In Java, the standard model is a mutable graph of objects.

It seems natural to represent references to symbols using mutable fields (initially null, `resolved` during name analysis).

Alternative way is functional:
- Store the **backbone** of the graph as an algebraic data type (immutable).
- Pass around a map linking from identifiers to their declarations.

Note that a field `class A { var f:T }` is like `f: Map[A,T]`. 
Symbol Table ($\Gamma$) Contents

- Map identifiers to the symbol with relevant information about the identifier
- All information is derived from syntax tree - symbol table is for efficiency
  - In old one-pass compilers there was only symbol table, no syntax tree
  - In modern compiler: we could always go through entire tree, but symbol table can give faster and easier access to the part of syntax tree, or some additional information

- Goal: efficiently supporting phases of compiler
- In the name analysis phase:
  - Finding which identifier refers to which definition
  - We store definitions
- What kinds of things can we define? What do we need to know for each ID?
  - Variables (globals, fields, parameters, locals):
    - Need to know types, positions - for error messages
    - Later: memory layout. To compile $x.f = y$ into $\text{memcpy}(\text{addr}_y, \text{addr}_x+6, 4)$
      - E.g. 3rd field in an object should be stored at offset e.g. $+6$ from the address of the object
      - The size of data stored in $x.f$ is $4$ bytes
    - Sometimes more information explicit: whether variable local or global
      - Methods, functions, classes: recursively have with their own symbol tables
Functional: Different Points, Different $\Gamma$

class World {
    int sum;
    void add(int foo) {
        sum = sum + foo;
    }
    void sub(int bar) {
        sum = sum - bar;
    }
    int count;
}
class World {
    int sum;
    void add(int foo) {
        sum = sum + foo;
    }
    void sub(int bar) {
        sum = sum - bar;
    }
    int count;
}

\[ \Gamma_0 = \{(\text{sum, int}), (\text{count, int})\} \]

\[ \Gamma_1 = \Gamma_0 \ [ \text{foo:=int} ] \]
change table, record change

\[ \Gamma_2 = \Gamma_0 \ [ \text{bar:=int} ] \]
change table, record change

revert changes from table
revert changes from table
Imperative Symbol Table

- Hash table, mutable Map[ID,Symbol]
- Example:
  - hash function into array
  - array has linked list storing (ID,Symbol) pairs
- Undo stack: to enable entering and leaving scope
- Entering new scope (function,block):
  - add beginning-of-scope marker to undo stack
- Adding nested declaration (ID,sym)
  - lookup old value symOld, push old value to undo stack
  - insert (ID,sym) into table
- Leaving the scope
  - go through undo stack until the marker, restore old values
class World {
    int sum;
    void add(int foo) {
        sum = sum + foo;
    }
    void sub(int bar) {
        sum = sum - bar;
    }
    int count;
}
Functional Symbol Table Implemented

- Typical: Immutable Balanced Search Trees

```scala
sealed abstract class BST
case class Empty() extends BST
case class Node(left: BST, value: Int, right: BST) extends BST
```

- Updating returns new map, keeping old one
  - lookup and update both $\log(n)$
  - update creates new path (copy $\log(n)$ nodes, share rest!)
  - memory usage acceptable

Simplified. In practice, BST[A], store Int key and value A
def contains(key: Int, t : BST): Boolean = t match {
  case Empty() => false
  case Node(left,v,right) => {
    if (key == v) true
    else if (key < v) contains(key, left)
    else contains(key, right)
  }
}

Running time bounded by tree height.

contains(6,t) ?
**Insertion**

```scala
def add(x : Int, t : BST) : Node = t match {
  case Empty() => Node(Empty(),x,Empty())
  case t @ Node(left,v,right) => {
    if (x < v) Node(add(x, left), v, right)
    else if (x==v) t
    else Node(left, v, add(x, right))
  }
}
```

Both `add(x,t)` and `t` remain accessible.

