



***Facultad de Ciencias***

# **Intérprete del lenguaje Lox en Python3**

Lox language interpreter in Python3

Trabajo de fin de grado  
para acceder al

**GRADO EN INGENIERÍA INFORMÁTICA**

Autor: Eduardo del Rio Ruiz

Director: Domingo Gómez Pérez

julio de 2023



## *Agradecimientos*

Quiero darle las gracias a toda mi familia y amigos por el apoyo que me han dado siempre y a la Universidad de Cantabria por los servicios ofrecidos a lo largo de la carrera y por el buen trato de los docentes. Especial agradecimiento a Domingo mi tutor por la ayuda. Gracias.



## Resumen

**palabras clave:** intérprete, parser, lexer, lox, python3

En este trabajo de fin de grado se propuso crear un intérprete para el lenguaje Lox, tal como se explica en el libro «Crafting Interpreters» de Robert Nystrom. El objetivo es que esta implementación se utilice en la asignatura «Lenguajes de programación».

Se llevó a cabo esta implementación en Python, sin librerías adicionales, realizando las siguientes tareas:

- presentar especificaciones de los tokens e implementación mediante expresiones regulares
- generar una gramática libre de contexto para el lenguaje Lox que sea adecuada para un analizador recursivo descendente
- programar un intérprete, basado en el recorrido del árbol de derivación
- diseñar una batería de tests para comprobar la validez de la implementación.

El diseño ha sido modificado para centrarse en la adquisición de conceptos y la sencillez del código a programar.

---

*Lox language interpreter in Python3*

## Abstract

**keywords:** interpreter, parser, lexer, lox, python3

In this work, we proposed to create an interpreter for the Lox language, as explained in the book "Crafting Interpreters" by Robert Nystrom. The objective is that this implementation will be used in the subject "Programming Languages".

It was carried out using Python, without additional libraries, by performing the following tasks:

- present token specifications and implementation using regular expressions.
- generate a context-free grammar for the Lox language that is suitable for a downstream recursive parser
- to program an interpreter, based on the path of the derivation tree
- design a battery of tests to check the validity of the implementation.

The design has been modified to focus on concept acquisition and simplicity of the code to be programmed.



# Índice

<b>1. Introducción</b>	<b>1</b>
1.1. <i>Motivación</i>	1
1.2. <i>Objetivos</i>	1
1.3. <i>Desarrollo Software</i>	2
1.4. <i>Introducción al Intérprete y al lenguaje Lox</i>	3
<b>2. Text Scanning</b>	<b>7</b>
2.1. <i>Framework</i>	7
2.2. <i>Lexemas y Tokens</i>	8
2.3. <i>La clase Lexer</i>	8
<b>3. Token Parsing</b>	<b>11</b>
3.1. <i>Representación del código</i>	11
3.2. <i>El patrón Visitante y Metaprogramación</i>	12
3.3. <i>La clase Parser</i>	14
<b>4. Evaluando Expresiones y Sentencias</b>	<b>17</b>
4.1. <i>La clase Intérprete</i>	17
4.2. <i>Sentencias y Estado</i>	18
4.3. <i>Variables Globales</i>	18
4.4. <i>Entornos</i>	19
<b>5. Añadiendo Funcionalidades</b>	<b>23</b>
5.1. <i>Control de Flujo</i>	23
5.2. <i>Funciones</i>	25
5.3. <i>Resolución y vinculación</i>	28
5.4. <i>Clases</i>	31
<b>6. Testing</b>	<b>37</b>
6.1. <i>Comprobando la lectura de Tokens</i>	37
6.2. <i>Resolviendo expresiones</i>	37
6.3. <i>Pruebas del Intérprete</i>	37
<b>7. Trabajos Futuros y Conclusión</b>	<b>39</b>
7.1. <i>Trabajo Futuro</i>	39
7.2. <i>Conclusión</i>	40
<b>Referencias</b>	<b>43</b>
<i>Apéndice 1: Expresiones Regulares TokenType</i>	<b>45</b>
<i>Apéndice 2: Gramática final</i>	<b>47</b>





# 1 *Introducción*

## 1.1. *Motivación*

El desarrollo de compiladores es una de las áreas de la Ingeniería Informática que está mejor comprendida. Todos los nuevos lenguajes de programación suelen incluir un compilador o un intérprete programado en el mismo como una forma de comprobar su utilidad y facilidad para realizar un proyecto de gran envergadura. La diferencia principal entre un intérprete y un compilador es la siguiente: un compilador es un programa que traduce un código fuente a otro idioma de manera simultánea para la totalidad del código sin ejecutarlo. Sin embargo, un intérprete es un programa que traduce un código fuente a otro idioma línea a línea, en tiempo de ejecución, considerando valores previos que pueden modificar el valor de las líneas actuales. Información tomada del vídeo [What is an Interpreter?](#). Los intérpretes de lenguajes de programación añaden al lenguaje una posibilidad de depurar errores de una forma más cómoda y rápida por parte del programador.

Adicionalmente a este motivo y por una decisión a nivel personal, se ha elegido hacer un intérprete por las siguientes razones. Primero de todo, la relación con diversas asignaturas cursadas durante la carrera, en especial con las cursadas en los últimos años en la mención de computación. Otro motivo es que es la creación de un programa que interpreta un texto y lo traduce a otro me resultaba un tema muy interesante. Al fin y al cabo es la base de la programación como la conocemos, sin estos interpretes tendríamos que programar en binario, y no sería posible utilizar lenguajes de programación de tan alto nivel como los actuales.

Por otro lado, despierta mi curiosidad ya que de la misma manera que se traduce ensamblador a 0s y 1s, estos interpretes se utilizan para traducir del castellano al chino mandarín. Los traductores de idiomas que utilizamos cada vez más no dejan de ser interpretes de un idioma a otro, programas mucho más complejos que la traducción entre lenguajes de programación por la dificultad de los mismos. Mediante un intérprete se puede traducir “El Quijote” de Cervantes a un dialecto regional de una pequeña zona de la India o incluso a un idioma ficticio como el élfico de las obras de Tolkien. En resumen, estas razones he elegido este tema para este trabajo de fin de grado.

## 1.2. *Objetivos*

El objetivo de este proyecto consiste en la creación de un programa capaz de procesar y ejecutar código en el lenguaje Lox utilizando única y específicamente python. Como no tenía conocimiento ninguno respecto al tema, más allá de lo aprendido en la asignatura de Lenguajes de Programación, el tutor sugirió seguir un libro que explica paso a paso como crear tu propio intérprete llamado “Crafting Interpreters” de [Nystrom \[2021\]](#). Este libro se centra en un lenguaje bastante simple llamado Lox y utiliza java en todo su desarrollo. En mi caso, por preferencia y para no repetir el código del libro, he elegido python como lenguaje para desarrollar el intérprete de este lenguaje Lox. Ya que python es un lenguaje versátil, se ha tratado de realizar este proyecto usando el menor número de librerías externas posibles y escribiendo el menor número de líneas de código posibles ciñéndonos al principio DRY o *Don't Repeat Yourself* tratando de “refactorizar” el código para no repetir líneas de código que hagan una misma tarea o similar.

El resultado de este proyecto se ha ejecutado y preparado como material didáctico en la asignatura de Lenguajes de Programación, por lo que se ha realizado un esfuerzo importante en tener un código claro y conciso. Esto influye de manera importante a la estructura de la memoria. En vez de explicar el producto final separado por los distintos componentes del intérprete, se adopta otro enfoque. La memoria sigue una estructura similar a la del libro [Nystrom, 2021], siguiendo el orden en que se va desarrollando cada parte del intérprete, volviendo atrás cuando se tiene que modificar algún trabajo ya realizado. El objetivo es que el material esté fragmentado en secciones no muy amplias que se puedan explicar los principios generadores en más en detalle de forma magistral y que al final de cada una de estas secciones se disponga de un código funcional que se pueda ejecutar y probar.

### 1.3. *Desarrollo Software*

Como se ha mencionado, la creación de un compilador o un intérprete es una tarea bien entendida, que tienen un desarrollo muy estandarizado. Este proyecto se ha realizado siguiendo el modelo de desarrollo software incremental iterativo.

El desarrollo software incremental iterativo se centra en pequeños incrementos de un producto existente, cada uno con una cantidad de funcionalidad relativamente pequeña. Éste se caracteriza por realizar un ciclo de desarrollo completo en cada iteración, lo que significa que se realizan todas las fases de un proyecto de desarrollo de software en cada iteración. El objetivo de cada iteración es agregar un conjunto pequeño, pero completo, de funcionalidad al producto existente.

El desarrollo incremental iterativo se considera un enfoque flexible para el desarrollo de software, ya que permite que los requisitos y el diseño del producto evolucionen a medida que se construye el mismo. Este enfoque también facilita la incorporación de cambios en el producto durante el proceso de desarrollo, ya que cada iteración produce un producto funcional que puede ser evaluado y mejorado.

Para la organización del trabajo se ha creado un repositorio Git en el que se han ido actualizando los distintos ficheros de código, documentación, libros de referencia, reportes, etc.. Para facilitar la comunicación de estos entre el tutor y el alumno. Aquí el enlace al repositorio:

[https://github.com/err152/TFG\\_Compilador](https://github.com/err152/TFG_Compilador)

Se ha seguido un esquema de flujo, dividido en una rama master con el programa principal, en paralelo con una rama de desarrollo sobre la que se ejecutan las modificaciones. Una vez los cambios realizados en la rama de desarrollo eran probados como funcionales se juntaban a la rama master.

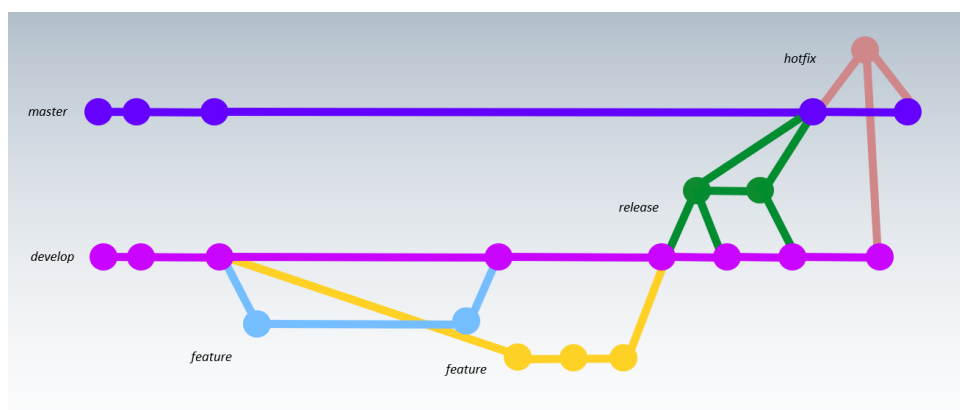


Ilustración 1: Diagrama de Flujo Git

Esta memoria ha sido realizada en Latex para lograr un diseño más pulido y mejor organizado, además de ser requerido por el tutor ya que utilizaba herramientas para ver, editar y guiar el desarrollo de la memoria.

Además se han llevado a cabo reuniones bisemanales para comprobar los avances que se iban haciendo, resolver cualquier duda que surgiese y dar consejo sobre como continuar. Cada reunión se realizó un informe con los contenidos de la reunión, metas y problemas solucionados.

