Lecture I:

Type Checking & Basic Code Analysis

CSE401/501m:

Introduction to Compiler Construction

Instructor: Gilbert Bernstein

Administrivia

- HW2 — LR parsing — was due last night
  - try to avoid using up your late days!
- Parser + AST/printing is due next Thurs. May 1
  - How's it going?
- Mini-HW3 — LL grammars — due MONDAY May 5
  - Homework is available and posted
  - More on LL grammars and hw 3 in sections next Thu
  - Only **1 late day maximum** (solutions handout before…)
- Midterm exam on Fri. May 8
  - Topic list and old exams on the website now!

Administrivia (Monday)

- Parser + AST/printing due Thursday
  - ✦ How's it going?
- Mini-HW3 due next Monday
  - ✦ More on LL grammars and hw 3 in sections this Thu
- Midterm exam on Fri. May 8
  - ✦ Topic list and old exams on the website now!

Overview

What makes a program “legal”?

The Checking Pass(es)

Symbol Tables & Names

Types

Relationships Between Types

Wrapup

Overview

What makes a program "legal"?

The Checking Pass(es)

Symbol Tables & Names

Types

Relationships Between Types

Wrapup

What do we need to know to check and verify that this is a legal program?

```
class C {
  int a;
  C(int initial) {
    a = initial;
  }
  void setA(int val) {
    a = val;
  }
}
```

```
class Main {
  public static void main() {
    C c = new C(17);
    c.setA(42);
  }
}
```

What do we need to know to check and verify that this is a legal program?

```
class C {
  int a;
  C(int initial) {
    a = initial;
  }
  void setA(int val) {
    a = val;
  }
}


class Main {
  public static void main() {
    C c = new C(17);
    c.setA(42);
  }
}
```

Some Things to Check

no class already declared named C
don't allow instantiating static classes
int is a type
a hasn't already been defined
the constructor's name is the name of the class
check that int is a type
check that the name initial isn't another argument
check that both variables in line 4 are in scope (defined)
line 4: check that both types are the same or that the right-hand-side (RHS) can be cast to the LHS type
line 4: Make sure that the LHS 'a' is mutable
line 4: in the constructor — check that a is assigned a value (if it were declared as final)

Beyond Syntactic Validity

- Not every program that is grammatical can be compiled
  - Has a variable that's being used been declared?
  - Do basic data types make sense in arithmetic and boolean expressions?
  - In the assignment `x=y;` is the value of `y` assignable to `x`? Is `x` something that can be assigned to?
  - Does a method have the right number and types of arguments?
  - In a selector `p.q`, is `q` a field or method of the type of `p`?
  - Is variable `x` guaranteed to be initialized before it is used?
  - Could `p` be null when `p.q` is executed?
  - etc…

Beyond Syntactic Validity

- Not every program that is grammatical can be compiled

Names

- Has a variable that's being used been declared?
- Do basic data types make sense in arithmetic and boolean expressions?
- In the assignment `x=y;` is the value of y assignable to x? Is x something that can be assigned to?
- Does a method have the right number and types of arguments?
- In a selector `p.q`, is `q` a field or method of the type of `p`?

Types

- Is variable x guaranteed to be initialized before it is used?
- Could `p` be null when `p.q` is executed?

Runtime?

- etc…

Dynamic vs. Static Errors

- **Static errors** — your compiler tells you there's a problem
  - ✦ No need to supply a program input
  - ✦ Will prevent successful compilation of your program
  - ✦ e.g. type checking errors, undefined variables, etc.
- **Dynamic errors** — your program crashes or behaves in a bad way
  - ✦ May or may not appear, depending on specific input
  - ✦ Compiler will let you compile the code
  - ✦ e.g. null pointer exception in Java

"Strength" of (Type) Checking

Weaker Checks

More Dynamic Errors
Less Static Errors

Stronger Checks

Less Dynamic Errors
More Static Errors

Static-Enough Checking

- Users of a programming language often like **dynamic** (type) checking, because it gives *them* more *freedom* 
  - ✦ "Yes, my program will crash if I pass in the wrong type, but I won't do that!"
- Compiler Implementers often like **static** (type) checking, because it gives *them* more *freedom* 
  - ✦ "Because the checker doesn't allow programs with ***___possible problem X___***, I don't need to worry about generating code to handle ***___possible problem X___***."
- Checking allows us to make ***useful assumptions*** about our programs, assumptions we can take advantage of

Example Checks — Ids & Literals

- Wherever we see a name *id*
  - **Check** — *id* has been declared and is in scope
  - **Type** — based on looking up *id*

- a Literal — *v*
  - **Check**
  - **Type** — immediate, e.g. 3 is an integer

Example Checks — Binary Ops

- Binary operator — $exp_1\ op\ exp_2$
  - **Check** — exp_1 and exp_2 have correct and consistent types as specified by op.
    - e.g. both $+$ and $<$ allow for two integers as arguments, but not two Booleans. $\&\&$ is the opposite
  - **Type** — as specified by op, potentially dependent on the types of the arguments.
    - e.g. $+$ with integer arguments produces an integer, whereas $<$ and $\&\&$ always produce Booleans.

