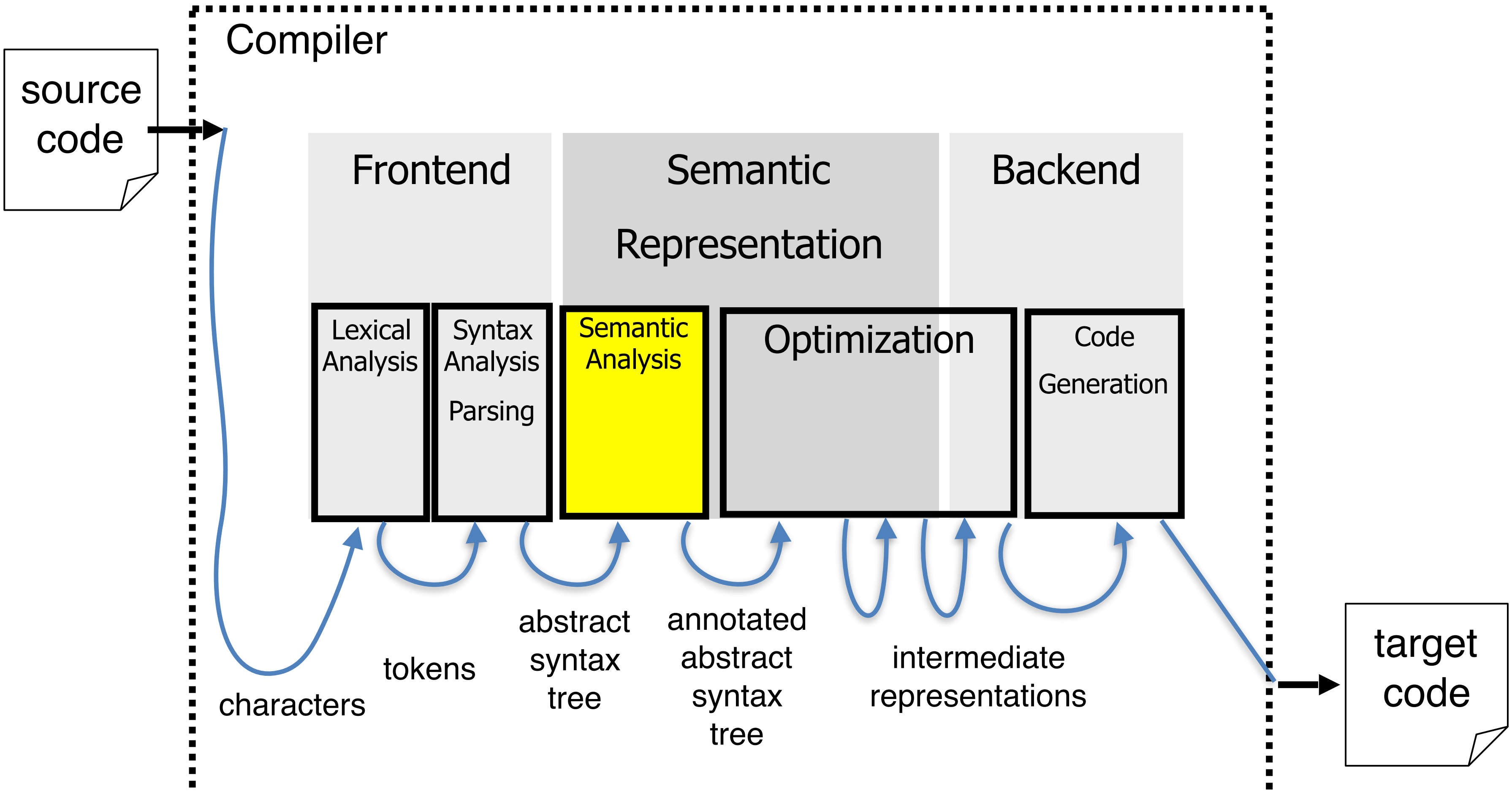
Type Checking

Greta Yorsh



Compiler
source code
Frontend
Semantic Representation
Backend
Lexical Analysis
Syntax Analysis
Parsing
Semantic Analysis
Optimization
Code Generation
characters
tokens
abstract syntax tree
annotated abstract syntax tree
intermediate representations
target code

Recap: semantic analysis

- Scope rules
- Symbol tables
- Inheritance graph

Types

- What is a type?
  - simplest answer: a set of values
  - examples: integers, real numbers, booleans, …