Running time and newly allocated nodes bounded by tree height.
Balanced Trees: Red-Black Trees

12 Binary Search Trees 286
12.1 What is a binary search tree? 286
12.2 Querying a binary search tree 289
12.3 Insertion and deletion 294
12.4 Randomly built binary search trees 299

13 Red-Black Trees 308
13.1 Properties of red-black trees 308
13.2 Rotations 312
13.3 Insertion 315
13.4 Deletion 323

14 Augmenting Data Structures 339
14.1 Dynamic order statistics 339
14.2 How to augment a data structure 345
14.3 Interval trees 348
Balanced Tree: Red Black Tree

Goals:

- ensure that tree height remains at most log(size)
- preserve efficiency of individual operations:
  - rebalancing arbitrary tree: could cost O(n) work

Solution: maintain mostly balanced trees: height still O(log size)

```scala
sealed abstract class Color
  case class Red() extends Color
  case class Black() extends Color

sealed abstract class Tree
  case class Empty() extends Tree
  case class Node(c: Color, left: Tree, value: Int, right: Tree) extends Tree
```
Properties of red-black trees

A red-black tree is a binary search tree with one extra bit of storage per node: its color, which can be either RED or BLACK. By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately balanced.

Each node of the tree now contains the attributes color, key, left, right, and p. If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL. We shall regard these NILs as being pointers to leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree.

A red-black tree is a binary tree that satisfies the following red-black properties:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

From 4. and 5.: tree height is $O(\log \text{size})$.
Analysis is similar for mutable and immutable trees.
for immutable trees: see book by Chris Okasaki
Balancing

def balance(c: Color, a: Tree, x: Int, b: Tree): Tree = (c, a, x, b) match {
  case (Black(), Node(Red(), Node(Red(), a, xV, b), yV, c), zV, d) =>
    Node(Red(), Node(Black(), a, xV, b), yV, Node(Black(), c, zV, d))
  case (Black(), Node(Red(), a, xV, Node(Red(), b, yV, c)), zV, d) =>
    Node(Red(), Node(Black(), a, xV, b), yV, Node(Black(), c, zV, d))
  case (Black(), a, xV, Node(Red(), Node(Red(), b, yV, c), zV, d)) =>
    Node(Red(), Node(Black(), a, xV, b), yV, Node(Black(), c, zV, d))
  case (Black(), a, xV, Node(Red(), b, yV, Node(Red(), c, zV, d))) =>
    Node(Red(), Node(Black(), a, xV, b), yV, Node(Black(), c, zV, d))
  case (c, a, xV, b) => Node(c, a, xV, b)
}
Insertion

```scala
def add(x: Int, t: Tree): Tree = {
  def ins(t: Tree): Tree = t match {
    case Empty() => Node(Red((), Empty(), x, Empty()))
    case Node(c, a, y, b) =>
      if (x < y) balance(c, ins(a), y, b)
    else if (x == y) Node(c, a, y, b)
    else balance(c, a, y, ins(b))
  }
  makeBlack(ins(t))
}

def makeBlack(n: Tree): Tree = n match {
  case Node(Red((), l, v, r)) => Node(Black((), l, v, r))
  case _ => n
}

Modern object-oriented languages (e.g. Scala) support abstraction and functional data structures. Just use Map from Scala.
```
Exercise

Determine the output of the following program assuming static and dynamic scoping. Explain the difference, if there is any.

```scala
object MyClass {
  val x = 5
  def foo(z: Int): Int = { x + z }
  def bar(y: Int): Int = {
    val x = 1; val z = 2
    foo(y)
  }
  def main() {
    val x = 7
    println(foo(bar(3)))
  }
}
```
```scala
i = 0
while (i < 10) {
  i = i + 1
}
```
Evaluating an Expression

scala prompt:
>def min1(x : Int, y : Int) : Int = { if (x < y) x else y+1 }
  min1: (x: Int,y: Int)Int
>min1(10,5)
  res1: Int = 6

How can we think about this evaluation?

x \rightarrow 10
y \rightarrow 5
if (x < y) x else y+1
Computing types using the evaluation tree

scala prompt:
>def min1(x : Int, y : Int) : Int = { if (x < y) x else y+1 }
min1: (x: Int,y: Int)Int
>min1(10,5)
res1: Int = 6

How can we think about this evaluation?

x : Int → 10
y : Int → 5
if (x < y) x else y+1
We can compute types without values

scala prompt:

```scala
> def min1(x : Int, y : Int) : Int = { if (x < y) x else y+1 }
min1: (x: Int,y: Int)Int
> min1(10,5)
res1: Int = 6
```

