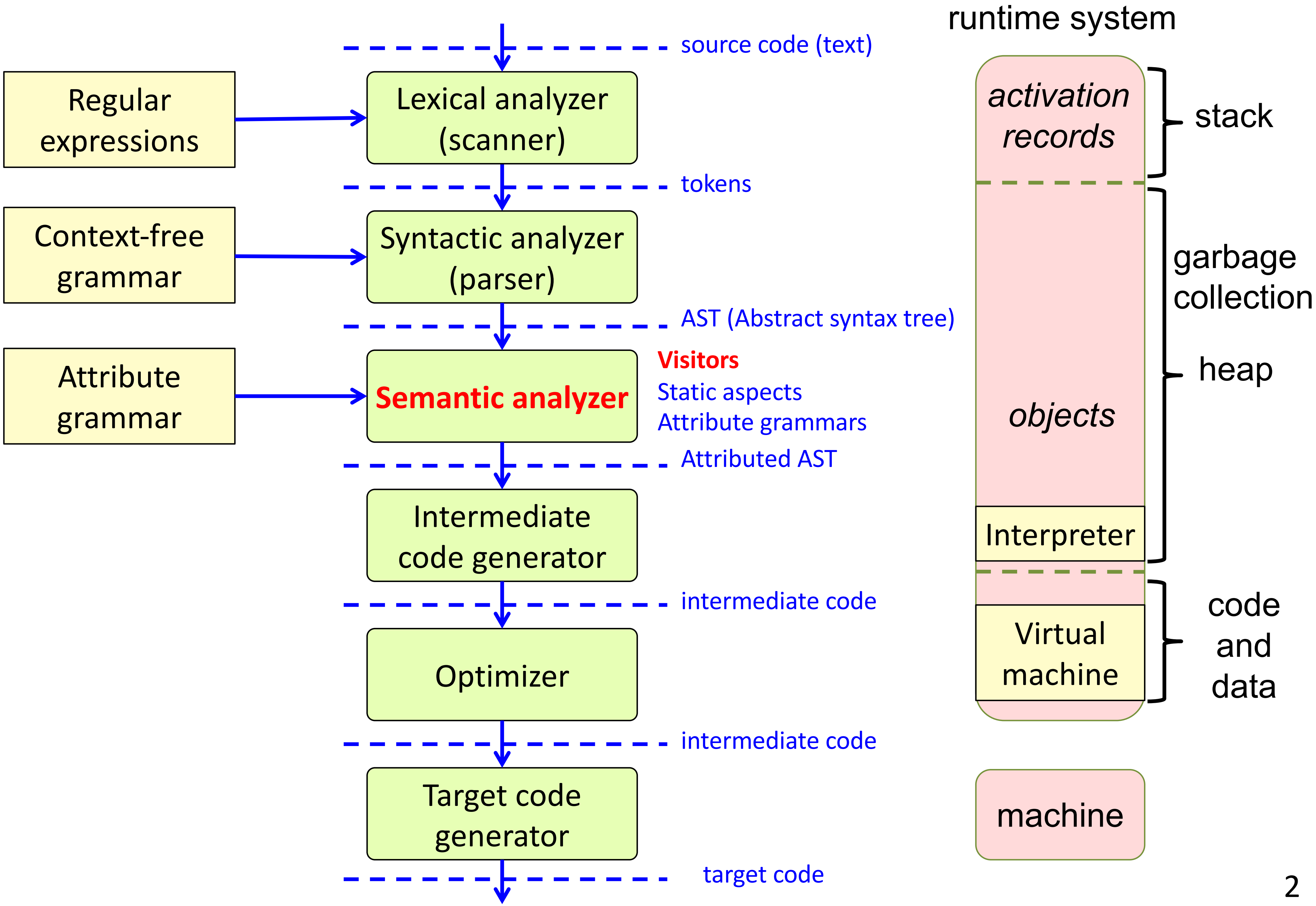
EDAN65: Compilers, Lecture 06 B

Visitors

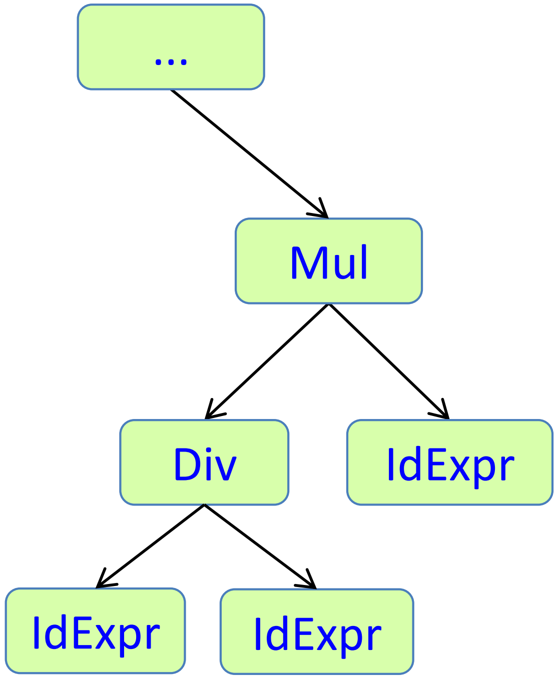
Görel Hedin

Revised: 2021-09-14

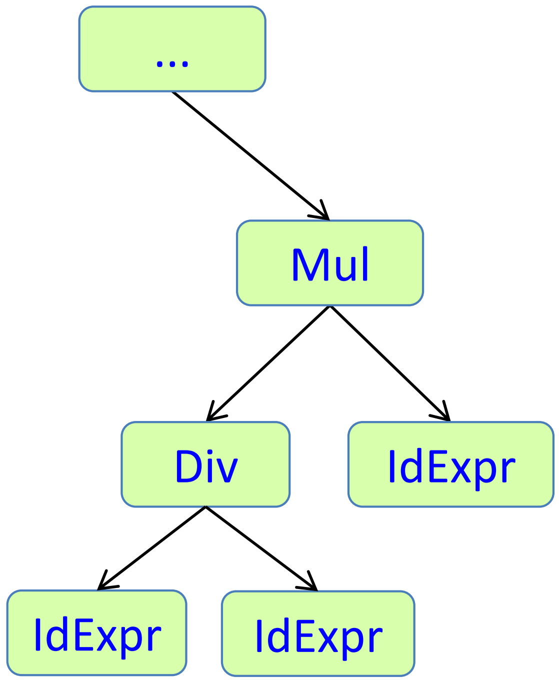
This lecture



Example computations on an AST



Example computations on an AST



Name analysis: find the declaration of an identifier

Type analysis: compute the type of an expression

Expression evaluation: compute the value of a constant expression

Code generation: compute an intermediate code representation of the program

Unparsing: compute a text representation of the program

Exercise: expression evaluation

Abstract grammar

```
abstract Expr;
BinExpr : Expr ::= Left:Expr Right:Expr;
Add : BinExpr;
Sub : BinExpr;
IntExpr : Expr ::= <INT:String>;
```

Generated AST classes

```
abstract class Expr extends ASTNode {

}
class BinExpr extends Expr { Expr getLeft() {...} Expr getRight {...} }
class Add extends BinExpr {

}
class Sub extends BinExpr {

}
class IntExpr extends Expr {
  String getINT() {...}

}
```

Solution: expression evaluation

Abstract grammar

```
abstract Expr;
BinExpr : Expr ::= Left:Expr Right:Expr;
Add : BinExpr;
Sub : BinExpr;
IntExpr : Expr ::= <INT:String>;
```

Edited AST classes

```
abstract class Expr extends ASTNode {
