Symbol Tables and Scope Checking
Where We Are

- Program is **lexically** well-formed:
  - Identifiers have valid names.
  - Strings are properly terminated.
  - No stray characters.
- Program is **syntactically** well-formed:
  - Class declarations have the correct structure.
  - Expressions are syntactically valid.
- Does this mean that the program is **legal**?
Beyond Syntax Errors

• What's wrong with this C code? (Note: it parses correctly)

foo(int a, char * s){...}

int bar() {
    int f[3];
    int i, j, k;
    char q, *p;
    float k;
    foo(f[6], 10, j);
    break;
    i->val = 42;
    j = m + k;
    printf("%s,%s.
",p,q);
    goto label42;
}
Semantic Analysis

● Ensure that the program has a well-defined meaning.
● Verify properties of the program that aren't caught during the earlier phases:
  – Variables are declared before they're used.
  – Expressions have the right types.
  – ...
● Once we finish semantic analysis, we know that the user's input program is legal.
Challenges in Semantic Analysis

- Reject the largest number of incorrect programs.
- Accept the largest number of correct programs.
- Do so quickly.
Other Goals of Semantic Analysis

• Gather useful information about program for later phases:
  • Determine what variables are meant by each identifier.
  • Build an internal representation of inheritance hierarchies.
  • Count how many variables are in scope at each point.
Scope Checking
What's in a Name?

• The same name in a program may refer to fundamentally different things:

• This is perfectly legal Java code:

```
public class Name {
    int Name;
    Name Name(Name Name) {
        Name.Name = 137;
        return Name((Name) Name);
    }
}
```
What's in a Name?

• The same name in a program may refer to fundamentally different things:

• This is **perfectly legal** Java code:

```java
public class Name {
    int Name;
    Name Name(Name Name) {
        Name.Name = 137;
        return Name((Name) Name);
    }
}
```
What's in a Name?

- The same name in a program may refer to completely different objects:
- This is perfectly legal C++ code:

```cpp
int Awful() {
    int x = 137;
    {
        string x = "Scope!"
        if (float x = 0)
            double x = x;
    }
    if (x == 137) cout << "Y";
}
```
What's in a Name?

• The same name in a program may refer to completely different objects:

• This is perfectly legal C++ code:

```cpp
int Awful() {
    int x = 137;
    {
        string x = "Scope!"
        if (float x = 0)
            double x = x;
    }
    if (x == 137) cout << "Y";
}
```
Scope

• The **scope** of an entity is the set of locations in a program where its name refers to itself.

• The introduction of new variables into scope may hide older variables.

• How do we keep track of what's visible?
Symbol Tables

- A **symbol table** is a mapping from a name to the thing that name refers to.
- As we run our semantic analysis, continuously update the symbol table with information about what is in scope.

Questions:
- What does this look like in practice?
- What operations need to be defined on it?
- How do we implement it?
0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x, y, z);
4:   {
5:     int x, z;
6:     z = y;
7:     x = z;
8:     {
9:       int y = x;
10:      {
11:         printf("%d,%d,%d\n", x, y, z);
12:       }
13:       printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x, y, z);
4:   {
5:     int x, z;
6:     z = y;
7:     x = z;
8:     {
9:       int y = x;
10:       {
11:         printf("%d,%d,%d\n", x, y, z);
12:       }
13:       printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }

Symbol Tables: The Intuition

0:  int x = 137;
1:  int z = 42;
2:  int MyFunction(int x, int y) {
3:    printf("%d,%d,%d\n", x, y, z);
4:    {
5:      int x, z;
6:      z = y;
7:      x = z;
8:      {
9:        int y = x;
10:       {
11:          printf("%d,%d,%d\n", x, y, z);
12:        }
13:        printf("%d,%d,%d\n", x, y, z);
14:      }
15:      printf("%d,%d,%d\n", x, y, z);
16:    }
17:  }

Symbol Table
Symbol Tables: The Intuition

0: `int x = 137;`
1: `int z = 42;`
2: `int MyFunction(int x, int y) {
3:     printf("%d,%d,%d\n", x, y, z);
4:     {
5:         int x, z;
6:         z = y;
7:         x = z;
8:         {
9:             int y = x;
10:             {
11:                 printf("%d,%d,%d\n", x, y, z);
12:             }
13:             printf("%d,%d,%d\n", x, y, z);
14:         }
15:         printf("%d,%d,%d\n", x, y, z);
16:     }
17: }`
0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x, y, z);
4:   {
5:     int x, z;
6:     z = y;
7:     x = z;
8:     {
9:       int y = x;
10:      {
11:        printf("%d,%d,%d\n", x, y, z);
12:      }
13:      printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x, y, z);
4:   {
5:     int x, z;
6:     z = y;
7:     x = z;
8:     {
9:       int y = x;
10:       {
11:         printf("%d,%d,%d\n", x, y, z);
12:       }
13:       printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }

Symbol Table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>Z</td>
<td>1</td>
</tr>
</tbody>
</table>
0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x, y, z);
4:   {
5:     int x, z;
6:     z = y;
7:     x = z;
8:     {
9:       int y = x;
10:      {
11:         printf("%d,%d,%d\n", x, y, z);
12:      }
13:      printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }

Symbol Table

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>z</td>
<td>1</td>
</tr>
</tbody>
</table>
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x, y, z);
4:   {
5:     int x, z;
6:     z = y;
7:     x = z;
8:     {
9:       int y = x;
10:      {
11:        printf("%d,%d,%d\n", x, y, z);
12:      }
13:      printf("%d,%d,%d\n", x, y, z);
14:    }
15:    printf("%d,%d,%d\n", x, y, z);
16:  }
17: }

