# Compiler Project

## Syntax Analysis

Florent De Geeter & Pascal Fontaine

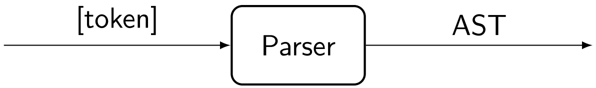
University of Liège

February 27, 2024

## Outline

## Outline

## Syntax Analysis



Converts a stream of tokens into an **abstract syntax tree** (AST)

Detects **syntactically invalid** source code

`vsopc -p example.vsop` should:

- If the source file is syntactically valid, return 0 and print on `stdout` the abstract syntax tree following the format given in the statement
- Otherwise, return a non-zero value and print on `stderr` (at least) one syntax or lexical error

## Assignment

Due March 20, 2024

Automated tests worth 5% of your grade

You can use a **parser generator** (e.g. bison, PLY, ANTLR)

Support for **custom tests** in `tests` subfolder

**Two modes**: `-p` and `-l`

## Output Format

```
[Class(List, Object, [],
       [Method(isNil, [], bool, true),
        Method(length, [], int32, 0)]),
 Class(Nil, List, [], []),
 Class(Cons, List, [Field(head, int32),
                    Field(tail, List)],
       [Method(init, [hd : int32, tl : List], Cons,
               [Assign(head, hd),
                Assign(tail, tl), self]),
        Method(head, [], int32, head),
        Method(isNil, [], bool, false),
        Method(length, [], int32,
               BinOp(+, 1, Call(tail, length, [])))]),
 ...
```

## Error Management

Error messages on `stderr`, fail with code $\neq 0$

```
input_file.vsop:4:12: syntax error: description
```

Tests do not check the positions

Automated tests don't check the description, but **we do** !

Syntax error reporting is **challenging**, see lectures !

Lexical errors can still happen !

## Conflicts

If your parser generator creates bottom-up parsers, they may face:

- **shift/reduce** conflicts
- **reduce/reduce** conflicts

Try to solve them, especially for the final deadline of the project (in May)

You will be penalized if you don't !

## Questions and (Possibly) Answers

?

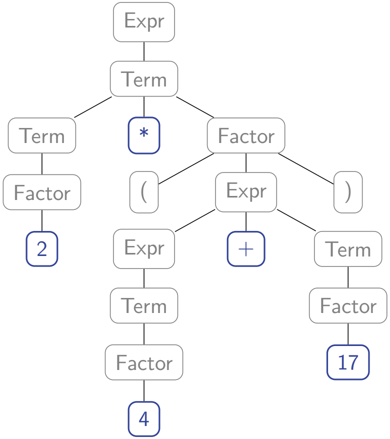
## Outline

# A Concrete Parse Tree is Very Redundant

A parse tree represents the **syntactic structure** of a string (which can be a mathematical expression, a source code, etc)

But it is very **redundant**

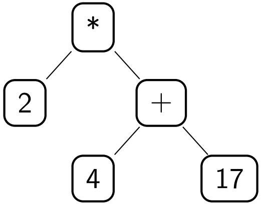
E.g., for `2 * (4 + 17)`:

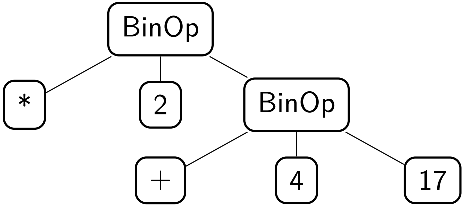


## Use an Abstract Syntax Tree (AST)

An abstract parse tree **does not represent** all the details, but keep the important ones

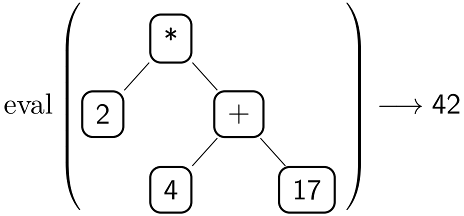
E.g., for `2 * (4 + 17)`:





## Processing Abstract Syntax Trees

Example: evaluating arithmetic expressions



In your compiler:

- Generate the tree (**syntax analysis**) and print it (AST dump)
- Annotate it with types (**semantic analysis**)
- Generate code from it (**code generation**)

## Outline

## Avoid Generic Trees

```
class ASTNode {
    private List<ASTNode> children;
    private String value;
    ...
    public double eval() {
        if (children.length() == 3) {
            if (children.get(1).value == "+") {
                return children.get(0).eval()
                    + children.get(2).eval();
            } else if ...
```

