INSTITUTO MAUÁ DE TECNOLOGIA

MAUÁ

ECM253 – Linguagens Formais, Autômatos e Compiladores

Projeto

Analisador CUP com árvore AST

Marco Furlan

Outubro de 2024

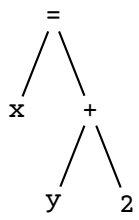
1 Árvore AST

Uma **árvore AST** (*Abstract Syntax Tree*) permite armazenar **elementos essenciais** que, posteriormente, podem ser **utilizados para gerar ações semânticas**. Neste caso, em particular, uma AST é importante, pois é uma **forma de conter** todo (ou parte de, dependendo do projetista) **o programa analisado** para **posterior geração de código**, permitindo a manipulação dos elementos sintáticos de uma determinada gramática.

Por exemplo, a expressão:

```
x = y + 2;
```

Pode ser **representada** pela seguinte **árvore AST**:



Notar as **diferenças** entre uma **árvore de análise sintática** e uma **AST**:

- **Árvore de análise sintática**:
  - **Usada principalmente** para **verificar** se o **código segue** as **regras gramaticais da linguagem**. Ela é gerada na fase de análise sintática do compilador.
  - **Geralmente maior** que uma **AST**, pois contém **mais informações**, incluindo **nós** para cada **regra gramatical aplicada**.
- **AST**:
  - Usada na **fase de análise semântica e otimização** do compilador. Ela serve como **base para gerar código intermediário** ou **código de máquina**, capturando a **estrutura lógica do programa**.
  - É **menor e mais simples** do que uma árvore de análise sintática, com menos nós, pois **omite as regras gramaticais** que **não** são **relevantes** para a **semântica do código**.

Em uma **árvore AST**, é comum que a **raiz** de toda **subárvore represente** algum tipo de **operador** ou **comando**, enquanto que os **filhos representem operandos** ou **expressões**. Assim, uma AST **pode conter diferentes tipos de nós**, como:

- **Declarações**: declaração de variáveis, atribuições, chamadas de função.
- **Expressões**: Operadores e operandos, como em operações aritméticas ou lógicas.
- **Blocos de controle**: Estruturas condicionais (if, while, etc.).

2 O padrão de projeto Visitor

Uma forma de **implementar árvores AST** em **Java** é utilizar como base o **padrão de projeto**[1] **Visitor** (ou visitante).

O padrão de projeto **Visitor** é um **padrão comportamental** que permite **adicionar** novas **operações** a **objetos** de uma **estrutura sem alterar as classes desses objetos**. Ele é **útil** quando se tem uma **estrutura de objetos complexa** e deseja **executar operações específicas** em **vários objetos**, mas não quer modificar suas classes cada vez que uma nova operação for necessária.

Elementos principais do padrão Visitor:

- **`Visitor`** (Visitante): Uma **interface** ou **classe abstrata** que declara uma **operação** de **visita** para cada **tipo** de **elemento** concreto na **estrutura**. Cada **método** de **visita** é **sobrecarregado** para **diferentes tipos** de **objetos** que o visitante pode encontrar.
- **`ConcreteVisitor`** (Visitante Concreto): **Implementa** as **operações** definidas pelo **visitante**, aplicando a **lógica** necessária **para cada tipo** de objeto.
- **`Element`** (Elemento): Uma **interface** ou **classe abstrata** que **define** um **método** `accept(Visitor)` para **aceitar** o **visitante**.
- **`ConcreteElement`** (Elemento Concreto): **Classes concretas** que **implementam** a **interface** `Element`. Elas implementam o **método** `accept()`, **permitindo** que o **visitante** realize **operações** sobre **elas**.

[1] <https://en.wikipedia.org/wiki/Software_design_pattern>

- **`Object Structure`** (Estrutura de Objetos): Uma **coleção** de **elementos**. Pode ser uma **árvore de objetos** ou uma **lista** que permite ao **visitante percorrer** e **operar** em todos os **elementos**.

