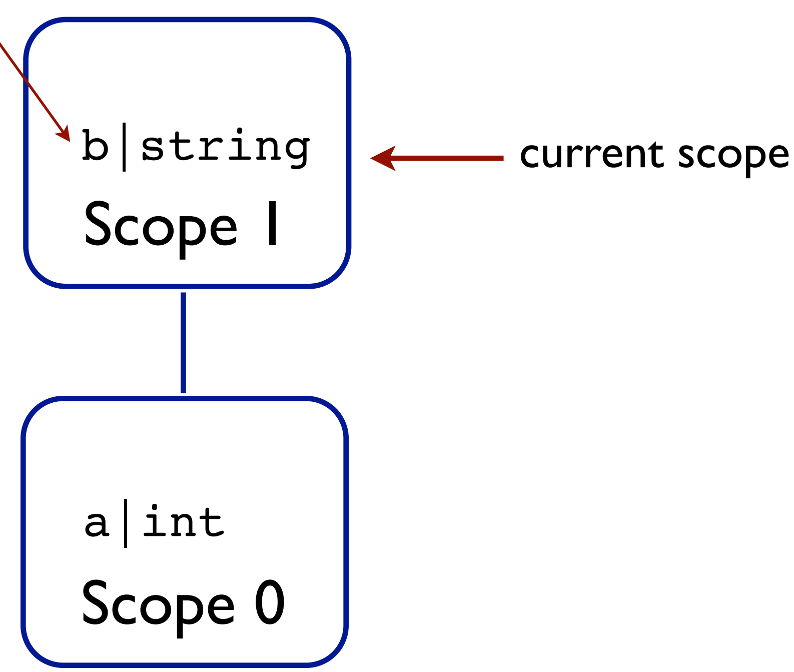
Semantic Analysis



Source Code

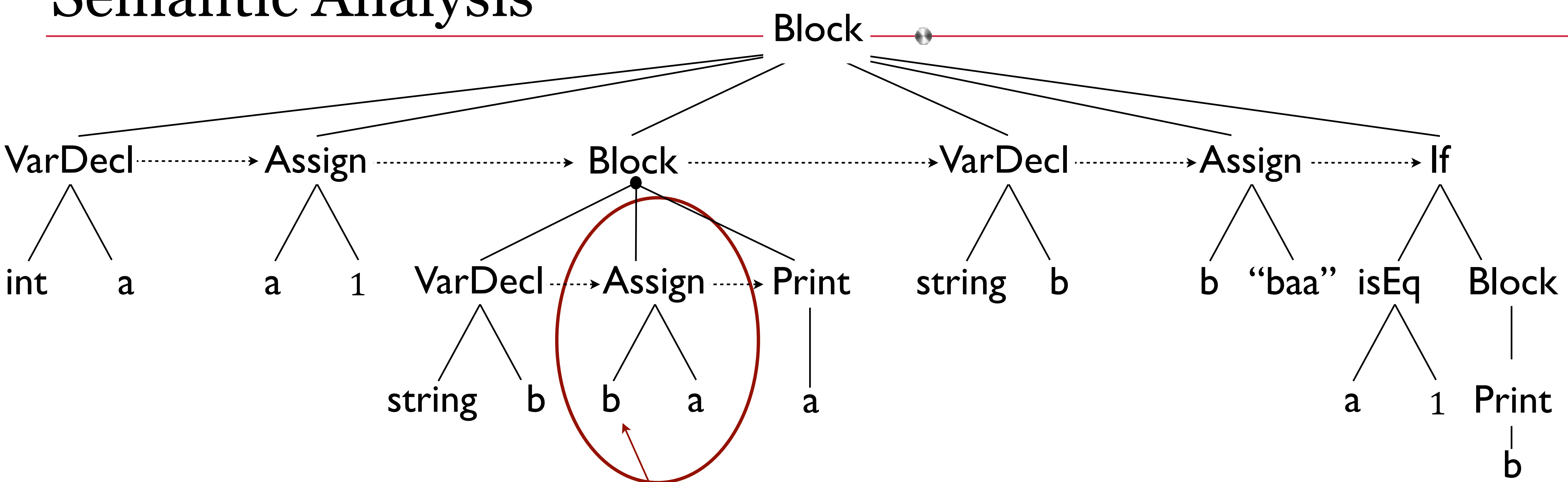
```
{
    int a
    a = 1
    {
      string b
      b = a
      print(a)
    }
    string b
    b = “baa"
    if (a == 1) {
      print(b)
    }
}
```

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



Symbol Table

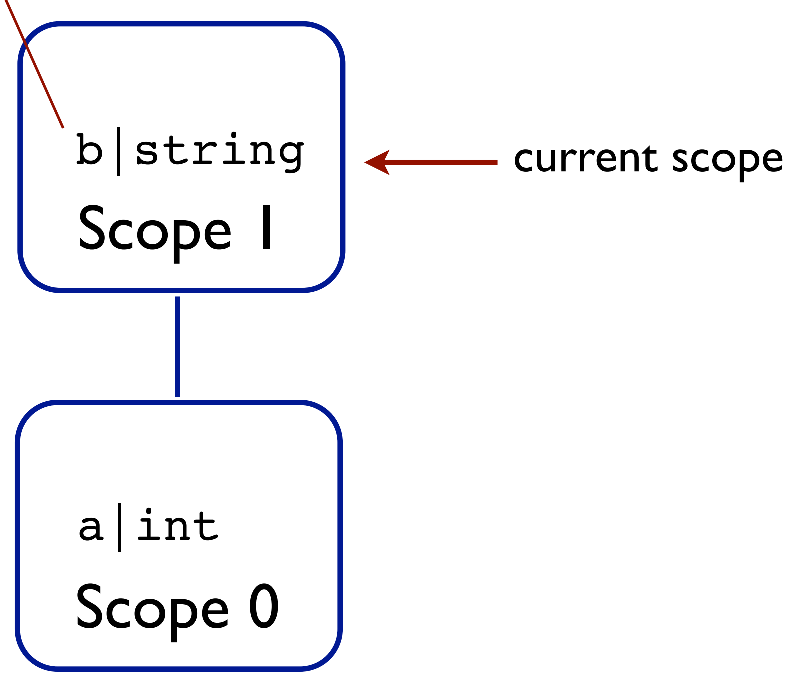
Semantic Analysis



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

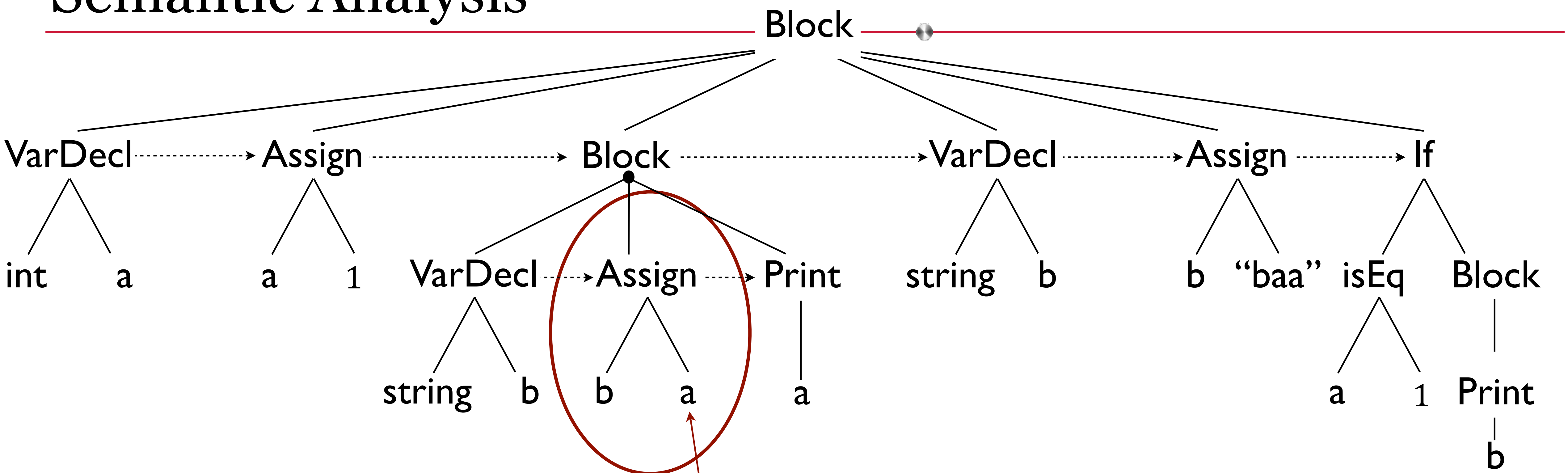
Source Code

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



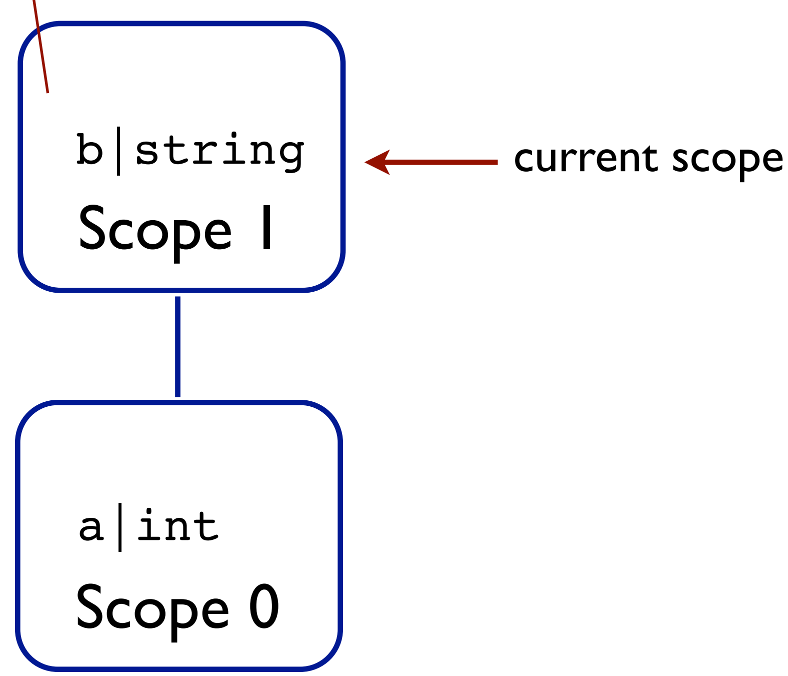
Symbol Table

Semantic Analysis



Source Code

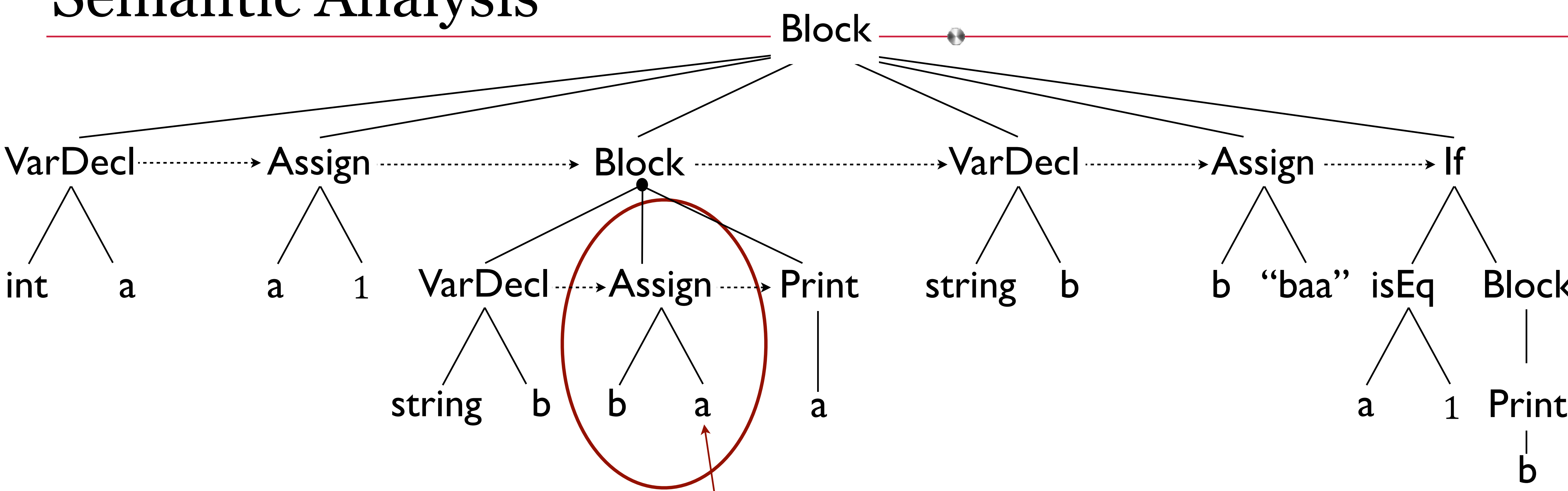
```
{
   int a
   a = 1
   {
      string b
      b = a
      print(a)
   }
   string b
   b = "baa"
   if (a == 1) {
      print(b)
   }
}
```



Symbol Table

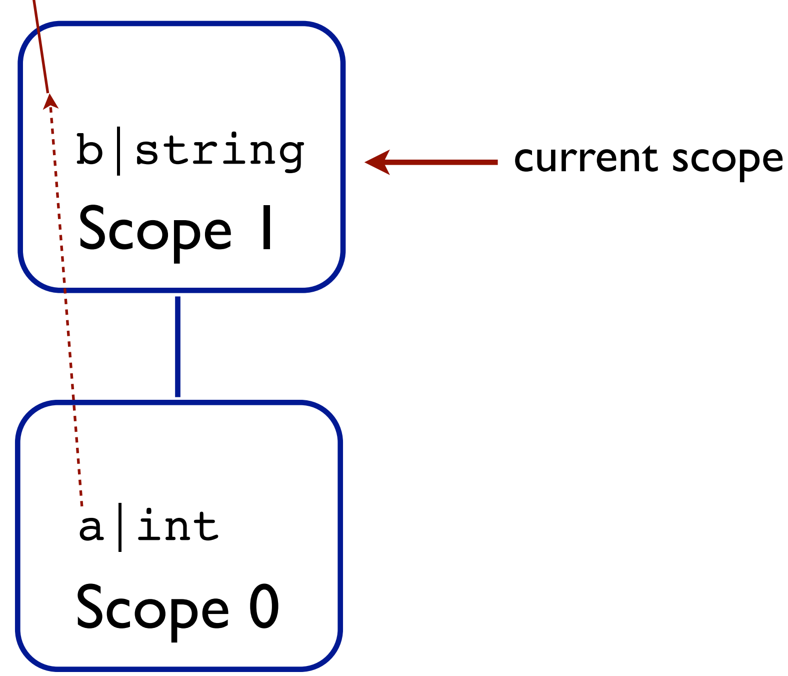
Initialize Scope 0
add symbol a
lookup symbol a
check types
Initialize Scope 1
add symbol b
lookup symbol b
lookup symbol a
in the ***current scope***
It's not there, so…

Semantic Analysis



Source Code

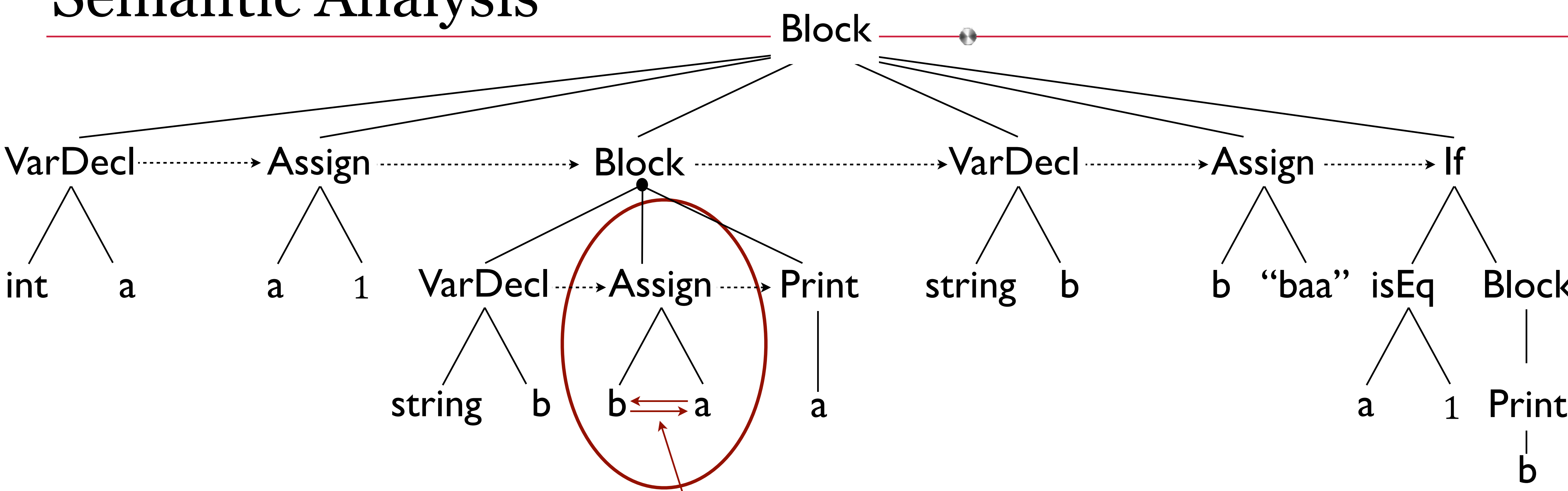
```
{
   int a
   a = 1
   {
      string b
      b = a
      print(a)
   }
   string b
   b = "baa"
   if (a == 1) {
      print(b)
   }
}
```



Symbol Table

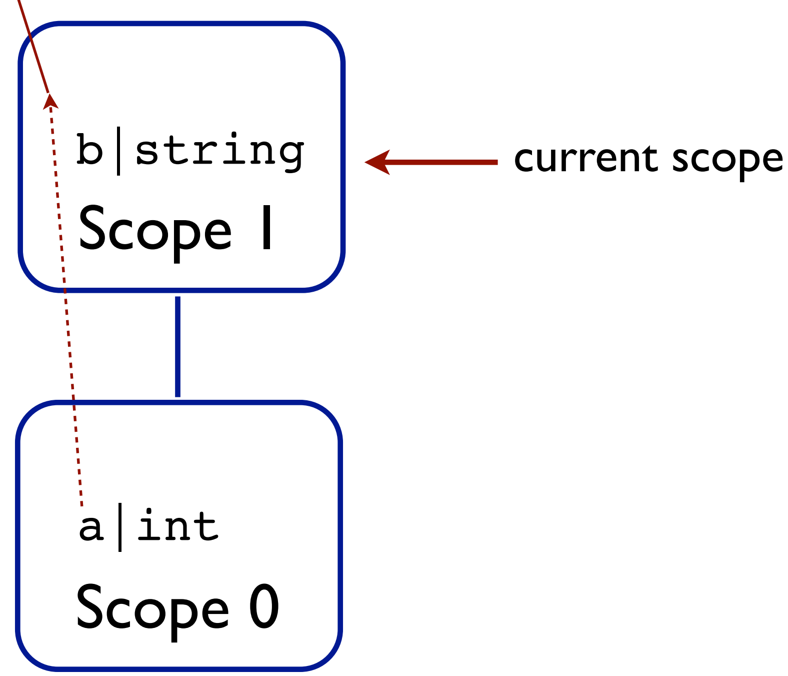
Initialize Scope 0
add symbol a
lookup symbol a
check types
Initialize Scope 1
add symbol b
lookup symbol b
lookup symbol a
in the ***current scope***
It's not there, so…
look in the
parent scope