A partir de los informes, se ha realizado el siguiente diagrama de Gantt que representa el tiempo empleado en cada parte del proyecto a lo largo de los meses:

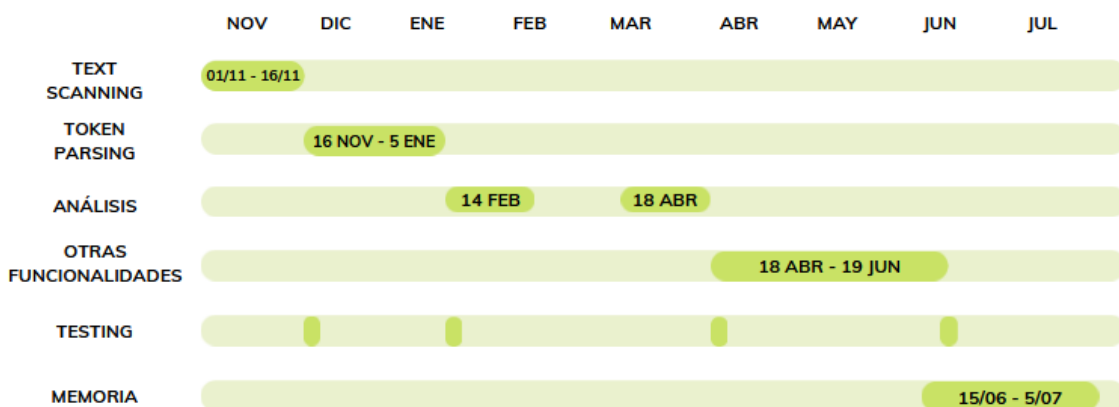


Ilustración 2: Diagrama de Gantt

## 1.4. Introducción al Intérprete y al lenguaje Lox

A priori, programar un idioma es un tarea muy compleja que se ha resuelto poco a poco.

"You must have a map, no matter how rough. Otherwise you wander all over the place. In "The Lord of the Rings" I never made anyone go farther than he could on a given day."

*J.R.R. Tolkien*

El programa comienza siendo un código fuente compuesto por cadenas de caracteres. Cada fase analiza el programa y lo transforma en una representación de una capa superior donde la semántica irá tomando forma.

Llegado a la cima, se tendrá ya un programa de usuario con un código ya comprensible. Ahora, mientras se desciende, se irá transformando el código ya de alto nivel en uno más de bajo nivel hasta llegar a algo que sepa como ejecutar la CPU.

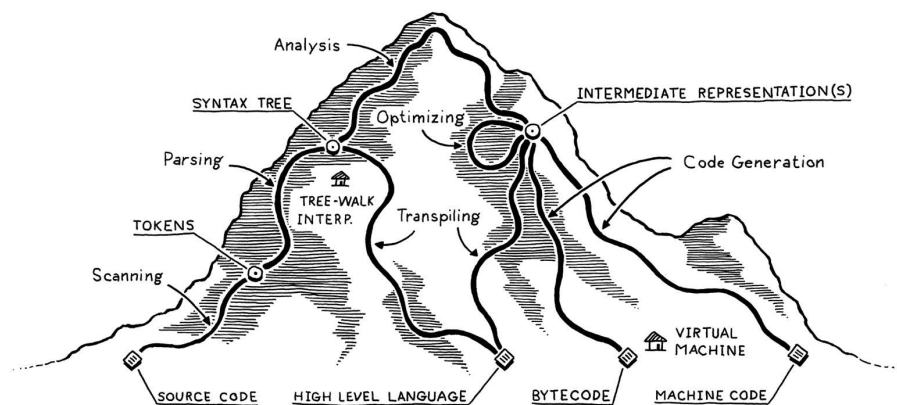


Ilustración 3: Estructura de un Intérprete [Nystrom, 2021]

En este proyecto se implementará un intérprete del tipo «tree-walk», el cual puede ser considerado un atajo ya justo después del análisis sintáctico, avanza ejecutando y evaluando cada rama y cada hoja del árbol sintáctico resultante. Este tipo de intérpretes suele ser lento pero es el más accesible para aprender sobre el tema.

La ruta seguida para este intérprete será la siguiente:

1. «Scanning» (Análisis Léxico). Donde segmentaremos el texto en tokens de distintos tipos.
2. «Parsing» (Análisis Sintáctico). Donde creará un árbol semántico de expresiones y sentencias con los tokens.
3. «Analysis» (Análisis Semántico). Donde se evaluarán las distintas expresiones y sentencias ejecutando el código como resultado.
4. «Runtime» (Ejecución). ¡Hora de ejecutar el programa y ver el resultado final!

Seguiremos esta estructura para **Lox**, que es un lenguaje sencillo que se escribe dinámicamente. Las variables pueden almacenar valores de cualquier tipo, o incluso una variable puede almacenar varios datos de distinto tipo, pero sin hacer conversiones implícitas. En el contexto académico este lenguaje es **fuertemente “tipado” y dinámico**.

En las siguientes líneas pondremos un resumen de como funciona el lenguaje. Como tipos de datos tenemos:

- Booleans: “true” y “false”
- Números: flotantes de doble precision “1234” o “12.34”
- Strings: “I am a string”
- Nil: representa un valor nulo “nil”

Respecto a su estructura, se podría decir que se asemeja a todos los lenguajes de programación más populares como Java y C++. Por ejemplo, las sentencias acaban en “;”, y abriendo corchetes para cada función, clase, scope, etc. Aquí un ejemplo:

```
1 fun sum(a, b) {  
2     return a + b;  
3 }  
4  
5 var result = sum(3, 5);  
6 print(result);
```



## 2 *Text Scanning*

”He who would climb a ladder must begin  
at the bottom.”

---

*Haruichi Furudate*

Este es el primer paso de cualquier intérprete. El *scanner* o *lexer* toma un código fuente como una serie de caracteres y los agrupa en lo que llamamos *lexemas*, los cuales tratados como información de un tipo concreto y en un contexto se convierten en *tokens*. Estos son las piezas que forman la gramática del lenguaje.

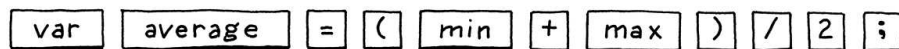


Diagrama que muestra la separación de la expresión `var average = ( min + max ) / 2 ;` en tokens individuales, cada uno en un recuadro: `var`, `average`, `=`, `(`, `min`, `+`, `max`, `)`, `/`, `2`, `;`.

Ilustración 4: Separación en tokens de una sentencia [Nystrom, 2021]

### 2.1. *Framework*

Antes que nada se desarrolla un «framework» al que llamamos *Lox.py* el cual será el encargado de leer el programa en lenguaje Lox como cadena de caracteres y llamar a las distintas partes de nuestro Interpretador. Este acepta el código de 3 maneras distintas:

1. Por terminal como parámetro.
2. De forma interactiva por el terminal, si no se le pasa ningún parámetro.
3. Como ruta al fichero en el que se encuentre el código a procesar.

Principalmente, una vez recibido el código en Lox, este programa lo procesa llamando a los distintos componentes de nuestro Interpretador. Primeramente, crea un Lexer con la cadena de texto a procesar, llama a *extrae\_tokens* y guarda el resultado (los tokens) en una lista. A continuación, crea un Parser con estos tokens, llama a *parse* y guarda el resultado (las expresiones y sentencias) en otra lista. Ahora, se crea un Interpretador, con este, se crea un Resolver que preprocesa las variables, y una vez hecho esto se llama a *interpret*, que ejecuta las distintas expresiones y sentencias, o lo que es lo mismo, ejecuta el código del programa Lox.

Además este contará con una simple gestión de errores.

## 2.2. *Lexemas y Tokens*

En nuestro caso, los tokens son tuplas compuestas de un número de línea, el tipo de token al que pertenecen y un valor. Estos son definidos en una clase con estos 3 atributos en el fichero “Token.py”.

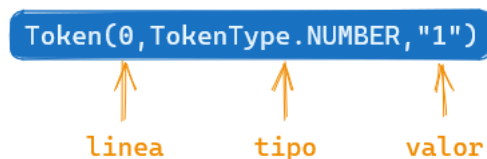


Ilustración 5: Estructura de un Token

En el lenguaje Lox se definen 3 tipos primitivos de tokens a los que llamamos *literales*, estos son: **IDENTIFIER**, **NUMBER**, **STRING**. Por otro lado, se diferencian tokens de un solo carácter (como los paréntesis o la suma y la resta), de dos caracteres (como algunos operadores lógicos o el comentario “//”), palabras reservadas o keywords (como “and”, “func” o “true”) y caracteres especiales como el fin de línea. Todos los distintos tipos de tokens están definidos en un enumerado de nombre *TokenType* en el fichero “Token.py”.

## 2.3. *La clase Lexer*

Una vez definidos los tokens, se desarrolla el *scanner* también llamado **lexer**. Este programa es el encargado de separar en Tokens una cadena de caracteres que recibe como dato de entrada.

Esta parte del interprete se realizó de manera independiente al libro, siendo la implementación completamente distinta a la planteada, por lo que se intentará explicar más detalladamente lo realizado.

Se define en el fichero “lexer.py” la clase *lexer*, que cuenta con unos atributos llamados *entrada*, *pos*, *inicio* y *línea* que respectivamente indican la cadena de texto a interpretar que se inicializa en el constructor, la posición actual dentro de la cadena de caracteres, el inicio del token que se está procesando actualmente, y el número de línea actual. La implementación del *lexer* se realiza como un autómata finito con estados.

Las funciones *devolver\_tokens* y *transición* son las que se encargan de procesar el estado actual y cambiar al siguiente, siguiendo el siguiente autómata finito.

La función *transición* es la encargada de ir reconociendo los caracteres e interpretarlos como los distintos tipos de tokens que tenemos. Para esto fija una posición inicial “*inicio*” y va avanzando analizando el carácter en la posición “*pos*” en la que se encuentra. Cuando reconoce algún tipo de token en los caracteres entre “*inicio*” y “*pos*” retorna el tipo de token que ha reconocido en forma de cadena de texto. Para los números, las cadenas de texto y los comentarios se crean casos especiales que consumen caracteres para avanzar de sus respectivas maneras hasta llegar a cierto carácter que indica el final del token. Por ejemplo, en el caso de la cadena de texto una vez se identifican los caracteres `.o'` se consumen caracteres avanzando “*pos*” hasta volver a encontrar el mismo carácter que empezó la secuencia.



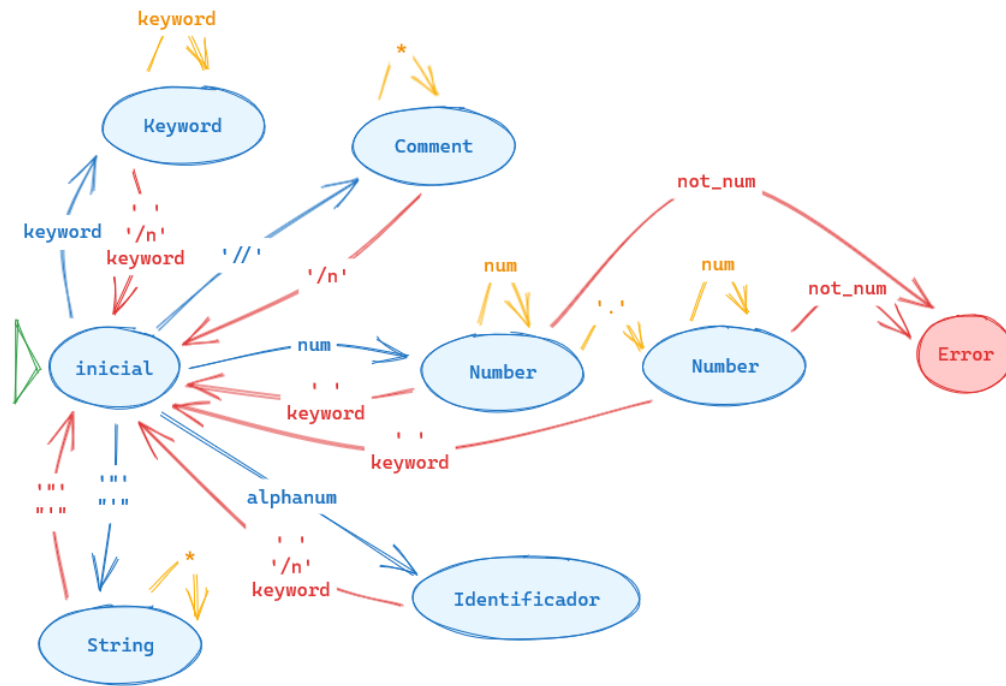


Ilustración 6: Autómata Finito Lexer

En esta función, y en otras partes del intérprete, el código está ordenado de manera que siempre se gestionan antes los casos más restrictivos y después los más genéricos. Por ejemplo, en esta función se tratan primero los tokens de uno y dos caracteres y las palabras clave o «keywords» antes de tratar los identificadores, porque de lo contrario todos los tokens serían aceptados como identificador sin llegar si quiera a comprobar si son de otro tipo.

