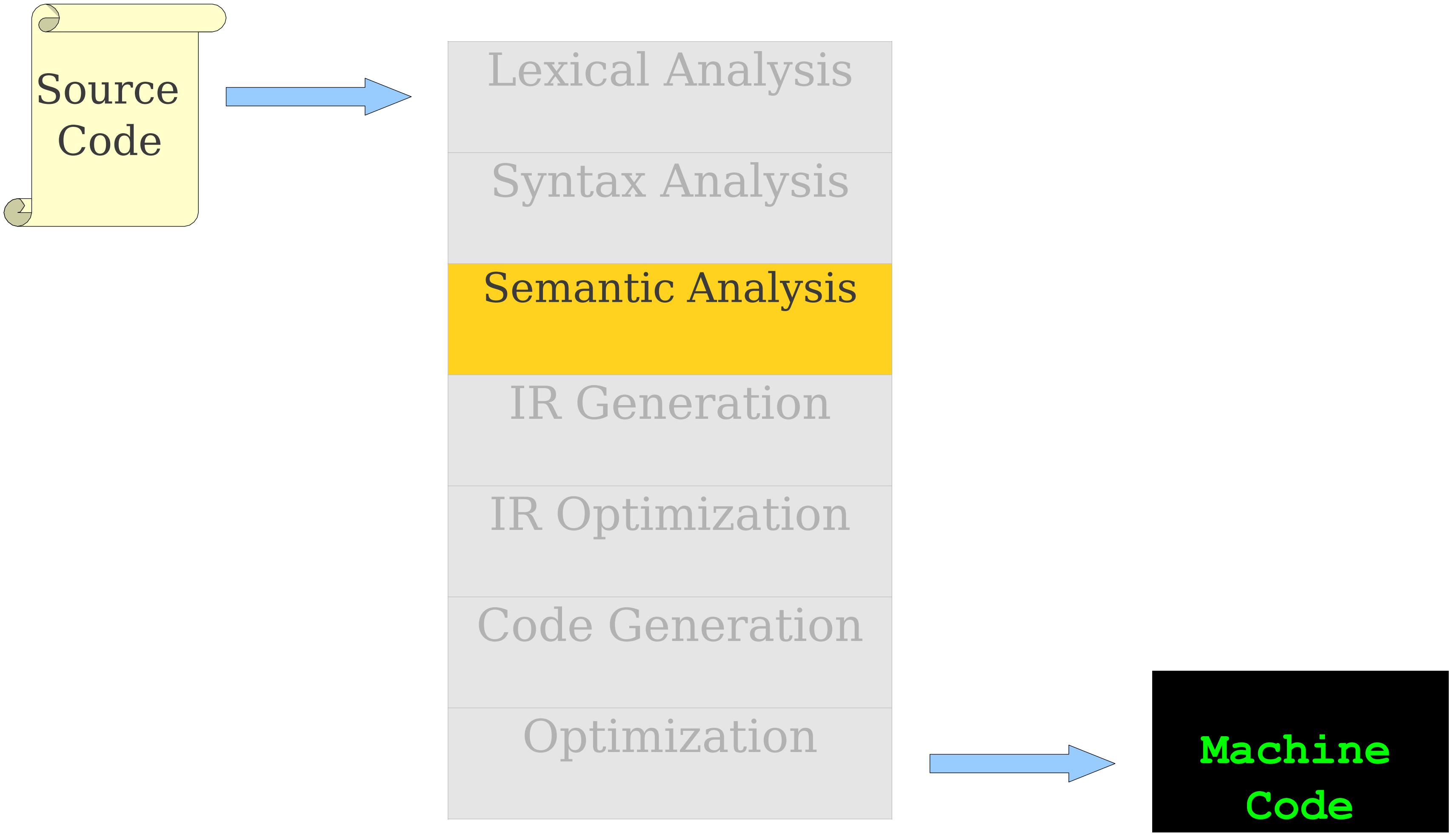
بسم الله الرحمن الرحيم

Semantic Analysis, Type checking

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Review from Last Time

```
class MyClass implements MyInterface
    {  string myInteger;

    void doSomething() {
         int[] x;
          x = new string;

          x[5] = myInteger * y;
     }
    void doSomething() {

     }
    int fibonacci(int n) {
          return doSomething() + fibonacci(n - 1);
     }
}
```

Review from Last Time

```
class MyClass implements MyInterface {
    string myInteger;

    void doSomething() {
        int[] x;
        x = new string;

        x[5] = myInteger * y;
    }
    void doSomething() {

    }
    int fibonacci(int n) {
        return doSomething() + fibonacci(n - 1);
    }
}
```

Interface not declared

Wrong type

Can't multiply strings

Variable not declared

Can't redefine functions

Can't add void

No main function

Review from Last Time

```
class MyClass implements MyInterface {
      string myInteger;

      void doSomething() {
            int[] x;
            x = new string;

            x[5] = myInteger * y;
      }
      void doSomething() {

      }
      int fibonacci(int n) {
            return doSomething() + fibonacci(n - 1);
      }
}
```

Wrong type

Can't multiply strings

Variable not declared

Can't redefine functions

Can't add void

No main function

Review from Last Time

```
class MyClass implements MyInterface {
    string myInteger;

    void doSomething() {
        int[] x;
        x = new string;

        x[5] = myInteger *  y;
    }
    void doSomething() {

    }
    int fibonacci(int n) {
        return doSomething() + fibonacci(n - 1);
    }
}
```

Wrong type

Can't multiply strings

Variable not declared

Can't add void

No main function

Review from Last Time

```
class MyClass implements MyInterface {
      string myInteger;

      void doSomething() {
            int[] x;
            x = new string;

            x[5] = myInteger * y;
      }
      void doSomething() {

      }
      int fibonacci(int n) {
            return doSomething() + fibonacci(n - 1);
      }
}
```

Wrong type

Can't multiply strings

Can't add void

No main function

Review from Last Time

```
class MyClass implements MyInterface {
    string myInteger;

    void doSomething() {
        int[] x;
        x = new string;

        x[5] = myInteger * y;
    }
    void doSomething() {

    }
    int fibonacci(int n) {
        return doSomething() + fibonacci(n - 1);
    }
}
```

Wrong type

Can't multiply strings

Can't add void

What Remains to Check?

- **Type errors**.
- Today:
  - What are types?
  - What is type-checking?
  - A type system for Decaf.

What is a Type?

- This is the subject of some debate.
- To quote Alex Aiken:
  - **“The notion varies from language to language**.
  - The consensus:
    - A set of values.
    - A set of operations on those values**”**
- **Type errors** arise when operations are performed on values that do not support that operation.

Types of Type-Checking

- **Static type checking.**
  - Analyze the program during compile-time to prove the absence of type errors.
- **Dynamic type checking.**
  - Check operations at runtime before performing them.
  - More precise than static type checking, but usually less efficient.
  - (Why?)
- **No type checking.**
  - Throw caution to the wind!

Type Systems

- The rules governing permissible operations on types forms a **type system**.
- **Strong type systems** are systems that never allow for a type error.
  - Java, Python, JavaScript, LISP, Haskell, etc.
- **Weak type systems** can allow type errors at runtime.
  - C, C++

Static vs Dynamic

- Static Pros:
  - Static checking catches many programming errors at compile time
  - Guarantees that all executions will be safe *but it may reject some type-safe programs*!
  - Avoids overhead of runtime type checks

- Dynamic Advantage
  - Static type systems are restrictive
  - Rapid prototyping difficult within a static type system

Type Wars

- *Endless* debate about what the “right” system is.
- Dynamic type systems make it easier to prototype; static type systems have fewer bugs.
- Strongly-typed languages are more robust, weakly-typed systems are often faster.
- **I'm staying out of this!**

Our Focus

- Decaf is typed **statically** and **weakly**:
  - Type-checking occurs at compile-time.
  - Runtime errors like dereferencing **`null`** or an invalid object are allowed.
- Decaf uses **class-based inheritance**.
- Decaf distinguishes primitive types and classes.

Typing in Decaf

Static Typing in Decaf

- Static type checking in Decaf consists of two separate processes:
  - Inferring the type of each expression from the types of its *components*.
  - Confirming that the types of expressions in certain contexts matches what is expected.
- Logically two steps, but you will probably combine into one pass.

An Example

```
while (numBitsSet(x + 5) <= 10) {

    if (1.0 + 4.0) {
        /* … */
    }

    while (5 == null) {
        /* … */
    }

}
```

An Example

```
while (numBitsSet(x + 5) <= 10) {

      if (1.0 + 4.0) {
            /* … */
      }

      while (5 == null) {
            /* … */
      }

}
```

An Example

```
while (numBitsSet(x + 5) <= 10) {

    if (1.0 + 4.0) {
        /* … */
    }

    while (5 == null) {
        /* … */
    }

}
```

An Example

```
while (numBitsSet(x + 5) <= 10) {

    if (1.0 + 4.0) {
        /* … */
    }

    while (5 == null) {
        /* … */
    }

}
```

Well-typed expression with wrong type.

