Universidad de Cantabria

Facultad de Ciencias

LENGUAJE DE DOMINIO PARA ESPECIFICACIÓN DE GEOMETRÍA CONSTRUCTIVA

Domain-Specific Language for Constructive Geometry Specification

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Manuel Murillo García

Director: Domingo Gómez Pérez

Co-Director: Esteban Stafford Fernández

Julio - 2025

Agradecimientos

Gracias a mi familia, amigos y profesores que me han apoyado y acompañado en este viaje.

Lenguaje de Dominio para Especificación de Geometría Constructiva

Resumen

palabras clave: Lexer, Parser, Python3, Intérprete, Geometría Constructiva

En este Trabajo de Fin de Grado se desarrolla un lenguaje de dominio específico orientado a la definición constructiva de geometrías a partir de cuerpos sencillos. El proyecto incluye la implementación de un intérprete para dicho lenguaje y el diseño de una interfaz gráfica básica que permite visualizar, editar e interactuar con los objetos generados por el usuario.

El lenguaje diseñado incluye las funcionalidades básicas de un lenguaje de programación como asignaciones, bucles, operaciones aritméticas y definición de funciones, así como un conjunto de funciones predefinidas para la creación de cuerpos primitivos y su manipulación mediante transformaciones afines (traslación, escalado y rotación), permitiendo así la construcción de geometrías complejas mediante técnicas de Geometría Constructiva.

El intérprete ha sido implementado en Python, utilizando un analizador léxico y sintáctico (Lexer y Parser). Con el apoyo de librerías especializadas en gráficos 3D, es capaz de generar los objetos definidos, calcular sus dimensiones reales, coordenadas cartesianas y otros aspectos técnicos relevantes.

Domain-Specific Language for Constructive Geometry Specification

Abstract

keywords: Lexer, Parser, Python3, Interpreter, Constructive Geometry

This Final Degree Proyect presents the development of a domain-specific language aimed at constructive geometry definition based on simple solid shapes. The project includes the implementation of an interpreter for this language and the design of a basic graphical interface that allows users to visualize, edit, and interact with the generated objects.

The designed language incorporates core features found in most programming languages, such as assignments, loops, arithmetic operations, and function definitions, as well as a set of predefined functions for creating primitive solids and manipulating them through affine transformations (translation, scaling, and rotation). This enables the construction of complex geometries using Constructive Solid Geometry techniques.

The interpreter has been implemented in Python, using a lexical and syntactic analyzer (Lexer and Parser). With the help of specialized 3D graphics libraries, it is capable of generating the defined objects, calculating their real-world dimensions, Cartesian coordinates, and other relevant technical properties.

Índice general

1. Introducción **1**
- 1.1. Motivación . . . 1
- 1.2. Objetivos . . . 2

2. Herramientas y Métodos utilizados **3**
- 2.1. Metodología . . . 3
- 2.2. Planificación . . . 3
  - 2.2.1. Planificación Inicial . . . 4
  - 2.2.2. Desarrollo . . . 4
  - 2.2.3. Memoria . . . 4
- 2.3. Herramientas y Tecnologías empleadas . . . 4

3. Análisis de Requisitos **7**
- 3.1. Visión General . . . 7
- 3.2. Requisitos Funcionales . . . 8
- 3.3. Requisitos No Funcionales . . . 9

4. Diseño del Lenguaje de dominio **11**
- 4.1. Análisis Léxico . . . 11
  - 4.1.1. Lexer . . . 11
- 4.2. Análisis Sintáctico . . . 14
  - 4.2.1. Tipos de Datos . . . 14
  - 4.2.2. Tipos de Operaciones . . . 15
  - 4.2.3. Reglas Gramaticales . . . 18
  - 4.2.4. Análisis Semántico . . . 21

5. Diseño de la Aplicación **23**
- 5.1. Arquitectura . . . 23
  - 5.1.1. Estructura General . . . 23
  - 5.1.2. Flujo General de Uso . . . 24
- 5.2. Diseño de la Interfaz . . . 25
  - 5.2.1. Componentes Principales . . . 25
  - 5.2.2. Limitaciones y Futuras Mejoras . . . 26

6. Pruebas **27**
- 6.1. Pruebas del Lenguaje de Dominio . . . 27
  - 6.1.1. Pruebas unitarias del Lexer . . . 27
  - 6.1.2. Pruebas unitarias del Parser . . . 29
  - 6.1.3. Pruebas de integración del lenguaje de dominio . . . 31
- 6.2. Pruebas de la Aplicación . . . 32
  - 6.2.1. Pruebas unitarias . . . 32
  - 6.2.2. Pruebas de integración . . . 33

6.3. Pruebas de sistema de la Aplicación 34
6.3.1. Pruebas de portabilidad 34
6.3.2. Pruebas de usabilidad 35
6.3.3. Pruebas de escalabilidad 36
6.4. Pruebas de aceptación de la Aplicación 36

7. Conclusiones **37**
7.1. Conclusión 37
7.2. Trabajo Futuro 37

Bibliografía **39**

A. Apéndice 1: Análisis Léxico (Lexer) Ampliación **41**

B. Apéndice 2: Análisis Sintáctico (Parser) Ampliación **43**

1 Introducción

1.1. Motivación

La representación y manipulación de objetos tridimensionales es una necesidad fundamental en múltiples campos, como la ingeniería, la animación digital, la fabricación aditiva y la simulación. Para abordar esto, se han creado diversas metodologías para representar y manipular modelos 3D, cada una adaptada a sus distintas necesidades y requerimientos.

En cuanto a la representación geométrica sólida, existen principalmente dos enfoques: la *Representación Por Fronteras (Boundary Representation, B-Rep)*, que describe objetos a través de sus superficies, y la *Geometría Constructiva (Constructive Solid Geometry, CSG)*, que construye modelos combinando sólidos primitivos mediante operaciones booleanas. Aunque ciertamente B-Rep es más flexible y preciso para geometrías complejas, su implementación resulta notablemente más compleja. Por ello, CSG ha sido históricamente más utilizado en entornos educativos, prototipado, diseño conceptual y en herramientas CAD de nivel medio o básico, donde la claridad estructural y la facilidad de construcción jerárquica son prioritarias, como bien se describe en el artículo de Requicha [1980].

Herramientas como AutoCAD han permitido desde sus primeras versiones la construcción de geometría 2D (y posteriormente 3D) mediante secuencias de comandos. Aunque AutoCAD está basado principalmente en la representación por B-Rep, también utiliza operaciones derivadas de CSG para ciertas funciones booleanas y modelado sólido. Sin embargo, uno de sus límites tradicionalmente fue la falta de una representación estructurada, modificable y reutilizable del proceso de modelado. Esta limitación ha sido parcialmente superada en versiones más recientes mediante tecnologías como Bloques Dinámicos, vinculación con parámetros y lenguajes de scripting integrados (como por ejemplo AutoLISP y .NET APIs), que permiten almacenar, editar y reutilizar definiciones geométricas y secuencias de comandos. En el artículo de Zou et al. [2023] se respalda la evolución de sistemas que combinan B-Rep y CSG, justo en la línea de las mejoras en AutoCAD y herramientas similares.

Una versión más sencilla de esta capacidad de representación estructurada y reutilizable también se aborda en el lenguaje de dominio específico que se desarrolla en este trabajo, facilitando la definición, modificación y reutilización de modelos mediante scripts claros y estructurados.

Por otro lado, uno de los enfoques más avanzados es el **Diseño Basado En Restricciones (Constraint-Driven Design)**, ampliamente utilizado en sistemas CAD paramétricos como SolidWorks, Siemens NX o Varkon. En este caso, la geometría se define mediante relaciones entre entidades que pueden ser modificadas dinámicamente, permitiendo una actualización automática del modelo en función de nuevas condiciones; esta metodología se describe más detalladamente en el artículo escrito por Ohlbrock et al. [2017]. Esta flexibilidad resulta ideal en procesos de diseño iterativo, donde los requisitos cambian constantemente.

En este contexto, en este trabajo se busca abordar esta necesidad desde un enfoque más ac-

cesible y simplificado. Para ello, se desarrolla un lenguaje de dominio específico centrado en la definición de geometrías constructivas a partir de cuerpos primitivos y transformaciones básicas (traslación, escalado y rotación). Este lenguaje incluye también las funcionalidades básicas de programación, como asignaciones, bucles y definición de funciones, junto con funciones predefinidas para la creación y manipulación de sólidos sencillos, facilitando así la construcción de modelos 3D mediante scripts claros y estructurados.

Para dar soporte a este lenguaje, se implementa también un intérprete que, con la ayuda de librerías especializadas en gráficos 3D, permite generar los objetos definidos, calcular sus propiedades y representar sus coordenadas cartesianas. Además, se diseña también una interfaz gráfica básica que facilita la visualización e interacción con los modelos creados, completando así una herramienta sencilla pero potente para el diseño geométrico tridimensional.

Este proyecto constituye la primera fase para sentar las bases de un sistema que combine la claridad y expresividad del modelado por comandos con la potencia estructural de la Geometría Constructiva, orientado tanto a fines educativos como a aplicaciones de diseño conceptual.

1.2. Objetivos

El objetivo principal de este Trabajo de Fin de Grado es el diseño e implementación de un lenguaje de dominio específico para la definición programática de geometrías tridimensionales basadas en cuerpos sencillos, mediante operaciones booleanas y transformaciones afines.

Para alcanzar este objetivo general, se plantean los siguientes objetivos específicos:

- Diseñar un lenguaje que incorpore funcionalidades básicas de programación como asignaciones, estructuras de control (bucles y condicionales), operaciones aritméticas y definición de funciones, junto con un conjunto de funciones predefinidas para la creación y manipulación de sólidos primitivos.
- Implementar un intérprete capaz de procesar este lenguaje, generando las estructuras geométricas correspondientes, calculando sus propiedades y facilitando la reutilización y modificación del código.
- Desarrollar una interfaz gráfica básica que permita visualizar y editar los modelos 3D definidos mediante scripts, proporcionando al usuario herramientas para interactuar mediante transformaciones euclídeas.
- Sentar las bases para una futura ampliación del sistema hacia funcionalidades más complejas, como la integración de restricciones geométricas o la mejora en la representación visual y la usabilidad.

Con este proyecto se pretende crear una herramienta accesible que combine la expresividad de un lenguaje formal con la claridad y simplicidad de la Geometría Constructiva, que sirva tanto para explorar conceptos de programación y geometría computacional como para desarrollar prototipos de modelos tridimensionales de forma accesible y flexible.

2 Herramientas y Métodos utilizados

En este apartado se detallan distintos aspectos vinculados con la realización del presente proyecto, entre ellos la metodología empleada, la planificación de las tareas y las herramientas y tecnologías utilizadas.

