Lecture H:

ASTs, Visitors, & Structural Recursion

CSE401/501m:

Introduction to Compiler Construction

Instructor: Gilbert Bernstein

Administrivia

- HW2 (grammars and LR parsing) due Thursday night, 11:59 pm
- Next project part, parser + AST up soon; due a week from Thursday
  - More details in sections this week, but please start looking at the assignment as soon as I post it, and tinkering then
  - Probably good to finish HW2 first though
  - Then, get the CUP grammar working and cleaned up, before you add actions (the Java code that builds the AST; will cover today)

Outline

The Role of Abstract Syntax Trees

Implementing ASTs

Implementing Passes

The Visitor Pattern

Outline

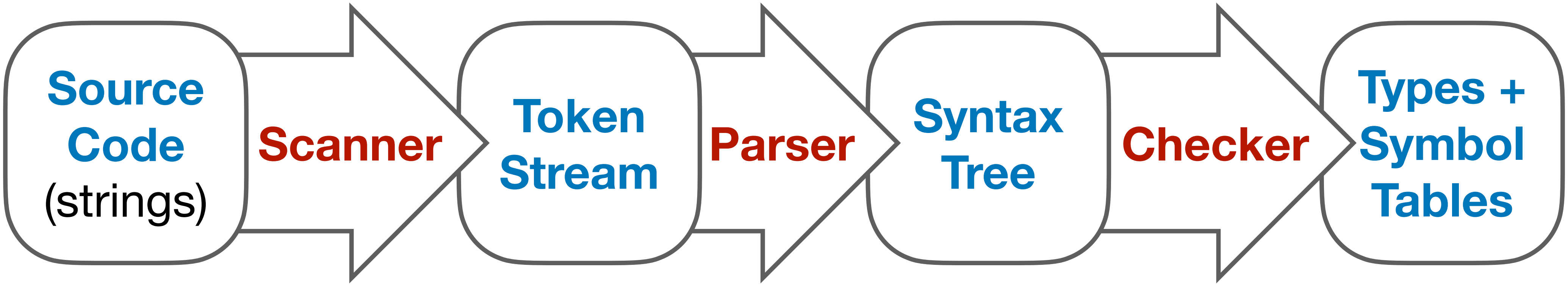
The Role of Abstract Syntax Trees

Implementing ASTs

Implementing Passes

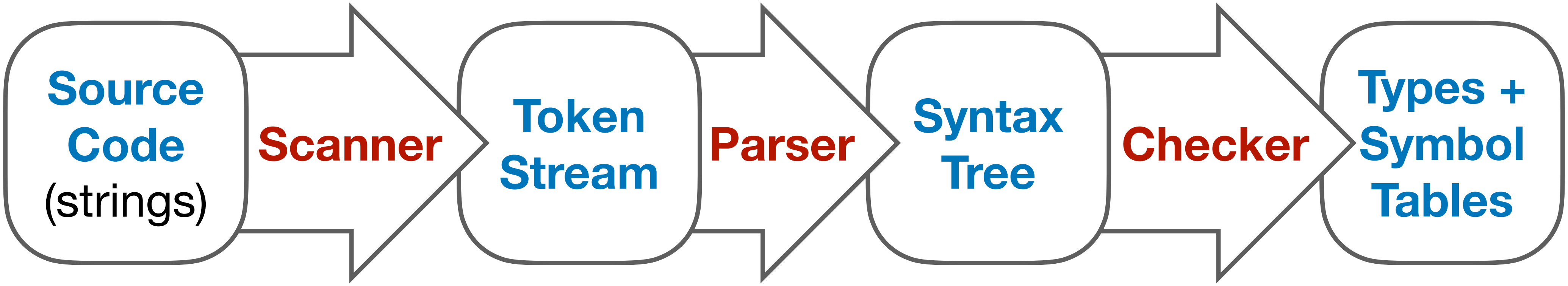
The Visitor Pattern

Our Basic Compiler Front-End



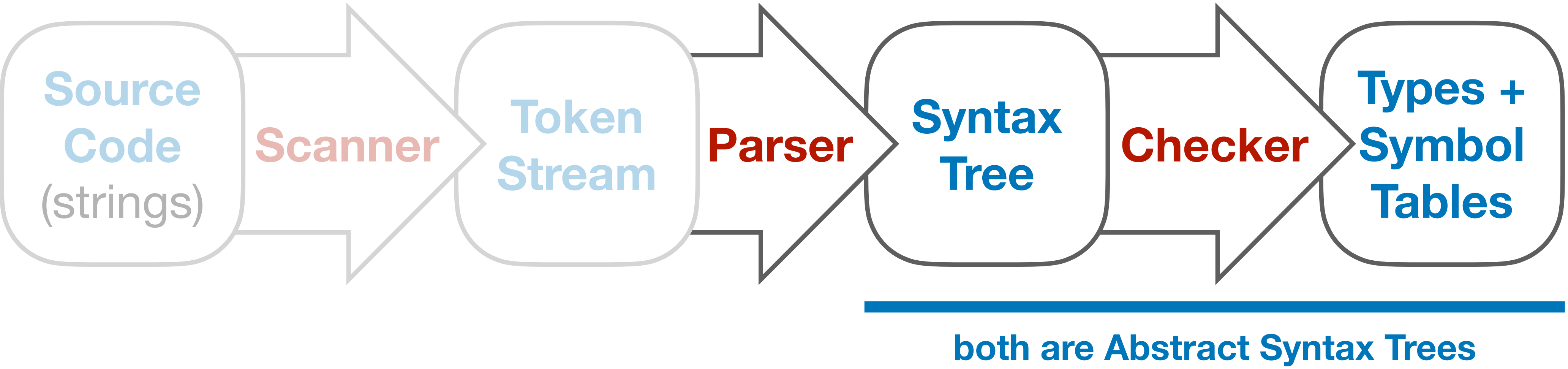
- The software architecture of compilers
  - A sequence of **Intermediate Representations**
  - with **Passes** converting or **lowering** between IRs
- Why? — Separation of concerns
  - Abstraction — Each IR **discards** unnecessary details, while **introducing** a little bit more structure
  - Each pass only worries about these differences

