

# Compiler Construction

~ AST in C++ ~

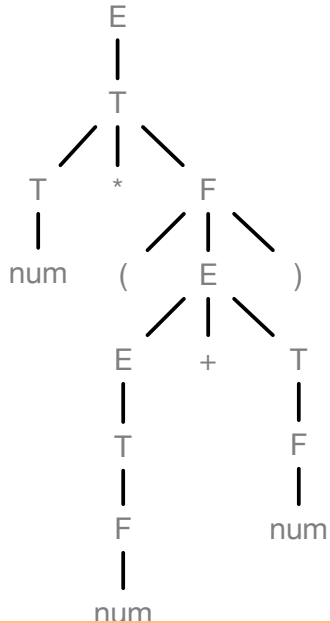
# Simple Unambiguous Concrete Grammar (BNF)

```
<E> ::= <E> "+" <T>
      | <E> "-" <T>
      | <T>
<T> ::= <T> "*" <F>
      | <T> "/" <F>
      | <F>
<F> ::= "(" <E> ")"
      | <num>
```

- **E** stands for expression
- **T** stands for term
- **F** stands for factor

Conventionally, terms are things you add, factor are things you multiply

# Parse Tree: $1*(2+3)$



# A need for abstraction

**Parse trees** are not adapted to writing the whole compiler with them only (even if we could):

- they retain syntactic information (punctuation symbols, ...)
- they reflect the underlying grammar
- they have redundant information: parenthesis and tree child for instance

All these elements could clutter the semantic analysis!

# Abstract Grammar (RTG)

## RTG

A regular tree grammar (RTG) is a formal grammar used to describe trees

```
<exp> ::= Add(<exp>, <exp>)  
        | Sub(<exp>, <exp>)  
        | Mul(<exp>, <exp>)  
        | Div(<exp>, <exp>)  
        | Num(<num>)
```

# AST

## Abstract Syntax Tree

An AST is the translation of the parse tree in order to match the abstract grammar.

## From Parse trees to AST

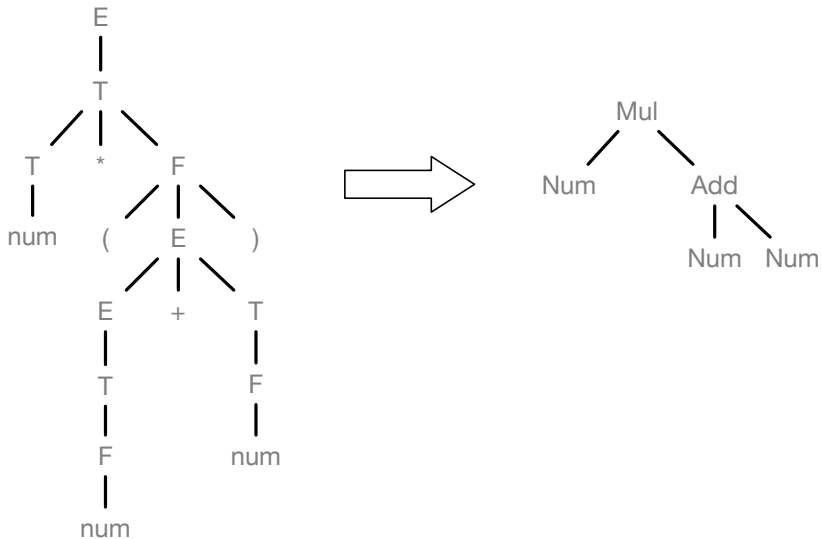
The translation from parse tree to AST is straightforward!

⇒ Just use production rules of the parser

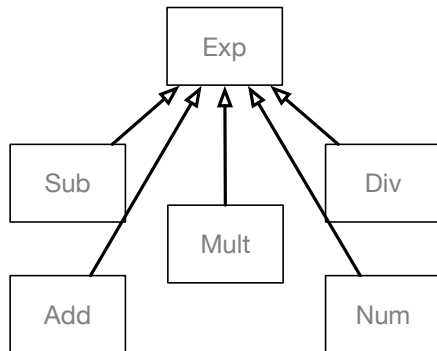
exp:

```
exp "+" term { $$ = new Add($1, $3); }
```

# From Parse tree to AST



# First AST Hierarchy

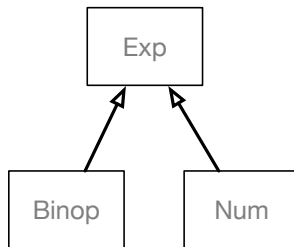


**BAD OO Design!**

Sub, Add, Div and Mult share a lot !  
⇒ Refactor it!



# Refined AST Hierarchy



## C++ code (1/3)

```
class Exp
{
protected:
    // default constructor
    Exp() = default;

    // default copy-constructor
    Exp(const Exp& e) = default;

    // default assign operator
    Exp& operator=(const Exp& e) = default;

public:
    virtual ~Exp();
};
```

## C++ code (2/3)

```
class Num : public Exp
{
public:
    Num(int val)
        : Exp(), val_(val)
    {}

private:
    int val_;
};
```

## C++ code (3/3)

```
class Binop : public Exp
{
public:
    Binop(char oper, Exp* lhs, Exp* rhs)
        : Exp(), oper_(oper), lhs_(lhs), rhs_(rhs)
    {}

    ~Binop() override {
        delete lhs_; delete rhs_;
    }

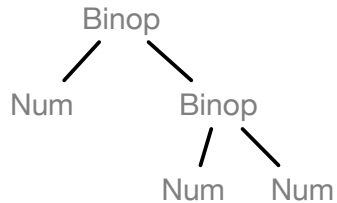
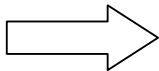
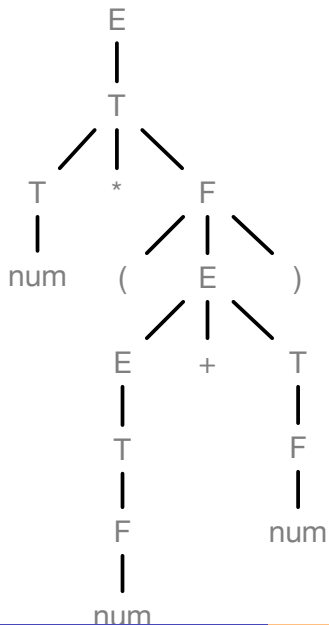
private:
    char oper_; // For simplicity, in reality use an enum
    Exp* lhs_; Exp* rhs_;
};
```

# Constructing an AST

```
int main()
{
    Exp* tree = new Bin(
        '+',
        new Num(42),
        new Num(51)
    );
    delete tree;
}
```

Memory reclamation is (transitively) done through destructors.  
Explicit (and recursive) `delete` instructions! Maybe use smart pointers?

# From Parse tree to refined AST



# Summary

Concrete  
grammar

Abstract  
grammar

Parse tree

AST