2.1. Metodología

Para el desarrollo de este Trabajo de Fin de Grado se ha seguido una metodología iterativa e incremental, combinando fases de análisis, diseño, implementación y validación. Dado que el proyecto se centra en el diseño de un lenguaje de dominio específico, se ha puesto especial atención en la definición clara de la gramática, el diseño del intérprete y la prueba de los scripts a través de casos prácticos.

El trabajo se ha dividido en varios bloques: definición del lenguaje, desarrollo del intérprete, creación de la interfaz gráfica y pruebas de uso. Cada uno de estos bloques ha seguido su propio ciclo de desarrollo y pruebas, permitiendo ajustar errores y mejorar el diseño a lo largo del proceso. Esta metodología no solo ha permitido flexibilidad durante el desarrollo del proyecto, sino que también ha ayudado a construir una base sólida y extensible para futuras funcionalidades, especialmente en el desarrollo de operaciones booleanas y mejoras visuales.

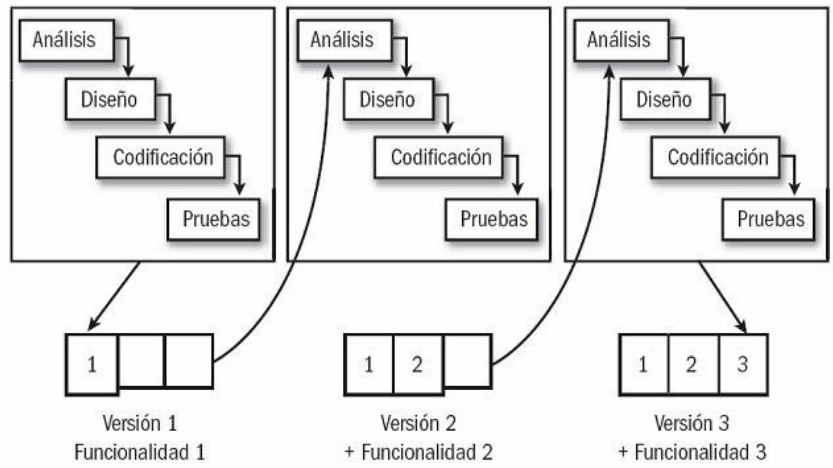

Figura 1: Representación de la metodología iterativa e incremental. Fuente: Micolini et al. [2022].

2.2. Planificación

El proyecto se desarrolla en 3 etapas distintas: planificación inicial, desarrollo y redacción de memoria.

2.2.1. Planificación Inicial

La planificación inicial del proyecto se estructuró en varias fases claramente definidas:

- Reunión inicial con el tutor en la cual se estableció el tema principal del proyecto y se compartieron distintas propuestas iniciales, además de que se compartió la idea de un diseño realizado por Esteban Stafford Fernández en Perl y C++ que se puede observar en la carpeta antecedentes del repositorio git del proyecto.
- Investigación previa sobre metodologías de modelado geométrico (CSG, B-Rep, diseño paramétrico, etc.) y análisis de herramientas y librerías capaces de visualizar geometría 3D.
- Diseño del lenguaje DSL, definiendo su sintaxis, estructuras básicas, primitivas geométricas y transformaciones soportadas.
- Implementación del intérprete, incluyendo el desarrollo del lexer y parser utilizando herramientas en Python.
- Diseño de una interfaz gráfica, que permite al usuario visualizar los modelos y modificar de forma sencilla los scripts.
- Validación y pruebas, con la ejecución de ejemplos de scripts para verificar el funcionamiento del lenguaje e intérprete.
- Redacción de la memoria, incluyendo documentación técnica y ejemplos prácticos.

2.2.2. Desarrollo

Como se mencionó anteriormente, se ha seguido una metodología iterativa e incremental mediante una estrategia centrada en cumplir objetivos específicos en cada fase. Esto ha permitido probar y validar tanto el funcionamiento del intérprete como la coherencia del diseño del lenguaje.

La interfaz gráfica ha sido desarrollada en paralelo con el intérprete, empleando librerías de visualización 3D para verificar la correcta representación de los objetos definidos por el usuario. Aunque básica, esta interfaz sienta las bases para una aplicación más completa en futuras versiones.

2.2.3. Memoria

Aunque la memoria se ha realizado al final del proyecto, se ha hecho un seguimiento regular mediante informes de forma progresiva a lo largo del desarrollo. En ellos se recogen las decisiones de diseño, la justificación de las tecnologías empleadas, la especificación del lenguaje, los requisitos definidos y una serie de ejemplos de uso del sistema. Además, incluyen reflexiones sobre posibles mejoras futuras y líneas de continuación del trabajo. Todo esto ha permitido mayor fluidez al escribir la memoria.

2.3. Herramientas y Tecnologías empleadas

A continuación se detallan las principales herramientas y tecnologías utilizadas en este proyecto:

- **Lenguaje de programación**: Para el desarrollo del lenguaje y su intérprete, se evaluaron varias opciones en cuanto al lenguaje de programación, considerando principalmente Python, Java y Perl. Aunque Perl fue utilizado en el ejemplo de referencia presentado en el apartado 2.2.1, fue descartado por mi falta de experiencia previa con dicho lenguaje, lo que podría haber acabado ralentizando el desarrollo. Por otro lado, Java fue una opción atractiva debido a la experiencia adquirida en la asignatura de *Gráficos por Computador y Realidad Virtual*, en la que se realizaron visualizaciones de objetos en 2D. Sin embargo, Python fue finalmente la opción elegida debido a varios factores clave:

  - Experiencia previa en la construcción de analizadores léxicos y sintácticos en Python, adquirida en la asignatura de *Lenguajes de Programación*.
  - Disponibilidad de librerías potentes y accesibles para la representación de gráficos en 3D.
  - Facilidad de desarrollo rápido y lectura clara del código, especialmente útil en prototipado y en el diseño de lenguajes de dominio específico.

  De esta forma, Python proporcionó el equilibrio ideal entre facilidad de desarrollo, soporte para gráficos 3D y experiencia previa, siendo la base sobre la cual se implementaron tanto el intérprete como la interfaz gráfica de usuario.

- **Análisis léxico y sintáctico**: Para implementar el lexer y el parser se ha utilizado la librería SLY, que permite definir reglas léxicas y sintácticas directamente en código Python.

  Se optó por esta herramienta por dos razones principales: debido a su claridad y facilidad de integración, además de mi experiencia previa con su uso, lo que facilitó su adopción en este trabajo, y a que implementar desde cero un analizador léxico y sintáctico supone una tarea altamente compleja, ya que requiere definir manualmente reglas detalladas para reconocer cada tipo de token, construir estructuras gramaticales mediante lógica condicional y recursiva, y gestionar posibles errores o ambigüedades sintácticas. Este proceso no solo es propenso a errores, sino también difícil de mantener y extender.

  Por ello, el uso de bibliotecas como SLY permite automatizar gran parte de esta lógica, facilitando el desarrollo de analizadores robustos, reutilizables y más sostenibles a lo largo del tiempo.

- **Representación y visualización 3D**: Para la visualización de los modelos geométricos generados mediante el lenguaje diseñado, se ha utilizado la librería VPython. Esta herramienta permite crear representaciones tridimensionales de manera sencilla y directa dentro del entorno de Python, facilitando la visualización interactiva sin necesidad de configurar entornos gráficos complejos.

  VPython ofrece clases predefinidas para cuerpos geométricos como cubos, cilindros y esferas, lo que la convierte en una herramienta especialmente adecuada para un proyecto centrado en geometría constructiva. Estas clases han servido como base para la implementación de varios de los objetos definidos en el lenguaje desarrollado. Además, permite manipular de forma intuitiva propiedades como la posición, el tamaño o el color de los objetos, lo que ha resultado fundamental para enlazar la salida del intérprete con una representación visual clara, interactiva y comprensible.

  Esta elección responde tanto a criterios de simplicidad como a la necesidad de contar con una herramienta eficaz para validar visualmente los resultados de los scripts generados por el lenguaje.

- **Diseño de la interfaz gráfica**: Para el desarrollo de la interfaz gráfica se ha utilizado la librería Tkinter, incluida por defecto en Python, lo que facilita la portabilidad y reduce dependencias externas. Su uso ha permitido construir una interfaz simple y funcional sin

necesidad de herramientas externas más complejas, permitiendo al usuario funcionalidades como cargar, editar y visualizar los modelos generados a partir de los scripts.

- **Control de versiones**: Se ha usado Git en BitBucket (un servicio de alojamiento de repositorios Git) como sistema de control de versiones para organizar el desarrollo del código y facilitar el seguimiento del proyecto.
- **Documentación**: La memoria se ha redactado en LaTeX, permitiendo un formato limpio, estructurado y adecuado para la entrega del TFG.

3 Análisis de Requisitos

A continuación, se ofrece una visión general de la aplicación, describiendo de forma integral los requisitos funcionales y no funcionales que debe cumplir.

3.1. Visión General

ScriptyCAD será una aplicación diseñada para permitir la creación, manipulación y visualización de modelos tridimensionales que sientan las bases para la ampliación de funciones de Geometría Constructiva. La herramienta se basa en un lenguaje de dominio específico que facilita la definición de cuerpos primitivos y su combinación mediante operaciones booleanas. El sistema también incluye un intérprete que traduce dichos scripts en estructuras geométricas visualizables a través de una interfaz gráfica básica. Como se menciona en el capítulo 2, existieron desarrollos previos de esta aplicación implementados en Perl. Sin embargo, dichos antecedentes presentaban varias limitaciones significativas, como la ausencia de una estructura formal para la definición de funciones y la imposibilidad de reutilizarlas en otros scripts, entre otras carencias funcionales.

La Geometría Constructiva es un enfoque que permite construir modelos 3D complejos a partir de combinaciones de volúmenes simples como cubos, cilindros o esferas, mediante operaciones como unión, intersección o diferencia. Este método resulta especialmente útil en contextos donde se requiere una representación sintética y estructurada del proceso de modelado, como ocurre en entornos CAD sencillos o automatizados.

El núcleo de la aplicación es el lenguaje diseñado específicamente para describir operaciones de construcción geométrica. Este lenguaje permite utilizar estructuras comunes en programación como asignaciones, bucles y funciones, además de incluir instrucciones predefinidas para la creación y transformación de sólidos básicos mediante traslaciones, rotaciones y escalados. Así, el usuario puede construir modelos complejos de forma programática y reutilizable.