Symbol Table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>z</td>
<td>1</td>
</tr>
</tbody>
</table>
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x, y, z);
4:   {
5:     int x, z;
6:     z = y;
7:     x = z;
8:     {
9:       int y = x;
10:      {
11:         printf("%d,%d,%d\n", x, y, z);
12:       }
13:       printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x, y, z);
4:   {
5:     int x, z;
6:     z = y;
7:     x = z;
8:     {
9:       int y = x;
10:      {
11:        printf("%d,%d,%d\n", x, y, z);
12:      }
13:      printf("%d,%d,%d\n", x, y, z);
14:    }
15:      printf("%d,%d,%d\n", x, y, z);
16:   }
17: }

Symbol Table

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>Z</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>2</td>
</tr>
<tr>
<td>Y</td>
<td>2</td>
</tr>
</tbody>
</table>
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x, y, z);
4:   {
5:     int x, z;
6:     z = y;
7:     x = z;
8:     {
9:       int y = x;
10:       {
11:         printf("%d,%d,%d\n", x, y, z);
12:       }
13:         printf("%d,%d,%d\n", x, y, z);
14:       }
15:         printf("%d,%d,%d\n", x, y, z);
16:     }
17:   }

Symbol Table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>z</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
</tr>
</tbody>
</table>
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x, y, z);
4:   {
5:     int x, z;
6:     z = y;
7:     x = z;
8:     {
9:       int y = x;
10:      {
11:        printf("%d,%d,%d\n", x, y, z);
12:      }
13:      printf("%d,%d,%d\n", x, y, z);
14:    }
15:   printf("%d,%d,%d\n", x, y, z);
16: }
17: }

