Semantic Analysis

- type checking
- other context-sensitive syntactic properties

Semantic Analysis

- matching /appropriate types for operators
- number of arguments and arg. types for functions
- return types of functions
- redeclarations of variables /fields
- handling of recursive types
- scope of loop variables
- break only in for/while
- nil subtype of every record type

Symbol Tables

Environment (aka. symbol table)

$= \text{IDENTS} \rightarrow \text{TYPES} \times \text{LOCATION}$

Example:

```
function f(a:int, b:int, c:int)=
   (print_int(a+c);
    let var j := a+b
          var a := "hello"
    in print(a); print_int(j)
   end;
   print_int(b);
  )
```

$\sigma_1 = \sigma_0 + \{a \mapsto int, b \mapsto int, c \mapsto int\}$

$\sigma_2 = \sigma_1 + \{j \mapsto int\}$

$\sigma_3 = \sigma_2 + \{a \mapsto string\}$

Implementation

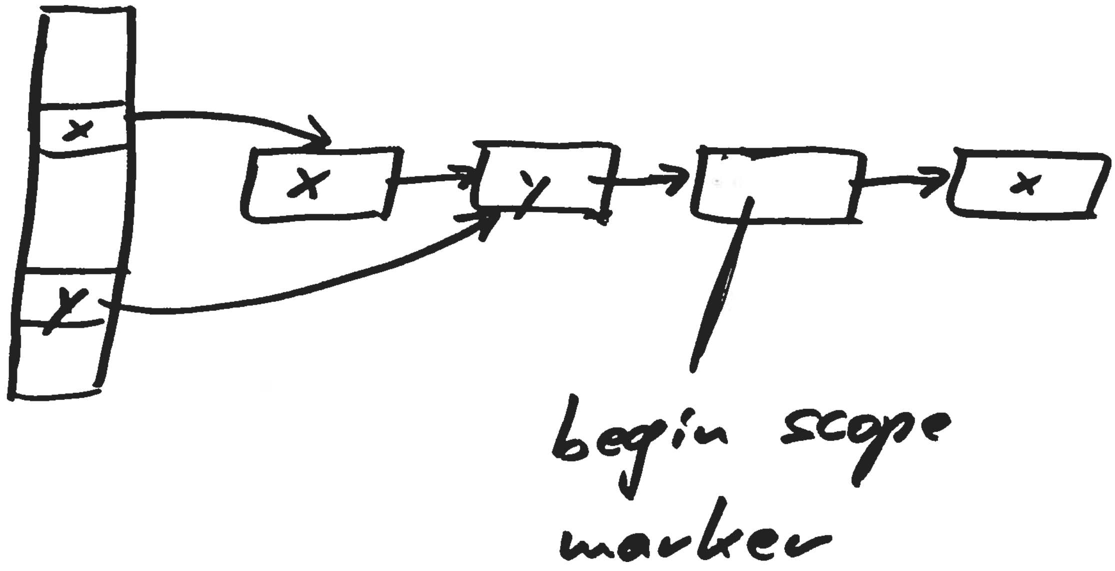
Symbol.Symbol:

- hashes strings into symbols

Symbol.Table:

- hashes symbols into bindings
- maintains environments with scopes

Data Structure

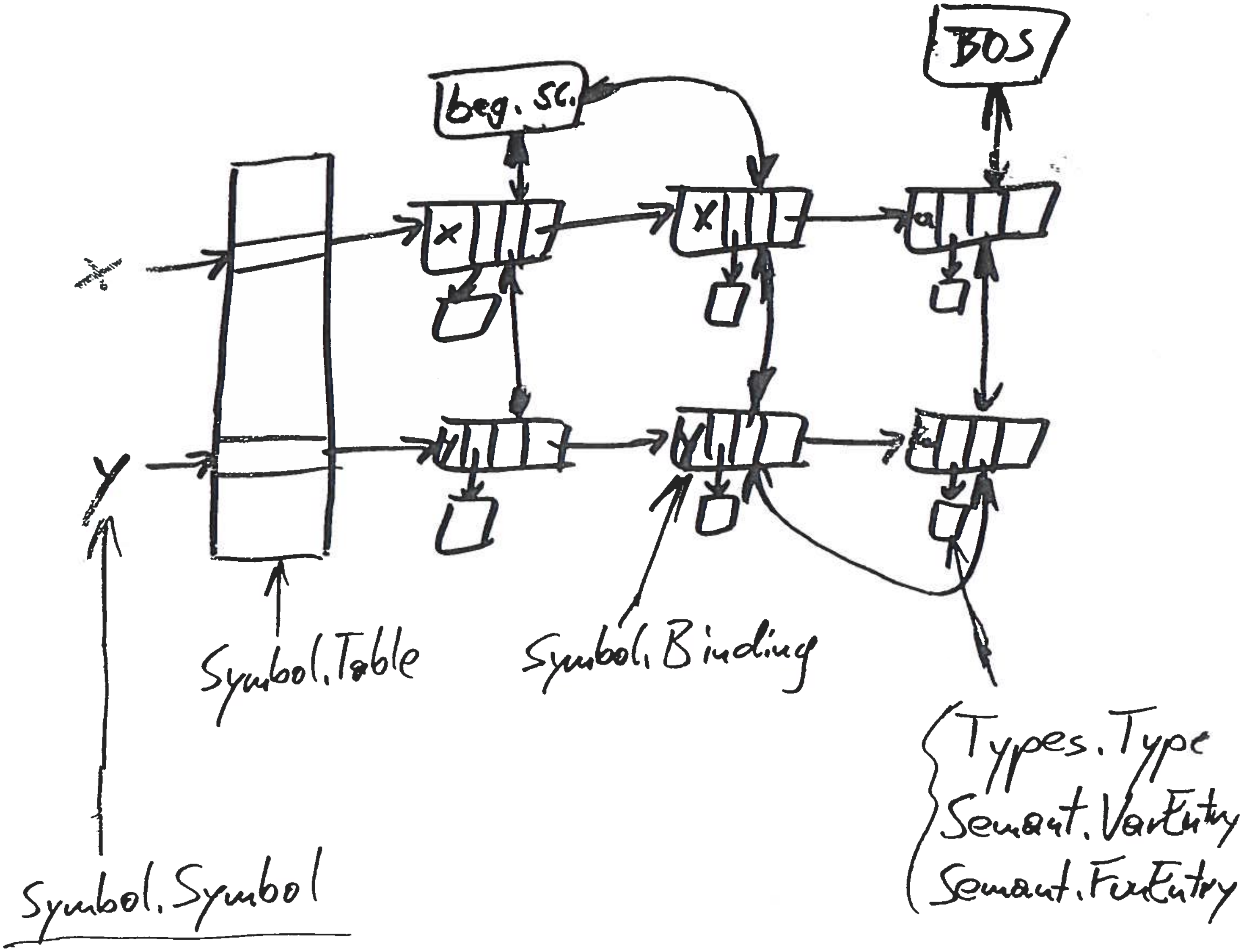


Operations on Symbol. Table:

```
void put (Symbol key,
          Object value)
Object get (Symbol key)
void beginScope()
void endScope()
java.util.Enumeration keys()
```