El intérprete, implementado en Python, se encarga de analizar los scripts escritos en el lenguaje mediante un analizador léxico y sintáctico, generando una representación interna de los objetos definidos. Gracias al uso de librerías especializadas para gráficos tridimensionales, es posible visualizar el resultado y calcular propiedades técnicas como coordenadas cartesianas o dimensiones físicas de los objetos.

La interfaz de usuario se ha diseñado de forma sencilla para facilitar la edición de scripts, su ejecución y la visualización del resultado geométrico. Aunque su funcionalidad es básica, esta interfaz cumple el objetivo de proporcionar una primera experiencia de interacción con el lenguaje. En proyectos futuros se prevé el desarrollo de una interfaz más completa que permita edición gráfica, navegación estructurada del modelo y depuración del código.

En resumen, *ScriptyCAD* se centra en el desarrollo del lenguaje de dominio específico y su intérprete, constituyendo una base técnica sólida sobre la que construir una herramienta CAD

programática basada en geometría constructiva. La aplicación actual permite la edición y ejecución de scripts geométricos, visualización 3D del resultado y sienta las bases para futuras ampliaciones orientadas a mejorar la experiencia del usuario.

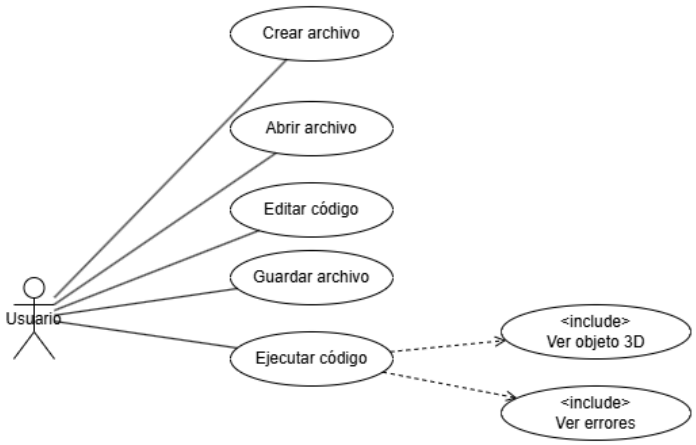

Figura 2: Casos de Uso de *ScriptyCAD*.

3.2. Requisitos Funcionales

Id	Descripción
RF1	**Definición de objetos primitivos:** Creación de cuerpos básicos tridimensionales como cubos, cilindros y esferas.
RF2	**Operaciones booleanas sobre cuerpos:** Se debe permitir la construcción de geometrías complejas mediante operaciones booleanas entre cuerpos primitivos.
RF3	**Transformaciones geométricas:** Aplicación de transformaciones afines como traslación (translate), escalado (scale), y rotación (rotate).
RF4	**Referencia a entidades:** Transformaciones basadas en la posición de otros cuerpos (por ejemplo, alinear con otro objeto).
RF5	**Definición de funciones personalizadas:** Creación de funciones propias para operaciones geométricas repetidas o complejas.
RF6	**Control de flujo:** Estructuras como bucles (for, while) y condicionales (if, else).
RF7	**Asignación de variables y operaciones aritméticas:** Asignación de valores a variables y operaciones matemáticas básicas.
RF8	**Visualización de modelos 3D:** La aplicación debe representar gráficamente los modelos generados.
RF9	**Cálculo de propiedades geométricas:** Cálculo de volumen, superficies, dimensiones, coordenadas de vértices, etc.

Cuadro 1: Requisitos Funcionales de *ScriptyCAD*

3.3. Requisitos No Funcionales

Id	Descripción
RNF1	**Modularidad:** El sistema debe estar estructurado en módulos independientes (intérprete, motor gráfico, interfaz de usuario).
RNF2	**Usabilidad:** La interfaz debe ser clara y sencilla para permitir un uso intuitivo, incluso para usuarios sin experiencia previa.
RNF3	**Portabilidad:** La aplicación debe poder ejecutarse en diferentes sistemas operativos modernos (Windows, Linux, macOS) gracias al uso de tecnologías multiplataforma como Python y bibliotecas compatibles.
RNF4	**Rendimiento aceptable:** La aplicación debe responder de manera fluida en operaciones de modelado de complejidad baja o media.
RNF5	**Extensibilidad:** La arquitectura del lenguaje y del sistema debe permitir la incorporación futura de nuevas operaciones, primitivas o mejoras gráficas.
RNF6	**Documentación básica:** Debe proporcionarse una guía mínima para el uso del lenguaje y la aplicación.
RNF7	**Lenguaje accesible:** El lenguaje diseñado debe ser sintácticamente sencillo, similar a otros lenguajes de programación conocidos.

Cuadro 2: Requisitos No Funcionales de *ScriptyCAD*

4 Diseño del Lenguaje de dominio

4.1. Análisis Léxico

El análisis léxico es la primera fase del procesamiento del lenguaje, y tiene como objetivo transformar la entrada textual en una secuencia de tokens que puedan ser interpretados por el parser. Este proceso es conocido también por el nombre *tokenizado*.

4.1.1. Lexer

En este trabajo se ha diseñado una clase denominada *SimpleLexer* que extiende la clase *Lexer* proporcionada por la librería SLY. Esta clase se encarga de identificar las palabras clave (*keywords*), identificadores, operadores, literales y signos de puntuación relevantes para el dominio del lenguaje.

```
1  import sys
2  from sly import Lexer
3
4  class SimpleLexer(Lexer):
5      # Token names
6      tokens = { 'NUMBER', 'ID', 'BOOLEAN', 'STRING', 'PLUS', 'LESS', 'MUL',
           'DIV', 'ASSIGN','DEFINE', 'LT', 'LE', 'GT', 'GE', 'EQ', 'NE', 'IF',
           'ELSE', 'RETURN', 'DOT', 'ARROW', 'IMPORT', 'AS', 'WHILE', 'AND',
           'OR'}
7
8      # Ignored characters
9      ignore = ' \t'
10
11     # Token regexs
12     NUMBER = r'(\d)+(\.\d+)?'
13     STRING = r'"([^"\\]|\\"|\\n|\\t)*"'
14     ARROW = r'->'
15     PLUS =r'\+'
16     ...
17     AND = r'&&'
18     OR = r'\|\|'
19     literals = {',', ';', '{', '}', '(', ')', '[', ']', '|'}
20
21     @_("#.*")
22     def Comentario(self, t):
23         pass
24     ...
25     def error(self, t):
26         print(f'Bad character {t.value} in line {t.lineno}')
27         self.index += 1
```

Código 4.1: Código abreviado del Lexer. Fichero: *simplelex.py*

Para identificar cada token, el lexer construye un autómata finito determinista a partir de los atributos y funciones definidos en la clase. Luego, analiza el texto de entrada carácter a carácter, buscando identificar la cadena más larga que coincida con un patrón válido. Si varias expresiones coinciden, se aplica una prioridad: primero se consideran los tokens definidos como variables, seguidos por las funciones decoradas (@_) como el caso de los comentarios que se puede ver en el código 4.1, y en ambos casos se respeta el orden de aparición en la clase. Si no se encuentra ningún patrón válido, se lanza una excepción.

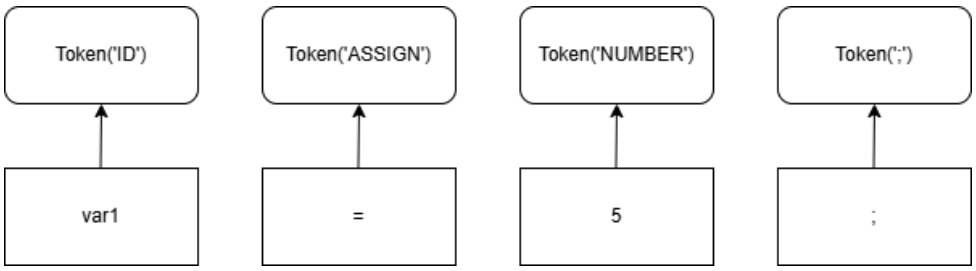

Figura 3: Ejemplo de tokenización del Lexer.

En el caso de nuestro Lexer (*SimpleLexer*), se emplean expresiones regulares para definir cada token del lenguaje. Esta estrategia facilita una integración clara entre el análisis léxico y el posterior análisis sintáctico, manteniendo todo el flujo de procesamiento del lenguaje en una única base de código.

SimpleLexer está diseñado para reconocer y clasificar una amplia variedad de tokens que cubren las necesidades específicas del lenguaje propuesto. Entre los principales tipos de tokens definidos se encuentran los siguientes:

- **Operadores aritméticos y relacionales:**

  $$\text{PLUS } (+), \quad \text{LESS } (-), \quad \text{MUL } (*), \quad \text{DIV } (/)$$
  $$\text{LT } (<), \quad \text{LE } (\leq), \quad \text{GT } (>), \quad \text{GE } (\geq)$$
  $$\text{EQ } (==), \quad \text{NE } (!=), \quad \text{ASSIGN } (=)$$

- **Palabras clave (*keywords*):**

  IF, ELSE, RETURN, IMPORT, AS, WHILE, DEFINE

  Estas palabras clave permiten estructurar programas de forma clara y modular, ofreciendo control de flujo, definición de funciones y mecanismos básicos de modularización y reutilización de código.

- **Operadores lógicos y de sintaxis:**

  $$\text{AND } (\&\&), \quad \text{OR } (||), \quad \text{DOT } (.), \quad \text{ARROW } (->)$$

- **Tipos de identificadores y valores primitivos:**

  ID (identificadores válidos para variables y funciones)

  NUMBER (valores numéricos, tanto enteros como decimales)

  STRING (cadenas de texto delimitadas por comillas)

  BOOLEAN (true, false como valores lógicos booleanos)

- **Símbolos de puntuación:** fundamentales para la definición de bloques de código y listas:

  coma (,), punto y coma (;), llaves {}, paréntesis (), corchetes [], barra vertical (|)

Además, se ha incorporado la funcionalidad de comentarios de una sola línea, los cuales permiten al usuario añadir anotaciones en el código del script que no afectan a la ejecución del programa. Estos comentarios se identifican mediante el símbolo # al inicio de la línea, y el Lexer está configurado para ignorarlos completamente, sin generar tokens asociados a ellos.