  abstract int value();
}
class BinExpr extends Expr { Expr getLeft() {...} Expr getRight {...} }
class Add extends BinExpr {
  int value() { return getLeft().value() + getRight().value(); }
}
class Sub extends BinExpr {
  int value() { return getLeft().value() - getRight().value(); }
}
class IntExpr extends Expr {
  String getINT() {...}
  int value() { return String.parseInt(getINT()); }
}
```

Solution: expression evaluation

Abstract grammar

```
abstract Expr;
BinExpr : Expr ::= Left:Expr Right:Expr;
Add : BinExpr;
Sub : BinExpr;
IntExpr : Expr ::= <INT:String>;
```

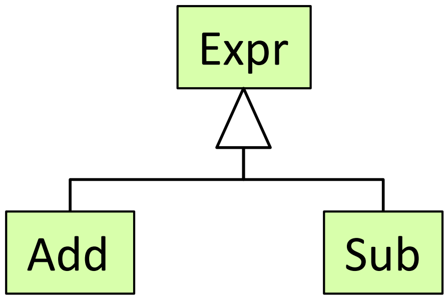
Problem 1: NEVER EDIT GENERATED CODE!!
Problem 2: The code is not modular!
We have to edit every AST class!
The computation of value() is a cross-cutting concern, leading to tangled code.

Edited AST classes

```
abstract class Expr extends ASTNode {
  abstract int value();
}
class BinExpr extends Expr { Expr getLeft() {...} Expr getRight {...} }
class Add extends BinExpr {
  int value() { return getLeft().value() + getRight().value(); }
}
class Sub extends BinExpr {
  int value() { return getLeft().value() - getRight().value(); }
}
class IntExpr extends Expr {
  String getINT() {...}
  int value() { return String.parseInt(getINT()); }
}
```