Abstraction in Compilers



- The Scanner
  - **introduces** — token structure on the sequence
  - **discards** — whitespace, comments
- The Parser
  - **introduces** — tree structure to program
  - **discards** — (via AST) delimiters, “syntactic markers”

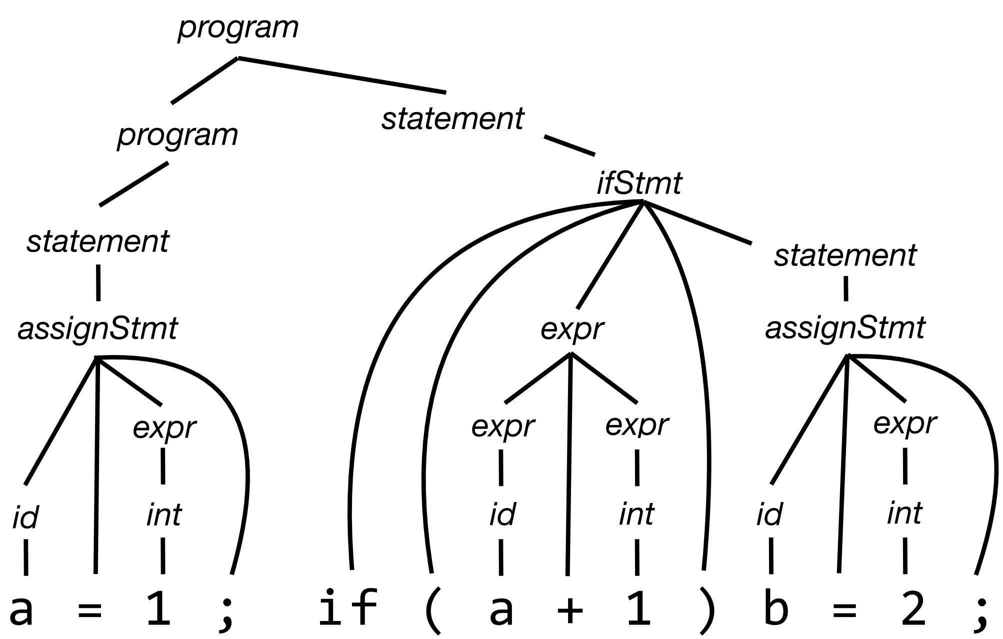
Abstract Syntax Trees (ASTs)



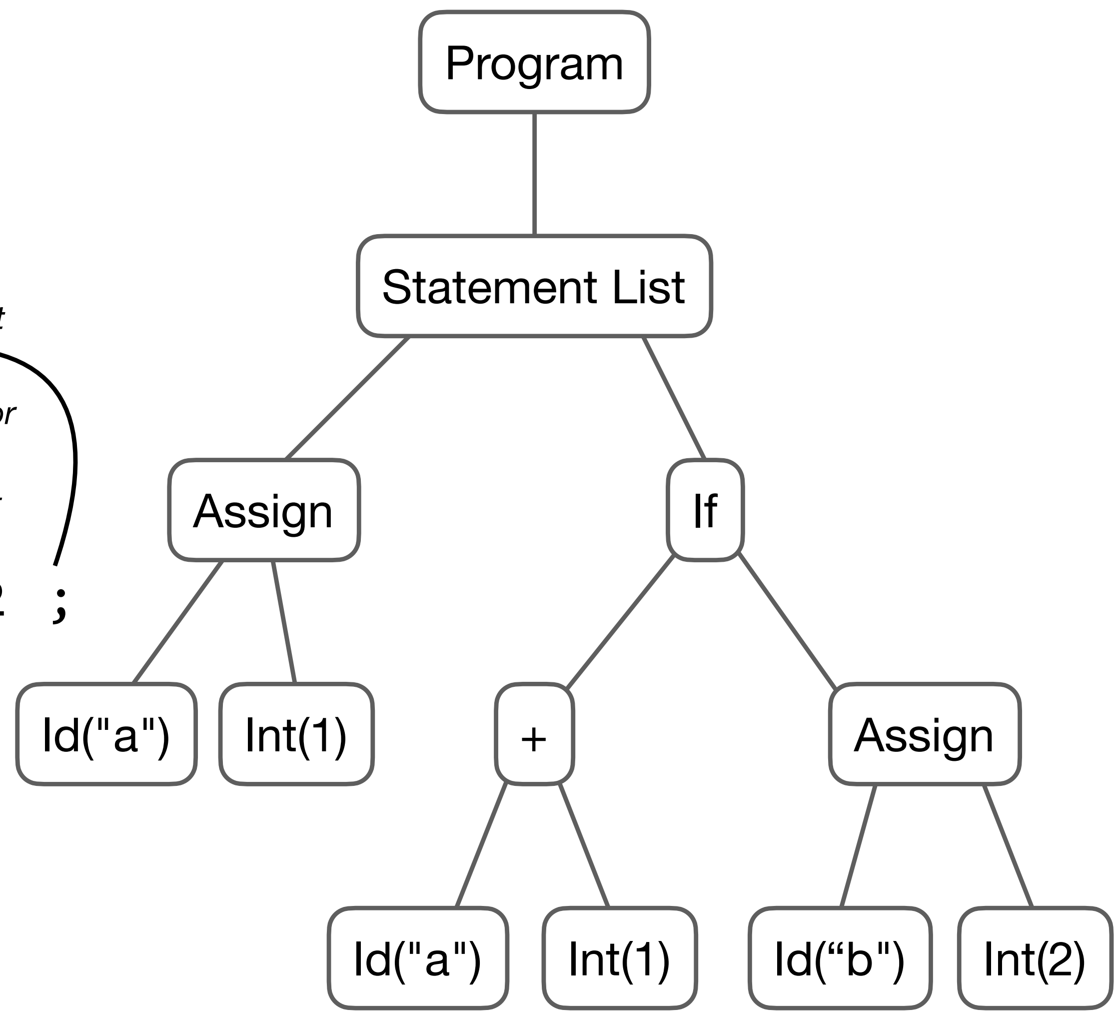
- The grammar describes concrete syntax trees
  - ✦ But we should ***abstract away*** useless details
- Abstract Syntax Trees (ASTs) are the primary IR used by compiler front-ends
  - The checker will both consume and produce ASTs
  - "When you leave ASTs, you've left the front-end"