Example Checks — Assignment

- Assignment — $exp_1 = exp_2$
  - **Check** — exp_1 can be assigned to (is a valid lvalue)
  - **Check** — the type of exp_2 can be coerced into the type of exp_1
    - if the types exp_1 and exp_2 are the same, or
    - e.g. integers can be converted into doubles, or
    - exp_2 is a subclass of exp_1
  - **Type** — Statements don't have types

Example Checks — Cast

- Cast — $(exp_1)\ exp_2$
  - **Check** — exp_1 is a type
  - **Check** — exp_2 can be coerced into the type of exp_1
    - same as for assignment (upcast, coercion, etc.)
    - (depending on language) downcast — exp_1 is a subclass of the type of exp_2; in this case the execution of the cast may trigger an error or return a null pointer at runtime (depending on language)
  - **Type** — the resulting type is exp_1

Example Checks — Fields

- Cast — $exp_1.f$
  - **Check** — the type of exp_1 is a class C
  - **Check** — f is actually a field of C (or a superclass)
  - **Type** — based on looking up f in C

Example Checks — Methods

- Method Call — exp_b.m(e_1, e_2, …, e_n)
  - **Check** — the type of exp_b is a class C which has a method named m (inherited or directly)
  - **Check** — the method `m` has n parameters
  - **Check** — each argument e_i has a type that can be assigned to the associated parameter of method `m`
    - same as for assignment (upcast, coercion, etc.)
  - **Type** — type is given by declaration of method `m`

Example Checks — Return

- Method Call — `return` exp`;` or `return;`
  - **Check** — If the method this statement is in has a return type T, then the type of exp must be assignable to T.
  - **Check** — if the method has `void` return type, then there should not be a returned expression.
  - **Type** — statements don't have types

Overview

What makes a program “legal”?

The Checking Pass(es)

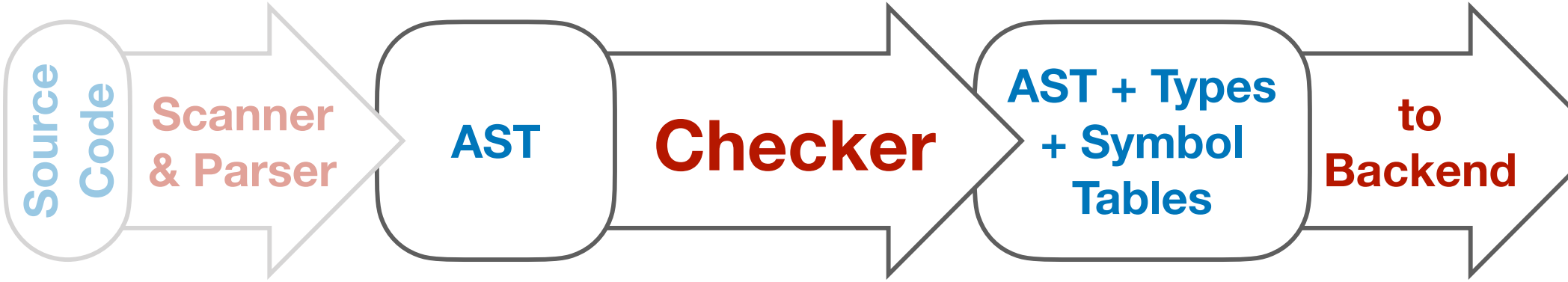
Symbol Tables & Names

Types

Relationships Between Types

Wrapup

Where we are in the Compiler



- Unlike the Scanner & Parser, the Checker usually does not **discard** anything from the AST
- What structure does the Checking pass **introduce**?
  - **Symbol Tables** — i.e. management of ***names***
  - **Type** Information
- Lastly, what information does the next pass need?

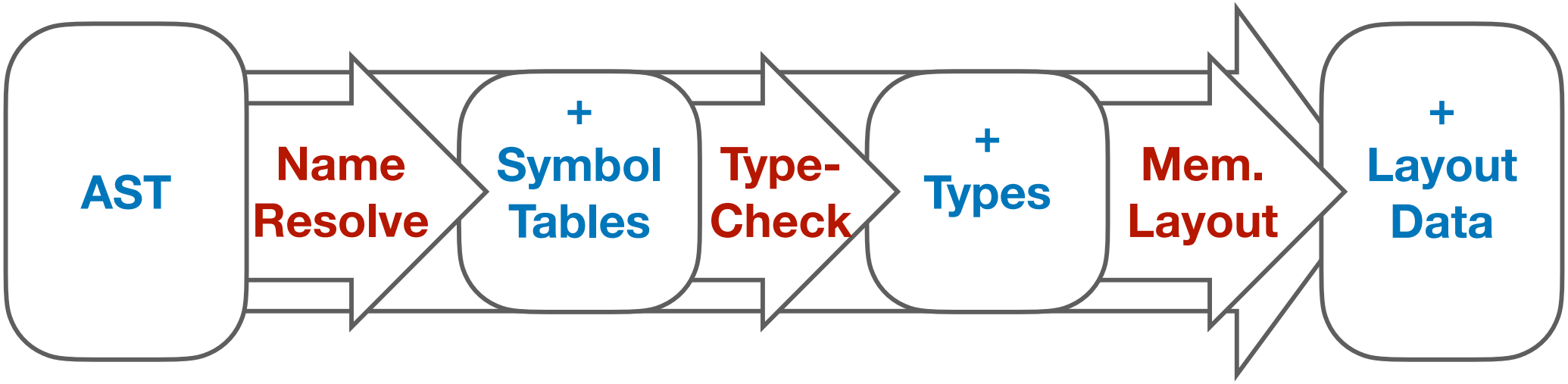
The Front-to-Back-end Jump

- When we get to the lecture on IRs, we will talk about what "**Lower-Level IR**" might be
  - For now, key point — the backend does not know what objects, classes, and other OO concepts are
- The pass that takes us **to the Backend** must **discard** the OO concepts. This will require making decisions about the **memory layout** of data structures & objects

Decisions About How to Compile Away OO Concepts

- Where are fields allocated in an object?
- How big are objects? (i.e. how many bytes need to be allocated when calling `new Thing(…)` ?)
- Where are local variables stored while a method is called?
- Which methods are associated with an object/class?
  - ✦ How do we determine which method to call based on the run-time type of an object?

Checking Pass(es) Proposal



- My recommendation (3-4 passes)
  - ✦ Passes 1 & 2 — Symbol & Name Resolution
  - ✦ Pass 3 — Typechecking
  - ✦ Pass 4 — Memory Layout
- Important! **WAIT** until the code-gen project to work on memory layout

Reporting an **ERROR** in a Compiler

- There are 2 main kinds of compiler error
  - *Blame the user* — the input program is not valid
  - *Blame the compiler* — the compiler itself is incorrect
- Validity errors can be further distinguished — e.g. scanner vs. parser vs. undefined variable vs. type checking errors
- After a program exits checking, any errors are not the user's fault any more!
  - **Input Validation** — a good system design principle!
- (Practical) Compiler projects should define distinct error reporting mechanisms for each type of error!

