Semantic Analysis

Alan G. Labouseur, Ph.D.
Alan.Labouseur@Marist.edu
Compiler — High Level View

Source Code

Lex

Token Stream

Parse

CST

Semantic Analysis

Errors and Warnings

Symbol Table

AST Optimizations

Instruction Selection

Instruction Optimizations

Register Allocation

AST (IR)

Code Generation

Machine Language
Compiler — High Level View

Source Code

Lex

Token Stream

Parse

Errors and Warnings

CST

Semantic Analysis

Semantic Analysis

Symbol Table

AST (IR)

AST Optimizations

Instruction Selection

Instruction Optimizations

Register Allocation

Code Generation

Machine Language
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 for Code Generation.

• Focus on meaning: scope and type.


Review: 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);
    }
}
```

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.
Review: Static and Dynamic Scope

Dynamic Scope

```java
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);
    }
}
```

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

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

1. \( \langle \text{list} \rangle \ ::= \langle \text{list} \rangle + \langle \text{digit} \rangle \)
2. \( \langle \text{list} \rangle \ ::= \langle \text{list} \rangle - \langle \text{digit} \rangle \)
3. \( \langle \text{list} \rangle \ ::= \langle \text{digit} \rangle \)
4. \( \langle \text{digit} \rangle \ ::= 0|1|2|3|4|5|6|7|8|9 \)

Abstract Syntax Trees

Just the good parts.
Abstract Syntax Trees

Consider a program in our grammar:

```plaintext
{ 
    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 [TType] 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 [PrintStatement], 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

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
```

Note the epsilon production of StatementList that stops the recursion.

This is not really part of the CST. It’s only here to illustrate how it works.
Abstract Syntax Trees

CST for a program in our grammar:

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

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:

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

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)  
}

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:

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

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) 
}
```

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:

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

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:

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

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:

```plaintext
{ int a
  a = 1
  print(a)
}
```
Abstract Syntax Trees

Building an AST for a program in our language:

```plaintext
{ int a 
a = 1
print(a)
}
```
Abstract Syntax Trees

CST and AST for a program in our language:

```plaintext
{  
  int a
  a = 1
  print(a)
}
```
Abstract Syntax Trees

CST and AST for a program in our language:

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

Q: How do we identify the good parts?
Semantic Analysis

Turning a CST into an AST

Q: How do we identify the good parts are?
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.

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

Or, re-“parse” the token stream, looking for key elements, building the AST as you go according to the AST subtree patterns. But this is not a real parse, because you already know that everything is syntactically correct. It’s just another pass over the tokens to build the AST. This may be easier than traversing the CST.
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 ::= $  
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 )  
boolop ::= boolean  
Id ::= char  
CharList ::= char CharList | space CharList  
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?
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)
    }
}
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

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

Symbol Table
```plaintext
Source Code
{
    int a
    a = 1
    {
        string a
        a = "a"
        print(a)
    }
    string b
    b = "b"
    if (a == 1) {
        print(b)
    }
}
```

Semantic Analysis

AST

Block

VarDecl → Assign → Block → VarDecl → Assign → If

Semantic Analysis

30

Symbol Table
Semantic Analysis

Source Code

```c
{  
    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
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
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)
    }
    
}
Semantic Analysis

Source Code
{
    int a
    a = 1
    {
        string a
        a = "a"
        print(a)
    }
    string b
    b = "b"
    if (a == 1) {
        print(b)
    }
}
Semantic Analysis

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

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

Source Code

```c
{ 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
Semantic Analysis

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

Initializing Scope 0
add symbol a
lookup symbol a
check types

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

Symbol Table

Scope 0
a | int

Scope 1
a | string

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

- **Scope 0**
  - `a` | int
  - `b` | string

- **Scope 1**
  - `a` | string
  - `b` | string

- `a` in the current scope
- `b` in the current scope
**Semantic Analysis**

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

**Symbol Table**

- Current scope:
  - **Scope 0**
    - `int a` (Variable Declaration)
    - `a = 1` (Assignment)
  - **Scope 1**
    - `string a` (Variable Declaration)
    - `a = "a"` (Assignment)
    - `print(a)` (Print)
    - `string b` (Variable Declaration)
    - `b = "b"` (Assignment)
    - `if (a == 1) { print(b) }` (Conditional Statement)

**Steps**:
- 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`

**Flow**:
- **Block**
  - **VarDecl** → Assign → **Block**
  - **VarDecl** → Assign → **Print**
  - **VarDecl** → Assign → **If**
  - **VarDecl** → Assign → **Print**

**Current Scope**:
- `a` in **Scope 1**
- `b` in **Scope 0**

**Additional Info**:
- `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
Semantic Analysis

Source Code

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

Symbol Table
Source Code
{
    int a
    a = 1
    {
        string a
        a = "a"
        print(a)
    }
    string b
    b = "b"
    if (a == 1) {
        print(b)
    }
}
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

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

Symbol Table
Source Code
{
    int a
    a = 1
    {
        string a
        a = "a"
        print(a)
    }
    string b
    b = "b"
    if (a == 1) {
        print(b)
    }
}
Semantic Analysis

Source Code
{
    int a
    a = 1
    {
        string a
        a = "a"
        print(a)
    }
    string b
    b = "b"
    if (a == 1) {
        print(b)
    }
}
Semantic Analysis

Source Code

```c
{ 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)
    }
}
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

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

There's an error here. Where is it?
```
Semantic Analysis

Yet another example in our language

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

Type-mismatch error.
Semantic Analysis

Yet another example in our language

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

Type-mismatch error.