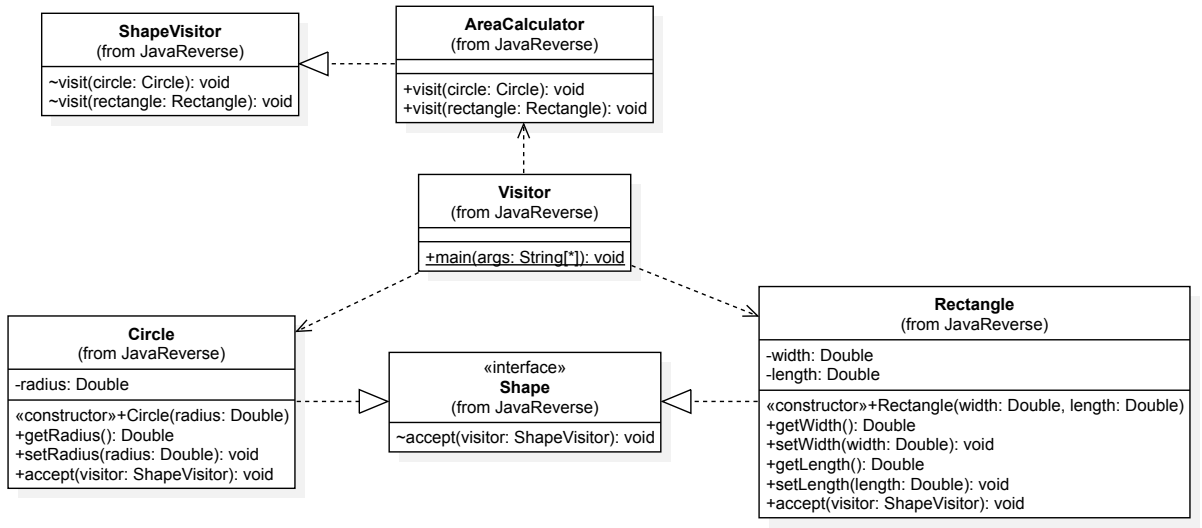
Funcionamento:

- O **objeto visitante** é **passado** para os **elementos** da **estrutura**. Cada **elemento chama** o método **adequado** do **visitante** (com base no tipo do elemento), e o **visitante executa** a **operação** necessária **sem** que os **elementos** precisem **saber** os **detalhes** da **operação**.
- Isso **separa a lógica da operação da estrutura dos objetos**. Se houver necessidade de **adicionar novas operações**, **não será necessário modificar os elementos**; basta criar uma nova implementação do visitante.

Exemplo prático:

- Imaginar uma **estrutura** de objetos que representa **figuras geométricas** (círculo, quadrado, retângulo).
- **Deseja-se calcular a área de cada uma dessas figuras** ou exportá-las para um formato de arquivo.
- **Em vez de adicionar essas operações nas classes de cada figura** (quebrando o princípio de responsabilidade única), você pode **implementar visitantes** para cada **operação**.

O diagrama de **classes UML** para este **exemplo** está apresentado a seguir:



E a implementação em Java do exemplo está apresentada a seguir:

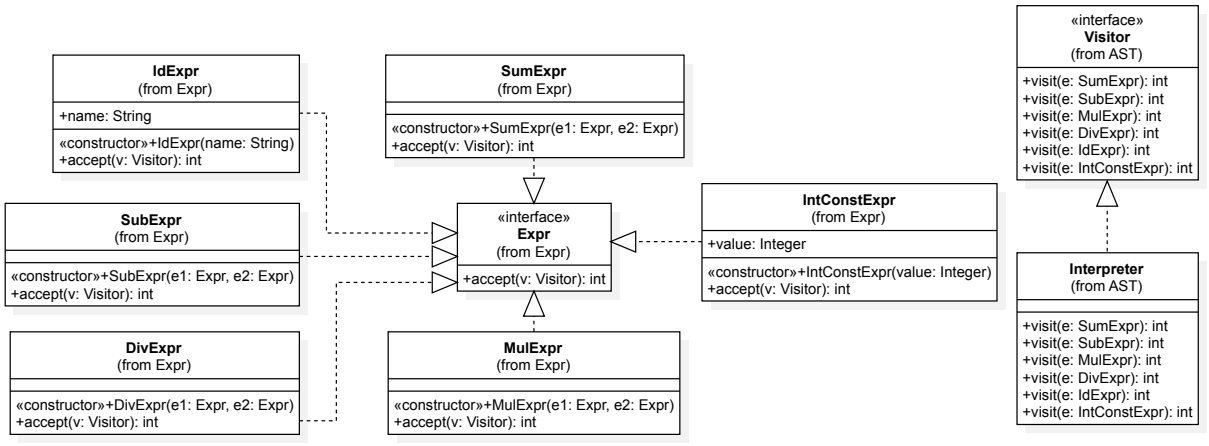
```java
import java.util.ArrayList;

public class Visitor {
    public static void main(String[] args) {
        // list of shapes
        ArrayList<Shape> shapes = new ArrayList<>();
        // ass some shapes
```

```
        shapes.add(new Circle(3.0));
        shapes.add(new Rectangle(5.0, 4.0));
        // create a visitor
        AreaCalculator calculator = new AreaCalculator();
        // visit all the shapes
        for (Shape s : shapes) {
            s.accept(calculator);
        }
    }
}

// Interface Visitor
interface ShapeVisitor {
    void visit(Circle circle);

    void visit(Rectangle rectangle);
}

// Interface Element
interface Shape {
    void accept(ShapeVisitor visitor);
}

// Implementação de um elemento concreto (Círculo)
class Circle implements Shape {

    public Circle(Double radius) {
        this.radius = radius;
    }

    private Double radius;

    public Double getRadius() {
        return this.radius;
    }

    public void setRadius(Double radius) {
        this.radius = radius;
    }

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

// Implementação de outro elemento concreto (Retângulo)
class Rectangle implements Shape {

    public Rectangle(Double width, Double length) {
        this.width = width;
        this.length = length;
    }

    private Double width;
```

```
    private Double length;

    public Double getWidth() {
        return this.width;
    }

    public void setWidth(Double width) {
        this.width = width;
    }

    public Double getLength() {
        return this.length;
    }

    public void setLength(Double length) {
        this.length = length;
    }

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

// Implementação do visitante concreto para cálculo de área
class AreaCalculator implements ShapeVisitor {
    public void visit(Circle circle) {
        // lógica de cálculo de área do círculo
        System.out.println(Math.PI * Math.pow(circle.getRadius(), 2));
    }

    public void visit(Rectangle rectangle) {
        // lógica de cálculo de área do retângulo
        System.out.println(rectangle.getLength() * rectangle.getWidth());
    }
}
```