Semantic Analysis



Source Code

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



Symbol Table

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

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

string ≠ int

Type Mismatch error

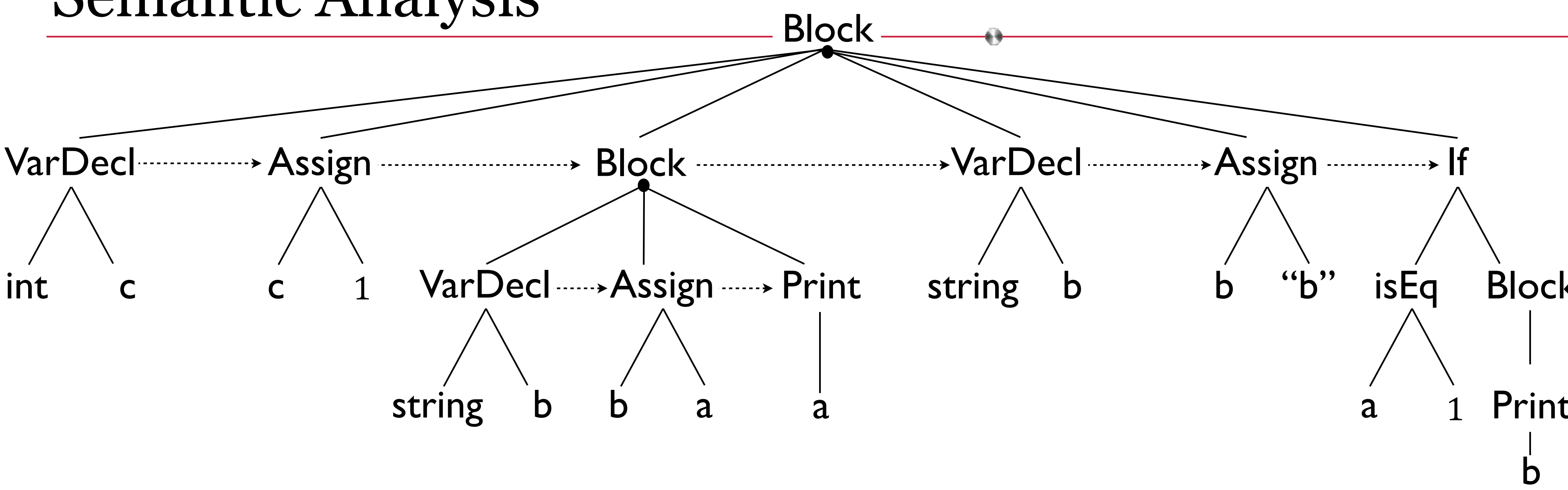
Semantic Analysis

Still another example in our language

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

What's the error here?
Let's see…

Semantic Analysis



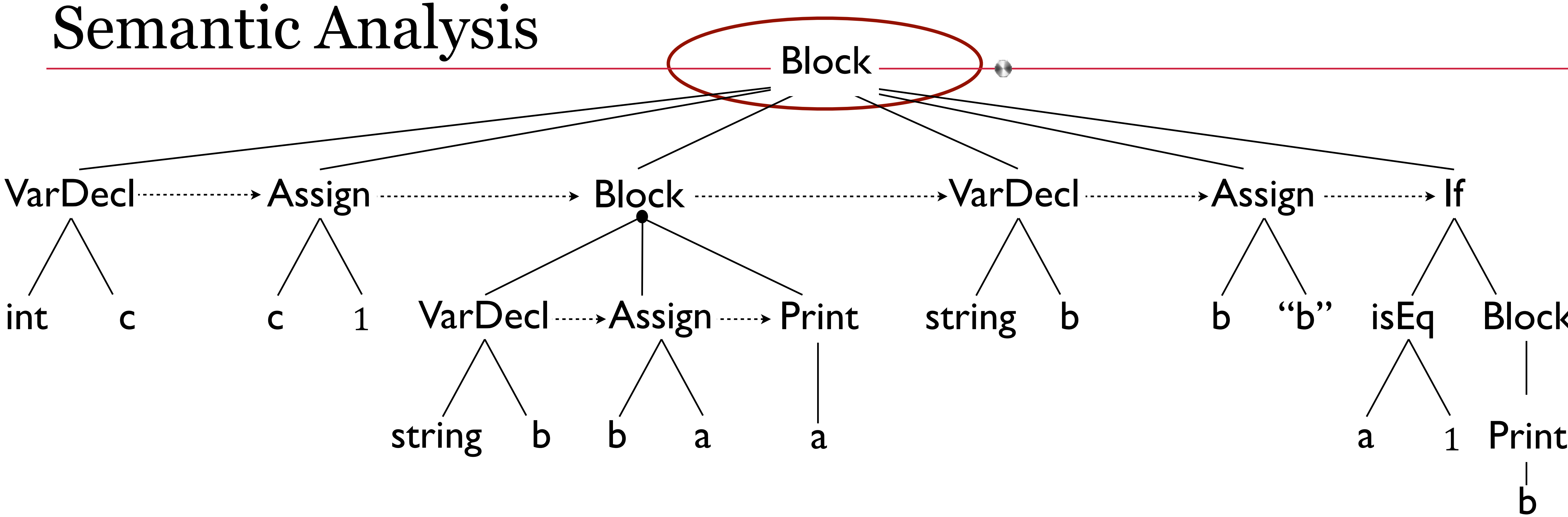
Source Code

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

Symbol Table

Semantic Analysis

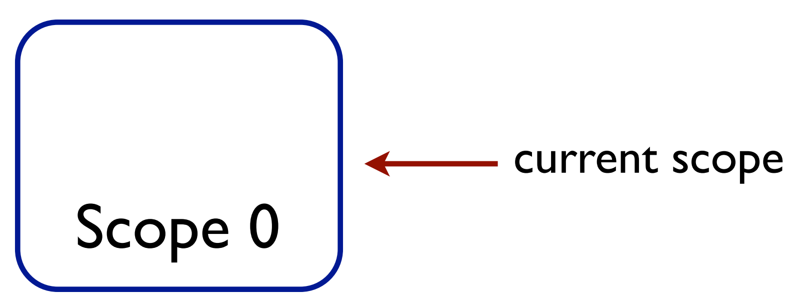
AST



Initialize Scope 0
Set the ***current scope*** *pointer.*

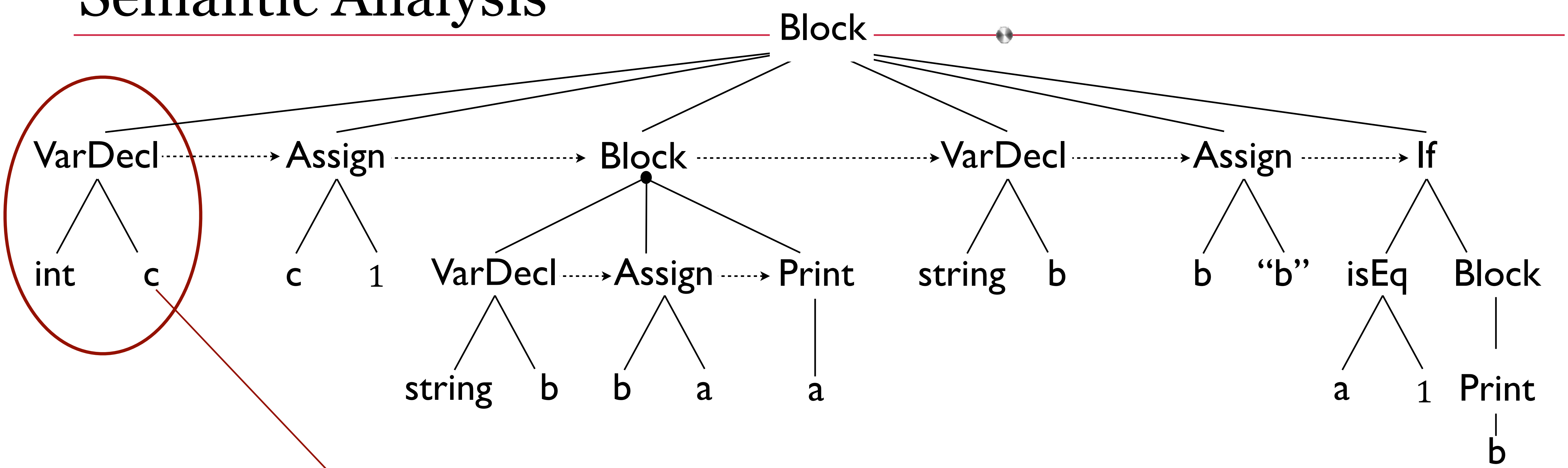
Source Code

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



Symbol Table

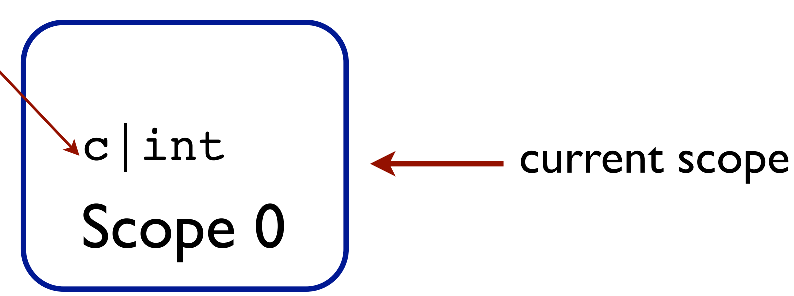
Semantic Analysis



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

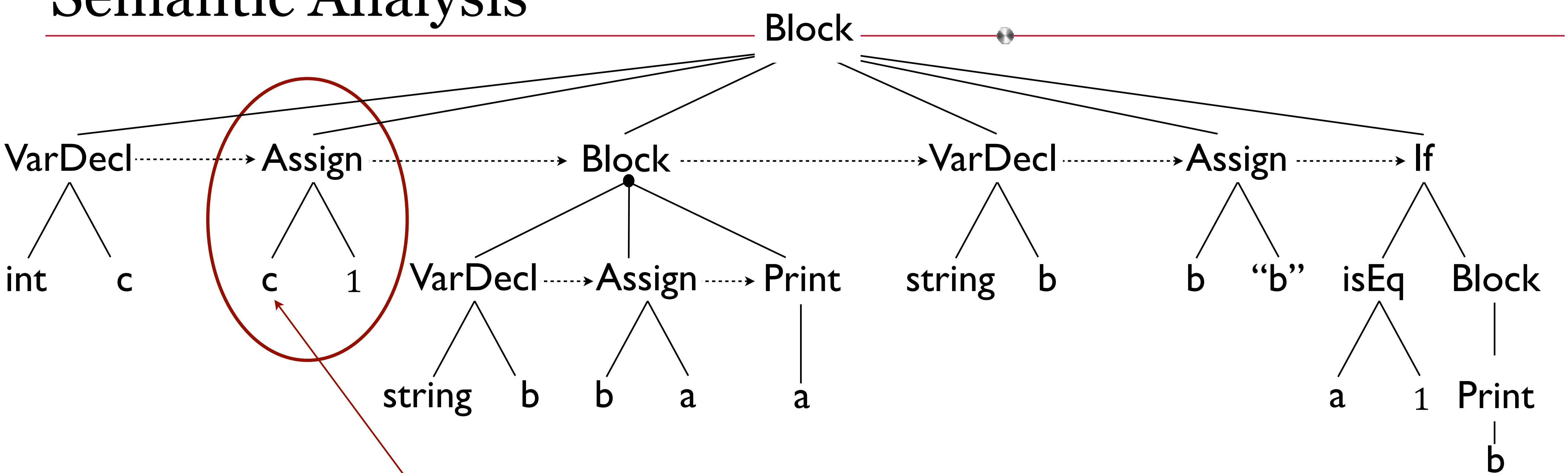
Source Code

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



Symbol Table

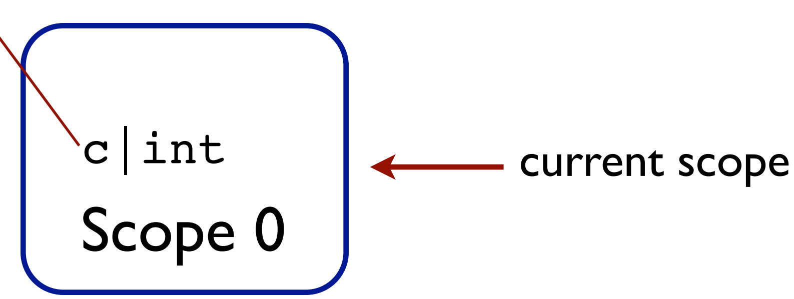
Semantic Analysis



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

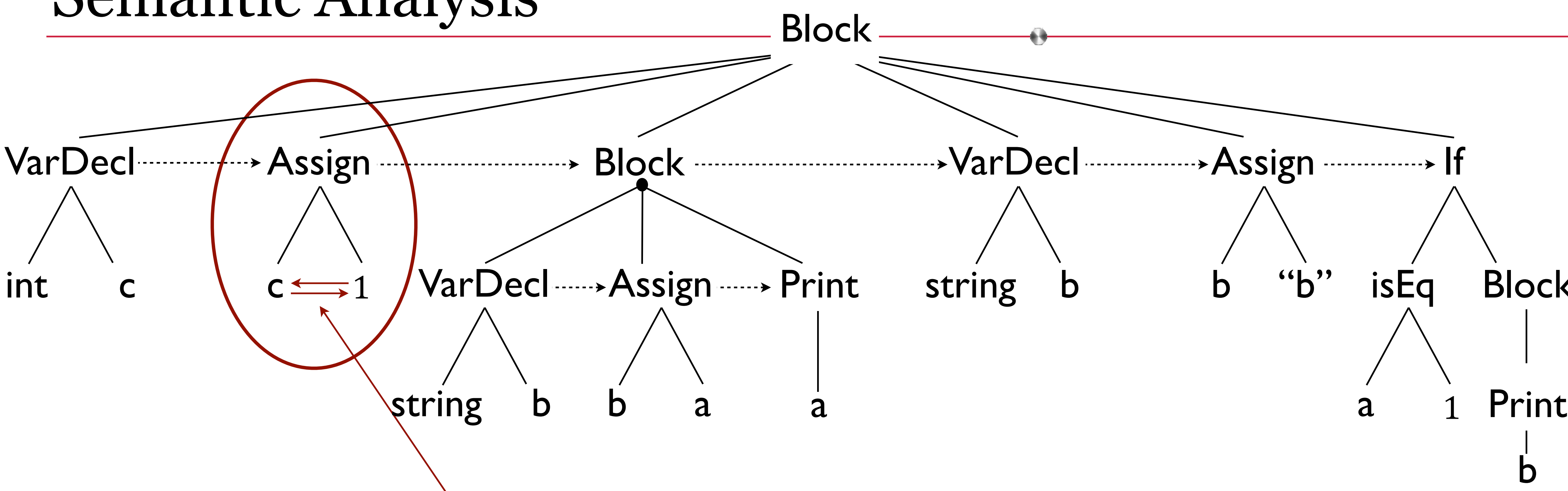
Source Code

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



Symbol Table

Semantic Analysis

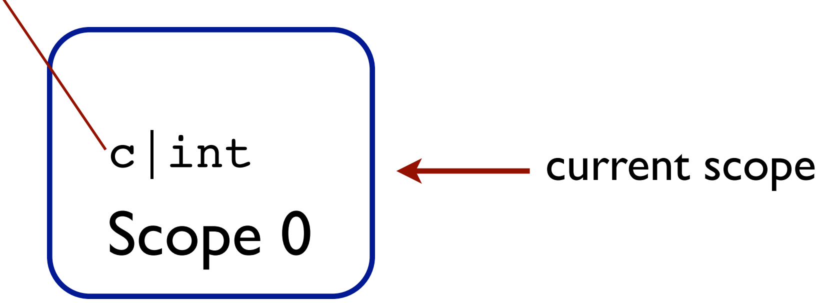


Source Code

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

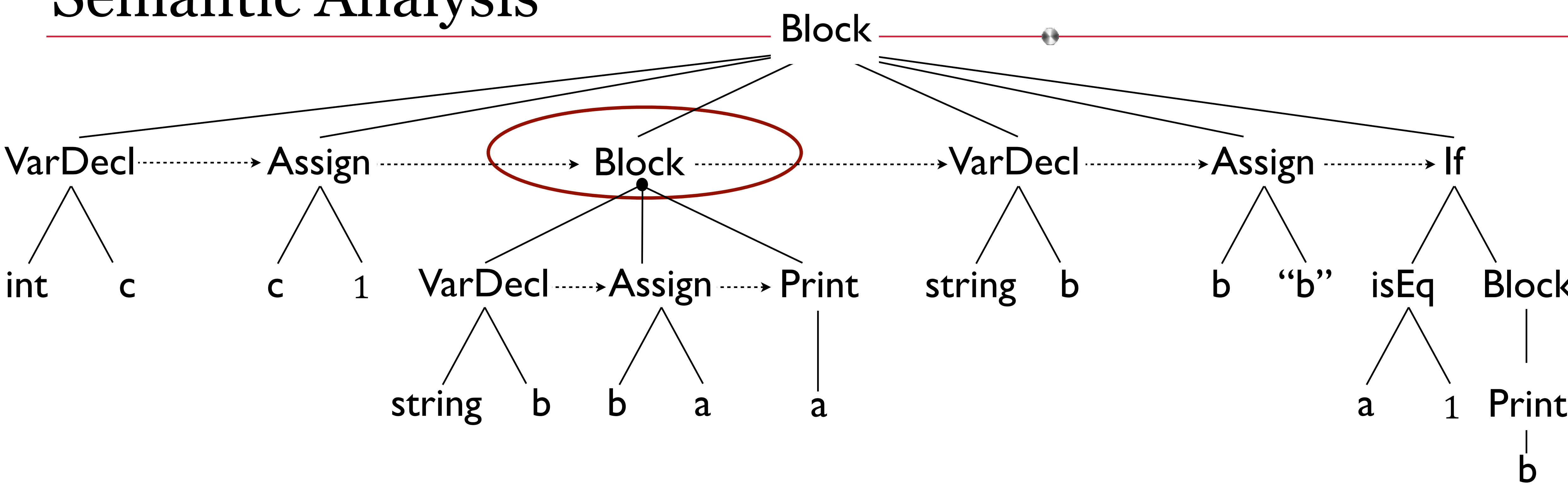
Initialize Scope 0
add symbol c
lookup symbol c
check types

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



Symbol Table

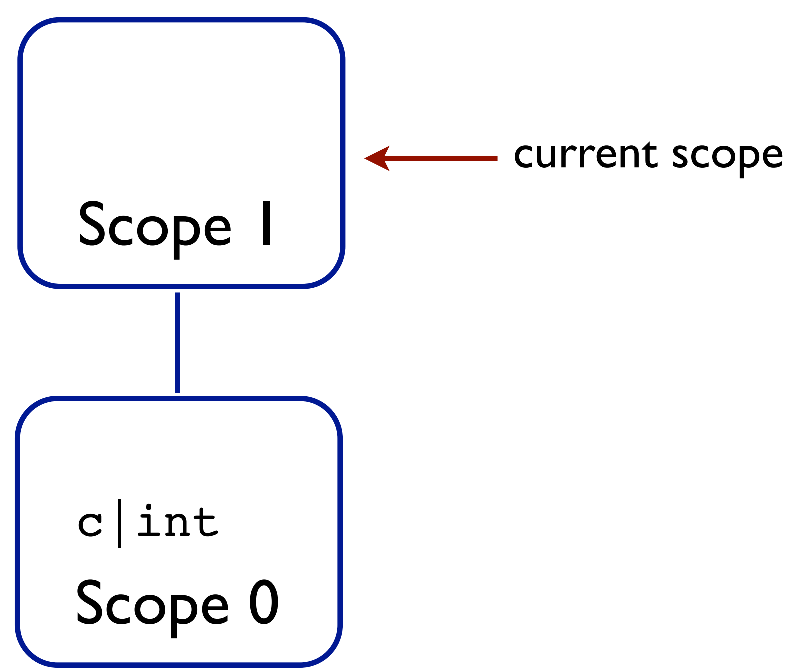
Semantic Analysis



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

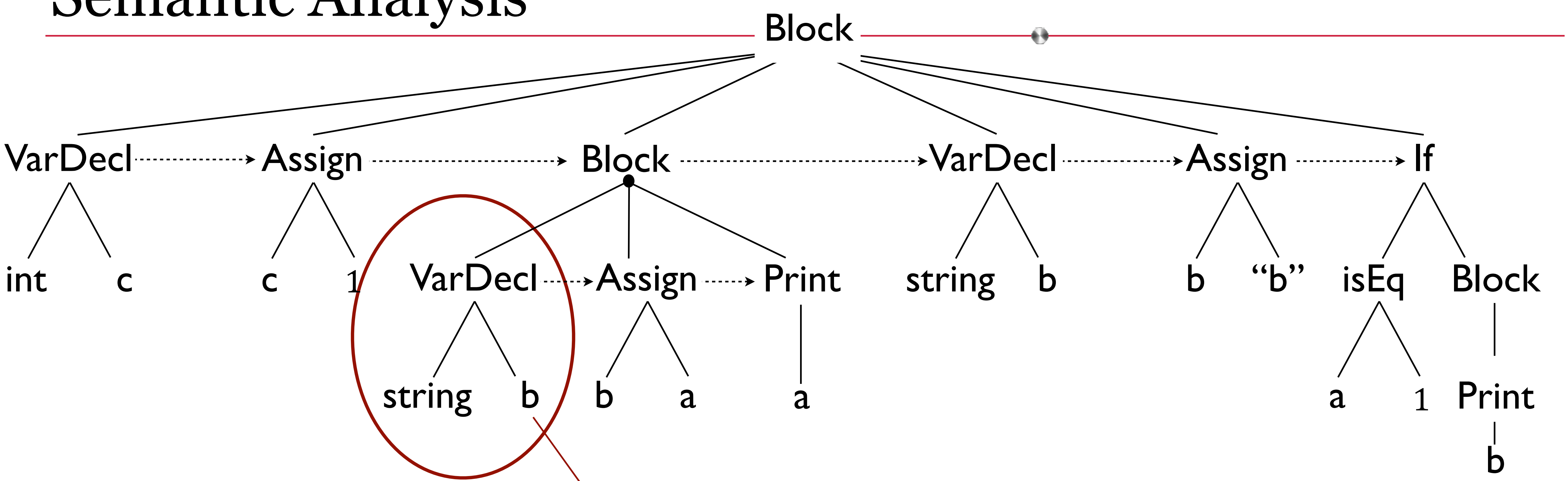
Source Code

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



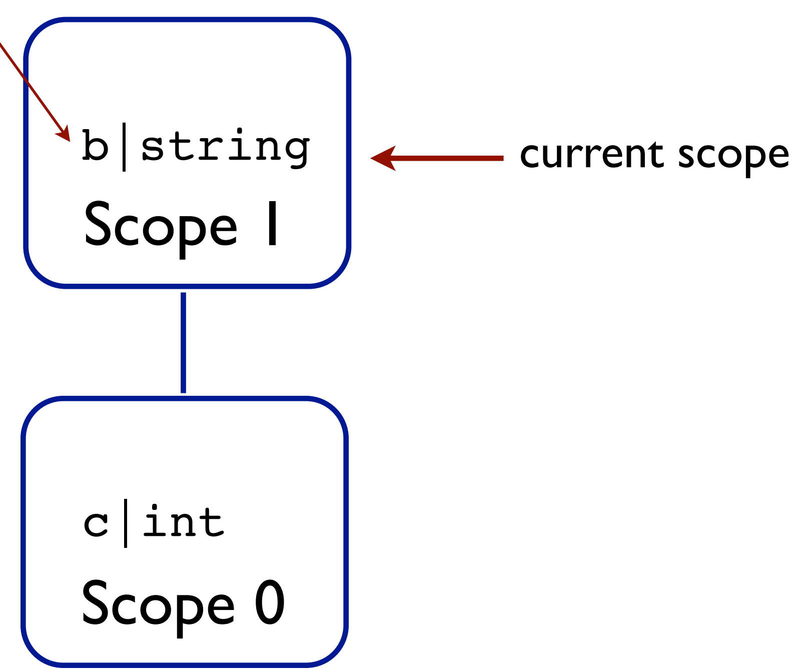
Symbol Table

Semantic Analysis



Source Code

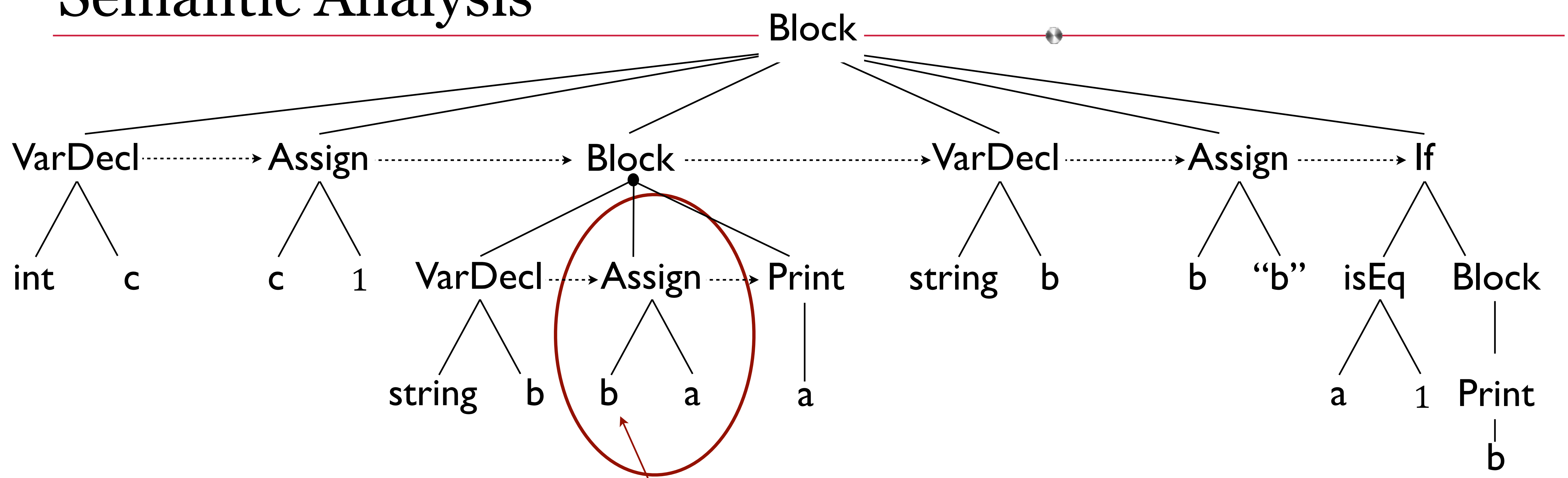
```
{
   int c
   c = 1
   {
      string b
      b = a
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```



Symbol Table

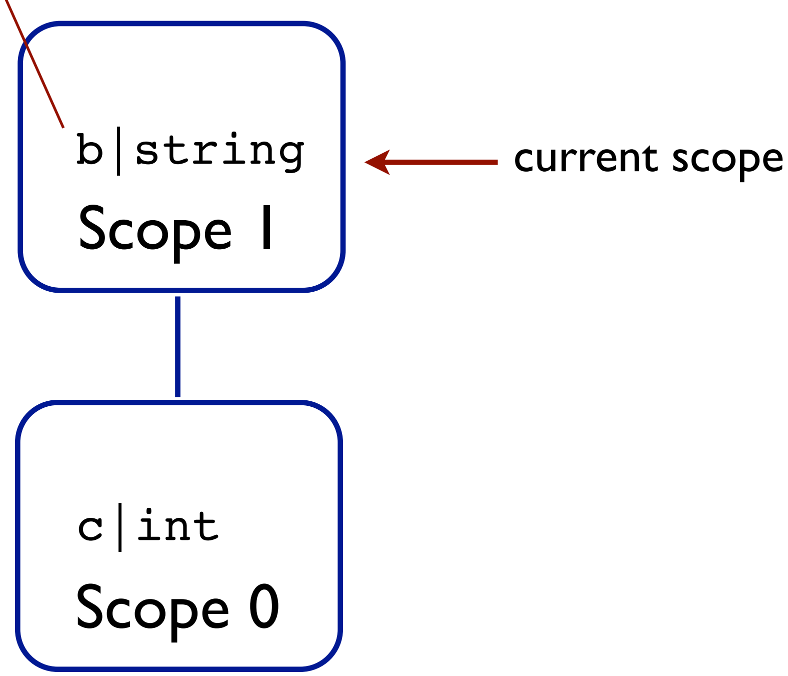
Initialize Scope 0
add symbol c
lookup symbol c
check types
Initialize Scope 1
add symbol b
*in the **current scope***

Semantic Analysis



Source Code

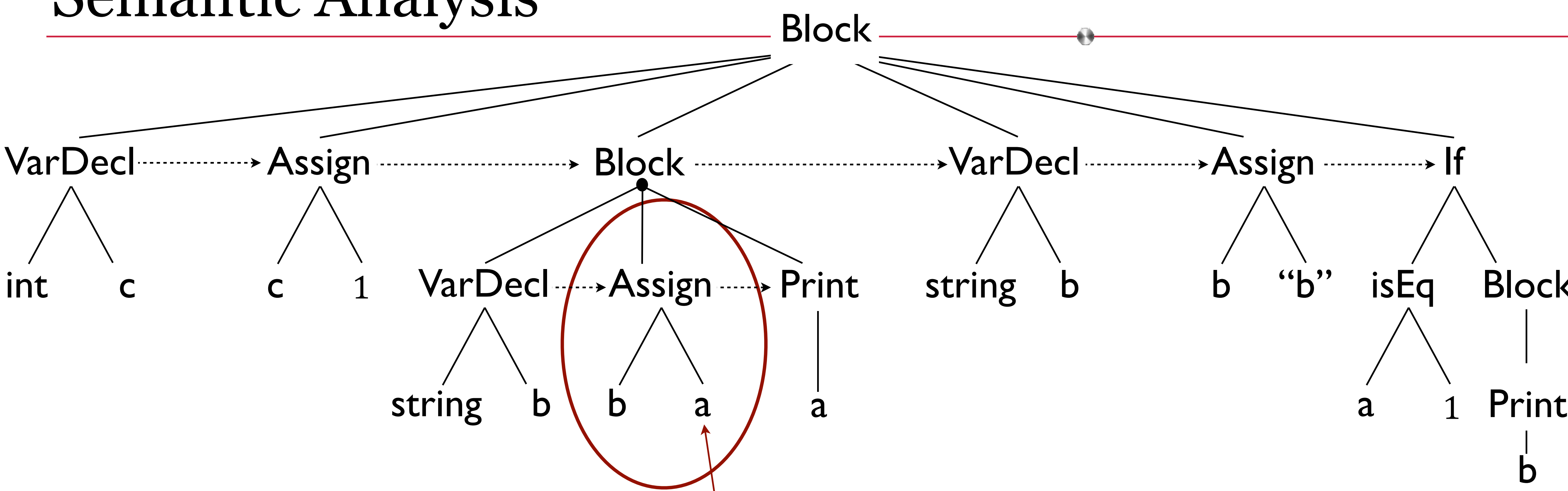
```
{
   int c
   c = 1
   {
      string b
      b = a
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```



Symbol Table

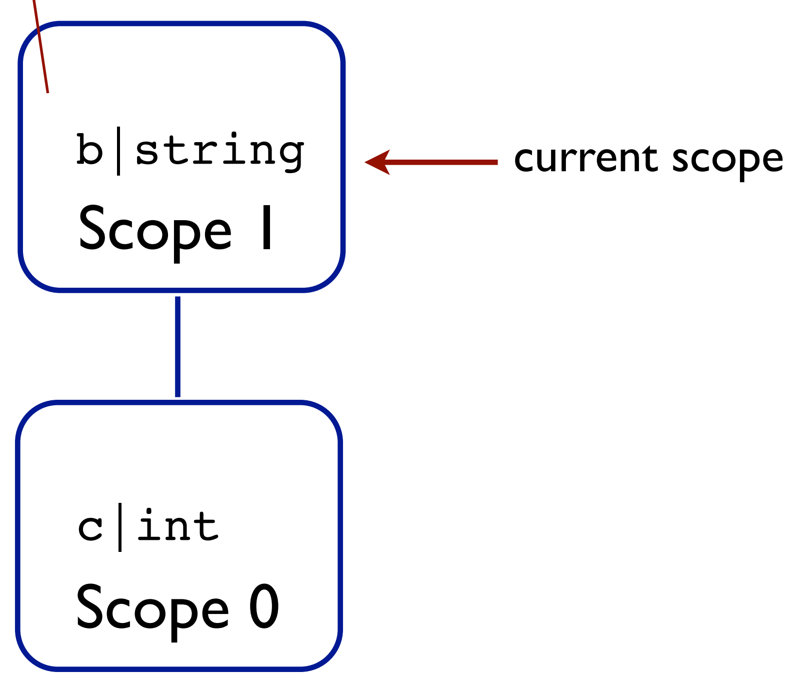
Initialize Scope 0
add symbol c
lookup symbol c
check types
Initialize Scope 1
add symbol b
lookup symbol b
*in the **current scope***