- Why do we care?
  - safety: guarantee that certain errors cannot occur at runtime
  - abstraction: hide implementation details

Type declarations

- Explicit type declarations

```
TYPE Int_Array = ARRAY [Integer 1..42] OF Integer;
```

- Anonymous types

```
Var a : ARRAY [Integer 1..42] OF Real;
```

Anonymous types

```
Var a : ARRAY [Integer 1..42] OF Real;
```

```
TYPE #type01_in_line_73 = ARRAY [Integer 1..42] OF Real;
Var a : #type01_in_line_73;
```

Forward references

```
TYPE Ptr_List_Entry = POINTER TO List_Entry;
TYPE List_Entry =
    RECORD
        Element : Integer;
        Next : Ptr_List_Entry;
    END RECORD;
```

- Type added to symbol table as **forward reference**
- **Update symbol table when the type declaration is met**
- At the end of scope, check that all forward refs have been resolved
- Check for **cycles**

Type equivalence: **name** equivalence

```
Type t1 = ARRAY[Integer] OF Integer;
Type t2 = ARRAY[Integer] OF Integer;
```

t1 not (name) equivalent to t2

```
Type t3 = ARRAY[Integer] OF Integer;
Type t4 = t3
```

t3 equivalent to t4

Type equivalence: **structural** equivalence

```
Type t5 = RECORD c: Integer; p: POINTER TO t5; END
RECORD;
Type t6 = RECORD c: Integer; p: POINTER TO t6; END
RECORD;
Type t7 =
   RECORD
      c: Integer;
      p: POINTER TO
         RECORD
            c: Integer;
            p: POINTER to t5;
         END RECORD;
END RECORD;
```

t5, t6, t7 are all (structurally) equivalent

In practice…

- Almost all modern languages use **name equivalence**
- Why?

Types: strong vs. weak

- Coercion
- Strongly typed: C, C++, Java,…
- Weakly typed: Perl, PHP, …
- Not everybody agrees on this classification

Output: 73

perl

```perl
$a=31;
$b="42x";
$c=$a+$b;
print $c;
```

warning: initialization makes integer from pointer without a cast

C

```c
main() {
 int a=31;
 char b[3]="42x";
 int c=a+b;
}
```

error: Incompatible type for declaration. Can't convert java.lang.String to int

Java

```java
class A {
 public static void main() {
   int a=31;
   String b ="42x";
   int c=a+b;
 }
}
```

Types: strong vs. weak

- Coercion
- Strongly typed: C, C++, Java,…
- Weakly typed: Perl, PHP, …
- Not everybody agrees on this classification

perl

```
$a=31;
$b="42x";
$c=$a+$b;
print $c;
```

Output: 73

C

```
main() {
 int a=31;
 char b[3]="42x";
 int c=a+b;
}
```

warning: initialization makes integer from pointer without a cast

Java

```
public class… {
 public static void main() {
   int a=31;
   String b ="42x";
   String c=b+a;
 }
}
```

OK

Coercions

- Suppose that at some point in the program, we expect a value of type T1 and find a value of type T2
- Is that acceptable?

```
float x = 3.141;
int y = x;
```