Symbol Table

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
4:   {
5:     int x, z;
6:     z = y;
7:     x = z;
8:     {
9:       int y = x;
10:      {
11:        printf("%d,%d,%d\n", x, y, z);
12:      }
13:      printf("%d,%d,%d\n", x, y, z);
14:    }}
15:    printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
Symbol Tables: The Intuition

```c
0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x @2, y@2, z@1);
4:   {
5:     int x, z;
6:     z = y;
7:     x = z;
8:     {
9:       int y = x;
10:      {
11:         printf("%d,%d,%d\n", x, y, z);
12:       }
13:       printf("%d,%d,%d\n", x, y, z);
14:      }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x @2, y @2, z @1);
4:   {
5:     int x, z;
6:     z = y;
7:     x = z;
8:       {
9:         int y = x;
10:            {
11:               printf("%d,%d,%d\n", x, y, z);
12:            }
13:               printf("%d,%d,%d\n", x, y, z);
14:            }
15:               printf("%d,%d,%d\n", x, y, z);
16:       }
17:   }

Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
   printf("%d,%d,%d\n", x @2, y@2, z@1);
   {
      int x, z;
      z = y;
      x = z;
      {
         int y = x;
         {
            printf("%d,%d,%d\n", x, y, z);
         }
         printf("%d,%d,%d\n", x, y, z);
      }
      printf("%d,%d,%d\n", x, y, z);
   }
   printf("%d,%d,%d\n", x, y, z);
}

Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x @2, y @2, z @1);
4:   {
5:     int x, z;
6:     z = y;
7:     x = z;
8:     {
9:       int y = x;
10:       {
11:         printf("%d,%d,%d\n", x, y, z);
12:       }
13:       printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }

Symbol Table

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>Z</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>2</td>
</tr>
<tr>
<td>Y</td>
<td>2</td>
</tr>
<tr>
<td>X</td>
<td>5</td>
</tr>
<tr>
<td>Z</td>
<td>5</td>
</tr>
</tbody>
</table>
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x @2, y @2, z @1);
4:   {
5:     int x, z;
6:     z = y;
7:     x = z;
8:     {
9:       int y = x;
10:      {
11:        printf("%d,%d,%d\n", x, y, z);
12:      }
13:      printf("%d,%d,%d\n", x, y, z);
14:    }
15:    printf("%d,%d,%d\n", x, y, z);
16:  }
17: }

Symbol Table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>z</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td>x</td>
<td>5</td>
</tr>
<tr>
<td>z</td>
<td>5</td>
</tr>
</tbody>
</table>
0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:     printf("%d,%d,%d\n", x @2, y@2, z@1);
4:     {
5:         int x, z;
6:         z = y@2;
7:         x = z;
8:         {
9:             int y = x;
10:             {
11:                 printf("%d,%d,%d\n", x, y, z);
12:             }
13:             printf("%d,%d,%d\n", x, y, z);
14:             }
15:             printf("%d,%d,%d\n", x, y, z);
16:         }
17:     }

Symbol Table

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
<td>x</td>
<td>2</td>
<td>x</td>
</tr>
<tr>
<td>z</td>
<td>1</td>
<td>y</td>
<td>2</td>
<td>z</td>
</tr>
</tbody>
</table>
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x @2, y @2, z @1);
4:   {
5:     int x, z;
6:     z = y @2;
7:     x = z;
8:     {
9:       int y = x;
10:      {
11:         printf("%d,%d,%d\n", x, y, z);
12:      }
13:      printf("%d,%d,%d\n", x, y, z);
14:    }
15:      printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:     printf("%d,%d,%d\n", x @2, y@2, z@1);
4:     {
5:         int x, z;
6:         z = y@2;
7:         x = z@5;
8:         {
9:             int y = x;
10:            {
11:                printf("%d,%d,%d\n", x, y, z);
12:            }
13:                printf("%d,%d,%d\n", x, y, z);
14:            }
15:                printf("%d,%d,%d\n", x, y, z);
16:         }
17:     }
}
0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:     printf("%d,%d,%d\n", x, y, z);
4:     {
5:         int x, z;
6:         z = y;
7:         x = z;
8:         {
9:             int y = x;
10:             {
11:                 printf("%d,%d,%d\n", x, y, z);
12:             }
13:             printf("%d,%d,%d\n", x, y, z);
14:         }
15:         printf("%d,%d,%d\n", x, y, z);
16:     }
17: }

<table>
<thead>
<tr>
<th>Symbol Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>z</td>
</tr>
<tr>
<td>y</td>
</tr>
<tr>
<td>x</td>
</tr>
<tr>
<td>y</td>
</tr>
<tr>
<td>z</td>
</tr>
<tr>
<td>x</td>
</tr>
<tr>
<td>z</td>
</tr>
</tbody>
</table>
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x @2, y @2, z @1);
4:   {
5:     int x, z;
6:     z = y @2;
7:     x = z @5;
8:       {
9:         int y = x;
10:            printf("%d,%d,%d\n", x, y, z);
11:       }
12:       printf("%d,%d,%d\n", x, y, z);
13:       printf("%d,%d,%d\n", x, y, z);
14:     }
15:   printf("%d,%d,%d\n", x, y, z);
16:  }
17: }
0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
   printf("%d,%d,%d\n", x @2, y@2, z@1);
   {
      int x, z;
      z = y@2;
      x = z@5;
      { 
         int y = x;
         { 
            printf("%d,%d,%d\n", x, y, z);
         }
         printf("%d,%d,%d\n", x, y, z);
      }
      printf("%d,%d,%d\n", x, y, z);
   }
   printf("%d,%d,%d\n", x, y, z);
}
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x @2, y @2, z @1);
4:   {
5:     int x, z;
6:     z = y @2;
7:     x = z @5;
8:     {
9:       int y = x;
10:      {
11:        printf("%d,%d,%d\n", x, y, z);
12:      }
13:      printf("%d,%d,%d\n", x, y, z);
14:    }
15:    printf("%d,%d,%d\n", x, y, z);
16:  }
17: }

Symbol Table

- x 0
- z 1
- y 2
- x 2
- y 2
- x 5
- z 5
- y 9
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
4:   {
5:     int x, z;
6:     z = y@2;
7:     x = z@5;
8:     {
9:       int y = x@5;
10:      {
11:        printf("%d,%d,%d\n", x, y, z);
12:      }
13:      printf("%d,%d,%d\n", x, y, z);
14:    }
15:    printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x @2, y @2, z @1);
4:   {
5:     int x, z;
6:     z = y @2;
7:     x = z @5;
8:     {
9:       int y = x @5;
10:      {
11:        printf("%d,%d,%d\n", x, y, z);
12:      }
13:      printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x @2, y@2, z@1);
4:   {
5:     int x, z;
6:     z = y@2;
7:     x = z@5;
8:     {
9:       int y = x@5;
10:      {
11:        printf("%d,%d,%d\n", x, y, z);
12:      }
13:      printf("%d,%d,%d\n", x, y, z);
14:    }
15:    printf("%d,%d,%d\n", x, y, z);
16:  }
17: }

Symbol Table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>z</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td>x</td>
<td>5</td>
</tr>
<tr>
<td>z</td>
<td>5</td>
</tr>
<tr>
<td>y</td>
<td>9</td>
</tr>
</tbody>
</table>
Symbol Tables: The Intuition

```
0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
4:   {
5:     int x, z;
6:     z = y@2;
7:     x = z@5;
8:     {
9:       int y = x@5;
10:      {
11:         printf("%d,%d,%d\n", x, y, z);
12:      }
13:      printf("%d,%d,%d\n", x, y, z);
14:    }
15:    printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```

<table>
<thead>
<tr>
<th>Symbol Table</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>Z</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td>x</td>
<td>5</td>
</tr>
<tr>
<td>Z</td>
<td>5</td>
</tr>
<tr>
<td>y</td>
<td>9</td>
</tr>
</tbody>
</table>
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x @2, y@2, z@1);
4:   {
5:     int x, z;
6:     z = y@2;
7:     x = z@5;
8:     {
9:       int y = x@5;
10:      {
11:        printf("%d,%d,%d\n", x@5, y@9, z@5);
12:      }
13:      printf("%d,%d,%d\n", x, y, z);
14:    }
15:    printf("%d,%d,%d\n", x, y, z);
16:  }
17: }

Symbol Table

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>5</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>y</td>
</tr>
</tbody>
</table>
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x @2, y@2, z@1);
4:   {
5:     int x, z;
6:     z = y@2;
7:     x = z@5;
8:     {
9:       int y = x@5;
10:      {
11:        printf("%d,%d,%d\n", x @5, y@9, z@5);
12:      }
13:      printf("%d,%d,%d\n", x, y, z);
14:      }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }

Symbol Table

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x @2, y@2, z@1);
4:   {
5:     int x, z;
6:     z = y@2;
7:     x = z@5;
8:     {
9:       int y = x@5;
10:      {
11:        printf("%d,%d,%d\n", x @5, y@9, z@5);
12:      }
13:      printf("%d,%d,%d\n", x, y, z);
14:    }
15:    printf("%d,%d,%d\n", x, y, z);
16:  }
17: }

Symbol Table

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>Z</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td>x</td>
<td>5</td>
</tr>
<tr>
<td>z</td>
<td>5</td>
</tr>
<tr>
<td>Y</td>
<td>9</td>
</tr>
</tbody>
</table>
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x @2, y@2, z@1);
4:   {
5:     int x, z;
6:     z = y@2;
7:     x = z@5;
8:     {
9:       int y = x@5;
10:      {
11:        printf("%d,%d,%d\n", x @5, y@9, z@5);
12:      }
13:      printf("%d,%d,%d\n", x, y, z);
14:    }
15:    printf("%d,%d,%d\n", x, y, z);
16:  }
17: }

Symbol Table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>z</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td>x</td>
<td>5</td>
</tr>
<tr>
<td>z</td>
<td>5</td>
</tr>
<tr>
<td>y</td>
<td>9</td>
</tr>
</tbody>
</table>
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x @2, y @2, z @1);
4:   {
5:     int x, z;
6:     z = y @2;
7:     x = z @5;
8:     {
9:       int y = x @5;
10:      {
11:        printf("%d,%d,%d\n", x @5, y @9, z @5);
12:      }
13:      printf("%d,%d,%d\n", x @5, y @9, z @5);
14:      }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }

Symbol Table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>z</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td>x</td>
<td>5</td>
</tr>
<tr>
<td>y</td>
<td>9</td>
</tr>
<tr>
<td>z</td>
<td>5</td>
</tr>
<tr>
<td>y</td>
<td>9</td>
</tr>
</tbody>
</table>
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
4:   {
5:     int x, z;
6:     z = y@2;
7:     x = z@5;
8:     {
9:       int y = x@5;
10:      {
11:        printf("%d,%d,%d\n", x@5, y@9, z@5);
12:      }
13:      printf("%d,%d,%d\n", x@5, y@9, z@5);
14:    }    
15:    printf("%d,%d,%d\n", x, y, z);
16:  }
17: }

Symbol Table

<table>
<thead>
<tr>
<th>Variable</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>z</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td>x</td>
<td>5</td>
</tr>
<tr>
<td>z</td>
<td>5</td>
</tr>
<tr>
<td>y</td>
<td>9</td>
</tr>
</tbody>
</table>
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
4:   {
5:     int x, z;
6:     z = y@2;
7:     x = z@5;
8:     {
9:       int y = x@5;
10:      {
11:        printf("%d,%d,%d\n", x@5, y@9, z@5);
12:      }
13:        printf("%d,%d,%d\n", x@5, y@9, z@5);
14:      }
15:      printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
4:   {
5:     int x, z;
6:     z = y@2;
7:     x = z@5;
8:     {
9:       int y = x@5;
10:      {
11:        printf("%d,%d,%d\n", x@5, y@9, z@5);
12:      }
13:        printf("%d,%d,%d\n", x@5, y@9, z@5);
14:      }
15:      printf("%d,%d,%d\n", x, y, z);
16:     }
17:   }
}
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
4:   {
5:     int x, z;
6:     z = y@2;
7:     x = z@5;
8:     {
9:       int y = x@5;
10:      {
11:        printf("%d,%d,%d\n", x@5, y@9, z@5);
12:      }
13:      printf("%d,%d,%d\n", x@5, y@9, z@5);
14:     }
15:    printf("%d,%d,%d\n", x@5, y@2, z@5);
16:   }
17: }
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x @2, y @2, z @1);
4:   {
5:     int x, z;
6:     z = y @2;
7:     x = z @5;
8:     {
9:       int y = x @5;
10:      {
11:         printf("%d,%d,%d\n", x @5, y @9, z @5);
12:       }
13:       printf("%d,%d,%d\n", x @5, y @9, z @5);
14:     }
15:     printf("%d,%d,%d\n", x @5, y @2, z @5);
16:   }
17: }

Symbol Table

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
   printf("%d,%d,%d\n", x @2, y @2, z @1);
   {
      int x, z;
      z = y @2;
      x = z @5;
      {
         int y = x @5;
         {
            printf("%d,%d,%d\n", x @5, y @9, z @5);
         }
      }
      printf("%d,%d,%d\n", x @5, y @9, z @5);
      printf("%d,%d,%d\n", x @5, y @2, z @5);
   }
}
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x @2, y@2, z@1);
4:   {
5:     int x, z;
6:     z = y@2;
7:     x = z@5;
8:     {
9:       int y = x@5;
10:      {
11:        printf("%d,%d,%d\n", x @5, y@9, z@5);
12:      }
13:      printf("%d,%d,%d\n", x @5, y@9, z@5);
14:    }
15:    printf("%d,%d,%d\n", x @5, y@2, z@5);
16:  }
17: }

