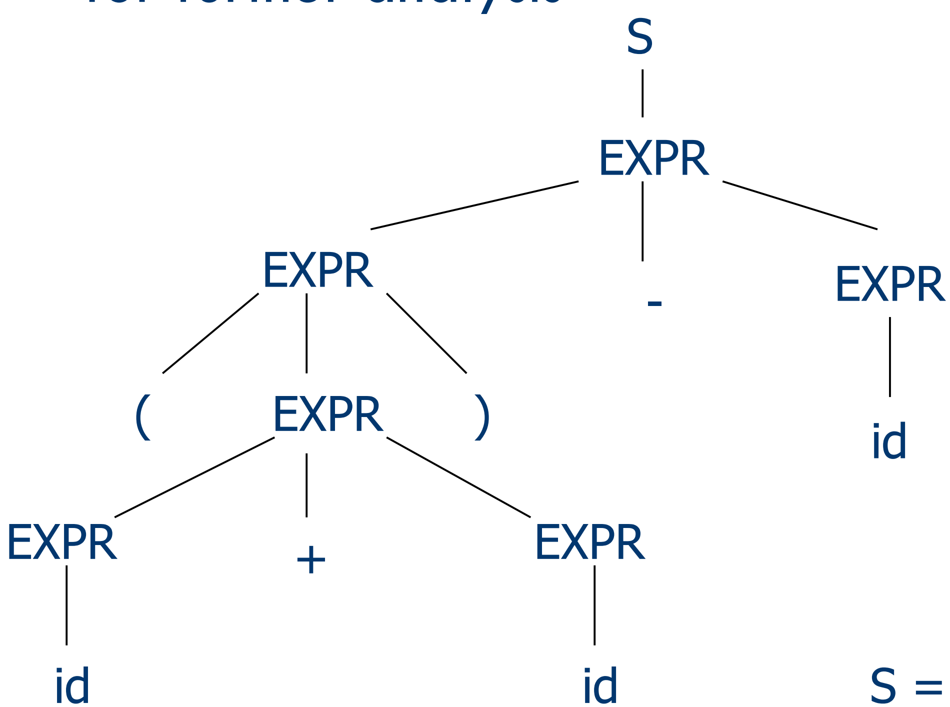
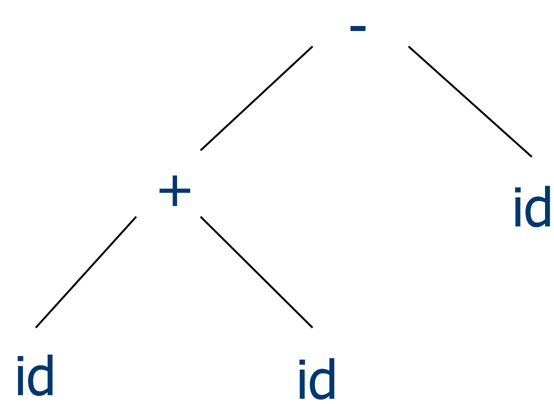
Abstract Syntax Trees

CIS*4650 (Winter 2020)

Parse Trees vs. Abstract Syntax Trees

- Parse Tree: graphically show a derivation with LHS connected to its RHS components.
- Abstract Syntax Tree: only capture the information needed for further analysis





S =>* (id + id) - id

Code Example in Tiny

```
{ Sample program in Tiny language --
computing
     factorial }

read x;    { input an integer  }
if x > 0 then  { don't compute if x <= 0 }
   fact := 1;
   repeat
      fact := fact * x;
      x := x - 1
   until x = 0;
   write fact  { output factorial of x }
end
```

Class-Based AST for Tiny

```
package absyn;

abstract public class Absyn {
    public int pos;
}

abstract public class Exp extends Absyn {
}

// Subclasses of Exp:
public class AssignExp extends Exp {
    public VarExp lhs;
    public Exp rhs;
    public AssignExp( int pos, VarExp lhs; Exp rhs ) {
        this.pos = pos;
        this.lhs = lhs;
        this.rhs = rhs;
    }
}

// Constructors for other subclasses
IntExp(int pos, int value)
VarExp(int pos, String name)
OpExp(int pos, Exp left, int op, Exp  right)
IfExp(int pos, Exp test, ExpList thenpart, ExpList elsepart)
RepeatExp(int pos, ExpList exps, Exp test)
WriteExp(int pos, Exp output)
ReadExp(int pos, VarExp input)
```

```
// miscellaneous classes
ExpList(Exp head, ExpList tail)

// constants for op field of  OpExp
final static int OpExp.PLUS, OpExp.MINUS,
   OpExp.TIMES, OpExp.OVER,
   OpExp.EQ, OpExp.LT, OpExp.GT
```