Overview

What makes a program “legal”?

The Checking Pass(es)

Symbol Tables & Names

Types

Relationships Between Types

Wrapup

Scopes / Namespaces

```
public class C {
  int x;
  public int run(int x) {
    return x(x) + y(x);
  }
  public int x(int y) {
    x = y;
    return 0;
  }
  public int y(int y) {
    return x + y;
  }
}
public class Main {
  public static void main(String[] args) {
    System.out.println((new C()).run(42));
  }
}
```

Q: Is this legal (Mini)Java code?

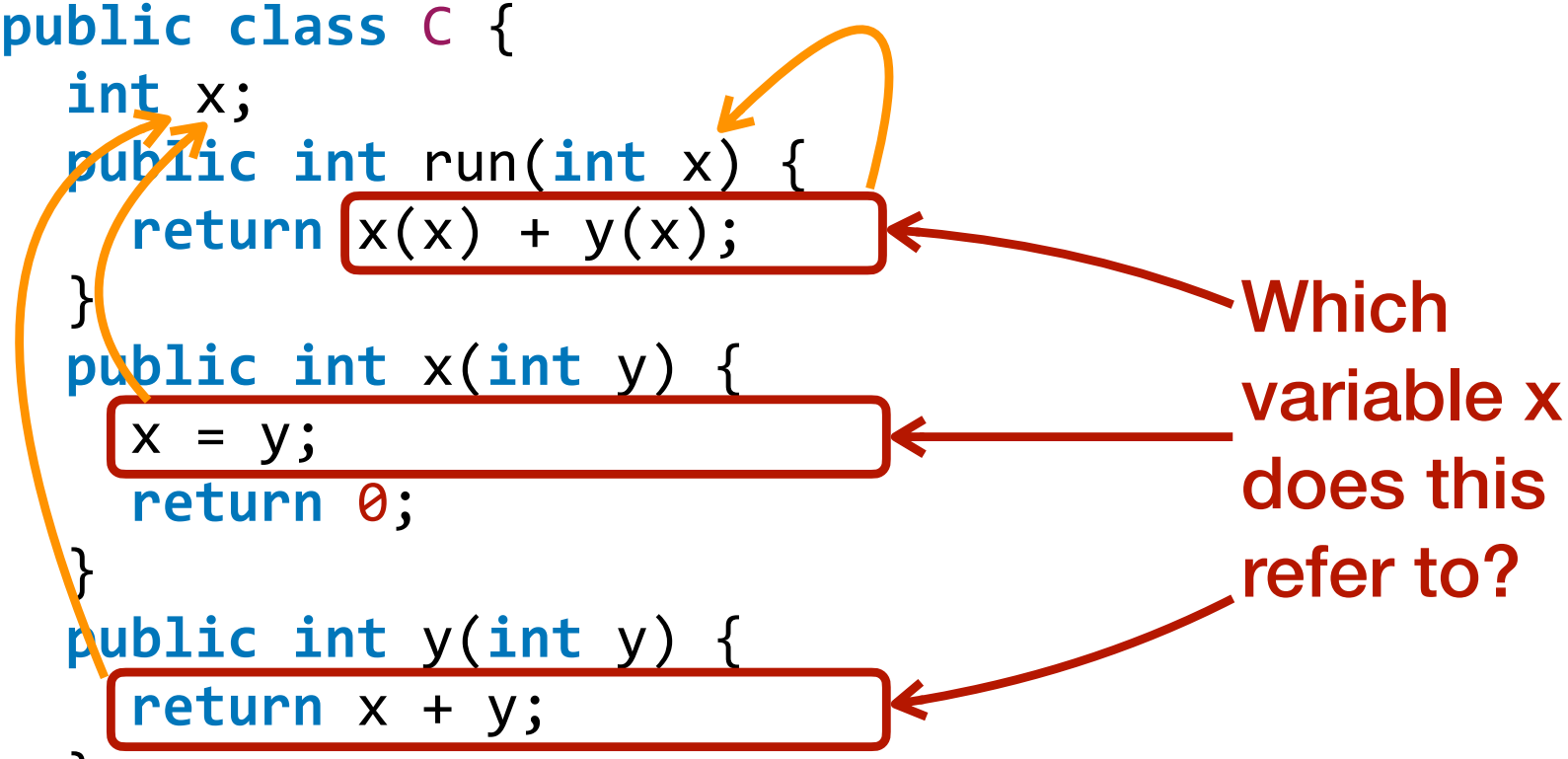
Scopes / Namespaces

```
public class C {
  int x;
  public int run(int x) {
    return x(x) + y(x);
  }
  public int x(int y) {
    x = y;
    return 0;
  }
  public int y(int y) {
    return x + y;
  }
}
public class Main {
  public static void main(String[] args) {
    System.out.println((new C()).run(42));
  }
}
```

Why is it legal for a function and variable to have the same name?