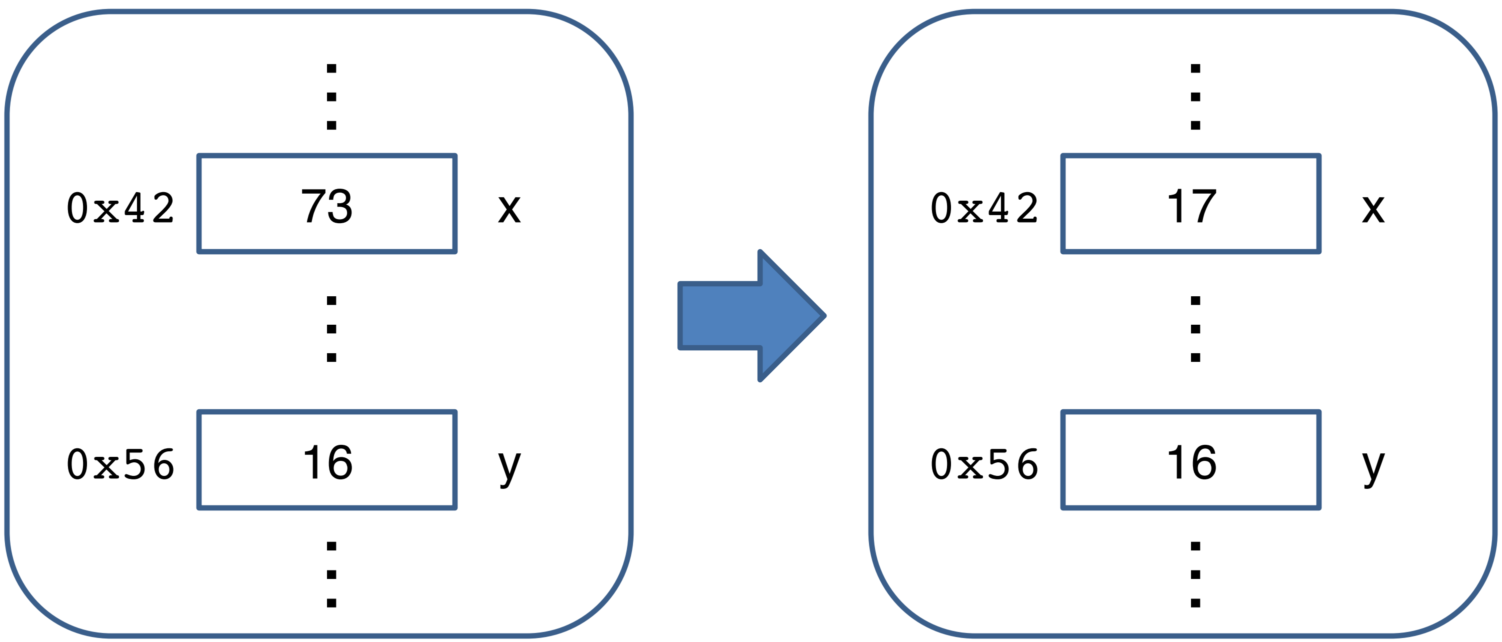
l-values and r-values

```
dst := src
```

- What is dst? dst is a **memory location** where the value should be stored
- What is src? src is a **value**
- "location" on the left of the assignment called an l-value
- "value" on the right of the assignment is called an r-value

Example: l-values and r-values

```
x:= y + 1
```



Example: l-values and r-values

```
x := A + 1

x := A[1]

x := A[A[1]]
```

l-values and r-values

expected

found

	lvalue	**rvalue**
lvalue	ok	deref
rvalue	error	ok

Type table

- All types in a compilation unit are collected in a type table

- For each type, table entry contains
  - type constructor: basic, record, array, pointer,…
  - size and alignment requirements
    - to be used later in code generation
  - types of components (if applicable)
    - example: types of record fields

Type checking vs inference

- Type **checking**
  - use declared types of variables
  - check types of operands are legal
  - infer types for expressions

- Type **inference**
  - automatically infer types of variables
    or show that there is no valid typing

Type checking: static vs dynamic

- Static type checking is **conservative**
  - most checking at compile time
  - any program that is determined to be well-typed is free from certain kinds of errors
  - may reject programs that cannot be statically determined as well typed (why?)

- Dynamic type checking
  - most checking at runtime
  - may accept more programs as valid (runtime info)
  - errors not caught at compile time
  - runtime cost

So far…

- Static correctness checking
- **Type Identification**: match applied occurrences of identifier to its defining occurrence
  - symbol table maintains this information
- **Type Checking**
  - which type combinations are legal
  - type equivalence: nominal vs structural
  - type coercion
  - each node in the AST of an expression represents either an l-value (location) or an r-value (value)

Type checking implementation

- How does this magic happen?
- We probably need to go over the AST?
- How does this relate to the clean formalism of the parser?

- Different approaches
  - attribute grammars
  - type systems

Type system (textbook definition)

- "A type system is a tractable **syntactic** method for **proving the absence of certain program behaviors** by classifying phrases according to the kinds of values they compute"

-- Types and Programming Languages
/ Benjamin C. Pierce

Type system

- A type system of a programming language is a way to define how "good" programs behave
- Good programs are well-typed programs
- Bad programs are not well-typed

Type rules

- which types can be combined with certain operator
- assignment of expression to variable
- formal and actual parameters of a method call

string string
“drive” + “drink”
string

int string
42 + “the answer”
ERROR

Type rules

- Specify for each operator
  - types of operands
  - type of result
- Basic types
  - building blocks for the type rules
  - example: int, boolean, (sometimes) string
- Type expressions
  - array types
  - function types
  - record types and classes