10/23

Symbol Tables in Tiger Compiler

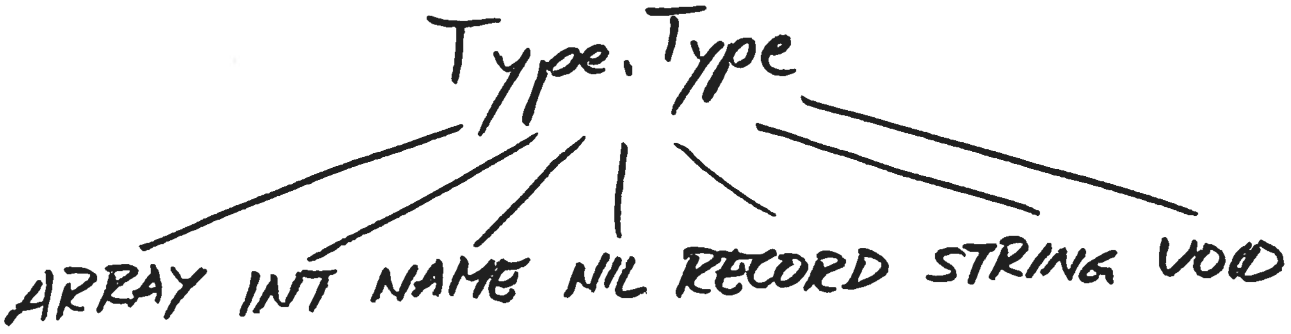


Semant.Env: — tenv — for types

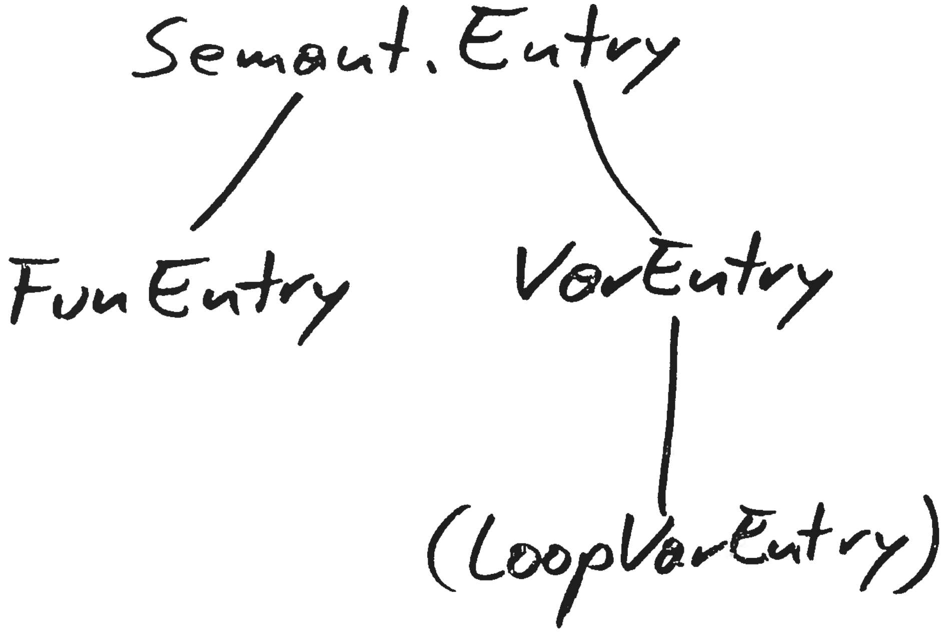
— venv — for variables/functions

Symbol Table Entries

Type Environment (tenv):



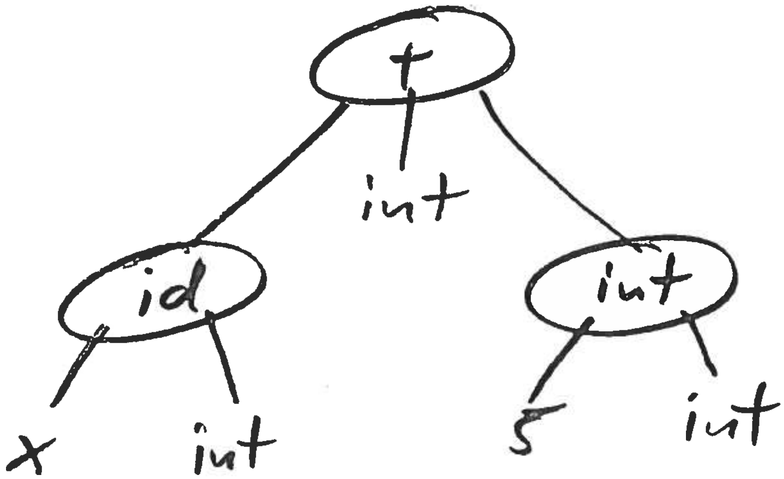
Value Environment (venv):



- Keep copy of symtab entry in tree!