Parse Tree vs. AST (ex. 1)

Full Parse Tree

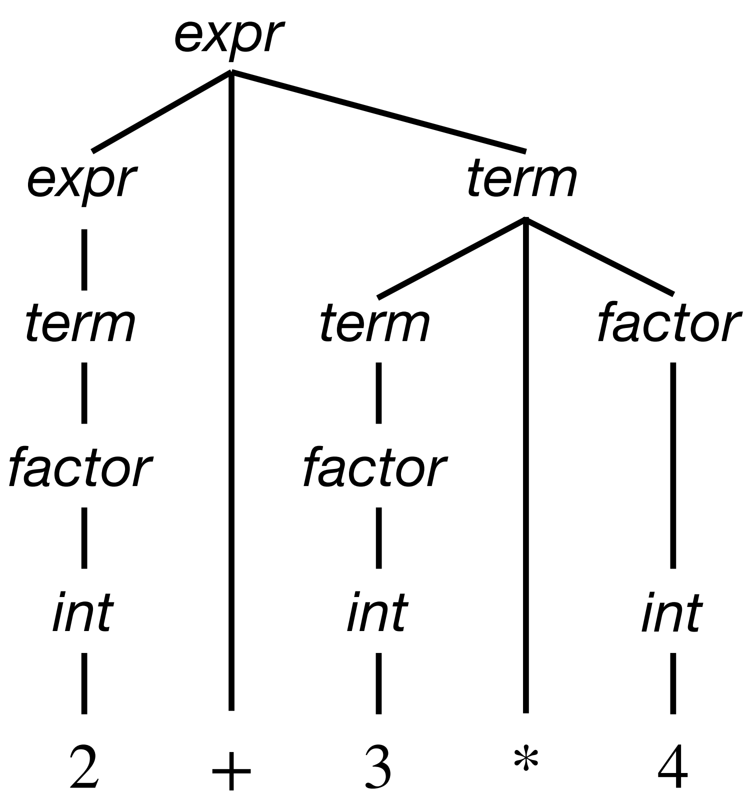


Abstract Syntax (AST)

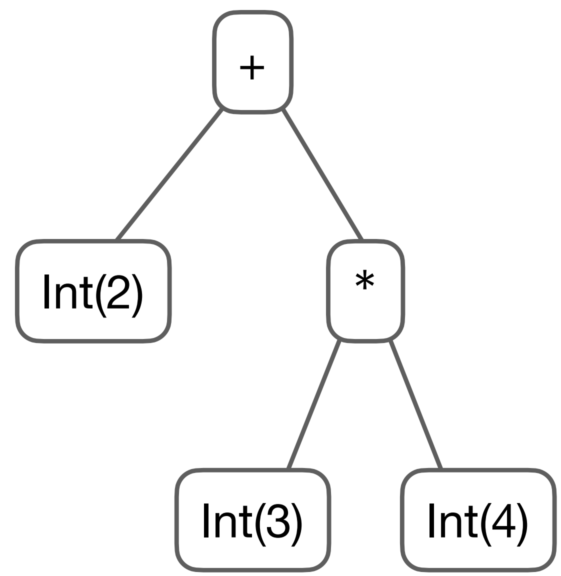


Parse Tree vs. AST (ex. 2)

Full Parse Tree



Abstract Syntax (AST)



Anticipating Checking

- The checking step will filter out programs that are grammatical but still invalid in some way
  - ✦ e.g.

```
public class Foo {
    public int bar(int y) { return x; }
}
```

- But checking will also analyze the program, producing useful information, e.g. types like “y is an integer”
- So, we will want to annotate or **decorate** our ASTs
  - ✦ Should we have different kinds of ASTs?
  - ✦ Should we support not-yet-complete ASTs?

Outline

The Role of Abstract Syntax Trees

Implementing ASTs

Implementing Passes

The Visitor Pattern

“Abstract” Grammars*

- Backus-Naur Form (BNF) was invented to specify the concrete syntax of programming languages
- However, many languages/compilers also use BNF to specify the abstract syntax (e.g. CPython, Haskell, WebAssembly)
  - ✦ BNF specification of abstract syntax is ***extremely*** common in research papers or for whiteboard/napkin design of new languages
- Ambiguity, LR, LL, etc. are all *not important* for “abstract grammars.” The purpose is a shorthand for describing an IR (i.e. data structures) not specifying concrete syntax!