Type rules

If E1 has type int and E2 has type int,
then E1 + E2 has type int

$$\frac{E1 : int \qquad\qquad E2 : int}{E1 + E2 : int}$$

Notations for rules

- An inference rule consists of **premises** and **conclusion**

$$\frac{\mathbf{A} \qquad \mathbf{B}}{\mathbf{C}}$$

- An inference rule without any premises is an **axiom**

$$\frac{}{\mathbf{A}} \qquad\qquad \frac{}{\mathbf{B}}$$

- A proof is a sequence of lines, each of which is either an axiom or follows from earlier lines by an inference rule

$$\frac{\dfrac{}{\mathbf{A}} \qquad \dfrac{}{\mathbf{B}}}{\mathbf{C}}$$

More type rules

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

$$\frac{}{\text{int-literal : int}} \qquad \frac{}{\text{string-literal : string}}$$

$$\frac{E1 : \text{int} \qquad E2 : \text{int}}{E1 \text{ op } E2 : \text{int}} \qquad \text{op} \in \{ +, -, /, *, \% \}$$

$$\frac{E1 : \text{int} \qquad E2 : \text{int}}{E1 \text{ op } E2 : \text{boolean}} \qquad \text{op} \in \{ <=, <, >, >= \}$$

$$\frac{E1 : T \qquad E2 : T}{E1 \text{ op } E2 : \text{boolean}} \qquad \text{op} \in \{ ==, != \}$$

And even more type rules

$$\frac{E1 : \text{boolean} \qquad E2 : \text{boolean}}{E1 \text{ op } E2 : \text{boolean}} \qquad \text{op} \in \{ \&\&, || \}$$

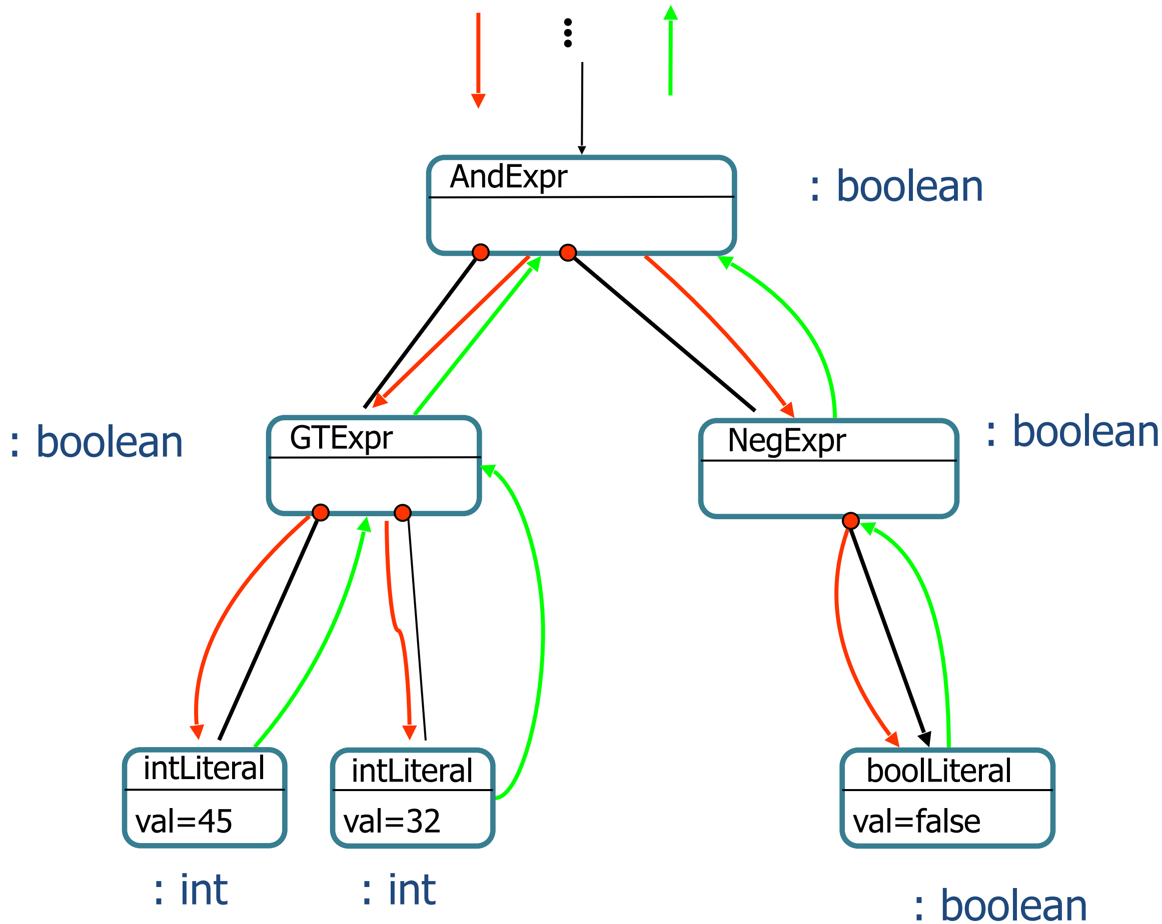
$$\frac{E1 : \text{int}}{- E1 : \text{int}} \qquad \frac{E1 : \text{boolean}}{! E1 : \text{boolean}}$$

