

# Compiling Techniques

## Lecture 7: Abstract Syntax

Christophe Dubach

1 October 2019

# Table of contents

- 1 Syntax Tree
  - Semantic Actions
  - Examples
  - Abstract Grammar
- 2 Abstract Syntax Tree
  - Internal Representation
  - AST Builder
- 3 AST Processing
  - Object-Oriented Processing
  - Visitor Processing
  - AST Visualisation

A parser does more than simply recognising syntax. It can:

- evaluate code (interpreter)
- emit code (simple compiler)
- build an internal representation of the program (multi-pass compiler)

In general, a parser performs semantic actions:

- **recursive descent parser**: integrate the actions with the parsing functions
- **bottom-up parser** (automatically generated): add actions to the grammar

# Syntax Tree

In a multi-pass compiler, the parser builds a syntax tree which is used by the subsequent passes

A syntax tree can be:

- a **concrete syntax tree** (or parse tree) if it directly corresponds to the context-free grammar
- an **abstract syntax tree** if it corresponds to a simplified (or abstract) grammar

The abstract syntax tree (AST) is usually used in compilers.

## Example: Concrete Syntax Tree (Parse Tree)

### Example: CFG for arithmetic expressions (EBNF form)

```
Expr ::= Term ( ('+' | '-') Term )*
Term  ::= Factor ( ('*' | '/') Factor )*
Factor ::= number | '(' Expr ')'
```

### After removal of EBNF syntax

```
Expr ::= Term Terms
Terms ::= ('+' | '-') Term Terms | ε
Term  ::= Factor Factors
Factors ::= ('*' | '/') Factor Factors | ε
Factor ::= number | '(' Expr ')'
```

### After further simplification

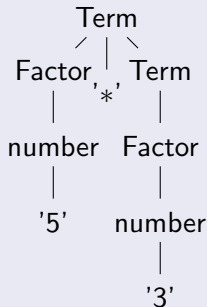
```
Expr ::= Term (('+' | '-') Expr | ε)
Term  ::= Factor (('*' | '/') Term | ε)
Factor ::= number | '(' Expr ')'
```

# Example: Concrete Syntax Tree (Parse Tree)

## CFG for arithmetic expressions

$\text{Expr} ::= \text{Term} (('+' | '-' ) \text{Expr} | \epsilon)$   
 $\text{Term} ::= \text{Factor} (('*' | '/' ) \text{Term} | \epsilon)$   
 $\text{Factor} ::= \text{number} | '(' \text{Expr} ')'$

## Concrete Syntax Tree for $5 * 3$



The concrete syntax tree contains a lot of unnecessary information.

It is possible to simplify the concrete syntax tree to remove the redundant information.

For instance parenthesis are not necessary.

### Exercise

- 1 Write the concrete syntax tree for  $3 * (4 + 5)$
- 2 Simplify the tree.

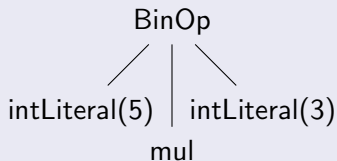
# Abstract Grammar

These simplifications leads to a new simpler context-free grammar called **Abstract Grammar**.

Example: abstract grammar for arithmetic expressions

```
Expr ::= BinOp | intLiteral
BinOp ::= Expr Op Expr
Op ::= add | sub | mul | div
```

5 \* 3



This is called an **Abstract Syntax Tree**



## Example: abstract grammar for arithmetic expressions

```
Expr ::= BinOp | intLiteral  
BinOp ::= Expr Op Expr  
Op ::= add | sub | mul | div
```

Note that for given concrete grammar, there exist numerous abstract grammar:

```
Expr ::= AddOp | SubOp | MulOp | DivOp | intLiteral  
AddOp ::= Expr add Expr  
SubOp ::= Expr sub Expr  
MulOp ::= Expr mul Expr  
DivOp ::= Expr div Expr
```

We pick the most suitable grammar for the compiler.

# Abstract Syntax Tree

The Abstract Syntax Tree (AST) forms the main intermediate representation of the compiler's front-end.

For each non-terminal or terminal in the abstract grammar, we define a class.

- If a non-terminal has any alternative on the rhs (right hand side), then the class is abstract (cannot instantiate it). The terminal or non-terminal appearing on the rhs are subclasses of the non-terminal on the lhs.
- The sub-trees are represented as instance variable in the class.
- Each non-abstract class has a unique constructor.
- If a terminal does not store any information, then we can use an Enum type in Java instead of a class.

## Example: abstract grammar for arithmetic expressions

```
Expr ::= BinOp | intLiteral  
BinOp ::= Expr Op Expr  
Op ::= add | sub | mul | div
```

## Corresponding Java Classes

```
abstract class Expr { }  
  
class IntLiteral extends Expr {  
    int i;  
    IntLiteral(int i){...}  
}  
  
class BinOp extends Expr {  
    Op op;  
    Expr lhs;  
    Expr rhs;  
    BinOp(Op op, Expr lhs, Expr rhs) {...}  
}  
  
enum Op {ADD, SUB, MUL, DIV}
```

## CFG for arithmetic expressions

```
Expr ::= Term (('+' | '-' ) Expr |  $\epsilon$ )  
Term ::= Factor (('*' | '/' ) Term |  $\epsilon$ )  
Factor ::= number | '(' Expr ')'
```

## Current Parser (class)

```
void parseExpr () {  
    parseTerm ();  
    if (accept (PLUS | MINUS))  
        nextToken ();  
    parseExpr ();  
}  
  
void parseTerm () {  
    parseFactor ();  
    if (accept (TIMES | DIV))  
        nextToken ();  
    parseTerm ();  
}  
  
void parseFactor () {  
    if (accept (LPAR))  
        parseExpr ();  
    expect (RPAR);  
    else  
        expect (NUMBER);  
}
```