Wait. Is this okay?
Semantic Analysis

Yet another example in our language

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

Type-mismatch error.

Wait. Is this okay? Yes.
Semantic Analysis

Source Code
{
  int a
  a = 1
  {
    string b
    b = a
    print(a)
  }
  string b
  b = "b"
  if (a == 1) {
    print(b)
  }
}
Semantic Analysis

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

Symbol Table

AST

Block

VarDecl → Assign → Block → VarDecl → Assign → If

int a a 1
VarDecl → Assign → Print
string b b a a
string b b a a

If

Initialize Scope 0
Set the current scope pointer.

current scope

Scope 0
Semantic Analysis

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

Symbol Table

Initialize Scope 0
add symbol a
in the current scope
Semantic Analysis

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

Symbol Table

Initialize Scope 0
add symbol a
lookup symbol a in the current scope
Semantic Analysis

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

Scope 0

Symbol Table
Semantic Analysis

Source Code

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

Symbol Table
Semantic Analysis

Source Code
{
    int a
    a = 1
    {
        string b
        b = a
        print(a)
    }
    string b
    b = "b"
    if (a == 1) {
        print(b)
    }
}
Semantic Analysis

Source Code
{
    int a
    a = 1
    {
        string b
        b = a  // b = a
        print(a)
    }
    string b
    b = "b"
    if (a == 1) {
        print(b)
    }
}
Semantic Analysis

Source Code

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

Symbol Table

- **Scope 0**
  - a | int (current scope)
- **Scope 1**
  - b | string
  - b = a
  - print(a)
  - b | string
  - b = "b"
  - if (a == 1) {
      print(b)
    }

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 = "b"
    if (a == 1) {
        print(b)
    }
}

Symbol Table

- VarDecl → Assign → Block
  - int a
  - a = 1
  - VarDecl → Assign → Print
    - string b
    - b = a
    - Print
      - a = 1

- Block
- VarDecl → Assign → If
  - string b
  - b = "b"
  - isEq
  - Block

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

Semantic Analysis
Semantic Analysis

Source Code

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

Symbol Table

- `int a`
- `string b`

Type Mismatch error
Semantic Analysis

Still another example in our language

```c
{
    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

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

Symbol Table
Initialize Scope 0

Set the current scope pointer.

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

Source Code

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

Scope 0

VarDecl → Assign → Block → VarDecl → Assign → If

Initialize Scope 0
add symbol c
lookup symbol c
current scope
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 = a
        print(a)
    }
    string b
    b = "b"
    if (a == 1) {
        print(b)
    }
}
Semantic Analysis

Source Code

```c
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)
    }
}
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

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

Symbol Table
Semantic Analysis

One more example in our language

```plaintext
{
    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

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

Symbol Table
**Semantic Analysis**

**Source Code**

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

**AST**

- Block
  - VarDecl
    - Assign
      - Block
        - VarDecl
          - Assign
            - Print
          - Block
            - VarDecl
              - Assign
                - Print
              - If
                - Block
```
Semantic Analysis

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

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

Symbol Table

```
Scope 0
```

```
Semantic Analysis
```

```
Block
  VarDecl
  Assign
  Block
  VarDecl
  Assign
  Block
```

```
Semantic Analysis
```

```
Symbol Table
```
Semantic Analysis

Source Code
{
    int c
    c = 1
    {
        string b
        b = "on my"
        print(b)
    }
    string c
    b = "b"
    if (a == 1) {
        print(b)
    }
}

Symbol Table
Semantic Analysis

Source Code

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

Block

VarDecl → Assign → Block

int c

c = 1

Assign → Block

VarDecl → Assign → Print

string c

b = "b"

if (a == 1) {
  print(b)
}

Print

VarDecl → Assign

string c

b = "oh my"

Scope 0

Symbol Table

current scope
Semantic Analysis

Source Code
{
    int c
    c = 1
    {
        string b
        b = "on my"
        print(b)
    }
    string c
    b = "b"
    if (a == 1) {
        print(b)
    }
}

Symbol Table
Source Code
{
    int c
    c = 1
    {
        string b
        b = "on my"
        print(b)
    }
    string c
    b = "b"
    if (a == 1) {
        print(b)
    }
}
Semantic Analysis

Source Code
{
    int c
    c = 1
    {
        string b
        b = "on 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 = "on 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 = "on 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 = "on 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
close 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 = "on my"
        print(b)
    }
    string c
    b = "b"
    if (a == 1) {
        print(b)
    }
}

Symbol Table
Semantic Analysis

One more example in our language

```c
{
    int c
    c = 1
    {
        string b
        b = "oh my"
        print(b)
    }
    string c → Identifier redeclared in the same scope error.
    b = "b"
    if (a == 1) {
        print(b)
    }
}
```

What else is wrong here?
Semantic Analysis

Source Code
{
    int c
    c = 1
    {
        string b
        b = "on my"
        print(b)
    }
    string c
    b = "b"
    if (a == 1) {
        print(b)
    }
}

Symbol Table

The identifiers a and b are undeclared in this scope.
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:

- Static (compile time) or Dynamic (runtime) type checking?
- Strong or weak typing?

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