Scopes / Namespaces

```
public class C {
  int x;
  public int run(int x) {
    return x(x) + y(x);
  }
  public int x(int y) {
    x = y;
    return 0;
  }
  public int y(int y) {
    return x + y;
  }
}
public class Main {
  public static void main(String[] args) {
    System.out.println((new C()).run(42));
  }
}
```



Namespaces

- Many programming languages distinguish between names used in ***distinct grammatical contexts***.
  - ✦ e.g. in the expression `x(x)`, the first `x` must be a method name because we're calling it, and the second `x` must be a variable name because it's an argument expression
- This implies that **different namespaces use different symbol tables** (i.e. different data structures for resolving name lookups)

Scopes

- If we are looking up a variable name within a particular method, where do we look first?
- If we can't find a variable name declared as a *local variable* or *formal argument* to the method, then where should we look next?
- If we can't find a variable name declared as a field of the class we're in, where should we look next?
- Different scopes have different symbol tables (e.g. x can be both a local variable and class field) but **scopes are nested**. If we can't find the id we're looking for in one scope, then we keep *widening the scope* to find it.
  - ✦ *until we reach the outermost scope; then we're done*

Symbol Tables

- A map from ***identifiers*** to *other information*
  - identifier — a name (string) or symbol (object standing in for a string)
  - other information — e.g. types, memory layout data, pointers to other parts of the code, etc.
- Required Features
  - Lookup(id) — returns information or “no match”
  - Add(id, information) — create an entry in the table
  - Wider Scopes — if there is “no match” should we look elsewhere before giving up?

Implementing Symbol Tables

- In old-school compiler courses, this was a big topic
  - ✦ i.e. how to implement a hash table
- Today, use some combination of standard container data structures
  - ✦ In Java, HashMap (mainly) and ArrayList (as needed)
- Not just a HashMap!
  - ✦ e.g. symbol tables need to track where to redirect lookups to
  - ✦ on project — think about it What's a good design?

Name	Info
…	
…	

Visual shorthand for a Symbol Table

Representing Identifiers/Names

- Option 1 — leave as **strings**
  - **Pro** — simple; retains name from source
  - **Con** — ambiguous; need to keep resolving scope in all passes
  - **Con** — inefficient; string comparison is expensive
  - **Recommendation** 👎— definitely don't do in a real compiler

Representing Identifiers/Names

- Option 2 — Convert Strings to unique **numeric IDs**
  - Use a global table to maintain the **string ↔ ID-number** mapping; table has `newId(string name)` method
  - **Pro** — more efficient than strings (compare numbers)
  - **Con** — still ambiguous; requires repeating scope resolution
  - **Recommendation** — ok, but we can do better…

Representing Identifiers/Names

- Option 3 — Convert Strings to **(Symbol) Objects**
  - Have tables maintain the **string** ↔ **object** mapping instead. (Symbol) Objects store a unique ID at minimum; potentially other data as well
  - **Important!** — Designate ONE PASS as responsible for all string to symbol conversion (to prevent errors)
  - **Pro** — now unambiguous, regardless of scope; just as efficient (compare pointers)
  - **Con** — most involved solution (not a big con)
  - **Recommendation** — Do this on serious projects

MiniJava Design (Recommended)

- We'll outline a scheme that does what we need here, but you are not required to do things this way. Enjoy your *freedom* 🦅!
- We may cover a few more features here than are strictly needed for the MiniJava project — want to cover basic ideas too

MiniJava Design — Scopes

```
public class C {
  int x;
  public int run(int x) {
    return x(x) + y(x);
  }
  public int x(int y) {
    x = y;
    return 0;
  }
  public int y(int y) {
    return x + y;
  }
}
public class Main {
  public static void main(String[] args) {
    System.out.println((new C()).run(42));
  }
}
```

MiniJava Design — Scopes

```
public class C {
  int x;
  public int run(int x) {
    return x(x) + y(x);
  }
  public int x(int y) {
    x = y;
    return 0;
  }
  public int y(int y) {
    return x + y;
  }
}
public class Main {
  public static void main(String[] args) {
    System.out.println((new C()).run(42));
  }
}
```

Global Scope

Class Scope

Method Scope

MiniJava Design — **Namespaces**

```
public class C {
  int x;
  public int run(int x) {
    return x(x) + y(x);
  }
  public int x(int y) {
    x = y;
    return 0;
  }
  public int y(int y) {
    return x + y;
  }
}
public class Main {
  public static void main(String[] args) {
    System.out.println((new C()).run(42));
  }
}
```