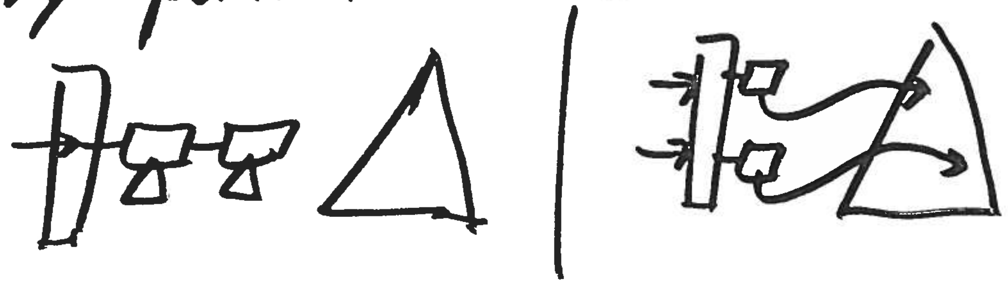
Type checking

$x + 5$



Symbol Table Design Decisions

- one namespace vs. multiple namesp.
- keep symbol info in symbol table vs. keep info in tree, let symtable entry point into tree



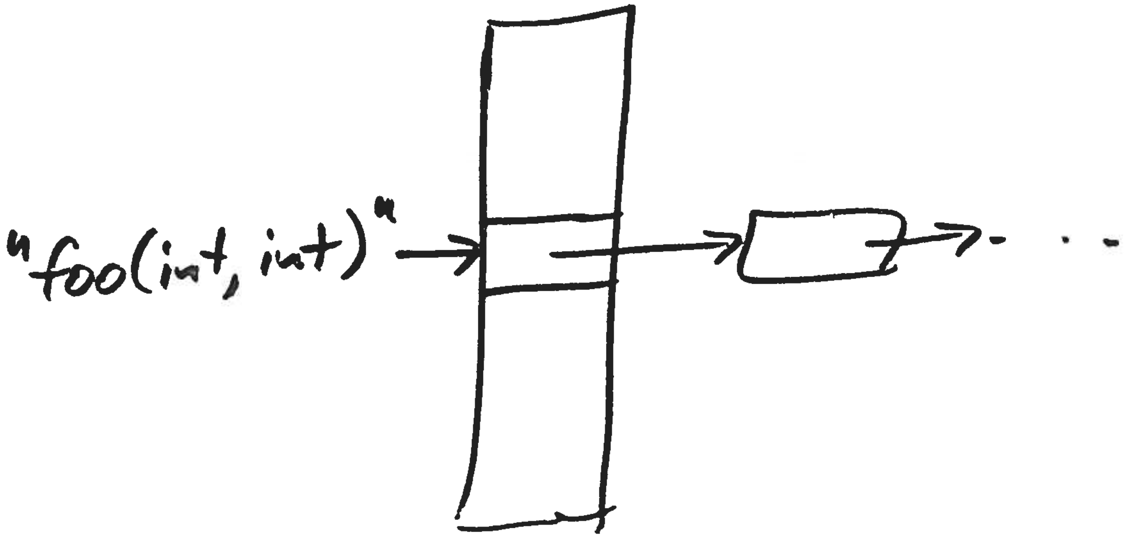
- one symbol table vs. one symbol table per scope
- destroy symtable after compiler pass vs. keep it around for rest of compilation and keep extending it
- imperative table data structure (hash table) vs. functional data structure (red-black trees)