Semantic Analysis



Source Code

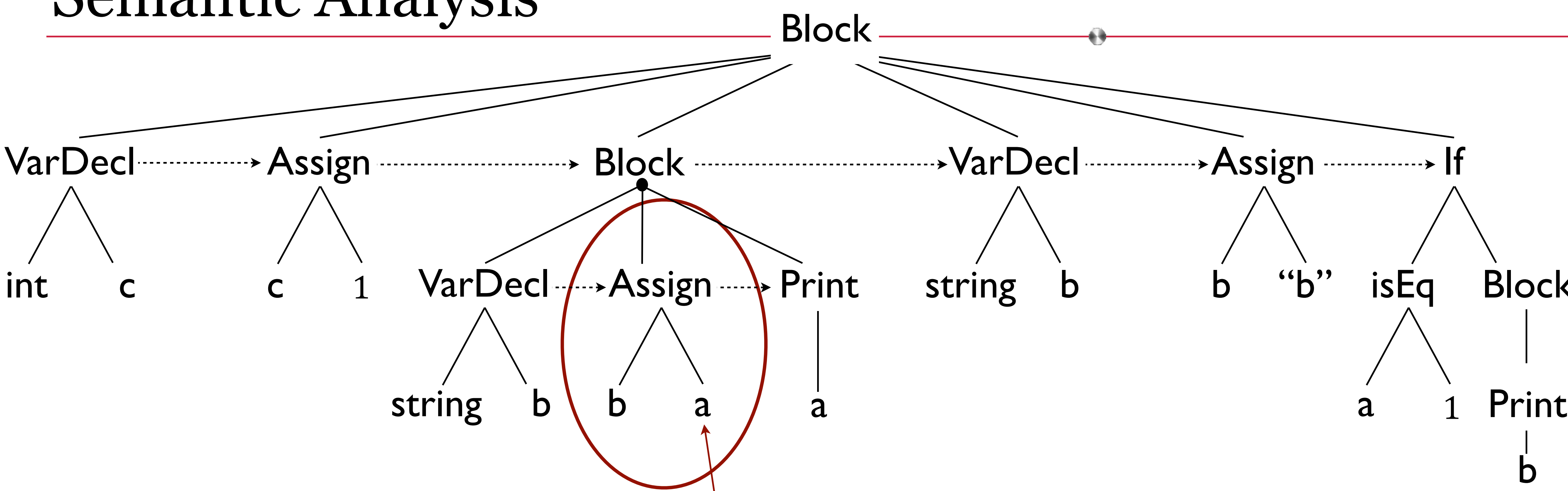
```
{
   int c
   c = 1
   {
      string b
      b = a
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```



Symbol Table

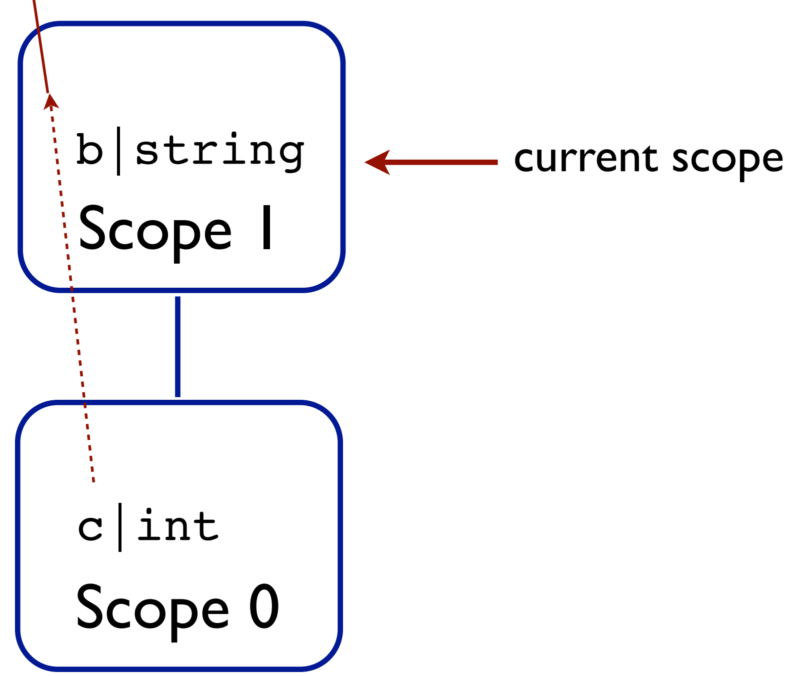
Initialize Scope 0
add symbol c
lookup symbol c
check types
Initialize Scope 1
add symbol b
lookup symbol b
lookup symbol a
in the ***current scope***
It's not there, so…

Semantic Analysis



Source Code

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



Symbol Table

Initialize Scope 0
add symbol c
lookup symbol c
check types
Initialize Scope 1
add symbol b
lookup symbol b
lookup symbol a
in the ***current scope***
It's not there, so…
look in the
parent scope
It's not there either!

Undeclared Identifier error

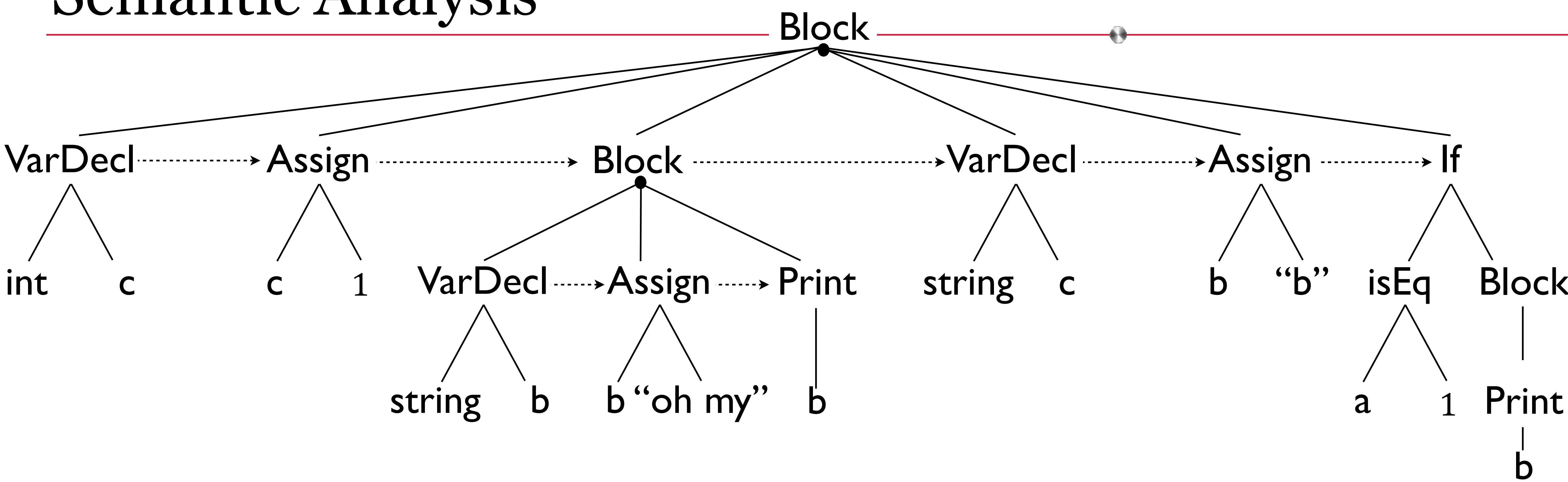
Semantic Analysis

One more example in our language

```
{
   int c
   c = 1
   {
      string b
      b = “oh my”
      print(b)
   }
   string c
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```

Where are the errors in this one?
Let’s see…

Semantic Analysis



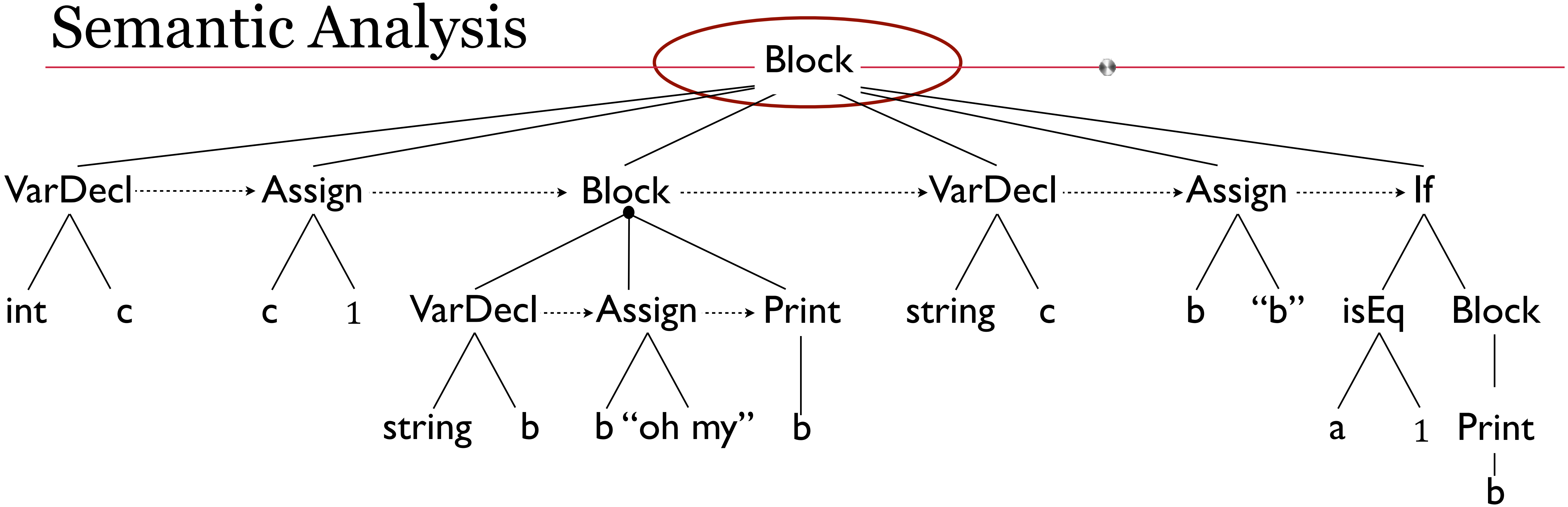
Source Code

```
{
   int c
   c = 1
   {
      string b
      b = “oh my”
      print(b)
   }
   string c
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```

Symbol Table

Semantic Analysis

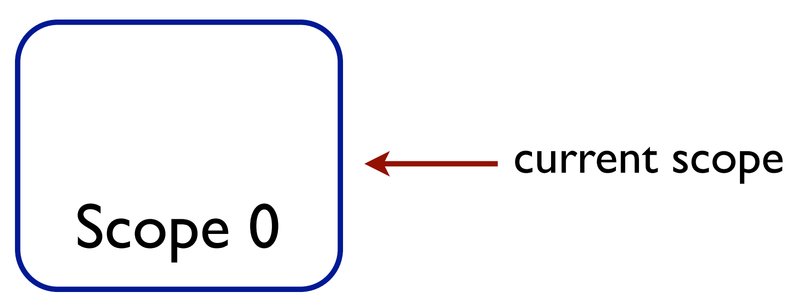
AST



Initialize Scope 0
Set the ***current scope*** *pointer.*

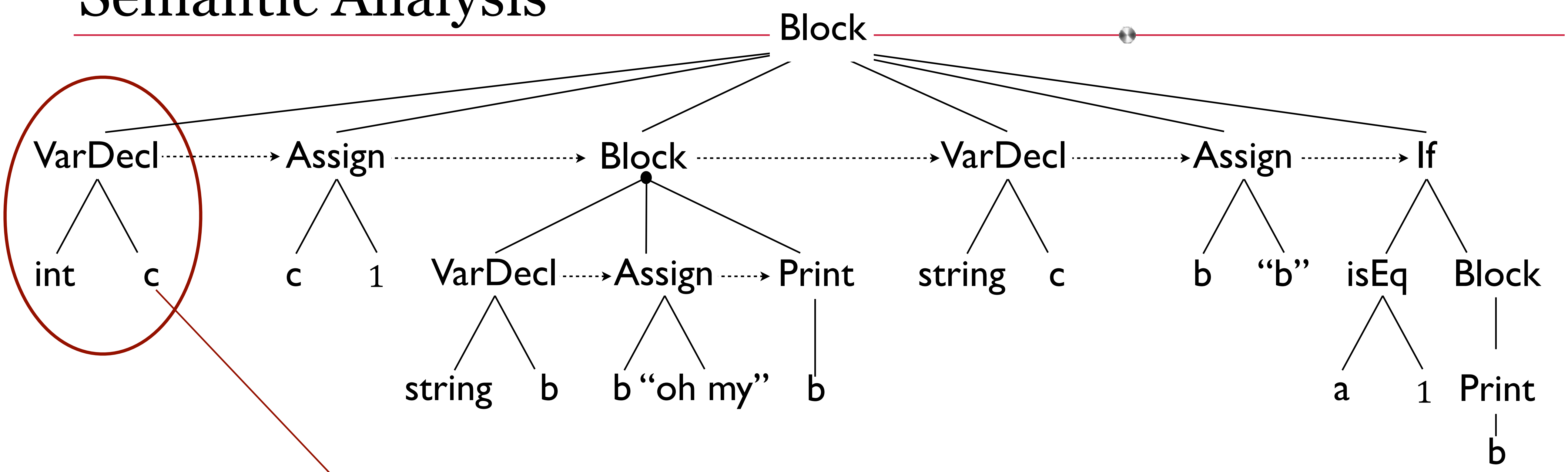
Source Code

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



Symbol Table

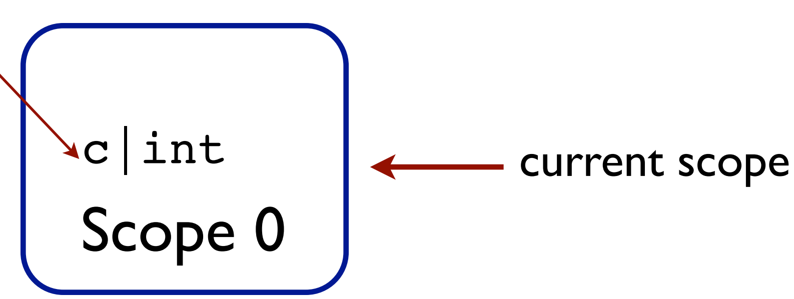
Semantic Analysis



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

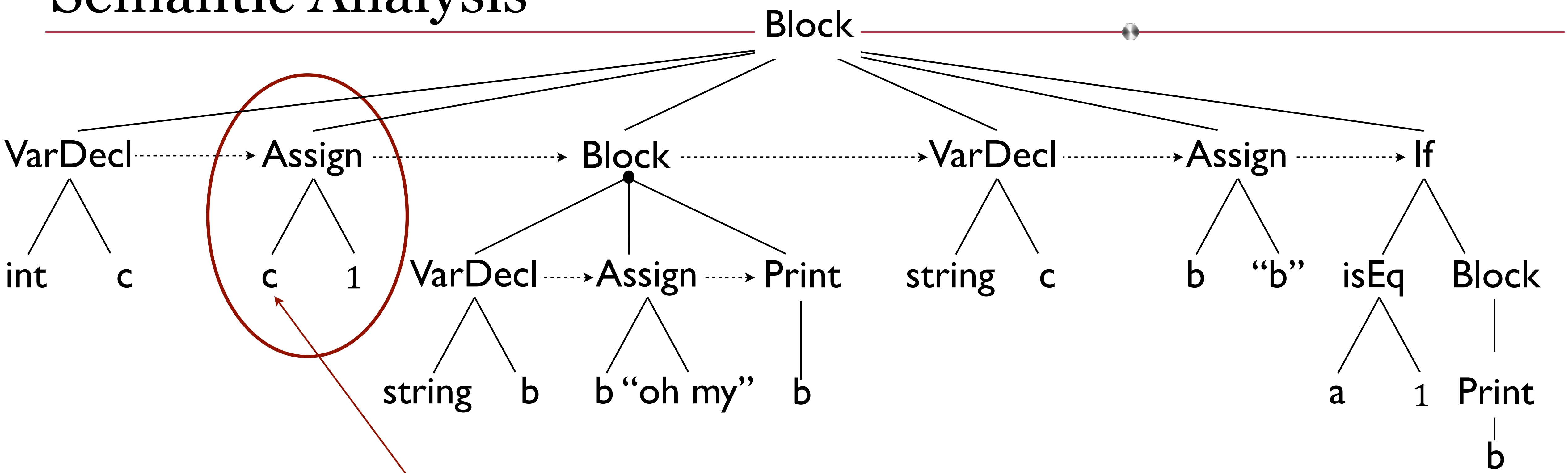
Source Code

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



Symbol Table

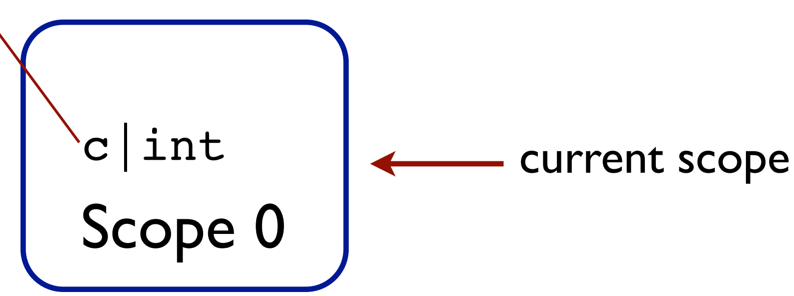
Semantic Analysis



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

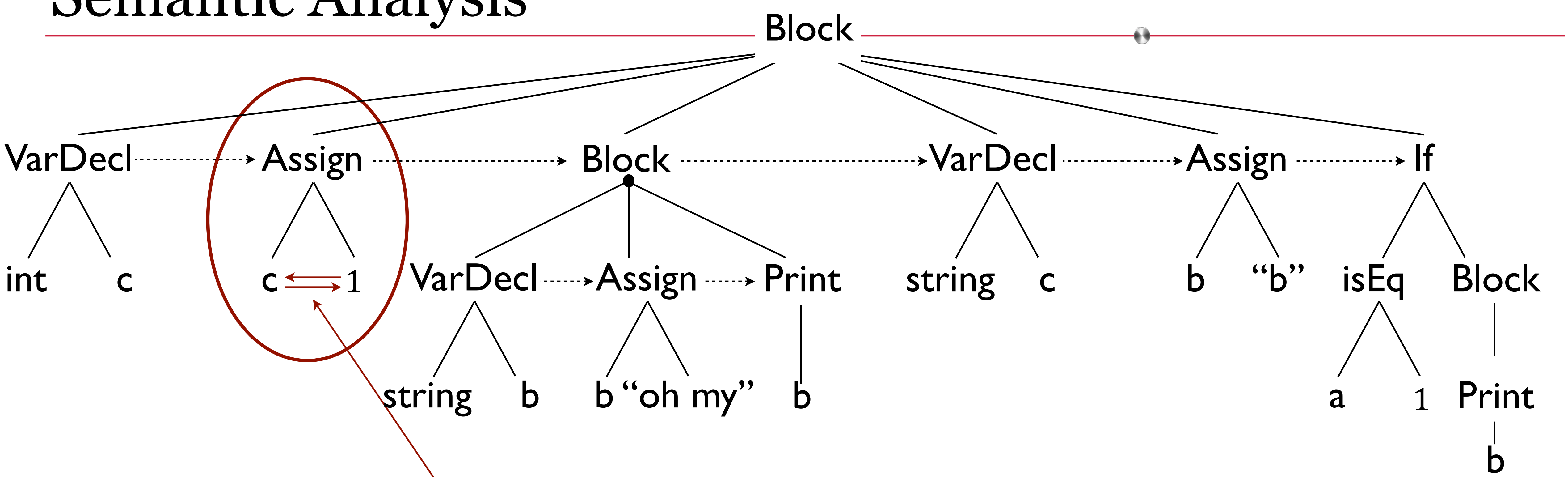
Source Code

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



Symbol Table

Semantic Analysis

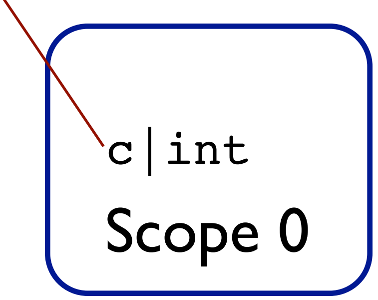


Initialize Scope 0
add symbol c
lookup symbol c
check types

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

Source Code

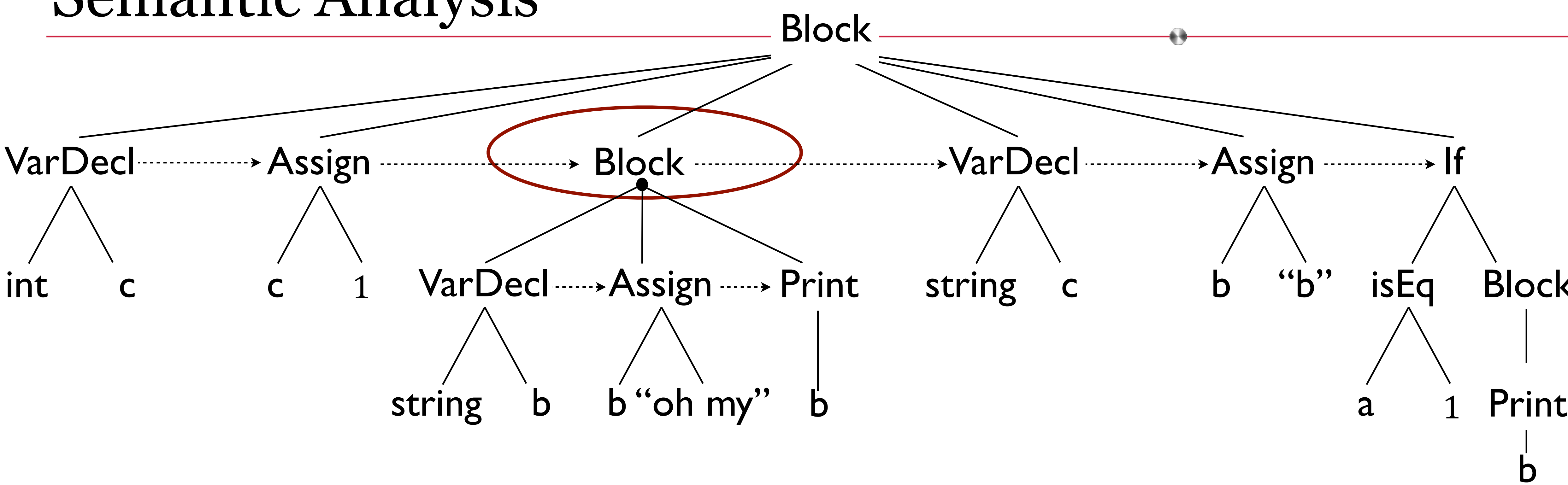
```
{
   int c
   c = 1
   {
      string b
      b = “oh my”
      print(b)
   }
   string c
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```



current scope

Symbol Table

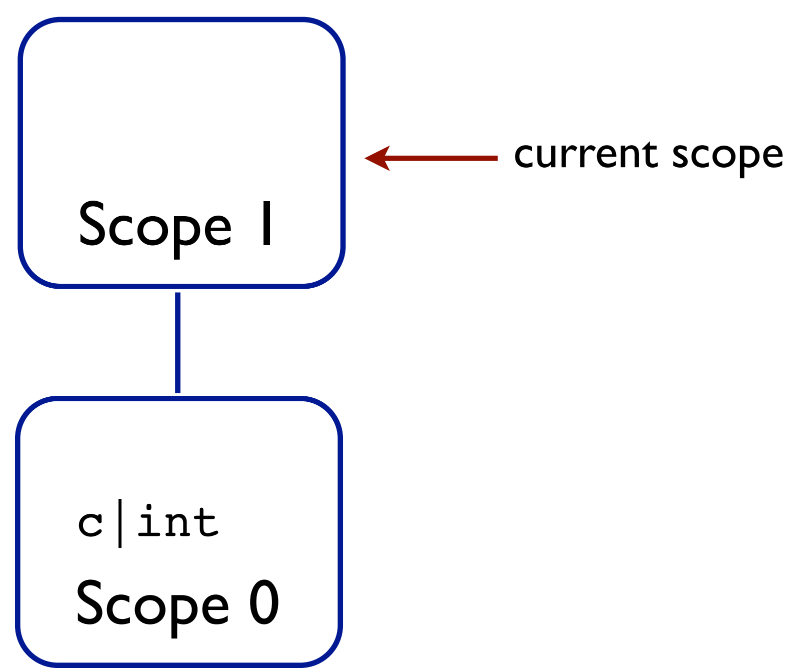
Semantic Analysis



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

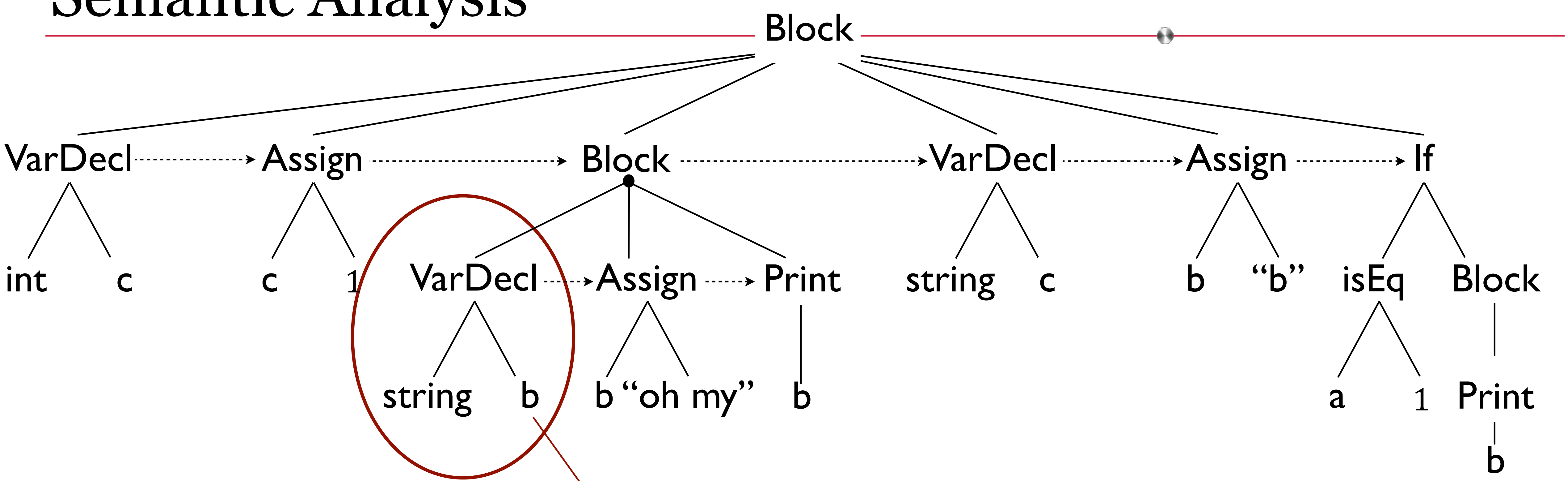
Source Code

```
{
    int c
    c = 1
    {
        string b
        b = “oh my”
        print(b)
    }
    string c
    b = "b"
    if (a == 1) {
        print(b)
    }
}
```