$$\frac{E1 : T[]}{E1.\text{length} : \text{int}} \qquad \frac{E1 : T[] \qquad E2 : \text{int}}{E1[E2] : T} \qquad \frac{E1 : \text{int}}{\text{new } T[E1] : T[]}$$

Type checking implementation

- Traverse AST
- Assign types for AST nodes
- Use typing rules to compute node types

Example



45 > 32 && !false

$$\frac{E1 : \text{boolean} \qquad E2 : \text{boolean}}{E1 \text{ op } E2 : \text{boolean}}$$

$$\text{op} \in \{\ \&\&, ||\ \}$$

$$\frac{E1 : \text{boolean}}{!E1 : \text{boolean}}$$

$$\frac{E1 : \text{int} \qquad E2 : \text{int}}{E1 \text{ op } E2 : \text{boolean}}$$

$$\text{op} \in \{ <=, <, >, >= \}$$

$$\frac{}{\text{false} : \text{boolean}}$$

$$\frac{}{\text{int-literal} : \text{int}}$$

Plan

- Cool type rules
- Implementing type checking for Cool

Cool types

- The types are
  - Class names
  - SELF_TYPE
- The user **declares** types for identifiers
- The semantic analysis **infers** types for expressions
  - every expression has a unique type
- Cool **type rules** specify which operations are valid for which types
- The goal of **type checking** is to ensure that operations are used with the correct types
  - enforces intended interpretation of values
- Cool is **statically** typed: type checking during compilation

Notations for rules

$$\frac{\begin{array}{l}\textbf{Environment}_1 \vdash \textbf{Statement}_1 \\ \textbf{Environment}_2 \vdash \textbf{Statement}_2 \\ \ldots \\ \textbf{Environment}_n \vdash \textbf{Statement}_n\end{array}}{\textbf{Environment} \vdash \textbf{Statement}} \quad \textbf{[NAME] conditions}$$

- $A \vdash B$ means "given A, it is provable that B"
- $A \vdash B$ is called a judgement
- A is called context or environment
- B is called statement

Cool type judgements

- Cool type rules have judgements of the form

$$\underbrace{O,M,C}_{\textbf{type environment}} \vdash e: T$$

- **O** gives types to free identifiers in the current scope
- **M** gives information about the formal parameters and return type of methods
- **C** is the class in which expression e appears

Cool type environment

$$\underbrace{O,M,C}_{\textbf{type environment}} \vdash e: T$$

- **O** mapping Object Id's to types
  - symbol table for the current scope
  - O(x) = T
- **M** mapping methods to method signatures
  - M(K, f) = (A, B, D)
    means there is a method f(a:A, b:B): D defined in class K (or its ancestor)
- **C** the class in which expression e appears
  - used when SELF_TYPE is involved

Cool type environment

$$\underbrace{O,M,C}_{\textbf{type environment}} \vdash e: T$$

- Why separate object/methods?
- In Cool, the method and object identifiers live in different name spaces

Type rules

- Rules are schemas for inferring types of expressions

$$\frac{O, M, C \vdash e1 : Int \qquad O, M, C \vdash e2 : Int}{O, M, C \vdash e1 + e2 : Int} \qquad \frac{}{O,M,C \vdash int\_const : Int} \qquad \frac{O(id) = T}{O,M,C \vdash id : T}$$