Ahora, la función *devolver\_tokens* es la que gestiona los estados del autómata, llama *transición* y en función del resultado crea los tokens y ejecuta ciertas acciones u otras. Esto se debe a que *transición* solo se ejecuta una vez y no tiene ningún tipo de bucle o recursión. La función *devuelve\_tokens* mediante dos variables *estado* y *nuevo\_estado* gestiona el cambio de estado entre las llamadas a *transición* y reinicia el estado a inicial una vez termina con un token.

Por último, una función llamada *extrae\_tokens* es la que retorna en forma de lista todos los tokens que devuelve *devuelve\_tokens* añadiendo al final de esta lista el token especial de final de fichero o «EOF» (End Of File).

En el apéndice 7.2 se muestran las expresiones regulares utilizadas para los distintos tipos de tokens.



## 3 *Token Parsing*

### 3.1. *Representación del código*

A la hora de definir el lenguaje de forma sintáctica tendremos que crear una gramática libre de contexto o **CFG** (Context Free Grammar), mediante la cual definiendo reglas o **producciones** se abarcará todo el lenguaje sin definir implícitamente cada punto y coma.

Estas producciones tienen la forma:

$$A \rightarrow Bc;$$

Donde A es la **cabeza** y esta se transforma en el **cuerpo** Bc siendo B una **variable** y c un **terminal**. Los terminales serán tokens que vienen del lexer, y se llaman así porque son un punto final, estos no pueden derivar mediante producciones. Por otro lado, las variables o no-terminales hacen referencia a otras producciones de la gramática.

En nuestra notación también se utilizarán los siguientes símbolos:

- $A \rightarrow b \mid c \mid d$  ; Funcionando el símbolo “|” como un OR.
- $A \rightarrow (b \mid c) d$  ; Los paréntesis dan prioridad de elección al grupo que rodean.
- $A \rightarrow BB^*$  ; El símbolo “\*” indica que la variable a la que sigue se puede repetir entre 0 e infinitas veces. De esta forma logramos recursión.
- $A \rightarrow B^+$  ; El símbolo “+” indica que la variable a la que sigue se puede repetir entre 1 e infinitas veces. De esta forma logramos recursión.
- $A \rightarrow b (CdE)?$  ; Indicando “?” que el grupo o variable al que acompaña aparece una o ninguna vez, es opcional.

La teoría sobre la gramática y autómatas ha sido sacada de [Hopcroft et al. \[2001\]](#), libro utilizado en la asignatura de Lenguajes Formales.

En nuestra gramática para el lenguaje Lox se definen de comienzo las siguientes expresiones:

- Literales: números, strings, booleanos, nil.
- Expresiones Unarias: el prefijo “!” para realizar un NOT lógico o “-” para negar un número.
- Expresiones Binarias: los operadores aritméticos y lógicos que conocemos.
- Paréntesis: Un par de “(” y “)” que envuelven una expresión.

Esta sería nuestra gramática resultante:

```

expression → literal ;
            | unary ;
            | binary ;
            | grouping ;
literal → NUMBER | STRING | “true“ | “false“ | “nil“ ;
grouping → “(“ expression “)” ;
unary → ( “-“ | “!” ) expression ;
binary → expression operator expression ;
operator → “==“ | “!=“ | “<“ | “<=“ | “>“ | “>=“ | “+“ | “-“ | “*“ | “/“ ;

```

A lo largo de este proyecto la gramática es modificada en múltiples ocasiones. En el apéndice 7.2 se puede encontrar la gramática completa.

## 3.2. El patrón Visitante y Metaprogramación

Definida la gramática, se implementan los árboles sintácticos. Para esto se crea una clase Expr en la que se definen los distintos tipos de expresiones, con los elementos que componen cada tipo de expresión. Para el desarrollo de esta clase se ha utilizado el **patrón visitante** y **metaprogramación**.

### 3.2.1. El Patrón Visitante

El **patrón visitante** separa las operaciones de los objetos en una interfaz de visitante, permitiendo agregar nuevas operaciones sin modificar las clases existentes. Por ejemplo:

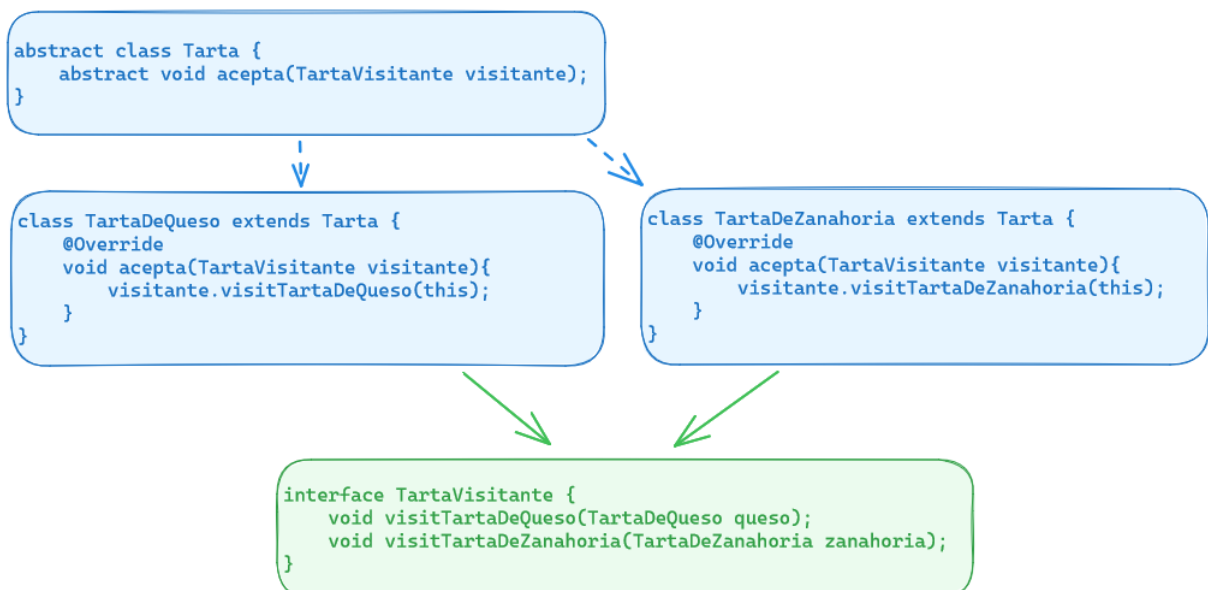


Ilustración 7: Ejemplo Patrón Visitante

Teniendo dos subclases “TartaDeZanahoria” y “TartaDeQueso” que extienden a otra clase principal “Tarta”, se crea una interfaz “visitaTarta” con un nuevo método para cada una de las subclases llamado “visitaTartaDeZanahoria” y “visitaTartaDeQueso”, en la clase “Tarta” se agrega un método llamado “acepta”, y por último en cada una de las subclases se define este método “acepta” de forma que se le pase la interfaz “visitaTarta” como parámetro y ahí se elija si “visitaTartaDeZanahoria” o “visitaTartaDeQueso”. Ahora cuando se desee realizar una operación se llama al método ‘acepta’ y se le pasa como parámetro el visitante (la interfaz) que se quiera ejecutar.

Esto se conoce como **polimorfismo**, definido como la capacidad de llamar a objetos de distintos tipos con la misma sintaxis.

### 3.2.2. Metaprogramación

La **metaprogramación** es una técnica que permite a un programa generar, modificar y analizar código fuente de forma automática. Se utiliza para automatizar tareas repetitivas, generar código dinámicamente y mejorar la flexibilidad del software.

Se obtuvo información sobre este ámbito visualizando el vídeo [Python 3 Metaprogramming](#), en el cual David Beazley explica múltiples conceptos de la metaprogramación en python, y mediante lo que se explica en el propio libro de Crafting Interpreters.

Dicho esto, se crea el fichero metaExpr que contiene una clase *GenerateAst* y 3 funciones: *defineAst*, *defineVisitor* y *defineType*.

La clase *GenerateAst* recibe una ruta a la carpeta del proyecto, a partir de ahí llama a *defineAst* pasándole la ruta añadiéndole el nombre del fichero a generar, el tipo de árbol que va a generar (“Expr” para expresiones y “Stmt” para sentencias) y una lista con las distintas expresiones y sentencias con sus atributos.

La función *defineAst* escribe la base del fichero incluyendo los imports, la definición de la clase principal y las cabeceras de las clases heredadas. Por cada cabecera de clase heredada llama a *defineType*, que se encarga de escribir un método constructor y una implementación del método acepta para cada uno. Antes de definir la clase principal llama a la función restante *defineVisitor* que se encarga de crear la clase Visitante y sus subclases para cada tipo de expresión o sentencia que hayamos pasado como parámetro.

Por ejemplo, la siguiente ejecución de *defineAST*:

```
1 defineAst(self, outputDir+ '/expressions.py', "Expr", ["Get : Expr object, Token
2         name",
                                     "Set : Expr object, Token name, Expr value"])
```

Generaría el siguiente código:

```
1 from Token import Token
2 from typing import Any
3 from abc import ABC, abstractmethod
4 from typing import List
5
6 class ExprVisitor(ABC):
7     @abstractmethod
8     def visit_get_expr(self, expr: 'Expr'):
9         pass
10
11     @abstractmethod
12     def visit_set_expr(self, expr: 'Expr'):
```

```

13     pass
14
15
16 class Expr(ABC):
17     @abstractmethod
18     def acepta(ExprVisitor):
19         pass
20
21 class Get(Expr):
22     def __init__(self, object: Expr, name: Token):
23         self.object = object
24         self.name = name
25
26     def acepta(self, visitor: ExprVisitor):
27         return visitor.visit_get_expr(self)
28
29 class Set(Expr):
30     def __init__(self, object: Expr, name: Token, value: Expr):
31         self.object = object
32         self.name = name
33         self.value = value
34
35     def acepta(self, visitor: ExprVisitor):
36         return visitor.visit_set_expr(self)

```

### 3.3. La clase Parser

La palabra “parse” viene del francés “pars” y significa tomar un texto y mapear cada palabra en la gramática de un lenguaje. El lenguaje en este caso es Lox no el antiguo francés.

Nuestro *parser* toma una secuencia de tokens y crea una estructura en árbol (parse tree/abstract syntax tree) que imita la naturaleza anidada de una gramática, lleva a cabo un análisis sintáctico con los tokens. En esta parte se reportan los *syntax errors* o errores de sintaxis.