Class-Based AST Example

Code fragment:

fact := fact * x;
x := x –1

- Abstract syntax tree in Java:

```
ExpList(
   AssignExp(0,
     VarExp(0,"fact"),
     OpExp(0,VarExp(0,"fact"),OpExp.TIMES,VarExp(0,"x"))),
   ExpList(
     AssignExp(1,
        VarExp(1,"x"),
        OpExp(1,VarExp(1,"x"),OpExp.MINUS,IntExp(1,1))),
     null))
```

- Creating an abstract syntax tree "x := x –1" in Java:

```
new ExpList(
   new AssignExp(1,
      new VarExp(1,"x"),
      new OpExp(1,
         new VarExp(1,"x"),
         OpExp.MINUS,
         new IntExp(1,1))),
   null))
```

Code Example in C-minus

```
/* A program that uses Euclid's algorithm to
   compute gcd */

int gcd (int u, int v ) {
   if (v == 0)
      return u;
   else
      // u-u/v*v == u mod v
      return gcd(v, u - u/v*v);
}

void main (void) {
   int x;
   int y;
   x = input();
   y = input();
   output( gcd(x, y) );
}
```

Class-Based AST for C-minus

package absyn;

abstract class Absyn
NameTy(int pos, int typ)

abstract class Var extends Absyn
SimpleVar(int pos, String name)
IndexVar(int pos, String name, Exp index)

abstract class Exp extends Absyn
NilExp(int pos)
VarExp(int pos, Var variable)
IntExp(int pos, int value)
CallExp(int pos, String func, ExpList args)
OpExp(int pos, Exp left, int op, Exp right)
AssignExp(int pos, Var lhs, Exp rhs)
IfExp(int pos, Exp test, Exp then, Exp else)
WhileExp(int pos, Exp test, Exp body)
ReturnExp(int pos, Exp exp)
CompoundExp(int pos, VarDecList decs, ExpList exps)

abstract class Dec extends Absyn
FunctionDec(int pos, NameTy result, String func,VarDecList params,
CompoundExp body)

abstract class VarDec extends Dec
SimpleDec(int pos, NameTy typ, String name)
ArrayDec(int pos, NameTy typ, String name, IntExp size)

// miscellaneous classes
DecList(Dec head, DecList tail)
VarDecList(VarDec head, VarDecList tail)
ExpList(Exp head, ExpList tail)

// constants for op *field of* **OpExp**
final static int OpExp.**PLUS**, OpExp.**MINUS**,
OpExp.**MUL**, OpExp.**DIV**, OpExp.**EQ**, OpExp.**NE**,
OpExp.**LT**, OpExp.**LE**, OpExp.**GT**, OpExp.**GE**;

// constants for typ field of **NameTy**:
final static int NameTy.**INT**, NameTy.**VOID**

// Note that concrete classes have boldface names
// and are listed under their super-classes

Traversing the AST's

- Issues with AST processing:
  - For Java-like languages, AST's have about 50 node types
  - For GNU Compiler Collection (GCC), there are about 200 phases in the compilation process

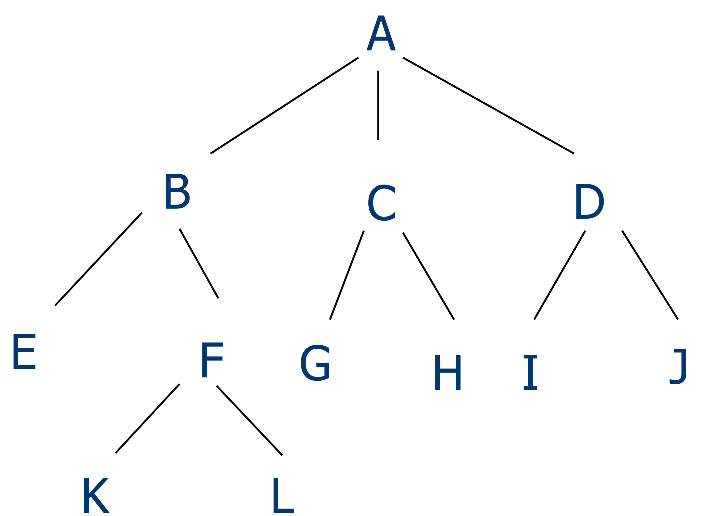
- Better to isolate the code for each phase in single classes rather than distribute it among the various node types