An Example

```
while (numBitsSet(x + 5) <= 10) {

    if (1.0 + 4.0) {
        /* … */
    }

    while (5 == null) {
        /* … */
    }

}
```

An Example

```
while (numBitsSet(x + 5) <= 10) {

      if (1.0 + 4.0) {
            /* … */
      }

      while (5 == null) {
            /* … */
      }

}
```

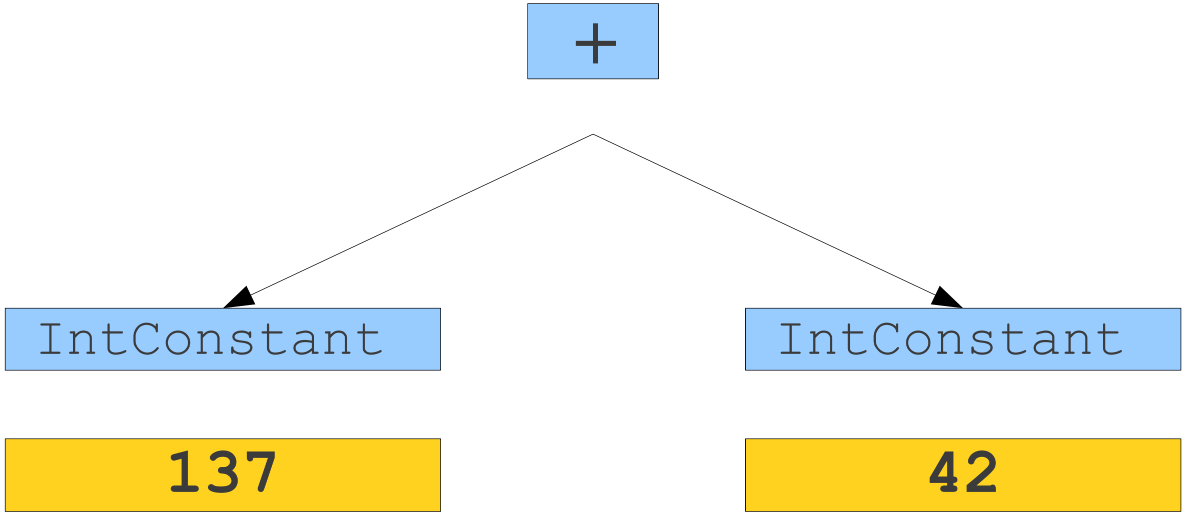
Expression with type error

Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as **logical inference**.

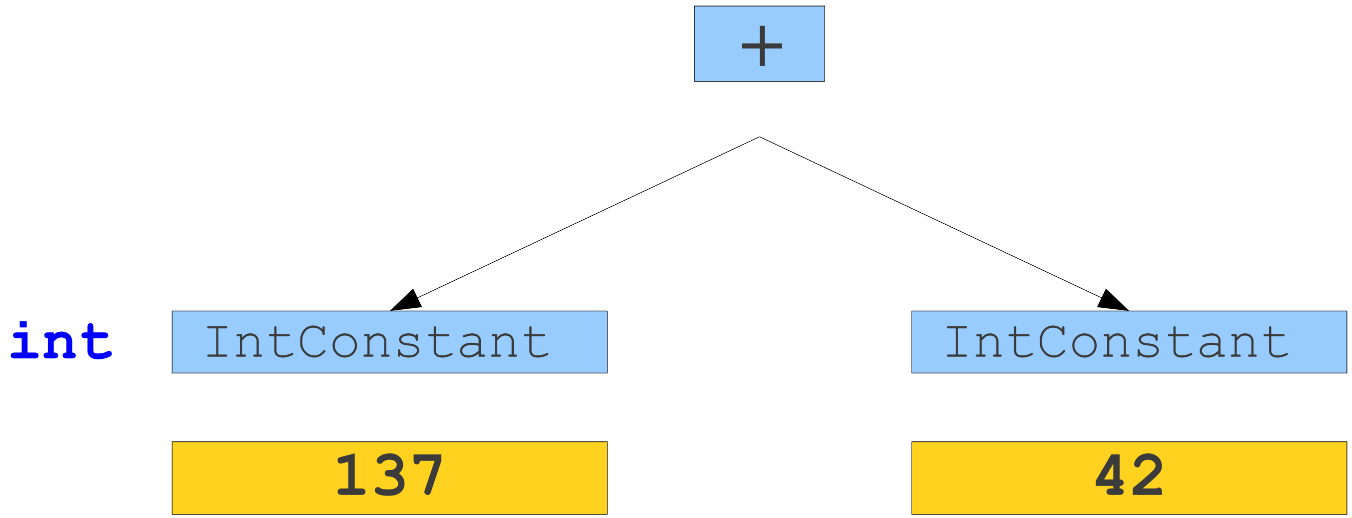
Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as **logical inference**.



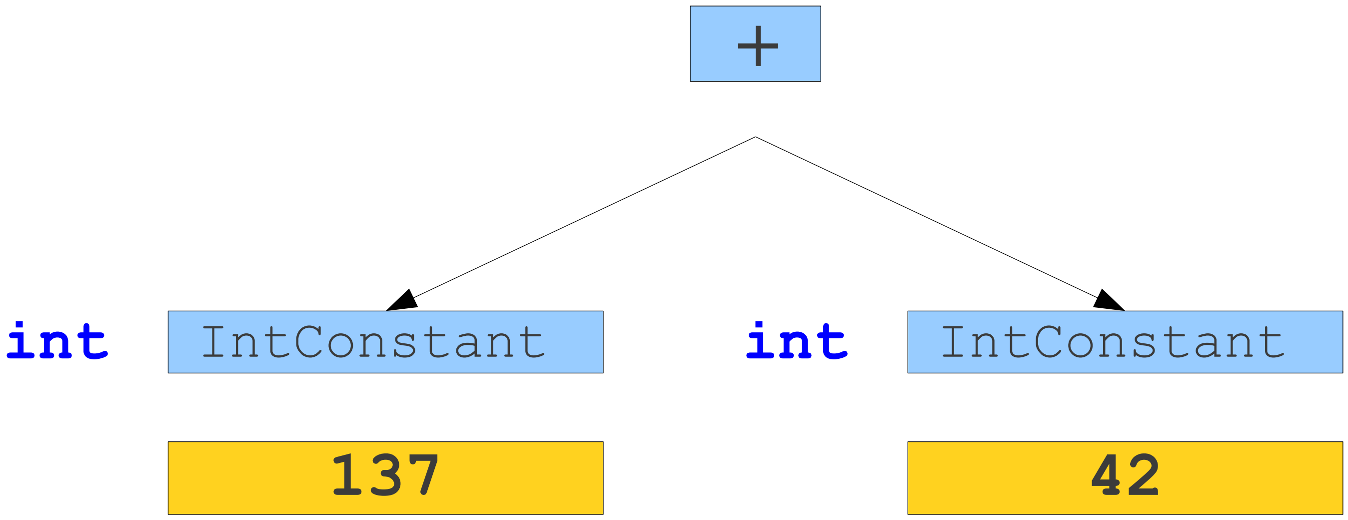
Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as **logical inference**.



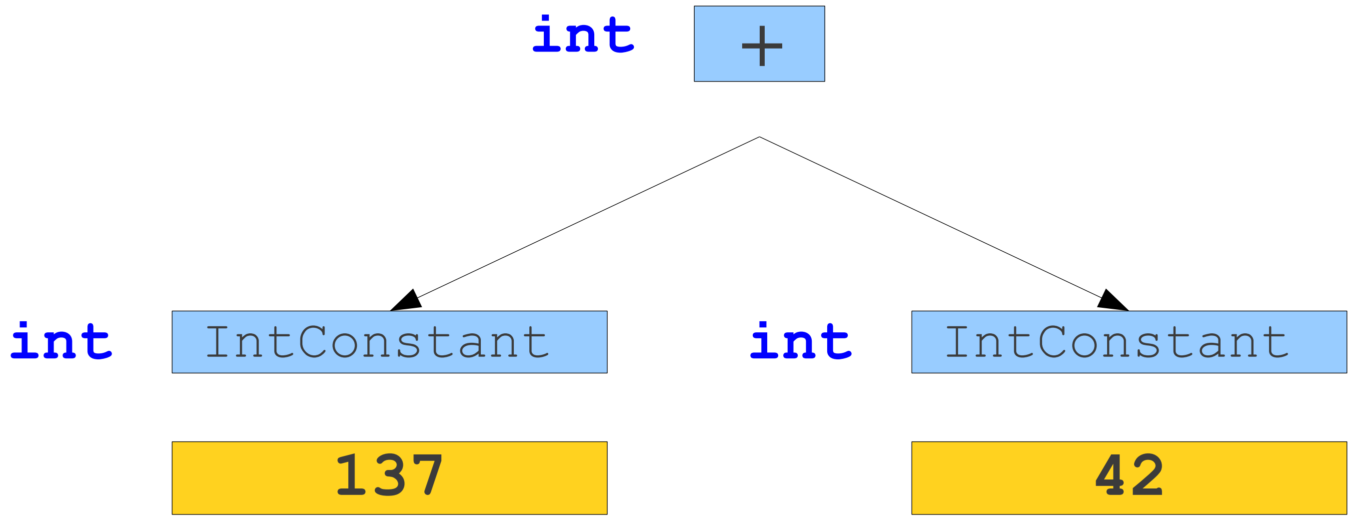
Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as **logical inference**.



Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as **logical inference**.

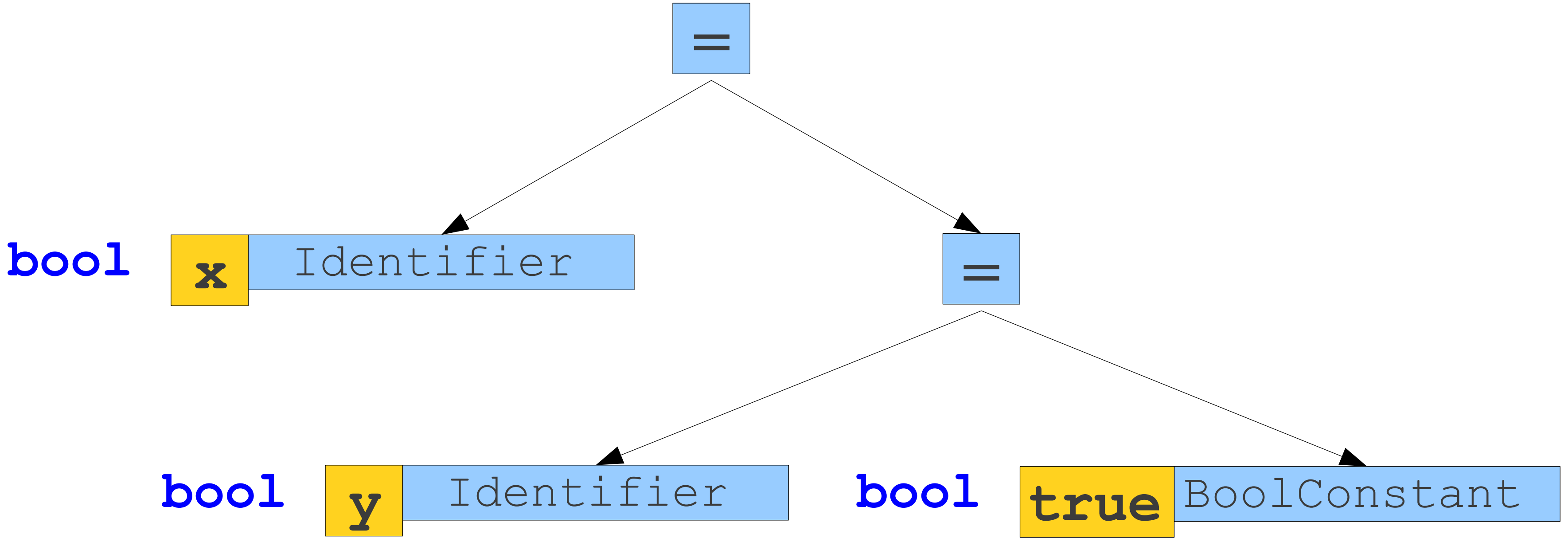


Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as **logical inference**.

