Tarea Independiente 04/09/2025

David Núñez Franco

September 7, 2025

Inventario de Conceptos Claves

- Modelo AST: De datos + expresiones + operaciones a AST
- Refactoring (programar es como arte)
- FP en el AST
- Combinador Array::join

(

Ejercicio 1)

1. Dibuje el AST en cada caso usnado el modelo ast que hemos estado desarrollando (ast.mjs)
a) -2 + 3 * 4
b) -(2 + 3 * 4)
c) -((2 + 3) * 4 ** 5)
Note en cada caso que, en principio, había ambigüedades. Señálelas. ¿Cómo se resuelven?

Solución 3a

```
        function test_case_3a() {
            // a) -2 + 3 * 4 -> (-2 + (3 * 4))
            const two = new Num(2);
            const three = new Num(3);
            const four = new Num(4);
            const opMinus = new Oper("-");
            const opPlus = new Oper("+");
            const opMul = new Oper("*");

            const neg2 = new UnaryOp(opMinus, two);
```

```
        const mul = new BinaryOp(opMul, three, four);
        const expr = new BinaryOp(opPlus, neg2, mul);

        console.log("expr_a =", expr.toString()); // out : "(-2
            + (3 * 4))"
    }
```

Primero, se colocan los paréntesis para efectos de orden: **(-2 + (3 * 4))**
Ahora, estas son las ambigüedades y resolución

- * vs +: se resuelve por precedencia estándar.
- - unario vs binario: aquí es unario y aplica solo a 2.

Solución 3b

```
    function test_case_3b() {
        // b) -(2 + 3 * 4) -> "-(2 + (3 * 4))"
        const two = new Num(2);
        const three = new Num(3);
        const four = new Num(4);
        const opMinus = new Oper("-");
        const opPlus = new Oper("+");
        const opMul = new Oper("*");

        const mul = new BinaryOp(opMul, three, four);
        const sum = new BinaryOp(opPlus, two, mul);
        const expr = new UnaryOp(opMinus, sum);

        console.log("expr_b =", expr.toString()); // out : s"-(2
            + (3 * 4))"
    }
```

Primero, se colocan los paréntesis para efectos de orden: **-(2 + (3 * 4))**
Ahora, estas son las ambigüedades y resolución

- Paréntesis fuerzan que el - unario abarque toda la suma.
- Dentro de los paréntesis, * ¿ + como antes.

Solución 3c

```
    function test_case_3c() {
        // c) -((2 + 3) * 4 ** 5) -> "-((2 + 3) * (4 ** 5))"

        const two = new Num(2);
        const three = new Num(3);
```

```
        const four = new Num(4);
        const five = new Num(5);
        const opMinus = new Oper("-");
        const opPlus = new Oper("+");
        const opMul = new Oper("*");
        const opPow = new Oper("**");

        const sum = new BinaryOp(opPlus, two, three);
        const pow = new BinaryOp(opPow, four, five);
        const prod = new BinaryOp(opMul, sum, pow);
        const expr = new UnaryOp(opMinus, prod);

        console.log("expr_c =", expr.toString()); // out : "-((2
            + 3) * (4 ** 5))"
    }
```

Primero, se colocan los paréntesis para efectos de orden: **-((2 + 3) * (4 ** 5))**
Ahora, estas son las ambigüedades y resolución

- Paréntesis obligan a sumar primero 2 + 3.
- ** tiene mayor precedencia que * y es asociativo a derecha; 4 ** 5 va primero.
- El - unario aplica a todo el producto externo.