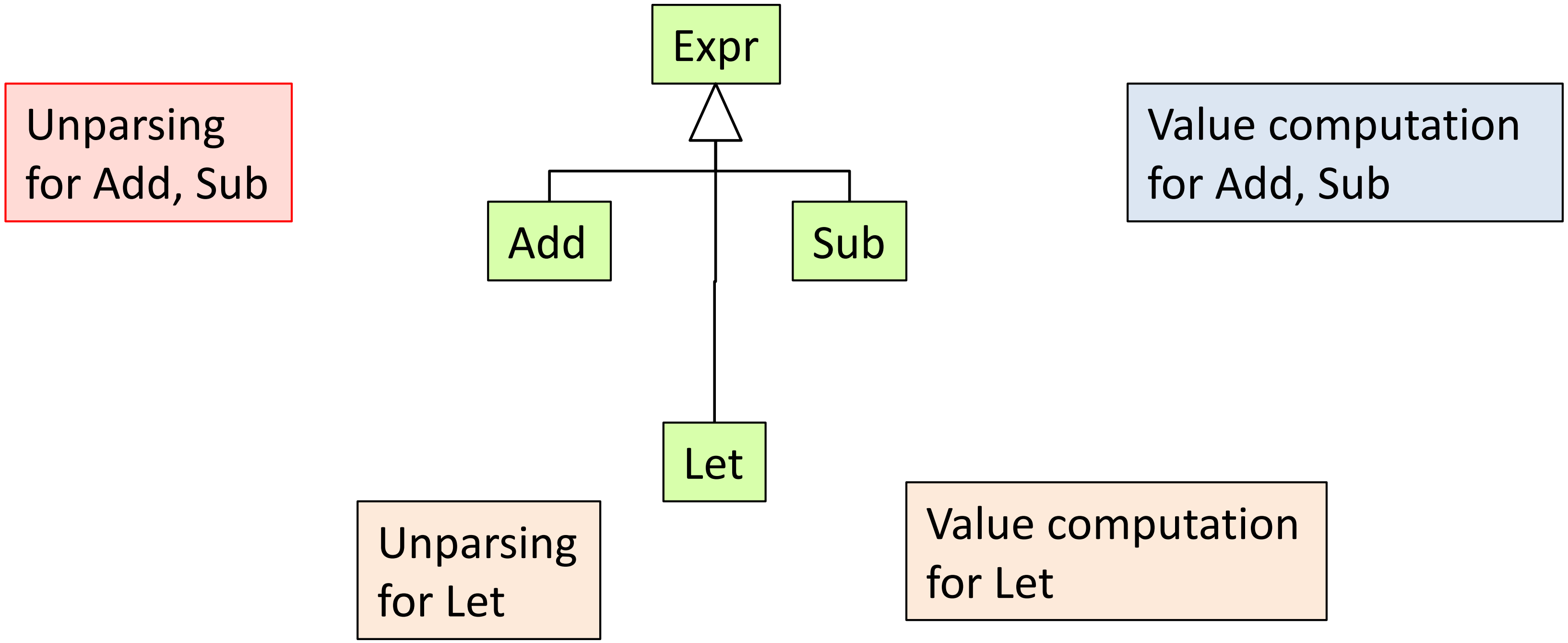
The Expression Problem

- We would like to
  - define language constructs in a modular way (in a class hierarchy).
  - define computations in a modular way (on those classes)
  - compose these modules as we like
  - preferrably, with separate compilation of the modules
  - and with full static type safety (without need for casts or instanceof)



The Expression Problem

- We would like to
  - define language constructs in a modular way (in a class hierarchy).
  - define computations in a modular way (on those classes)
  - compose these modules as we like
  - preferrably, with separate compilation of the modules
  - and with full static type safety (without need for casts or instanceof)



The simplest solution: Static aspects

```
abstract class Exp extends ASTNode {
  abstract int value();
}
class Add extends Exp {
  int value() {
    return getLeft().value() +
            getRight().value();
  }
}
class IntExp extends Exp{
  String getINT() {...}
  int value() {
    return String.parseInt(getINT());
  }
}
```

Factor out the tangled code into an aspect.
Requires the language to support static aspects.
Not supported in Java. Requires another language like AspectJ, or JastAdd.

The simplest solution: Static aspects

```
abstract class Exp extends ASTNode {

}
class Add extends Exp {





}
class IntExp extends Exp{
  String getINT() {...}




}
```

```
aspect ValueComputation{
  abstract int Exp.value();


  int Add.value() {
    return getLeft().value() +
              getRight().value();
  }




  int IntExp.value() {
    return String.parseInt(getINT());
  }
}
```

Factor out the tangled code into an aspect.
Requires the language to support static aspects.
Not supported in Java. Requires another language like AspectJ, or JastAdd.

Dealing with the expression problem

Dealing with the expression problem

- Edit the AST classes (i.e., actually not solving the problem)
    - Non-modular, non-compositional.
    - **It is always a VERY BAD IDEA to edit generated code!**
    - Sometimes used anyway in industry.
- Visitors: an OO design pattern.
    - Modularize through clever indirect calls.
    - Not full modularization, not composition.
    - Supported by many parser generators.
    - Reasonably useful, commonly used in industry.
- Static Aspect-Oriented Programming (AOP)
    - Also known as *inter-type declarations* (ITDs)
    - Use new language constructs (aspects) to factor out code.
    - Solves the expression problem in a nice simple way.
    - The drawback: you need a new language: AspectJ, JastAdd, ...
- Advanced language constructs
    - Use more advanced language constructs: virtual classes in gbeta, traits in Scala, typeclasses in Haskell, ...
    - Drawbacks: More complex than static AOP. You need an advanced language. Not much practical experience (so far).

This lecture: Visitors

Visitors

How to modularize compilers in Java
(or any other OO language without AOP mechanisms).

The Visitor design pattern lets you define a new operation without changing the elements on which it operates.
[Gamma, Helm, Johnson, Vlissides, 1994]

A simple example (without visitors)

Original code

```
class Add extends Exp {
  Exp e1, e2;
}
class IntExp extends Exp {
  int value;
}
```

A simple example (without visitors)

Original code

```
class Add extends Exp {
  Exp e1, e2;
}
class IntExp extends Exp {
  int value;
}
```

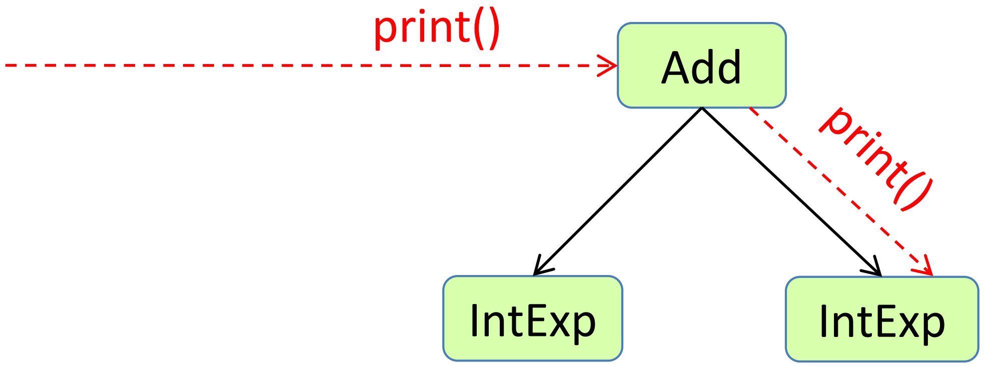
After adding the print method

```
class Add extends Exp {
  Exp e1, e2;
  void print() {
    e1.print();
    System.out.print("+");
    e2.print();
  }
}
class IntExp extends Exp {
  int value;
  void print() {
     System.out.print(value);
  }
}
```

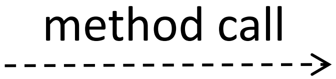
Could we add the print methods, without changing the original code?