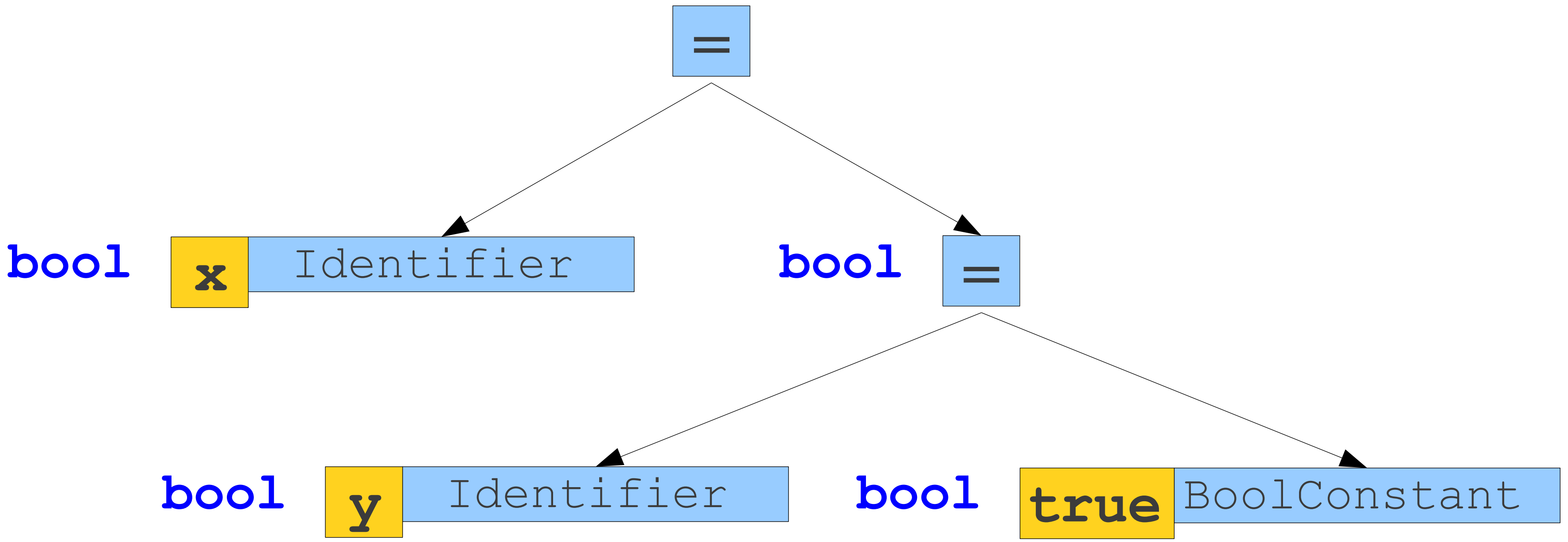
Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as **logical inference**.



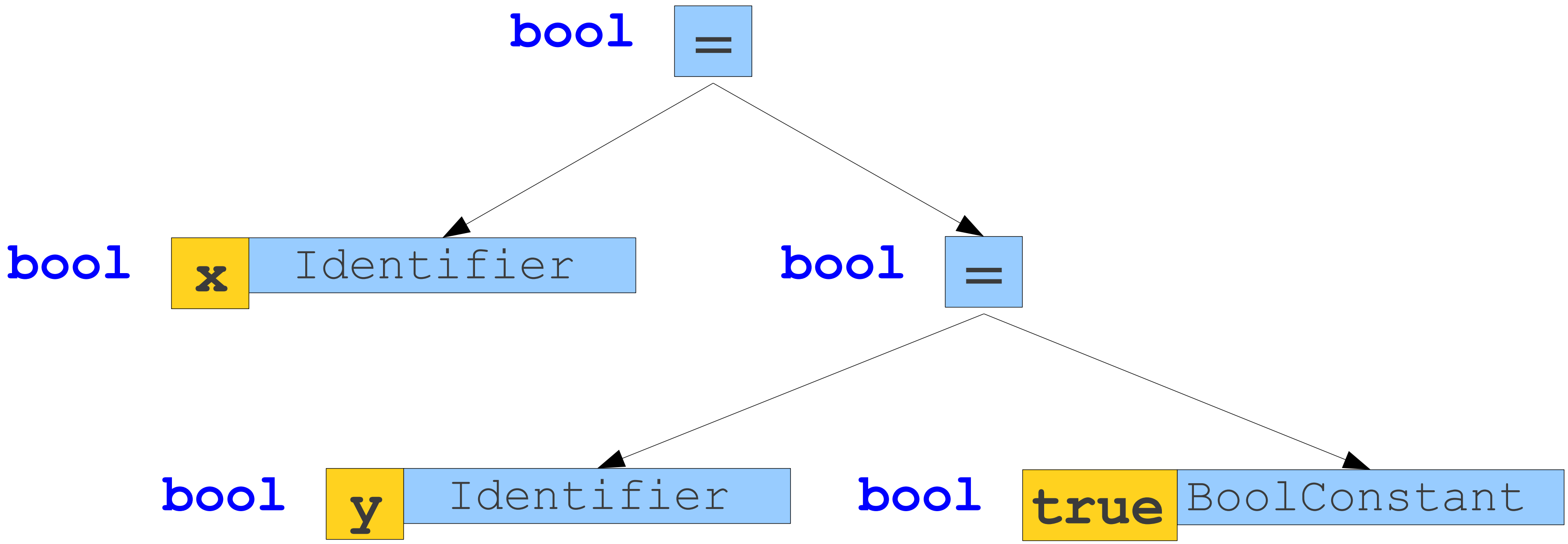
Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as **logical inference**.



Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as **logical inference**.



Type Checking as Proofs

- We can think of semantic analysis as proving claims about the types of expressions.
- We begin with a set of **axioms**, then apply our **inference rules** to determine the types of expressions.
- Many type systems can be thought of as proof systems.

Sample Inference Rules

- "If **x** is an identifier that refers to an object of type **t**, the expression **x** has type **t**."
- "If **e** is an integer constant, **e** has type **`int`**."
- "If the operands $\mathbf{e_1}$ and $\mathbf{e_2}$ of $\mathbf{e}_1 + \mathbf{e_2}$ are known to have types **`int`** and **`int`**, then $\mathbf{e}_1 + \mathbf{e_2}$ has type **`int`**."

Formalizing our Notation

- We will encode our axioms and inference rules using this syntax:

$$\frac{\text{Preconditions}}{\text{Postconditions}}$$

- This is read "if *preconditions* are true, we can infer *postconditions*."

Examples of Formal Notation

$$\frac{A \rightarrow \mathtt{t}\omega \text{ is a production.}}{\mathtt{t} \in \text{FIRST}(A)}$$

$$\frac{A \rightarrow \varepsilon \text{ is a production.}}{\varepsilon \in \text{FIRST}(A)}$$

$$\frac{\begin{array}{c} A \rightarrow \omega \text{ is a production.} \\ \mathtt{t} \in \text{FIRST*}(\omega) \end{array}}{\mathtt{t} \in \text{FIRST}(A)}$$

$$\frac{\begin{array}{c} A \rightarrow \omega \text{ is a production.} \\ \varepsilon \in \text{FIRST*}(\omega) \end{array}}{\varepsilon \in \text{FIRST}(A)}$$

Formal Notation for Type Systems

- We write

$$\vdash \mathbf{e} : \mathbf{T}$$

  if the expression **e** has type **T**.
- The symbol $\vdash$ means "we can infer…"

Our Starting Axioms

Our Starting Axioms

$$\frac{}{\vdash \texttt{true} : \texttt{bool}} \qquad \frac{}{\vdash \texttt{false} : \texttt{bool}}$$

Some Simple Inference Rules

Some Simple Inference Rules

$$\frac{i \text{ is an integer constant}}{\vdash i : \texttt{int}}$$

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

$$\frac{d \text{ is a double constant}}{\vdash d : \texttt{double}}$$

More Complex Inference Rules

More Complex Inference Rules

$$\frac{\vdash e_1 : \mathbf{int} \qquad \vdash e_2 : \mathbf{int}}{\vdash e_1 + e_2 : \mathbf{int}}$$

$$\frac{\vdash e_1 : \mathbf{double} \qquad \vdash e_2 : \mathbf{double}}{\vdash e_1 + e_2 : \mathbf{double}}$$

More Complex Inference Rules

If we can show that e_1 and e_2 have type int…

$$\frac{\vdash e_1 : \texttt{int} \qquad \vdash e_2 : \texttt{int}}{\vdash e_1 + e_2 : \texttt{int}}$$

$$\frac{\vdash e_1 : \texttt{double} \qquad \vdash e_2 : \texttt{double}}{\vdash e_1 + e_2 : \texttt{double}}$$

More Complex Inference Rules

If we can show that e_1 and e_2 have type int…

$$\frac{\vdash e_1 : \texttt{int} \qquad \vdash e_2 : \texttt{int}}{\vdash e_1 + e_2 : \texttt{int}}$$

$$\frac{\vdash e_1 : \texttt{double} \qquad \vdash e_2 : \texttt{double}}{\vdash e_1 + e_2 : \texttt{double}}$$

… then we can show that $e_1 + e_2$ has type int as well

Even More Complex Inference Rules

Even More Complex Inference Rules

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

$$\frac{\vdash e_1 : T \qquad \vdash e_2 : T \qquad T \text{ is a primitive type}}{\vdash e_1 \texttt{ != } e_2 : \texttt{bool}}$$

Why Specify Types this Way?