Global Scope

Class Namespace

Class Scope

Field Namespace

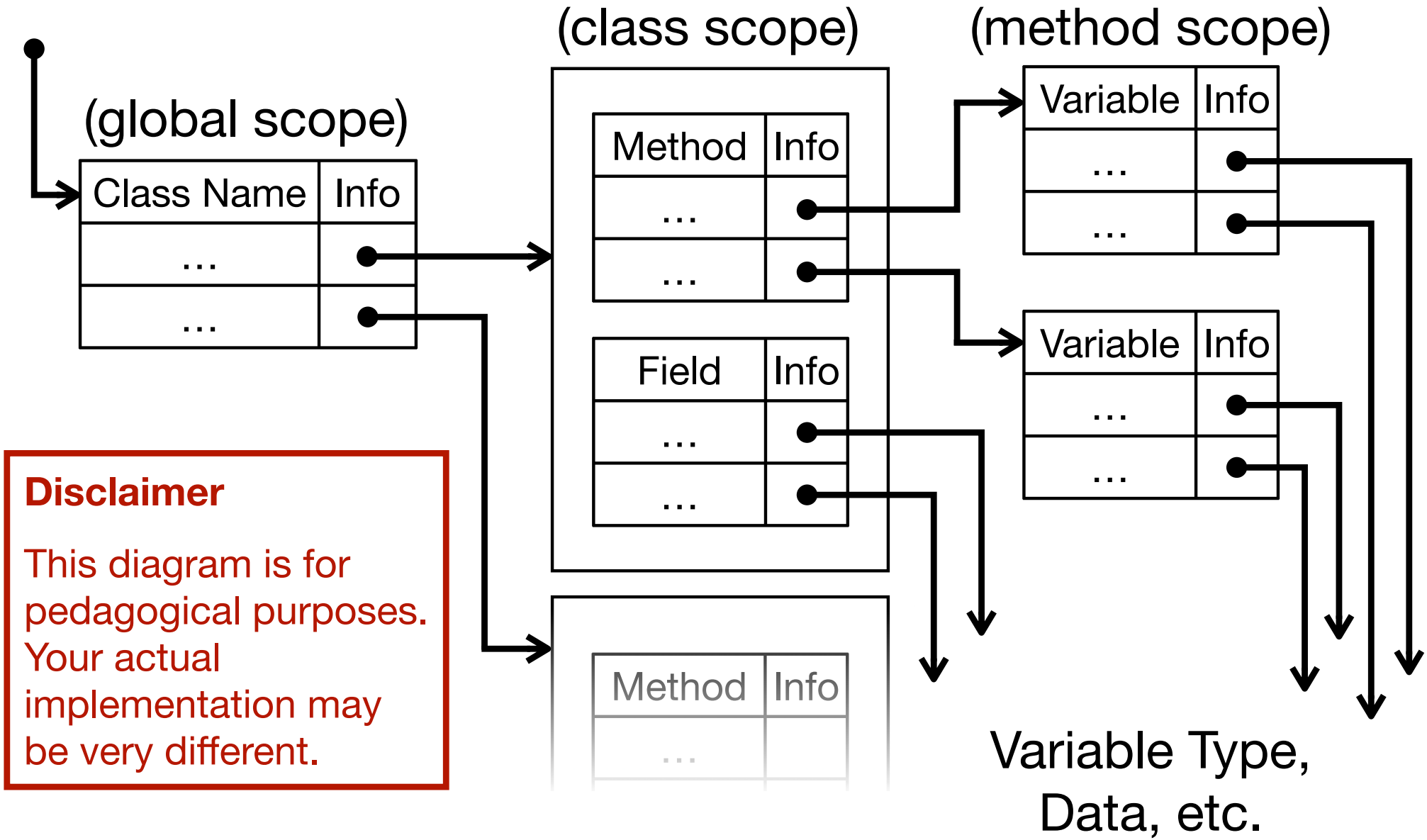
Method Namespace

Method Scope

Variable Namespace

MiniJava Design

Symbol Table Hierarchy

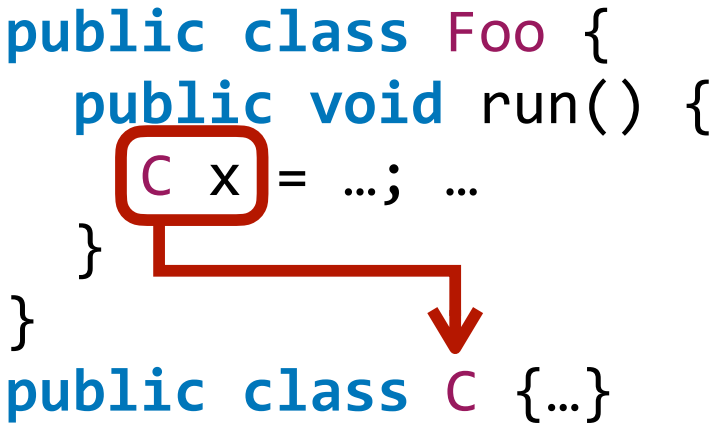


Disclaimer

This diagram is for pedagogical purposes. Your actual implementation may be very different.

MiniJava Design

Adding Fields to AST Nodes



- Suppose we look up variable x during a checking pass and find it is an object of type C (where C is a class)
- When we get to a later pass in the compiler, should we look up the type of x again?
- Unless we have somewhere to store the type of x we will have to look up the type repeatedly, which is error-prone
  - ✦ Design AST nodes to remember types, and possibly other references to symbol tables (& vice versa?)
  - ✦ see AST lecture for more — just mutate field for project

Recursion / Cyclic Symbol Tables

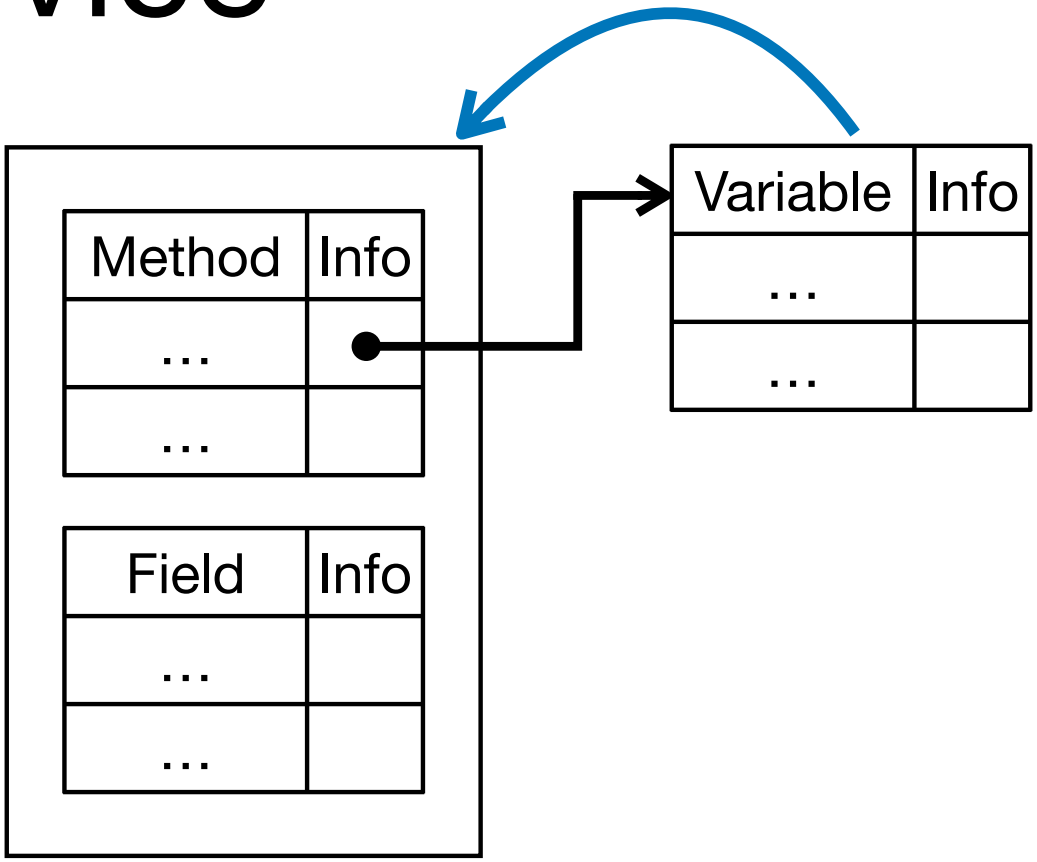
- Consider the class definition
  `public class X { X child; … }`
  - The class table for `X` has a field `child`, which has an object type `X`, which has a class table, which …
- How do we handle recursive definitions using structurally recursive compiler passes over an AST?
  - pass 1 — Create some objects / symbol tables, but leave some details missing (e.g. create global, class, and method symbol tables, but don't fill out types)
  - pass 2 — Fill in the rest of the details, using results of pass 1 to "tie the knot" on any cycles (e.g. now fill out types)