Symbol Table

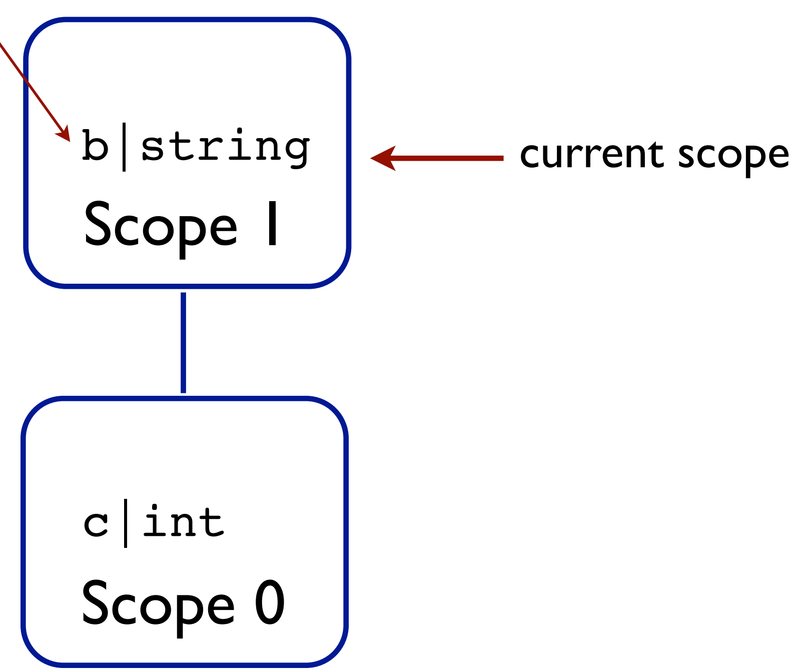
Semantic Analysis



Source Code

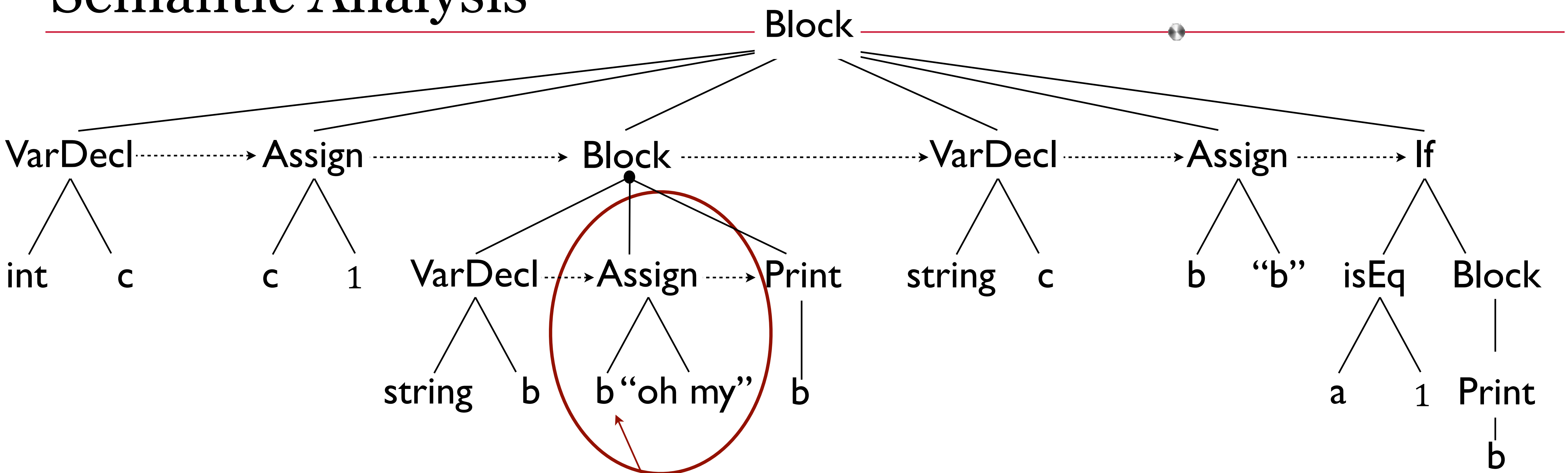
```
{
    int c
    c = 1
    {
      string b
      b = "oh my"
      print(b)
    }
    string c
    b = "b"
    if (a == 1) {
      print(b)
    }
}
```

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



Symbol Table

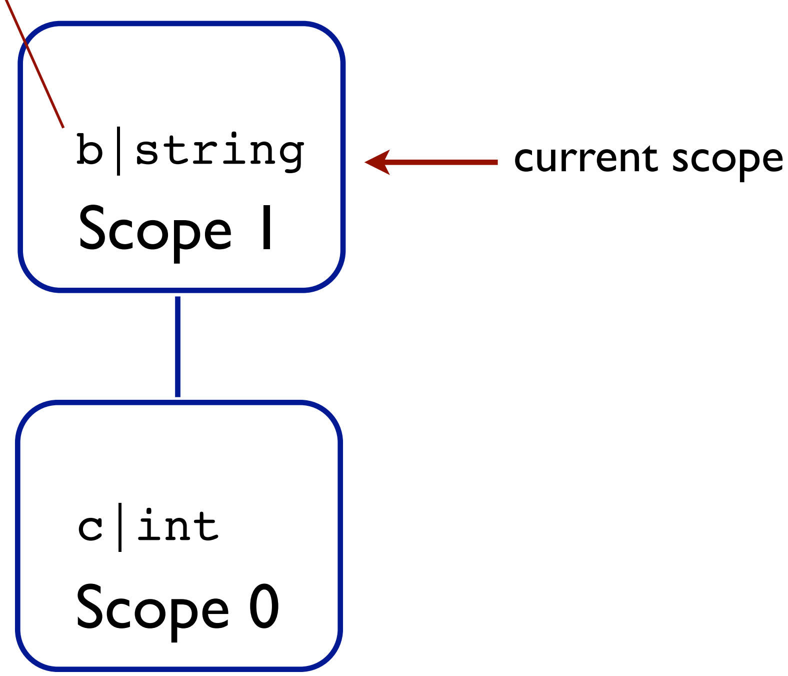
Semantic Analysis



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

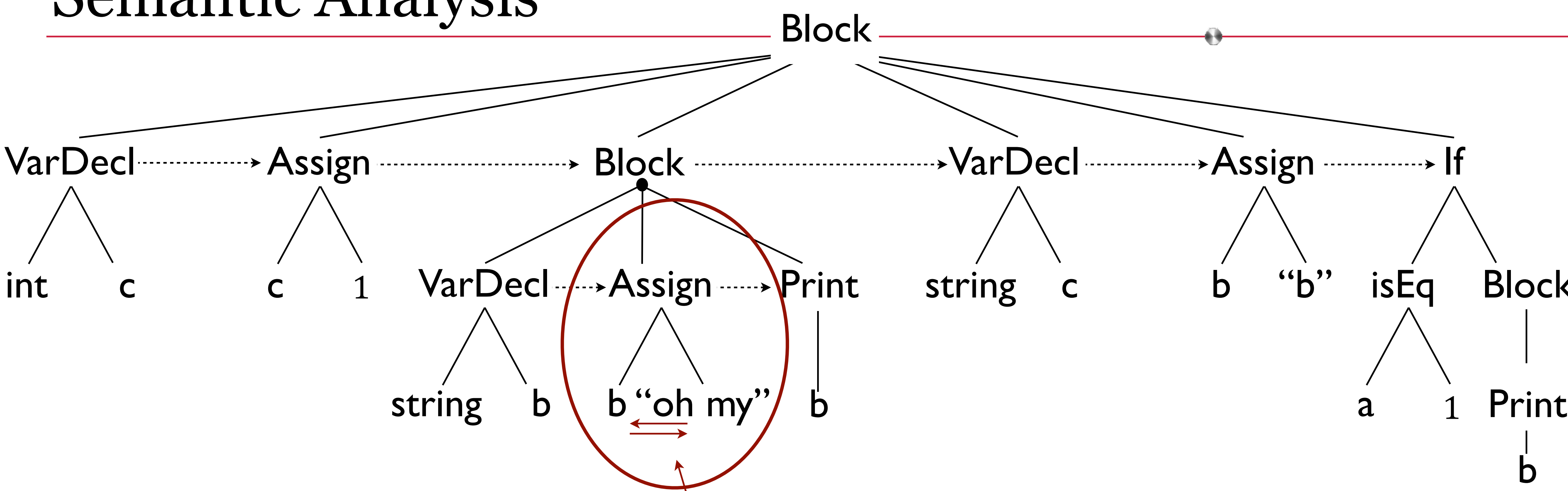
Source Code

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



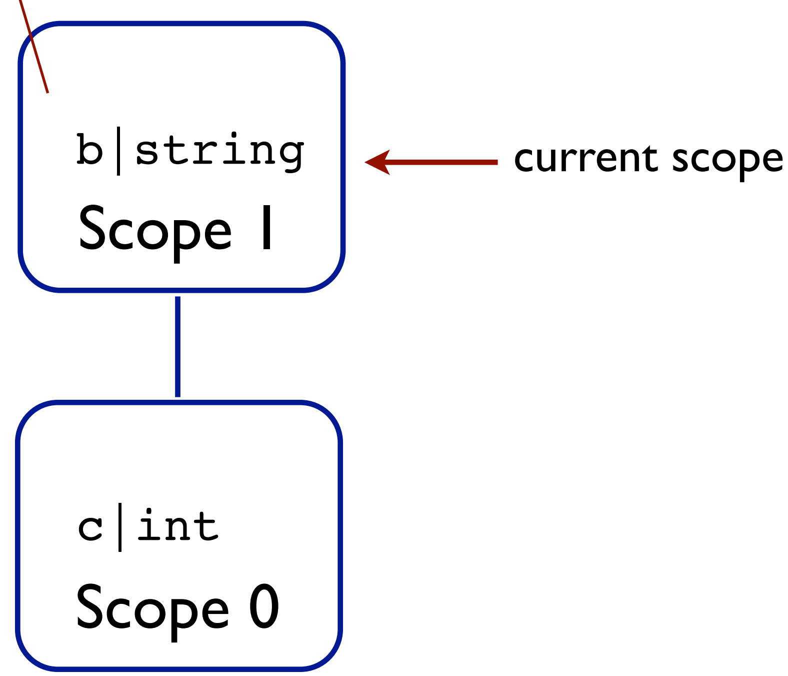
Symbol Table

Semantic Analysis



Source Code

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

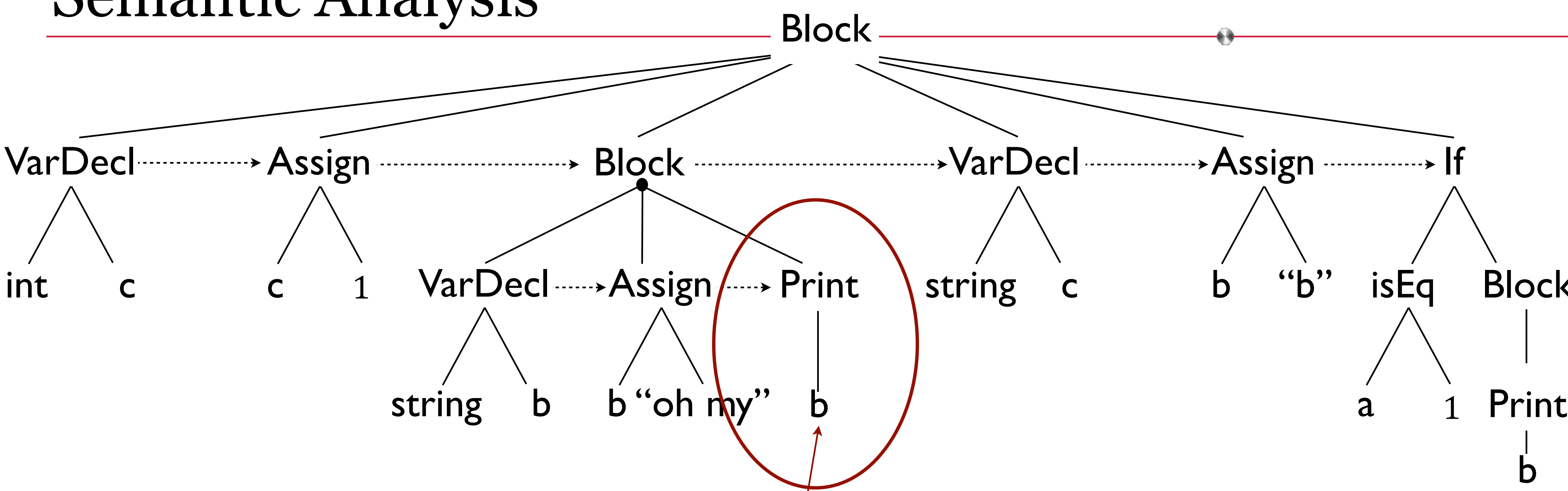


Symbol Table

Initialize Scope 0
add symbol c
lookup symbol c
check types
Initialize Scope 1
add symbol b
lookup symbol b
check types

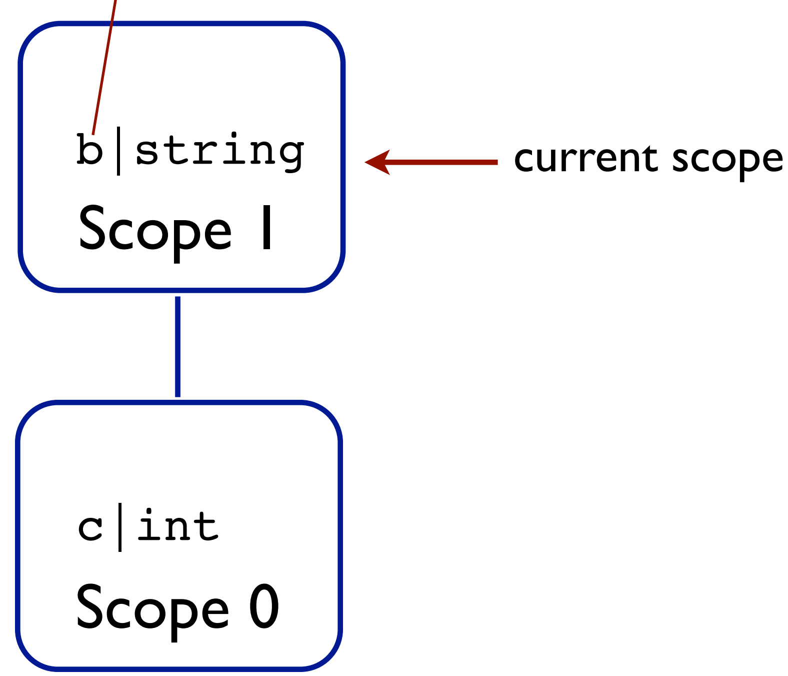
Verify that the left child and right child are type compatible for assignment.

Semantic Analysis



Source Code

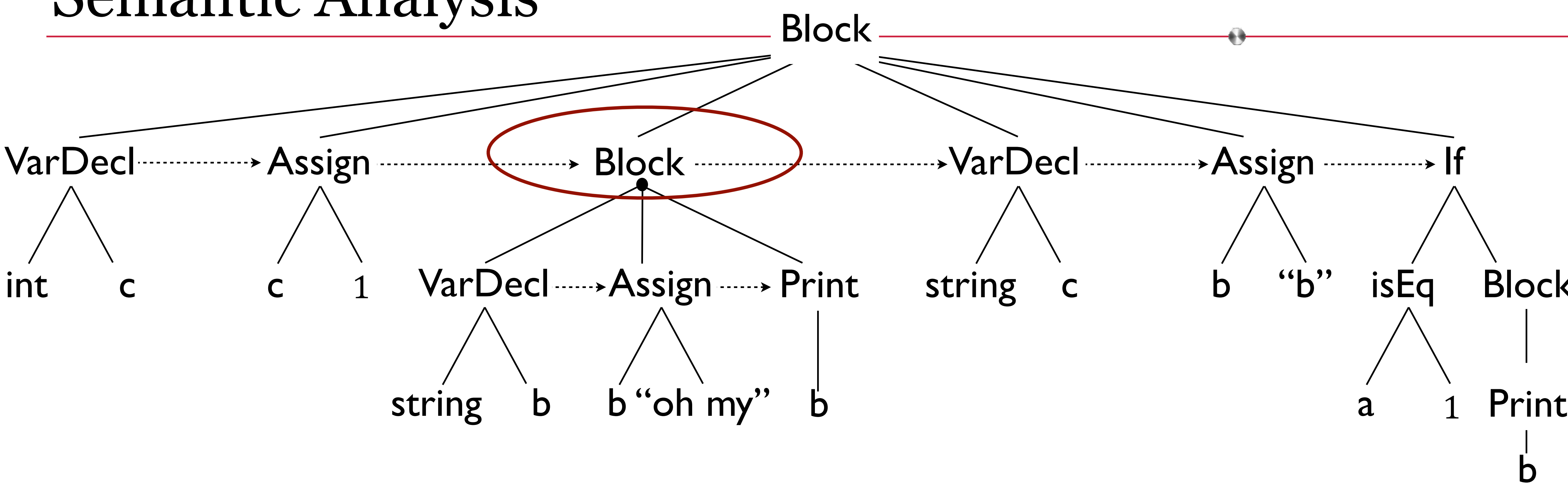
```
{
   int c
   c = 1
   {
      string b
      b = “oh my”
      print(b)
   }
   string c
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```



Symbol Table

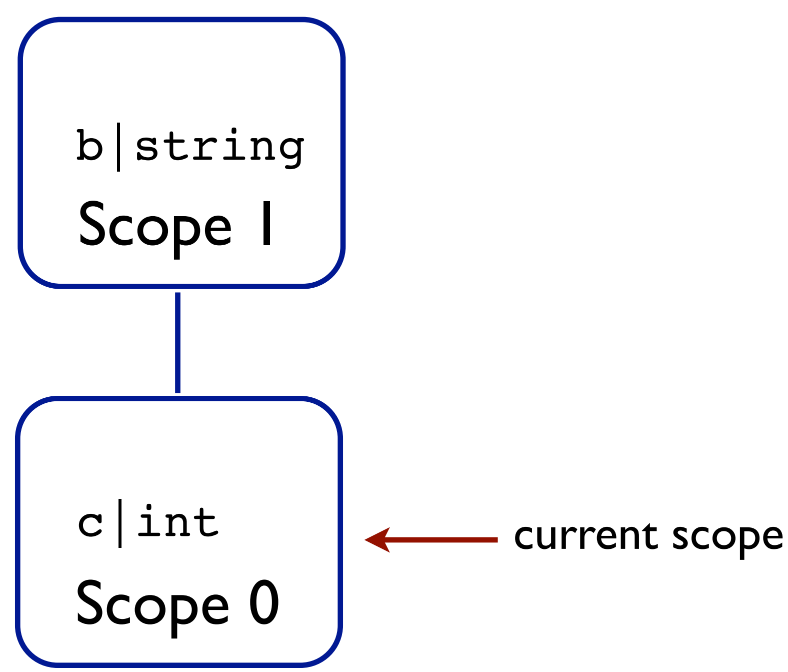
Initialize Scope 0
add symbol c
lookup symbol c
check types
Initialize Scope 1
add symbol b
lookup symbol b
check types
lookup symbol b
*in the **current scope**.*
Print can take any type, so there’s no need to type check here.
We must still check the scope, of course!

Semantic Analysis



Source Code

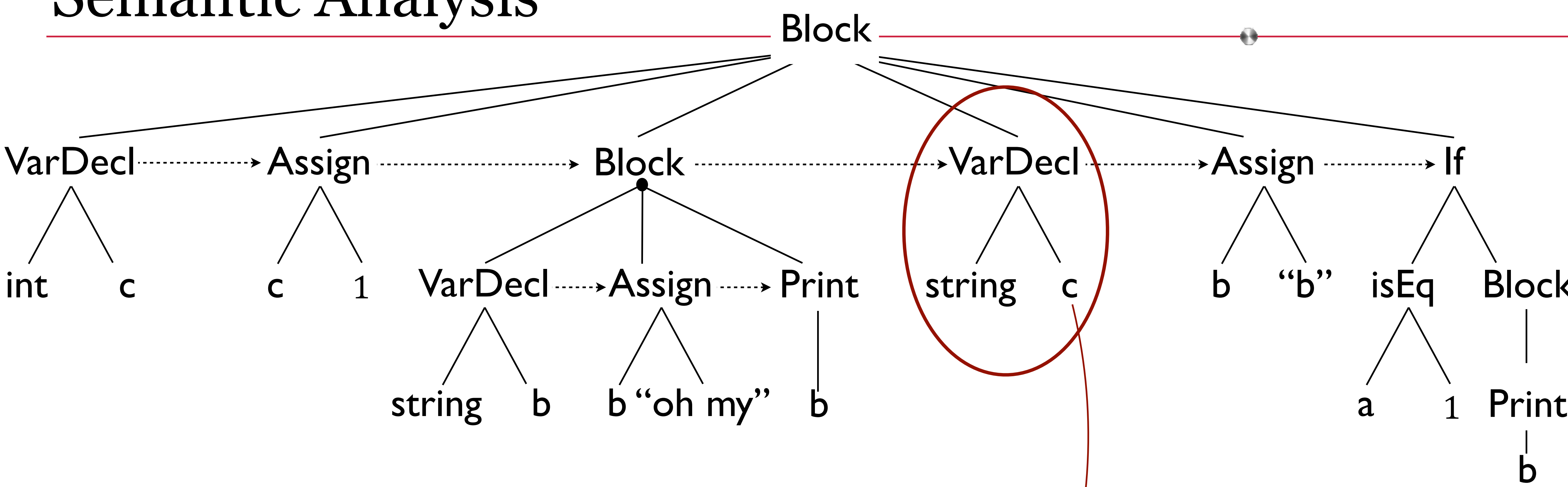
```
{
   int c
   c = 1
   {
      string b
      b = “oh my”
      print(b)
   }
   string c
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```



Symbol Table

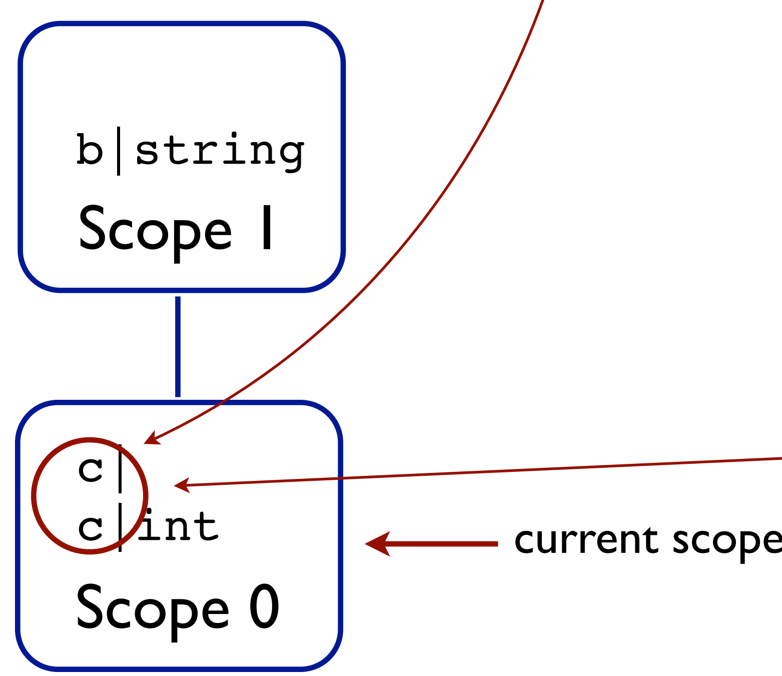
Initialize Scope 0
add symbol c
lookup symbol c
check types
Initialize Scope 1
add symbol b
lookup symbol b
check types
lookup symbol b
Close Scope 1
*Move the **current scope** pointer to its parent.*

Semantic Analysis



Source Code

```
{
   int c
   c = 1
   {
      string b
      b = “oh my”
      print(b)
   }
   string c
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```



Initialize Scope 0
add symbol c
lookup symbol c
check types
Initialize Scope 1
add symbol b
lookup symbol b
check types
lookup symbol b
Close Scope 1
add symbol c
*in the **current scope***
Fail!
Collision in the hash table.

Identifier redeclared in the same scope error

Symbol Table

Semantic Analysis

One more example in our language

```
{
   int c
   c = 1
   {
      string b
      b = “oh my”
      print(b)
   }
   string c
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```

Identifier redeclared in the same scope error.