Ejercicio 2

Parametrice ast::Operation para que los paréntesis y el delimitador se puedan cambiar

Solución

```
    export class Operation extends Node {
        constructor(oper, ...args) {
            super(oper, ...args);
        }

        get oper() {
            return this.head;
        }

        get args() {
            return this.children;
        }

        /* Permite configurar el parentesis y el delimitador.
        * opts = { lparen: '(', rparen: ')', sep: ', ' }
```

```
        */
        toString(opts = {}) {
            // Estos son los valores por defecto
            const cfg = {
                lparen: "(",
                rparen: ")",
                sep: ", ",
                ...opts, // SOBREESCRIBE SI SE PASAN OTROS ARGUMENTOS, POR EJEMPLO, sep = " | "
            };

            // convertimos operador + argumentos a string
            const parts = [this.oper, ...this.args].map(
            (arg) => arg.toString?.(opts) ?? String(arg)
            );

            // construimos la cadena final
            return cfg.lparen + parts.join(cfg.sep) + cfg.rparen;
        }
    }
```

Listing 1: Operation

```
    function test_case_4() {
        const x = new Id("x");
        const n666 = new Num(666);
        const plus = new Oper("+");
        const add = new Operation(plus, x, n666);

        console.log("Default =", add.toString());
        console.log("Parentesis cuadrados =", add.toString({ lparen: "[", rparen: "]" }));
        console.log("Llaves + coma =", add.toString({ lparen: "{", rparen: "}", sep: ", " }));
    }
```

Listing 2: Test Case

Ejercicio 3

3. En el modelo ast.mjs: Implemente lo necesario para manejar (y tener su propio toString) en cada caso: a) Un operador ** (pow) b) Un operador ternario como AST c) Implemente lo necesario par manejar un operador coma (disponible en C/C++, JS)

Solución

3a

```
// potencia
export class PowOp extends BinaryOp {
    constructor(left, right) {
        super(new Oper("**"), left, right);
    }

    toString() {
        return `(${this.left.toString()} ** ${this.right.
            toString()})`;
    }
}

function test_case_5() {
    const n2 = new Num(2), n3 = new Num(3), n4 = new Num(4);
    const rightAssoc = new PowOp(n2, new PowOp(n3, n4)); //
        2 ** (3 ** 4)
    const leftAssc = new PowOp(new PowOp(n2, n3), n4); // (2
        ** 3) ** 4

    console.log("Right associative: ", rightAssoc.toString()
        );
    console.log("Left associative: ", leftAssc.toString());
}
```

Listing 3: PowOp and TestCase

3b

```
// ternario ?
export class Ternary extends Operation {
    constructor(test, cons, alt) {
        super(new Oper("?:"), test, cons, alt);
    }

    get test() {
        return this.args[0];
    }

    get cons() {
        return this.args[1];
    }

    get alt() {
```

```
            return this.args[2];
        }

        toString() {
            return `(${this.test.toString()} ? ${this.cons.
                toString()} : ${this.alt.toString()})`;
        }
    }

    function test_case_6() {
        const a = new Id("a"), b = new Id("b"), c = new Id("c");
        const one = new Num(1), two = new Num(2);
        const plus = new Oper("+"), mul = new Oper("*");

        const cons = new BinaryOp(plus, b, one); // b + 1
        const alt = new BinaryOp(mul, c, two); // c * 2
        const expre = new Ternary(a, cons, alt); // a ? (b + 1)
            : (c * 2)

        console.log("Ternary expr: ", expre.toString());
    }
```

Listing 4: Ternary and TestCase

3c

```
    // coma ,
    export class Comma extends BinaryOp {
        constructor(left, right) {
            super(new Oper(","), left, right);
        }

        toString() {
            return `(${this.left.toString()}, ${this.right.
                toString()})`;
        }
    }

    function test_case_7() {
        const x = new Id("x"), y = new Id("y");
        const one = new Num(1), two = new Num(2);
        const assign = new Oper("="), plus = new Oper("+");

        const setX = new BinaryOp(assign, x, one);
        const setY = new BinaryOp(assign, y, two);
        const sum  = new BinaryOp(plus, x, y);

```

```
            const seq  = new Comma(new Comma(setX, setY), sum); // (x=1, y=2, x+y)
            console.log("comma_seq =", seq.toString());     // "((x = 1), (y = 2), (x + y))"
        }
```

Listing 5: Comma and TestCase