Symbol Table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>Z</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>2</td>
</tr>
<tr>
<td>Y</td>
<td>2</td>
</tr>
</tbody>
</table>
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x @2, y @2, z @1);
4:   {
5:     int x, z;
6:     z = y @2;
7:     x = z @5;
8:     {
9:       int y = x @5;
10:      {
11:        printf("%d,%d,%d\n", x @5, y @9, z @5);
12:      }
13:      printf("%d,%d,%d\n", x @5, y @9, z @5);
14:      }
15:      printf("%d,%d,%d\n", x @5, y @2, z @5);
16:    }
17:   }
}
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x @2, y @2, z @1);
4:   {
5:     int x, z;
6:     z = y @2;
7:     x = z @5;
8:     {
9:       int y = x @5;
10:      {
11:        printf("%d,%d,%d\n", x @5, y @9, z @5);
12:      }
13:      printf("%d,%d,%d\n", x @5, y @9, z @5);
14:      }
15:      printf("%d,%d,%d\n", x @5, y @2, z @5);
16:   }
17: }
Symbol Tables: The Intuition

0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
4:   {
5:     int x, z;
6:     z = y@2;
7:     x = z@5;
8:     {
9:       int y = x@5;
10:      {
11:        printf("%d,%d,%d\n", x@5, y@9, z@5);
12:      }
13:      printf("%d,%d,%d\n", x@5, y@9, z@5);
14:     }
15:     printf("%d,%d,%d\n", x@5, y@2, z@5);
16:   }
17: }

Symbol Table

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Symbol Table Operations

- Typically implemented as a **stack of tables**.
- Each table corresponds to a particular scope.
- Stack allows for easy “enter” and “exit” operations.
- Symbol table operations are
  - **Push scope**: Enter a new scope.
  - **Pop scope**: Leave a scope, discarding all declarations in it.
  - **Insert symbol**: Add a new entry to the current scope.
  - **Lookup symbol**: Find what a name corresponds to.
Using a Symbol Table

- To process a portion of the program that creates a scope (block statements, function calls, classes, etc.)
  - Enter a new scope.
  - Add all variable declarations to the symbol table.
  - Process the body of the block/function/class.
  - Exit the scope.
- Much of semantic analysis is defined in terms of recursive AST traversals like this.
Another View of Symbol Tables
Another View of Symbol Tables

```c
0: int x;
1: int y;
2: int MyFunction(int x, int y)
3: {
4:   int w, z;
5:   {
6:     int y;
7:   };
8:   {
9:     int w;
10:   };
11: }
```
0: int x;
1: int y;
2: int MyFunction(int x, int y)
3: {
4:   int w, z;
5:   {
6:     int y;
7:   }
8:   {
9:     int w;
10:   }
11: }

Root Scope
Another View of Symbol Tables

```
0: int x;
1: int y;
2: int MyFunction(int x, int y)
3: {
4:   int w, z;
5:   {
6:     int y;
7:   }
8:   {
9:     int w;
10:   }
11: }
```
Another View of Symbol Tables

0: `int x;`
1: `int y;`
2: `int MyFunction(int x, int y)`
3: {
4:   `int w, z;`
5:   {
6:     `int y;`
7:   }
8:   {
9:     `int w;`
10: }
11: }
Another View of Symbol Tables

0: int x;
1: int y;
2: int MyFunction(int x, int y)
3: {
4:   int w, z;
5:   {
6:     int y;
7:   }
8:   {
9:     int w;
10:   }
11: }
Another View of Symbol Tables

0: int x;
1: int y;
2: int MyFunction(int x, int y)
3: {
4:   int w, z;
5:   {
6:     int y;
7:   }
8:   {
9:     int w;
10: } 
11: }
Another View of Symbol Tables

0: int x;
1: int y;
2: int MyFunction(int x, int y)
3: {
4:   int w, z;
5:   {
6:     int y;
7:   }
8:   {
9:     int w;
10: }
11: }

Root Scope

x 0

y 1
Another View of Symbol Tables

0: int x;
1: int y;
2: int MyFunction(int x, int y)
3: {
4:   int w, z;
5:   {
6:     int y;
7:   }
8:   {
9:     int w;
10:   }
11: }

Root Scope

x 0
y 1
x 2
y 2
Another View of Symbol Tables

0: int x;
1: int y;
2: int MyFunction(int x, int y)
3: {
4:   int w, z;
5:   {
6:     int y;
7:   }
8:   {
9:     int w;
10: }
11: }

Root Scope

x 0

y 1

x 2

y 2
Another View of Symbol Tables

0: int x;
1: int y;
2: int MyFunction(int x, int y)
3: {
4:   int w, z;
5:   {
6:     int y;
7:   }   
8:   {
9:     int w;
10:   }
11: }

Root Scope

0: x
1: y
2: x
2: y
Another View of Symbol Tables

0: int x;
1: int y;
2: int MyFunction(int x, int y)
3: {
4:   int w, z;
5:   {
6:     int y;
7:   }
8:   {
9:     int w;
10:   }
11: }

Another View of Symbol Tables

```c
0: int x;
1: int y;
2: int MyFunction(int x, int y)
   3: {
   4:   int w, z;
   5:     { int y;
   6:       int w;
   7:     }
   8:   }
11: }
```
0: int x;
1: int y;
2: int MyFunction(int x, int y)
3: {
4:   int w, z;
5:   {
6:     int y;
7:   }
8:   {
9:     int w;
10:   }
11: }
Another View of Symbol Tables

0: int x;
1: int y;
2: int MyFunction(int x, int y)
3: {
4:   int w, z;
5:   {
6:     int y;
7:   }
8:   {
9:     int w;
10:   }
11: }
12: }
Another View of Symbol Tables

0: int x;
1: int y;
2: int MyFunction(int x, int y)
3: {
4:   int w, z;
5:   {
6:     int y;
7:   }
8:   {
9:     int w;
10:   }
11: }
Another View of Symbol Tables