Main idea of visitors

Without visitor:

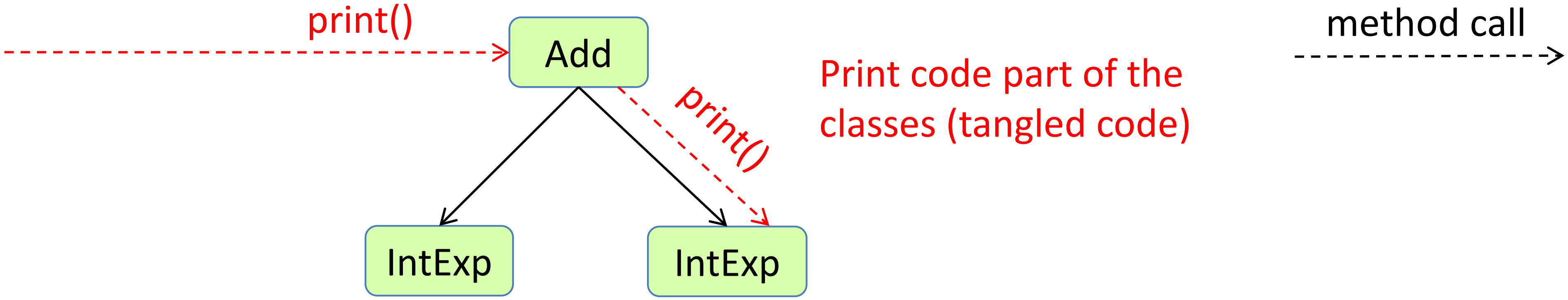


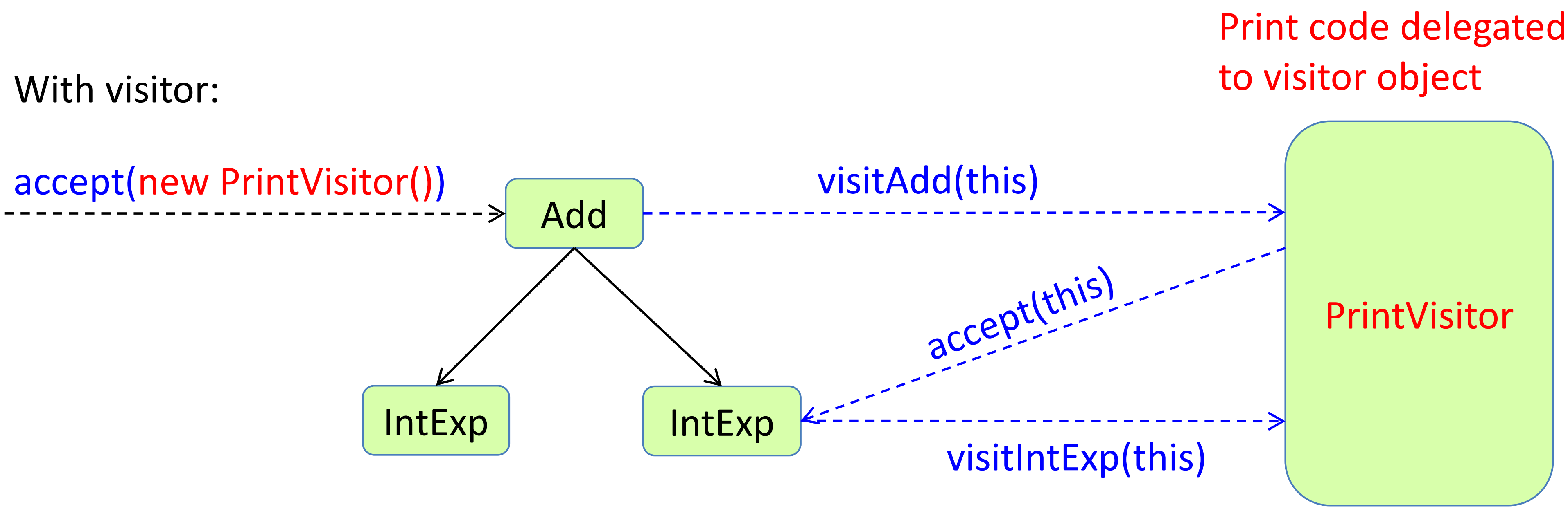
Print code part of the classes (tangled code)



Main idea of visitors

Without visitor:





Example implementation

AST classes

```
class Add extends Exp {
  Exp e1, e2;
  void accept(Visitor v) {
    v.visitAdd(this);
  }
}
class IntExp extends Exp {
  int value;
  void accept(Visitor v) {
    v.visitIntExp(this);
  }
}
```

General boilerplate code for visitors,
can be generated from the grammar.

Example implementation

AST classes

```
class Add extends Exp {
  Exp e1, e2;
  void accept(Visitor v) {
    v.visitAdd(this);
  }
}
class IntExp extends Exp {
  int value;
  void accept(Visitor v) {
    v.visitIntExp(this);
  }
}
```

General visitor

```
interface Visitor {
  void visitAdd(Add n);
  void visitIntExp(IntExp n);
}
```

General boilerplate code for visitors,
can be generated from the grammar.