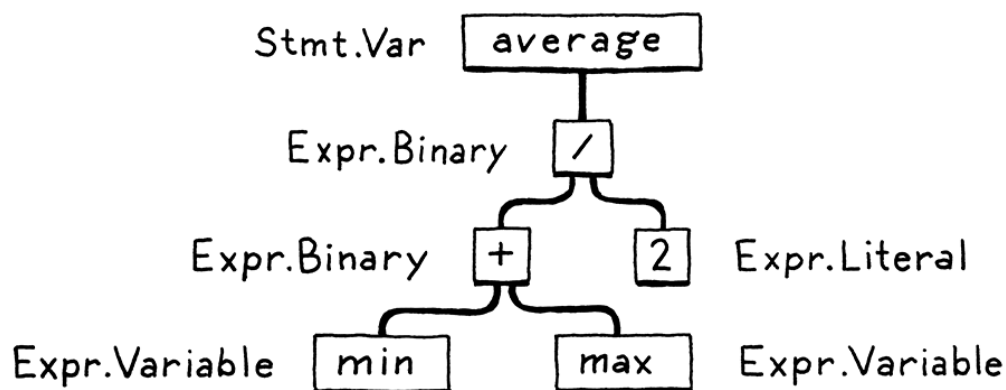


Ilustración 8: Representación de árbol al aplicar parsing [Nystrom, 2021]

#### 3.3.1. Ambigüedad

Cuando se realiza el análisis sintáctico no solo se determina si la cadena de texto es código válido para Lox, se está haciendo un seguimiento de qué reglas coinciden con qué partes de este, de manera

que se sepa a qué parte del lenguaje pertenece dicho token. Se pueden generar dos árboles binarios válidos y distintos pero que devuelvan un resultado distinto:

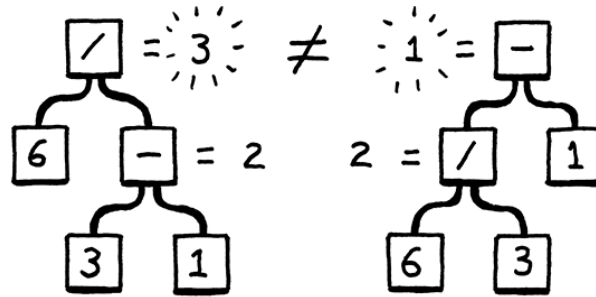


Ilustración 9: Distintos árboles válidos, distintos resultado. [Nystrom, 2021]

Para esto se establecen reglas de precedencia y asociatividad.

- La **precedencia** determina qué operador es evaluado primero en una expresión que contiene una mezcla de distintos operadores.
- La **asociatividad** determina que operador es evaluado primero en una serie del mismo operador.

Se resuelve este problema en Lox utilizando las mismas reglas de precedencia que C, yendo de menor a mayor:

Name	Operators	Associates
Equality	<code>==</code> <code>!=</code>	Left
Comparison	<code>&gt;</code> <code>&lt;</code> <code>&gt;=</code> <code>&lt;=</code>	Left
Term	<code>-</code> <code>+</code>	Left
Factor	<code>/</code> <code>*</code>	Left
Unary	<code>!</code> <code>-</code>	Right

La gramática de expresiones resulta de la siguiente manera:

```

expression → equality ;
equality → comparison ( ( "!=" | "==" ) comparison )* ;
comparison → term ( ( ">" | ">=" | "<" | "<=" ) term )* ;
term → (factor ( ( "-" | "+" ) factor )* ;
factor → unary ( ( "/" | "*" ) unary )* ;
unary → ( "!" | "-" ) unary
        | primary ;
primary → NUMBER | STRING | "true" | "false" | "nil"
        | "(" expression ")" ;

```

### 3.3.2. Parsing Recursivo Descendente

Para realizar el parser se ha utilizado la técnica de recursión descendente, que se considera top-down empezando por la regla gramatical superior hasta llegar a la última de ellas.

Se crea la clase `Parser` en el fichero `parser.py` la cual tiene como atributos una lista de `Tokens` que se le pasa en el constructor como parámetro, y un contador *current* que indica el token en el que se encuentra actualmente.

A continuación se crean funciones para las distintas reglas de la gramática. En cada función se llama a la inferior pasándole unos tipos como parámetro. Una vez se llega al nivel más bajo se empieza a comparar el token con estos tipos, si coinciden se devuelve la expresión en la que se encuentra en ese instante, si no coincide sube a la regla superior y compara con los siguientes tipos. Así hasta que el token coincide con alguno de los tipos y devuelve la expresión o sentencia concreta, en caso contrario devuelve un error.

Esto lo hace con cada uno de los tokens contenidos en su lista `tokens`, avanzando al siguiente según va identificando las distintas expresiones y sentencias.

Por otro lado, Un parser tiene dos tareas:

- Dada una secuencia válida de tokens, producir el árbol sintáctico correspondiente.
- Dada una secuencia no válida de tokens, detectar cualquier error y comunicar al usuario de su fallo.

En este caso se utiliza la técnica de recuperación **panic mode**. Una vez se detecta un error se entra en modo pánico, se para la ejecución y se espera una sincronización.

Esta sincronización se utiliza más adelante cuando se implementa el Intérprete, de momento los errores pararán el programa mostrando un mensaje por pantalla.

Por último, se crea una función *parse* que llama a la función expresión que analizará todos los tokens encapsulada en un `catch` que detecte `ParseErrors`.



## 4 *Evaluando Expresiones y Sentencias*

En el momento que se creó el parser este solo soportaba expresiones. Para ejecutar código se evalúa la expresión y se produce un valor. Por cada expresión sintáctica que se puede “parsear” se necesita un trozo de código que sepa como evaluar ese árbol produciendo un resultado.

Para representar los valores de cada tipo se necesitará una clase correspondiente en python:

Lox type	Python Representation
Any Lox value	Any
nil	None
Boolean	bool
number	float
string	str

### 4.1. *La clase Intérprete*

#### 4.1.1. *Evaluando Expresiones*

Se crea una clase Interpreter que implementa “expressions.Visitor”. Principalmente, cuenta con un método “interpret” que recibe como parámetro la lista de expresiones (o sentencias más adelante) resultante del parser, y ejecuta uno a uno todos los elementos de esta lista. La ejecución de cada expresión se lleva a cabo reutilizando el patrón visitante llamando a un método “execute” que llama a su vez a *accept* y así entrando en el método correspondiente de cada sentencia o expresión.

Para los distintos tipos de expresiones se crea un método que devuelve distintos valores:

- Literales: se devuelve el valor de la expresión.
- Paréntesis: se llama a *evaluate* que vuelve a mandar la expresión que contiene el paréntesis al intérprete.
- Unarias: se comprueba si es un menos o una exclamación y se devuelve el valor en negativo o llama a *isTruthy*. En este método se resuelve el uso de no booleanos en comparaciones lógicas, de manera que si lo que se pasa no es un booleano pero es null devuelve falso.
- Binarias: se devuelven las operaciones aritméticas y comparaciones lógicas correspondientes. Para la suma se define un resultado en caso de que los operandos sean cadenas de texto. Para las igualdades se crea otra función *isEqual* que compare en caso de no ser números si ambos son null o no.

#### 4.1.2. *Runtime Errors*

Hasta ahora los errores controlados eran estáticos o sintácticos, pero ahora necesitamos controlar los errores en tiempo de ejecución. Estos se dan cuando pasamos tipos de dato no válido a nuestras operaciones. En este momento, el programa se detendría y mostraría un error en pantalla, pero esto

no es lo que se quiere.

Para esto en la misma clase `Interprete` se crean funciones que comprueban el tipo de los datos y en caso de no ser válidos lanzan un `LoxRuntimeError`, error que creamos también en este fichero.

Hecho esto, se llama a estas funciones en todas nuestras expresiones antes de operar un resultado.

## 4.2. *Sentencias y Estado*

Se añaden **sentencias** a la gramática de Lox, comenzando por las dos más simples:

- Declaración de una expresión, que permite utilizar una expresión dónde se esperaría una sentencia, y nos permite evaluar expresiones con efectos secundarios.
- Sentencia Print, que evalúa una expresión y la muestra al usuario.

La nueva sintaxis conlleva nuevas reglas gramáticas:

```
_program → statement* EOF ;  
statement → exprStmt  
           | printStmt ;  
exprStmt  → expression “;” ;  
printStmt → “print” expression “;” ;
```

Ahora la primera regla es “programa” que abarca un fichero de lox completo terminando en “EOF”. Se modifica *GenerateAst / metaExpr* para que metaprograme en un nuevo fichero “statements.py” con las distintas sentencias al igual que lo hacía para las expresiones.

Ahora se modifica el método “parse” del parser, y creamos uno nuevo llamado “statement”, que devuelve una sentencia print si coincide con lo que lee o una sentencia de una expresión en su defecto. Al contrario de las expresiones, las sentencias no producen valores si no vacío. Se añaden en el intérprete las clases visitantes para estas sentencias que se han creado. Se modifica el método “interpret” para que interprete sentencias y se crea el método “execute” para validar a estas. Por último, se modifica la clase Lox, más específicamente el “run” para que analice sintácticamente la sentencia en vez de una sola expresión.

## 4.3. *Variables Globales*

Se quiere poder almacenar valores en alguna parte de modo que sea puedan modificar y reutilizar a lo largo del programa, o lo que es lo mismo, variables. Se crean dos nuevos constructos:

- Declaración de una variable, lo que crea la variable.
- Expresión de una variable, la cual accede a dicha variable.

Se modifica una vez más la gramática:

```

__program → declaration* EOF ;
declaration → varDecl
              | statement ;
varDecl → "var" IDENTIFIER ( "=" expression )? ";" ;
statement → exprStmt
           | printStmt ;
primary → "true" | "false" | "nil"
         | NUMBER | STRING
         | "(" expression ")"
         | IDENTIFIER ;

```

Se modifica el fichero `metaExpr` para que agregue estas variables tanto a `expressions` como a `statements`.

El método de entrada al parser cambia, ya que el nivel superior del programa ahora es una lista de sentencias y no expresiones.

Para ello se crea un nuevo método *declaration* que conecta con la recuperación de errores realizada anteriormente que no se llegó a utilizar. Este método es un buen punto para sincronizar el parser cuando entre en modo pánico ya que se le llama repetidas veces según se hace el análisis sintáctico. Ahora definimos la función *varDeclaration* que crea las variables de forma que inicializa la variable como null si no se detecta un token `"="`.

Finalmente se añade el caso del identificador al la función *primary*.

## 4.4. Entornos

Las asignaciones que asocian las variables con sus valores deben estar almacenadas en algún sitio. Este sitio se llama entorno (Environment).

Se crea una nueva clase Entorno que tendrá como atributo un diccionario con las variables y sus respectivos valores. Se definen las siguientes funciones:

- `define()`: para añadir las nuevas variables a esta lista.
- `get()`: para comprobar si ya existe una variable.

Volviendo al Interprete, ahora se crea un entorno como atributo de manera que guarde los valores hasta que este termine de ejecutar.

Se han añadido dos nuevas ramas al árbol sintáctico, la declaración y expresión de la variable, por lo que se crean dos nuevas funciones `visit`, una para cada una, que inicializa y guarda en el entorno la variable y accede a ella respectivamente.

### 4.4.1. Asignación

En Lox se permite la reasignación de las variables. Aquí se presenta un problema y es que la comparación de igualdad está a uno de los niveles más bajos en nuestra gramática. Se tendrá que modificar el parser y añadir una nueva expresión.

Se añade a parser la función *assignment* para solucionar el problema, de manera que se comprueben las asignaciones en un nivel superior de la gramática.

Nuevamente se modifica la gramática:

$$\begin{aligned} \text{expression} &\rightarrow \text{assignment} ; \\ \text{assignment} &\rightarrow \text{IDENTIFIER} \text{ "=" assignment} \\ &\quad | \text{equality} ; \end{aligned}$$

Ahora se tiene un nuevo nodo en nuestro árbol semántico que se define en el intérprete como *visit\_assign\_expr*.

En Entorno se añade *assign* que modifica los valores del diccionario.

#### 4.4.2. *Scope*

Un «**scope**» define la región donde los nombre se mapea a una cierta entidad. Múltiples scopes permiten al mismo nombre referirse a diferentes cosas en diferentes contextos.

«Lexical Scope» es un tipo específico de scoping donde el texto de un programa se muestra donde empieza y termina un scope. Por otro lado tenemos Scope Dinámico en donde no se sabe a que se refiere un nombre hasta que se ejecuta el código. Lox no tiene variables scopeadas dinámicamente, pero si métodos y campos en objetos.