What else is wrong here?

Semantic Analysis

Okay, this is really the last one.

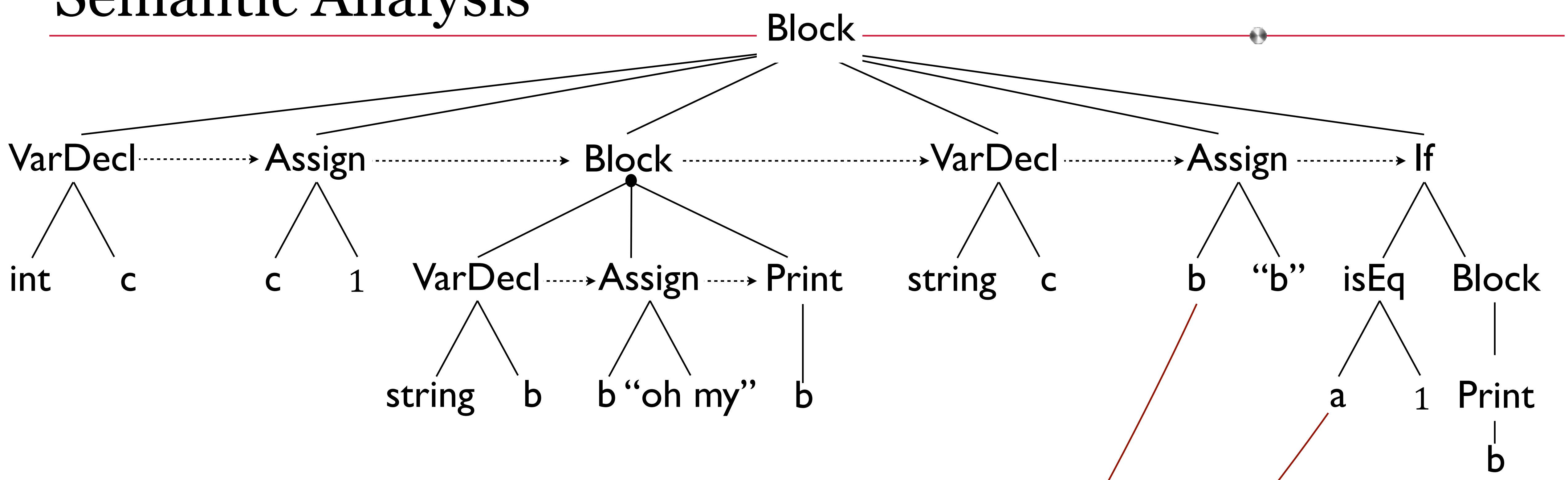
```
{
   int a
   a = 7
   string b
   b = " bstring "
   {
      print(a)
      print(b)
      b = " outer b is still a string "
      int b
      b = 0
      print(b)
      b = a
      print(b)
   }
   print(b)
}$
```

Same-named identifier declared in different scopes but referenced in the same block.

Who **does** this?

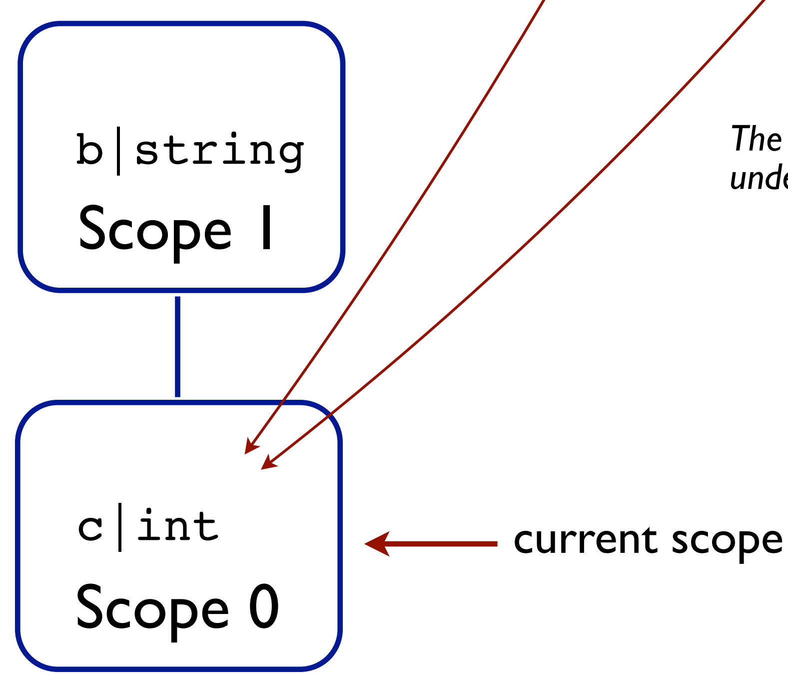
What should we do here?

Semantic Analysis



Source Code

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



The identifiers a and b are undeclared in this scope.

Symbol Table

Semantic Analysis

We can detect errors.

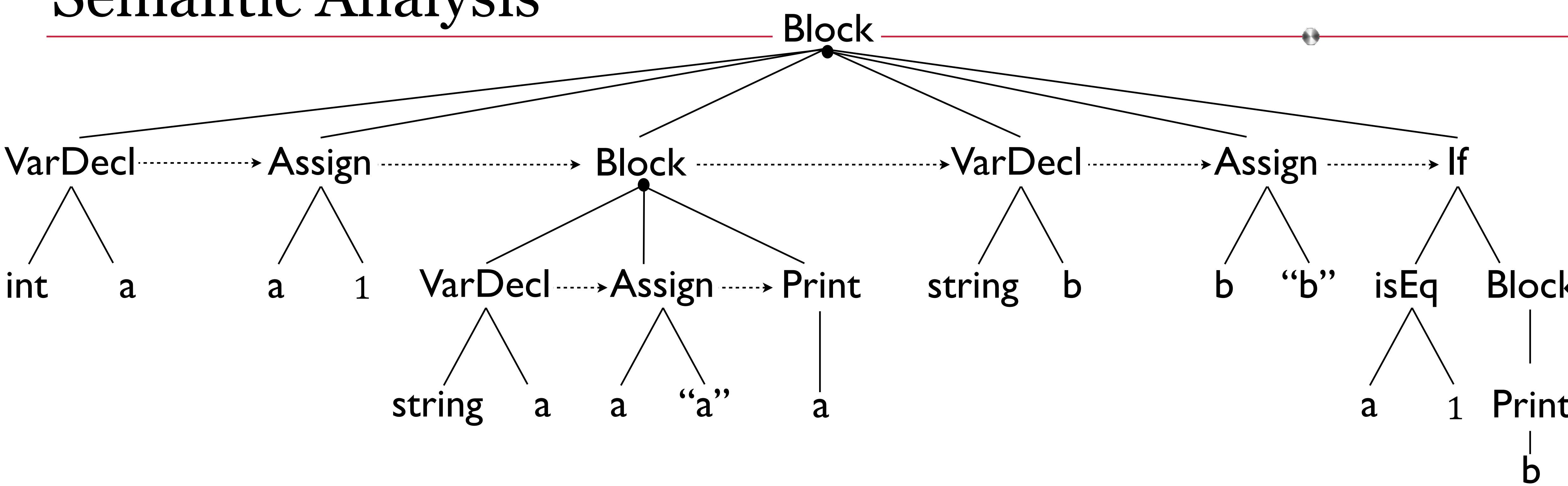
- Type mismatches
- Undeclared identifiers
- Redeclared identifiers in the same scope

What about warnings?

- Variables declared but unused
- Variables used without being initialized
- Variables declared and initialized but never used elsewhere

How do we handle those?

Semantic Analysis



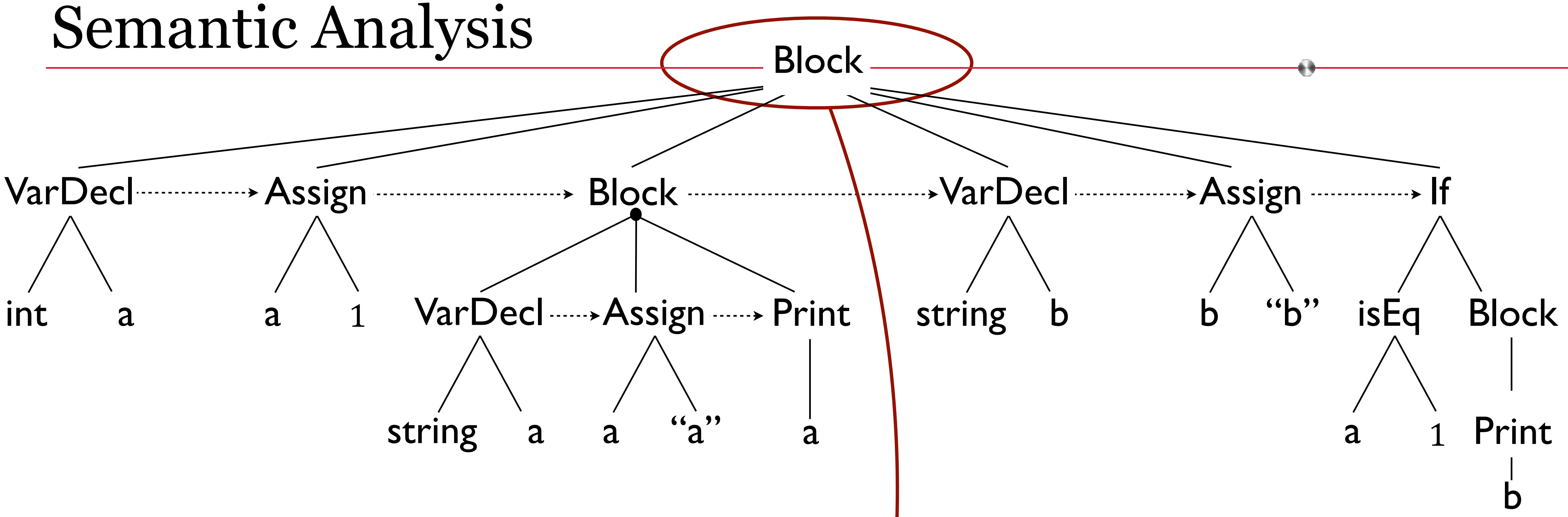
Source Code

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

Symbol Table

Semantic Analysis

AST

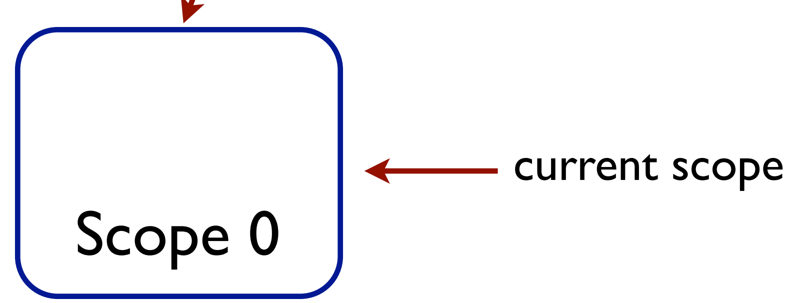


Initialize Scope 0

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

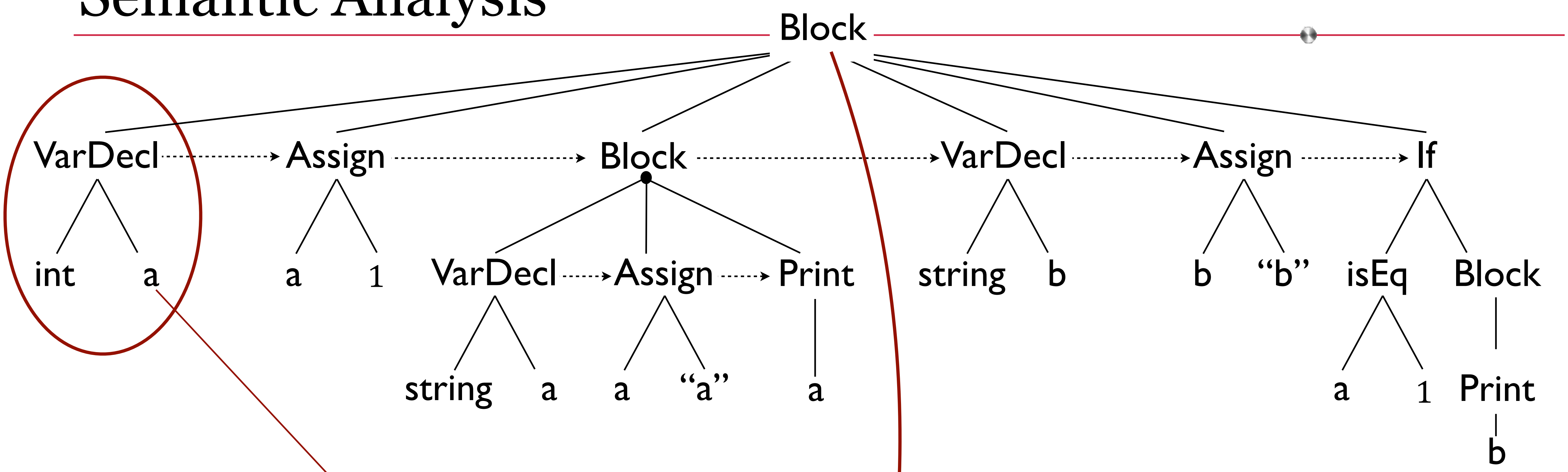
Source Code

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



Symbol Table

Semantic Analysis

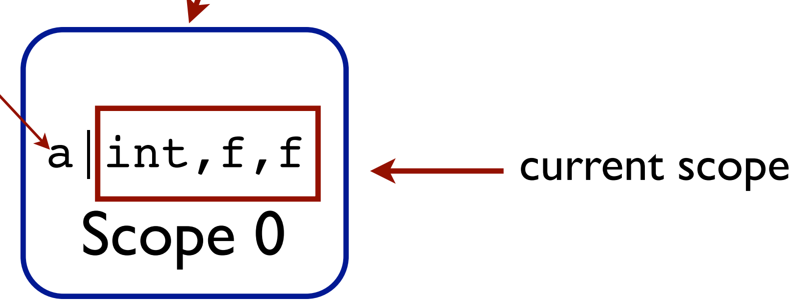


Initialize Scope 0
add symbol a
in the ***current scope***

Source Code

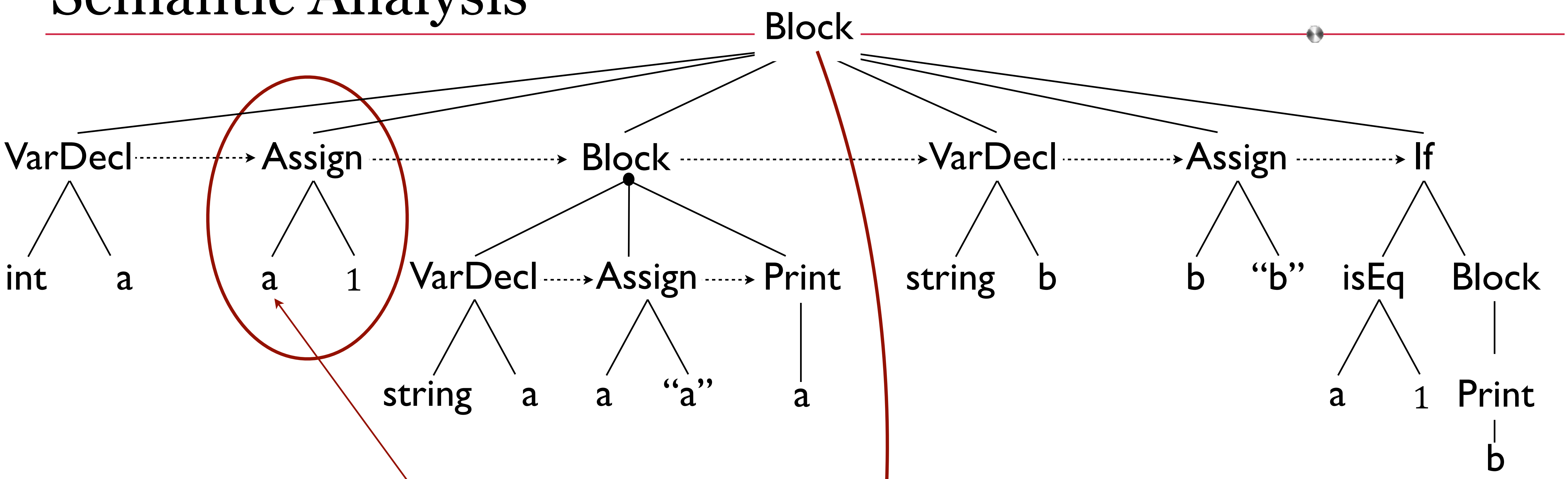
```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```

Add an object as the value in the key-value pair hash table with the attributes: type, isInitialized, isUsed.



Symbol Table

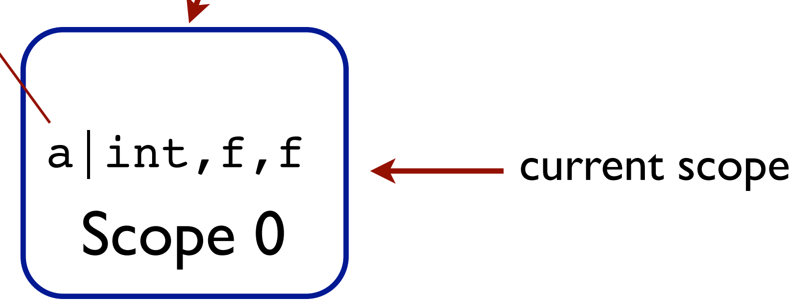
Semantic Analysis



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

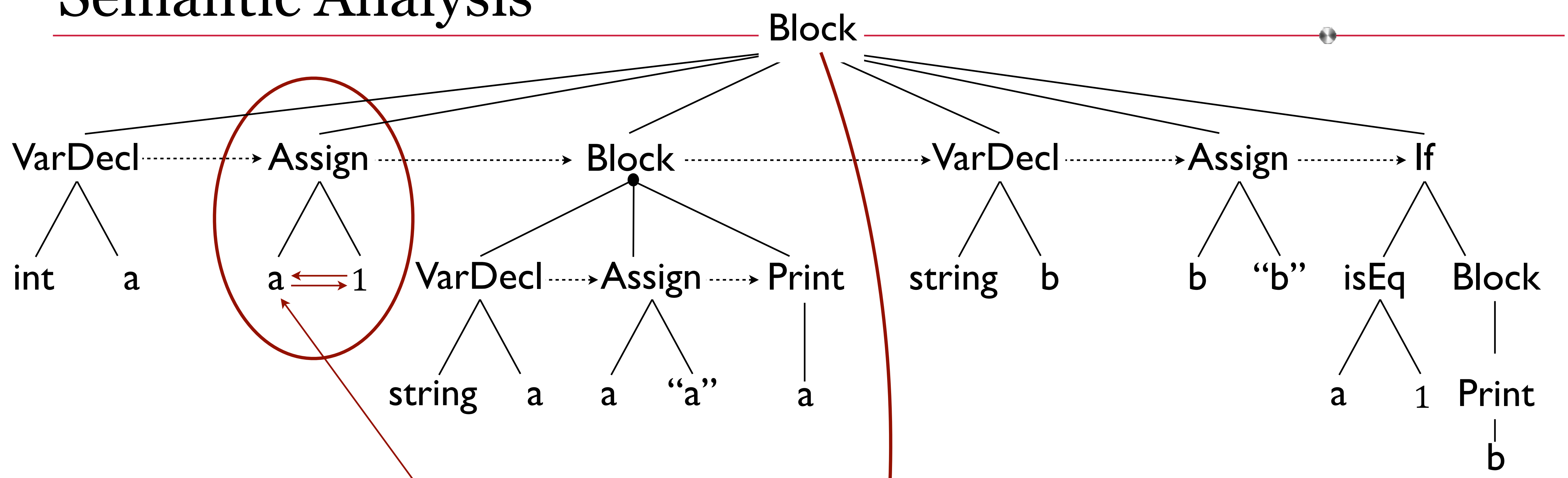
Source Code

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



Symbol Table

Semantic Analysis



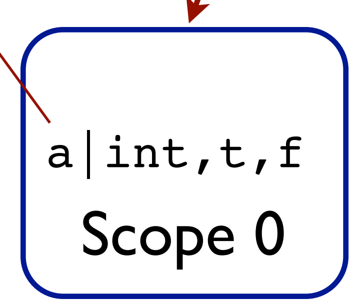
Initialize Scope 0
add symbol a
lookup symbol a
check types

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)
   }
}
```

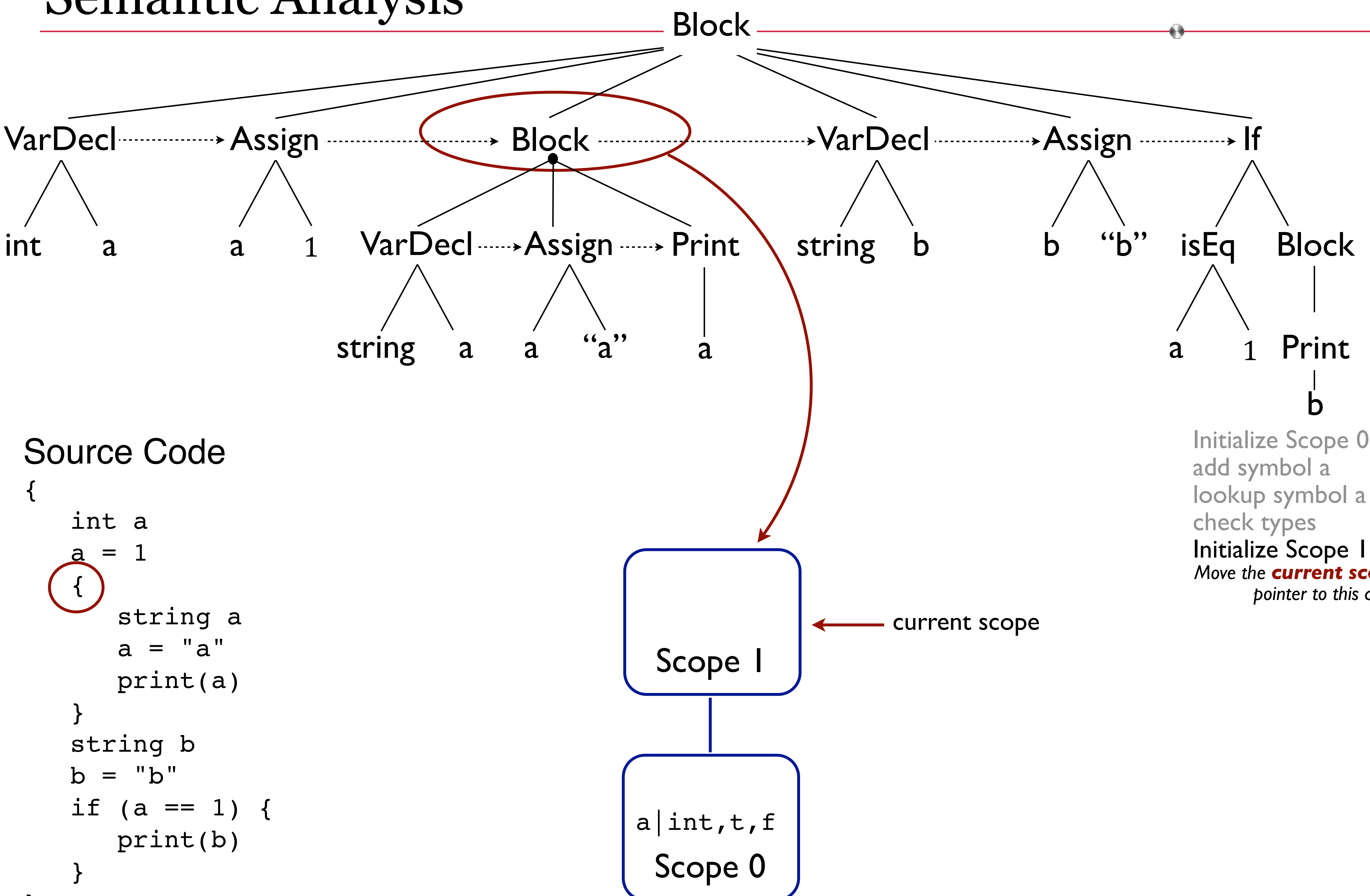
Note that id "a" in scope 0 has been initialized.