- Gives a **rigorous definition of types** independent of any particular implementation.
  - No need to say "you should have the same type rules as my reference compiler."
- Gives **maximum flexibility in implementation**.
  - Can implement type-checking however you want, as long as you obey the rules.
- Allows **formal verification of program properties**.
  - Can do inductive proofs on the structure of the program.
- **This is what's used in the literature**.
  - Good practice if you want to study types.

A Problem

A Problem

$$\frac{x \text{ is an identifier.}}{\vdash x : \mathbf{??}}$$

How do we know the type of x if we don't know what it refers to?

An Incorrect Solution

x is an identifier.
x is in scope with type T.

$$\frac{x \text{ is an identifier.} \quad x \text{ is in scope with type T.}}{\vdash x : T}$$

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

```
int MyFunction(int x) {
      {
            double x;
      }

      if (x == 1.5) {
            /* … */
      }
}
```

Facts
$\vdash x$: **double**
$\vdash x$: **int**
$\vdash$ **1.5** : **double**
$\vdash x$ == **1.5** : **bool**

Strengthening our Inference Rules

- The facts we're proving have no *context*.
- We need to strengthen our inference rules to remember under what circumstances the results are valid.

Adding Scope

- We write

$$\mathbf{S} \vdash \mathbf{e} : \mathbf{T}$$

  if, in scope **S**, expression **e** has type **T**.

- Types are now proven relative to the scope they are in.

What is the Scope?

- Recall scope contains variables’ definition.

S = {(i, int), (j, float)}

Old Rules Revisited

$$\frac{}{S \vdash \texttt{true} : \texttt{bool}}$$

$$\frac{}{S \vdash \texttt{false} : \texttt{bool}}$$

$$\frac{i \text{ is an integer constant}}{S \vdash i : \texttt{int}}$$

$$\frac{s \text{ is a string constant}}{S \vdash s : \texttt{string}}$$

$$\frac{d \text{ is a double constant}}{S \vdash d : \texttt{double}}$$

$$\frac{S \vdash e_1 : \texttt{double} \quad S \vdash e_2 : \texttt{double}}{S \vdash e_1 + e_2 : \texttt{double}}$$

$$\frac{S \vdash e_1 : \texttt{int} \quad S \vdash e_2 : \texttt{int}}{S \vdash e_1 + e_2 : \texttt{int}}$$

A Correct Rule

$$\frac{x \text{ is an identifier.} \quad x \text{ is a variable in scope S with type T.}}{S \vdash x : T}$$

A Correct Rule

$$\frac{x \text{ is an identifier.} \quad x \textbf{ is a variable in scope S} \text{ with type T.}}{S \vdash x : T}$$

Rules for Function Calls

f is an identifier.

$$\frac{}{S \vdash f(e_1, \ldots, e_n) \quad : \; ??}$$

Rules for Function Calls

f is an identifier.
f is a non-member function in scope S.

$$\frac{}{S \vdash f(e_1, \ldots, e_n) \quad : \; ??}$$

Rules for Function Calls

f is an identifier.
f is a non-member function in scope S.
f has type $(T_1, \ldots, T_n) \to U$

$$\frac{}{S \vdash f(e_1, \ldots, e_n) \quad : \ ??}$$

Rules for Function Calls

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ f \text{ is a non-member function in scope S.} \\ f \text{ has type } (T_1, \ldots, T_n) \to U \\ S \vdash e_i : T_i \quad \text{for } 1 \le i \le n \end{array}}{S \vdash f(e_1, \ldots, e_n) \quad : \;\; ??}$$

Rules for Function Calls

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ f \text{ is a non-member function in scope S.} \\ f \text{ has type } (T_1, \ldots, T_n) \rightarrow U \\ S \vdash e_i : T_i \quad \text{for } 1 \leq i \leq n \end{array}}{S \vdash f(e_1, \ldots, e_n) \quad :}$$

Rules for Function Calls

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ f \text{ is a non-member function in scope S.} \\ f \text{ has type } (T_1, \ldots, T_n) \to U \\ S \vdash e_i : T_i \quad \text{for } 1 \le i \le n \end{array}}{S \vdash f(e_1, \ldots, e_n) \quad : \quad U}$$

Rules for Arrays

$$\frac{S \vdash e_1 : T\texttt{[]} \qquad S \vdash e_2 : \texttt{int}}{S \vdash e_1[e_2] : T}$$

Rule for Assignment

$$\frac{S \vdash e_1 : T \qquad S \vdash e_2 : T}{S \vdash e_1 = e_2 : T}$$

Rule for Assignment

$$\frac{\begin{array}{c} S \vdash e_1 : T \\ S \vdash e_2 : T \end{array}}{S \vdash e_1 = e_2 : T}$$

If **`Derived`** extends **`Base`**, will this rule work for this code?

```
Base    myBase;
Derived myDerived;

myBase = myDerived;
```

Rule for Comparison

$$\frac{S \vdash e_1 : \text{int} \qquad S \vdash e_2 : \text{int}}{S \vdash e_1 < e_2 : \text{bool}}$$

Example

- Assume we know that i and j are defined integers within scope S.
- What is the type of $i + 1 < j$?

Example

- Assume we know that *i* and *j* are defined integers within scope S.
- What is the type of $i + 1 < j$?

$$\dfrac{\dfrac{\dfrac{(i,\ \text{int}) \in S}{S \vdash i : \text{int}} \qquad \dfrac{1 \text{ is an integer}}{1\!: \text{int}}}{S \vdash i + 1\!: \text{int}} \qquad \dfrac{(j,\ \text{int}) \in S}{S \vdash j\!: \text{int}}}{S \vdash i + 1 < j\!: \text{bool}}$$

More Rules-simplified

$$\frac{S \vdash x: T \quad S \vdash e: T}{S \vdash x = e; : \text{void}}$$

$$\frac{}{S \vdash \{\,\} : \text{void}}$$

$$\frac{S \vdash \text{e: bool} \qquad S \vdash \text{s: void}}{S \vdash \text{while } (\text{e}) \text{ s} : \text{void}}$$

$$\frac{S \vdash \text{s: void}}{S \vdash \{\text{s}\}: \text{void}}$$

$$\frac{S \vdash \text{e: bool} \quad S \vdash \text{s}_1\text{: void} \quad S \vdash \text{s}_2\text{: void}}{S \vdash \text{if } (\text{ e }) \text{ s}_1 \text{ else s}_2 : \text{void}}$$

$$\frac{S \vdash \text{s}_1\text{: void} \quad S \vdash \text{s}_2\text{: void}}{S \vdash \text{s}_1 \text{ s}_2 : \text{void}}$$

Example

- Assume we know that *i* and *j* are defined integers within scope S.

Example

- Assume we know that i and j are defined integers within scope S.

$$\frac{}{S \vdash \{j = 0; \mathbf{if}(i == 0) j = 1; \}:\ \text{void}}$$

$$\frac{S \vdash j = 0; \mathbf{if}(i == 0) j = 1; : \ \mathbf{void}}{S \vdash \{j = 0; \mathbf{if}(i == 0) j = 1; \}\!: \ \mathbf{void}} \ \text{Block-Rule}$$

$$\cfrac{\cfrac{S \vdash j = 0\colon \ \mathbf{void} \qquad S \vdash \mathbf{if}(i == 0) j = 1; : \ \mathbf{void}}{S \vdash j = 0; \mathbf{if}(i == 0) j = 1; : \ \mathbf{void}}\ \text{Composition-Rule}}{S \vdash \{j = 0; \mathbf{if}(i == 0) j = 1; \}\colon \ \mathbf{void}}\ \text{Block-Rule}$$

$$\dfrac{\dfrac{\dfrac{S \vdash 0\colon\ \mathbf{int} \qquad S \vdash j\colon\ \mathbf{int}}{S \vdash j = 0\colon\ \mathbf{void}}\ \text{Assignment-Rule} \qquad S \vdash \mathbf{if}(i == 0)j = 1;\ :\ \mathbf{void}}{S \vdash j = 0; \mathbf{if}(i == 0)j = 1;\ :\ \mathbf{void}}\ \text{Composition-Rule}}{S \vdash \{j = 0; \mathbf{if}(i == 0)j = 1;\}\colon\ \mathbf{void}}\ \text{Block-Rule}$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{0 \text{ is an integer} \in S}{S \vdash 0\colon\ \mathbf{int}}\ \text{ASS} \qquad \dfrac{(i,\ \text{int}) \in S}{S \vdash j\colon\ \mathbf{int}}\ \text{ASS}}{S \vdash j = 0\colon\ \mathbf{void}}\ \text{Assignment-Rule} \qquad S \vdash \mathbf{if}(i == 0)j = 1;\ :\ \mathbf{void}}{S \vdash j = 0; \mathbf{if}(i == 0)j = 1;\ :\ \mathbf{void}}\ \text{Composition-Rule}}{S \vdash \{j = 0; \mathbf{if}(i == 0)j = 1;\}\colon\ \mathbf{void}}\ \text{Block-Rule}
$$

$$\dfrac{\dfrac{\dfrac{\dfrac{(i,\ \text{int}) \in S}{S \vdash j\colon\ \mathbf{int}}\ \text{ASS}}{\mathbf{void}}\ \text{Assignment-Rule} \qquad \dfrac{S \vdash i == 0\colon\ \mathbf{bool} \qquad S \vdash j = 1\colon\ \mathbf{void}}{S \vdash \mathbf{if}(i == 0)j = 1;\ :\ \mathbf{void}}\ \text{IF-Rule}}{S \vdash j = 0; \mathbf{if}(i == 0)j = 1;\ :\ \mathbf{void}}\ \text{Composition-Rule}}{S \vdash \{j = 0; \mathbf{if}(i == 0)j = 1;\}\colon\ \mathbf{void}}\ \text{Block-Rule}$$