Una primera aproximación a implementar el scoping podría funcionar así:

1. Según se visitan las sentencias dentro de un bloque, se realiza un seguimiento de cualquier variable declarada.
2. Cuando la última sentencia se ejecuta, se le pide al entorno que borre todas esas variables.

Esto no funciona del todo bien. Cuando una variable local tiene el mismo nombre que una variable en dentro de un scope, esta hace sombra a la exterior de manera que ya no puede acceder a esta otra, pero sigue estando ahí y volverá una vez se ejecute el scope.

Antes que nada se añaden los bloques de código, definidos entre corchetes, los cuales abren un scope propio, ejecutan su interior y cierran dicho scope recuperando el contexto. Para esto se modifica la gramática:

$$\begin{aligned} \text{statement} &\rightarrow \text{exprStmt} \\ &\quad | \text{printStmt} \\ &\quad | \text{block} ; \\ \text{block} &\rightarrow \text{"[" declaration* "]" ;} \end{aligned}$$

En este caso se ha utilizado una implementación distinta a la del libro ya que esta no daba buenos resultados a la hora de recuperar el contexto al salir de un scope.

Lo que se ha hecho es lo siguiente:

- Se ha añadido al Entorno un atributo de tipo lista llamado *stack* en el que se almacenan los distintos Entornos según se avanza en el código.

- Se han definido dos nuevas funciones:
  - *enter\_scope*: en el cual se guardan las variables y sus valores en el stack en forma de Entorno y se crea un nuevo diccionario de valores Values.
  - *exit\_scope*: se hace pop sobre el stack eliminando el último entorno existente, y con el sus variables y modificaciones sobre valores.
- Se han modificado los métodos *get* y *assign* para que recorran este stack en busca de las variables a la hora de buscarlas para retornarlas o modificarlas.
- En el Intérprete, el método *executeBlock* ya no recibe un entorno como parámetro.
- La gestión de los distintos entornos creados por los scopes se gestionan ahora en la clase Entorno por lo que el método *executeBlock* tan solo llama a *enter\_scope* para guardar el entorno, ejecuta los statements que contiene el bloque y llama a *exit\_scope()* para recuperar el entorno anterior.



## 5 *Añadiendo Funcionalidades*

A partir de aquí se irán añadiendo distintas funcionalidades al intérprete para cubrir el lenguaje Lox de forma lo más completa posible.

### 5.1. *Control de Flujo*

”Logic, like whiskey, loses its beneficial effect when taken in too large quantities.”

---

*Edward John Moreton Drax Plunkett, Lord Dunsany*

El siguiente paso para nuestro intérprete fue lograr que sea Turing-completo.

#### 5.1.1. *Máquinas Turing*

Las máquinas de Turing son un pequeño sistema que con la mínima maquinaria es capaz de calcular cualquiera de una clase (muy) grande de funciones.

Cualquier lenguaje de programación con un mínimo nivel de expresividad es suficientemente potente como para calcular cualquier función computable.

Esto se puede probar escribiendo un simulador de máquinas de Turing en tu lenguaje. Ya que Turing lo demostró para su máquina, eso significaría que tu lenguaje también. Solo hay que traducir la función en una máquina de Turing, y luego probarlo en tu simulador.

Si tu lenguaje puede hacerlo, se le considera Turing-completo. Las máquinas de Turing son muy simples, solamente se necesita aritmética, un poco de control de flujo y la habilidad de asignar y utilizar cantidades arbitrarias de memoria. Se tiene lo primero, ahora se hace lo segundo.

#### 5.1.2. *Ejecución Condicional*

Podemos dividir el control de flujo en dos tipos:

- Condicional o control de flujo en ramas («**Branching**»), es utilizado para no ejecutar ciertos trozos del código. Saltar por encima de partes del código.
- Control de flujo en bucles («**Looping**»), ejecuta cierta parte del código más de una vez. Salta hacia atrás para volver a hacer algo, y para no entrar en bucles infinitos se tiene alguna condición lógica para salir de este.

El branching es más simple así que se empezará por ahí. Lox no tiene operadores condicionales, así que se implementará la sentencia **if** lo primero de todo. Nuestra gramática de sentencias gana una

nueva producción:

```
statement → exprStmt
           | ifStmt
           | printStmt
           | block ;
ifStmt → “if” “(“ expression “)” statement
       | ( “else” statement )? ;
```

Una sentencia if tiene una expresión para la condición, y una sentencia que ejecutar si la condición es cierta. Opcionalmente, puede tener un else y otra sentencia que ejecutar si la condición es falsa. Se agrega If al árbol sintáctico.

Se añade también que el parser reconozca estas sentencias cuando lee el Token IF y llame al método *ifStatement* que parsee el resto. Si encuentra un Token else crea la sentencia y si no lo deja a None.

Esta sentencia else genera un problema de ambigüedad. Considera:

```
1 if (first) if (second) whenTrue(); else whenFalse();
```

¿A qué if pertenece el else? Se tiene que solucionar esto. En nuestro caso el else pertenecerá al if más cercano a este mismo, ya que if busca un else antes de retornar. De momento esto nos sirve y podemos pasar al intérprete, donde se añade un método *visit\_if\_stmt* que procese los ifs.

Ahora se implementan los operadores lógicos **and** y **or**. Estos funcionan de distinta manera ya que dependen del resultado de uno de los dos lados de la operación para producir su resultado. Es por esto que van separados del resto de operadores binarios.

```
expression → assignment ;
assignment → IDENTIFIER “=” assignment
           | logic_or ;
logic_or → logic_and ( “or” logic_and )* ;
logic_and → equality ( “and” equality )* ;
```

Se añade al generador y en el parser ahora *assignment* en vez de llamar a *equality* llamará a *orr*. Se definen los métodos *orr* y *andd*. Y en el intérprete se añade el método *visit\_logical\_expr* después de *visit\_literal\_expr*.

### 5.1.3. Bucles While

Lox cuenta con dos tipos de control de flujo por bucles, **while** y **for**. Se empieza por el bucle while al ser este más sencillo.

```
statement → exprStmt ;
           | ifStmt ;
           | printStmt ;
           | whileStmt ;
           | block ;
whileStmt → “while” “(“ expression “)” statement
```



No tiene mucho misterio, se añade While al generador metaExpr, al parser con su método *whileStatement*, y al intérprete con el método *visit\_while\_stmt*.

#### 5.1.4. Bucles For

Por último falta implementar el bucle for cuya gramática sería la siguiente:

```
statement → exprStmt
           | forStmt
           | ifStmt
           | printStmt
           | whileStmt
           | block ;
forStmt → “for“ “(“ (varDecl | exprStmt | “;“ )
           expression? “;“
           expression? “)” statement ;
```

Dentro del paréntesis se tiene:

- Una inicialización, que se ejecuta una única vez y suele ser una expresión pero se permitirá que sea una variable.
- Una condición, que se ejecuta una vez al comienzo de cada iteración y controla cuando salir del bucle.
- Un incremento, que es una expresión arbitraria que realiza algún tipo de trabajo al final de cada iteración.

En verdad Lox no necesita bucles for, se podría hacer lo mismo con un bucle while, pero es más conveniente y más dinámico tenerlo por separado.

Añadimos al parser un método *forStatement* que gestiona estos elementos del bucle for y llama a *statement* para el cuerpo del bucle. La parte que puede resultar más complicada en la implementación del bucle for es el añadir el incremento al bloque que se ejecuta por cada iteración, se podría decir que el bucle for es la definición de una variable, y un bucle while al que se le añade una expresión/sentencia extra al final.

## 5.2. Funciones

Es el momento de introducir las funciones en nuestro interprete.

### 5.2.1. Funciones de llamada

Las funciones de llamada son aquellas que ejecutan una función ya definida pudiendo pasar ciertos parámetros para su ejecución. Además tienen mayor precedencia que cualquiera de los otros operadores

por lo que se modifican las siguientes reglas:

$$\begin{aligned}\text{unary} &\rightarrow (\text{"i"} \mid \text{"-"}) \text{ unary} \mid \text{call} ; \\ \text{call} &\rightarrow \text{primary} ( \text{"(" arguments? ")" } )^* ; \\ \text{arguments} &\rightarrow \text{expression} ( \text{"," expression } )^* ;\end{aligned}$$

Se agrega un nuevo nodo al generador del árbol sintáctico. Ahora en el parser, el método *unary* en vez de llamar a *primary* llamará a un nuevo método *call*. Este método llama a *primary* y entra en un bucle que si lee un paréntesis llama a *finishCall*. Este otro método consume las expresiones que serían los argumentos de la función separándolos por las comas que va encontrando.

En el intérprete, se define una función *visit\_call\_expr*, que evalúa los argumentos y se los pasa a la función *call*.

Se crea una nueva interfaz *LoxCallable*.

Se añade al interprete en el nuevo método anteriormente creado *visit\_call\_expr* una breve gestión de errores.

Para terminar se trata de resolver la **aridad** de las funciones. Se dice aridad al requerimiento de cierto número de parámetros de la función, si una función en su definición recibe 3 parámetros, se le deberán pasar 3 parámetros a la hora de llamarla. Esto lo solucionamos en la misma función *visit\_call\_expr*.

### 5.2.2. Declaración de funciones

Ahora se puede llamar a funciones, pero no se tiene ninguna todavía. Toca declarar funciones de manera que queden guardadas y se pueda acceder a ellas mediante las llamadas. Se añaden las siguientes reglas a nuestra gramática:

$$\begin{aligned}\text{declaration} &\rightarrow \text{funDecl} \\ &\quad \mid \text{varDecl} \\ &\quad \mid \text{statement} ; \\ \text{funDecl} &\rightarrow \text{"fun"} \text{ function} ; \\ \text{function} &\rightarrow \text{IDENTIFIER} ( \text{"(" parameters? ")" } \text{block} ; \\ \text{parameters} &\rightarrow \text{IDENTIFIER} ( \text{"," IDENTIFIER } )^* ;\end{aligned}$$

Se añade *Function* a *metaExpr* y se realizan un par de cambios en el parser:

- Se modifica *declaration* para que identifique la definición de funciones al leer el token "fun".
- Se añade la función *function* que ya dentro de la declaración de la función consume el identificador de la misma o devuelve un error en caso contrario.
- Ahora dentro de *function* gestionamos los parámetros y llamamos al cuerpo de la función.

### 5.2.3. Objetos de funciones

Falta definir algún tipo de recipiente donde guardar las funciones, esto lo hace *statements.Function* pero es necesaria una clase *LoxFunction* que implemente a *LoxCallable* para poder llamar a las funciones.

Se crea esta clase `LoxCachable` con sus respectivos métodos *arity*, un `__str__` para imprimir por pantalla su valor, y *call* que será el que ejecute la función.

Se añade al intérprete un *visit\_function\_stmt* para procesar estas funciones.

#### 5.2.4. Sentencias de retorno

Es el momento de añadir las sentencias de retorno.

```
statement → exprStmt
           | forStmt
           | ifStmt
           | printStmt
           | returnStmt
           | whileStmt
           | block ;
returnStmt → "return" expression? ";" ;
```

Un retorno comienza con el token "return" y puede ir acompañado de una expresión o no, para poder salir de una función que retorna un valor inútil antes.

Toda función debe retornar algo, en su defecto `None`.

Se añade *return* a *metaExpr* y se modifica el parser:

- Se modifica que *statement* llame a *returnStatement* cuando lee el token "return".
- Se crea el método *returnStatement* que trata la sentencia de retorno.

Retornar puede ser algo complicado. Se puede retornar desde cualquier parte dentro del cuerpo de una función, incluso desde dentro de múltiples sentencias anidadas. Cuando el retorno es ejecutado, el intérprete tiene que saltar fuera del contexto en el que esté y de alguna manera completar la función.

