

Parser and STD

The intelligent form for validating languages



Problema

- Crear un nuevo lenguaje de programación es un reto enorme.
- Definir una gramática es solo el primer paso.
- Probar un intérprete o compilador es complejo, consume tiempo y es fácil cometer errores.
- ¿Cómo garantizas que tu lenguaje hace lo que dice que hace, y que sigue funcionando después de cada cambio?

solución



- Ofrecemos un entorno integrado de desarrollo y validación diseñado para la fiabilidad y la velocidad.
- Va más allá del simple 'parsing'. Entregamos velocidad de desarrollo y confianza en el producto.

Arquitectura

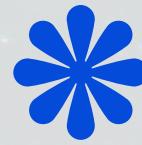
El Parser (El Cerebro)

- Es el componente que lee el código fuente (texto plano) que escribe el usuario.
- ¿Qué hace? Lo analiza, entiende su estructura y lo traduce a un formato que la máquina puede ejecutar (un 'Árbol de Sintaxis Abstracta' o AST).

La Biblioteca Estándar (std)

- ¿Qué hace? Incluimos una biblioteca con funciones listas para usar: imprimir() en pantalla, leer_archivo(), operaciones matemáticas, etc."
- Beneficio: El creador del lenguaje no pierde tiempo programando esto desde cero.

Qué es parser y STD?



STD

Es el **verificador de significado**
Revisa que tu código "tenga sentido" lógico

”

Parser

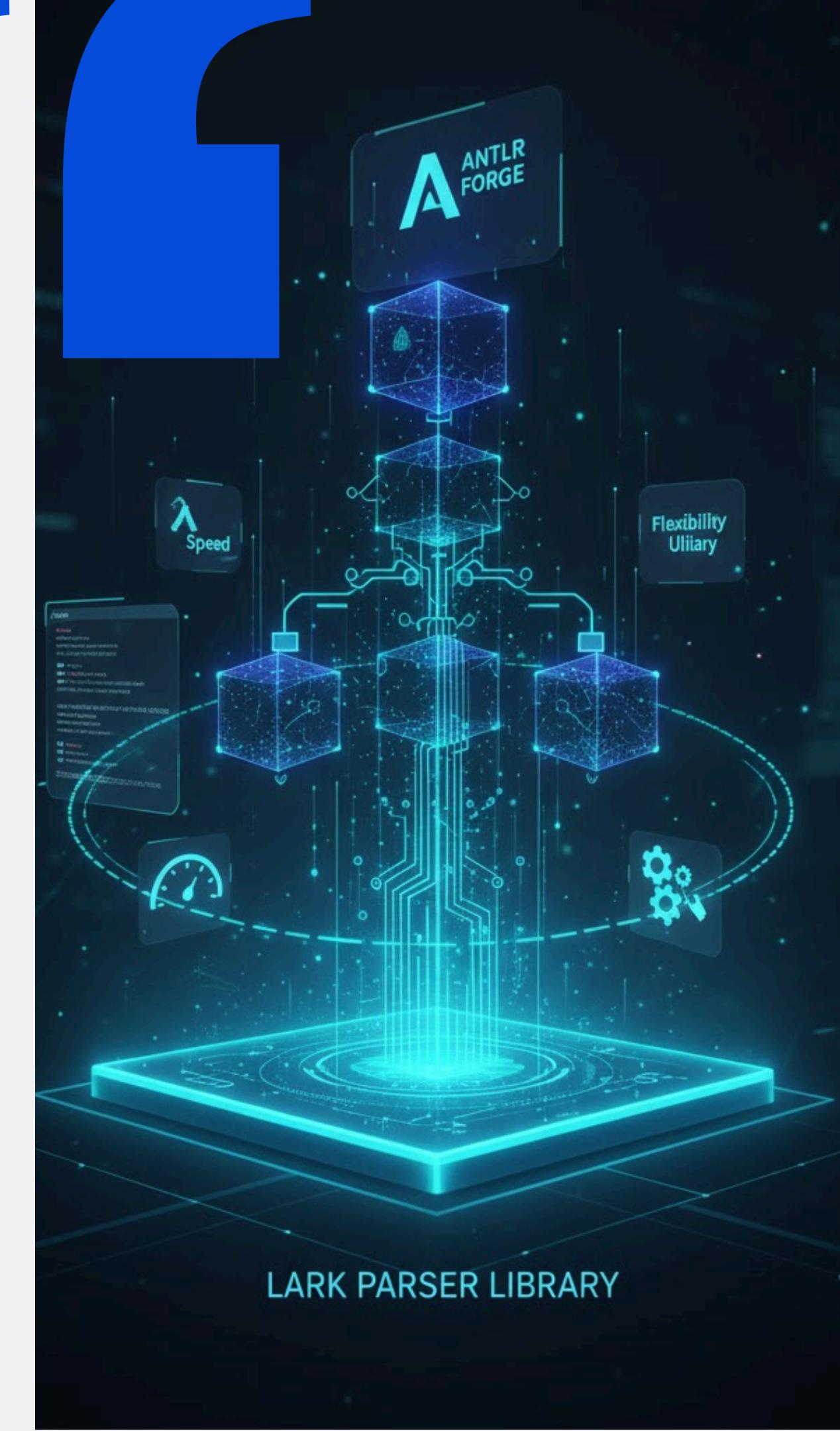
Es el **verificador de gramática**.
Revisa que tu código esté "bien escrito" y siga las reglas estructurales (la sintaxis)