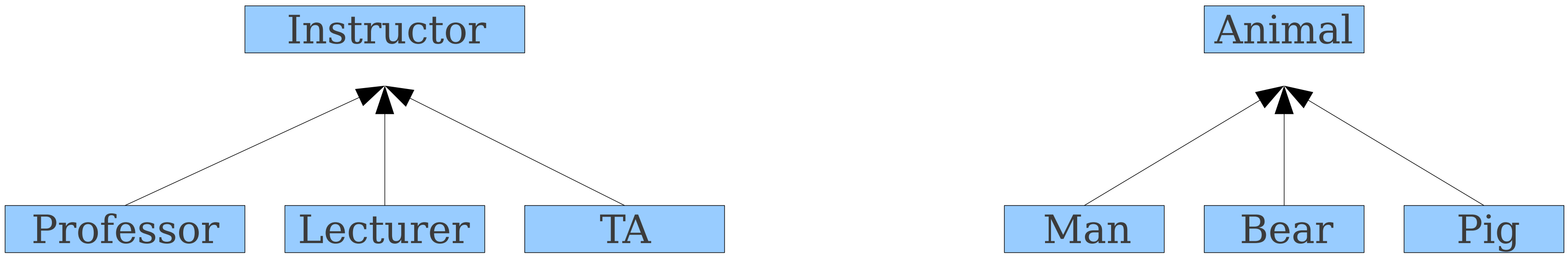
$$
\dfrac{
  \dfrac{
    \text{nent-Rule}
    \qquad
    \dfrac{
      \dfrac{
        \dfrac{0 \text{ is an integer} \in S}{S \vdash 0\colon\ \mathbf{int}}\,\text{ASS}
        \qquad
        \dfrac{(i,\ \text{int}) \in S}{S \vdash i\colon\ \mathbf{int}}\,\text{ASS}
      }{S \vdash i == 0\colon\ \mathbf{bool}}\,\text{Comparison-Rule}
      \qquad
      S \vdash j = 1\colon\ \mathbf{void}
    }{S \vdash \mathbf{if}(i == 0) j = 1;\ :\ \mathbf{void}}\,\text{IF-Rule}
  }{S \vdash j = 0; \mathbf{if}(i == 0) j = 1;\ :\ \mathbf{void}}\,\text{Composition-Rule}
}{S \vdash \{j = 0; \mathbf{if}(i == 0) j = 1; \}\colon\ \mathbf{void}}\,\text{Block-Rule}
$$

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{0 \text{ is an integer} \in S}{S \vdash 0\colon \mathbf{int}}\,\text{ASS}
      \qquad
      \dfrac{(j,\, \text{int}) \in S}{S \vdash j\colon \mathbf{int}}\,\text{ASS}
    }{S \vdash j = 0\colon \mathbf{void}}\,\text{Assignment-Rule}
    \qquad
    \dfrac{
      \dfrac{
        \dfrac{0 \text{ is an integer} \in S}{S \vdash 0\colon \mathbf{int}}\,\text{ASS}
        \qquad
        \dfrac{(i,\, \text{int}) \in S}{S \vdash i\colon \mathbf{int}}\,\text{ASS}
      }{S \vdash i == 0\colon \mathbf{bool}}\,\text{Comparison-Rule}
      \qquad
      \dfrac{
        \dfrac{1 \text{ is an integer} \in S}{S \vdash 1\colon \mathbf{int}}\,\text{ASS}
        \qquad
        \dfrac{(j,\, \text{int}) \in S}{S \vdash j\colon \mathbf{int}}\,\text{ASS}
      }{S \vdash j = 1\colon \mathbf{void}}\,\text{Assignment-Rule}
    }{S \vdash \mathbf{if}(i == 0)j = 1;\, :\ \mathbf{void}}\,\text{IF-Rule}
  }{S \vdash j = 0; \mathbf{if}(i == 0)j = 1;\, :\ \mathbf{void}}\,\text{Composition-Rule}
}{S \vdash \{j = 0; \mathbf{if}(i == 0)j = 1;\}\colon \mathbf{void}}\,\text{Block-Rule}
$$

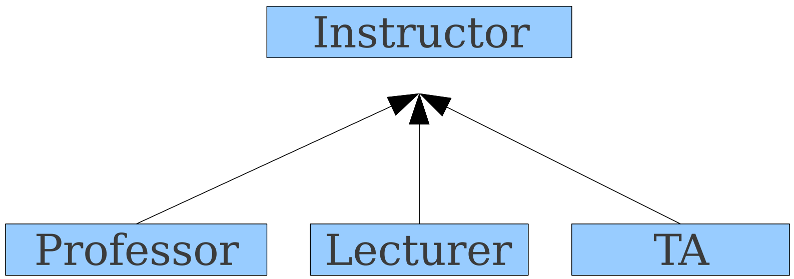
Typing with Classes

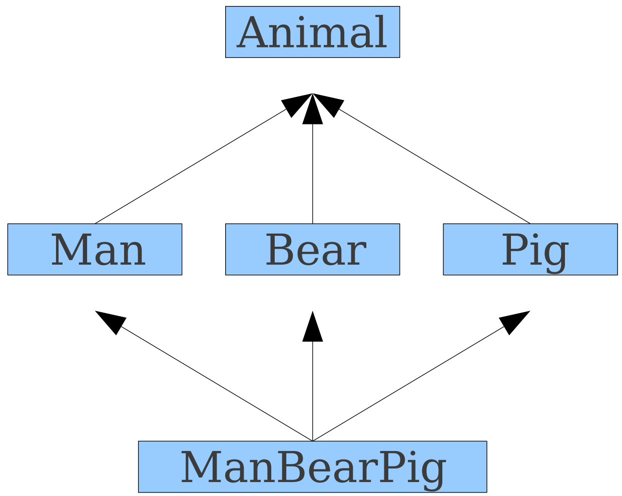
- How do we factor inheritance into our inference rules?
- We need to consider the shape of class hierarchies.

Single Inheritance



Multiple Inheritance





Properties of Inheritance Structures

- Any type is convertible to itself. (**reflexivity**)
- If A is convertible to B and B is convertible to C, then A is convertible to C. (**transitivity**)
- If A is convertible to B and B is convertible to A, then A and B are the same type. (**antisymmetry**)
- This defines a **partial order** over types.

Types and Partial Orders

- We say that A $\leq$ B if A is convertible to B.
- We have that
  - A $\leq$ A
  - A $\leq$ B and B $\leq$ C implies A $\leq$ C
  - A $\leq$ B and B $\leq$ A implies A = B

Updated Rule for Assignment

$$\frac{}{S \vdash e_1 = e_2 : ??}$$

Updated Rule for Assignment

$$\frac{S \vdash e_1 : T_1 \qquad S \vdash e_2 : T_2}{S \vdash e_1 = e_2 : ??}$$

Updated Rule for Assignment

$$\frac{S \vdash e_1 : T_1 \qquad S \vdash e_2 : T_2 \qquad T_2 \leq T_1}{S \vdash e_1 = e_2 : ??}$$

T2 inherits T1

Updated Rule for Assignment

$$\frac{\begin{array}{c} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_2 \leq T_1 \end{array}}{S \vdash e_1 = e_2 : T_1}$$

T2 inherits T1

Updated Rule for Comparisons

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

Updated Rule for Comparisons

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

$$\frac{S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad T_1 \text{ and } T_2 \text{ are of class type.} \quad T_1 \leq T_2 \text{ or } T_2 \leq T_1}{S \vdash e_1 == e_2 : \textbf{bool}}$$

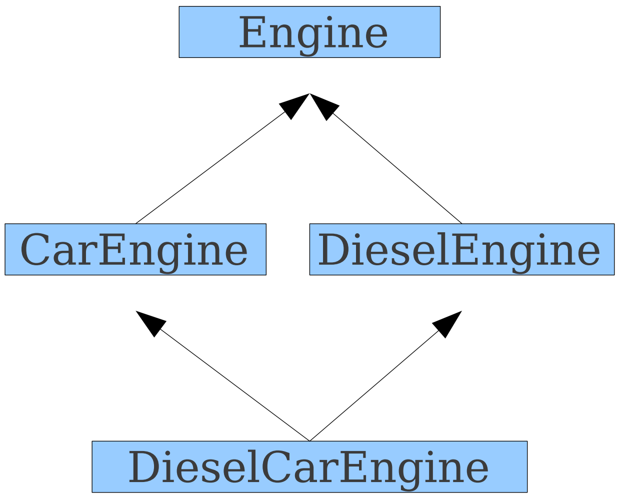
Updated Rule for Comparisons

Can we unify these rules?