Example implementation

AST classes

```
class Add extends Exp {
  Exp e1, e2;
  void accept(Visitor v) {
    v.visitAdd(this);
  }
}
class IntExp extends Exp {
  int value;
  void accept(Visitor v) {
    v.visitIntExp(this);
  }
}
```

General visitor

```
interface Visitor {
  void visitAdd(Add n);
  void visitIntExp(IntExp n);
}
```

Modular addition of print

```
class Print implements Visitor {
  void visitAdd(Add n) {
    n.e1.accept(this);
    System.out.print("+");
    n.e2.accept(this);
  }
  void visitIntExp(IntExp n) {
    System.out.print(n.value);
  }
}
```

General boilerplate code for visitors, can be generated from the grammar.

Many implementations use Java overloading

for the visit methods

AST classes

```
class Add extends Exp {
  Exp e1, e2;
  void accept(Visitor v) {
    v.visit(this);
  }
}
class IntExp extends Exp {
  int value;
  void accept(Visitor v) {
    v.visit(this);
  }
}
```

General visitor

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

Modular addition of print

```
class Print implements Visitor {
  void visit(Add n) {
    n.e1.accept(this);
    System.out.print("+");
    n.e2.accept(this);
  }
  void visit(IntExp n) {
    System.out.print(n.value);
  }
}
```

Tricky question: The accept methods all look the same! Can it be factored out to a superclass?

Many implementations use Java overloading

for the visit methods

AST classes

```
class Add extends Exp {
  Exp e1, e2;
  void accept(Visitor v) {
    v.visit(this);
  }
}
class IntExp extends Exp {
  int value;
  void accept(Visitor v) {
    v.visit(this);
  }
}
```

General visitor

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

Modular addition of print

```
class Print implements Visitor {
  void visit(Add n) {
    n.e1.accept(this);
    System.out.print("+");
    n.e2.accept(this);
  }
  void visit(IntExp n) {
    System.out.print(n.value);
  }
}
```

Tricky question: The accept methods all look the same! Can it be factored out to a superclass?

Answer: No! Because the calls go to *different* visit methods: "this" has different types for the different calls. The visit methods are *overloaded* (same name but different argument types).

Typical Visitor interface

has return value and data parameter

The Visitor interface

```
interface Visitor {
  Object visit(Add node, Object data);
  Object visit(IntExp node, Object data);
}
```

The AST classes

```
class Add extends Exp {
  ...
  Object accept(Visitor v, Object data) {
    return v.visit(this, data);
  }
}
class IntExp extends Exp {
  ...
  Object accept(Visitor v, Object data) {
    return v.visit(this, data);
  }
}
```

Example visitor: expression evaluation

Without visitors:

```
class Exp{
  abstract int value();
}
```

```
class Add extends Exp{
  int value() {
    return getLeft().value() +
      getRight().value(); }
}
```

```
class Sub extends Exp{
  int value() { ... }
}
```

```
class IntExp extends Exp{
  int value() { ... }
}
```

Example visitor: expression evaluation

Without visitors:

```
class Exp{
  abstract int value();
}
```

```
class Add extends Exp{
  int value() {
    return getLeft().value() +
      getRight().value(); }
}
```

```
class Sub extends Exp{
  int value() { ... }
}
```

```
class IntExp extends Exp{
  int value() { ... }
}
```

Corresponding Visitor

```
class Evaluator implements Visitor {
  Object visit(Add node, Object data) {
    int n1 =  (Integer) node.getLeft().accept(this, data);
    int n2 =  (Integer) node.getRight().accept(this, data);
    return new Integer(n1+n2);
  }
  Object visit(Sub node, Object data) { ... }
  Object visit(IntExp node, Object data { ...}
}
```

quite a lot of boilerplate

extra type casts

Example visitor: expression evaluation

Without visitors:

```
class Exp{
  abstract int value();
}
```

```
class Add extends Exp{
  int value() {
    return getLeft().value() +
      getRight().value(); }
}
```

```
class Sub extends Exp{
  int value() { ... }
}
```

```
class IntExp extends Exp{
  int value() { ... }
}
```

Corresponding Visitor

```
class Evaluator implements Visitor {
  Object visit(Add node, Object data) {
    int n1 =  (Integer) node.getLeft().accept(this, data);
    int n2 =  (Integer) node.getRight().accept(this, data);
    return new Integer(n1+n2);
  }
  Object visit(Sub node, Object data) { ... }
  Object visit(IntExp node, Object data { ...}
}
```

quite a lot of boilerplate

extra type casts

Casts needed to access return and data values.
(Could be solved by type parameters on the visitor interface.)

Calling the visitor

Method 1: Create the visitor and call the accept method

```
Exp e = ...;
int result = (Integer) e.accept(new Evaluator(), null);
```

Calling the visitor

Method 1: Create the visitor and call the accept method

```
Exp e = ...;
int result = (Integer) e.accept(new Evaluator(), null);
```

Method 2: Much simpler client code. Provide a static convenience method

```
Exp e = ...;
int result = Evaluator.result(e);
```

Calling the visitor

Method 1: Create the visitor and call the accept method

```
Exp e = ...;
int result = (Integer) e.accept(new Evaluator(), null);
```

Method 2: Much simpler client code. Provide a static convenience method

```
Exp e = ...;
int result = Evaluator.result(e);
```

Implementation of convenience method

```
class Evaluator implements Visitor {
  static int result(Exp node) {
    return (Integer) node.accept(new Evaluator(), null);
  }
  Object visit(Add node, Object data) {...}
  Object visit(Sub node, Object data) { ... }
  Object visit(IntExp node, Object data { ...}
}
```

Example: Visitor with local field

Without visitors:

```
class Exp{
  abstract void unparse(Stream s);
}
```

Pass the stream as a parameter

```
class Add ... {
  void unparse(Stream s) {
    getLeft().unparse(s);
    s.print("+");
    getRight().unparse(s);
  }
}
```

```
class Sub ...{
  ...
}
```

```
class IntExp ... {
  ...
}
```