LARK

biblioteca principal

Se usa para crear el analizador sintáctico (parser) a partir de la gramática.

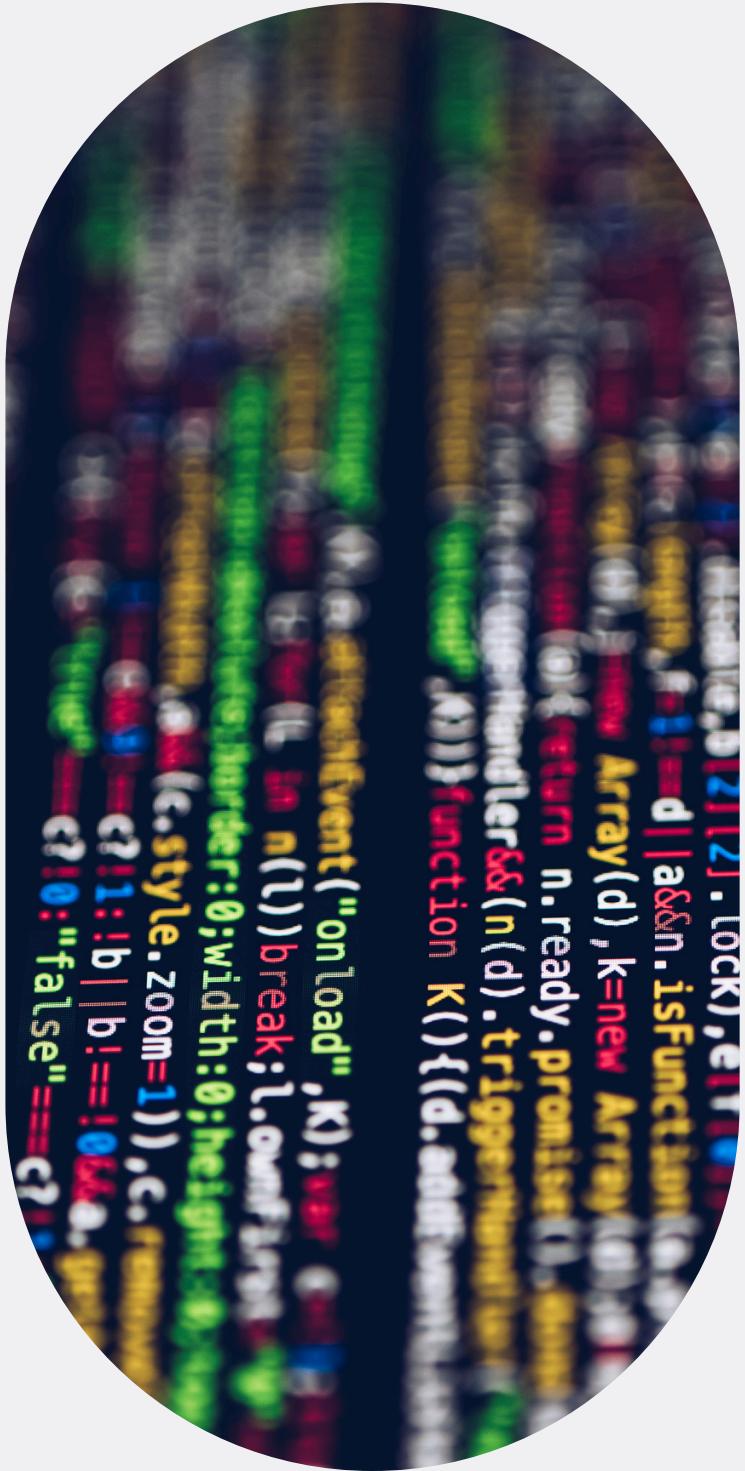
`Lark()`
`.parse()`
`Transformer`
`.transform():`