Design Decisions depend on languag

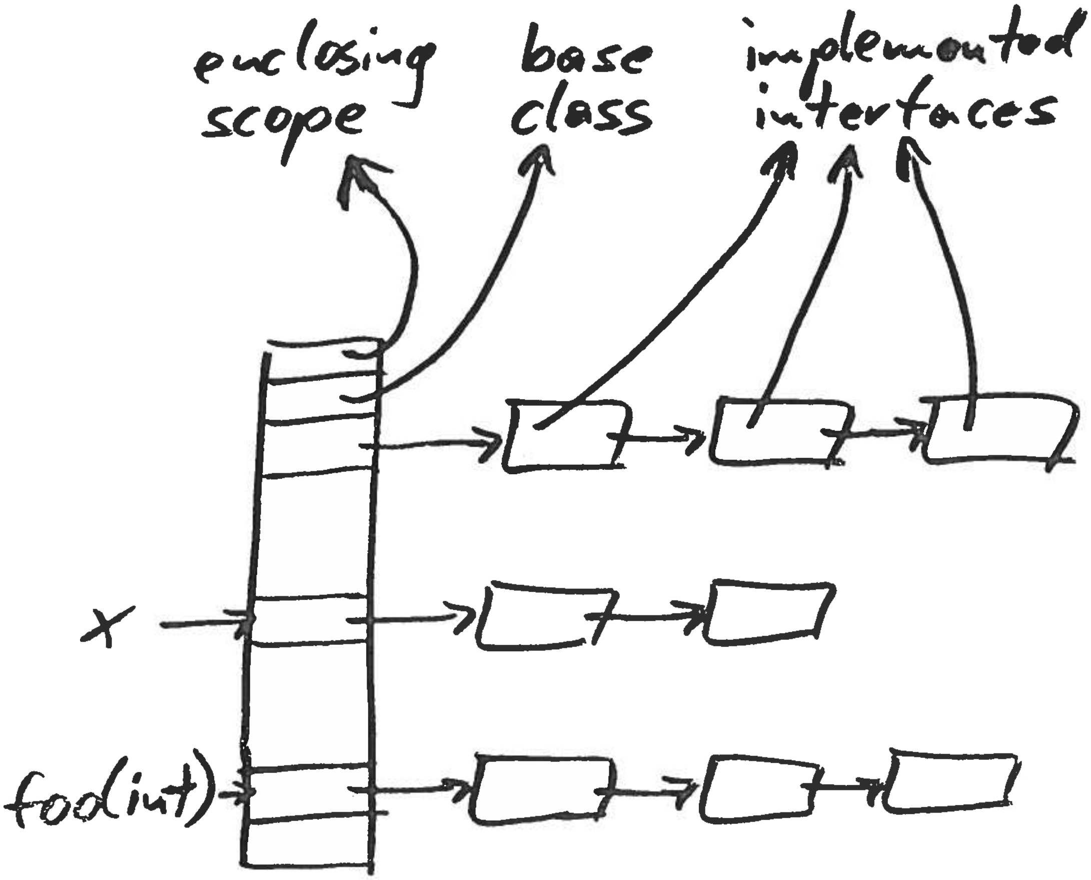
Java:

```
class A {
    int foo () {...}
}

class B {
    int bar () {...}
}

class C extends B {
    int blah (A x)
    {
        return bar() + x.foo();
    }
    int blah (B y) {...}
}
```