3 Implementação de AST em Java com Visitor

Para implementar uma **árvore AST** com **Visitor**, pode-se proceder a organização de classes apresentada a seguir, em **UML**:



A **ideia**, neste caso, é **tratar** um **interpretador** ou **compilador** (ou qualquer outro elemento que manipulará um programa) como uma **implementação** de **Visitor**. As funções da **interface Visitor** são aquelas que "**visitarão**" cada **elemento** da **AST**, **efetuando ações semânticas** sobre os mesmos, separadamente, por elemento

A **AST** será **criada** durante a **análise sintática**, no caso, pelo **CUP**. Aqui, para melhor entendimento, será criada uma árvore manualmente. Supor que se deseja realizar a **seguinte operação**: `10 * (5-8) + 4`. Utilizando objetos das classes apresentadas, ela poderia ser assim implementada (**código completo** no arquivo `AST Visitor.zip`, no Canvas):

```
import AST.Expr.*;
import AST.Interpreter;

public class App {
    public static void main(String[] args) throws Exception {
        // 10 * (5-8) + 4
        SumExpr e = new SumExpr(
            new MulExpr(
                new IntConstExpr(10),
                new SubExpr(
                    new IntConstExpr(5),
                    new IntConstExpr(8))),
            new IntConstExpr(4));
        System.out.println(new Interpreter().visit(e));
    }
}
```

A **árvore AST** é **criada** por meio das **referências** para **objetos** que serão **visitados** (neste exemplo – **expressões**). Um objeto de **expressão de soma**, por exemplo, tem **duas referências** para seus **operandos** (que são expressões), como apresentado no código da **classe** `SumExpr` apresentado a seguir:

```
package AST.Expr;
```

```
import AST.Visitor;

public class SumExpr implements Expr {
    public Expr e1, e2;

    public SumExpr(Expr e1, Expr e2) {
        this.e1 = e1;
        this.e2 = e2;
    }

    @Override
    public int accept(Visitor v) {
        return v.visit(this);
    }

}
```

Desse modo, apenas as folhas (neste exemplo, objetos das classes `IdExpr` e `IntConstExpr`) não terão referências para outros objetos.

Como o interpretador visita a árvore formada? Isso **ocorre aqui**: `new Interpreter().visit(e)`. Quando a **operação** `visit()` da classe `Interpreter`, ela é resolvida de acordo com o objeto em questão, que é uma expressão de soma. Assim, o **interpretador** executará a soma dos dois operandos da expressão de soma executando `accept()` de cada um deles, que **receberá** o próprio **interpretador** como **parâmetro**:

```
public class Interpreter implements Visitor {

    @Override
    public int visit(SumExpr e) {
        return e.e1.accept(this) + e.e2.accept(this);
    }
//...
}
```

Desse modo, e assim recursivamente, o interpretador passará por todos os nós da árvore até chegar nas folhas, quando então obterá valores que posteriormente serão retornados em direção à raiz.

4 Tarefa

No arquivo `SimpleExprAST-VSCode` do Canvas tem uma implementação (parcial) do uso de AST no projeto de expressões simples já visto. **Pede-se**:

(1) **Estudar** o **código** do **projeto** e **identificar** os elementos que implementam a **AST** e como o **interpretador funciona**.

(2) No **arquivo** `Parser.cup` tem vários **comentários** com a palavra `TODO`. Implantar as classes necessárias para terminar o interpretador, cobrindo todas as funcionalidades ausentes.

(3) Criar um arquivo de teste (pode ser no `teste.input`) e testar todas as funcionalidades faltantes.