Names, Beyond MiniJava

- There are 2 hard problems in Computer Science
  - **Naming things**, Cache Invalidation, and off-by-one errors
- In real Java, we can declare variables inside nested scopes (if-then, while, etc.)
  - This requires deeper hierarchies of symbol tables reflecting the { block } structure of source code
- Java encapsulation features (static, private, public, protected, import, etc.) require more complex rules/ handling of name lookups

Engineering advice

- When managing any kind of nested scope, where you want to **redirect unsuccessful lookups** to an outer scope, it's a good idea to maintain an **explicit redirection pointer**



- If your scopes may become deeply nested, then you probably want to eventually optimize this redirection mechanism to ensure O(1) amortized lookup cost (see a compiler textbook for ideas)

Error Reporting Advice

- What do we do if a name lookup fails, and all redirected lookups fail? Report an error!
  - ✦ Our Goal — say “x undefined” only once! (why?)
- We can do this using ***undefined*** objects/types
  - ✦ When a lookup fails, report an error and add an ***undefined*** object or type as the “information” in that symbol table
  - ✦ How does this help?

Predefined Global Entries

- In full Java, some ideas are pre-defined / "built in" to the language. e.g. `Object` is a special class.
  - `Object` is not part of the Java grammar (not a keyword)
    e.g. `int Object = 32; foo(Object);` is legal Java
  - `Object` cannot be defined in a standard library file
    why not?
- Solution — pre-populate the global symbol table with a "built-in" class named `Object`.
  - Minimizes special case logic in the compiler (recall the Expression Problem), but we can still special case as needed — e.g. every class is a subclass of `Object`

Overview

What makes a program “legal”?

The Checking Pass(es)

Symbol Tables & Names

Types

Relationships Between Types

Wrapup

Why Types? (Naively)

- What is the meaning of x + y?
  - If x and y are integers, then x + y is their sum
  - If x and y are strings, then x + y is the concatenation
- "At the end of the day everything is just bits"
  - Does x + y mean add bits as 2s-complement integers?
  - Does x + y mean add bits as IEEE 754 doubles?
  - The two additions are different assembly instructions and very different circuits on your processor!
- ***meaning*** *— basic data types are necessary to say what* ***the meaning of the bits*** *you are computing on is!*

Why Types?

- **Safety** — types allow for error detection
- **Expressiveness** — overloaded methods and operators can make code nicer to write
  - e.g. `x + y` vs. `add_int(x,y)`
- **Optimization** — more information for the compiler
  - Allows the compiler to make useful assumptions about the program
  - e.g. qualifiers like const, final, restrict (in C)
  - note — qualifiers don't change the *meaning* of the data; they only change what is permissible to do with it

Strong vs. Weak vs. Static vs. Dynamic

Recommendation:

Not a well-defined or useful distinction; avoid

- Static vs. Dynamic Typing
  - **static** — means checks done at compile time
  - **dynamic** — means checks done at runtime
- Strong vs. Weak Typing
  - **strong** — exceptional / error behavior of the language is well specified
  - **weak** — meaning of the language is poorly specified
- Weak typing is a huge problem for legacy code bases (e.g. C/C++) — a major source of security vulnerabilities
- Weakly typed languages arose from a desire for performance at the "expense of safety"

Correcting Our Earlier Picture

Dynamic Typing

"Weak Typing"

"WeLL aCtuALLy"

Static Typing

Base vs. Constructed Types

- **base / primitive** types
  - e.g. int, double, char, boolean
  - introduction of many new floating point types has made this very complicated!
- **constructed / compound** types
  - built from other types, recursively (like grammars!)
  - This allows for **user-defined types**
  - e.g. records / structs / classes
  - e.g. arrays, pointers, function types, etc.

Encoding Types (in a Compiler)

- 100s-1000s of research papers on types all use a BNF variant to specify what types can be formed
  - ✦ This is the same idea of an "abstract grammar" we saw when discussing ASTs
- e.g.

$$
\begin{aligned}
type ::= & \texttt{int} \mid \texttt{boolean} \mid \ldots \\
& \mid \; Record(entry^*) \\
& \mid \; Ptr(type) \mid Array(type) \mid Func(formal^*, type) \\
entry ::= & \; (name, type) \\
formal ::= & \; (name, type)
\end{aligned}
$$

- Thus, we can use the same ideas (a class hierarchy) we learned for representing ASTs earlier

AST Nodes vs. Types

- (MiniJava) The MiniJava AST Nodes include types, in order to represent the output of the parser (as an AST)
  - ✦ You **MUST** define a **separate hierarchy** of type objects (called ADT = Abstract Data Type in previous quarters)
- Reason 1 — "terms" depend on "types" not vice-versa
  - ✦ Most languages allow statements and expressions to include/refer to types — e.g. `int x = 0;`
  - ✦ Most languages do not allow types to include/refer to expressions or statements — e.g. `Array<x + y>`
- Reason 2 — we will want to define helper methods on types (e.g. are two types equal? assignable? etc.)