***I don’t know of any standard name for this practice, but it is very common**

Example from First Lecture

Classes for Simplified Calculator

```
abstract public class Expr {}

public class Num extends Expr {
    public int i;
    public Num(int v) { i = v; }
}

public class Add extends Expr {
    public Expr e0, e1;

    public Add(Expr a0, Expr a1) {
        e0 = a0;
        e1 = a1;
    }
}
```

"Abstract" BNF

$$
\begin{aligned}
Expr ::= &\ Num\ int \\
\mid &\ Add\ Expr\ Expr
\end{aligned}
$$

"Abstract" BNF (w/names)

$$
\begin{aligned}
Expr ::= &\ Num(int\ i) \\
\mid &\ Add(Expr\ e0, Expr\ e1)
\end{aligned}
$$

***Note: this slide is meant to help you understand the ideas; you will not be tested on it**

Expanded Example

Classes for Simplified Calculator

```
abstract public class Expr {}

public class Num extends Expr {
    public int i;
    public Num(int v) { i = v; }
}

public class Var extends Expr {
    public String s;
    public Var(String v) { s = v; }
}

public class Add extends Expr {
    public Expr e0, e1;

    public Add(Expr a0, Expr a1) {
        e0 = a0;
        e1 = a1;
    }
}

public class Mul extends Expr {
    public Expr e0, e1;

    public Mul(Expr a0, Expr a1) {
        e0 = a0;
        e1 = a1;
    }
}

abstract public class Stmt {}

public class Assign extends Stmt {
    public String lname;
    public Expr rhs;

    public Assign(String nm, Expr r) {
        lname = nm;
        rhs = r;
    }
}
```

```
public class StmtList {
    public List<Stmt> stmts;
    public StmtList(List<Stmt> xs)
    { stmts = xs; }
}

public class IfStmt extends Stmt {
    public Expr cond;
    public StmtList tbody, fbody;

    public IfStmt(Expr c,
                  StmtList t,
                  StmtList f) {
        cond = c;
        tcase = t;
        fcase = f;
    }
}

public class WhileStmt extends Stmt {
    public Expr cond;
    public StmtList body;

    public WhileStmt(Expr c,
                     StmtList b) {
        cond = c;
        body = b;
    }
}
```

“Abstract” BNF

$$
\begin{aligned}
Expr ::= \; & Num(int\ i) \\
\mid \; & Var(String\ s) \\
\mid \; & Add(Expr\ e0, Expr\ e1) \\
\mid \; & Mul(Expr\ e0, Expr\ e1)
\end{aligned}
$$

$$StmtList ::= Stmt^*$$

$$
\begin{aligned}
Stmt ::= \; & Assign(String\ nm, Expr\ rhs) \\
\mid \; & If(Expr\ cond, StmtList\ tbody, \\
& \qquad StmtList\ fbody) \\
\mid \; & While(Expr\ cond, \\
& \qquad StmtList\ body)
\end{aligned}
$$

***Note: this slide is meant to help you understand the ideas; you will not be tested on it**

Translating to Classes (1)

- For each “abstract” non-terminal, create a base class e.g.

```
abstract public class Expr {}
```

- For each “abstract” production, create a sub-class e.g.

```
public class Add extends Expr { … }
```

Translating to Classes (2)

- Simple Refinements on naive strategy
  - Use native lists in the implementation language

```
public class StmtList {
    public List<Stmt> stmts; …
```

  - Derive all nodes from a common ancestor, which keeps track of universal data (e.g. location in original source)

```
abstract public class ASTNode {}
public class Expr extends ASTNode { … }
```

- No need to follow this idea *exactly.* However, systems (including compilers) are easier to build and maintain if they consistently follow simple, boring rules

Decorating ASTs (1)

- Approach 1 — Add fields to AST Nodes (Mega IRs)
  - e.g. change $Var(String\ s)$ to $Var(String\ s, Type\ t)$
  - Pros — Conceptually Simple; smallest change to code; efficient
  - Con — Exposed to all passes using this AST IR;
  - Big Con — Mutating IRs makes code harder to reason about; e.g. which fields are defined or used at different points in the compiler?
  - **primary approach suggested for your project**

Decorating ASTs (2)

- Approach 2 — Create a New IR with extra fields
  - ✦ Same as approach 1, but a separate IR
  - ✦ Pros — IR is **immutable** after being constructed; allows us to ***discard*** fields not used in later passes
  - ✦ Cons — Usually leads to highly duplicative data structure definitions (opportunity for bugs); can lead to compiler inefficiencies (lots of small memory allocations)
  - ✦ Used when implementing compilers with functional languages; also common in teaching & research; production compilers written in imperative languages avoid this approach

Decorating ASTs (3)

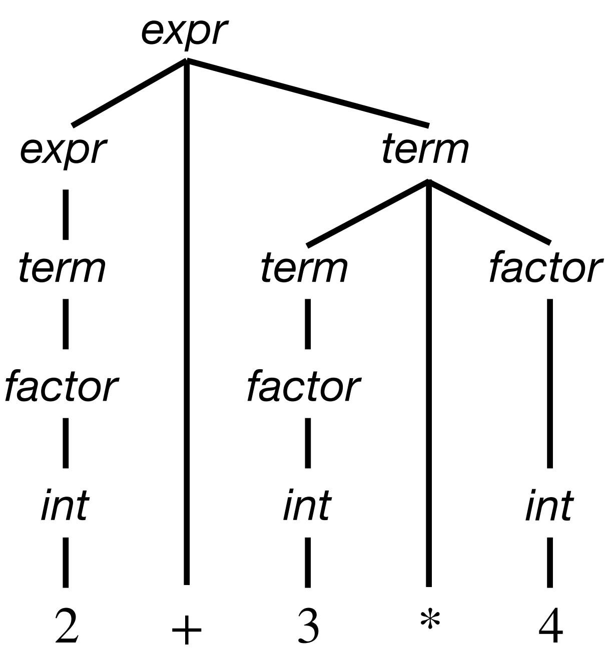
- Approach 3 — Create a map from nodes to data
  - e.g. define an auxiliary `HashMap<Expr, Type> types;`
  - Pros — Does not require redefining a new IR; leaves the base AST immutable; still quite efficient
  - Cons — Requires managing and passing around auxiliary mapping tables; less convenient if *many* passes will use the data (e.g. types are broadly used)
  - This approach can be highly effective if the decorations are **transient** — e.g. one pass **computes** decorations, another pass consumes, and then they are **discarded**