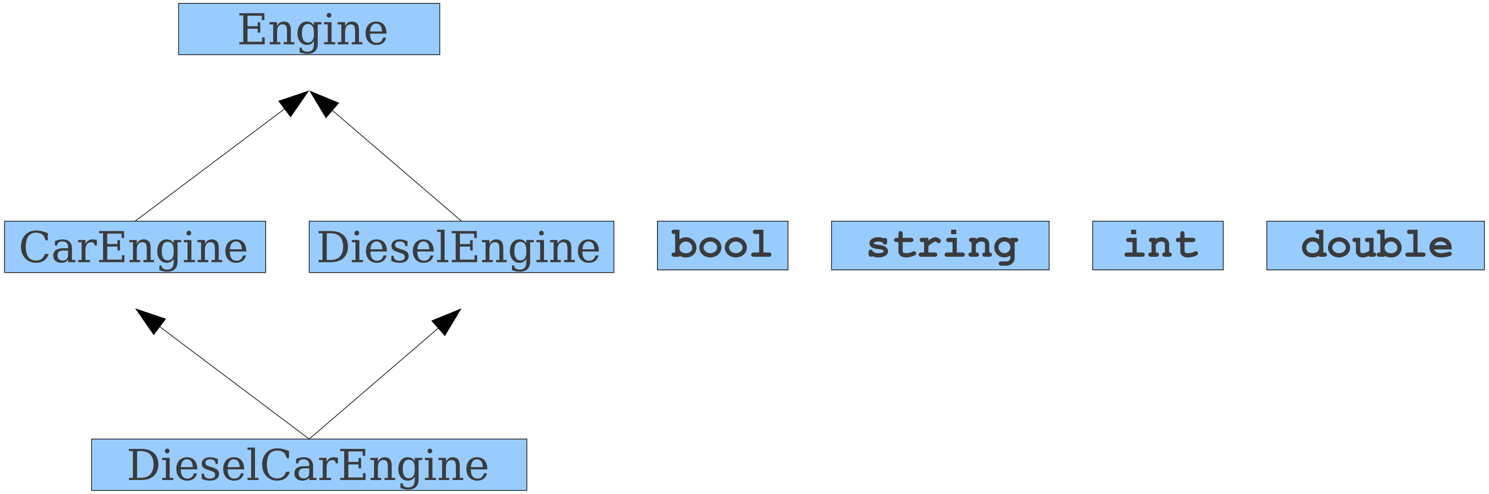
$$\frac{S \vdash e_1 : T \qquad S \vdash e_2 : T \qquad T \text{ is a primitive type}}{S \vdash e_1 == e_2 : \texttt{bool}}$$

$$\frac{S \vdash e_1 : T_1 \qquad S \vdash e_2 : T_2 \qquad T_1 \text{ and } T_2 \text{ are of class type.} \qquad T_1 \leq T_2 \text{ or } T_2 \leq T_1}{S \vdash e_1 == e_2 : \texttt{bool}}$$

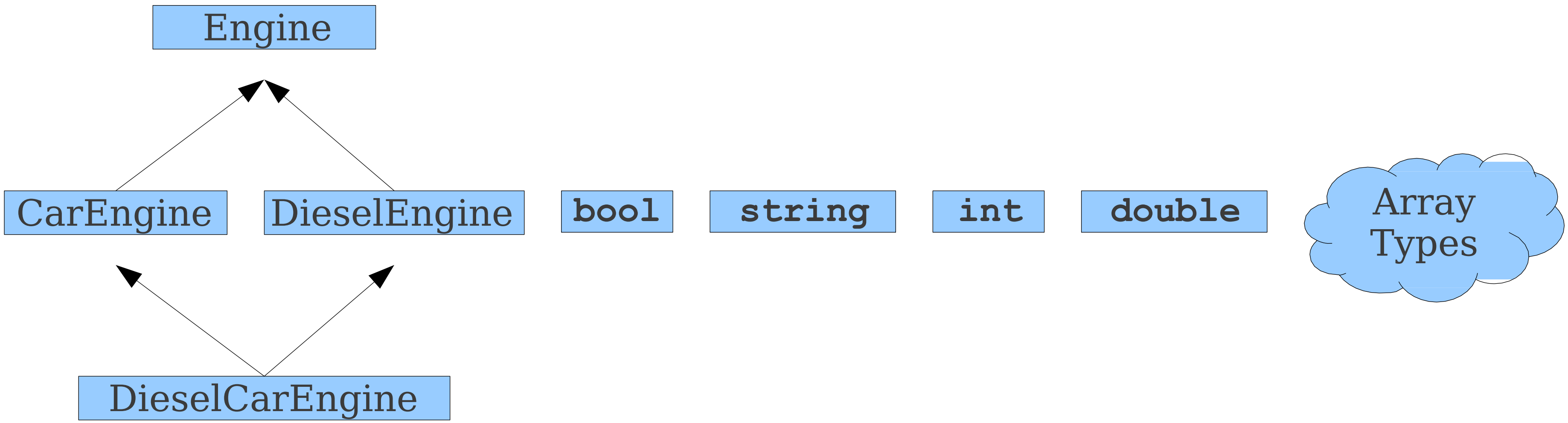
The Shape of Types



The Shape of Types



The Shape of Types



Extending Convertibility

- If A is a primitive or array type, A is only convertible to itself.
- More formally, if A and B are types and A is a primitive or array type:
  - $A \leq B$ implies $A = B$
  - $B \leq A$ implies $A = B$

Updated Rule for Comparisons

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

$$\frac{S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad T_1 \text{ and } T_2 \text{ are of class type.} \quad T_1 \leq T_2 \text{ or } T_2 \leq T_1}{S \vdash e_1 \texttt{ == } e_2 : \textbf{bool}}$$

Updated Rule for Comparisons

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

$$\frac{S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad T_1 \text{ and } T_2 \text{ are of class type.} \quad T_1 \leq T_2 \text{ or } T_2 \leq T_1}{S \vdash e_1 \texttt{==} e_2 : \textbf{bool}}$$

$$\frac{S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad T_1 \leq T_2 \text{ or } T_2 \leq T_1}{S \vdash e_1 \texttt{==} e_2 : \textbf{bool}}$$

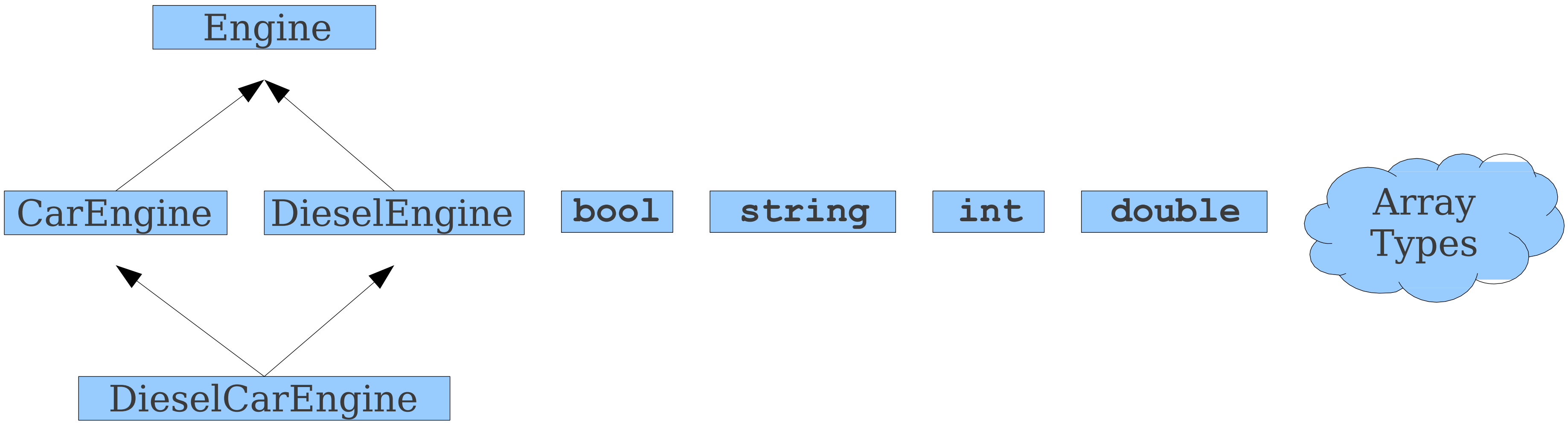
Updated Rule for Function Calls

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ f \text{ is a non-member function in scope S.} \\ f \text{ has type } (T_1, \ldots, T_n) \to U \\ S \vdash e_i : R_i \quad \text{for } 1 \leq i \leq n \\ R_i \leq T_i \quad \text{for } 1 \leq i \leq n \end{array}}{S \vdash f(e_1, \ldots, e_n) \quad : U}$$