current scope

Symbol Table

Semantic Analysis



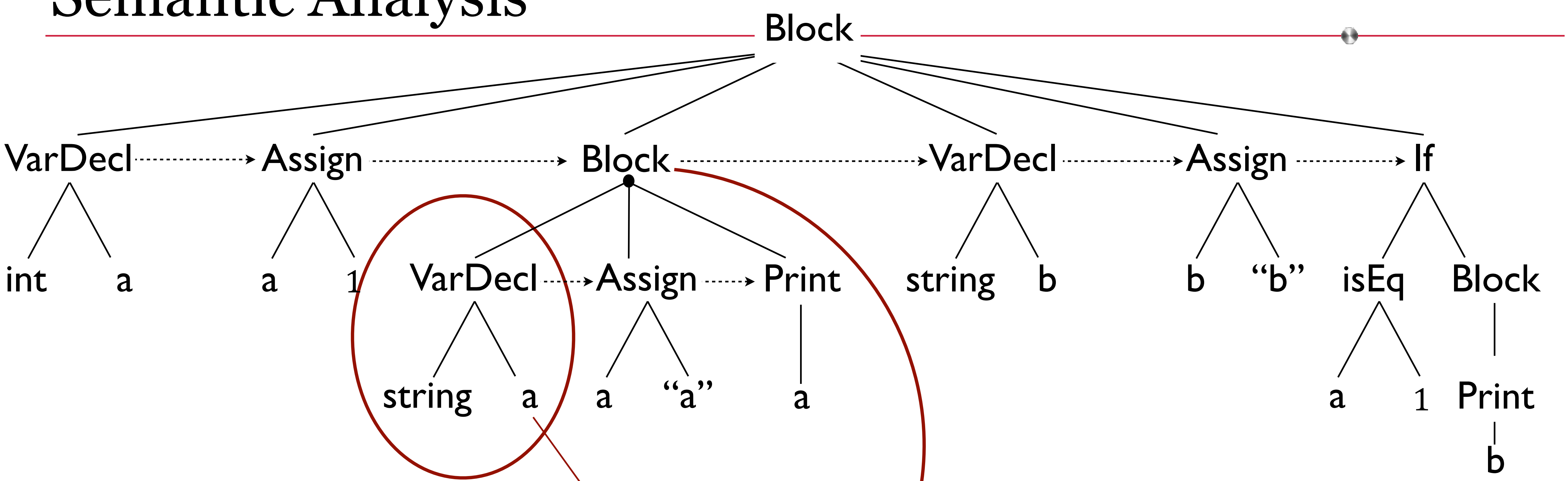
Source Code

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

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

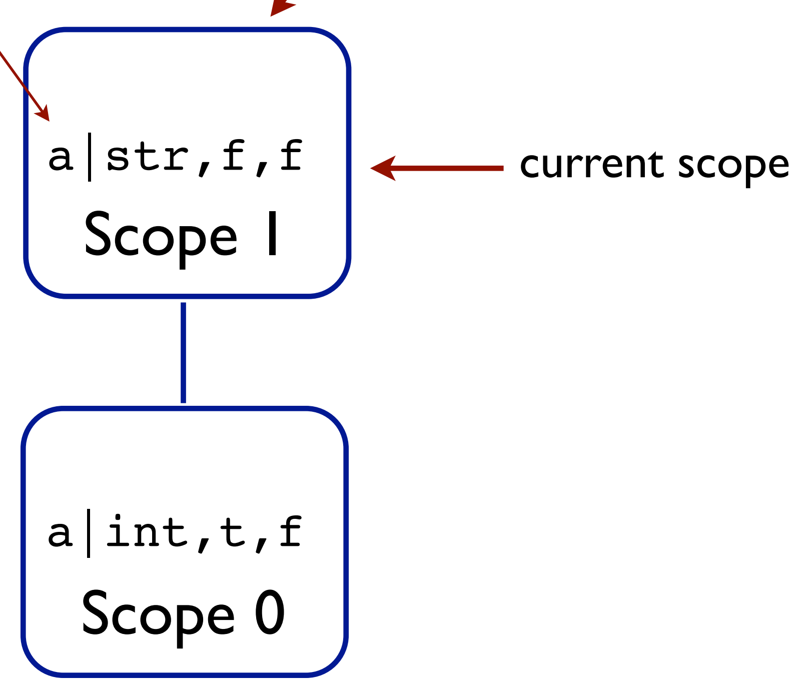
Symbol Table

Semantic Analysis



Source Code

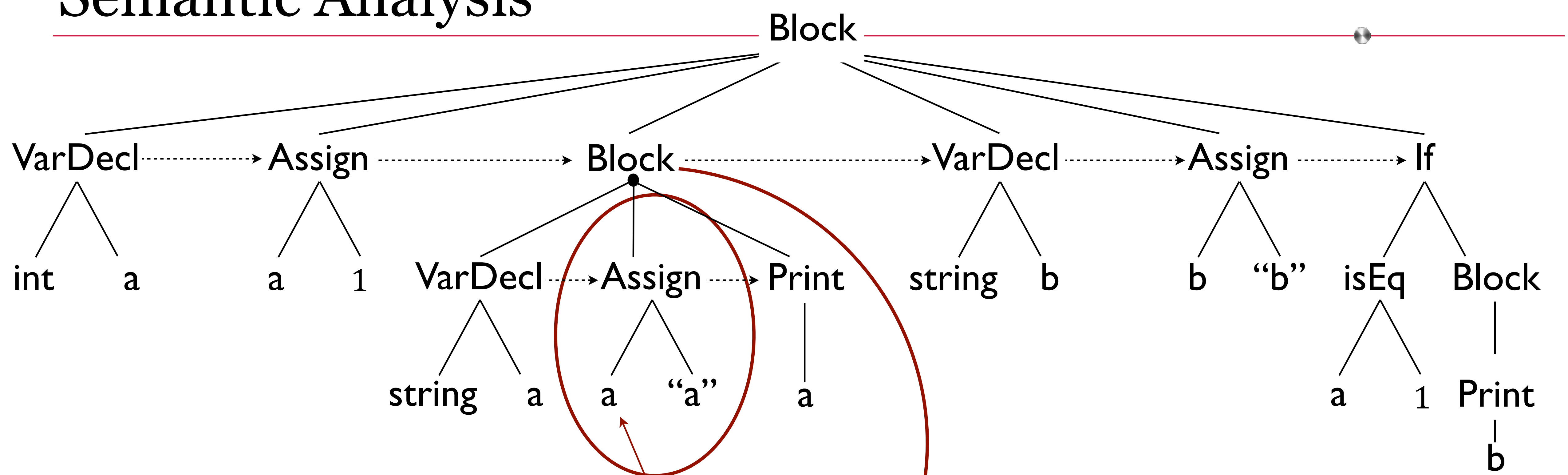
```
{
    int a
    a = 1
    {
      string a
      a = "a"
      print(a)
    }
    string b
    b = "b"
    if (a == 1) {
      print(b)
    }
}
```



Symbol Table

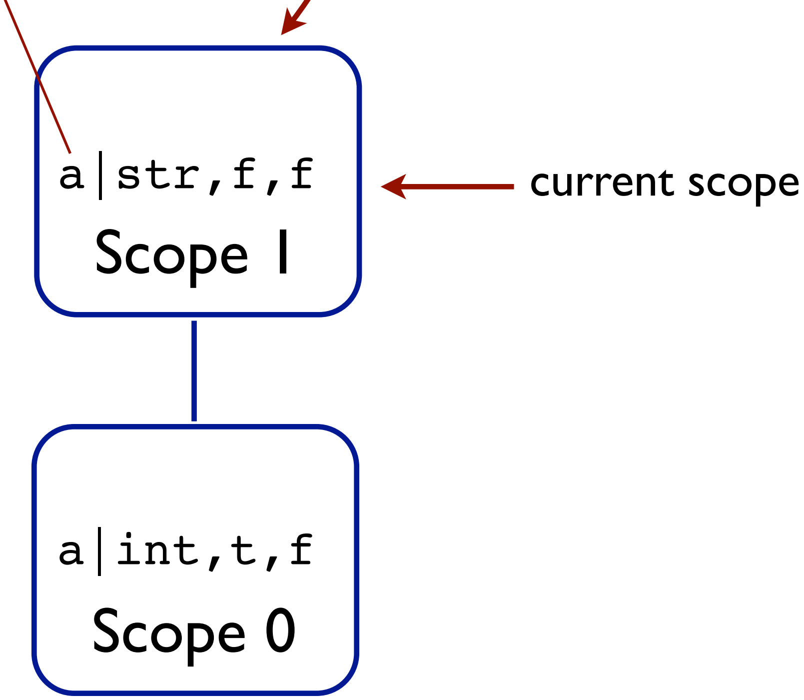
Initialize Scope 0
add symbol a
lookup symbol a
check types
Initialize Scope 1
add symbol a
in the ***current scope***

Semantic Analysis



Source Code

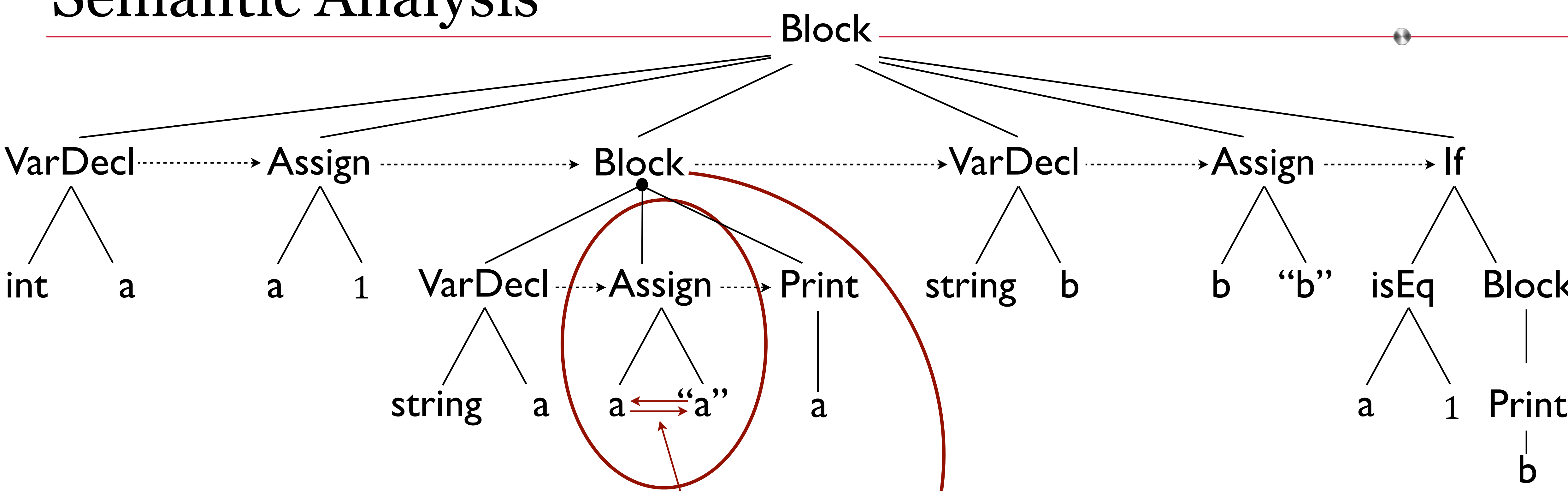
```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```



Symbol Table

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

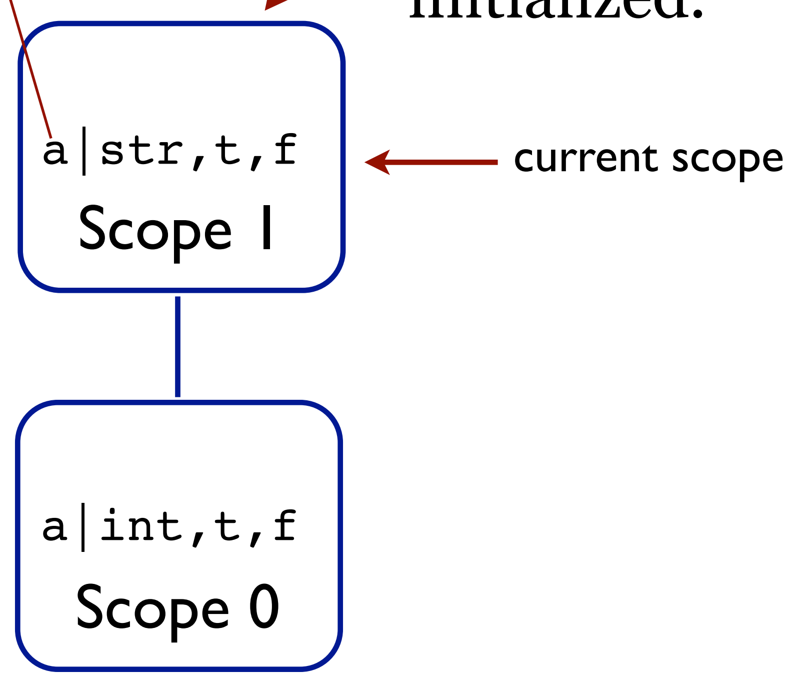
Semantic Analysis



Note that id “a” in scope 1 has been initialized.

Source Code

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

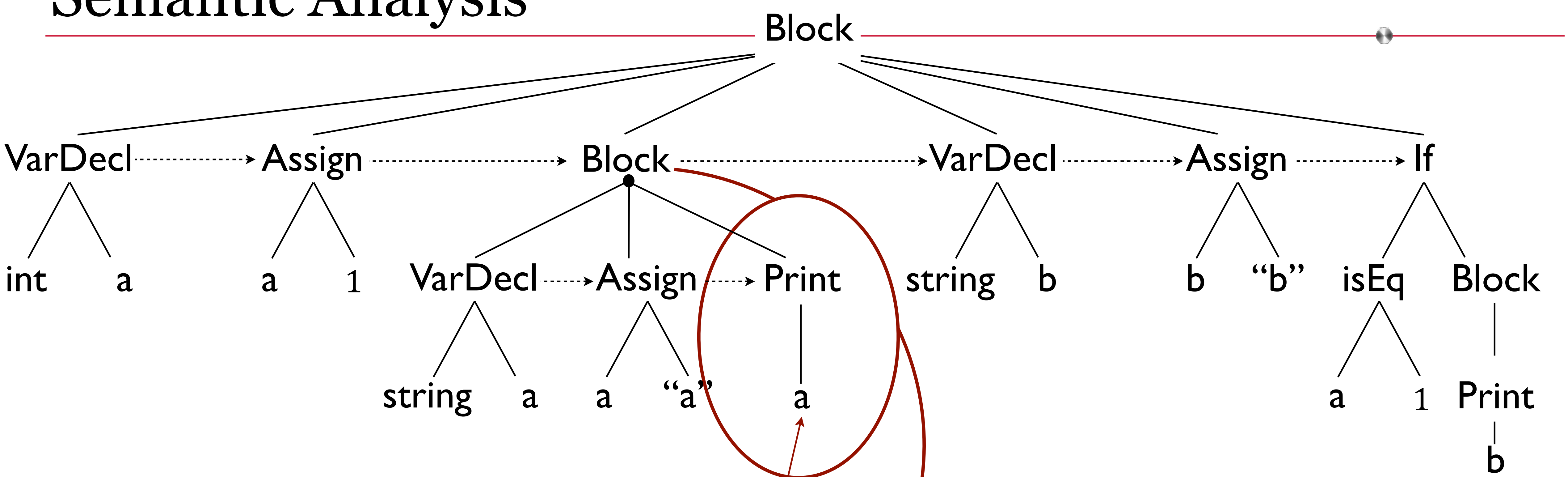


Symbol Table

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.

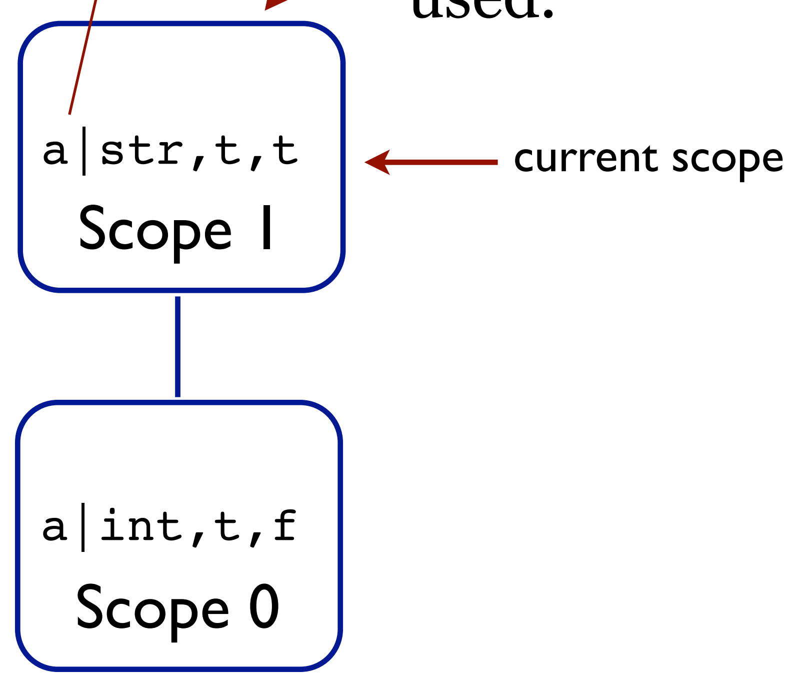
Semantic Analysis



Source Code

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

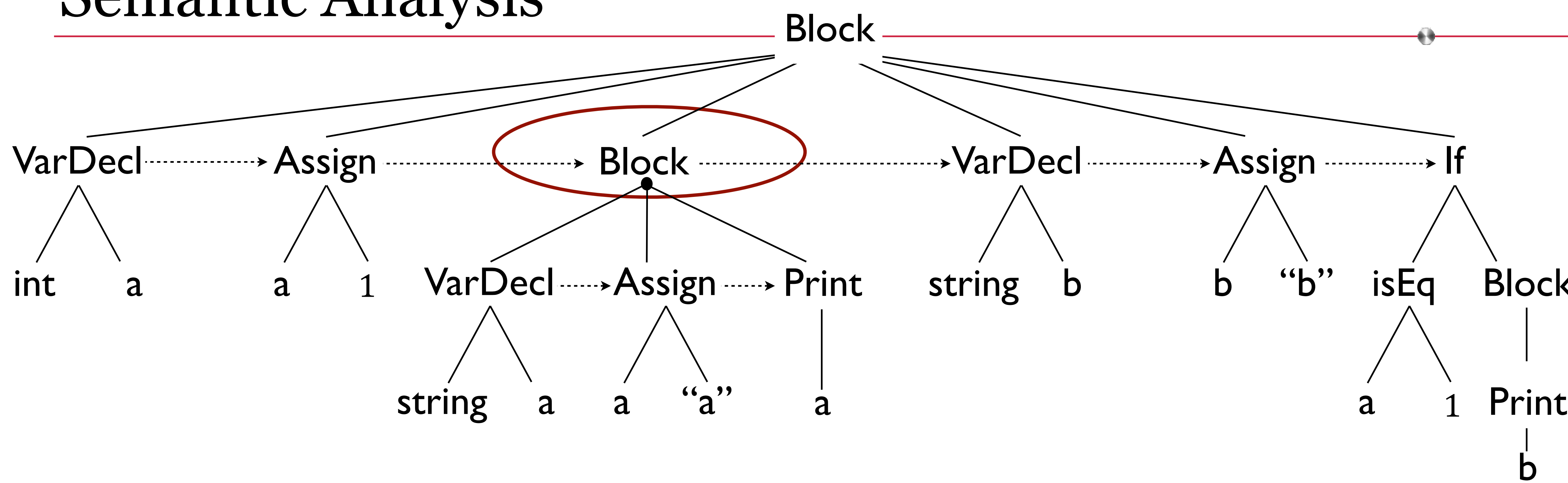
Note that id “a” in scope 1 has been used.



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!

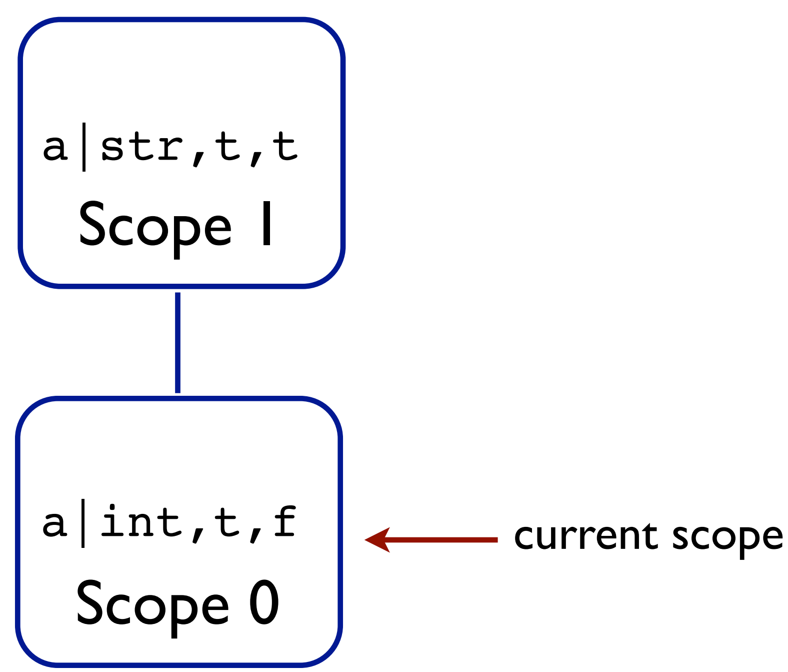
Semantic Analysis



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

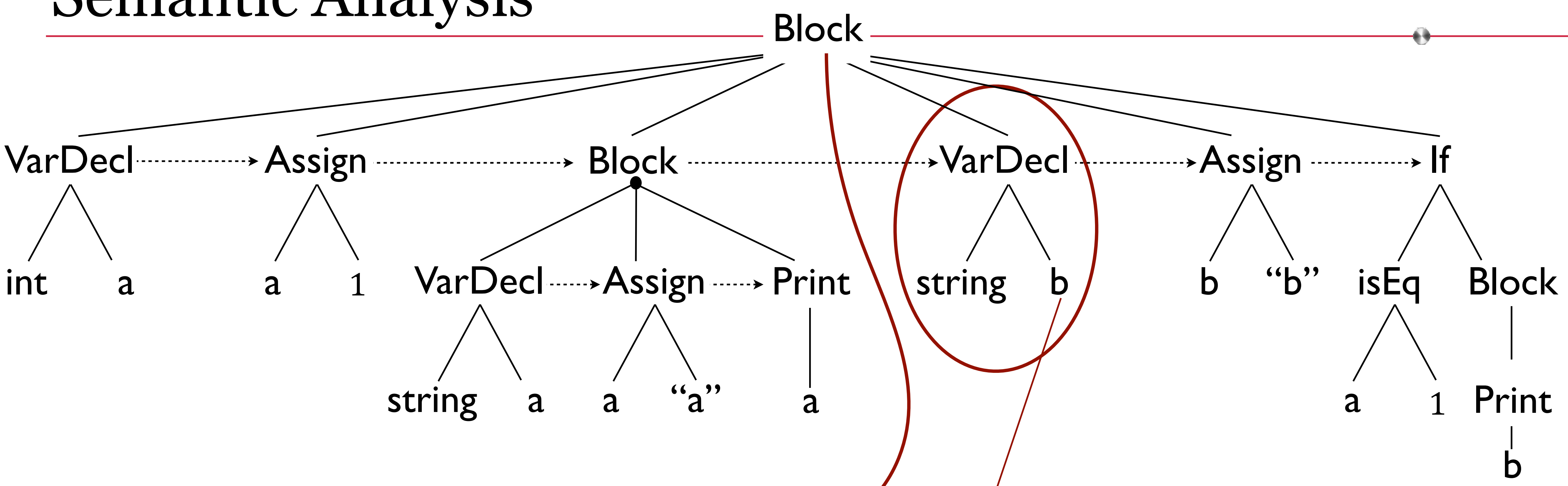
Source Code

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



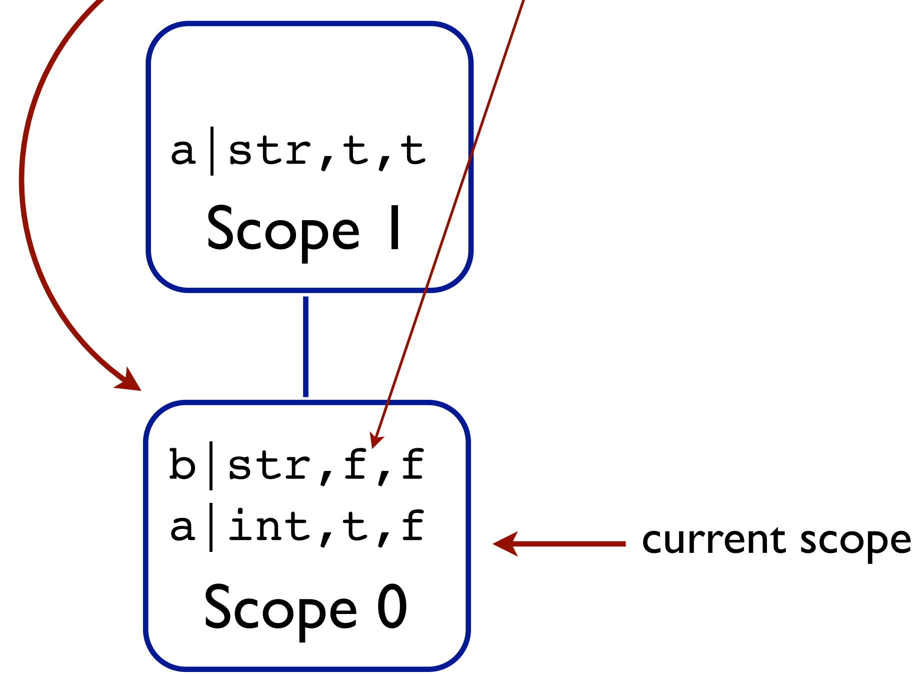
Symbol Table

Semantic Analysis



Source Code

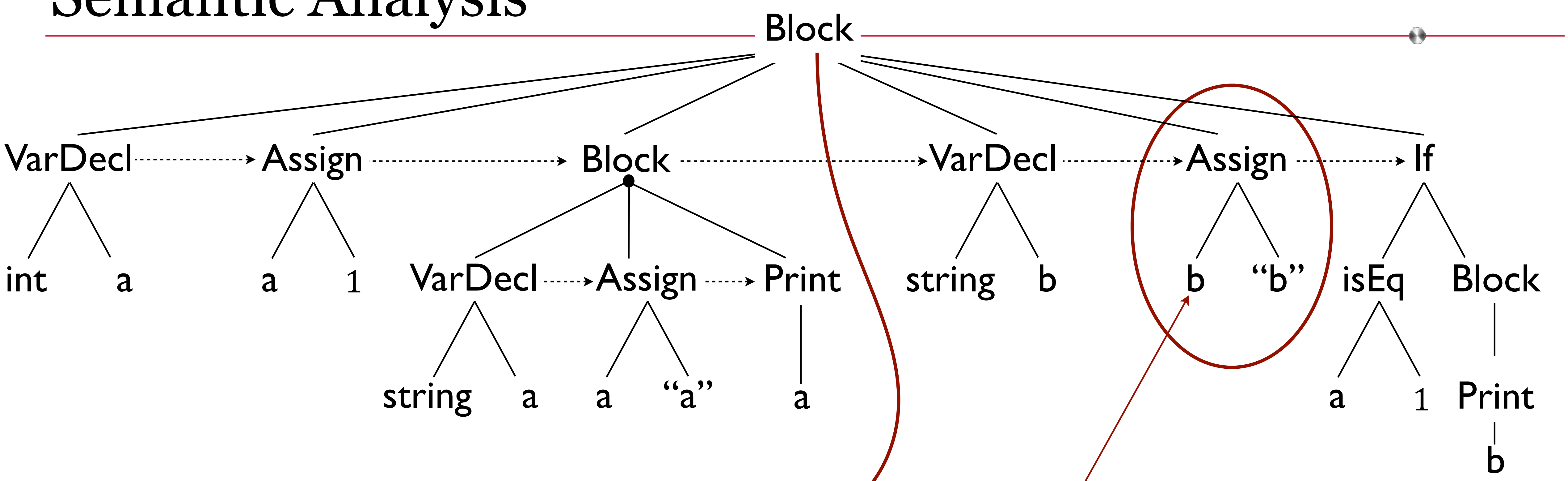
```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```