Example: Visitor with local field

Without visitors:

```
class Exp{
  abstract void unparse(Stream s);
}
```

Pass the stream as a parameter

```
class Add ... {
  void unparse(Stream s) {
    getLeft().unparse(s);
    s.print("+");
    getRight().unparse(s);
  }
}
```

```
class Sub ...{
  ...
}
```

```
class IntExp ... {
  ...
}
```

Corresponding Visitor

```
class Unparser implements Visitor {
  Unparser(Stream s) { this.s = s; }
  Stream s;
  Object visit(Add node, Object data) {
    node.getLeft().accept(this, data);
    s.print("+");
    node.getRight().accept(this, data);
    return null;
  }
  ...
}
```

No need for stream parameter.
Keep it in the visitor.
Nice!

Adding a convenience method for clients

Adding a convenience method for clients

Client code

```
Exp e = ...;
Stream s = ...;
Unparser.doit(e, s);
```

Adding a convenience method for clients

Client code

```
Exp e = ...;
Stream s = ...;
Unparser.doit(e, s);
```

Visitor

```
class Unparser implements Visitor {
  static void doit(Exp e, Stream s) {
    e.accept(new Unparser(s), null);
  }
  Unparser(Stream s) { this.s = s; }
  Stream s;
  Object visit(Add node, Object data) {
    node.getLeft().accept(this, data);
    s.print("+");
    node.getRight().accept(this, data);
    return null;
  }
  ...
}
```

One more example

Count the number of identifiers in a program

Abstract grammar

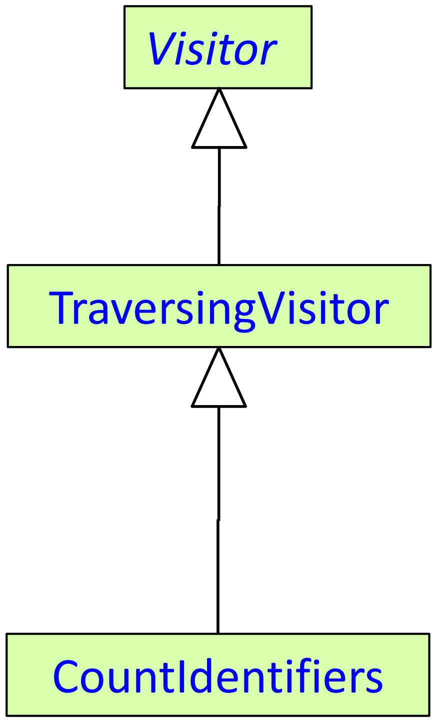
```
abstract Stmt;
IfStmt : Stmt ::= Cond:Exp Then:Stmt [Else:Stmt]
...
abstract Expr;
BinExpr : Expr ::= Left:Expr Right:Expr;
Add : BinExpr;
Sub : BinExpr;
IntExpr : Expr ::= <INT:String>;
IdExpr : Expr ::= <ID:String>
...
```

How can we implement the visitor?

Problem: We need to write lots of boring traversal code...

Solution:

Introduce a general traversing Visitor



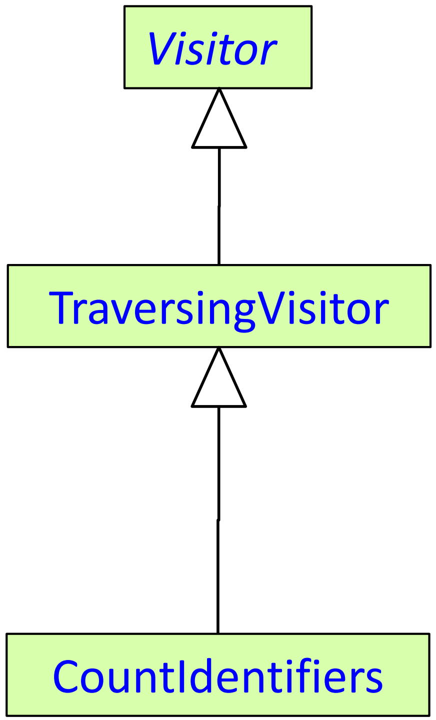
Interface, as before

Default implementations that just traverse the AST.
Could be generated from the grammar.

Override default traversal when needed.

Solution:

Introduce a general traversing Visitor



Interface, as before

Default implementations that just traverse the AST.
Could be generated from the grammar.

Override default traversal when needed.

Some parser generators generate several different kinds of visitors, for different kinds of traversals.

Implementation of TraversingVisitor

```
class TraversingVisitor implements Visitor {

  private Object visitChildren(ASTNode node, Object data) {
    for (int i = 0; i < node.getNumChild(); ++i) {
      node.getChild(i).accept(this, data);
    }
    return data;
  }

  Object visit(IfStmt node, Object data) {
    return visitChildren(node, data);
  }
  Object visit(Add node, Object data) {
    return visitChildren(node, data);
  }
  Object visit(Sub node, Object data) {
    return visitChildren(node, data);
  }
  ...
}
```

CountIdentifiers as a traversing visitor

Example use:

```
Program p = ...
System.out.print("The number of identifiers is: ");
System.out.println(CountIdentifiers.result(p));
```

Visitor

```
class CountIdentifiers extends TraversingVisitor {
  int count = 0;
  static int result(Program root) {
    CountIdentifiers v = new CountIdentifiers();
    root.accept(v);
    return v.count;
  }
  Object visit(IdExpr node, Object data) {
    count++;
    return null;
  }
}
```