El uso de expresiones regulares presenta una considerable complejidad debido a la diversidad de reglas y caracteres involucrados, los cuales pueden variar su significado según el contexto en que se empleen. Debido a la extensión que implicaría detallar el funcionamiento de todas las expresiones regulares utilizadas, se ha optado por explicar únicamente uno de los casos más complejos: la definición del tipo `STRING`.

```
1 STRING = r'"([^"\\]|\\"|\\n|\\t)*"'
```

Código 4.2: Definición de un token `STRING`

En el lenguaje diseñado, un valor de tipo `STRING` se define como cualquier cadena de texto delimitada por comillas dobles (`"`). Estas cadenas pueden incluir comillas dobles internas, siempre que estén correctamente escapadas mediante una barra invertida (`\"`). Este formato es similar al utilizado en lenguajes de programación ampliamente conocidos, como Python. No obstante, una diferencia clave respecto a Python es que en nuestro lenguaje no se permite el uso de comillas simples (') como delimitadores de cadenas; únicamente se aceptan comillas dobles.

Volviendo al ejemplo, la expresión regular se desglosaría de la siguiente forma:

- **`"`:** Comilla doble de apertura, indica el inicio de la cadena.
- **`([^"\\]|\\"|\\n|\\t)*`:** Captura cero o más repeticiones de alguno de los siguientes patrones:
  - **`[^"\\]`:** Cualquier carácter excepto una comilla doble (`"`) o una barra invertida (\), esta última para evitar conflictos con los siguientes patrones.
  - **`\"`:** Una comilla doble escapada.
  - **`\n`:** Representa un salto de línea.
  - **`\t`:** Representa una tabulación horizontal.
- **`"`:** Comilla doble de cierre, indica el final de la cadena.

Esta expresión es capaz de reconocer cadenas con texto normal y ciertos caracteres especiales escapados. Algunos ejemplos serían `"Hello World"`, `"Text with \"quoted\"words"`, `"Line 1\nLine 2"` y `"\tHello\tWorld"`. Las expresiones regulares para detectar el resto de tokens se encuentran en el apéndice A.

En conjunto, esta definición exhaustiva de tokens proporciona una base sólida para el posterior análisis sintáctico y semántico del lenguaje, permitiendo al usuario escribir scripts con una sintaxis clara, expresiva y adaptable a las necesidades del modelado geométrico constructivo.

4.2. Análisis Sintáctico

Una vez realizada la tokenización, el análisis sintáctico se encarga de organizar los tokens en forma de árbol mediante el uso de reglas gramaticales. Esto ha sido implementado con la librería SLY, que proporciona un enfoque moderno y basado en clases para definir analizadores sintácticos de tipo *bottom-up* mediante gramáticas LR(1). SLY permite declarar reglas sintácticas como métodos en una clase que hereda de `Parser`, y utiliza decoradores para asociar cada regla a una acción semántica específica.

En este trabajo, la clase encargada del análisis sintáctico es *SimpleParser*, la cual define todas las reglas gramaticales del lenguaje diseñado. Cabe destacar que, a diferencia de otros enfoques más comunes que construyen árboles de sintaxis abstracta (AST), aquí el parser no genera ningún AST de forma explícita. En su lugar, el parser devuelve **funciones evaluables (closures)** en lugar de nodos del árbol; es decir, cada regla genera una función `f()` que encapsula el comportamiento de la evaluación. Las ventajas que aporta esta implementación con respecto a la tradicional son:

1. **Evaluación inmediata integrada:** Al encapsular la lógica de cálculo en funciones evaluables, el parser produce directamente una estructura que puede ejecutarse para obtener el resultado final, sin necesidad de un evaluador adicional.
2. **Código más compacto y directo:** Se elimina la necesidad de construir y recorrer un árbol intermedio, lo que reduce la cantidad de clases, estructuras auxiliares o código para evaluación posterior y permite tener de forma intrínseca los controles semánticos en las funciones.
3. **Mayor eficiencia en tareas simples:** Para lenguajes simples, este enfoque es más eficiente en términos de ejecución y consumo de memoria al evitar estructuras adicionales.
4. **Abstracción del proceso de ejecución:** Cada función representa un nodo lógico de la operación, encapsulando su comportamiento. Esto favorece una ejecución modular y evita errores al separar la evaluación de forma clara.
5. **Facilidad de implementación inicial:** No es necesario definir estructuras de datos para representar nodos del AST ni crear un visitor o intérprete separado. La evaluación queda resuelta durante el análisis sintáctico.

Además, como parte del proceso de compilación del analizador, SLY genera de forma automática el autómata LR(1) correspondiente a la gramática definida. Este autómata es fundamental para la resolución de ambigüedades y la construcción eficiente del parser. El fichero `parser.out` contiene una descripción detallada de dicho autómata, incluyendo los estados, transiciones y acciones asociadas a cada token y producción. Esto permite inspeccionar y depurar el funcionamiento interno del parser.

Se ha creado también una estructura denominada `scope` que sirve para almacenar tanto las funciones predefinidas del lenguaje como las variables y funciones definidas por el usuario, además de permitir **la independencia de contexto de variables entre funciones**.

Este diseño permite que el parser reconozca la sintaxis del lenguaje y que el análisis semántico y la ejecución se lleven a cabo directamente desde las funciones almacenadas en el entorno.

4.2.1. Tipos de Datos

El lenguaje diseñado para este Trabajo de Fin de Grado incorpora una serie de tipos de datos fundamentales que permiten tanto el manejo de información básica como la construcción

y manipulación de geometría 3D. Estos tipos han sido seleccionados para cubrir las necesidades expresivas del modelado geométrico constructivo, así como para facilitar la programación estructurada de scripts. A continuación se detallan los principales tipos de datos soportados:

- **Number**: Representa valores numéricos, tanto enteros como decimales. Es el tipo más comúnmente utilizado para especificar dimensiones, coordenadas y parámetros geométricos como radios, alturas, longitudes, etc.
- **String**: Cadenas de texto delimitadas por comillas dobles `"..."`. Este tipo se emplea para definir nombres, etiquetas, rutas o para proporcionar descripciones asociadas a objetos.
- **Boolean**: Tipo lógico que admite los valores `true` y `false`. Es especialmente útil para condiciones en estructuras de control (`if`, `while`) y en parámetros opcionales de funciones geométricas o transformaciones.
- **Objetos 3D Simples**:
  - `Box`: Cubo o paralelepípedo definido por sus dimensiones y posición.
  - `Cylinder`: Cilindro definido por su radio y altura.
  - `Sphere`: Esfera definida por su radio.

  Estos objetos son definidos por sus respectivas funciones y pueden ser manipulados mediante transformaciones afines como traslación, rotación y escalado, permitiendo la composición de formas más complejas.
- **Objetos 3D Compuestos (`Group`)**: Este tipo permite representar una fusión de varios objetos 3D en una única entidad geométrica. A diferencia de una simple agrupación visual o jerárquica, un `Group` se comporta como un objeto unificado, resultante de la unión geométrica de todos los elementos que lo componen. Esta operación es conceptualmente equivalente a una operación de `union` en modelado mediante Geometría Constructiva, donde los volúmenes individuales son integrados en un único objeto, coherente y manipulable como una unidad. Esto permite aplicar transformaciones, operaciones booleanas posteriores o exportar el conjunto como un único modelo 3D.
- **Listas**: Colecciones ordenadas de elementos del mismo o distinto tipo. Son útiles para almacenar secuencias de valores o conjuntos de objetos geométricos.
- **Vectores (`Vector`)**: Representan posiciones o direcciones en el espacio tridimensional, definidos mediante tres componentes: `x`, `y` y `z`. Se crean a través de la función `Vector(x, y, z)`, y se utilizan para establecer posiciones, ejes de rotación, direcciones de traslación, etc. Internamente, los vectores permiten realizar operaciones de álgebra lineal como suma, producto escalar y normalización.

Esta combinación de tipos permite al lenguaje expresar de forma concisa tanto lógica de control como geometría tridimensional compleja, lo cual es esencial para lograr una solución flexible, modificable y expresiva en el contexto del diseño 3D basado en comandos.

4.2.2. Tipos de Operaciones

El lenguaje diseñado para este proyecto incluye una variedad de operaciones que permiten tanto el control del flujo de ejecución como la manipulación directa de objetos geométricos. Estas operaciones se dividen en dos grandes categorías: operaciones de control y operaciones geométricas.

Operaciones de Control

Estas operaciones son las que permiten estructurar el flujo del programa, siguiendo los patrones típicos de lenguajes de programación imperativos. Incluyen:

- **Asignación**: Permite almacenar valores o referencias a objetos en variables mediante el operador `=`.
- **Condicionales**: El uso de `if` y `else` permite ejecutar bloques de código en función del resultado de una condición booleana.
- **Bucles**: Se permite la repetición de instrucciones a través de `while`, evaluando una condición antes de cada iteración.
- **Funciones**: Se pueden definir funciones mediante la palabra clave `define`, lo que permite encapsular bloques de código reutilizables con parámetros.
- **Importación**: Es posible importar definiciones externas mediante el uso de `import` y `as`, lo que facilita la modularización del código.

Operaciones Geométricas

Estas operaciones permiten la creación, transformación y combinación de objetos tridimensionales, que constituyen el núcleo del lenguaje en su orientación hacia la Geometría Constructiva.

- **Creación de Objetos3D**: Se incluyen funciones para generar cuerpos básicos como `Box`, `Cylinder` o `Sphere`, cada una con parámetros personalizables (posición, tamaño, etc.).
- **Transformaciones Afines**: El lenguaje proporciona soporte completo para transformaciones afines aplicables a objetos tridimensionales. Estas incluyen:
  - `translate(p1, p2)`: modifica la posición del objeto en el espacio 3D.
  - `scale(num)`: cambia el tamaño del objeto según factores definidos en cada eje.
  - `rotate(edge, ang)`: aplica una rotación alrededor de un eje dado y un ángulo específico.
  - `clone()`: permite duplicar un objeto manteniendo sus propiedades y transformaciones actuales.

  Estas transformaciones pueden aplicarse de dos formas:

  - **Transformación individual**: Cada objeto 3D (por ejemplo, `Box`, `Cylinder`, `Sphere` o `Group`) incluye métodos propios como `obj.translate(p1, p2)`, `obj.scale(num)`, `obj.rotate(edge, ang)`, que permiten modificar directamente sus propiedades espaciales.
  - **Transformación múltiple**: Para casos donde se desea transformar varios objetos a la vez, el lenguaje ofrece funciones globales como `translate(objList, p1, p2)`, `scale(objList, num)` y `rotate(objList, edge, ang)`, las cuales aceptan una lista de objetos como parámetro junto con los argumentos específicos de transformación (vector de traslación, factores de escala o ángulo y eje de rotación). Esto permite aplicar una transformación homogénea a un conjunto de elementos sin necesidad de modificar uno por uno.

  Esta doble modalidad proporciona flexibilidad y eficiencia, favoreciendo la reutilización de código y la creación de estructuras geométricas más dinámicas.