Symbol Table

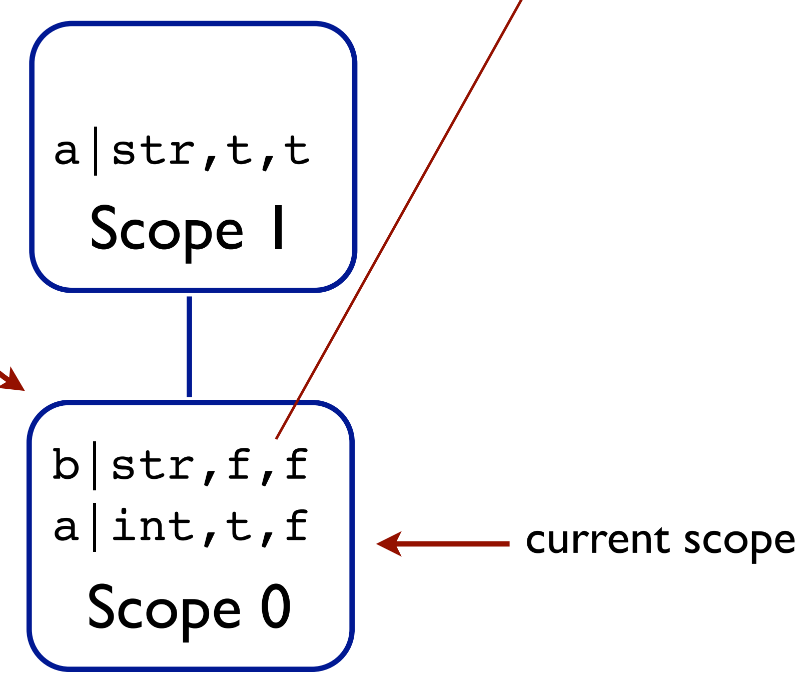
Initialize Scope 0
add symbol a
lookup symbol a
check types
Initialize Scope 1
add symbol a
lookup symbol a
check types
lookup symbol a
Close Scope 1
add symbol b
in the ***current scope***

Semantic Analysis



Source Code

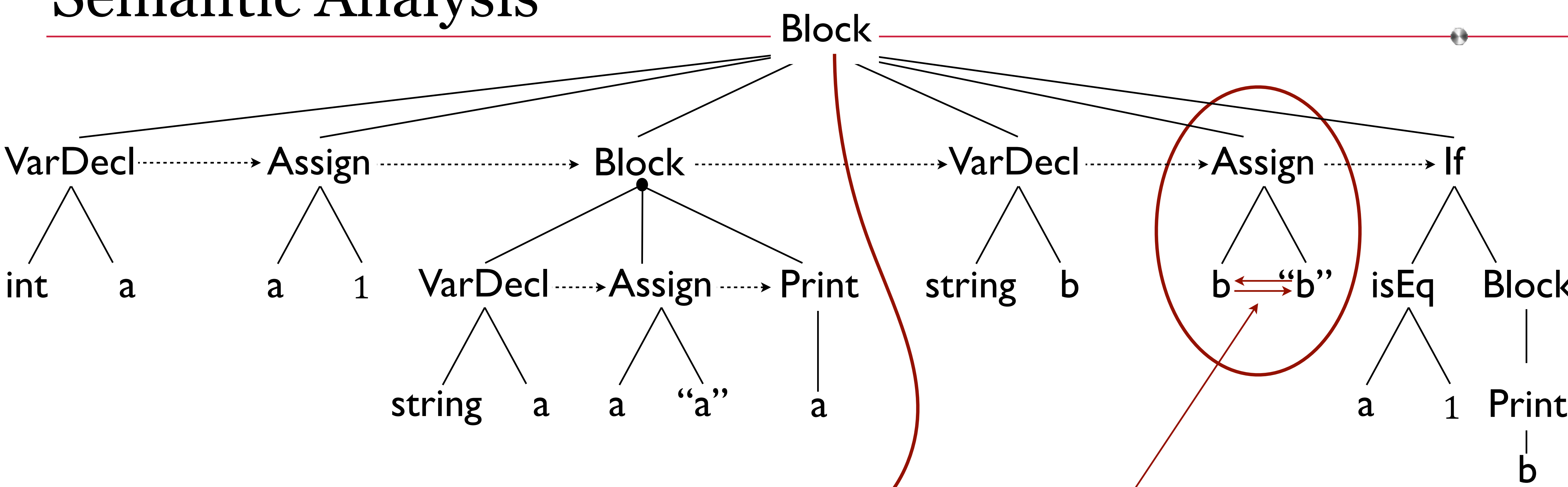
```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```



Symbol Table

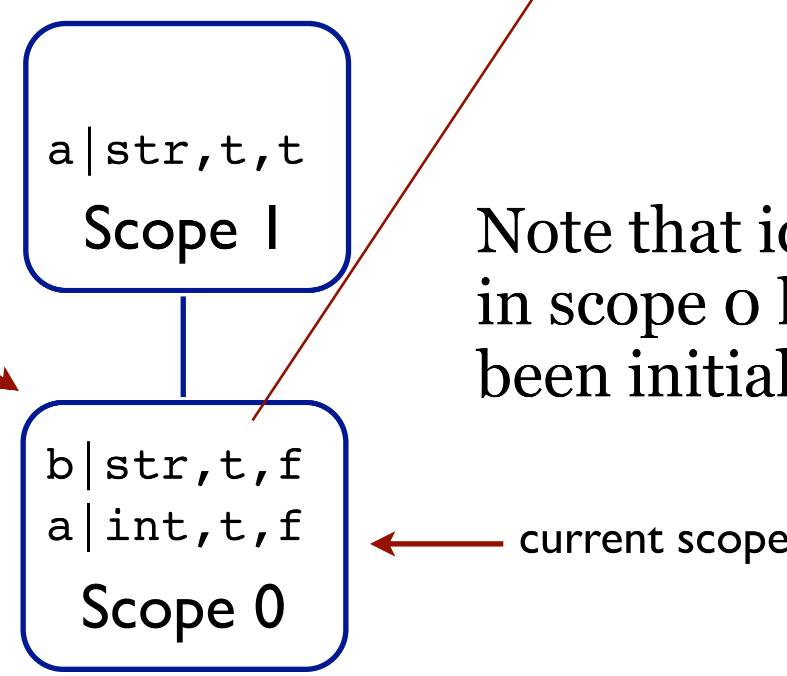
Initialize Scope 0
add symbol a
lookup symbol a
check types
Initialize Scope 1
add symbol a
lookup symbol a
check types
lookup symbol a
Close Scope 1
add symbol b
lookup symbol b
*in the **current scope***

Semantic Analysis



Source Code

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



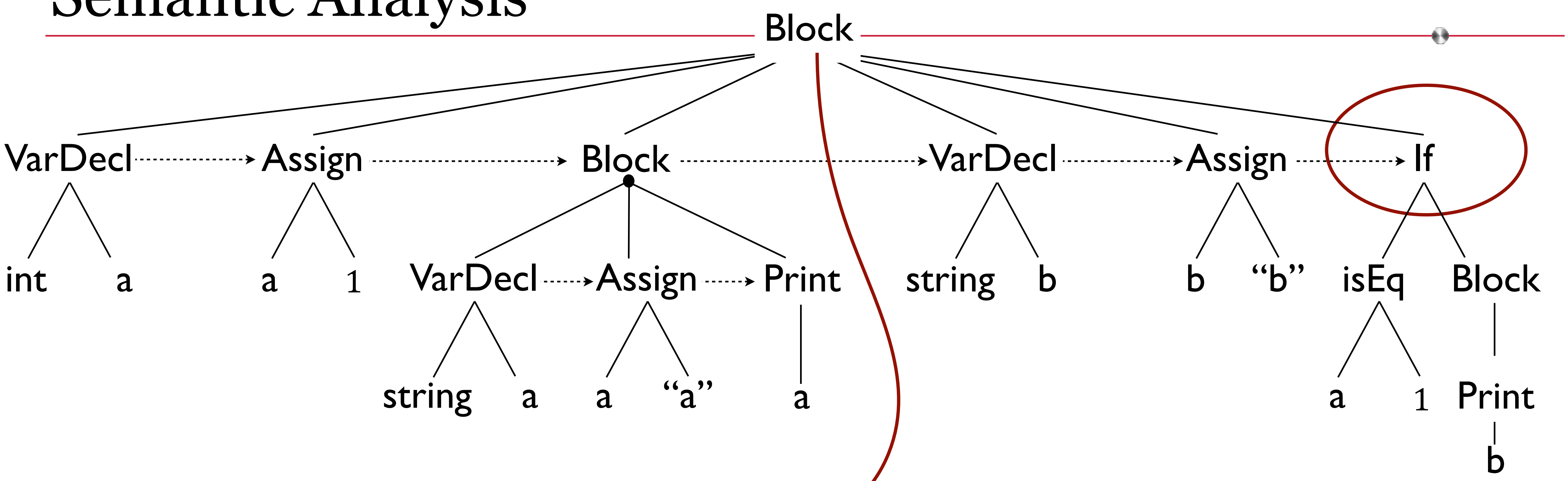
Symbol Table

Note that id “b” in scope 0 has been initialized.

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

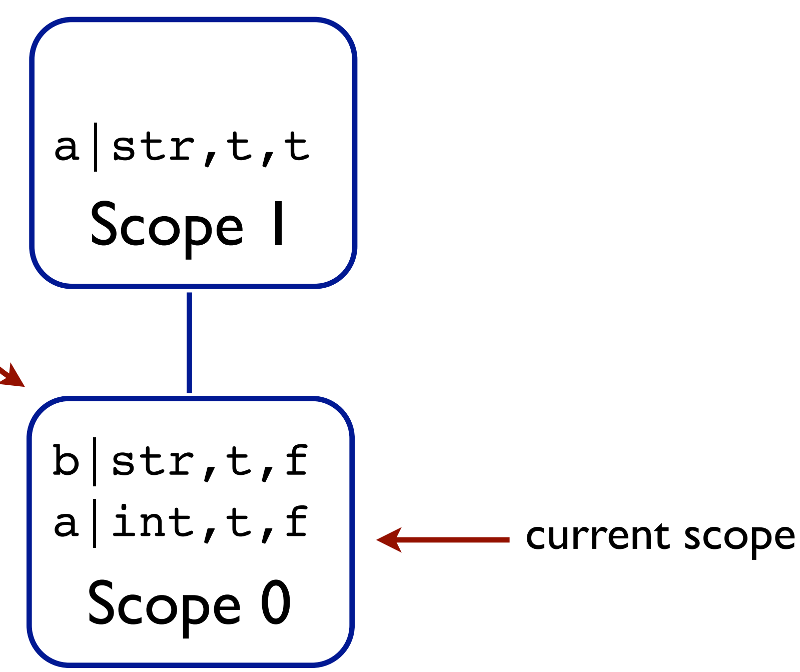
Verify that the left child and right child are type compatible for assignment.

Semantic Analysis



Source Code

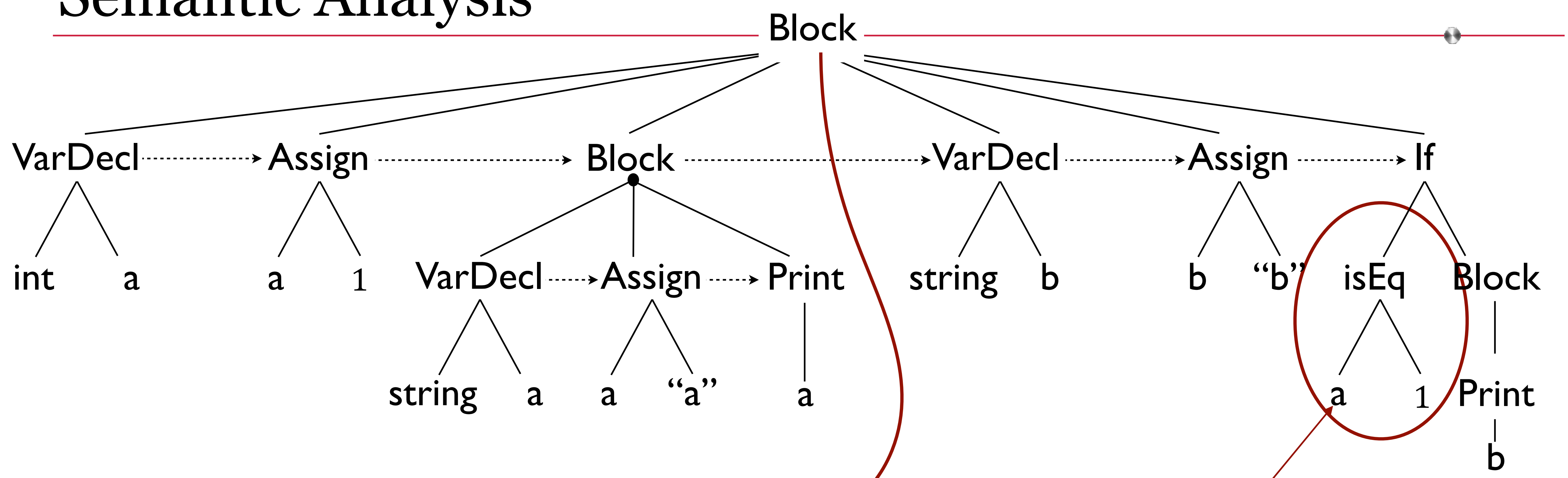
```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```



Symbol Table

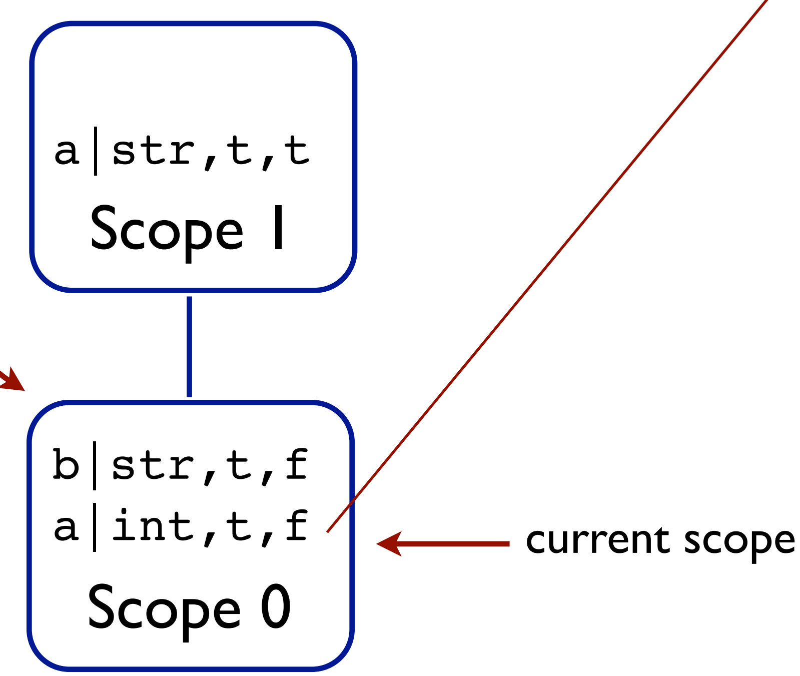
Initialize Scope 0
add symbol a
lookup symbol a
check types
Initialize Scope 1
add symbol a
lookup symbol a
check types
lookup symbol a
Close Scope 1
add symbol b
lookup symbol b
check types

Semantic Analysis



Source Code

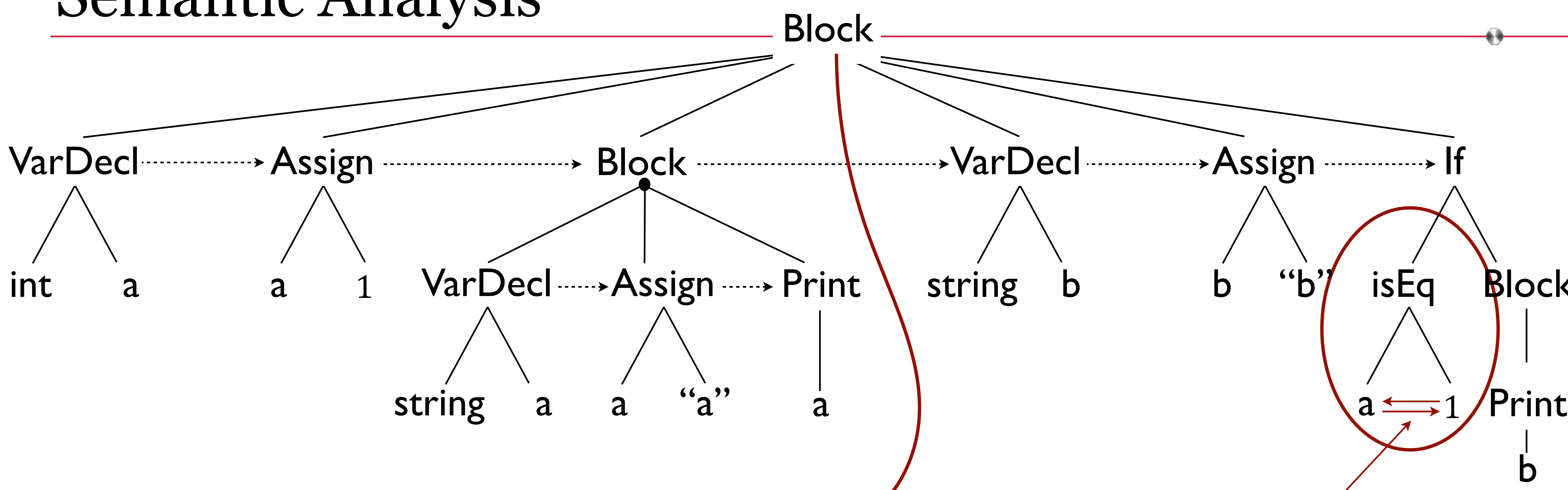
```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```



Symbol Table

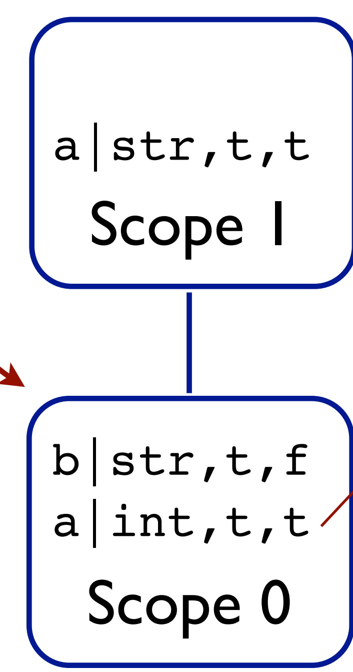
Initialize Scope 0
add symbol a
lookup symbol a
check types
Initialize Scope 1
add symbol a
lookup symbol a
check types
lookup symbol a
Close Scope 1
add symbol b
lookup symbol b
check types
lookup symbol a
in the ***current scope***

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

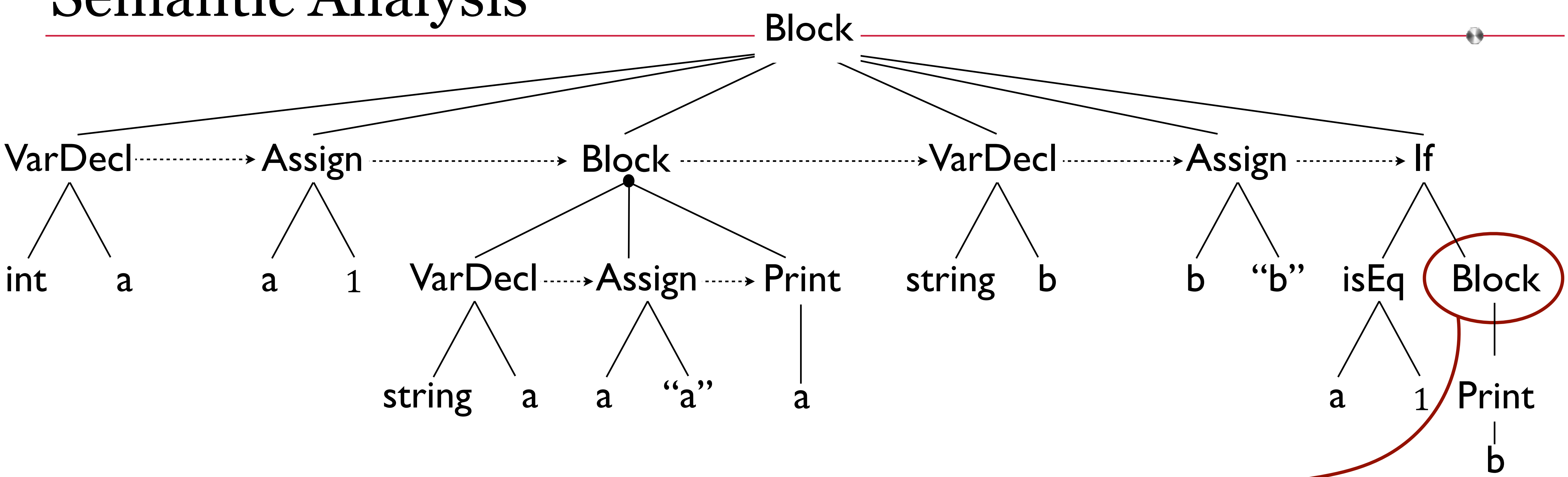
current scope

Note that id “a” in scope 0 has been used.

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

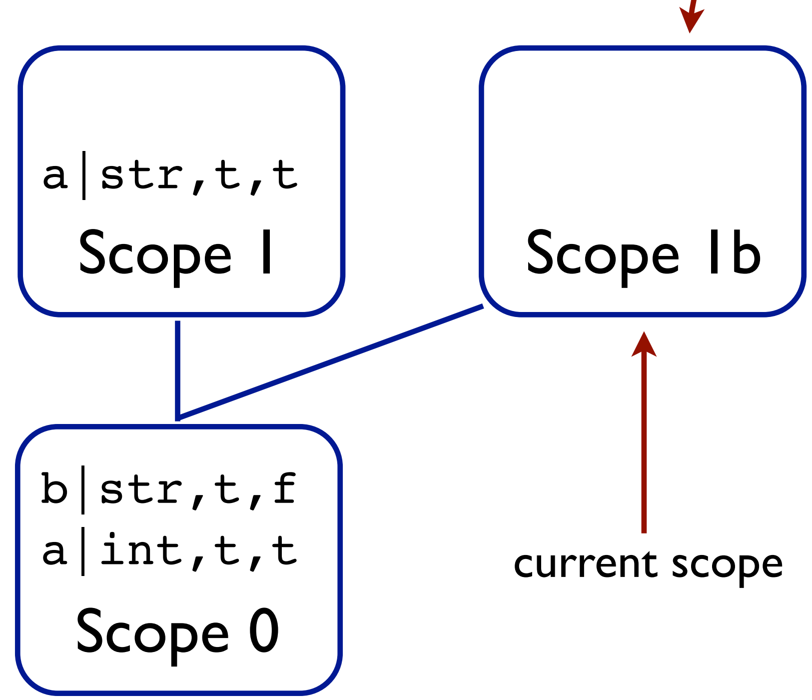
Verify that the left child and right child are ***type comparable***

Semantic Analysis



Source Code

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

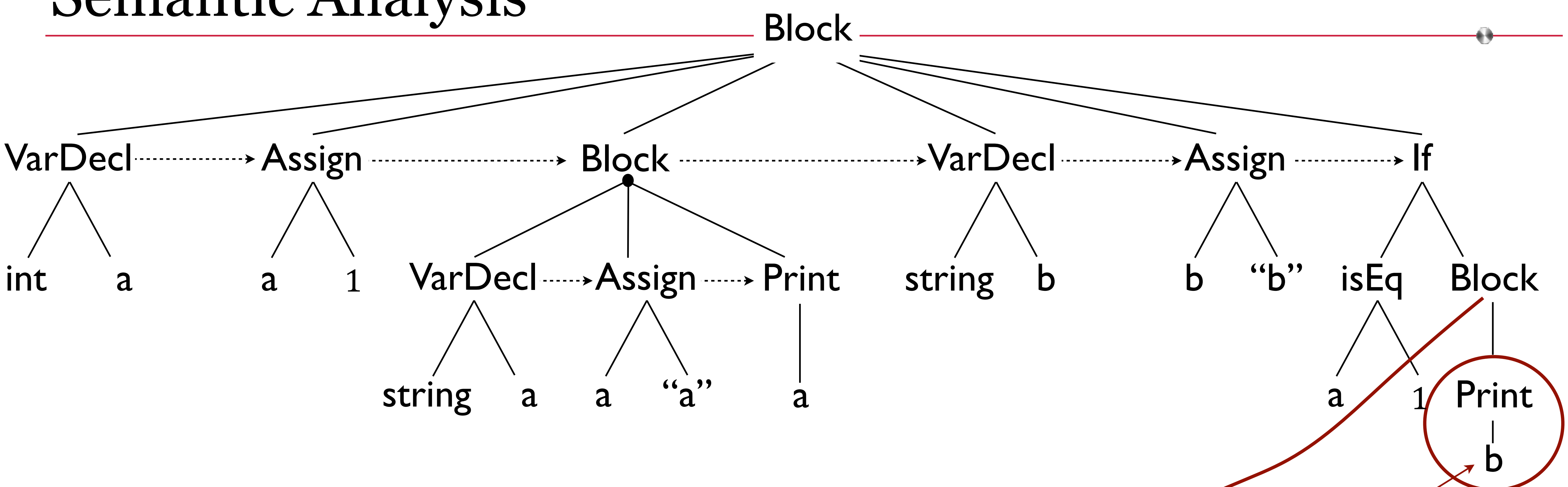


Initialize Scope 0
add symbol a
lookup symbol a
check types
Initialize Scope 1
add symbol a
lookup symbol a
check types
lookup symbol a
Close Scope 1
add symbol b
lookup symbol b
check types
lookup symbol a
check types
Initialize Scope 1b

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

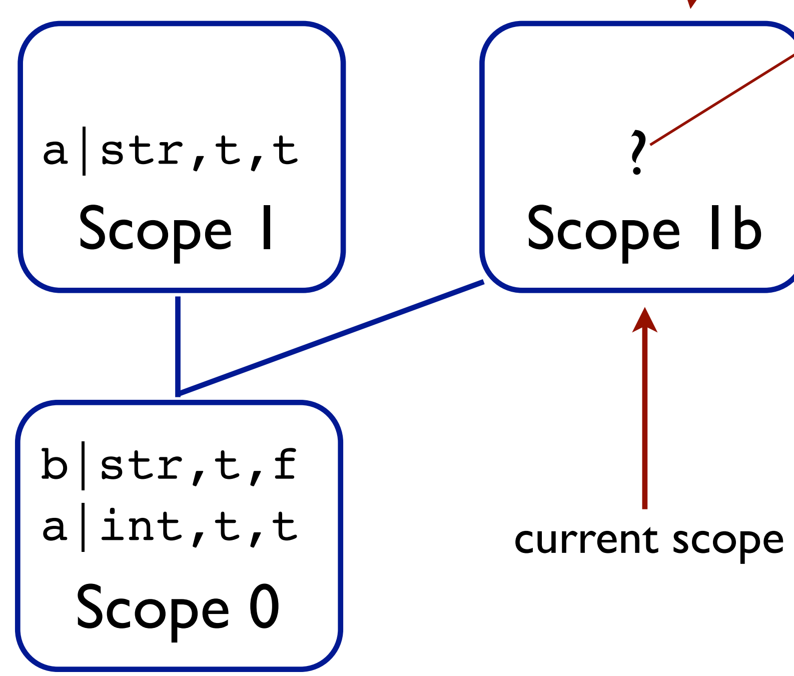
Symbol Table

Semantic Analysis



Source Code

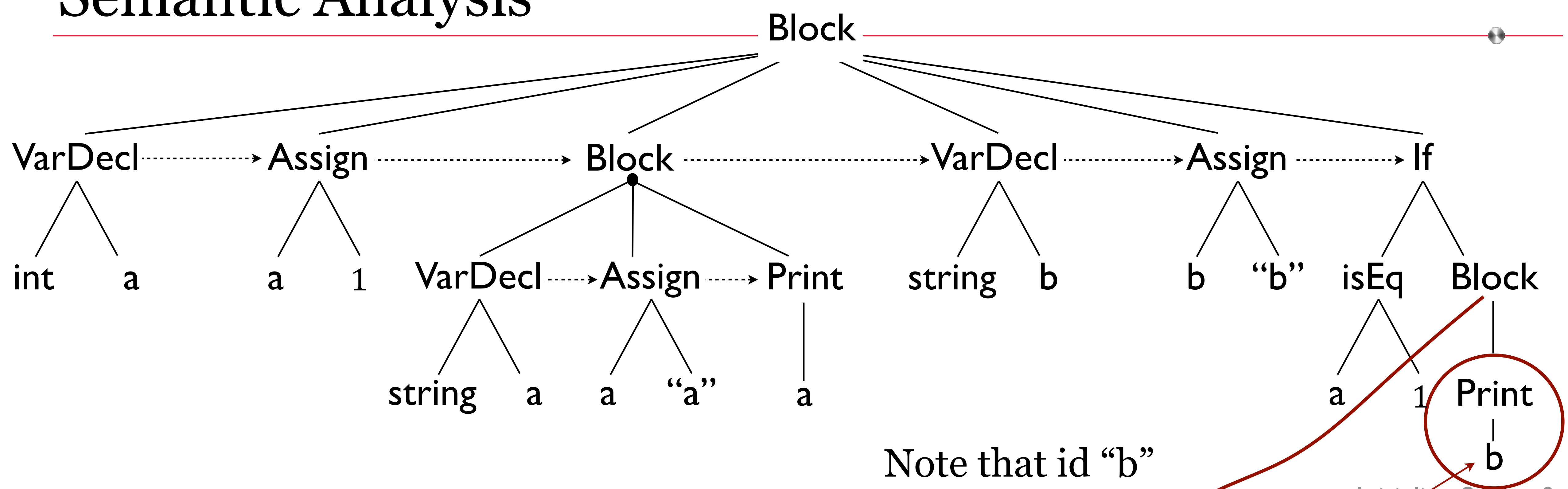
```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```