How can we think about this evaluation?

```
x : Int
y : Int
if (x < y) x else y+1
```
We do not like trees upside-down
Leaves are Up

type rules move from leaves to root
Type Judgements and Type Rules

- \( e \) type checks to \( T \) under \( \Gamma \) (type environment)
  \[ \Gamma \vdash e : T \]
  - Types of constants are predefined
  - Types of variables are specified in the source code
- If \( e \) is composed of sub-expressions

\[ \Gamma \vdash e_1 : T_1 \quad \cdots \quad \Gamma \vdash e_n : T_n \]

\[ \Gamma \vdash e : T \]
Type Judgements and Type Rules

\[ \Gamma \vdash e : T \]

if the (free) variables of e have types given by the type environment \( \Gamma \), then \( e \) (correctly) type checks and has type \( T \)

\[ \begin{array}{c}
\Gamma \vdash e_1 : T_1 \quad \cdots \\
\Gamma \vdash e_n : T_n \\
\hline \\
\Gamma \vdash e : T 
\end{array} \]

Type rule

If \( e_1 \) type checks in \( \Gamma \) and has type \( T_1 \) and ... and \( e_n \) type checks in \( \Gamma \) and has type \( T_n \) then \( e \) type checks in \( \Gamma \) and has type \( T \)
Type Rules as Local Tree Constraints

\[
\begin{align*}
x : \text{Int} \\
y : \text{Int} \\
b : \text{Boolean} \\
\text{if} \quad \text{Int} \quad \text{Int} \\
\text{Int} \\
\text{Boolean} \\
\end{align*}
\]

Type Rules

\[
\begin{align*}
e_1 : \text{Int} & \quad e_2 : \text{Int} \\
e_1 < e_2 : \text{Boolean} \\
\text{for every type } T, \text{ if } b \text{ has type Boolean, and } \\
\text{then } \\
b : \text{Boolean} & \quad e_1 : T \quad e_2 : T \\
\text{if}(b) \quad e_1 \text{ else } e_2 : T \\
\end{align*}
\]
Type Rules with Environment

**Type Rules**

\[
\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T}
\]

\[
\frac{\text{Int Const}(k) : \text{Int}}{}
\]

\[
\frac{\Gamma \vdash e_1 : \text{Int} \quad \Gamma \vdash e_2 : \text{Int}}{\Gamma \vdash (e_1 < e_2) : \text{Boolean}}
\]

\[
\frac{\Gamma \vdash e_1 : \text{Int} \quad \Gamma \vdash e_2 : \text{Int}}{\Gamma \vdash (e_1 + e_2) : \text{Int}}
\]

\[
\frac{(\text{then}) \text{ in the (same) environment } \Gamma \text{ the expression } e_1 < e_2 \text{ has type Bool.}}{}
\]

\[
\frac{\Gamma \vdash b : \text{Boolean} \quad \Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash (\text{if}(b) \ e_1 \text{ else } e_2) : T}
\]
Type Checker Implementation Sketch

def typeCheck(Γ : Map[ID, Type], e : ExprTree) : TypeTree = {
    e match {
        case Var(id) => { ?? }
        case If(c,e1,e2) => { ?? }
        ...
    }
}

case Var(id) => { Γ(id) match
    case Some(t) => t
    case None => error(UnknownIdentifier(id,id.pos))
}
Type Checker Implementation Sketch

- \texttt{case If(c,e1,e2) => \{ }
  
  \texttt{val tc = typeCheck(\Gamma, c)}
  
  \texttt{if (tc != BooleanType) error(IfExpectsBooleanCondition(e.pos))}
  
  \texttt{val t1 = typeCheck(\Gamma, e1); val t2 = typeCheck(\Gamma, e2)}
  
  \texttt{if (t1 != t2) error(IfBranchesShouldHaveSameType(e.pos))}
  
  \texttt{t1}
  
\texttt{\}}
Derivation Using Type Rules

Let $\Gamma = \{(x, \text{Int}), (y, \text{Int})\}$

\[
\begin{align*}
\Gamma &\vdash x : \text{Int} \\
\Gamma &\vdash y : \text{Int} \\
\Gamma &\vdash x : \text{Int} \\
\Gamma &\vdash y : \text{Int} \\
\Gamma &\vdash (x < y) : \text{Boolean} \\
\Gamma &\vdash (\text{if}(x < y) \times \text{else} \ y + 1) : \text{Int}
\end{align*}
\]
Type Rule for Function Application

\[ \Gamma \vdash e_1 : T_1 \quad \cdots \quad \Gamma \vdash e_n : T_n \quad \Gamma \vdash f : (T_1 \times \cdots \times T_n) \rightarrow T \]

\[ \Gamma \vdash f(e_1, \cdots, e_n) : T \]
Type Rule for Function Application
[Cont.]