- The "visitor" pattern allows us to add a new function to a family of classes without modifying the classes

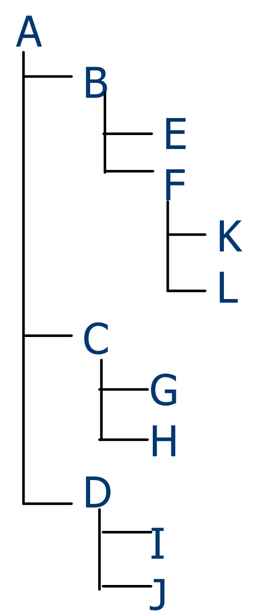
Pre-order Traversal

- A root is processed before its children:

```
void preorder(Absyn tree) {
    process(tree);
    for (int i = 0; i < tree.children.size(); i++)
       preorder(tree.children.get(i));
}
```



Display with indentation:

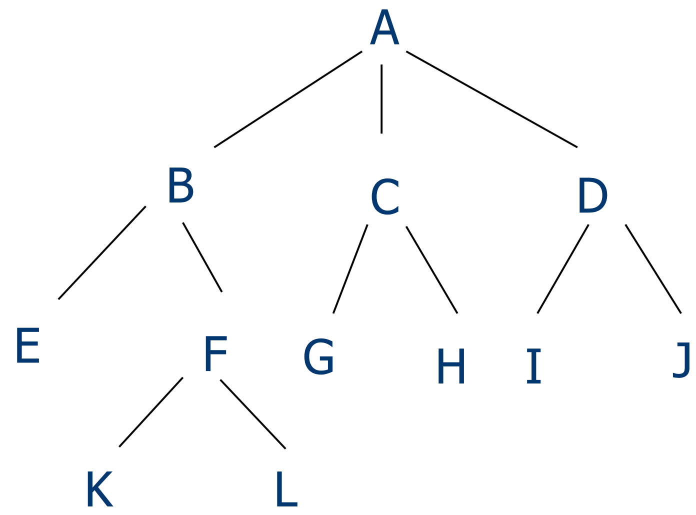


Pre-order results: A, B, E, F, K, L, C, G, H, D, I, J.

Post-order Traversal

- A root is processed after its children:

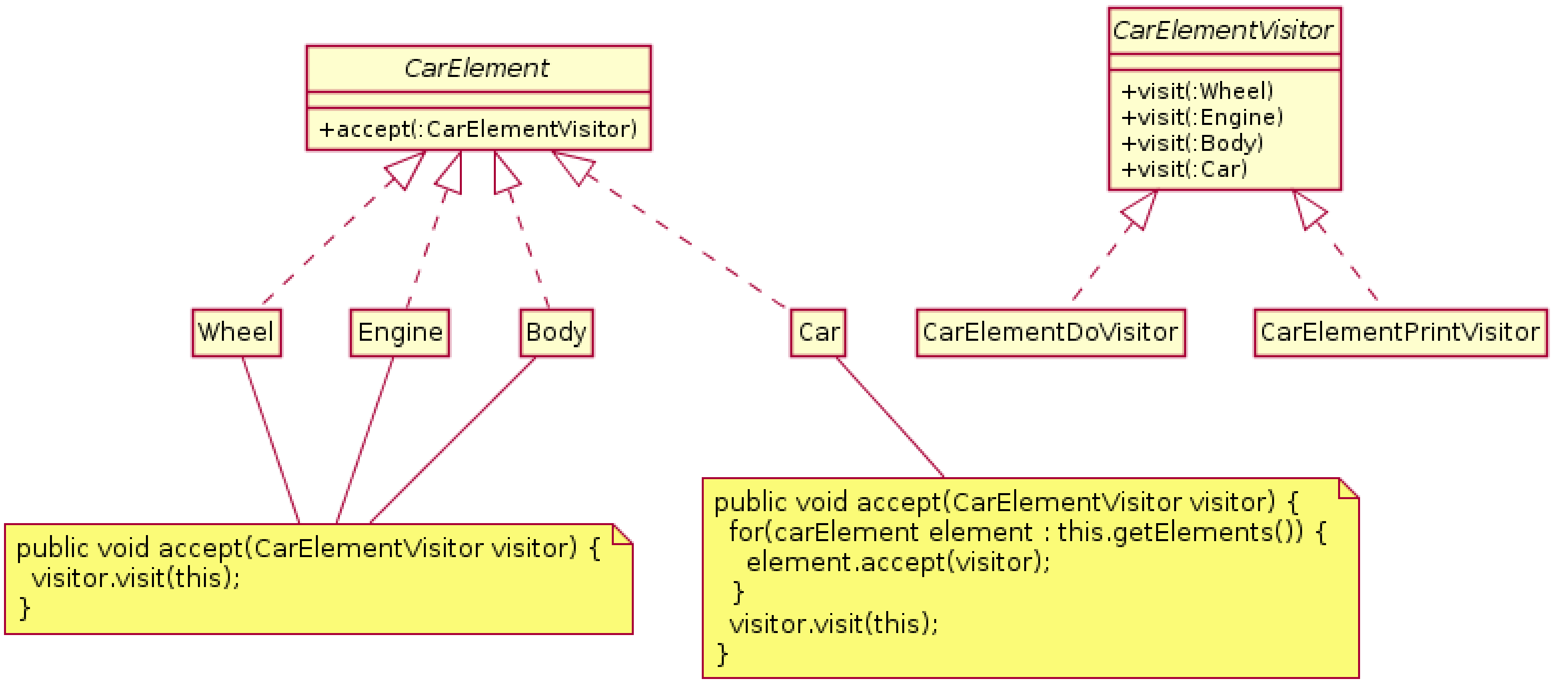
```
void postorder(Absyn tree) {
    for (int i = 0; i < tree.children.size(); i++)
       preorder(tree.children.get(i));
    process(tree);
}
```