Base Types

$$type ::= \texttt{int} \mid \texttt{boolean} \mid \ldots$$

- Implementation — use singleton classes, create a canonical object for each base type.
- Special cases you might want
  - **`void`** — defining a **`void`** type allows one to say that all methods/functions have a return type (potentially **`void`**)
  - **`error`**/unknown — the same idea as unknown objects for symbol tables; track where an error has already been reported
  - How is **`error`**/unknown different from **`void`**?

Struct / Record Types

$$type ::= \ldots \mid Record(entry^*) \mid \ldots$$
$$entry ::= (name, type)$$

- a struct or record type is built from a list of constituent types with names/labels for each entry. (e.g. C structs)
  - ✦ values are built from values for each entry sub-type
- e.g. {a=3, b=2.4, c=true} is a value of type Record((a, int), (b, double), (c, boolean))
- Structs/Records are like classes, but without any functions or inheritance
  - ✦ In Java, we just use classes instead

Method / Function Types

$$type ::= \ldots \mid Func(formal^*\ args,\ type\ return) \mid \ldots$$
$$formal ::= (name,\ type)$$

- The type of a function (aka. its ***signature***) is specified by giving an ordered list of its arguments (with their types) and a return type
- In (Mini)Java, we might simply use the associated method object, including its symbol tables and any other relevant data.

Class Types

$$type ::= \ldots \mid Class(entry^*,\ func\text{-}type^*,\ cls\text{-}type^?\ super) \mid \ldots$$
$$entry ::= (name, type)$$

- A class type is very similar to a struct type, except it also includes a list of function signatures and optionally a super-class.
  - ✦ In Java, there is always a superclass (except Object)
- For MiniJava, it is much easier to use a very simple ClassType with a single field, pointing to the symbol table/object representing the class in question.
  - ✦ Note — this is where the types can become ***recursive*** (see last lecture for how to handle)

Array Types

$$type ::= \ldots \mid Array(type) \mid \ldots$$

- Array types specify that there is an array of objects/values of the specified type
- Sometimes, a constant size for the array is known; usually not. In MiniJava, we don't know array length.
- What is Array(Array(int)) ?
- Some other languages support explicit multi-dimensional arrays; common in numeric computing (e.g. Fortran, Matlab, Numpy in Python, etc.)

Overview

What makes a program “legal”?

The Checking Pass(es)

Symbol Tables & Names

Types

Relationships Between Types

Wrapup

How are Types Related?

- Equivalence — two types are "the same"
- Subtyping
  - Liskov Substitution Principle — if A can always substitute for B, then A is a subtype of B
- Language specs can be more of a mess
  - "is the same as"
  - "is assignable to"
  - "is a subclass or equal to"
  - "is convertible to"

Type Equivalence

- Base types are equal if they're the same (e.g. int = int) and not equal otherwise
  - (if representing with singletons, comparison is trivial)
- Constructed Types
  - **Nominal Equivalence** — two types are the same only if they have literally the same name (requires the type constructor to supply a name, as in class C)
  - **Structural Equivalence** — two types are the same if they are the same constructor and all of their component/constituent types (recursively) are the same

Nominal vs. Structural Equivalence

- Type constructors that use **nominal equivalence**
  - Classes
- Type constructors that use **structural equivalence**
  - Records/Structs (sometimes)
  - Pointers
  - Arrays
  - Functions
- In OO languages, nominal equivalence of classes solves a nasty problem with recursive types (no time to cover)

Implementation of Type Equality

- **Nominal** — (assume classes)
  - $T_1 = T_2$? — If both T_1 and T_2 are class types, for the same class (i.e. has the same name) then they are equal. Otherwise they are not.
- **Structural**
  - $T_1 = T_2$? — If both T_1 and T_2 are the same type constructor and all constituent types are also equal, then they are equal. Otherwise they are not.
  - We can compare recursively, but then comparison is not guaranteed to be O(1) …

Efficient Structural Comparisons

- Instead of comparing recursively, we can make sure every type has a unique representation — then equality comparison becomes trivial
- Idea — instead of simply constructing types (from other types) store a table of already constructed types; if you can find an already constructed, structurally equivalent object, then return that existing object instead of creating a new one. (similar to how we use symbol tables)
- known as **hash-cons** or **memoization**
  - ✦ This is a very powerful and broadly applicable trick in compiler construction

Dynamic vs. Static Type of Objects

- Consider this code

```
class C {…}
class D extends C {…}
… { … C x = new D(); … } …
```

- What type does x have?
- What **static** type (aka compile time type) does x have?
- What **dynamic** type (aka runtime type) does x have?
- Does the checking pass of the compiler reason about **static** or **dynamic** types?

Casting and Coercing Types

- **Type Cast** — an explicitly written conversion of data from one type to another
- **Type Coercion** — an implicitly introduced conversion of data from one type to another
- Different languages have different rules for allowable **type conversions**, and whether a particular conversion can be a **coercion** or must be an explicit **cast**
  - MiniJava does not have type casts to simplify this all for you — but you can still assign objects of type A to variables of (static) type B, when A is a sub-class of B

Coercions and Typechecking

- If a language supports type coercions, it is the **responsibility of the type-checker** to **insert explicit type casts** into the AST wherever a coercion is required.
  - ✦ Why should we design things this way?
- Type checking actually does **discard** something. It discards ***type coercions***, which are implicit and thus don't appear as nodes or fields in the IR.
  - ✦ Thus, all passes after type checking have one less thing they need to worry about!
  - ✦ This is another important way the compiler pass architecture works — can make **assumptions**!