- **Operación de Unión (Group)**: Es posible agrupar varios objetos mediante la clase `Group`. Esta operación es conceptualmente equivalente a una operación de unión en el modelado mediante Geometría Constructiva. Aunque internamente se representa como una agrupación, funcionalmente actúa como un único objeto tridimensional fusionado.

- **Manipulación de Vectores y Listas**: El lenguaje también permite trabajar con listas de objetos y con vectores tridimensionales (instancias de la función `Vector(x, y, z)`), lo cual facilita la creación dinámica y parametrizada de geometrías.

- **Operaciones Aritméticas y Lógicas**: Están disponibles operadores clásicos como `+`, `-`, `*`, `/`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `and`, `or`, útiles para construir expresiones numéricas o lógicas que controlan tanto el comportamiento como la geometría generada. Se ha incluido `->` que permite sacar el vector dirección de un punto a otro, generalmente usado para representar el eje de rotación de un objeto.

Este conjunto de operaciones proporciona al lenguaje una gran expresividad y capacidad de abstracción, permitiendo al usuario crear geometrías complejas de forma estructurada, modificable y reutilizable.

Funciones geométricas auxiliares

El lenguaje incluye también un conjunto de funciones auxiliares orientadas a facilitar cálculos y operaciones geométricas frecuentes. Estas funciones permiten trabajar con puntos y vectores de forma más expresiva y compacta, y son especialmente útiles para definir relaciones espaciales o construir geometrías de forma programática. Las funciones principales son:

- `partway(p1, p2, l)`: recibe dos puntos en el espacio tridimensional $p_1 = (x_1, y_1, z_1)$ y $p_2 = (x_2, y_2, z_2)$, y un valor escalar $l \in [0, 1]$. Devuelve un nuevo punto p situado a una fracción l del trayecto de p_1 a p_2. Su fórmula matemática es:

$$p = p_1 + (p_2 - p_1) \cdot l = (x_1 + (x_2 - x_1)l,\ y_1 + (y_2 - y_1)l,\ z_1 + (z_2 - z_1)l)$$

- `distance(p1, p2)`: calcula la distancia euclídea entre dos puntos $p_1 = (x_1, y_1, z_1)$ y $p_2 = (x_2, y_2, z_2)$. La fórmula utilizada es:

$$d(p_1, p_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

- `angle(v1, v2)`: devuelve el ángulo θ entre dos vectores de dirección v_1 y v_2, en grados. Se utiliza el producto escalar para calcularlo. Dados $v_1 = (x_1, y_1, z_1)$ y $v_2 = (x_2, y_2, z_2)$:

$$\theta = \arccos\left(\frac{v_1 \cdot v_2}{\|v_1\| \cdot \|v_2\|}\right) \cdot \frac{180}{\pi}$$

  donde:

$$v_1 \cdot v_2 = x_1x_2 + y_1y_2 + z_1z_2, \quad \|v_i\| = \sqrt{x_i^2 + y_i^2 + z_i^2}$$

Estas funciones permiten que los scripts puedan contener cálculos espaciales dinámicos que afectan directamente a la construcción y posicionamiento de los objetos geométricos generados; se pueden ver con más detalle en *common_functions.py*.

4.2.3. Reglas Gramaticales

El lenguaje desarrollado se basa en una **gramática libre de contexto (CFG, Context-Free Grammar)**, un conjunto formal de reglas utilizado para describir la estructura sintáctica de lenguajes de programación y lenguajes formales en general. Se define de la siguiente forma:

$$G = (V, \Sigma, R, S)$$

donde:

- V es un conjunto finito de **símbolos no terminales**, que representan categorías sintácticas abstractas (por ejemplo, `<expr>`, `<statement>`).
- Σ es un conjunto finito de **símbolos terminales**, que son los elementos del lenguaje que no pueden descomponerse más (como palabras clave, operadores, identificadores, etc.).
- R es un conjunto finito de **reglas de producción**, de la forma:

  $$A \to \alpha$$

  donde $A \in V$ y $\alpha \in (V \cup \Sigma)^*$, es decir, una secuencia finita (posiblemente vacía) de terminales y/o no terminales.
- $S \in V$ es el **estado inicial**, a partir del cual comienza la derivación de cualquier cadena válida del lenguaje.

En las gramáticas libres de contexto, las reglas de producción se aplican *independientemente del contexto* en el que se encuentre el símbolo no terminal. Esto significa que para cada no terminal A, su expansión está definida únicamente por la regla $A \to \alpha$, sin considerar qué símbolos la rodean.

Este tipo de gramáticas son lo suficientemente expresivas para modelar la mayoría de las construcciones sintácticas de los lenguajes de programación modernos, y al mismo tiempo permiten algoritmos de análisis sintáctico eficientes, como los usados por autómatas LR.

A continuación, se muestra una representación parcial en BNF simplificada de algunos de los símbolos clave del lenguaje:

```
<program>     ::= <orders>
<orders>      ::= <orders> <order> | <order>
<order>       ::= <statements> ; | <function_def> | <import>
<statements>  ::= <statements> ; <statement> | <statement>
<statement>   ::= <assig> | <condStruct> | <loop> | <expr>
```

Se puede ver la versión completa de la gramática desarrollada en el apéndice B. Estos símbolos definen la estructura general del código fuente, comenzando por el símbolo inicial `<program>`, que consiste en una secuencia de órdenes (asignaciones, estructuras de control, etc.) separadas por punto y coma. Cada orden puede ser una sentencia, una definición de función o un import.

Reglas Básicas

En el lenguaje diseñado, cada sentencia debe finalizar obligatoriamente con un **punto y coma (`;`)**. Esta regla permite delimitar de forma clara las distintas instrucciones del script y es fundamental para el análisis sintáctico.

A continuación, se muestran ejemplos de las construcciones básicas del lenguaje:

- **Asignación**:

```
a = 5;
b = Vector(1, 2, 3);
```

- **Bloque condicional (if / else)**:

```
if (x > 3) {
    y = 5;
} else {
    y = x;
};
```

- **Bucle while**:

```
while (i < 10) {
    i = i + 1;
};
```

- **Definición de funciones**:

```
define suma(a, b) {
    return a + b;
};
```

- **Importación de módulos**:

```
import dollhouse as dh;
```

- **Llamada a función**:

```
resultado = suma(3, 4);
```

Reglas Avanzadas

Existen ciertas reglas sintácticas más restrictivas que ayudan a evitar ambigüedades y errores durante la interpretación. Estas incluyen:

- Los **identificadores** no pueden coincidir con:
  - Palabras clave del lenguaje (`if`, `else`, `return`, etc.).
  - Los valores booleanos `true` y `false`.
  - Nombres de funciones o clases ya definidas en el entorno o definidas en el script.
- No se permite el **anidamiento de operadores de comparación** en una misma expresión como:

```
1 < y < 6  // Inválido
```

En su lugar, deben expresarse con conectores lógicos:

```
1 < y && y < 6;
```

- Las llamadas a funciones y estructuras de control deben respetar una sintaxis estricta en cuanto a delimitadores (`( )` para argumentos y `{ }` para bloques).
- El uso del punto y coma es obligatorio después de cada orden, incluso después de bloques como `if`, `while` o `define`.
- Se ha incorporado una regla especial para el uso del operador `|` en contextos específicos, a petición del usuario final. Por norma general, el operador `|` permite acceder al valor de una componente de un vector (coordenada) asociada a un objeto 3D. Por ejemplo:
  - `obj|ox` devuelve la componente `x` del origen (`o`) del objeto `obj`.
  - `obj|oy` devuelve la componente `y`.
  - `obj|oz` devuelve la componente `z`.

  Sin embargo, como mejora de usabilidad, se permite un atajo sintáctico específico en el contexto del argumento `edge` en llamadas a funciones de rotación. En este caso, el operador `|` puede sustituir de forma equivalente a expresiones más largas, con la siguiente correspondencia:
  - `obj|ox` se interpreta como `obj.o ->obj.ox`
  - `obj|oy` se interpreta como `obj.o ->obj.oy`
  - `obj|oz` se interpreta como `obj.o ->obj.oz`

  Esto proporciona una forma concisa de especificar ejes de rotación relativos al objeto, sin necesidad de construir explícitamente el vector desde el origen del objeto hacia una dirección concreta.

Precedencia

El manejo adecuado de la precedencia de operadores es fundamental para que el parser resuelva de manera correcta expresiones complejas y ambiguas. En este proyecto, se ha definido una tabla de precedencia específica que permite establecer el orden en que se deben evaluar los distintos operadores del lenguaje, evitando así la necesidad de añadir paréntesis en todos los casos.

Como bien se describe en Aho et al. [2006], "An ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence."

Como se puede observar en el ejemplo de la Figura 4, la sentencia $2 + 6 * 9$ puede interpretarse correctamente de dos formas: $((2 + 6) * 9)$ o $(2 + (6 * 9))$. Es por esta razón que existe la necesidad de tener definido **un orden de prioridad** a la hora de realizar el *parsing*.

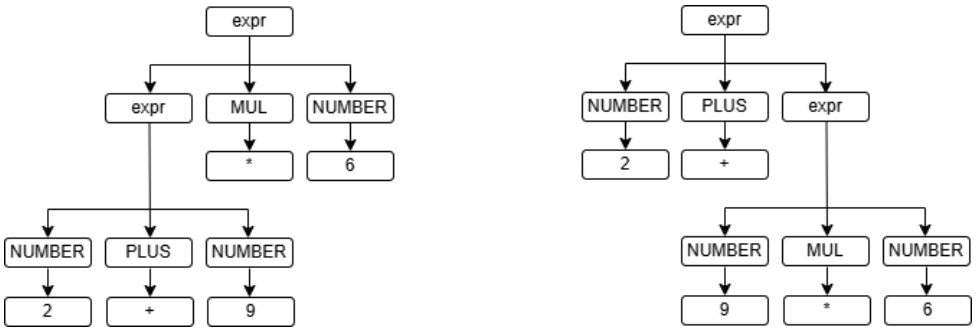

Figura 4: Ejemplo de dos posibles interpretaciones de $2 + 6 * 9$.

La precedencia ha sido definida de forma explícita en la clase `SimpleParser`, utilizando la sintaxis de SLY. En la tabla 3 se muestra la precedencia utilizada:

Nivel	Asociatividad	Operadores
1	Izquierda	`;`
2	Izquierda	`||`, `&&`
3	No asociativo	`==`, `!=`, `<`, `>`, `<=`, `>=`
4	Izquierda	`+`, `-`
5	Izquierda	`*`, `/`
6	No asociativo	`->`

Cuadro 3: Tabla de Precedencia de Operadores

4.2.4. Análisis Semántico

Una vez superadas las fases de análisis léxico y sintáctico, el análisis semántico se encarga de validar que el código escrito por el usuario no solo tenga una estructura gramatical correcta, sino que también tenga sentido dentro del contexto del lenguaje. A diferencia del análisis sintáctico, que se limita a la forma, el análisis semántico asegura que se respeten las reglas contextuales del lenguaje, como tipos compatibles, declaración previa de variables o restricciones estructurales.