Building ASTs

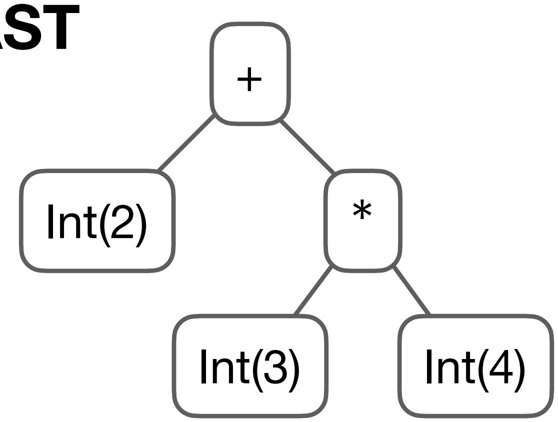
- Underlying idea — define a correspondence between the *concrete parse tree* and the *abstract syntax tree*
  - More specifically, define how to construct the abstract syntax tree *using structural recursion* on the parse tree
- Realization of idea — the parser implicitly traverses the concrete syntax tree in a post-fix traversal
  - At each production $A ::= X_1 \cdots X_n$, a **parser action** takes **intermediate values** for $X_1 \cdots X_n$ as input and produces an **intermediate value** for A
  - for us, **intermediate values** are AST nodes
- More in section and in the Parser project

Example: Building AST

Parse Tree



AST



Grammar	Actions
$expr ::= expr + term$	`{ return new Add(lhs, rhs); }`
$\mid\ expr - term$	`{ return new Sub(lhs, rhs); }`
$\mid\ term$	`{ return arg; }`
$term ::= term * factor$	`{ return new Mul(lhs, rhs); }`
$\mid\ term\ /\ factor$	`{ return new Div(lhs, rhs); }`
$\mid\ factor$	`{ return arg; }`
$factor ::= int$	`{ return arg; }`
$\mid\ (expr)$	`{ return arg; }`
$int ::= 0 \mid 1 \mid 2 \mid 3 \mid 4$	`{ return new Int(0); }`
$\mid\ 5 \mid 6 \mid 7 \mid 8 \mid 9$	…

Outline

The Role of Abstract Syntax Trees

Implementing ASTs

Implementing Passes

The Visitor Pattern

Operations on ASTs

- Many different **passes** are defined on ASTs (and other IRs)
  - Print a readable dump of the tree data structure
  - Print a parseable (source-code) version of the tree; aka. "pretty-printing" (the reverse of scanning & parsing)
  - Perform checking, annotate types, & report errors
  - (less common) optimize the AST code directly
  - Generate another IR from the AST
  - Generate assembly code from the AST directly
  - …

Obj-Oriented Approach (1)

- "Good" object-oriented style says we should define an interface for AST Nodes that requires each node to implement each pass

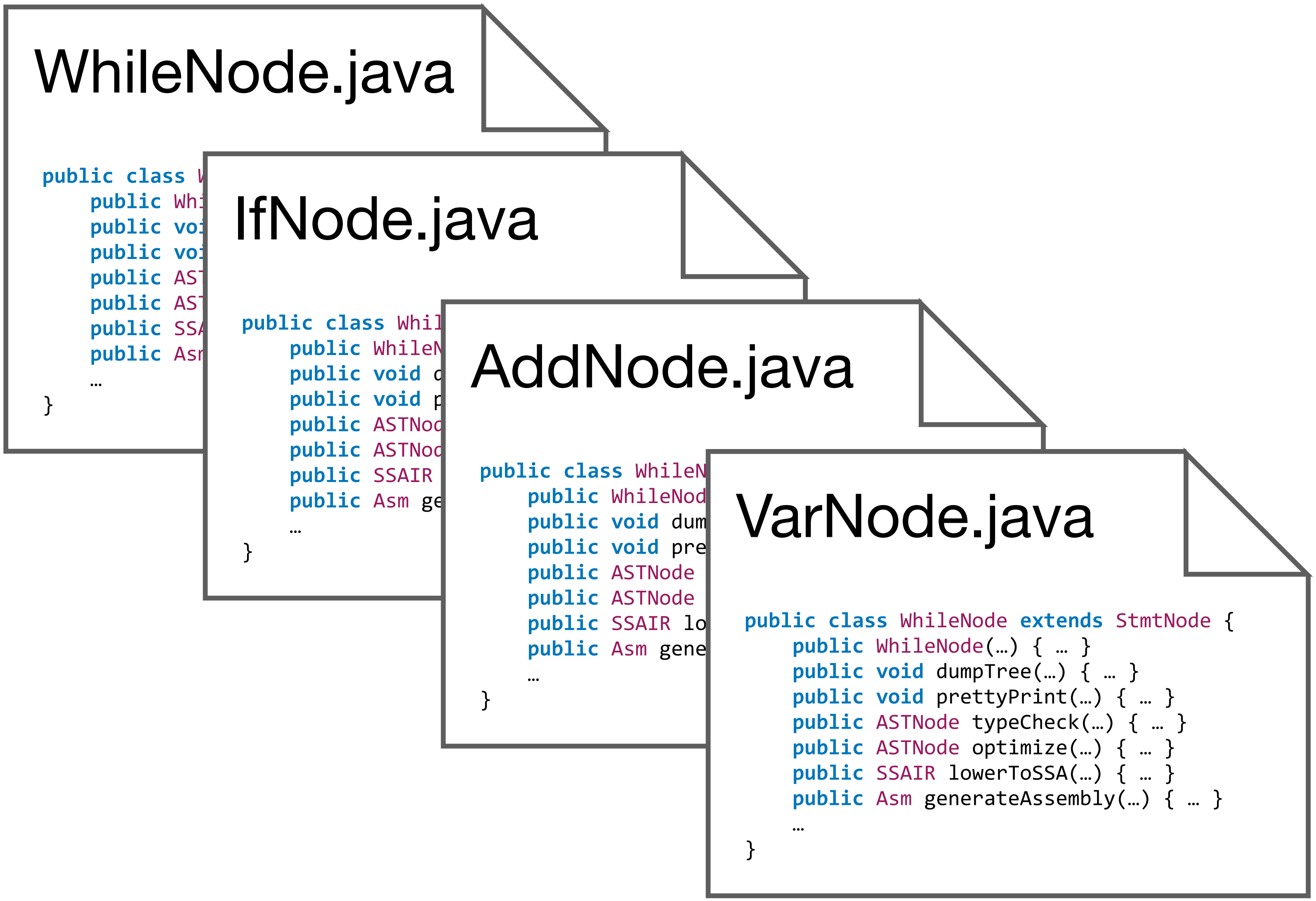
```
abstract public class ASTNode {
  abstract public void dumpTree(…) { … }
  abstract public void prettyPrint(…) { … }
  abstract public ASTNode typeCheck(…) { … }
  abstract public ASTNode optimize(…) { … }
  abstract public SSAIR lowerToSSA(…) { … }
  abstract public Asm generateAssembly(…) {…}
  …
}
```