Se modifica el interprete con un nuevo método *visit\_return\_statement*. En este método se toma el valor de la sentencia en el que nos encontramos y se genera una excepción de retorno.

`Return` es una nueva clase que se define y extiende `RuntimeError`. Esta llama al constructor de `RuntimeError` sin parámetros, y guarda el valor de la sentencia que se le pasa.

A continuación, en `LoxFunction` en la función *call*, rodeamos la ejecución del bloque en un `try` que en caso de detectar un error de tipo `Return` retorne el valor del error, que sería el de la sentencia actual.

#### 5.2.5. Funciones y cierres locales

Nuestra implementación de *call* crea un nuevo entorno al que vincula los parámetros de las funciones. Pero, ¿quién es el "padre" de este entorno?

Hasta ahora, siempre han sido el entorno global superior al actual. De esta manera cuando no se encuentra un identificador en un entorno local se busca en el inmediatamente superior. Pero en `Lox` las declaraciones de funciones son permitidas en cualquier lugar dónde se pueda vincular un nombre.

Se denomina **cierre** «closure» a una combinación de una función y el entorno léxico dentro del cual se define esa función. Es un concepto poderoso que permite a las funciones “recordar” y acceder a variables de su ámbito externo incluso después de que ese ámbito haya terminado de ejecutarse.

En este apartado, nuevamente, se realiza una implementación distinta a la del libro, siguiendo el acercamiento que se plantea en los capítulos 3.1 y 3.2 libro [Abelson y Sussman \[1996\]](#). Aquí un ejemplo del estado de

Se añade en `LoxFunction` un nuevo entorno de cierre “closure” que funciona como copia del entorno en el momento de definición de la función. Esto se guarda en su constructor al cual se llama dentro del Interpretador en la función `visit_function_stmt`.

Hasta aquí todo está como en el libro, pero ahora en la clase `Entorno` añadimos también un atributo `closure_function` el cual se utiliza para guardar el entorno capturado en el momento de la definición en el caso de las funciones que hemos mencionado anteriormente. Al hacer esto, modificando las funciones `get` y `assign`, se pueden buscar las variables tanto en valores, como en el stack, como en este nuevo entorno. Hecho esto solo falta asignar este `closure_function`. Esto se hace en el momento de la llamada a la función, en el `call`. Cuando se lleva a cabo la llamada se asigna el entorno que tenemos copiado en la `LoxFunction` al `closure_function` del entorno actual, para después operar la función como ya se había definido anteriormente.

En el siguiente ejemplo se ilustra cómo funciona esta implementación que puede resultar confusa a priori y que dio bastantes problemas en su resolución:

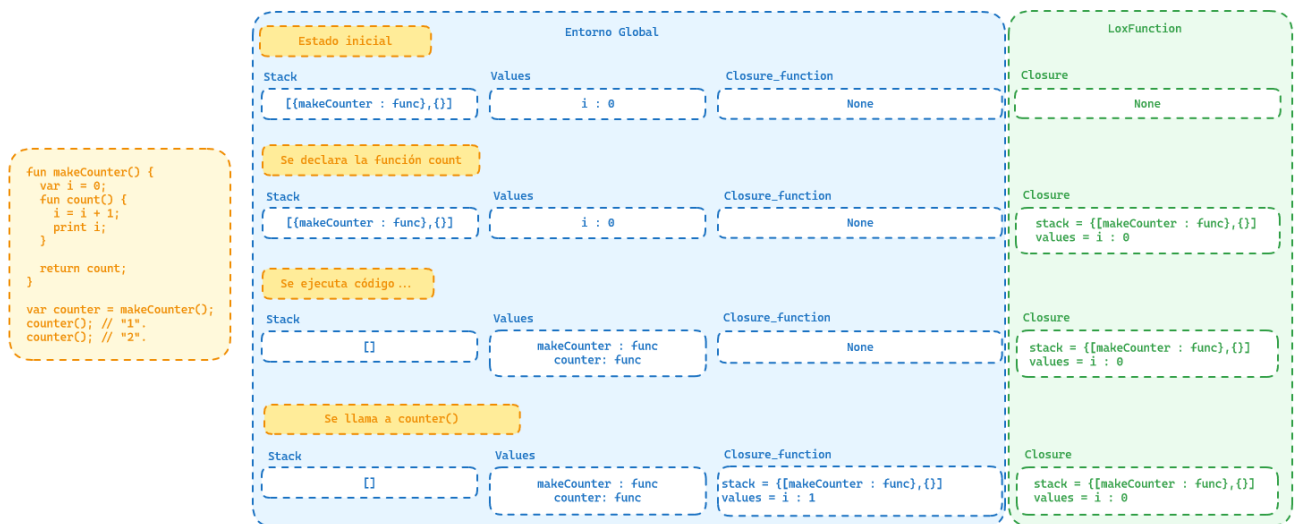


Ilustración 10: Ejemplo de Clausuras (*prueba\_func9*)

En este ejemplo se ejecuta el fichero `prueba_func9` y se enseña el estado del stack, valores y `closure_function` del entorno global junto al `Closure` de la función en el momento previo y posterior de su definición y llamada, en ese orden. Como se puede apreciar, si cuando se realiza la llamada a la función no se dispone de esta copia del entorno en el momento de creación de la función, no se podría acceder a la variable `i`, ya que esta no existe en el entorno exterior en el que se realiza la llamada.

### 5.3. Resolución y vinculación

Un brecha se ha producido en nuestro barco al implementar los cierres.

### 5.3.1. *Static Scope*

El uso de una variable se refiere a la sentencia previa con el mismo nombre en el ámbito más interno que engloba la expresión en la que se utiliza la variable.

Analicemos esta frase:

- Se dice “uso de una variable” en vez de “expresión de una variable” para abarcar expresiones y asignaciones.
- “Previa” significa que aparece antes en el texto del programa.
- “Más interno” se utiliza por el «shadowing». Puede haber más de una variable con el mismo nombre en un entorno circundante.

Dado el siguiente código:

```
1 var a = "global";
2 {
3   fun showA() {
4     print a;
5   }
6
7   showA();
8   var a = "block";
9   showA();
10 }
```

Si estas familiarizado con scopes y closures el resultado debería ser “global” dos veces. Ya que la misma función debería acceder siempre a la misma variable.

Pero en este caso se devuelve “global” “block”.

### 5.3.2. *Análisis Semántico*

Nuestro intérprete resuelve una variable cada vez que la expresión variable es evaluada. Si esta variable está dentro de un bucle que se ejecuta 100 veces, la variable será resuelta 100 veces.

La solución es escribir un programa que investigue todo el código, encuentre cada variable mencionada, y encuentre a que sentencia se refiere cada una. Este proceso es un ejemplo de **análisis semántico**. Mientras que el parser solo comprueba si el programa es gramáticamente correcto, el análisis semántico va un paso más allá y averigua lo que significa en realidad cada pieza del código.

La clave para solucionar nuestro error será medir los saltos entre entornos que se realizan para buscar las referencias a las variables. De manera que estos saltos se almacenen y siempre se realice el mismo número de saltos, accediendo siempre al mismo resultado.

Después de que el parser produzca el árbol sintáctico, pero antes de que el intérprete comience a ejecutarlo, se realizará un paseo por el árbol para resolver todas las variables que contenga.

En este paseo se visitan todos los nodos, pero un análisis estático es diferente de una ejecución dinámica:

- No tiene efectos secundarios. No ejecuta nada por lo que no tiene efecto.
- No tiene control de flujo. Los bucles son visitados una única vez. Ambas ramas son visitadas en los bucles if. Los operadores lógicos no son cortocircuitados.

### 5.3.3. Una clase *Resolver*

Se crea una nueva clase *Resolver*. A la hora de resolver variables solo interesan un par de nodos:

- La declaración de un bloque introduce un nuevo scope para las sentencias que contiene.
- La declaración de una función introduce un nuevo scope para su cuerpo y vincula sus parámetros a este scope.
- La declaración de una variable añade la variable al scope actual.
- Las expresiones de variables y asignaciones necesitan tener sus variables resueltas.

El resto de nodos no hacen nada especial, aunque se tendrán que visitar de todas formas para recorrer el árbol.

Esta clase tiene como atributos: un intérprete, en el que se guardan los resultados; una lista de diccionarios de nombre *scopes* que actúa como una cola en la que se van simulando la creación de entornos y guardando el estado (inicializadas o no) de las variables que se necesiten resolver; y un entorno de nombre *currentFunction* mediante el que se resuelven las clausuras.

Empezando por los bloques se crea un método *resolve* que pasada una lista de sentencias llama a *resolve* para cada una de ellas. A su vez si esta recibe una sentencia o una expresión llama a sus métodos *accepta* para ser resueltas.

Se crea ahora un método *visit\_block\_stmt* en el que se llama a *beginScope*, función que guarda un diccionario vacío en una la lista *scopes* de la clase *Revolver*. Seguido, llama a *resolve* para la lista de sentencias, y luego llama a *endScope*, función que extrae un diccionario de la lista *scopes*.

Para la declaración de variables se diferencia entre la declaración y la definición. Se añade un nuevo método *visit\_var\_stmt* que llama a *declare*, resuelve la variable y llama a *define*.

¿Qué ocurre cuando el “inicializador” de una variable local hace referencia a una variable con el mismo nombre que la variable que se está declarando? Se tienen diferentes formas de actuar:

1. Inicializar y luego añadir la nueva variable en el scope.
2. Añadir la nueva variable al scope y luego inicializarla.
3. Que esto devuelva un error.

Como las primeras tienden a producir errores de usuario, se implementa la tercera.

Primero se define la sentencia que agrega la variable al scope más interno, de manera que oculta cualquier scope externo y así saber que la variable existe. Se marca como “no lista todavía” al vincular su nombre a “False” en el diccionario *scope*.

Se crea ahora el método *visit\_var\_expr*, en el que primero se comprueba si la variable está siendo accedida desde dentro de su inicialización. Si la variable existe en el actual scope pero tiene valor False, se retorna un error. Si no, se llama a *resolveLocal*, función que de manera similar al Entorno busca desde el scope actual hacia arriba en búsqueda de la variable, si se encuentra se resuelve pasándole el número de saltos entre scopes que se han realizado.

Para las asignaciones, tan solo se resuelve la expresión en *visit\_assign\_expr*.

Para las funciones, se declara y define la sentencia y se llama a la función *resolveFunction*, la cual abre un scope, declara y define los parámetros de la función, resuelve el cuerpo de esta y cierra el scope.

Finalmente, se añaden los demás nodos en los que no se hace nada especial más allá de resolver sus diferentes componentes.

#### 5.3.4. Interpretando Variables Resueltas

Veamos de que sirve el Resolver. Cada vez que se visita una variable, comunica al interprete el número de scopes que hay entre el actual y donde está definida la variable. Se añade un método *resolve* que guarde en una nueva lista *locals* las expresiones y el numero de saltos para acceder a ellas.

Se modifica *visit\_var\_expr* para que llame a una nueva función *lookUpVariable()* que comprueba en *locals* la distancia de la expresión que se va a tratar. Si la distancia no es nula llama a una nueva función del entorno llamada *getAt*, de lo contrario toma el valor del entorno global.

En este momento se vuelve a desviar el contenido respecto del libro. En este caso, la función *getAt* recibe una lista de distancias y un nombre de variable. Primeramente llama a un método auxiliar el cual crea una lista de diccionarios en la que almacena los distintos entornos a los que se quiere acceder mediante las distancias. Luego, para cada diccionario busca la variable y si la encuentra la retorna. Si no la encuentra y el entorno de clausura no es nulo se llama recursivamente a *getAt* sobre este entorno de clausura con la misma lista de distancias; por último, si de ninguna de las maneras se encuentra la variable, se produce una excepción *RuntimeError* que el método *lookUpVariable* gestiona llamando al *get* básico que buscará la variable en el entorno global de forma normal.