We can treat operators as variables that have function type

\[ + : \text{Int} \times \text{Int} \rightarrow \text{Int} \]
\[ < : \text{Int} \times \text{Int} \rightarrow \text{Boolean} \]
\[ \&\& : \text{Boolean} \times \text{Boolean} \rightarrow \text{Boolean} \]

We can replace many previous rules with application rule:

\[
\Gamma \vdash e_1 : T_1 \quad \ldots \quad \Gamma \vdash e_n : T_n \quad \Gamma \vdash f : (T_1 \times \cdots \times T_n) \rightarrow T \]

\[
\Gamma \vdash f(e_1, \ldots, e_n) : T
\]

\[
\Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_2 : \text{Bool} \quad \Gamma \vdash \&\& : (\text{Bool} \times \text{Bool}) \rightarrow \text{Bool}
\]

\[
\Gamma \vdash e_1 \&\& e_2 : \text{Bool}
\]
Computing the Environment of a Class

object World {
    var data : Int
    var name : String
    def m(x : Int, y : Int) : Boolean { ... }
    def n(x : Int) : Int {
        if (x > 0) p(x - 1) else 3
    }
    def p(r : Int) : Int = {
        var k = r + 2
        m(k, n(k))
    }
}

\[ \Gamma_0 = \{ \]
\[
    (\text{data, Int}), \]
\[
    (\text{name, String}), \]
\[
    (\text{m, Int \times Int \rightarrow Boolean}), \]
\[
    (\text{n, Int \rightarrow Int}), \]
\[
    (\text{p, Int \rightarrow Int}) \]
\[
\}

We can type check each function m, n, p in this global environment.
Extending the Environment

class World {
    var data : Int
    var name : String
    def m(x : Int, y : Int) : Boolean { ... }
    def n(x : Int) : Int {
        if (x > 0) p(x - 1) else 3
    }
    def p(r : Int) : Int = {
        var k: Int
        k = r + 2
        m(k, n(k))
    }
}

\( \Gamma_0 = \{ \) 
\( (\text{data}, \text{Int}) \),
\( (\text{name}, \text{String}) \),
\( (m, \text{Int} \times \text{Int} \rightarrow \text{Boolean}) \),
\( (n, \text{Int} \rightarrow \text{Int}) \),
\( (p, \text{Int} \rightarrow \text{Int}) \} \)

\( \Gamma_1 = \Gamma_0 \oplus \{(r, \text{Int})\} \)
\( \Gamma_2 = \Gamma_1 \oplus \{(k, \text{Int})\} = \Gamma_0 \cup \{(r, \text{Int}), (k, \text{Int})\} \)
Type Rule for Method Definitions

\[ \Gamma \Theta \{(x_1,T_1), \ldots, (x_n,T_n)\} \vdash e : T \]

\[ \Gamma \vdash (\text{def } m(x_1:T_1, \ldots, x_n:T_n) : T = e) : \text{ok} \]

Type Rule for Assignments

\[ (x,T) \in \Gamma \quad \Gamma \vdash e : T \]

\[ \Gamma \vdash (x = e) : \text{void} \]

Type Rules for Block: \{ var x_1:T_1 \ldots var x_n:T_n; s_1; \ldots s_m; e \}

\[ \Gamma \Theta \{(x_1,T_1), \ldots, (x_n,T_n)\} \]

\[ \vdash S_i : \text{void} \]

\[ \vdash S_n : \text{void} \]

\[ \vdash e : T \]

\[ \Gamma \vdash \{ \text{var } x_1:T_1; \ldots; \text{var } x_n:T_n; S_1; \ldots; S_n; e \} : T \]
Blocks with Declarations in the Middle

\[
\Gamma \vdash \text{e} : T \\
\rightarrow \Gamma \vdash \{ \text{e} \} : T
\]
just
expression