Obj-Oriented Approach (2)

- Then each kind of AST Node implements this interface. The code for each (NodeClass, Pass) pair is located with the NodeClass

```
public class WhileNode extends StmtNode {
  public WhileNode(…) { … }
  public void dumpTree(…) { … }
  public void prettyPrint(…) { … }
  public ASTNode typeCheck(…) { … }
  public ASTNode optimize(…) { … }
  public SSAIR lowerToSSA(…) { … }
  public Asm generateAssembly(…) { … }
  …
}
```

The Whole System Viewpoint



■ ■ ■

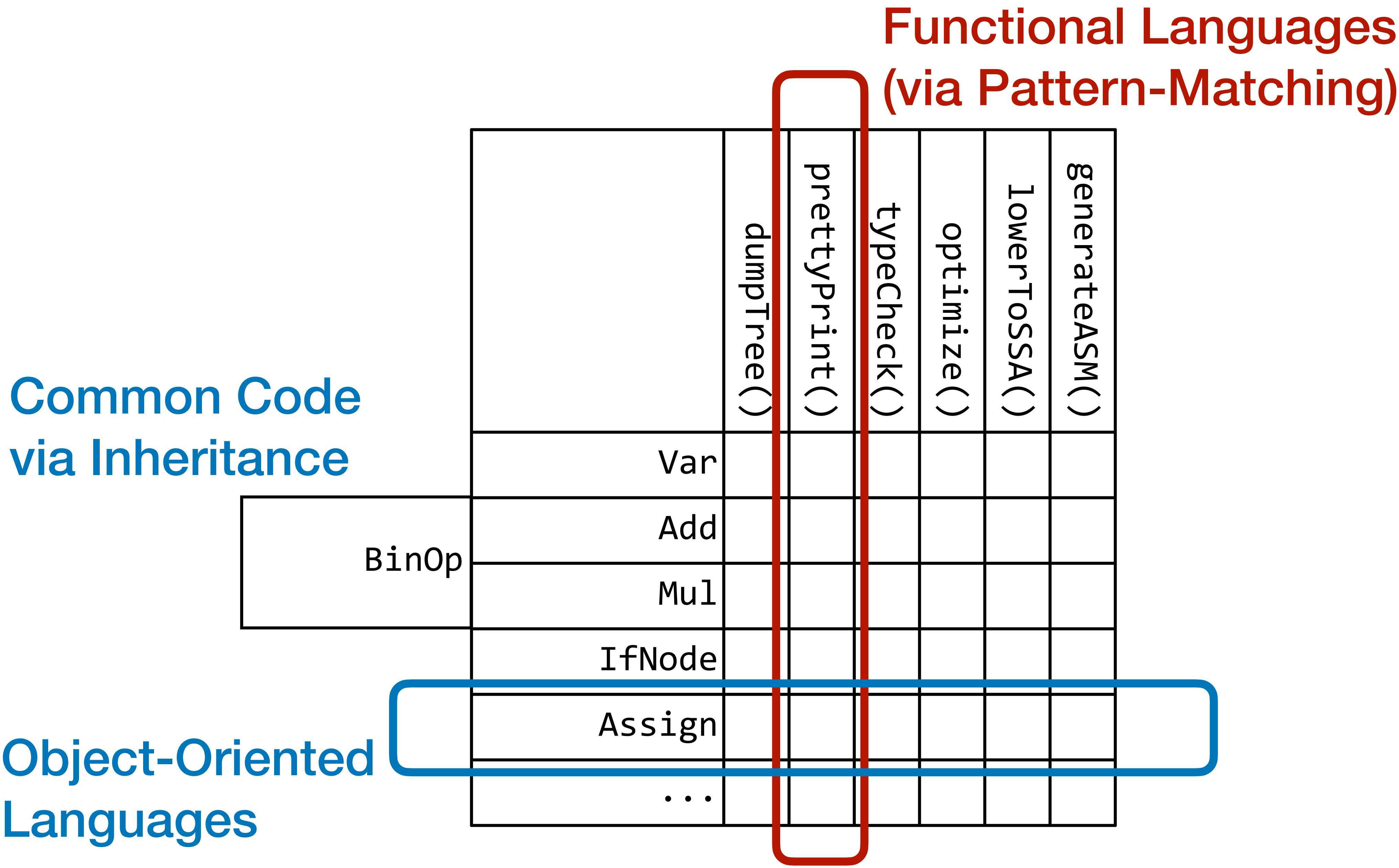
System Design Concepts

- Review of Terms
  - **Locality** (Lexical) — Which parts of the system are close to each other in the source code?
  - **Encapsulation** — Which parts of the system are allowed (via type system) to access which other parts?
  - **Extensibility** & Maintenance — What is required to add X to the system or modify X?