0: int x;
1: int y;
2: int MyFunction(int x, int y)
3: {
4:   int w, z;
5:   {
6:     int y;
7:   }
8:   {
9:     int w;
10:   }
11: }
Another View of Symbol Tables

0: int x;
1: int y;
2: int MyFunction(int x, int y)
3: {
4:   int w, z;
5:   {
6:     int y;
7:   } 
8:   {
9:     int w;
10: } 
11: }
Another View of Symbol Tables

0: int x;
1: int y;
2: int MyFunction(int x, int y)
3: {
4:   int w, z;
5:   {
6:     int y;
7:   }
8:   {
9:     int w;
10: }
11: }

Root Scope

x 0
y 1
x 2
y 2
w 4
z 4
y 6
Another View of Symbol Tables

0: int x;
1: int y;
2: int MyFunction(int x, int y)
3: {
4:   int w, z;
5:   {
6:     int y;
7:   }
8:   {
9:     int w;
10: } 
11: } 
12: }
Another View of Symbol Tables

0: int x;
1: int y;
2: int MyFunction(int x, int y)
3: {
4:   int w, z;
5:   {
6:     int y;
7:   }
8:   {
9:     int w;
10:   }
11: }
Another View of Symbol Tables

```
0: int x;
1: int y;
2: int MyFunction(int x, int y)
3: {
4:   int w, z;
5:    {
6:      int y;
7:    }
8:    {
9:      int w;
10:   }
11: }
```
Another View of Symbol Tables

0: int x;
1: int y;
2: int MyFunction(int x, int y)
3: {
4:   int w, z;
5:   {
6:     int y;
7:   }
8:   {
9:     int w;
10:   }
11: }
12: }
Another View of Symbol Tables

0: int x;
1: int y;
2: int MyFunction(int x, int y)
3: {
4:   int w, z;
5:   {
6:     int y;
7:   }
8:   {
9:     int w;
10: }
11: }

Root Scope

x 0
y 1
x 2
y 2
w 4
z 4
Y 6
w 9
Another View of Symbol Tables

0: int x;
1: int y;
2: int MyFunction(int x, int y)
3: {
4:   int w, z;
5:   {
6:     int y;
7:   }
8:   {
9:     int w;
10:   }
11: }
12: }
Spaghetti Stacks

- Treat the symbol table as a linked structure of scopes.
- Each scope stores a pointer to its parents, but not vice-versa.
- From any point in the program, symbol table appears to be a stack.
- This is called a spaghetti stack.
Why Two Interpretations?

- Spaghetti stack more accurately captures the scoping structure.
- Spaghetti stack is a **static** structure; explicit stack is a **dynamic** structure.
- Explicit stack is an optimization of a spaghetti stack; more on that later.
Scoping in Object-Oriented Languages
Scoping with Inheritance

```java
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}
```
Scoping with Inheritance

public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

Root Scope
Scoping with Inheritance

```java
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}
```
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

public class Derived extends Base {
    public int derivedInt = 3;
    public int publicBaseInt = 4;

    public void doSomething() {
        System.out.println(publicBaseInt);
        System.out.println(baseInt);
        System.out.println(derivedInt);

        int publicBaseInt = 6;
        System.out.println(publicBaseInt);
    }
}
Scoping with Inheritance

```java
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

public class Derived extends Base {
    public int derivedInt = 3;
    public int publicBaseInt = 4;

    public void doSomething() {
        System.out.println(publicBaseInt);
        System.out.println(baseInt);
        System.out.println(derivedInt);
        int publicBaseInt = 6;
        System.out.println(publicBaseInt);
    }
}
```
Scoping with Inheritance

```java
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

public class Derived extends Base {
    public int derivedInt = 3;
    public int publicBaseInt = 4;

    public void doSomething() {
        System.out.println(publicBaseInt);
        System.out.println(baseInt);
        System.out.println(derivedInt);

        int publicBaseInt = 6;
        System.out.println(publicBaseInt);
    }
}
```
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

public class Derived extends Base {
    public int derivedInt = 3;
    public int publicBaseInt = 4;

    public void doSomething() {
        System.out.println(publicBaseInt);
        System.out.println(baseInt);
        System.out.println(derivedInt);

        int publicBaseInt = 6;
        System.out.println(publicBaseInt);
    }
}

Scoping with Inheritance
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

public class Derived extends Base {
    public int derivedInt = 3;
    public int publicBaseInt = 4;

    public void doSomething() {
        System.out.println(publicBaseInt);
        System.out.println(baseInt);
        System.out.println(derivedInt);

        int publicBaseInt = 6;
        System.out.println(publicBaseInt);
    }
}
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

public class Derived extends Base {
    public int derivedInt = 3;
    public int publicBaseInt = 4;

    public void doSomething() {
        System.out.println(publicBaseInt);
        System.out.println(baseInt);
        System.out.println(derivedInt);
        int publicBaseInt = 6;
        System.out.println(publicBaseInt);
    }
}