Seems economical, but **hard to read** and **error-prone**

## OO Inheritance-based Approach

```
abstract class Expr { public abstract double eval(); }
class Add extends Expr {
    Expr lhs;
    Expr rhs;
    ...
    public double eval() {
        return lhs.eval() + rhs.eval();
    }
}
class Sub extends Expr { ... }
```

Simple, good encapsulation, easy to add new nodes

Logic of a pass spread across many classes, **Inheritance has a cost**

## Functional Approach

```
data Expr = Num Double | Add Expr Expr | Sub Expr Expr
          | ...

eval (Num num) = num
eval (Add lhs rhs) = eval lhs + eval rhs
eval (Sub lhs rhs) = eval lhs - eval rhs
...
```

Pass logic in a single file: Easier to add passes, to read

No information hiding, less flexible than OO

## Simulate Functional with Tagged Unions: Types

```
typedef enum { NUM, ADD, SUB, ... } Tag;
typedef struct Expr Expr;
typedef struct { double value } Num;
typedef struct { const Expr *lhs; const Expr *rhs; } Add;
...
struct Expr {
    Tag tag;
    union {
        Num num;
        Add add;
        Sub sub;
        ...
    };
};
```

Only one structure for all types of nodes

## Simulate Functional with Tagged Unions: Constructors

```
static Expr *new_expr(Tag tag) {
    Expr *ret = checked_malloc(sizeof(Expr));
    ret->tag = tag;
    return ret;
}
Expr *new_num(double value) {
    Expr *ret = new_expr(NUM);
    ret->num.value = value;
    return ret;
}
Expr *new_add(const Expr *lhs, const Expr *rhs) {
    Expr *ret = new_expr(ADD);
    ret->add.lhs = lhs;
    ret->add.rhs = rhs;
    return ret;
}
```

## Simulate Functional with Tagged Unions: Use

```
double eval(const Expr *e) {
    switch (e->tag) {
        case NUM:
            return e->num.value;
        case ADD:
            return eval(e->add.lhs) + eval(e->add.rhs);
        case SUB:
            return eval(e->sub.lhs) - eval(e->sub.rhs);
        ...
    }
}
```

Simple and efficient, no inheritance

**Not type-safe** (every type in single class)

## Approach with Introspection is Generally Slow

```
double eval(Expr e) {
    if (e instanceof Num) {
        return ((Num) e).getValue();
    } else if (e instanceof Add) {
        Add add = (Add) e;
        return eval(add.getLeft()) + eval(add.getRight());
    } else if (e instanceof Sub) {
        ...
```

Java's `instanceof` operator is expensive

Long chain of `if-else if` $\Longrightarrow \mathcal{O}(n)$ checks per node!

Except in selected languages (e.g. Darts)

## The Visitor Design Pattern

Idea: **Separate** the code that **stores the data** from the code that **manipulates it**

- **Data classes** store the data and implements the `accept` methods that allow a visitor to manipulate their data
- **Visitor classes** perform operations on the data classes: they implement a `visit` method for each type of data class

## The Visitor Design Pattern: Interface

```
interface Visitor<R> {
    public R visit(Num num);
    public R visit(Add add);
    public R visit(Sub sub);
    ...
}
```

An `interface` with one `visit` method per AST node.

Parameterized over return type.

## The Visitor Design Pattern: `accept()`

```
abstract class Expr {
    abstract public <R> R accept(Visitor<R> v);
}
class Num extends Expr {
    public <R> R accept(Visitor<R> v) {
        return v.visit(this);
    }
    ...
}
class Add extends Expr {
    public <R> R accept(Visitor<R> v) {
        return v.visit(this);
    }
    ...
}
```

## The Visitor Design Pattern: Use

```
class EvalVisitor implements Visitor<Double> {
    public Double visit(Num num) {
        return num.getValue();
    }
    public Double visit(Add add) {
        return add.lhs.accept(this) + add.rhs.accept(this);
    }
    public Double visit(Sub sub) {
        return sub.lhs.accept(this) - sub.rhs.accept(this);
    }
    ...
}
```

## The Visitor Design Pattern: Use

```
class EvalVisitor implements Visitor<Double> {
    public Double visit(Num num) {
        return num.getValue();
    }
    public Double visit(Add add) {
        return add.lhs.accept(this) + add.rhs.accept(this);
    }
    public Double visit(Sub sub) {
        return sub.lhs.accept(this) - sub.rhs.accept(this);
    }
    ...
}
```

Functional, type-safe, double dispatch faster than introspection

Little boilerplate, slightly heavy syntactically

## Questions and (Possibly) Answers

?