Symbol Table for Overloaded Methods/Functions

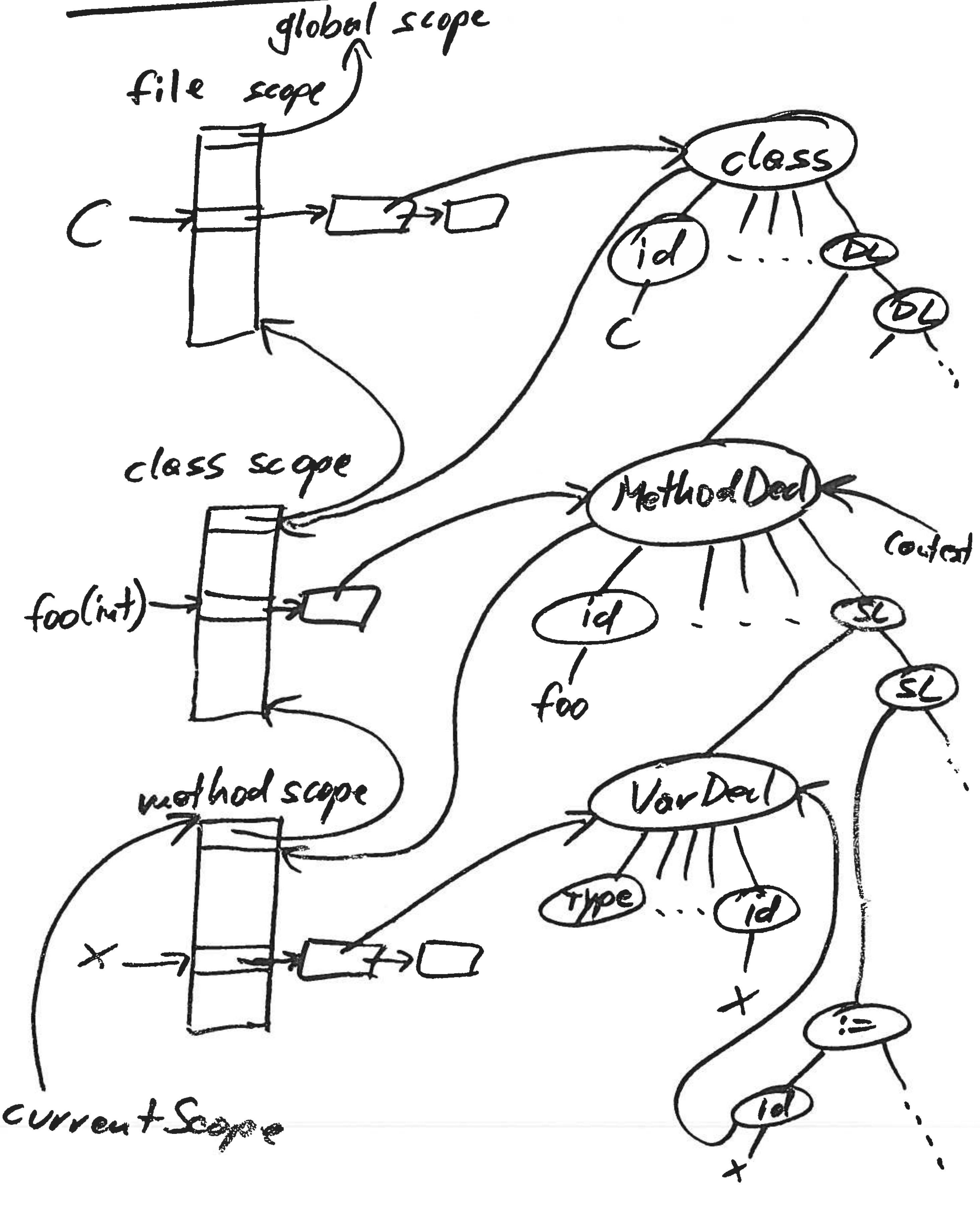


Example Design for Java



Symbol Table with Pointers into Parse Tree

10/2



Tree Traversals

In Tiger Compiler:

```
ExpTy transExp (Absyn.Exp e){
    ExpTy result;
    if (e == null)
        return new ExpTy (null, VOID);
    else if (e instanceof Absyn.VarExp)
        result = transExp ((Absyn.VarExp) e);
    else if ...
          .
          .
          .
    else throw new Error ("....");
    e.type = result.ty;
    return result;
}
```

Visitor Pattern

```
class Exp {
      ⋮
    abstract void accept (Visitor);
}

class VarExp extends ... {
      ⋮
    void accept (Visitor v) {
        v.visitVarExp(this);
    }
}
```

Visitor Pattern

```
class Visitor {
         void
    abstract visitVarExp(VarExp t);
        :
        :
}

class TransExp extends Visitor {
    void visitVarExp(VarExp t) {...}
    void visitAssign(Assign t) {
        t.lett.~~visit~~(this);
        t.right.~~visit~~(this);
        :          accept
        :
    }
    :
    :
}
```

Architectures for Tree Traversals

Given: Parse Tree Class Hierarchy
Write: Tree Traversal (e.g., Typechecker)

- object-oriented style
  (1 method in each of 100 classes)
- visitor using casts (Tiger compiler)
  (1 class with 100 methods, ugly)
- visitor using Visitor Pattern
  (better, inflexible for extensions of hierarchy)
- visitor using multimethods
  (better, requirers link-time check)
- statically type-safe visitor?