Scoping with Inheritance
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

public class Derived extends Base {
    public int derivedInt = 3;
    public int publicBaseInt = 4;

    public void doSomething() {
        System.out.println(publicBaseInt);
        System.out.println(baseInt);
        System.out.println(derivedInt);

        int publicBaseInt = 6;
        System.out.println(publicBaseInt);
    }
}

> 4
Scoping with Inheritance

public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

dynamic

public class Derived extends Base {
    public int derivedInt = 3;
    public int publicBaseInt = 4;

    public void doSomething() {
        System.out.println(publicBaseInt);
        System.out.println(baseInt);
        System.out.println(derivedInt);

        int publicBaseInt = 6;
        System.out.println(publicBaseInt);
    }
}
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

public class Derived extends Base {
    public int derivedInt = 3;
    public int publicBaseInt = 4;
    public void doSomething() {
        System.out.println(publicBaseInt);
        System.out.println(baseInt);
        System.out.println(derivedInt);
        int publicBaseInt = 6;
        System.out.println(publicBaseInt);
    }
}
Scoping with Inheritance

```java
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

public class Derived extends Base {
    public int derivedInt = 3;
    public int publicBaseInt = 4;

    public void doSomething() {
        System.out.println(publicBaseInt);
        System.out.println(baseInt);
        System.out.println(derivedInt);
        int publicBaseInt = 6;
        System.out.println(publicBaseInt);
    }
}
```

```
> 4
2
```
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

public class Derived extends Base {
    public int derivedInt = 3;
    public int publicBaseInt = 4;

    public void doSomething() {
        System.out.println(publicBaseInt);
        System.out.println(baseInt);
        System.out.println(derivedInt);

        int publicBaseInt = 6;
        System.out.println(publicBaseInt);
    }
}
Scoping with Inheritance

```java
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

public class Derived extends Base {
    public int derivedInt = 3;
    public int publicBaseInt = 4;

    public void doSomething() {
        System.out.println(publicBaseInt);
        System.out.println(baseInt);
        System.out.println(derivedInt);

        int publicBaseInt = 6;
        System.out.println(publicBaseInt);
    }
}
```
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

public class Derived extends Base {
    public int derivedInt = 3;
    public int publicBaseInt = 4;

    public void doSomething() {
        System.out.println(publicBaseInt);
        System.out.println(baseInt);
        System.out.println(derivedInt);

        int publicBaseInt = 6;
        System.out.println(publicBaseInt);
    }
}
Scoping with Inheritance

```java
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

class Derived extends Base {
    public int derivedInt = 3;
    public int publicBaseInt = 4;

    public void doSomething() {
        System.out.println(publicBaseInt);
        System.out.println(baseInt);
        System.out.println(derivedInt);

        int publicBaseInt = 6;
        System.out.println(publicBaseInt);
    }
}
```

The diagram illustrates the class hierarchy and variable scoping.
Scoping with Inheritance

```java
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

public class Derived extends Base {
    public int derivedInt = 3;
    public int publicBaseInt = 4;

    public void doSomething() {
        System.out.println(publicBaseInt);
        System.out.println(baseInt);
        System.out.println(derivedInt);

        int publicBaseInt = 6;
        System.out.println(publicBaseInt);
    }
}
```

> 4
  2
  3
  6

Root Scope

Base
- publicBaseInt 1
- baseInt 2

Derived
- derivedInt 3
- publicBaseInt 4

doSomething
- publicBaseInt 6
Scoping with Inheritance

public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

double class Derived extends Base {
    public int derivedInt = 3;
    public int publicBaseInt = 4;

    public void doSomething() {
        System.out.println(publicBaseInt);
        System.out.println(baseInt);
        System.out.println(derivedInt);

        int publicBaseInt = 6;
        System.out.println(publicBaseInt);
    }
}
Inheritance and Scoping

• Typically, the scope for a derived class will store a link to the scope of its base class.

• Looking up a field of a class traverses the scope chain until that field is found or a semantic error is found.
public class Base {
    public int value = 1;
}

double class Derived extends Base {
    public int value = 2;

    public void doSomething() {
        int value = 3;
        System.out.println(value);
        System.out.println(this.value);
        System.out.println(super.value);
    }
}
Explicit Disambiguation

```java
public class Base {
    public int value = 1;
}

direct class Derived extends Base {
    public int value = 2;

    public void doSomething() {
        int value = 3;
        System.out.println(value);
        System.out.println(this.value);
        System.out.println(super.value);
    }
}
```
public class Base {
    public int value = 1;
}

public class Derived extends Base {
    public int value = 2;

    public void doSomething() {
        int value = 3;
        System.out.println(value);
        System.out.println(this.value);
        System.out.println(super.value);
    }
}
public class Base {
    public int value = 1;
}

public class Derived extends Base {
    public int value = 2;

    public void doSomething() {
        int value = 3;
        System.out.println(value);
        System.out.println(this.value);
        System.out.println(super.value);
    }
}
public class Base {
    public int value = 1;
}

public class Derived extends Base {
    public int value = 2;
    public void doSomething() {
        int value = 3;
        System.out.println(value);
        System.out.println(this.value);
        System.out.println(super.value);
    }
}
public class Base {
    public int value = 1;
}

public class Derived extends Base {
    public int value = 2;

    public void doSomething() {
        int value = 3;
        System.out.println(value);
        System.out.println(this.value);
        System.out.println(super.value);
    }
}
Explicit Disambiguation

```java
public class Base {
    public int value = 1;
}

public class Derived extends Base {
    public int value = 2;

    public void doSomething() {
        int value = 3;
        System.out.println(value);
        System.out.println(this.value);
        System.out.println(super.value);
    }
}
```

```
> 3
```