Consequences of This Design

- The object-oriented design enforces **lexical locality** (i.e. which class in which file) according to the AST Nodes, in order to **encapsulate** private data fields
  - but we make data fields public on AST Nodes, so this isn’t necessary
- The object-oriented design makes it easy to **extend** the system with new AST Nodes, or **modify** single existing AST nodes, because of this **lexical** grouping.
- The object-oriented design makes it possible to **factor out** common code by using deeper inheritance hierarchies; e.g. ASTNode → Exp → BinaryOp → Add

The Expression Problem*



* *Originally formulated by Philip Wadler in 1998*

Expression Problem: Tradeoffs

- If code is **grouped lexically by data**, then it is easier to add, modify, and share across different **Classes in the IR**
  - This is the right choice if the set of operations changes less frequently in your system
- If code is **grouped lexically by function**, then it is easier to add, modify, and share across different **Passes**
  - This is the right choice if the set of AST/IR Classes changes less frequently in your system
- **Key conclusion** — The second option is usually the right choice for compilers. Why? …

Many Passes

Compilers have **a lot** of different passes on IRs, and the abstract syntax for a given language doesn't change often

```
targetlibinfo
tti
no-aa
tbaa
scoped-noalias
assumption-
  cache-tracker
basicaa
ipsccp
globalopt
deadargelim
domtree
instcombine
simplifycfg
basiccg
prune-eh
inline-cost
inline
functionattrs
domtree
sroa
early-cse
lazy-value-info
jump-threading
tailcallelim
simplifycfg
reassociate
domtree
loops
loop-simplify
lcssa
loop-rotate
licm
loop-unswitch
instcombine
scalar-evolution
loop-simplify
lcssa
indvars
loop-idiom
loop-deletion
loop-unroll
mldst-motion
domtree
memdep
gvn
memdep
memcpyopt
jump-threading
correlated-
propagation
domtree
memdep
dse
loops
loop-simplify
lcssa
licm
adce
simplifycfg
domtree
instcombine
barrier
float2int
domtree
loops
loop-simplify
lcssa
loop-rotate
branch-prob
block-freq
scalar-evolution
simplifycfg
domtree
instcombine
loops
loop-simplify
lcssa
scalar-evolution
loop-unroll
instcombine
loop-simplify
lcssa
licm
scalar-evolution
alignment-from-
  assumptions
strip-dead-
  prototypes
elim-avail-
  extern
globaldce
constmerge
verify
```

LLVM Optimization Passes using -O2 (first lecture)

The Expression Problem*

How do we achieve this in an Object-Oriented Language?

Functional Languages (via Pattern-Matching)

	dumpTree()	prettyPrint()	typeCheck()	optimize()	lowerToSSA()	generateASM()
Var						
Add						
Mul						
IfNode						
Assign						
...						

Object-Oriented Languages

** Originally formulated by Philip Wadler in 1998*

Simple Structural Recursion (1)

```
public class … {

  public void foo(ASTNode n) {
    if (n instanceof Var) {…}
    else if (n instanceof Add) {…}
    else if (n instanceof Mul) {…}
    else if (n instanceof IfNode) {…}
    else if (n instanceof Assign) {…}
    …
  }
}
```

	dumpTree()	prettyPrint()	typeCheck()	optimize()	lowerToSSA()	generateASM()
Var						
Add						
Mul						
IfNode						
Assign						
...						

- Pro — Simple and Direct
- Con — “missing case” bugs
- Con — no support for “defaults” (needs all cases)

Simple Structural Recursion (2)

- Different traversals can be implemented via recursive calls on sub-trees

```
public class … {
  public void foo(ASTNode n) {
    …
    else if (n instanceof Mul) {
      // do something - pre-order
      foo(n.e0);
      foo(n.e1);
      // do something - post-order
    }
    …
```

- A possible variation — instead of `void`, have the call return something (e.g. a new, modified `ASTNode`)

Pass in a Class

- Where should the structurally recursive function be located?
  - globally? (not in Java!)
  - on the AST Nodes? Which one?
- No! No! This will not do!
- Each operation / pass gets it's own class
  - We create **exactly one** instance of such a "pass class"
  - Helper functions can be **private** to the class
  - *Pass-local* data can be defined as fields on the class — will only exist while the pass is being executed

Image: Cat in the Hat by Dr. Seuss

Pass-Local Data

- Suppose you're trying to print out the AST, and you want to control *indentation*
  - ✦ How are you supposed to know how many tabs deep you are when traversing a given AST node?
- Approach 1 — change the function signature, e.g.

```
public void print(ASTNode n, int numTabs) {…}
```

  - ✦ This is verbose; every recursive call must supply it
- Approach 2 — store as ***pass-local*** data, and mutate it

```
public class PrintPass {
    private int numTabs;
    public void print(ASTNode n) {…}
```

  - ✦ Much less verbose; but you need to manage it correctly

Outline

The Role of Abstract Syntax Trees

Implementing ASTs

Implementing Passes

The Visitor Pattern

My Teacher Told Me..

- …to not use `instanceof` in Java!
- In Java, you're supposed to use **method dispatch** and **inheritance** to let the language control which code gets run

```
public class … {

  public void foo(ASTNode n) {
    if (n instanceof Var) {…}
    else if (n instanceof Add) {…}
    else if (n instanceof Mul) {…}
    else if (n instanceof IfNode) {…}
    else if (n instanceof Assign) {…}
    …
  }
}
```

- But if we do that, we get locality according to AST node, instead of locality according to pass
- The **visitor pattern** is an enhanced version of **pass-in-a-class** that's designed to avoid `instanceof`
- The **visitor pattern** will also help us **factor out code** that's common to multiple passes using inheritance

Double Dispatch

- We need to dispatch on two different *classes*
  - Which **AST Node**, and which **Pass**?
- In "Simple Structural Recursion" the code is located in the pass class, so there's not a good way to use overloading to dispatch on AST Nodes
- In the visitor pattern, each function call turns into two function calls
  - We ask an **AST Node** to **accept** a **Pass** (aka **Visitor**)
  - We ask that specific **Pass** to **visit** the **AST Node**

