

Compiler Design

Lecture 8: Abstract Syntax Tree Processing

Christophe Dubach

Winter 2025

Timestamp: 2025/01/30 16:11:00

Table of contents

AST Processing Techniques

Object-Oriented Processing

Pattern Matching

Java Example: AST Visualisation with Pattern Matching

Visitor Processing

Summary

AST Processing Techniques

AST pass

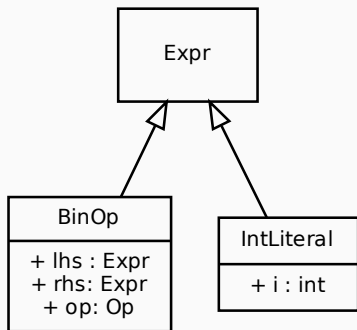
An AST pass is an action that process the AST in a single traversal.

For instance, a pass can:

- interpret the AST
- assign a type to each node of the AST
- perform an optimisation
- generate code

It is important to ensure that the different passes can access the AST in a flexible way.

Class diagram for arithmetic expressions



Naive approach: instanceof

Example: evaluating integer expressions

```
int evalExpr(Expr exp) {  
    if (exp instanceof IntLiteral) {  
        IntLiteral it = (IntLiteral) exp;  
        return (it.i);  
    }  
    else if (exp instanceof BinOp) {  
        BinOp bo = (BinOp) exp;  
        if (bo.op == ADD)  
            return(evalExpr(bo.lhs) + evalExpr(bo.rhs));  
        ...  
    } }  
}
```

⚠ What's wrong with this approach?

- a lot of boiler plate code (**instanceof** + **typecast**)
- casting is done dynamically at runtime $\Rightarrow$ no static guarantee
- no guarantee that all the type of nodes are dealt with
 - If a new type of node is added, might forget to add a case in a pass

Three Ways to Process an AST

- Object-Oriented Processing
- Pattern Matching
- Visitor Processing

AST Processing Techniques

Object-Oriented Processing

Object-Oriented Processing

Using this technique, a compiler pass is represented by a function $f()$ in each of the AST classes.

- The method is abstract if the class is abstract
- To process an instance of an AST class e , we simply call $e.f()$.
- The exact behaviour will depend on the concrete class implementations

Example for arithmetic expressions

- A function to print the AST: `String toString()`
- A function to evaluate the AST: `int eval()`

```

abstract class Expr {
    abstract String toStr();
    abstract int eval();
}

class IntLiteral extends Expr {
    int i;
    String toStr() { return ""+i; }
    int eval() { return i; }
}

class BinOp extends Expr {
    Op op;
    Expr lhs;
    Expr rhs;
    String toStr() { return lhs.toStr() + op.name() + rhs.toStr();}
    int eval() {
        switch(op) {
            case ADD: return (lhs.eval() + rhs.eval());
            case SUB: return (lhs.eval() - rhs.eval());
            case MUL: return (lhs.eval() * rhs.eval());
            case DIV: return (lhs.eval() / rhs.eval());
        }
    }
}

```

Main class

```
class Main {  
    void main(String[] args) {  
        Expr expr = ExprParser.parse(some_input_file);  
        String str = expr.toStr();  
        int result = expr.eval();  
    }  
}
```

Object Oriented Processing: Pros & Cons

😊 Pros:

- Type safety

😞 Cons:

- The logic is scattered all over the place in the various classes
- Passes are tightly integrated within the compiler IR, difficult to write standalone passes

❓ Can we do better ❓

What about the following?

Evaluation pass

```
class EvalPass {  
    int eval(IntLiteral it) {  
        return it.i;  
    }  
    int eval(BinOp bo) {  
        switch(op) {  
            case ADD: return (eval(lhs) + eval(rhs));  
            case SUB: return (eval(lhs) - eval(rhs));  
            case MUL: return (eval(lhs) * eval(rhs));  
            case DIV: return (eval(lhs) / eval(rhs));  
        }  
    }  
}
```

Main class

```
class Main {  
    void main(String[] args) {  
        Expr expr = ExprParser.parse(some_input_file);  
        int result = (new EvalPass()).eval(expr);  
    }  
}
```



Would require double-dispatch support at language level.

Detour: Single vs. double dispatch

In Java:

Single dispatch

```
class A {  
    void print() { System.out.print("A") };  
}  
class B extends A {  
    void print() { System.out.print("B") };  
}  
A a = new A();  
B b = new B();  
a.print();    // outputs A  
b.print();    // outputs B  
A b2 = new B();  
b2.print();   // output B
```

The “correct” print method is called based on the **instance type**.