Post-order results: E, K, L, F, B, G, H, C, I, J, D, A.

Example "Visitor" Pattern

- The "visitor" pattern creates a visitor class that implements all the specializations of the function for a family of classes



(Taken from https://en.wikipedia.org/wiki/Visitor_pattern#Java_example)

“Visitor” Pattern (2)

```
// interface and a family of classes
interface CarElement {
   void accept(CarElementVisitor visitor);
}


class Car implements CarElement {
   CarElement[] elements;

   public Car() {
       this.elements = new CarElement[] {
           new Wheel("front left"), new Wheel("front right"),
           new Wheel("back left"), new Wheel("back right"),
           new Body(), new Engine() };
   }

   public void accept(final CarElementVisitor visitor) {
       for (CarElement elem : elements) {
          elem.accept(visitor);
      }
      visitor.visit(this);
   }
}

class Body implements CarElement {
   public void accept(final CarElementVisitor visitor) { visitor.visit(this); }
}
```

“Visitor” Pattern (3)

```
class Engine implements CarElement {
   public void accept(final CarElementVisitor visitor) {
       visitor.visit(this);
   }
}

class Wheel implements CarElement {
   private String name;

   public Wheel(final String name) { this.name = name; }

   public String getName() { return name; }

   public void accept(final CarElementVisitor visitor) {
       visitor.visit(this);
   }
}


// interface and related classes for visitors
interface CarElementVisitor {
   void visit(Body body);
   void visit(Car car);
   void visit(Engine engine);
   void visit(Wheel wheel);
}
```

“Visitor” Pattern (4)

```
// The first visitor class
class CarElementDoVisitor implements CarElementVisitor {
    public void visit(final Body body) {
       System.out.println("Moving my body");
   }

   public void visit(final Car car) {
       System.out.println("Starting my car");
   }

   public void visit(final Wheel wheel) {
       System.out.println("Kicking my " + wheel.getName() + " wheel");
   }

   public void visit(final Engine engine) {
      System.out.println("Starting my engine");
   }
}
```

“Visitor” Pattern (5)

```
// The second visitor class
class CarElementPrintVisitor implements CarElementVisitor {
  public void visit(final Body body) {
    System.out.println("Visiting body");
  }

  public void visit(final Car car) {
    System.out.println("Visiting car");
  }

  public void visit(final Engine engine) {
    System.out.println("Visiting engine");
  }

  public void visit(final Wheel wheel) {
    System.out.println("Visiting " + wheel.getName() + " wheel");
  }
}
```

“Visitor” Pattern (6)

```
public class VisitorDemo {
  public static void main(final String[] args) {
     final Car car = new Car();

     car.accept(new CarElementPrintVisitor());
     car.accept(new CarElementDoVisitor());
  }
}
```

```
======================
Output:
   Visiting front left wheel
   Visiting front right wheel
   Visiting back left wheel
   Visiting back right wheel
   Visiting body
   Visiting engine
   Visiting car
   Kicking my front left wheel
   Kicking my front right wheel
   Kicking my back left wheel
   Kicking my back right wheel
   Moving my body
   Starting my engine
   Starting my car
```