\[
\Gamma \vdash \{ \} : \text{void}
\]
empty

\[
\Gamma \Theta \{(x:T_1)\} \vdash \{ t_2 ; \ldots ; t_n \} : T \\
\Gamma \vdash \{ \text{var} \ x : T_1 ; t_2 ; \ldots ; t_n \} : T
\]
declaration is first

\[
\Gamma \vdash \text{s}_1 : \text{void} \quad \Gamma \vdash \{ t_2 ; \ldots ; t_n \} : T \\
\Gamma \vdash \{ \text{s}_1 ; t_2 ; \ldots ; t_n \} : T
\]
statement is first
Rule for While Statement

\[ \Gamma \vdash b : \text{Boolean} \quad \Gamma \vdash s : \text{void} \]

\[ \Gamma \vdash (\text{while}(b) \ s) : \text{void} \]
Rule for a Method Call

class T0 {
    ...
    def m(x1:T1, ..., xn:TN): T = {
        ...
    }
    ...
}

\[ \Gamma \vdash x : T_0 \quad \Gamma_0 \vdash m : T_0 \times T_1 \times \ldots \times T_n \rightarrow T \quad \forall i \in \{1, 2, \ldots, n\} \exists \Gamma_i \vdash e_i : T_i \]

\[ \Gamma \vdash x.m(e_1, \ldots, e_n) : T \]

\[ m(x_1, e_1, \ldots, e_n) \]
Type Checking Expression in a Body

class World {
  var data : Int
  var name : String
  def m(x : Int, y : Int) : Boolean { ... }
  def n(x : Int) : Int {
    if (x > 0) p(x − 1) else 3
  }
  def p(r : Int) : Boolean = {
    var k: Int
    k = r + 2
    m(k, n(k))
  }
}

\[ \Gamma_0 = \{ \]
\[ \text{(data, Int)}, \]
\[ \text{(name, String)}, \]
\[ \text{(m, Int \times \text{Int} \rightarrow \text{Boolean})}, \]
\[ \text{(n, Int \rightarrow \text{Int})}, \]
\[ \text{(p, Int \rightarrow \text{Int})} \}

\[ \Gamma_2 \vdash k : \text{Int} \]
\[ \Gamma_2 \vdash n : \text{Int} \rightarrow \text{Int} \]
\[ \Gamma_2 \vdash k : \text{Int} \]
\[ \frac{\Gamma_2 \vdash n(k) : \text{Int}}{
  \Gamma_2 \vdash m(k, n(k)) : \text{Bool}} \]
Example to Type Check

object World {
    var z : Boolean
    var u : Int
    def f(y : Boolean) : Int {
        z = y
        if (u > 0) {
            u = u – 1
            var z : Int
            z = f(!y) + 3
            z+z
        } else {
            0
        }
    }
}

Γ₀ = { (z, Boolean), (u, Int), (f, Boolean → Int) }

Γ₁ = Γ₀ ⊕ {(y, Boolean)}

Γ₁ ⊢ z: Boolean
Γ₁ ⊢ y: Boolean
Γ₁ ⊢ (z=y): void

Exercise:

Γ ⊢ if(u > 0){ body } else { 0 }: Int
Solution

\[
\begin{align*}
\Gamma \vdash (u, \text{Int}) &\quad (a, \text{Int}) \in \Gamma' \\
\Gamma \vdash u : \text{Int} &\quad \vdash 1 : \text{Int} \\
\Gamma \vdash u > 0 : \text{Boolean} & \quad \Gamma' \vdash f(y) : \text{Int} \\
\Gamma' \vdash y : \text{Boolean} & \quad \Gamma' \vdash f : \text{Boolean} \rightarrow \text{Int} \\
\Gamma' \vdash z : \text{Int} & \quad \Gamma' \vdash \{\text{z=f(y)+3; z+z}\} : \text{Int} \\
\Gamma' \vdash \{\text{z=f(y)+3; z+z}\} : \text{Int} & \quad \vdash 0 : \text{Int} \\
\Gamma \vdash \text{if} \ (u > 0) \ \{ \ u = u - 1; \ \text{var z : Int; z = f(y) + 3; z + z} \} \ \text{else} \ \{ \ 0 \} : \text{Int}
\end{align*}
\]