Detour: Single vs. double dispatch

Double dispatch in Java?

```
class A { }  
class B extends A { }  
class Print() {  
    void print(A a) { System.out.print("A") };  
    void print(B b) { System.out.print("B") };  
}  
A a = new A();  
B b = new B();  
A b2 = new B();  
Print p = new Print();  
p.print(a); // outputs A  
p.print(b); // outputs B  
p.print(b2); // outputs A
```

Java does not support double dispatch (like many languages).

 The choice of print method is based on the argument **static type**.

⇒ Cannot use this approach.

AST Processing Techniques

Pattern Matching

Many Languages Support Pattern Matching

e.g. C#, F#, OCaml, Haskell, ML, Rust, Scala, Swift.

Eval function in Scala using pattern matching

```
def eval(e: Expr): Int =  
  e match {  
    case IntLiteral(i)      => i  
    case BinOp(lhs, rhs, ADD) => eval(lhs) + eval(rhs)  
    case BinOp(lhs, rhs, SUB) => eval(lhs) - eval(rhs)  
    case BinOp(lhs, rhs, MUL) => eval(lhs) * eval(rhs)  
    case BinOp(lhs, rhs, DIV) => eval(lhs) / eval(rhs)  
  }
```

Looks a lot like using **instanceof** in Java, without any drawback:

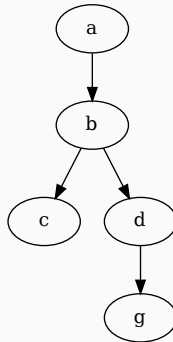
- No boiler plate code
- Static type guarantees
- Scala compiler guarantees no cases are missing

Java Example: AST Visualisation with Pattern Matching

Dot (graph description language)

Simple example dot file

```
digraph prog {  
    a → b → c;  
    b → d → g;  
}
```

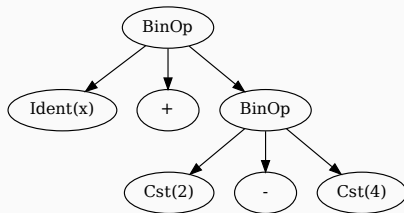


- digraph = directed graph
- prog = name of the graph (can be anything)
- to produce a pdf, simply type on Linux
`dot -Tpdf graph.dot -o graph.pdf`

Representing an AST in Dot

$x + 2 - 4$

```
digraph ast {  
  binOpNode1 [label="BinOp"];  
  idNode1 [label="Ident(x)"];  
  OpNode1 [label="+"];  
  binOpNode2 [label="BinOp"];  
  cstNode1 [label="Cst(2)"];  
  OpNode2 [label="-"];  
  cstNode2 [label="Cst(4)"];  
  
  binOpNode1 → OpNode1;  
  binOpNode1 → idNode1;  
  binOpNode1 → binOpNode2;  
  binOpNode2 → OpNode2;  
  binOpNode2 → cstNode1;  
  binOpNode2 → cstNode2;  
}
```



Pattern-Matching Implementation

Basic idea:

- Visit method writes a node and produces an edge for each child.
- Visit method returns the id of the node to the parent.

Dot Printer with Pattern Matching

```
DotPrinter {  
  
    PrintWriter writer;  
    int nodeCnt=0;  
  
    public DotPrinter(File f) {...}  
  
    String visit(ASTnode node) {  
        return switch(node) {  
            case IntLiteral i → {  
                nodeCnt++;  
                writer.println("Node"+nodeCnt+  
                    "[label=\"Cst(\"+i.value+\")\"]");  
                yield "Node"+nodeCnt;  
            }  
            ...  
        }  
    }  
}
```

Dot Printer with Pattern Matchin

```
...
case BinOp bo → {

    // write out node information
    String binOpNodeId = "Node"+nodeCnt++;
    writer.println(binOpNodeId+" [label=\"BinOp \"];");

    // traverse children
    String lhsNodeId = visit(lhs);
    String opNodeId = "Node"+nodeCnt++;
    writer.println(opNodeId+" [label=\"+ \"];");
    String rhsNodeId = visit(rhs);

    // write out edges
    writer.println(binOpNodeId + "→" lhsNodeId + ";");
    writer.println(binOpNodeId + "→" opNodeId + ";");
    writer.println(binOpNodeId + "→" rhsNodeId + ";");

    // return node id
    yield binOpNodeId;
}
...
```

Problem with Pattern Matching

It is not supported by all programming languages:

- Java only added official support in version 21, released 19 Sep 2023.
- See <https://docs.oracle.com/en/java/javase/21/language/pattern-matching.html>

⇒ need another solution if language does not offer pattern matching.