En este proyecto, el análisis semántico no se realiza sobre un árbol de sintaxis abstracta (AST) como ocurre tradicionalmente. En su lugar, se implementa directamente sobre el conjunto de instrucciones obtenidas tras el análisis sintáctico.

El sistema de ámbitos permite registrar funciones, variables, objetos 3D, y estructuras compuestas como listas o grupos. Cada vez que se introduce un nuevo bloque (como una función), se crea un subámbito que hereda del anterior. Esto facilita la validación semántica del contexto y evita conflictos entre nombres o accesos indebidos.

Durante esta fase, se comprueban aspectos como:

- Que los identificadores utilizados hayan sido previamente definidos en el ámbito actual o en alguno de sus ancestros.
- Que los nombres de variables no colisionen con los nombres de funciones, clases predefinidas (como `Box`, `Cylinder`, `Group`) o palabras reservadas del lenguaje.
- Que las operaciones aritméticas y lógicas se apliquen sobre tipos compatibles.
- Que las llamadas a funciones se realicen con el número y tipo correcto de argumentos.

- Que las estructuras de control (`if`, `while`) reciban expresiones booleanas evaluables.
- Que los parámetros definidos en funciones sean válidos.
- Que no se produzcan usos incorrectos de operadores como `|` o `->` fuera de sus contextos válidos.
- Que se cumplan las reglas avanzadas del lenguaje, como evitar anidamientos semánticamente inválidos (por ejemplo: `1 < x < 6`).

En resumen, el análisis semántico constituye una fase importante en la validación del código, garantizando no solo su corrección formal sino también su coherencia lógica y contextual dentro del lenguaje. La implementación del sistema de ámbitos y las comprobaciones detalladas mencionadas aseguran que el programa respete las reglas semánticas definidas, previniendo errores comunes y facilitando un procesamiento seguro y consistente del código. De este modo, se sientan las bases para la posterior generación y ejecución correcta de las instrucciones, consolidando la robustez y fiabilidad del compilador o intérprete desarrollado en este proyecto.

5 Diseño de la Aplicación

5.1. Arquitectura

La arquitectura de la aplicación se ha planteado con un enfoque modular, lo que permite una separación clara de responsabilidades entre los distintos componentes del sistema. Esta estructura favorece la mantenibilidad, escalabilidad y futuras extensiones del lenguaje o de la interfaz.

La arquitectura de la aplicación se ha planteado con un enfoque modular, lo que permite una separación clara de responsabilidades entre los distintos componentes del sistema. Esta estructura favorece la mantenibilidad, escalabilidad y posibles futuras extensiones del lenguaje o de la interfaz gráfica.

Además, se sigue una aproximación basada en el patrón Modelo-Vista-Controlador (MVC), que refuerza esta separación:

- El **Modelo** incluye toda la lógica del lenguaje, como el lexer, parser, intérprete y estructuras propias del dominio (objetos 3D, transformaciones, funciones auxiliares, etc.), funcionando de forma completamente independiente de la interfaz.

- La **Vista** implementada mediante la librería Tkinter, proporciona la interfaz gráfica que incluye el editor de texto y los botones de interacción. La zona de visualización 3D, por su parte, está gestionada por la librería VPython, que permite renderizar directamente los objetos generados por el lenguaje, facilitando la observación visual del resultado de los scripts escritos por el usuario.

- El **Controlador** coordina la interacción entre la vista y el modelo, gestionando acciones como cargar archivos, ejecutar scripts o actualizar la representación gráfica según la salida del lenguaje.

Cabe destacar que esta aplicación ha sido concebida como una prueba de concepto, donde la misión principal es probar y validar el lenguaje de dominio específico desarrollado, así como su integración con un sistema de visualización. Por tanto, se ha priorizado un diseño sencillo, funcional y orientado al testeo del lenguaje, más que a ofrecer una aplicación final para usuario final.

5.1.1. Estructura General

La aplicación se divide en los siguientes módulos principales:

- **Lexer y Parser (SLY):** Se encargan del análisis léxico y sintáctico del lenguaje definido. Implementados con la biblioteca SLY, traducen el código fuente en una secuencia de instrucciones válidas, construyendo un autómata LR(1) que valida la gramática libre de contexto definida por el usuario.

- **Módulo Semántico y Evalución:** Ejecuta las instrucciones interpretadas tras el análisis sintáctico. Se encarga de instanciar los objetos 3D, aplicar transformaciones y evaluar operaciones y expresiones definidas en el lenguaje. Gestiona los ámbitos de ejecución, la validación de tipos, la comprobación de identificadores, la interpretación de funciones y el control del flujo lógico del programa.
- **Módulo de Visualización 3D (VPython):** Encargado de representar gráficamente los objetos generados en el entorno tridimensional. VPython facilita la creación de figuras geométricas y permite la interacción dinámica con la escena.
- **Interfaz de Usuario (UI):** Se ha desarrollado una interfaz gráfica sencilla que permite al usuario escribir código DSL, ejecutar los scripts y visualizar los modelos 3D generados en tiempo real. Su diseño se ha enfocado en ofrecer una herramienta funcional y accesible que facilite la interacción con el lenguaje, sirviendo como entorno principal para realizar pruebas, verificar comportamientos y validar resultados durante el desarrollo.

5.1.2. Flujo General de Uso

Como se observa en el diagrama de la Figura 5, se representa el funcionamiento general de la aplicación. Esta arquitectura ha sido diseñada para permitir, en el futuro, la sustitución o

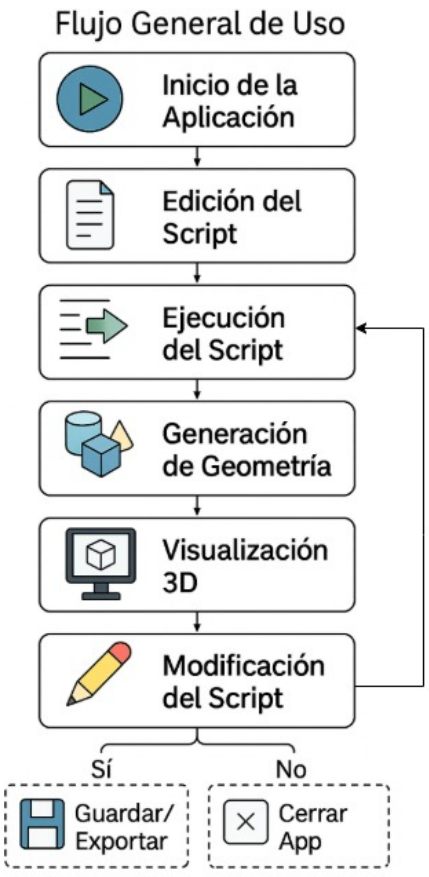

Figura 5: Flujo General de Uso

mejora independiente de cualquier módulo (por ejemplo, reemplazar la interfaz actual por una más avanzada o integrar más opciones en el módulo de visualización 3D).

5.2. Diseño de la Interfaz

Dado que el enfoque principal del proyecto es el desarrollo del lenguaje y su intérprete, la interfaz gráfica se ha planteado como una solución funcional y sencilla que permite al usuario probar scripts y visualizar su resultado.

5.2.1. Componentes Principales

La interfaz está compuesta por los siguientes elementos:

- **Área de edición de código:** Un campo de texto multilineal donde el usuario puede escribir o pegar su script DSL para ser interpretado. Este área es el componente central de interacción textual del usuario con el lenguaje.
- **Barra de acciones:** Conjunto de botones funcionales situados sobre el área de edición, que permiten al usuario gestionar sus scripts. Estos botones incluyen:
  - **New File:** Abre una nueva pestaña en el área de edición para comenzar un nuevo script.
  - **Open File:** Permite cargar en una nueva pestaña un archivo DSL existente desde el sistema de archivos.
  - **Save:** Guarda el script actual sobrescribiendo el archivo asociado.
  - **Save As:** Guarda el script actual en un nuevo archivo con el nombre elegido por el usuario.
  - **Run Program:** Ejecuta el código actualmente cargado en el área de edición. Este botón lanza el proceso de análisis léxico, sintáctico, semántico y finalmente la evaluación del código con salida visual en 3D.

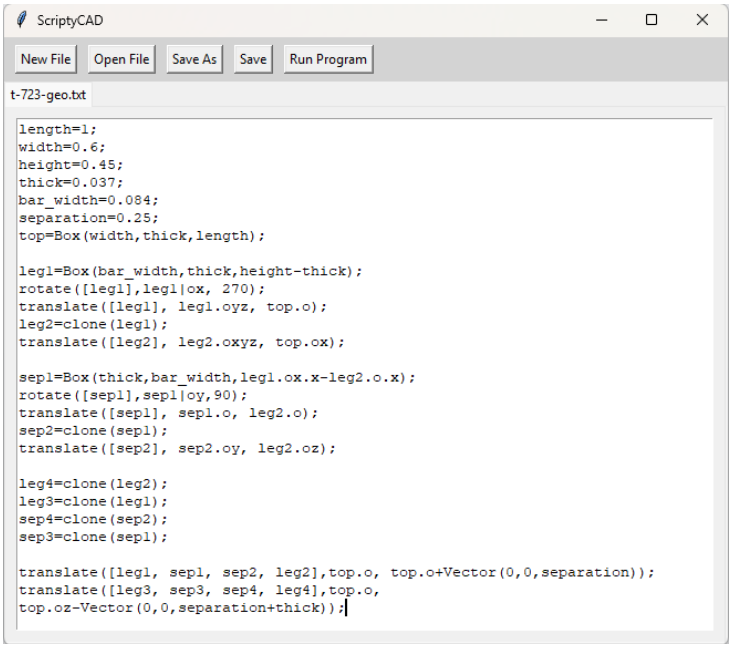

Figura 6: Interfaz de Usuario (UI)

- **Ventana de visualización 3D:** Proporcionada por VPython, esta ventana muestra en tiempo real la escena tridimensional generada a partir del script ejecutado. Permite rotar, hacer zoom y desplazarse para inspeccionar los objetos creados.