Funcionamiento

- Entrada: El usuario escribe código (ej. int x = 10).
- Lark (El Parser): Lark toma ese texto, lo "tokeniza" (divide en piezas). Nosotros definimos la gramática (el archivo EBNF) y se la damos a Lark.
Lark tiene un "lexer" integrado que automáticamente divide el texto de entrada en los tokens que definimos.
- El Árbol (Parse Tree): Si la sintaxis es válida, Lark nos entrega un Árbol de Parseo.
- Nosotros (El Transformer SDT): Nuestro código (la clase verifSTD) "camina" por ese árbol para darle significado, validararlo y ejecutarlo.

Diferenciador: El Framework de Pruebas Integrado



El producto no solo funciona con código correcto, sino que sabe cómo fallar de forma inteligente. Nuestro framework de pruebas demuestra esta robustez validando los dos niveles críticos de error:

Pruebas de sintaxis

Pruebas de semántica

Resultados

5.1 Case 1: Valid Arithmetic Expression

Input: 2 + 3 * 4

System Output:

Testing: 2 + 3 * 4

Parsing Success!

Parse Tree saved to 'parseTree.txt'

SDT Verified! Result: 14.0

5.2 Case 2: Valid Type-Checked Declaration

Input: int x = 10

System Output:

Testing: int x = 10

Parsing Success!

Parse Tree saved to 'parseTree.txt'

SDT Verified! Declaration: int x = 10.0

5.3 Case 3: Semantic Error - Type Incompatibility

Input: int x = "text"

System Output:

```
Testing: int x = "text"

Parsing Success!
Parse Tree saved to 'parseTree.txt'
SDT error...!
Detail: Type error: Cannot assign string value 'text'
to integer variable 'x'
```

5.4 Case 4: Variable Assignment

Input: y = 5 + 3

System Output:

```
Testing: y = 5 + 3

Parsing Success!
Parse Tree saved to 'parseTree.txt'
SDT Verified! Assignment: y = 8.0
```

5.5 Case 5: Syntactic Error Detection

Input: 2 + + 3

System Output:

```
Testing: 2 + + 3
```

Parsing error...!

Detail: No terminal matches input.

5.6 Case 6: Runtime Error - Division by Zero

Input: 10 / 0

System Output:

```
Testing: 10 / 0
```

```
Parsing Success!
```

```
Parse Tree saved to 'parseTree.txt'
```

```
SDT error...!
```

```
Detail: Division by zero detected
```

For the input $2 + 3 * 4$, the system generates the following parse tree:

```
add
  number 2
  mul
    number 3
    number 4
```

**PARSE
TREE**

Conclusiones

- Reconocimiento sintáctico
- Detección de errores
- Evaluación de expresiones
- Apoyo visual