## Current Parser

```
void parseExpr() {  
    parseTerm();  
    if (accept(PLUS|MINUS))  
        nextToken();  
    parseExpr();  
}
```

## AST building (modified Parser)

```
Expr parseExpr() {  
    Expr lhs = parseTerm();  
    if (accept(PLUS|MINUS))  
        Op op;  
        if (token == PLUS)  
            op = ADD;  
        else // token == MINUS  
            op = SUB;  
        nextToken();  
    Expr rhs = parseExpr();  
    return new BinOp(op, lhs, rhs);  
    return lhs;  
}
```

## Current Parser

```
void parseTerm() {  
    parseFactor();  
    if (accept(TIMES|DIV))  
        nextToken();  
    parseTerm();  
}
```

## AST building (modified Parser)

```
Expr parseTerm() {  
    Expr lhs = parseFactor();  
    if (accept(TIMES|DIV))  
        Op op;  
        if (token == TIMES)  
            op = MUL;  
        else // token == DIV  
            op = DIV;  
        nextToken();  
        Expr rhs = parseTerm();  
        return new BinOp(op, lhs, rhs);  
    return lhs;  
}
```

## Current Parser

```
void parseFactor() {  
    if (accept(LPAR))  
        parseExpr();  
    expect(RPAR);  
    else  
        expect(NUMBER);  
}
```

## AST building (modified Parser)

```
Expr parseFactor() {  
    if (accept(LPAR))  
        Expr e = parseExpr();  
    expect(RPAR);  
    return e;  
    else  
        IntLiteral il = parseNumber();  
    return il;  
}  
  
IntLiteral parseNumber() {  
    Token n = expect(NUMBER);  
    int i = Integer.parseInt(n.data);  
    return new IntLiteral(i);  
}
```

# Compiler Pass

## AST pass

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

A pass can for instance:

- 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. An inefficient solution would be to use `instanceof` to find the type of syntax node

## Example

```
if (tree instanceof IntLiteral)
    ((IntLiteral)tree).i;
```



# Two Ways to Process an AST

- Object-Oriented Processing
- Visitor 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 the arithmetic expression

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

```
abstract class Expr {
    abstract String toString();
    abstract int eval();
}
class IntLiteral extends Expr {
    int i;
    String toString() { return ""+i; }
    int eval() { return i; }
}
class BinOp extends Expr {
    Op op;
    Expr lhs;
    Expr rhs;
    String toString() { return lhs.toString() + op.name() + rhs.toString(); }
    int eval() {
        switch(op) {
            case ADD: lhs.eval() + rhs.eval(); break;
            case SUB: lhs.eval() - rhs.eval(); break;
            case MUL: lhs.eval() * rhs.eval(); break;
            case DIV: lhs.eval() / rhs.eval(); break;
        }
    }
}
```

## Main class

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

# Visitor Processing

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

For this, need a language that implements single dispatch:

- 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)

Note that if the language supports pattern matching, it is not needed to use a visitor since double-dispatch can be implemented more effectively.

# 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
```

# Single vs. double dispatch

In Java:

Double dispatch (Java does not support double dispatch)

```
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
```

## 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);  
    }  
}
```



## 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());  
    }  
}
```

# Extensibility

With an AST, there can extensions in two dimensions:

- 1 Adding a **new AST node**
  - For the object-oriented processing this means add a new sub-class
  - In the case of the visitor, need to add a new method in every visitor
- 2 Adding a **new pass**
  - For the object-oriented processing, this means adding a function in every single AST node classes
  - For the visitor case, simply create a new visitor

## Picking the right design

Facilitate **extensibility**:

- the object-oriented design makes it easy to add new type of AST node
- the visitor-based scheme makes it easy to write new passes

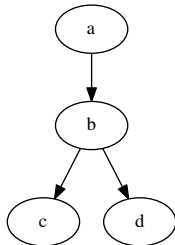
Facilitate **modularity**:

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

# Dot (graph description language)

## Simple example dot file

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

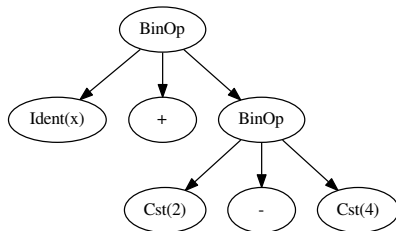


- digraph = directed graph
- prog = name of the graph (can be anything)
- to produce a pdf, simply type on linux  
dot -Tpdf graph.dot -o graph.pdf

# 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;  
}
```



## Dot Printer Visitor

```
DotPrinter implements Visitor<String> {  
    PrintWriter writer;  
    int nodeCnt=0;  
  
    public DotPrinter(File f) {...}  
  
    String visitIntLiteral(IntLiteral il) {  
        nodeCnt++;  
        writer.println("Node"+nodeCnt+  
            "[label=\"Cst(\"+il.value+\" )\"]");  
        return "Node"+nodeCnt;  
    }  
  
    ...  
}
```

## Dot Printer Visitor

```
...
String visitBinOp(BinOp bo) {
    String binOpNodeId = "Node"+nodeCnt++;
    writer.println(binOpNodeId+" [label=\"BinOp\"]");

    String lhsNodeId = lhs.accept(this);
    String opNodeId = "Node"+nodeCnt++;
    writer.println(opNodeId+" [label=\"+\"]");
    String rhsNodeId = rhs.accept(this);

    writer.println(binOpNodeId + " →" lhsNodeId + ";");
    writer.println(binOpNodeId + " →" opNodeId + ";");
    writer.println(binOpNodeId + " →" rhsNodeId + ";");

    return binOpNodeId;
}
...
}
```



## Next lecture

- Context-sensitive Analysis