CountIdentifiers as a traversing visitor

Example use:

```
Program p = ...
System.out.print("The number of identifiers is: ");
System.out.println(CountIdentifiers.result(p));
```

Visitor

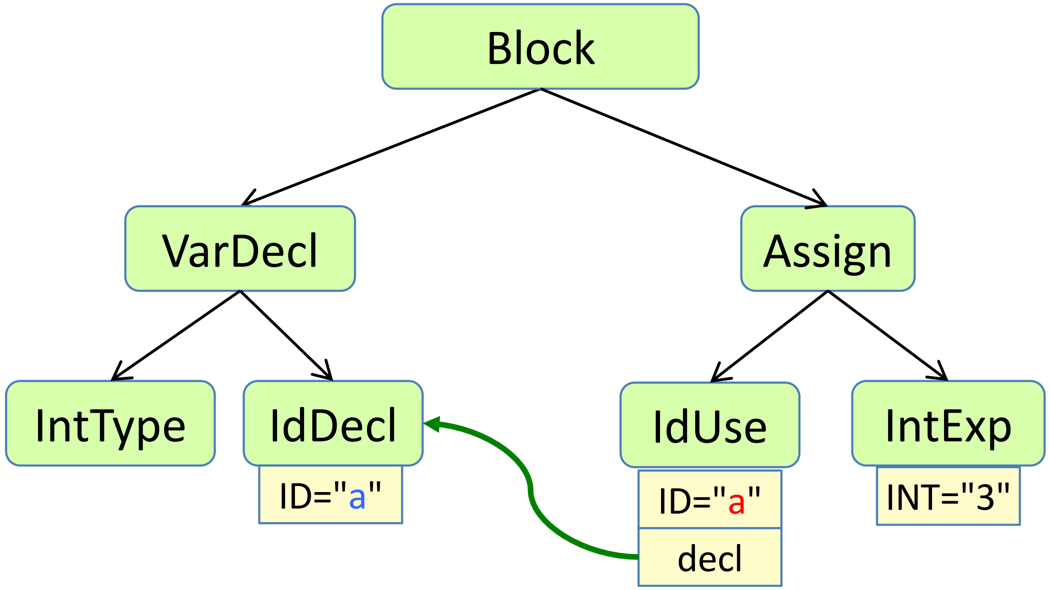
```
class CountIdentifiers extends TraversingVisitor {
  int count = 0;
  static int result(Program root) {
    CountIdentifiers v = new CountIdentifiers();
    root.accept(v);
    return v.count;
  }
  Object visit(IdExpr node, Object data) {
    count++;
    return null;
  }
}
```

Only one visit method needed.

Nice!

Representing name bindings in an AST

```
{
  int a;
  a = 3;
}
```



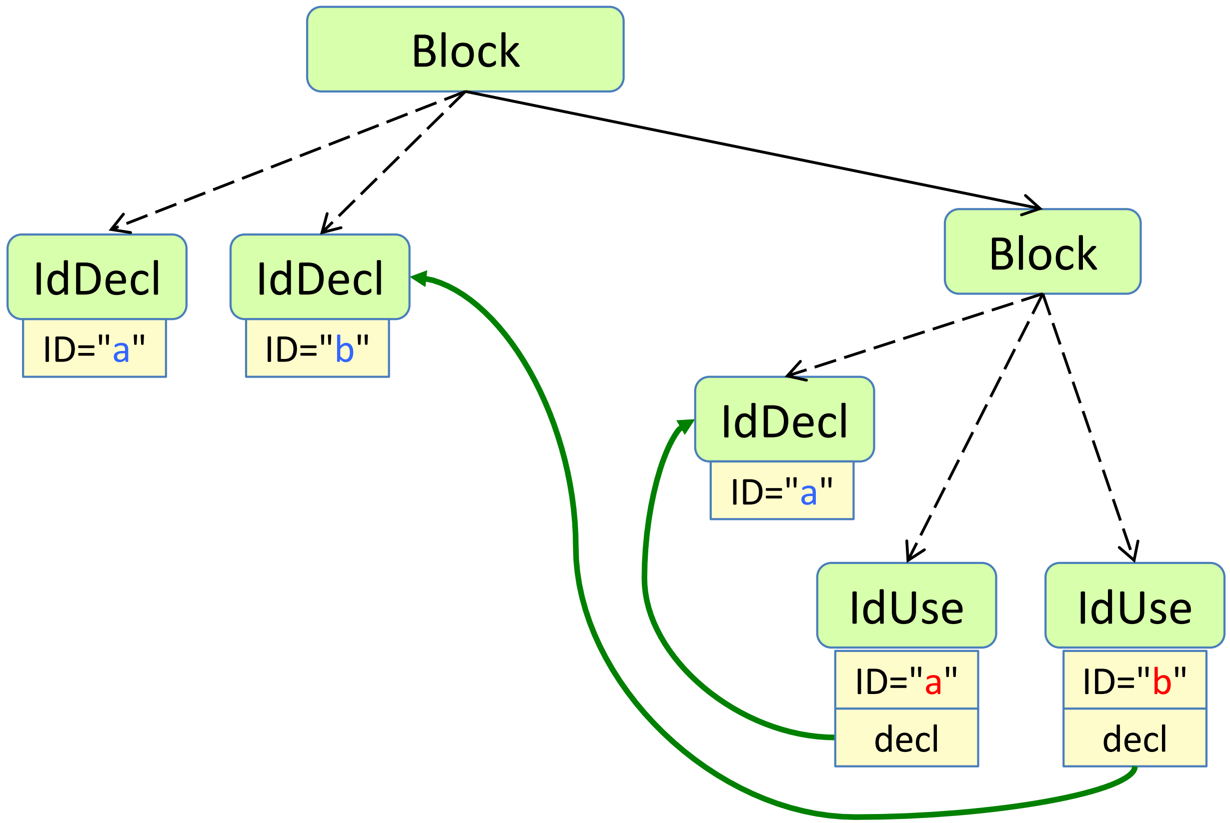
Differ between declarations and uses!