Type Conformance

- Builtin type

```
    type a = int
    var i : int
    var j : a
        .
        .
        .
    i := j;
```

- Records

  same type

- nil

  any record type

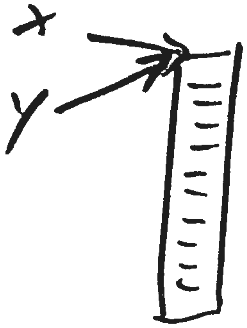
- arrays

  same type

Modula-2

```
var x : array [1..10] of integer;
    y : array [1..10] of integer;
```

```
x := y;
```



Inheritance

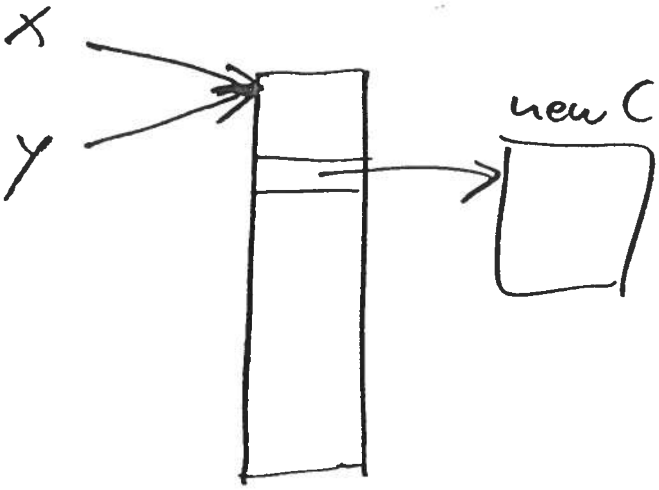
```
class C {...}
class D extends C {...}

C p = new D();
```

Structural Subtyping

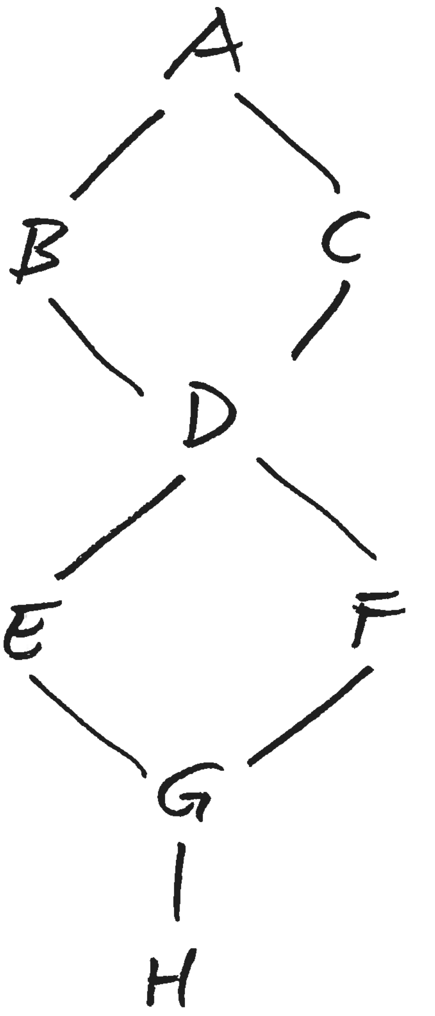
```
interface I {
    int foo();
}

class C {
    public int foo() {...}
    :
}

I p = new C();
```

Hole in Java's Type System

```
class C {...}
class D extends C {...}

D[] x = new D[10];
C[] y = x;
y[5] = new C(); ↯
```



Multiple Inheritance



Type Inference

ML:

```
- fun id(x) = x;
  val id : 'a → 'a

- fun length nil = 0
    | length (_::t) = 1 + length t;
  val length: 'a list → int

- fun map f nil = nil
    | map f (h::t) = f(h)::(map f t);
  val map ('a→'b) → 'a list → 'b list

- val l = map length [[1,2,3], [4,5]}
  val l : int list = [3, 2]
```