Root Scope

Base

value

1

Derived

value

2

Local

this

value

3

doSomething

value

3
public class Base {
    public int value = 1;
}

public class Derived extends Base {
    public int value = 2;

    public void doSomething() {
        int value = 3;
        System.out.println(value);
        System.out.println(this.value);
        System.out.println(super.value);
    }
}
public class Base {
    public int value = 1;
}

public class Derived extends Base {
    public int value = 2;

    public void doSomething() {
        int value = 3;
        System.out.println(value);
        System.out.println(this.value);
        System.out.println(super.value);
    }
}
Explicit Disambiguation

```java
public class Base {
    public int value = 1;
}

public class Derived extends Base {
    public int value = 2;

    public void doSomething() {
        int value = 3;
        System.out.println(value);
        System.out.println(this.value);
        System.out.println(super.value);
    }
}
```
public class Base {
    public int value = 1;
}

public class Derived extends Base {
    public int value = 2;
    public void doSomething() {
        int value = 3;
        System.out.println(value);
        System.out.println(this.value);
        System.out.println(super.value);
    }
}
Explicit Disambiguation

```java
public class Base {
    public int value = 1;
}

public class Derived extends Base {
    public int value = 2;

    public void doSomething() {
        int value = 3;
        System.out.println(value);
        System.out.println(this.value);
        System.out.println(super.value);
    }
}
```
Disambiguating Scopes

- Maintain a second table of pointers into the scope stack.
- When looking up a value in a specific scope, begin the search from that scope.
- Some languages allow you to jump up to any arbitrary base class (for example, C++).
Scoping in Practice
Scoping in C++ and Java

class A {
public:
    /* ... */
private:
    B* myB;
};

class B {
public:
    /* ... */
private:
    A* myA;
};
Scoping in C++ and Java

class A {
public:
    /* ... */
private:
    B* myB;
};

class B {
public:
    /* ... */
private:
    A* myA;
};

class A {
private B myB;
};

class B {
    private A myA;
};

Error: B not declared

Perfectly fine!
Single- and Multi-Pass Compilers

- Our predictive parsing methods always scan the input from left-to-right.
  - LL(1), LR(0), LALR(1), etc.

- Since we only need one token of lookahead, we can do scanning and parsing simultaneously in one pass over the file.

- Some compilers can combine scanning, parsing, semantic analysis, and code generation into the same pass.
  - These are called **single-pass compilers**.

- Other compilers rescan the input multiple times.
  - These are called **multi-pass compilers**.
Single- and Multi-Pass Compilers

- Some languages are designed to support single-pass compilers.
  - e.g. C, C++.
- Some languages require multiple passes.
  - e.g. Java, C#.
- Most modern compilers use a huge number of passes over the input.
Scoping in Multi-Pass Compilers

- Completely parse the input file into an abstract syntax tree (first pass).
- Walk the AST, gathering information about classes (second pass).
- Walk the AST checking other properties (third pass).
- Could combine some of these, though they are logically distinct.
Scoping with Multiple Inheritance

class A {
public:
    int x;
};

class B {
};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
};
Scoping with Multiple Inheritance

```cpp
class A {
public:
    int x;
};

class B {
};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
```
Scoping with Multiple Inheritance

class A {
public:
    int x;
};

class B {
};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
Scoping with Multiple Inheritance

class A {
public:
    int x;
};

class B {
};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
class A {
public:
    int x;
};

class B {
};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
};
Scoping with Multiple Inheritance

```
class A {
    public:
        int x;
};

class B {
};

class C: public A, public B {
    public:
        void doSomething() {
            cout << x << endl;
        }
}
```
class A {
public:
    int x;
};

class B {
public:
    int x;
};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
class A {
public:
    int x;
};

class B {
public:
    int x;
};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
Scoping with Multiple Inheritance

class A {
public:
    int x;
};

class B {
public:
    int x;
};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
Scoping with Multiple Inheritance

```cpp
class A {
public:
    int x;
};

class B {
public:
    int x;
};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
};
```

Ambiguous – which x?
Scoping with Multiple Inheritance

class A {
public:
    int x;
};

class B {
public:
    int x;
};

class C: public A, public B {
public:
    void doSomething() {
        cout << A::x << endl;
    }
};
class A {
public:
    int x;
};

class B {
public:
    int x;
};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
};
Scoping with Multiple Inheritance

class A {
    public:
        int x;
};

class B {
};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
};
class A {
public:
    int x;
};

class B {
};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
Scoping with Multiple Inheritance

```cpp
int x;

class A {
public:
    int x;
};

class B {
};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
};
```
Scoping with Multiple Inheritance

```cpp
int x;

class A {
public:
    int x;
};

class B {
}

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
```
Scoping with Multiple Inheritance

```cpp
int x;

class A {
public:
    int x;
};

class B {
};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
```
(Simplified) C++ Scoping Rules

- Inside of a class, search the entire class hierarchy to see the set of names that can be found.
  - This uses the standard scoping lookup.
- If only one name is found, the lookup succeeds unambiguously.
- If more than one name is found, the lookup is ambiguous and requires disambiguation.
- Otherwise, restart the search from outside the class.
Summary

- **Semantic analysis** verifies that a syntactically valid program is correctly-formed and computes additional information about the meaning of the program.

- **Scope checking** determines what objects or classes are referred to by each name in the program.

- Scope checking is usually done with a **symbol table** implemented either as an explicit stack or a spaghetti stack.

- In object-oriented programs, the scope for a derived class is often placed inside of the scope of a base class.

- Some semantic analyzers operate in multiple passes in order to gain more information about the program.

- With multiple inheritance, a name may need to be searched for along multiple paths.