Tarea Independiente 14/08/2025

David Núñez Franco

August 16, 2025

Inventario de Conceptos Claves

- Type como familia de valores (no lo vimos)
- Type safety (polución de memoria)
- Tipo wide (amplio) versus narrow (angosto)
- Ventajas de hacer narrow types
- Inferencia de tipos
- AST con lambda y apply (call, invoke) @
- Cálculo Lambda como máquina de calcular (computar, evaluar) AST's
- Scope (no lo vimos)
- Reducción alpha (cambio de variable)

Punto 0 'let versus var en JS'

Corra este ejemplo y saque conclusiones sobre scope (salga de node después de cada ejemplo)

```
// Ejemplo A (var)
var x = 666
if (true){
    var x = 0
    console.log("A then", x)
} else{
    console.log("A else", x)
}
console.log("A x=", x)

// Ejemplo B (let)
let x = 666
```

```
if (true){
    var x = 0
    console.log("B then", x)
} else{
    console.log("B else", x)
}
console.log("B x=", x)

// Ejemplo C (var function)
var x = 666
function foo(){
    if (false){
        var x = 0
        console.log("C then", x)
    } else{
        console.log("C else", x)
    }
}
console.log("C x=", x)
```

Solución

Al ejecutar los tres algoritmos en Node se obtuvieron los siguientes resultados:

Ejemplo A (var en bloque if)

Salida:

```
A then 0
A x= 0
```

Conclusión: `var` no es *block-scoped*, sino *function-scoped*. La instrucción `var x = 0` dentro del `if` sobrescribe la variable global. Por eso, al final `x` vale 0.

Ejemplo B (mezcla let + var)

Salida:

```
SyntaxError: Identifier 'x' has already been declared
```

Conclusión: no se pueden redeclarar en el mismo ámbito variables con `let` y `var`. `let` es *block-scoped* y bloquea redeclaraciones en el mismo ámbito, mientras que `var` es *function-scoped*.

Ejemplo C (var dentro de función)

Salida:

```
C x= 666
```

Conclusión: dentro de `foo`, la declaración `var x` es *hoisted* al inicio de la función, pero como `foo()` nunca se invoca, no afecta el valor de la variable global. La función define un ámbito independiente, por lo que afuera `x` sigue siendo 666.

Síntesis

En general, el uso de `let` evita efectos colaterales como los observados en el Ejemplo A, al respetar el alcance de bloque y prevenir redeclaraciones.

Punto 1

Considere estas declaraciones en JS

```
const choose = (p, f, g) => x => p(x) ? f(x) :
   g(x)
const False = x => false
const True = x => true
const and = (f, g) => x => f(x) && g(x)
const not = f => x => !f(x)
```

Punto 2 -SOLUCIONADO EN EL CUADERNO DE PRÁCTICA-

Punto 4

Considere el siguiente código en Java

```
<S, T, R> Function<S, R> comp(Function<S, T> f,
    Function<T, R> g){
    return (S x) -> g.apply(f.apply(x));
}
```

a) Pruebe que compila en Jshell b) Cambie por

```
<S, T, R> Function<S, R> comp(Function<S, T> f,
    Function<T, R> g){
    return x -> g.apply(f.apply(x));
}
```

Verfique que igualmente compila ¿Qué feature de Javac permitió que también compile a pesar del cambio

Solución

Primero, verificamos que el código solicitado en **4.a** compila en Jshell:

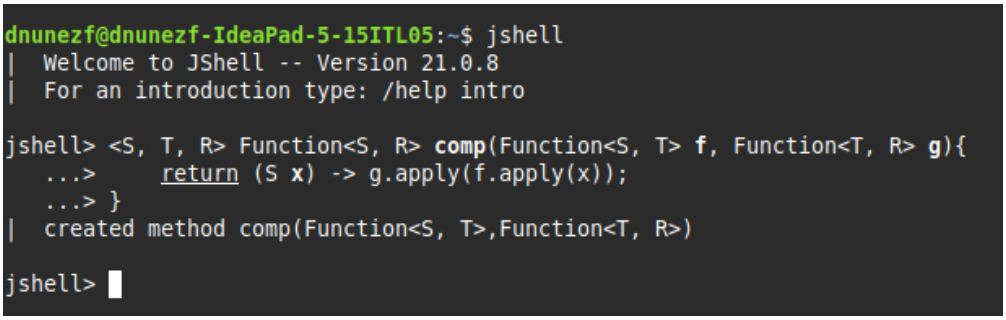

Figure 1: Ejecución en Jshell del código 4.a

Luego, verificamos que el código solicitado en **4.b** compila en Jshell:

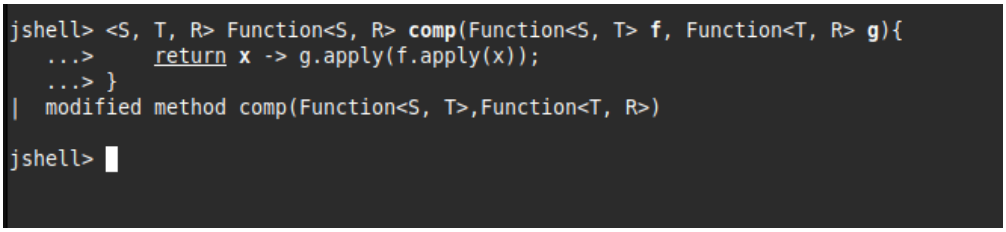

Figure 2: Ejecución en Jshell del código 4.b

La razón por la que la segunda versión también compila es que `javac` usa **type inference con *target typing* en lambdas**.

El compilador no necesita que se escriba `(S x)`, porque deduce automáticamente el tipo de `x` a partir del tipo esperado: la interfaz funcional `Function<S,R>` y los genéricos `S`, `T`, `R`.

En otras palabras, `javac` infiere el tipo del parámetro de la lambda usando el contexto, por eso basta con escribir `x -> g.apply(f.apply(x))`.