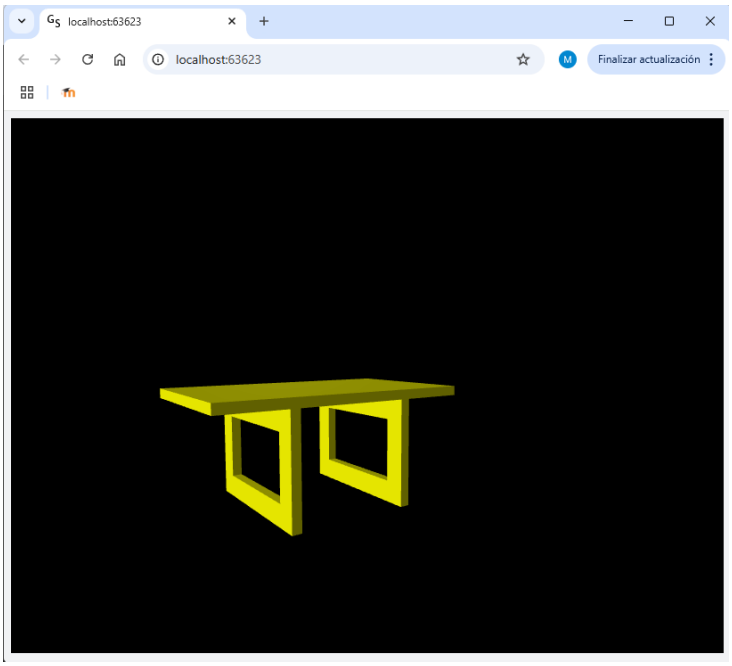

Figura 7: Ventana de visualización 3D de VPython

- **Consola:** Espacio dedicado a mostrar retroalimentación al usuario. Incluye errores léxicos, sintácticos o semánticos detectados durante la ejecución, así como mensajes de éxito o advertencias.

Figura 8: Consola

5.2.2. Limitaciones y Futuras Mejoras

La interfaz actual cumple su propósito como entorno de pruebas, pero presenta limitaciones en cuanto a usabilidad, interactividad y diseño gráfico. Se prevé en futuros proyectos:

- Integración con un entorno de desarrollo más completo (como una aplicación web o un IDE específico).
- Autocompletado y resaltado de sintaxis.
- Gestión avanzada de archivos y scripts.
- Interacción más completa con objetos 3D desde la visualización.
- Ejecución parcial del código, principalmente pensada para la depuración, marcando como límite la posición actual del cursor.

En resumen, la interfaz desarrollada actúa como una primera versión funcional que acompaña al diseño del lenguaje y su sistema de ejecución, cumpliendo con los objetivos iniciales del proyecto y sentando las bases para futuras mejoras.

6 Pruebas

6.1. Pruebas del Lenguaje de Dominio

A continuación, se presentan de forma detallada las pruebas unitarias realizadas tanto para el analizador léxico (Lexer) como para el analizador sintáctico (Parser), así como las pruebas de integración que verifican el correcto funcionamiento conjunto de ambos componentes del lenguaje.

6.1.1. Pruebas unitarias del Lexer

Las pruebas unitarias del Lexer se han realizado con el objetivo de comprobar que cada token definido en la gramática léxica del lenguaje se reconozca correctamente, conforme a sus patrones especificados mediante expresiones regulares. Estas pruebas son fundamentales para garantizar la fiabilidad del análisis léxico, que es la base del procesamiento del lenguaje.

A continuación, se detallan las categorías de pruebas realizadas:

- **Reconocimiento de identificadores válidos:** Se comprobó que secuencias de caracteres que representan nombres de variables, funciones u objetos se identifican correctamente como identificadores (`ID`). Para ello, se probaron combinaciones válidas que comienzan por una letra o guion bajo, seguidas de letras, números o guiones bajos. También se verificó que estos identificadores no coincidan con palabras reservadas del lenguaje.

- **Identificación de palabras clave:** Se verificó que palabras clave del lenguaje como `IF`, `ELSE`, `WHILE`, `RETURN`, `IMPORT`, `AS`, `DEFINE`, entre otras, fueran reconocidas como tokens independientes y no confundidas con identificadores comunes. Esta distinción es crucial para el análisis sintáctico, ya que estas palabras tienen un significado estructural específico dentro del lenguaje.

- **Detección de operadores aritméticos y lógicos:** Se realizaron pruebas sobre operadores aritméticos como `+`, `-`, `*`, `/`, así como operadores de comparación (`<`, `>`, `<=`, `>=`, `==`, `!=`) y operadores lógicos como `AND`, `OR`. Cada uno fue probado individualmente y en expresiones complejas para comprobar que se identifican correctamente y, en los casos de los comparativos, que mantienen su precedencia.

- **Tokens de puntuación:** Se validó que los símbolos de puntuación esenciales del lenguaje como paréntesis (`(`, `)`), llaves (`{`, `}`), corchetes (`[`, `]`), punto y coma (`;`) y coma (`,`) sean detectados como tokens válidos y diferenciados del resto de tipos de tokens.

- **Literales numéricos, booleanos y cadenas de texto:** Se incluyeron pruebas para comprobar el reconocimiento correcto de números enteros y decimales, valores booleanos (`true`, `false`) y literales de cadena definidos entre comillas dobles. Se validaron tanto casos correctos como casos límite (por ejemplo, cadenas vacías, números decimales, booleanos mal escritos).

- **Comprobación del tratamiento de comentarios:** El lenguaje permite comentarios de una sola línea iniciados por el símbolo #. Se verificó que dichos comentarios sean correctamente ignorados por el lexer, es decir, que no generen ningún token y no interfieran con la generación de tokens válidos en el resto del código fuente.

Estas pruebas permiten asegurar que el lexer cumple con la especificación formal del lenguaje y se comporta de forma predecible incluso ante entradas inesperadas o erróneas. Además, sirven como base para el correcto funcionamiento del parser, dado que una mala tokenización puede causar fallos en las etapas posteriores del intérprete.

A continuación se muestra un ejemplo de entrada de prueba y su correspondiente salida esperada:

```
# Código de entrada
source_code = '''
define square {
    a = 5;
    b = a + 3;
};
'''

# Tokens esperados
[
    Token(type='DEFINE', value='define'),
    Token(type='ID', value='square'),
    Token(type='{', value='{'),
    Token(type='ID', value='a'),
    Token(type='ASSIGN', value='='),
    Token(type='NUMBER', value=5),
    Token(type=';', value=';'),
    Token(type='ID', value='b'),
    Token(type='ASSIGN', value='='),
    Token(type='ID', value='a'),
    Token(type='PLUS', value='+'),
    Token(type='NUMBER', value=3),
    Token(type=';', value=';'),
    Token(type='}', value='}'),
    Token(type=';', value=';')
]
```

Código 6.1: Ejemplo de prueba unitaria del Lexer define.txt.

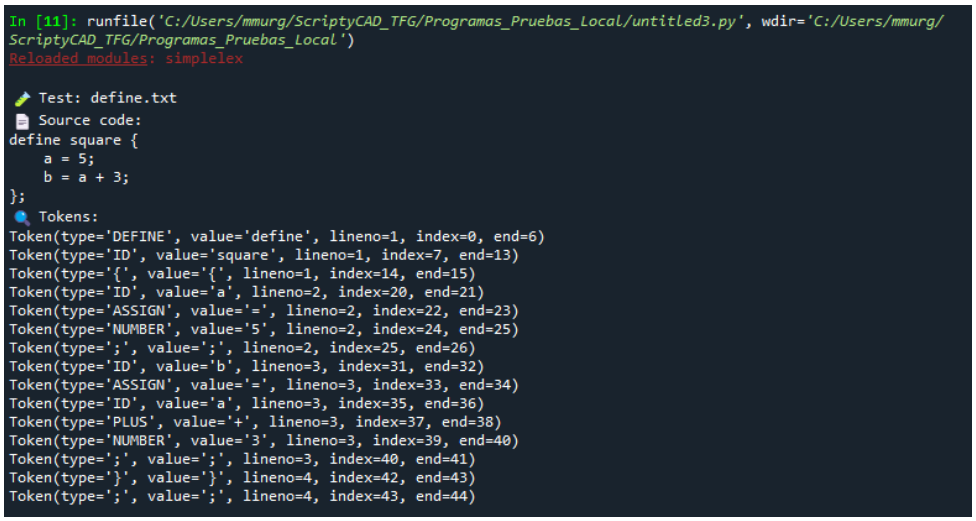

Figura 9: Salida de la prueba del Lexer define.txt

Este ejemplo permite comprobar que el lexer reconoce correctamente los distintos componentes léxicos, respetando además la prioridad de palabras clave sobre identificadores (por ejemplo, `define` es reconocido como palabra clave y no como identificador).

El repositorio del proyecto incluye un conjunto más amplio de casos de prueba que cubren diferentes estructuras del lenguaje, incluyendo expresiones complejas, estructuras condicionales y bucles en Test_Lexer.

6.1.2. Pruebas unitarias del Parser

Las pruebas unitarias del Parser se han realizado con el objetivo de validar que las reglas sintácticas definidas en la gramática del lenguaje se interpretan y ejecutan correctamente. Para ello, se han preparado diversos tests que cubren tanto las construcciones básicas como combinaciones más complejas de estructuras del lenguaje. Estas pruebas han sido esenciales para comprobar que el analizador sintáctico puede reconocer y procesar adecuadamente la estructura del programa.

Las categorías de pruebas llevadas a cabo incluyen:

- **Asignaciones:** Se probaron expresiones simples y compuestas del tipo `a = 5;`, `v = Vector(1, 2, 3);`, y `obj = Box(1, 1, 1);`, para verificar que las asignaciones se reconocen correctamente como instrucciones válidas y se almacenan en la estructura semántica adecuada.
- **Condicionales `if-else`:** Se incluyeron pruebas como:

```
if (x > 0) {
    y = 5;
} else {
    y = -5;
}
```

  Estas pruebas permitieron validar que las estructuras condicionales son aceptadas por el parser y que se realiza una correcta agrupación de bloques entre llaves.

- **Bucles `while`:** Se testearon expresiones del tipo:

```
while (i < 10) {
    i = i + 1;
}
```

  para verificar que las instrucciones iterativas son parseadas como bloques repetitivos válidos, con evaluación continua de la condición mientras esta sea verdadera.

- **Definición de funciones:** Se incluyeron pruebas para verificar la definición de funciones con y sin parámetros, por ejemplo:

```
define suma(a, b) {
    return a + b;
}
```

  Se comprobó que estas definiciones se almacenan como entidades propias en el `scope` y que sus parámetros y cuerpo se registran correctamente.

- **Llamadas a funciones:** Se validó que llamadas como `suma(3, 5);` o llamadas encadenadas en asignaciones o condiciones fueran aceptadas por el parser.