IdDecl for declared names
IdUse for used names

An attribute **decl** represents the name binding.

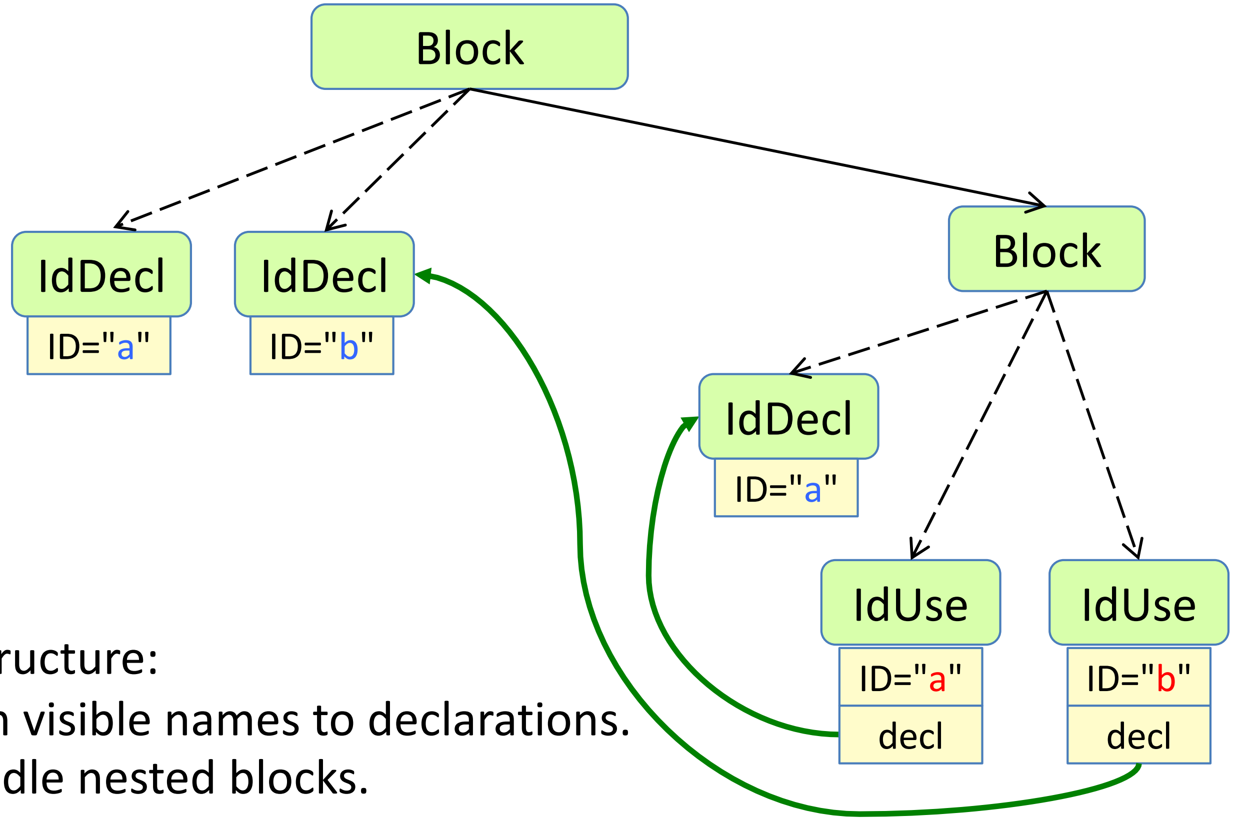
Computing name bindings imperatively

```
{
  int a = 3;
  int b = 4;
  {
    int a;
    a = b;
  }
}
```



Computing name bindings imperatively

```
{
  int a = 3;
  int b = 4;
  {
    int a;
    a = b;
  }
}
```



Use a **symbol table** data structure:
For each block, a **map** from visible names to declarations.
Use a **stack** of maps to handle nested blocks.

Algorithm:
Traverse the AST
push/pop symbol table when entering/leaving a block
add/lookup identifiers when encountering IdDecls/IdUses

Example API for block structured symbol table

```
class SymbolTable<M> {
	void add(String symbol, M meaning); // add to top table
	void enterBlock(); // push new table
	void exitBlock();  // pops top table
	M lookup(String symbol); // returns the meaning of the symbol
}
```

Example API for block structured symbol table

```
class SymbolTable<M> {
    void add(String symbol, M meaning); // add to top table
    void enterBlock(); // push new table
    void exitBlock();  // pops top table
    M lookup(String symbol); // returns the meaning of the symbol
}
```

Could be used, for example, in a visitor:

```
class NameAnalysis extends TraversingVisitor {
  SymbolTable<IdDecl> st = new SymbolTable<IdDecl>();
  void visit(Block node) {
    st.enterBlock();
    visitChildren(node);
    st.exitBlock();
  }
  void visit(IdDecl node) {
    st.add(node.getID(), node);
  }
  void visit(IdUse node) {
    node.decl = st.lookup(node.getID());
  }
}
```

Summary questions

- What is the Expression Problem?
- Why is solving the Expression Problem desirable for implementing compilers?
- Why is it a bad idea to edit generated code?
- Explain how the Visitor pattern can be implemented.
- Implement a computation over the AST using visitors.
- Add a convenience method to the visitor to make it easier to call from client code.
- Why can traversing visitors be useful?
- What is a symbol table?
- Why use both IdDecl and IdUse instead of just one AST type?