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

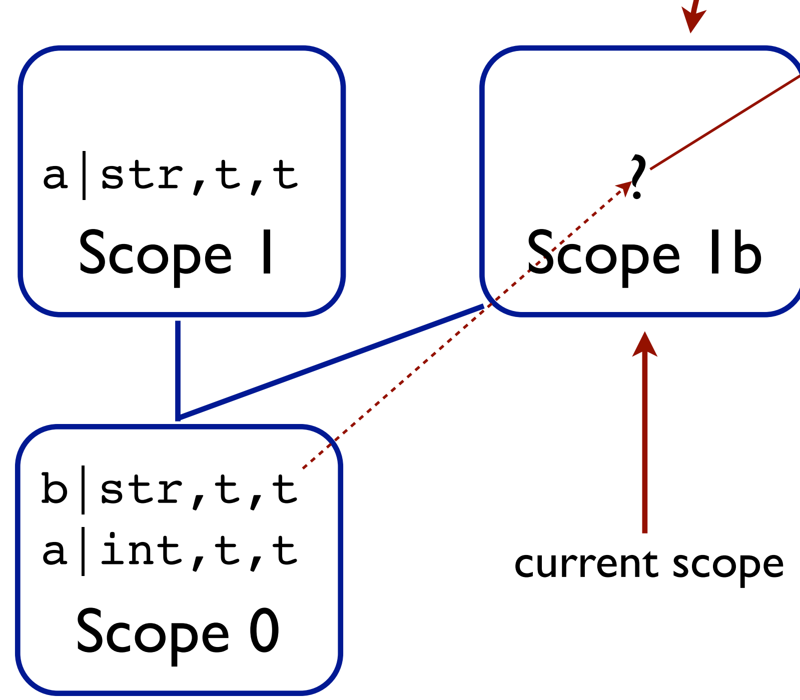
Semantic Analysis



Source Code

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

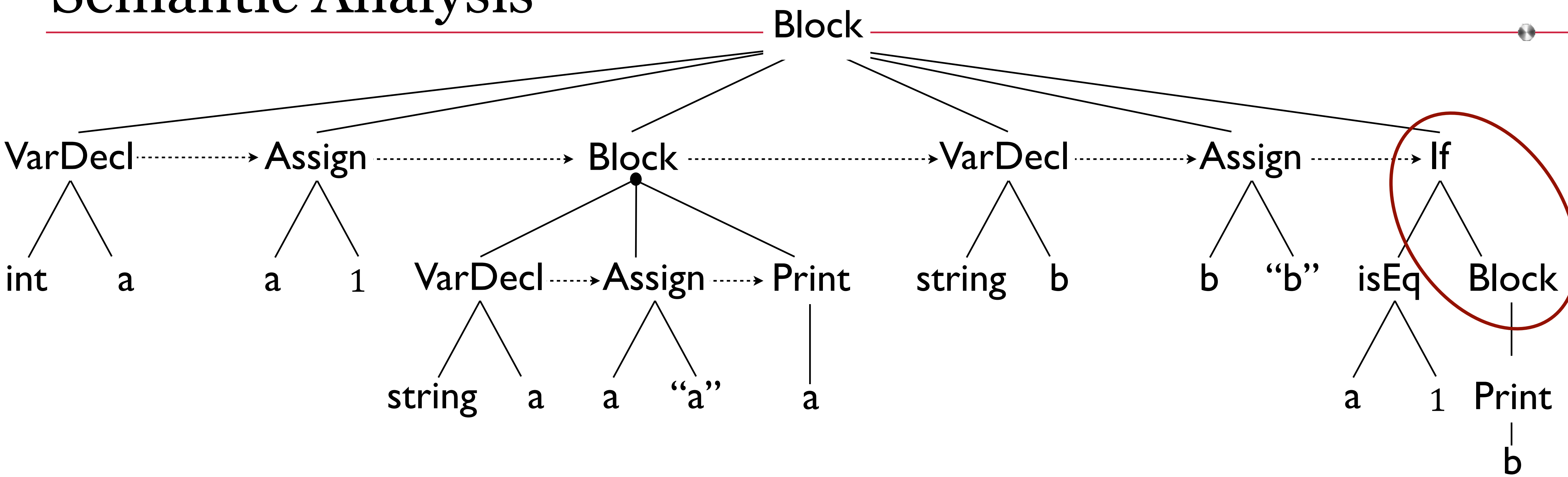
Note that id “b” in scope 0 has been used.



Symbol Table

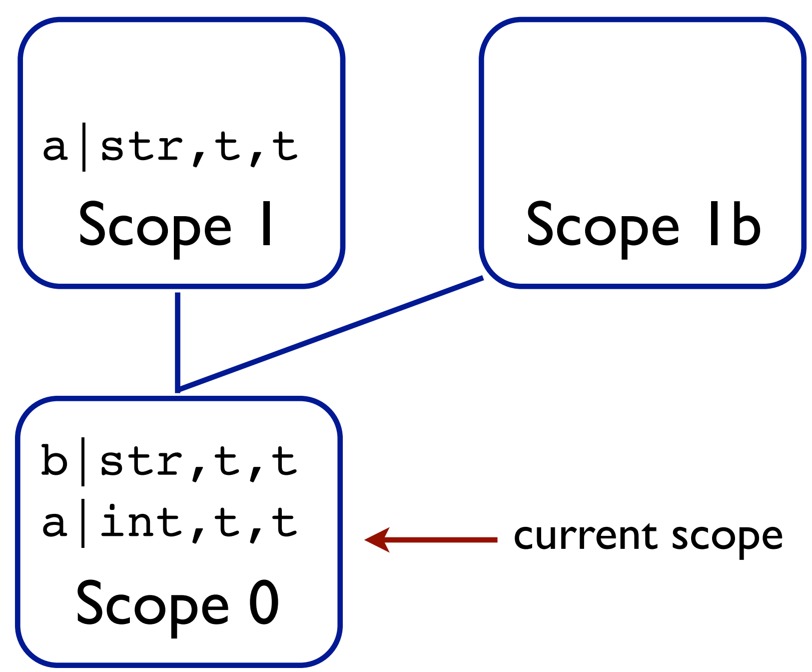
Initialize Scope 0
add symbol a
lookup symbol a
check types
Initialize Scope 1
add symbol a
lookup symbol a
check types
lookup symbol a
Close Scope 1
add symbol b
lookup symbol b
check types
lookup symbol a
check types
Initialize Scope 1b
lookup symbol b
*in the **parent scope**.*
Print can take any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

Semantic Analysis



Source Code

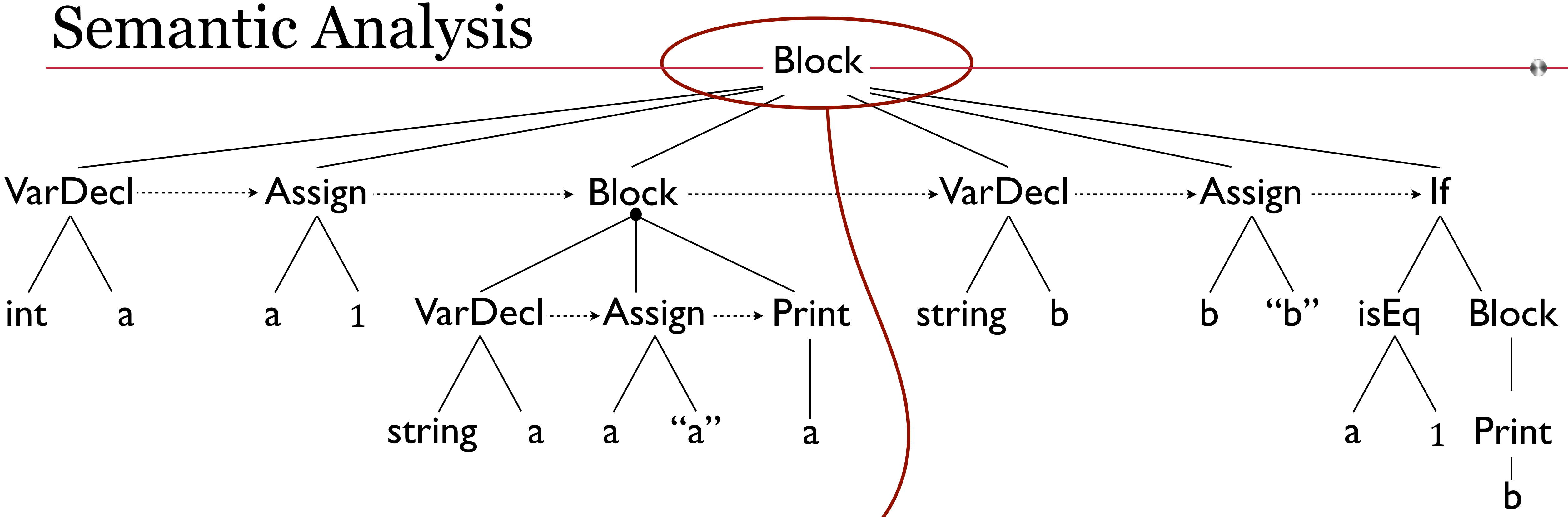
```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```



Symbol Table

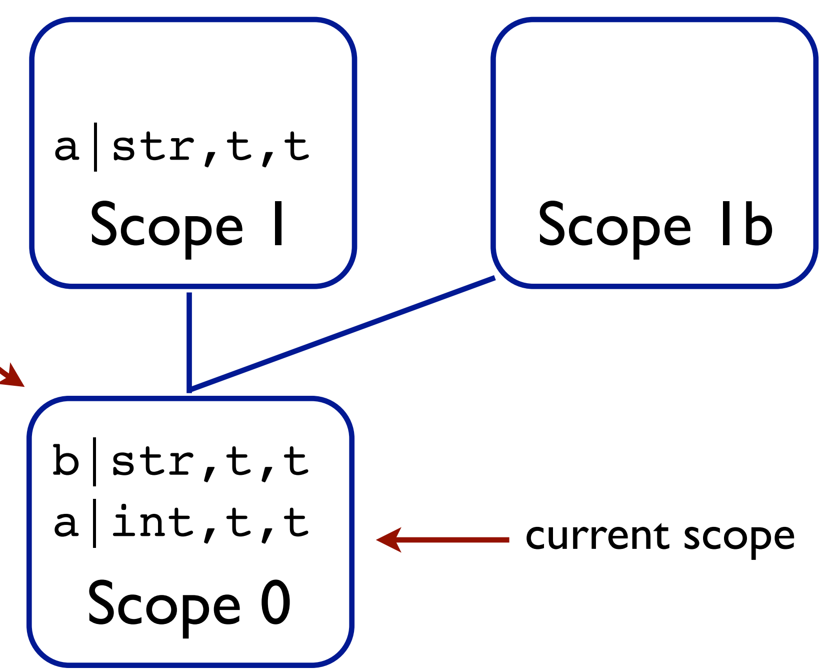
Initialize Scope 0
add symbol a
lookup symbol a
check types
Initialize Scope 1
add symbol a
lookup symbol a
check types
lookup symbol a
Close Scope 1
add symbol b
lookup symbol b
check types
lookup symbol a
check types
Initialize Scope 1b
lookup symbol b
Close Scope 1b

Semantic Analysis



Source Code

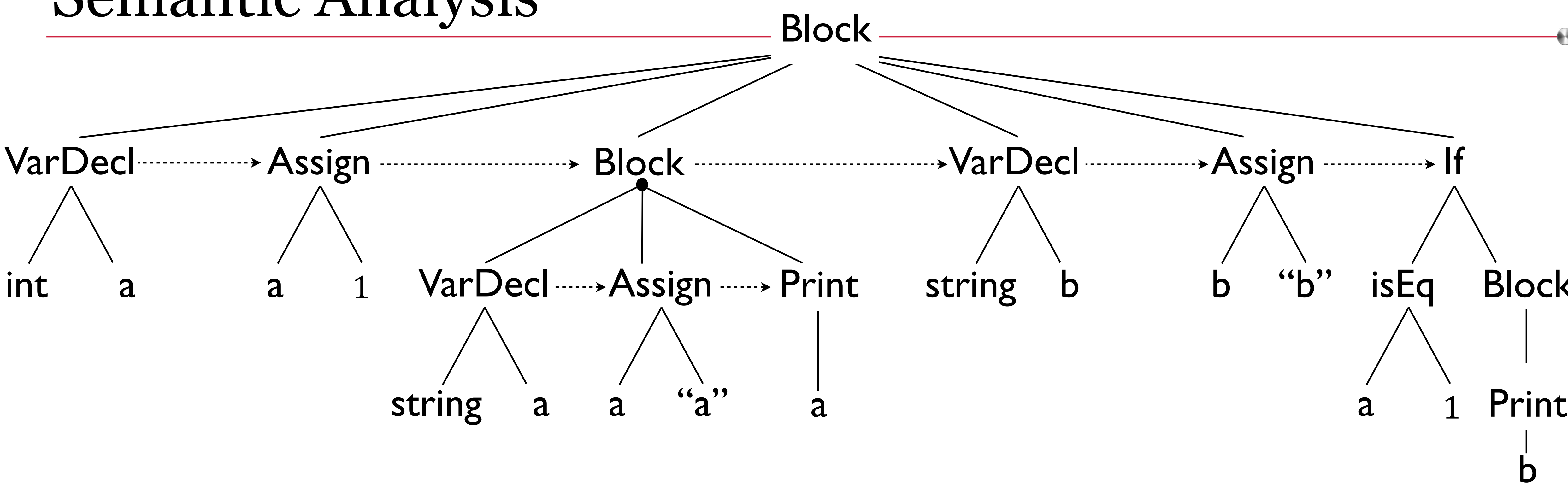
```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```



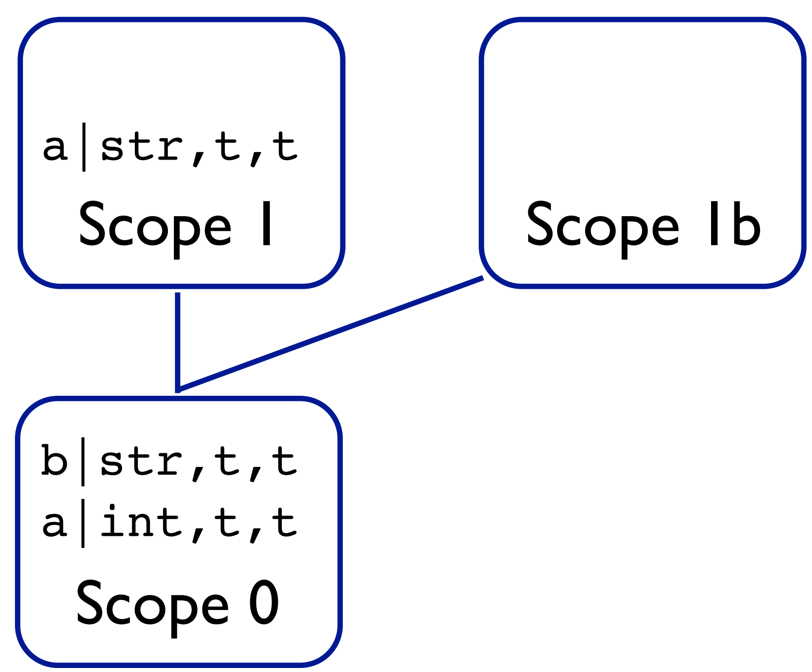
Symbol Table

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

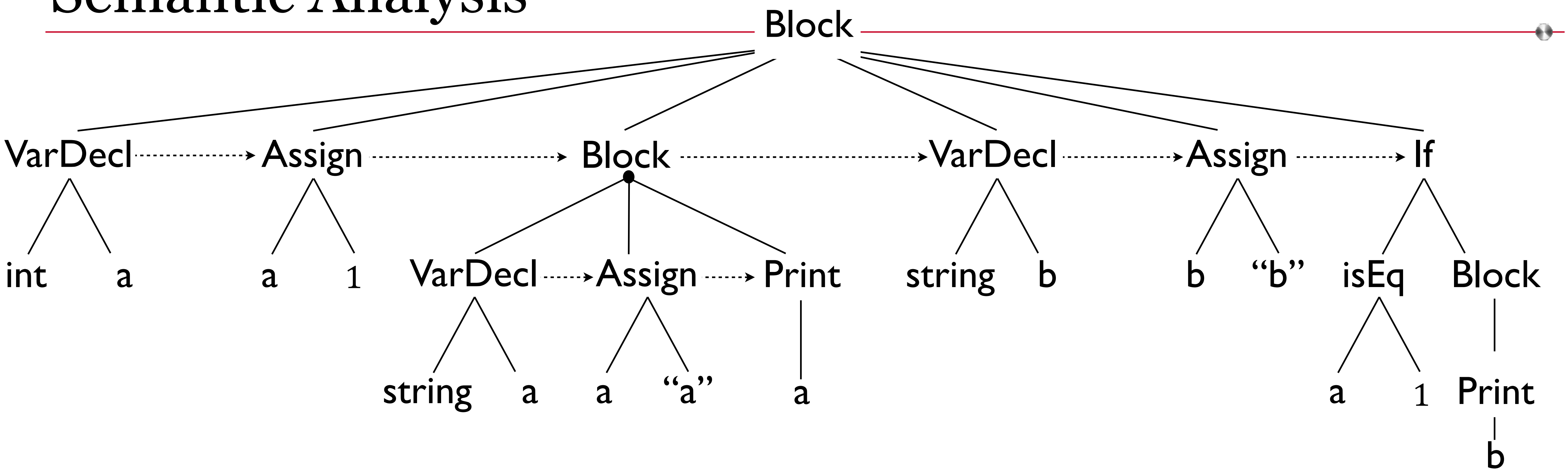
Semantic Analysis



Before going on to Code Generation, traverse the Symbol Table and give warnings based on state of the Boolean flags.

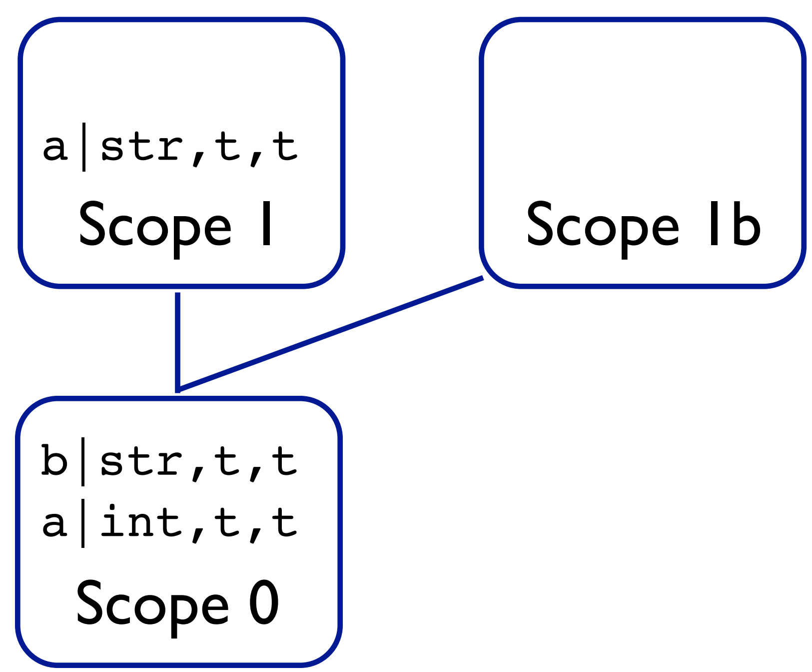


Semantic Analysis



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

We are really ready for Code Generation.



Semantic Analysis

So far we have assumed a very simple type system.
We can compare and assign …

- *int* to *int*
- *string* to *string*
- *boolean* to *boolean*

… anything else is a type error. This is all we need for our project.

But what if we wanted a more sophisticated type system?
Considerations:

- Strong or weak typing?
- assign *int* to *float*?

How do we even talk about it? Reason about it?
Context Free Grammars say nothing about types.
We need some new documentation for our language.
We need a **Type System**.