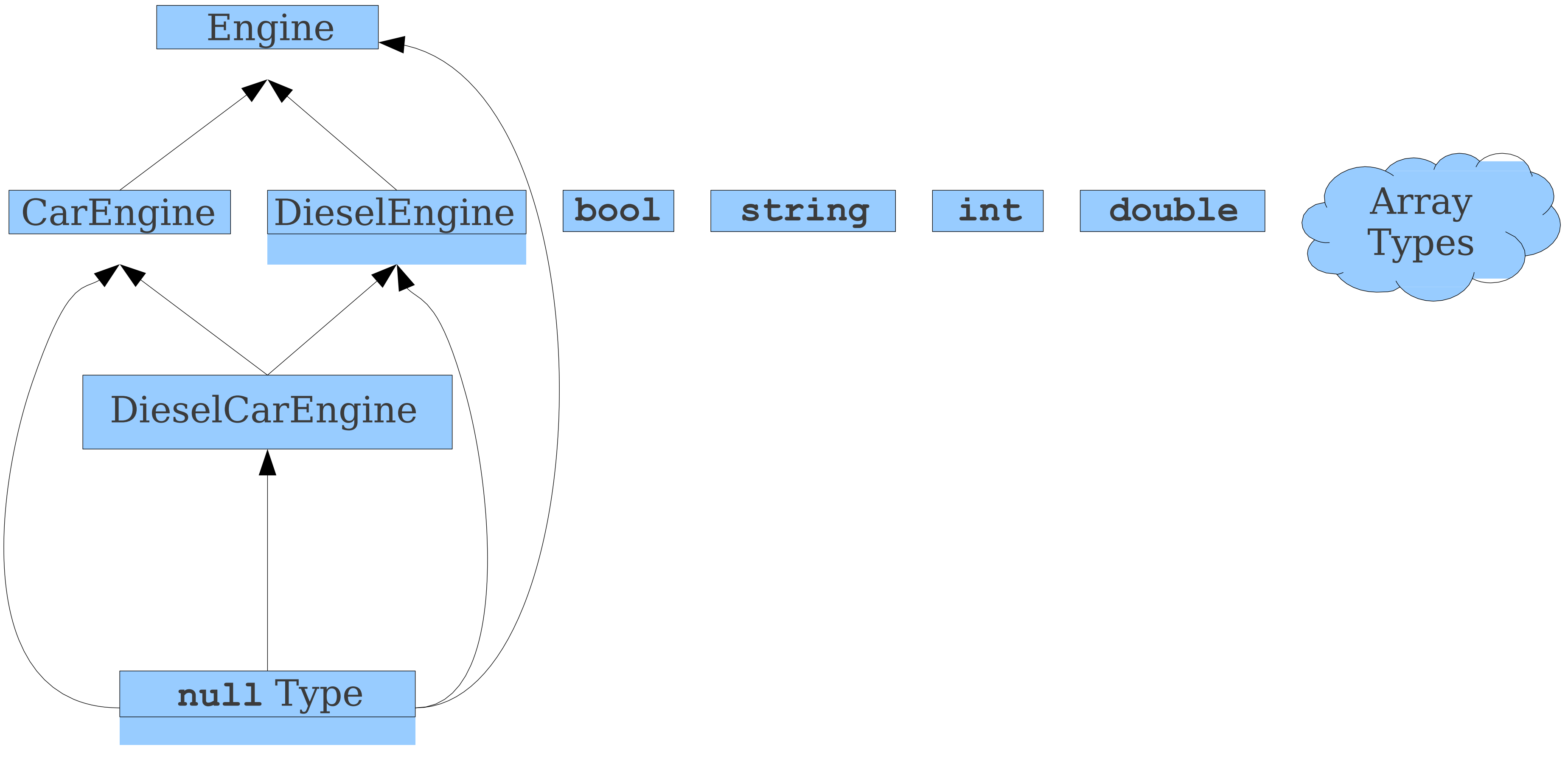
A Tricky Case

$$\frac{}{S \vdash \texttt{null} : ??}$$

Back to the Drawing Board



Back to the Drawing Board



Handling `null`

- Define a new type corresponding to the type of the literal **`null`**; call it "**`null` type**."
- Define **`null`** type ≤ A for any class type A.
- The **`null`** type is (typically) not accessible to programmers; it's only used internally.
- Many programming languages have types like these.

A Tricky Case

$$\frac{}{S \vdash \texttt{null} : ??}$$

A Tricky Case

$$\frac{}{S \vdash \texttt{null} : \texttt{null}\ \text{type}}$$

A Tricky Case

$$\frac{}{S \vdash \mathbf{\texttt{null}} : \mathbf{\texttt{null}} \text{ type}}$$

Object-Oriented Considerations

$$\frac{\text{S is in scope of class T.}}{S \vdash \texttt{this} : T}$$

$$\frac{\text{T is a class type.}}{S \vdash \texttt{new}\ \ T : T}$$

$$\frac{S \vdash e : \texttt{int}}{S \vdash \texttt{NewArray(}e, T\texttt{)} : T\texttt{[]}}$$

What's Left?

- We're missing a few language constructs:
  - Member functions.
  - Field accesses.
  - Miscellaneous operators.
- Good practice to fill these in on your own.

Typing is Nuanced

- The **ternary conditional operator ? :** evaluates an expression, then produces one of two values.
- Works for primitive types:
  - `int x = random()>1? 137 : 42;`
- Works with inheritance:
  - `Base b = isB? new Base : new Derived;`
- What might the typing rules look like?

A Proposed Rule

$$\frac{}{S \vdash \mathit{cond}\ \mathbf{?}\ \mathrm{e}_1\ \mathbf{:}\ \mathrm{e}_2 : ??}$$

A Proposed Rule

$$\frac{S \vdash \mathit{cond} : \texttt{bool}}{S \vdash \mathit{cond}\ \texttt{?}\ e_1\ \texttt{:}\ e_2 : ??}$$

A Proposed Rule

$$\frac{\begin{array}{c} S \vdash \mathit{cond} : \texttt{bool} \\ S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \end{array}}{S \vdash \mathit{cond}\ \texttt{?}\ e_1\ \texttt{:}\ e_2 : ??}$$

A Proposed Rule

$$\frac{\begin{array}{c} S \vdash \mathit{cond} : \mathtt{bool} \\ S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq T_2 \text{ or } T_2 \leq T_1 \end{array}}{S \vdash \mathit{cond}\ \mathtt{?}\ e_1\ \mathtt{:}\ e_2 : \text{??}}$$

A Proposed Rule

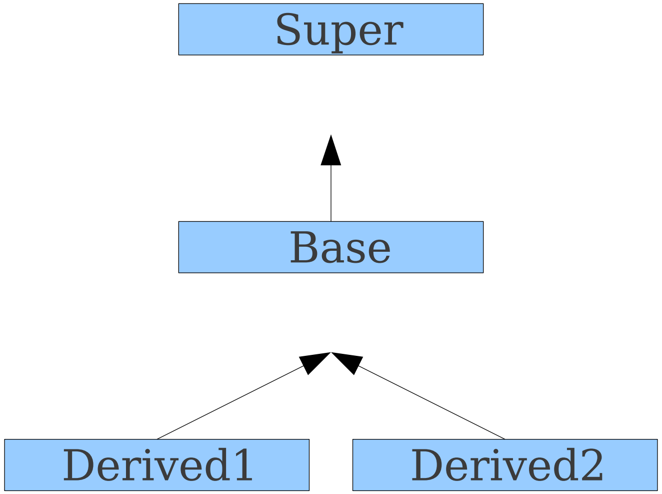
$$\frac{\begin{array}{c} S \vdash \mathit{cond} : \texttt{bool} \\ S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq T_2 \text{ or } T_2 \leq T_1 \end{array}}{S \vdash \mathit{cond}\ \texttt{?}\ e_1\ \texttt{:}\ e_2 : \max(T_1, T_2)}$$

A Proposed Rule

$$\frac{\begin{array}{c} S \vdash \textit{cond} : \texttt{bool} \\ S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq T_2 \text{ or } T_2 \leq T_1 \end{array}}{S \vdash \textit{cond} \; \textbf{?} \; e_1 \; \textbf{:} \; e_2 : \mathbf{max(T_1, T_2)}}$$

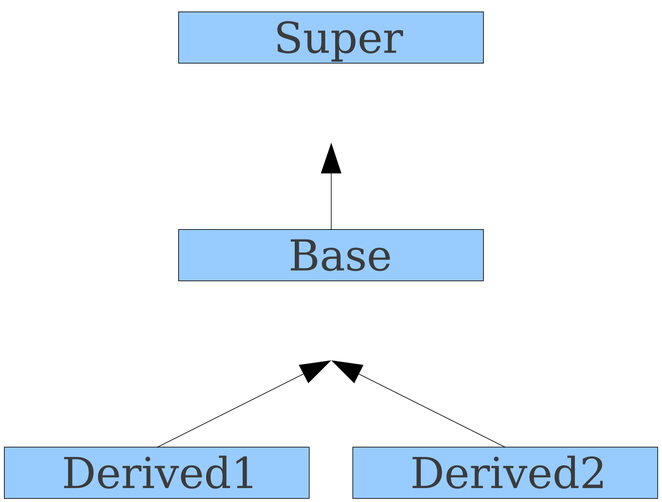
A Proposed Rule

$$\frac{S \vdash \mathit{cond} : \mathbf{bool} \quad S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad T_1 \le T_2 \text{ or } T_2 \le T_1}{S \vdash \mathit{cond} \;?\; e_1 : e_2 : \max(T_1, T_2)}$$



A Proposed Rule

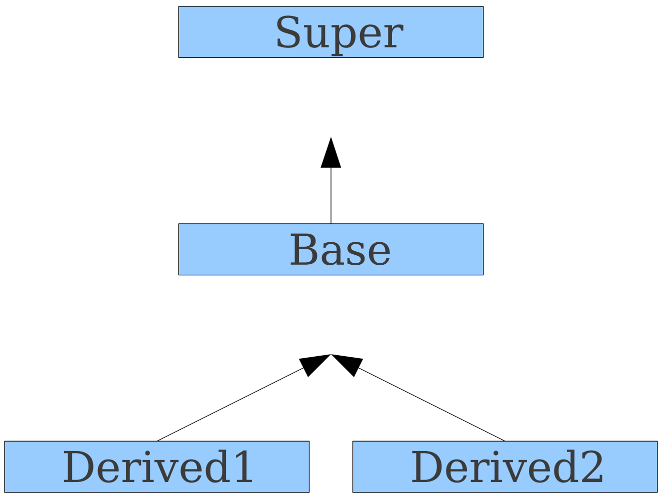
$$\frac{\begin{array}{c} S \vdash \mathit{cond} : \texttt{bool} \\ S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq T_2 \text{ or } T_2 \leq T_1 \end{array}}{S \vdash \mathit{cond} \;\textbf{?}\; e_1 \;\textbf{:}\; e_2 : \mathbf{max(T_1, T_2)}}$$



Is this **really**

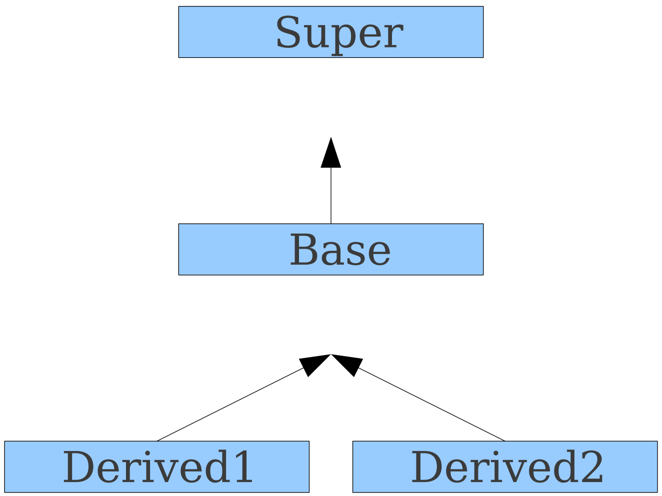
what we want?