Visitor Processing

Visitor Processing

With this technique, all the methods from a pass are grouped in a **visitor**.

It only relies on single dispatch mechanism:

- the method is chosen based on the dynamic type of the object (the AST node)

The **visitor design pattern** allows us to implement double dispatch, the method is chosen based on:

- the dynamic type of the object (the AST node)
- the dynamic type of the argument (the visitor)

- The logic to handle each type of node is split up in separate functions in a single class: the **Visitor**.
- Each AST node class implements an **accept** method which dispatches the processing of that node to the right function in the visitor.

Visitor Interface

```
interface Visitor<T> {  
    T visitIntLiteral(IntLiteral il);  
    T visitBinOp(BinOp bo);  
}
```

Modified AST classes

```
abstract class Expr {  
    abstract <T> T accept(Visitor<T> v);  
}  
  
class IntLiteral extends Expr {  
    ...  
    <T> T accept(Visitor<T> v) {  
        return v.visitIntLiteral(this);  
    }  
}  
  
class BinOp extends Expr {  
    ...  
    <T> T accept(Visitor<T> v) {  
        return v.visitBinOp(this);  
    }  
}
```

Examples

ToStr Visitor

```
ToStr implements Visitor<String> {  
    String visitIntLiteral(IntLiteral il) {  
        return ""+il.i;  
    }  
    String visitBinOp(BinOp bo) {  
        return bo.lhs.accept(this) + bo.op.name() + bo.rhs.accept(this);  
    }  
}
```

Eval Visitor

```
Eval implements Visitor<Integer> {  
    Integer visitIntLiteral(IntLiteral il) {  
        return il.i;  
    }  
    Integer visitBinOp(BinOp bo) {  
        switch(bo.op) {  
            case ADD: lhs.accept(this) + rhs.accept(this); break;  
            case SUB: lhs.accept(this) - rhs.accept(this); break;  
            case MUL: lhs.accept(this) * rhs.accept(this); break;  
            case DIV: lhs.accept(this) / rhs.accept(this); break;  
        }  
    }  
}
```

Main class

```
class Main {  
    void main(String[] args) {  
        Expr expr = ExprParser.parse(some_input_file);  
        String str = expr.accept(new ToStr());  
        int result = expr.accept(new Eval());  
    }  
}
```

Exercise

Trace all functions called when the `Eval` visitor processes $2 * 3$.

Eval Visitor Class

```
Eval implements Visitor<Integer> {  
    Integer visitIntLiteral(IntLiteral il) { return il.i; }  
    Integer visitBinOp(BinOp bo) {  
        switch(bo.op) {  
            case ADD: lhs.accept(this) + rhs.accept(this); break;  
            case SUB: lhs.accept(this) - rhs.accept(this); break;  
            case MUL: lhs.accept(this) * rhs.accept(this); break;  
            case DIV: lhs.accept(this) / rhs.accept(this); break; } }  
}
```

AST Nodes Class

```
class IntLiteral extends Expr {  
    ...  
    <T> T accept(Visitor<T> v) {  
        return v.visitIntLiteral(this); } }  
  
class BinOp extends Expr {  
    ...  
    <T> T accept(Visitor<T> v) {  
        return v.visitBinOp(this); } }
```

Summary

With an AST, there can extensions in two dimensions:

1. Adding a **new AST node**

- For object-oriented processing this means add a new sub-class
- For pattern matching, add a new case in each pass
- For visitors, add a new method in every visitor

2. Adding a **new pass**

- For object-oriented processing, this means adding a function in every single AST node classes
- For visitor/pattern-matching, simply create a new class for the pass

Picking the right design

Facilitate **extensibility**:

- object-oriented design: easy to add new type of AST node
- visitor-based/pattern matching: easy to write new passes
- Which one do you think is more common?

Facilitate **modularity**:

- the object-oriented design allows for code and data to be stored in the AST node and be shared between phases (e.g. types)
- visitor / pattern matching allows for code and data to be shared among the methods of the same pass

- Context-sensitive Analysis