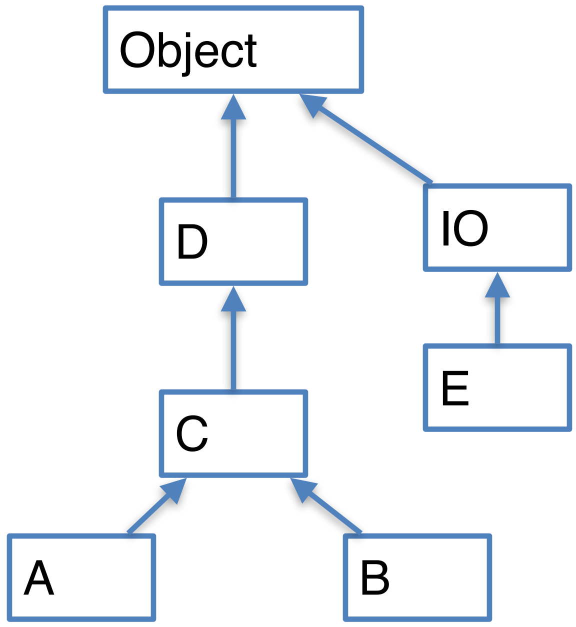
- Infer types by instantiating the schemas

$$\frac{O, M, C \vdash 1 : Int \qquad O, M, C \vdash 2 : Int}{O, M, C \vdash 1 + 2 : Int} \qquad \frac{}{O,M,C \vdash 1 : Int} \qquad \frac{O(y) = Int}{O,M,C \vdash y : Int}$$

$$\frac{O, M, C \vdash y : Int \qquad O, M, C \vdash (1 + 2) : Int}{O, M, C \vdash y + (1 + 2) : Int} \qquad \frac{}{O,M,C \vdash 2 : Int}$$

Subtyping

- Define a relation ≤ on classes
  - $X \leq X$
  - $X \leq Y$ if X inherits from Y
  - $X \leq Z$ if $X \leq Y$ and $Y \leq Z$



- Example
  - $A \leq C$
  - $B \leq$ Object
  - $E \not\leq D$ and $D \not\leq E$

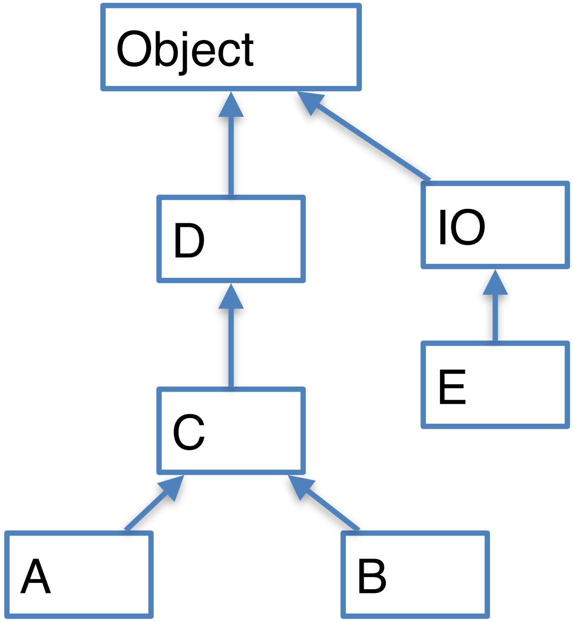
Least upper bounds

- Z is the least upper bound of X and Y
- lub(X, Y)=Z
  - $X \leq Z$ and $Y \leq Z$
    Z is **upper** bound
  - $X \leq Z'$ and $Y \leq Z' \Rightarrow Z \leq Z'$

    Z is the **least** upper bound

Least upper bounds

- In Cool, the least upper bound of two types is their **least common ancestor** in the inheritance tree



- Example
  - lub(A,B) = C
  - lub(C,D) = D
  - lub(C,E) = Object

Type rules: Assign

$$\frac{O(x) = T0 \quad O,M,K \vdash e1 : T1 \quad T1 \leq T0}{O,M,K \vdash x \leftarrow e1 : T1} \text{[Assign]}$$

```
class A {
        foo() : A { … }
};
class B inherits A { };
...
```

ERROR let x:B in x ← $\overbrace{\text{(new B).foo()}}^{A}$;

OK let x:A in x ←(new B).foo();

OK let x:Object in x ←(new B).foo();