A Small Problem



$$\frac{S \vdash \mathit{cond} : \mathtt{bool} \quad S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad T_1 \leq T_2 \text{ or } T_2 \leq T_1}{S \vdash \mathit{cond} \;?\; e_1 : e_2 : \max(T_1, T_2)}$$

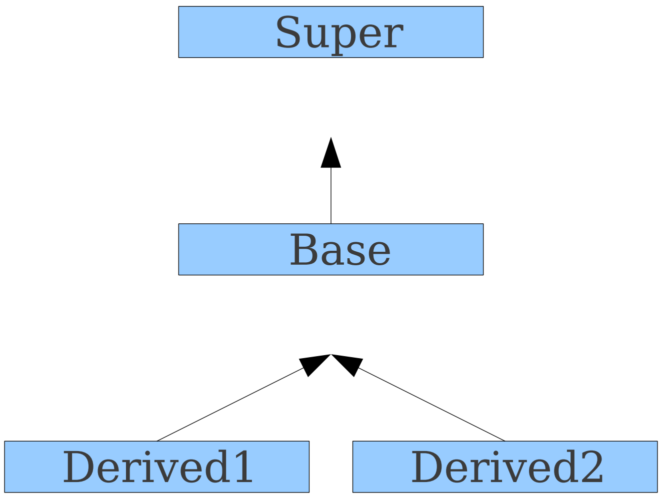
A Small Problem



$$\frac{S \vdash \mathit{cond} : \texttt{bool} \quad S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad T_1 \leq T_2 \text{ or } T_2 \leq T_1}{S \vdash \mathit{cond}\ ?\ e_1 : e_2 : \max(T_1, T_2)}$$

```
Base = isB?
          new Derived1 : new Derived2;
```

A Small Problem



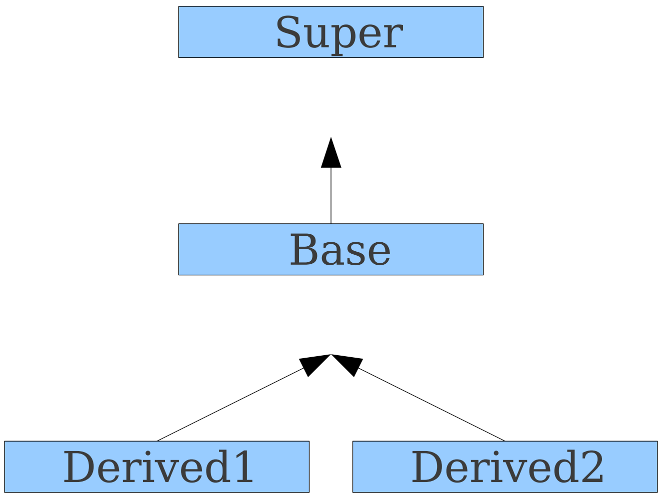
$$\frac{S \vdash \mathit{cond} : \texttt{bool} \quad S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad \color{red}{T_1 \leq T_2 \text{ or } T_2 \leq T_1}}{S \vdash \mathit{cond}\ ?\ e_1 : e_2 : \max(T_1, T_2)}$$

```
Base = isB?
             new Derived1 : new Derived2;
```

Least Upper Bounds

- An **upper bound** of two types A and B is a type C such that A ≤ C and B ≤ C.
- The **least upper bound** of two types A and B is a type C such that:
  - C is an upper bound of A and B.
  - If C' is an upper bound of A and B, then C ≤ C'.
- When the least upper bound of A and B exists, we denote it A ∨ B.
  - (When might it not exist?)

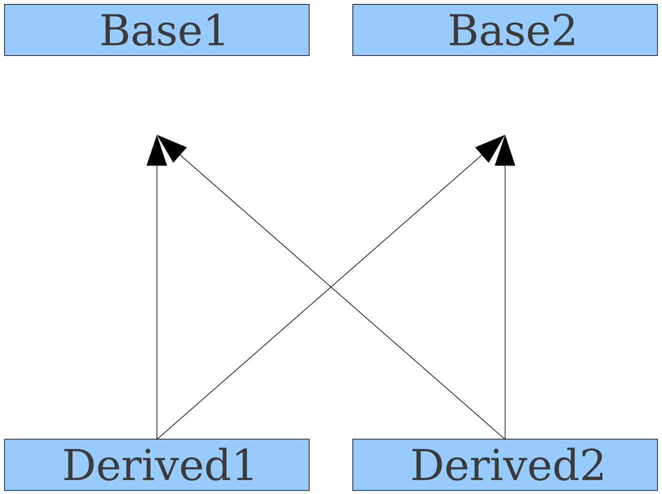
A Better Rule



$$\frac{S \vdash \mathit{cond} : \texttt{bool} \quad S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad T = T_1 \vee T_2}{S \vdash \mathit{cond}\ ?\ e_1 : e_2 : T}$$

```
Base = isB?
          new Derived1 : new Derived2;
```

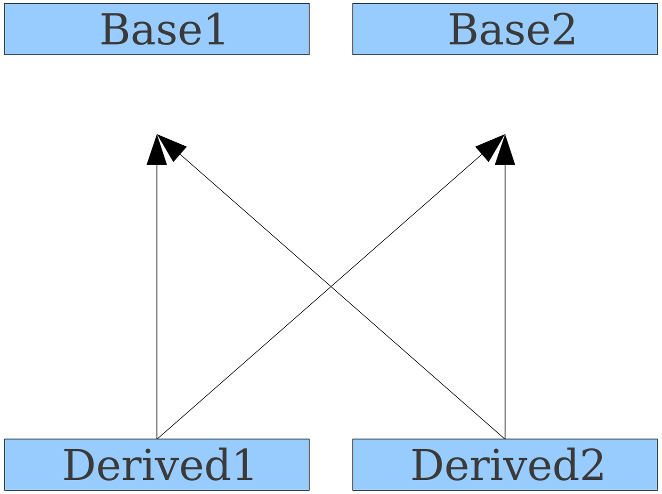
… that still has problems



$$\frac{S \vdash \mathit{cond} : \texttt{bool} \quad S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad T = T_1 \vee T_2}{S \vdash \mathit{cond}\ ?\ e_1 : e_2 : T}$$

```
Base = isB?
            new Derived1 : new Derived2;
```

… that still has problems



$$\frac{S \vdash \mathit{cond} : \texttt{bool} \quad S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad \mathbf{T = T_1 \vee T_2}}{S \vdash \mathit{cond}\ ?\ e_1 : e_2 : T}$$

```
Base = isB?
           new Derived1 : new Derived2;
```

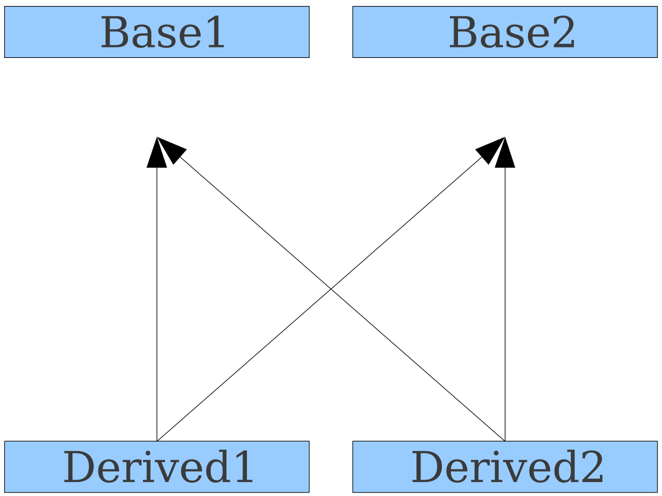
Multiple Inheritance is Messy

- Type hierarchy is no longer a tree.
- Two classes might not have a least upper bound.
- Occurs C++ because of multiple inheritance and in Java due to interfaces.
- Not a problem in Decaf; there is no ternary conditional operator.
- How to fix?

Minimal Upper Bounds

- An **upper bound** of two types A and B is a type C such that A ≤ C and B ≤ C.
- A **minimal upper bound** of two types A and B is a type C such that:
  - C is an upper bound of A and B.
  - If C' is an upper bound of C, then it is not true that C' < C.
- Minimal upper bounds are not necessarily unique.
- A least upper bound must be a minimal upper bound, but not the other way around.

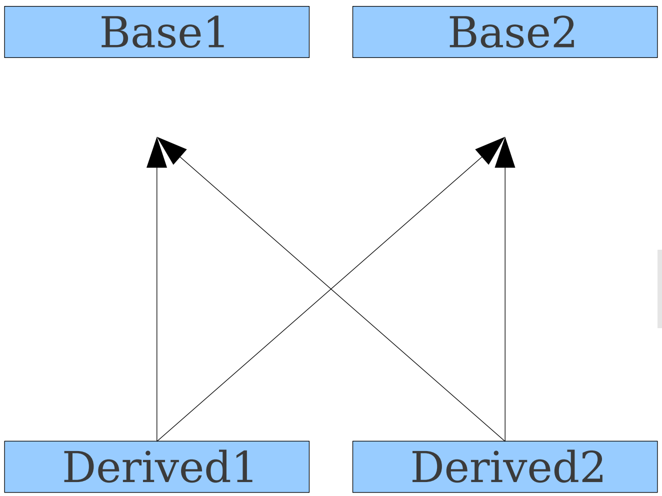
A Correct Rule



$$\dfrac{\begin{array}{c} S \vdash \mathit{cond} : \texttt{bool} \\ S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T \text{ is a minimal upper bound of } T_1 \text{ and } T_2 \end{array}}{S \vdash \mathit{cond} \;?\; e_1 : e_2 : T}$$

```
Base = isB?
            new Derived1 : new Derived2;
```

A Correct Rule



$$\frac{\begin{array}{c} S \vdash \mathit{cond} : \texttt{bool} \\ S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T \text{ is a minimal upper bound of } T_1 \text{ and } T_2 \end{array}}{S \vdash \mathit{cond}\ ?\ e_1 : e_2 : T}$$

Can prove both that expression has type **`Base1`** and that expression has type **`Base2`**.

```
Base = isB?
            new Derived1 : new Derived2;
```

So What?

- **Type-checking can be tricky**.
- Strongly influenced by the choice of operators in the language.
- Strongly influenced by the legal type conversions in a language.
- In C++, the previous example doesn't compile.
- In Java, the previous example does compile, but the language spec is ***enormously*** complicated.
    - See §15.12.2.7 of the Java Language Specification.

Next Time

- **Checking Statement Validity**
  - When are statements legal?
  - When are they illegal?
- **Practical Concerns**
  - How does function overloading work?
  - How do functions interact with inheritance?