Type Conversions (Primitives)

- Base Types (usually number types)
  - ✦ What does it mean to convert between `int` and `unsigned int`? between `int` and `double`?
- Base type conversions may simply reinterpret bits
- Or they may imply actual computation (i.e. generate code)

Type Conversions (Refs/Pointers)

- In C, `(int*)foo` doesn't check what type `T*` foo is
- In Java
  - Casting to a super-type is allowed (and coercible)
  - What about casting to a sub-type?
  - **down-casting** introduces a runtime (dynamic) check
  - however, **down-casting** to a class that isn't a sub-type causes a compile time (static) error!
- In C++
  - Different keywords and features support all of the above

Useful Functions on Types

- Make sure all classes representing types in the compiler support a basic set of comparison methods
  - are the types equal?
  - is T_1 assignable to T_2 ?
  - …
- Why should we do this?
- "Single source of truth" — code that defines what "assignable" means is grouped in one place, reducing the opportunity for errors arising from inconsistency
- **Put this in your new MiniJava type package**

Overview

What makes a program “legal”?

The Checking Pass(es)

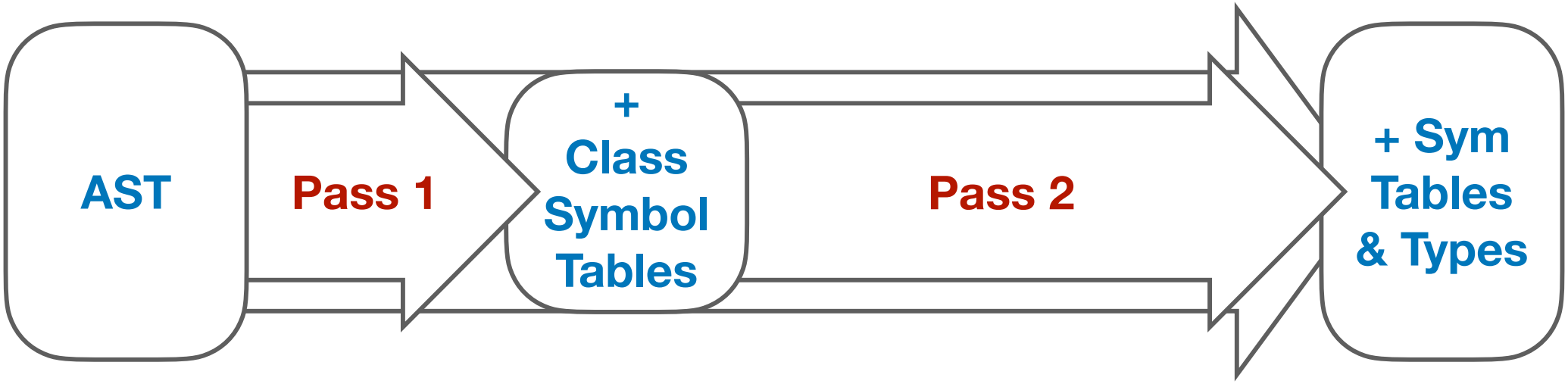
Symbol Tables & Names

Types

Relationships Between Types

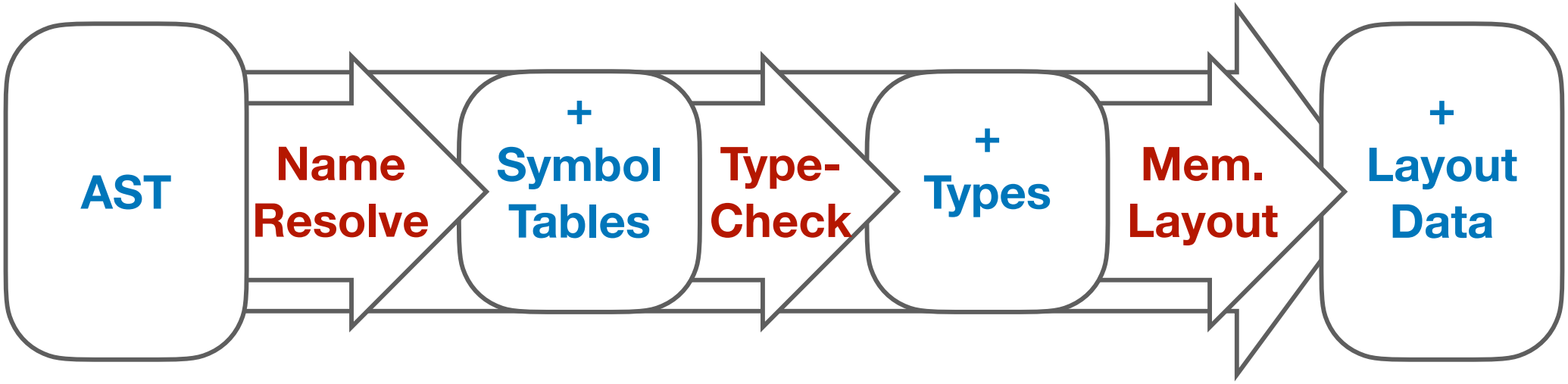
Wrapup

Checker Pass(es) — Simple



- One likely minimal design teams will use
  - ✦ pass 1 — do some of name resolution; at least classes
  - ✦ pass 2 — do rest of name resolution and all of type checking

Checker Pass(es) — Conceptual



- Pass 1 — First half of name resolution (at least classes)
- Pass 2 — Second half of name resolution
- Pass 3 — Typechecking
- Pass 4 — Memory Layout
- Important! **DON'T DO MEMORY LAYOUT NOW** wait until the code-gen project to work on memory layout

Disclaimer(s)

- This overview of checking, name, and types should give you a decent idea of what needs to be done in your project for the **checker** part of the project.
  - ✦ You'll need/want to adapt the ideas and advice here to fit what makes sense to you!
  - ✦ Enjoy your *freedom*  !
- You'll find good ideas in your compiler book too
- These slides also cover more than is needed for our specific project

Next Time…

- How should we start translating our code into x86?
- What does a typical compiler do with the backend?
  - (Friday) What IRs (Intermediate Representations) do compilers use in their back-ends?
  - (last 3 weeks of quarter) How does the backend of the compiler optimize code?
- What will our MiniJava compiler do?
  - (next 2 weeks) Review of x86 assembly & direct translation from the AST to x86 code