De igual manera se hace con *visit\_assign\_expr*, en este caso llamando a *assignAt* que actúa de la misma manera que *getAt* pero dando valor a las variables una vez las encuentra.

Para estos 3 métodos nos hemos ceñido una vez más al principio DRY (Don't Repeat Yourself) tratando de refactorizar el código todo lo posible para no repetir acciones.

Finalmente, se añade al programa principal la definición del resolver, y se hace una llamada a *resolve* sobre el interprete antes de ser este otro ejecutado, para que en el momento en que se llame a *interpret* tenga ya la lista con distancias para acceder a las variables de manera mucho más rápida.

#### 5.3.5. Errores de Resolución

Se añade control de errores al Resolver para los casos en los que en un scope local se crean dos variables con un mismo nombre, y para prevenir llamadas return fuera de ningún scope.

### 5.4. Clases

Se podría terminar el Interprete aquí pero hoy en día muchos lenguajes de programación populares soportan la programación orientada a objetos. Añadirlo será un extra para darle cierta familiaridad a los usuarios.

### 5.4.1. *OOP and Classes*

Existen 3 caminos hacia la programación orientada a objetos: clases, prototipos y multimétodos. Las clases fueron las primeras en inventarse y son las más populares actualmente. El principal objetivo de una clase es agrupar datos con el código que actúa sobre ellos. Para eso los usuarios declaran una clase que:

- Expone un constructor que crea e inicializa nuevas instancias de la clase.
- Proporciona una manera de guardar y acceder a los campos de estas instancias.
- Define un grupo de métodos compartidos por todas las instancias de la misma clase que operan sobre el estado de cada instancia.

Ese sería un resumen muy general. Muchos lenguajes programados a objetos también implementan herencia para reutilizar el comportamiento entre clases, pero hasta aquí llegará este proyecto.

### 5.4.2. *Declaración de Clases*

Se comienza modificando la sintaxis. Cambian las reglas de nuestra gramática.

$$\begin{aligned} \text{declaration} &\rightarrow \text{classDecl} \\ &\quad | \text{funDecl} \\ &\quad | \text{varDecl} \\ &\quad | \text{statement} ; \\ \text{classDecl} &\rightarrow \text{"class" IDENTIFIER " " function* " " ;} \end{aligned}$$

En Lox las clases se definen mediante la palabra clave “class” seguida del nombre de la clase. En su cuerpo se definen los métodos al igual que las funciones pero sin ser precedidas de la palabra clave “fun”.

```

1 class Breakfast {
2   cook() {
3     print "Eggs a-fryin'!";
4   }
5   serve(who) {
6     print "Enjoy your breakfast, " + who + ".";
7   }
8 }
```

Se añade la regla `classDecl` al generador AST `metaExpr`. Esta guarda el nombre de la clase y los métodos en su cuerpo

Se añade en el parser en la función `declaration` la detección del token “CLASS”. También se añade una nueva función `classDeclaration`. Esta función consume un token que corresponde al nombre, gestiona los corchetes que definen el cuerpo y crea una lista en la que guarda los métodos de la clase.

Se añade también un método `visit_class_stmt` tanto al resolver como al intérprete. En el resolver, declaramos y definimos la sentencia. En el intérprete, definimos la sentencia en el entorno global `ent`, creamos una clase `LoxClass` y asignamos esta clase a la sentencia.

`LoxClass` es una nueva clase que creamos en el fichero `LoxClass.py`. En un principio esta clase tan solo contiene un constructor que inicializa el atributo nombre de la clase, y un método `__repr__`



para mostrar el nombre cuando la clase se tenga que mostrar.

### 5.4.3. *Creando Instancias*

Lox no tiene métodos estáticos a los que llamar dentro de la propia clase, por lo que sin instancias las clases son inútiles. Para crear estas instancias se hace uso de las clases y las funciones de llamada, de forma que una instancia sea una llamada a las clases.

El primer paso es hacer que la clase `LoxClass` implemente `LoxCalleable` y por ello definir las funciones *call* y *arity* heredadas. La función *call* crea y retorna una instancia de la propia clase, mientras que *arity* retorna directamente 0 ya que la llamada a la clase no recibe argumentos.

Toca crear la clase `LoxInstance` en un nuevo fichero `LoxInstance.py`. De momento solo se define un `__init__` en el que inicializamos una clase como atributo, y un `__repr__` que imprima el nombre de la clase seguido de " instance".

### 5.4.4. *Propiedades de las Instancias*

Cada instancia es una colección de valores nombrados. Los métodos tanto de dentro como de fuera de la clase pueden modificar los valores o propiedades de las clases. Si estas propiedades se acceden desde fuera se utiliza el carácter `“.”`. Este punto tiene la misma precedencia que el paréntesis de una llamada a una función por lo que se modifica la regla de llamada para incluirlo.

$$\text{call} \rightarrow \text{primary} ( ( ( \text{arguments? } ) ) \mid ( \text{IDENTIFIER } )^* ;$$

#### *Expresiones Get*

Se añade un nodo al árbol sintáctico y se modifica el parser para que en la función *call* si detecta `“.”` cree una expresión *Get* con el identificador que continua el punto.

De igual manera se añade el método *visit\_get\_expr* tanto en el Resolver como en el Intérprete. Las propiedades no son resueltas ya que se visitan de forma dinámica. Por otro lado, en el intérprete se evalúa la expresión y si el resultado es del tipo `LoxInstance` se trata de acceder a la propiedad, si no lo fuera se devuelve un error.

Es momento de añadir estados a las instancias. Se crea un diccionario en `LoxInstance` que inicializamos vacío en el constructor.

Definimos el método *get* mediante el que se accede a las propiedades de la instancia.

#### *Expresiones Set*

Los «setters» tienen la misma sintaxis que los «getters» pero se sitúan en el lado izquierdo de la asignación. Por tanto, se modifica la regla de la gramática correspondiente a la asignación de la siguiente manera.

$$\text{assignment} \rightarrow ( \text{call } ( \text{IDENTIFIER } )^* \text{ "}" } \mid \text{logic\_or} ;$$

A diferencia de los getters, los setters no encadenan. Si tengo una sentencia “libro.novela.genero = romance” solo “genero” actúa como setter.

Se añade un nuevo nodo al generador AST.

Para el parser se hace lo siguiente. Se parsea la parte izquierda de la asignación como una expresión normal y en el momento en el que llegamos al símbolo de igualdad se toma esta expresión ya parseada y se transforma en el nodo del árbol sintáctico correcto para la asignación.

En el Resolver se añade *visit\_set\_expr* que resuelve el valor y el objeto a asignar.

En el Intérprete se evalúa el objeto cuya propiedad se va a modificar y se comprueba si es del tipo *LoxInstance*. Si no lo es se genera un error. Si lo es se evalúa el valor que se va a asignar y se guarda en la instancia mediante un método *set* que añadimos a la clase *LoxInstance* para añadir propiedades y sus respectivos valores al diccionario *fields*.

#### 5.4.5. *Métodos en las Clases*

En este punto se pueden crear instancias de clases y almacenar datos en ellas, pero las clases en sí no hacen nada. Las instancias no son más que simples diccionarios. El siguiente paso es añadir métodos de comportamiento.

En nuestra función *visit\_class\_stmt* del Resolver, añadimos un bucle que recorra los métodos de la clase, resolviéndolos pasándole como argumento un nuevo valor del enumerado *FunctionType* “METHOD”.

Ahora se pasa a modificar el Intérprete. En la misma función *visit\_class\_stmt* recorreremos de igual modo los métodos de la clase, esta vez creando un objeto *LoxFunction* para cada uno de los métodos y añadiéndolos a un diccionario *methods* que se pasará como parámetro en el momento de creación de la clase.

Para que esto funcione se modifica *LoxClass* para recibir en el constructor este diccionario de funciones.

Las clases guardan los métodos mientras que sus instancias guardan los atributos o propiedades. Pero aún así, se accede a estos métodos a partir de las instancias, por lo que se añade al método *get* de *LoxInstance* una búsqueda del método en cuestión en la clase que contiene como atributo mediante un nuevo método llamado *findMethod*. Este método se crea en la clase *LoxClass* y no hace más que buscar en el diccionario *methods* un método específico.

#### 5.4.6. *This*

Lo siguiente es añadir una manera de acceder a las propiedades de la clase desde los métodos de esta, es decir el “this” de Java o el “self” de python.

Se añade un nuevo nodo a la sintaxis y se modifica el método *primary* del parser para reconocer la palabra clave **this**.

En el resolver se crea un método *visit\_this\_expr* que llama a *resolveLocal*, pero *this* no es una variable que esté en el scope por lo que se modifica *visit\_class\_stmt* para que después de definir la variable,

abra un nuevo scope y añada `this` a este, y antes de terminar cierre este nuevo scope.

Se modifica en `get` de `LoxInstance` la línea que retornaba el método si este estaba vacío para que devuelva la llamada a `bind(self)` del método. La función `bind` de `LoxFunction` crea un nuevo entorno en el que define “`this`” y retorna una función pasándole la declaración de la función y este nuevo entorno a su constructor.

Por último se crea el método `visit_this_expr` en el Intérprete para que busca la variable.

### *Usos no válidos de This*

¿Qué ocurre si se intenta acceder a `this` fuera de un método? esto no debería poderse hacer, se tiene que resolver este problema.

Para ello se crea un nuevo enumerado en el Resolver llamado `ClassType` que diferencia entre clases y no clases, y añadimos un atributo al resolver de este tipo inicializado a `NONE`. Este atributo lo cambiaremos a `CLASS` cuando se entra a `visit_class_expr` y se retornará a su estado previo antes de salir de este mismo método.

En `visit_this_expr` ahora se comprobará el tipo de clase actual y en caso de no ser de tipo clase se generará un error.

### 5.4.7. Constructores e Inicializadores

La construcción de un objeto se puede separar en dos partes:

- Se reserva espacio en memoria para la nueva instancia.
- Se hace una llamada a un código que inicializa el objeto por formar.

Toca crear el **constructor** de nuestras clases. Cuando una clase es llamada, justo después de crear la nueva instancia se busca un método «`init`», utilizando el formato que sigue python para los constructores, y si se encuentra inmediatamente se vincula y se invoca pasándole la lista de argumentos. Ahora que la clase llama al constructor y esta puede recibir argumentos, se tiene que modificar el método de aridad de la clase que se tenía puesto a un valor fijo de 0.

### *Invocando init directamente*

En caso de llamarse directamente al método `init` de una clase ya inicializada se ha decidido retornar el valor `this`. Para esto se modifica `LoxFunction` para que en el método `call` compruebe antes de retornar un valor si la función es una función de inicialización, si lo es retorna `this`, si no lo es retorna el valor a devolver.

La manera de comprobar si es una función de inicialización es crear un nuevo atributo booleano `isInitializer` que lo indique, y modificar el constructor de `LoxFunction` para recibir este booleano como argumento para su constructor.

### ***Retornando de `init()`***

¿Qué pasaría si se tratase de retornar un valor en el constructor? Esto es algo que normalmente se trata de evitar ya que el constructor debe devolver el objeto creado.

La solución a este problema es crear un nuevo `FunctionType` en el Resolver “INITIALIZER“, en *visit\_class\_stmt* cuando se comprueban los métodos si el nombre del método es “init“ se declara la función bajo este nuevo tipo, y lanzar un error en *visit\_return\_stmt* si el tipo de la función a retornar es initializer.

Hecho esto todavía no se cubre el caso de hacer un return aislado en la inicialización. Esto se gestiona desde la función `call` de `LoxFunction` cuando se trata la excepción `Return`, si la función actual es inicializadora se busca `this` en el entorno de clausura.

## 6 Testing

Siguiendo el modelo software iterativo incremental, para comprobar que el código funcionaba se han ido utilizando distintos métodos de testeo sobre las distintas partes del intérprete.