Double Dispatch Visually

Each **Visitor** defines `visit(Var v)`, `visit(Add v)`, ... methods, one for each AST Node

Each **AST Node** defines a generic `accept(Visitor v)` method, which is only responsible for calling the visitor

	DumpTree	PrettyPrint	TypeCheck	Optimize	LowerToSSA	GenerateASM
Var		●				
Add		●				
Mul		●				
IfNode		●				
Assign		●				
...		●				

An Idea Whose Time Never Came

- OO Languages implement *dynamic* single dispatch
  - i.e. when you call `obj.method(arg1, arg2)`, we decide which implementation of `method` to call by ***dispatching*** on the runtime (dynamic) type of `obj`
- Dynamic Double (or multiple) Dispatch
  - When you call `obj.method(arg1, arg2)`, we decide which implementation of method to call by ***dispatching*** on the runtime (dynamic) type of `obj` ***AND*** on the dyn. type of `arg1` (***AND*** on the dyn. type of `arg2` if multiple dispatch)
- Topic of research at UW(!) in the 1990s
  - Was very complicated and not that useful

Visitor Pattern — Calculator

```
interface Visitor {
  public void visit(Num n);
  public void visit(Add n);
}

abstract public class Expr {
  abstract void accept(Visitor v);
}

public class Num extends Expr { …
  public void accept(Visitor v) {
    v.visit(this);
  }
}

public class Add extends Expr { …
  public void accept(Visitor v) {
    v.visit(this);
  }
}
```

Generic Code for all Passes

```
public class PrintPass implements Visitor {
  public void visit(Num n) {
    System.out.print(n.i);
  }
  public void visit(Add n) {
    System.out.print("(");
    n.e0.accept(this);
    System.out.print("+");
    n.e1.accept(this);
    System.out.print(")");
  }
}
```

Code for a Specific Pass

Visitor Pattern — Calculator

```
interface Visitor {
  public void visit(Num n);
  public void visit(Add n);
}

abstract public class Expr {
  abstract void accept(Visitor v);
}

public class Num extends Expr { …
  public void accept(Visitor v) {
    v.visit(this);
  }
}

public class Add extends Expr { …
  public void accept(Visitor v) {
    v.visit(this);
  }
}
```

```
public class PrintPass implements Visitor {
  public void visit(Num n) {
    System.out.print(n.i);
  }
  public void visit(Add n) {
    System.out.print("(");
    n.e0.accept(this);
    System.out.print("+");
    n.e1.accept(this);
    System.out.print(")");
  }
}
```

1. Call `node.accept(ppass)` (dispatch on **Node**)
2. Call `v.visit(node)` (dispatch on **Pass**)

Visitor Pattern — Calculator

```
interface Visitor {
  public void visit(Num n);
  public void visit(Add n);
}

abstract public class Expr {
  abstract void accept(Visitor v);
}

public class Num extends Expr { …
  public void accept(Visitor v) {
    v.visit(this);
  }
}

public class Add extends Expr { …
  public void accept(Visitor v) {
    v.visit(this);
  }
}
```

```
public class PrintPass implements Visitor {
  public void visit(Num n) {
    System.out.print(n.i);
  }
  public void visit(Add n) {
    System.out.print("(");
    n.e0.accept(this);
    System.out.print("+");
    n.e1.accept(this);
    System.out.print(")");
  }
}
```

Inside the visit method, we make structurally recursive calls by invoking

`child.accept(visitor);`

The Secret Trick of Visitors

- Why can't we just call `visitor.visit(node)` directly?
  - Why does it help to call `node.accept(visitor)` instead?
- In the `visitor.visit(node)` call, the dynamic dispatch is on the `visitor`. We have to statically dispatch on the type of `node`.
  - So we need to be in a snippet of code where the exact type of `node` is known. (This is why `accept` and the whole visitor pattern is fundamentally needed)

Variations on Visitor Pattern

- Instead of having a return type of `void`, we could define a visitor interface generically over return types **`RetT`**

```
interface Visitor<RetT> {
  public RetT visit(Num n);
  …
```

- We could hard code the structurally recursive calls into the `accept` methods as a fixed traversal order

```
public class Add extends Expr { …
  public void accept(Visitor v) {
    e0.accept(v);
    e1.accept(v);
    v.visit(this);
} }
```

Factoring Out Code w/Visitors

- Some compiler passes only need to visit a few types of AST Nodes
  - e.g. a pass to collect all variable names used in a program only needs to visit nodes that store variable names
- If we define a visitor superclass with a default traversal order (e.g. post-order) then we only need to overload a few visitors in a sub-class
- Large compiler projects using visitors will define a few (6-10) types of basic visitors to take advantage of this strategy

Why is this so complicated?

- (Gilbert's opinion) Object-oriented languages got overly focused on the "right way" to do things — requiring the development of confusing "design patterns"
  - Many of these were compensating for missing language features that are slowly and poorly being incorporated (e.g. Java now has pattern matching)
- But a lot of what people call The Visitor Pattern is really about various Pass-in-a-Class strategies
  - This is a fundamentally good idea in compiler design
  - In non object-oriented languages, similar modularity mechanisms are used to encapsulate passes

References on Design Patterns

- Two classic books on design patterns
  - **Design Patterns: Elements of Reusable Object-Oriented Software.** *Gamma, Helm, Johnson, and Vlissides.* Addison-Wesley, 1995. (the classic "Gang of Four" book on design patterns; code in C++ & Smalltalk)
  - **Object-Oriented Design & Patterns.** *Horstmann.* Wiley, 2nd edition, 2005. (code in Java)
- If you want information specifically about the MiniJava AST design, see the starter code and Appel's textbook