### 6.1. *Comprobando la lectura de Tokens*

Para el lexer se crearon test unitarios para cada uno de los casos posibles de tokens que se podían dar bajo distintas circunstancias. Estos test unitarios se realizaban comprobando el resultado obtenido al llamar al programa para una cadena de texto con el resultado esperado directamente escrito en otra cadena de texto. Esta manera resulta un poco rudimentaria aunque funcional por lo que se decidió tomar otro camino para las próximas pruebas a realizar.

### 6.2. *Resolviendo expresiones*

En el caso del parser, al final del capítulo correspondiente del libro [Nystrom \[2021\]](#) se desarrolla un programa ASTPrinter que imprime de manera más vistosa, siguiendo la notación polaca que utilizan lenguajes como LISP, las expresiones resultantes de la llamada al parser, separando operaciones por paréntesis. Con este programa se pudo comprobar que las reglas de precedencia y prioridad se cumplían y se generaban los árboles sintácticos de la manera correcta, sin sobrecargar ramas o tomando operaciones y datos de forma errónea. El programa fue desarrollado en un punto en el que solo existían las expresiones y la prioridad era comprobar si estas se estructuraban correctamente. En el momento en el que se introdujeron las sentencias, este programa se quedó obsoleto y dejó de utilizarse ya que ejecutando programas de prueba se podían comprobar los resultados. De todas formas se ha modificado para que funcione, ahora con el proyecto ya terminado, reteniendo su funcionalidad en el momento en que fue creado.

### 6.3. *Pruebas del Intérprete*

De aquí en adelante, las pruebas se realizaban ejecutando distintos ficheros en el lenguaje Lox que eran llamados desde Lox.py directamente. Desde que se introducen el Interpretador y las sentencias, mediante print, ya se dispone de resultados con los que comprobar la ejecución de ficheros de prueba, por lo que no hacía mucha falta la creación de testers de cualquier otro tipo.

Lo que sí se hizo, ya que era tedioso estar corriendo cada programa de uno en uno o realizar test unitarios como se hizo con el lexer para cada fichero, fue un programa `tester_general.py` que ejecuta todos los distintos ficheros de prueba, escribe el resultado en un fichero del mismo nombre pero formato .out, y lo compara con el resultado esperado que está guardado en otro fichero del mismo nombre pero terminado en .test. Los ficheros de prueba se encuentran todos en la carpeta de nombre “pruebas” y los resultados tanto generados como esperados se encuentran en la misma carpeta dentro de otra de

nombre “resultados”.

Dejando a un lado los tests realizados la clave de la corrección de errores y comprobación de que todo funcionase correctamente ha sido la depuración llevada a cabo desde el debugger para python de Visual Studio Code, software que se ha utilizado durante el desarrollo de todo el proyecto. Este ha sido de gran ayuda, permitiendo comprobar el valor y contenido de toda variable en el momento de ejecución de cada línea de código.

## 7 *Trabajos Futuros y Conclusión*

### 7.1. *Trabajo Futuro*

Hasta aquí llega el contenido de este proyecto, resultando en un Interprete funcional bastante completo del lenguaje Lox, pero el mundo de los Interpretes no termina aquí y es por eso que en este último capítulo comentaré qué se podría haber añadido a este interprete y cuál serían otras alternativas al “tree-walk interpreter“, dado que este es una versión sencilla que se salta algunas de las ‘etapas’ principales en la creación de un interprete.

A nuestro tree-walk interpreter se le podría añadir, como se hace en el libro [Nystrom \[2021\]](#), una funcionalidad que a día de hoy se considera imprescindible en los lenguajes de alto nivel más populares. Esto es la herencia, la capacidad de una clase hijo de adquirir sus características y métodos de otra clase a la que se considera padre y ampliarlos. Esta es una tarea que no debería suponer muchos problemas con las herramientas de las que ya dispone el intérprete. Ya se tienen clases, y una manera de abordar esta funcionalidad podría ser añadir a las clases un atributo padre, que sea la clase de la que hereda, de manera similar a los entornos conteniendo sus entornos de clausura. Si la clase hijo contiene a la clase padre, se podría tratar de acceder a los métodos y características de esta para acceder para ejecutarlos tratando de cubrir las sentencias «super» o comprobar que la clase hijo también los tiene para poder usarlos o incluso modificarlos a forma de «override».

Por otro lado, nuestro intérprete mientras se analiza el árbol sintáctico ya ejecuta el código saltando de la fase de análisis a la fase de runtime, pero para otros tipos de interprete esto no ocurre. Generalmente, los pasos a seguir tras el análisis son los siguientes:

1. **Representación Intermedia:** Representación de nuestro interprete entre el *front end* y *back end*, que nos permite el soporte de múltiples lenguajes fuente, creando un *front end* para cada lenguaje, un *back end* para cada lenguaje y luego conectándolos a gusto.
2. **Optimización:** Optimizar el código. Ejemplo **constant folding**: si una expresión siempre se evalúa para el mismo valor, se puede realizar este calculo en la compilación, y sustituir el código por el resultado.
3. **Generación de código:** Comienza el *back end*. Se transforma el código fuente en código máquina poco a poco, teniendo en cuenta en dónde deseamos ejecutar nuestro programa, CPU específica o una máquina virtual. El código generado para ser ejecutado en una máquina virtual se denomina **bytecode**.
4. **Máquina Virtual:** Se tienen dos opciones una vez se tiene el bytecode: crear otro mini-compilador que traduzca este código al chip específico que se desee; o crear una máquina virtual con un chip hipotético que soporta la arquitectura de tu código. Cómo nuestro interprete traduce a python, uno de los lenguajes más populares en la actualidad, este código podría utilizarse en cualquier máquina que tenga un compilador de python.
5. **Runtime:** ¡Hora de ejecutar el programa y ver el resultado final!

Como atajos o rutas alternativas a estas etapas y las que sigue nuestro intérprete tenemos:

1. **Single-pass compilers:** estos compiladores saltan de la fase de parsing directamente a la generación de código, sin tener estructuras donde guardar valores globales, y sin volver a líneas de código ya visitadas. Se ejecuta en orden. C y Pascal se diseñaron bajo esta manera de actuar.
2. **Transpilers:** a la hora de “bajar de nivel” a nuestro código podemos tratar de generar un código homólogo en otro idioma de alto lenguaje distinto, y utilizar las herramientas de compilación ya existentes de este para generar el código máquina.
3. **Just-in-time compilation:** compila el código fuente en tiempo de ejecución directamente en código máquina nativo para la cpu específica. En ocasiones, se hace desde el bytecode y no desde el código fuente.

## 7.2. Conclusión

Finalmente, remarcar que la implementación en python ha supuesto más problemas de los que se esperaban. Pensaba que no habría mucha diferencia en la manera de actuar entre java y python pero en muchas ocasiones el lenguaje y la arquitectura que hay detrás no funcionaban de la misma manera resultando en fallos que hicieron que se tuviese que tomar otro camino.

También quisiera remarcar las partes en las que más tiempo se ha invertido y a las que más vueltas se ha dado para lograr implementarlas de manera satisfactoria, con visión a tenerlo en cuenta en la impartición en clases de las mismas.

Sin duda alguna, lo más complicado ha sido lograr la correcta gestión de los entornos, scopes y cierres. En una primera instancia se trató de seguir la implementación que se plantea en [Nystrom \[2021\]](#) pero esta no se logró hacer funcionar, fue un auténtico quebradero de cabeza. El guardado y recuperación del contexto al entrar y salir de un scope y la gestión de los entornos de clausura dieron muchos problemas. Es por esto que en esta parte se cambió el planteamiento. Además, esta gestión de los entornos no solo ha dado problemas en el momento de su implementación, si no que ha sido el problema principal a lo largo del resto de apartados, las funciones recursivas no encontraban las variables correctas y cuando se añadió la resolución de las variables también se tuvo problemas buscando entre los entornos. Por otro lado el lexer, el parser y el intérprete en sí, llevaron bastante tiempo pero no dieron demasiados problemas. Si se sigue el libro con calma y pensando las cosas, se avanza a buen ritmo.

Por último, como reflexión personal, decir que a pesar de las complicaciones estoy contento y muy orgulloso del trabajo realizado y los resultados obtenidos. En ocasiones me he atascado con alguna parte y he estado días e incluso semanas sin avanzar nada de nada, bloqueado por completo, pero el momento en el que todo hacía click y solucionaba el problema me producía una satisfacción que eclipsaba el mal rato pasado. Además, me ha hecho repasar y ampliar conceptos dados a lo largo de la carrera que ya apenas recordaba y me ha dado un conocimiento más detallado de como funcionan las herramientas que he utilizado durante todos estos años. Repito que estoy muy satisfecho con el trabajo realizado y me quedo tanto con lo bueno como con lo malo de esta experiencia.



“You should enjoy the little detours to the fullest. Because that’s where you’ll find the things more important than what you want.”

---

*Yoshihiro Togashi*



# Referencias

- H. Abelson y G. J. Sussman. *Structure and interpretation of computer programs*. The MIT Press, 1996.
- J. E. Hopcroft, R. Motwani y J. D. Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001.
- R. Nystrom. *Crafting interpreters*. Genever Benning, 2021.



## *Apéndice 1: Expresiones Regulares TokenType*

Las expresiones regulares para los distintos tipos de tokens, excluyendo keywords ya que estas no utilizan expresiones regulares, son las siguientes:

- COMMENT:

$\backslash\backslash/.*$

- STRING:

$('.*')|('.*')$

- NUMBER:

$[0-9]*\.[0-9]^+$

- IDENTIFICADOR:

$(\backslash w|_)+$

Previo a detectar estos tipos de token se comprueba si se trata de un comentario con la siguiente expresión:

- COMMENT:

$\backslash\backslash/.*$

Si coincide es ignorado al igual que cuando detecta un espacio o un salto de línea. Estos modifican contadores pero no devuelven ningún token.



## *Apéndice 2: Gramática final*

Esta es la gramática final resultante:

```
_program → declaration* EOF ;
declaration → classDecl
              | funDecl
              | varDecl
              | statement ;
classDecl → “class“ IDENTIFIER ““ function* ““ ;
funDecl → “fun“ function ;
function → IDENTIFIER “(“ parameters? “)” block ;
parameters → IDENTIFIER ( “,” IDENTIFIER )* ;
varDecl → “var“ IDENTIFIER ( “=” expression )? “;“ ;
statement → exprStmt
            | forStmt
            | ifStmt
            | printStmt
            | returnStmt
            | whileStmt
            | block ;
exprStmt → expression “;“ ;
forStmt → “for“ “(“ (varDecl | exprStmt | “;“ )
           expression? “;“
           expression? “)” statement ;
ifStmt → “if“ “(“ expression “)” statement
         | ( “else“ statement )? ;
printStmt → “print“ expression “;“ ;
returnStmt → “return“ expression? “;“ ;
whileStmt → “while“ “(“ expression “)” statement ;
block → ““ declaration* ““ ;
```

```

expression → assignment ;
assignment → ( call “.” )? IDENTIFIER “=” assignment
              | logic_or ;
logic_or → logic_and ( “or“ logic_and )* ;
logic_and → equality ( “and“ equality )* ;
equality → comparison ( ( “!=“ | “==“ ) comparison )* ;
comparison → term ( ( “>“ | “>=“ | “<“ | “<=“ ) term )* ;
term → (factor ( ( “-“ | “+“ ) factor ))* ;
factor → unary ( ( “/“ | “*“ ) unary )* ;
unary → ( “! “ | “-“ ) unary | call ;
call → primary ( “(“ arguments? “)” | “:“ IDENTIFIER )* ;
arguments → expression ( “,” expression )* ;
primary → NUMBER | STRING | “true“ | “false“ | “nil“
          | “(“ expression “)”
          | IDENTIFIER ;

```