- **Importación de módulos:** Se testearon casos como `import dollhouse;` y `import dollhouse as dh;` para verificar que el parser reconoce correctamente esta estructura y que se gestiona adecuadamente el alias cuando es necesario.

- **Detección de errores sintácticos:** También se diseñaron pruebas negativas para comprobar que el parser es capaz de detectar errores como sentencias sin punto y coma, condicionales mal anidados, paréntesis no balanceados, uso incorrecto de palabras clave o llamadas a funciones inexistentes.

Todas estas pruebas permiten garantizar que el Parser, construido con SLY, reconoce las estructuras definidas en la gramática del lenguaje y es capaz de construir una representación interna coherente para su posterior análisis semántico y ejecución, ya que, al no construir un árbol de derivación (AST), la fase de análisis semántico también se produce en esta etapa. Esto ha hecho que la fase de verificación fuese algo más compleja, ya que gran parte de esto ha necesitado verificación manual usando el debugger. Esto ha aumentado la complejidad de la fase de verificación, ya que gran parte del comportamiento del parser no podía ser comprobado únicamente mediante pruebas automatizadas, requiriendo en su lugar una verificación manual a través del uso del depurador (debugger) para rastrear y validar el flujo de ejecución y el almacenamiento correcto de las instrucciones en el `scope`.

```
In [8]: runfile('C:/Users/mmurg/ScriptyCAD_TFG/Programas_Pruebas_Local/test_parser.py', wdir='C:/Users/mmurg/
ScriptyCAD_TFG/Programas_Pruebas_Local')
Reloaded modules: simplelex, simpleparser, basic_classes, common_functions, box_class, sphere_class,
group_class, cylinder_class

Parsing test: error.txt
Source code:
a = 1;
a = a + 2;
b = a - 1       #Falta ; debería dar error en la variable b de la siguiente línea
b = 2 + a - b;

Parsing result:
Error
Parsing error:
 variable 'a' is not defined.
sly: Syntax error at line 4, token=ID
```

Figura 10: Salida de la prueba del Parser error.txt

```
In [7]: runfile('C:/Users/mmurg/ScriptyCAD_TFG/Programas_Pruebas_Local/test_parser.py', wdir='C:/Users/mmurg/
ScriptyCAD_TFG/Programas_Pruebas_Local')
Reloaded modules: simplelex, simpleparser, basic_classes, common_functions, box_class, sphere_class,
group_class, cylinder_class

Parsing test: error_fixed.txt
Source code:
a = 1;
a = a + 2;
b = a - 1;       #Ahora debería funcionar correctamente
b = 2 + a - b;
Parsing result:
Correct
Scope: {'prevDict': None, 'translate': <function SimpleParser.__init__.<locals>.<lambda> at
0x0000020A7ECDD708>, 'escalate': <function SimpleParser.__init__.<locals>.<lambda> at 0x0000020A7ECDD678>,
'rotate': <function SimpleParser.__init__.<locals>.<lambda> at 0x0000020A7ECDE798>, 'Box': <function
SimpleParser.__init__.<locals>.<lambda> at 0x0000020A7C5B4B88>, 'Sphere': <function
SimpleParser.__init__.<locals>.<lambda> at 0x0000020A7C928948>, 'partway': <function
SimpleParser.__init__.<locals>.<lambda> at 0x0000020A7EB85558>, 'distance': <function
SimpleParser.__init__.<locals>.<lambda> at 0x0000020A7EB85318>, 'angle': <function
SimpleParser.__init__.<locals>.<lambda> at 0x0000020A7EB85948>, 'Vector': <function
SimpleParser.__init__.<locals>.<lambda> at 0x0000020A7EB854C8>, 'Group': <function
SimpleParser.__init__.<locals>.<lambda> at 0x0000020A7EB85708>, 'sleep': <function
SimpleParser.__init__.<locals>.<lambda> at 0x0000020A7EB850D8>, 'clone': <function
SimpleParser.__init__.<locals>.<lambda> at 0x0000020A7EB851F8>, 'Cylinder': <function
SimpleParser.__init__.<locals>.<lambda> at 0x0000020A7EB85A68>, 'a': 3.0, 'b': 3.0}
```

Figura 11: Salida de la prueba del Parser error_fixed.txt

Ejemplos completos de estas pruebas están disponibles en Test_Parser.

6.1.3. Pruebas de integración del lenguaje de dominio

Las pruebas de integración del lenguaje de dominio se han centrado en verificar que el flujo completo desde el análisis léxico, pasando por el análisis sintáctico y culminando en la ejecución semántica del programa, se realiza correctamente y sin errores.

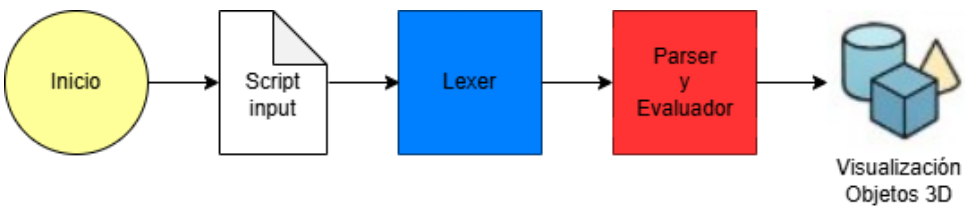

Figura 12: Diagrama de flujo para las pruebas de integración del lenguaje de dominio

Estas pruebas permiten comprobar que los diferentes componentes que forman el núcleo del lenguaje (lexer, parser y evaluador) interactúan entre sí de manera consistente y funcional.

Para ello, se han preparado múltiples ejemplos de código fuente en el nuevo lenguaje, los cuales han sido procesados en su totalidad por el sistema, desde la fase de tokenización hasta la generación del resultado final, ya sea una salida visual o textual.

Los casos de prueba incluyen:

- Definición y uso de variables de distintos tipos: `Number`, `String`, `Boolean`, objetos 3D, vectores y listas.
- Estructuras de control: condicionales `if/else`, bucles `while`, y definiciones de funciones con parámetros y valores de retorno.
- Operaciones aritméticas y lógicas entre variables y literales.
- Llamadas a funciones definidas por el usuario o importadas desde otros archivos.
- Generación y visualización de objetos 3D a partir de sentencias interpretadas.
- Transformaciones afines sobre grupos de objetos.

Cada uno de estos casos se ha verificado observando que la salida del programa coincide con la esperada, ya sea a través del renderizado correcto en la escena tridimensional, del comportamiento lógico del programa, o del resultado textual.

Por tanto, las pruebas de integración han sido esenciales para detectar incompatibilidades entre los módulos, validar el correcto manejo del ámbito de ejecución y asegurar que el lenguaje proporciona una experiencia coherente y funcional para el usuario final.

6.2. Pruebas de la Aplicación

Tras validar el correcto funcionamiento del lenguaje de dominio, se procedió a realizar pruebas sobre la propia aplicación desarrollada. Estas pruebas tienen como objetivo verificar que los distintos componentes de la interfaz gráfica y de la lógica de control funcionan de manera aislada y en conjunto.

6.2.1. Pruebas unitarias

Las pruebas unitarias de la aplicación se centraron en los distintos módulos que componen la interfaz y la lógica de interacción del usuario. Estas pruebas permitieron asegurar que cada parte funcional del sistema se comporta como se espera cuando se la evalúa de forma aislada.

Los componentes evaluados incluyen:

- **Editor de código**: Se comprobó que el área de texto permite la escritura fluida de código, con funcionalidades como tabulación, salto de línea, copia, corte y pegado.
- **Botones funcionales**: Se verificó que los botones de la barra de herramientas —`New File`, `Open File`, `Save`, `Save As`, y `Run Program`— responden correctamente. Por ejemplo, que abrir un archivo carga su contenido en el editor, o que guardar crea correctamente un fichero en disco.
- **Sistema de mensajes**: Se comprobó el correcto funcionamiento de los mensajes emergentes (pop-ups) para informar al usuario sobre errores de ejecución, guardado exitoso, advertencias, etc.

- **Consola de salida**: Se validó que los mensajes generados por el intérprete se imprimen correctamente en la zona de salida y que se actualiza dinámicamente en función del resultado de la ejecución.
- **Renderizado 3D**: Aunque forma parte también de pruebas de integración, se realizaron pruebas unitarias para comprobar que la inicialización del entorno 3D con VPython se produce correctamente al invocar los métodos básicos de creación de objetos geométricos.

Las pruebas se realizaron ejecutando la aplicación en distintos estados iniciales para comprobar que no se produjeran errores inesperados, implicando que muchas de estas pruebas se llevaron a cabo de forma manual debido a la naturaleza gráfica e interactiva de la aplicación, que hacía complejo su testeo mediante frameworks convencionales.

En conjunto, estas pruebas contribuyeron a detectar fallos tempranos en la interfaz, mejorar la experiencia del usuario y garantizar la robustez del entorno de desarrollo proporcionado.

6.2.2. Pruebas de integración

Las pruebas de integración se centraron en verificar la correcta interacción entre los distintos módulos que conforman la aplicación: la interfaz gráfica, el motor de interpretación del lenguaje, el sistema de visualización 3D y el manejo de archivos. El objetivo principal fue asegurar que los componentes se comunican entre sí de manera coherente y que el flujo general de la aplicación no se ve interrumpido por errores de integración.

Entre las integraciones más relevantes que fueron objeto de prueba, se destacan:

- **Editor de texto y ejecución**: Se verificó que el código escrito en el editor puede ser interpretado correctamente al pulsar el botón de ejecución (`Run Program`). Se comprobó que el texto se transmite al intérprete, que se analiza sintáctica y semánticamente, y que genera la salida esperada o informa de errores adecuadamente.
- **Interpretación y renderizado 3D**: Se validó que, tras una ejecución exitosa, las instrucciones válidas relacionadas con la creación de objetos 3D generan resultados visuales en la ventana de VPython. Esto incluye pruebas con primitivas geométricas básicas como `box`, `sphere`, `cylinder`, y estructuras más complejas como `Group` o transformaciones aplicadas.
- **Gestión de errores**: Se evaluó la correcta gestión de errores de interpretación y visualización. Por ejemplo, si el usuario escribe una instrucción mal formada, se espera que la consola de salida indique el tipo de error sin colapsar la aplicación ni interrumpir otras funcionalidades.
- **Carga y guardado de archivos**: Se integraron pruebas para asegurar que los archivos cargados desde el sistema de archivos se reflejan correctamente en el editor y que el contenido editado puede guardarse y recuperarse fielmente sin pérdida de información.
- **Comunicación entre interfaz e intérprete**: Se evaluó la correcta comunicación entre el frontend (interfaz de usuario) y el backend (intérprete y lógica de control). Esto incluyó verificar que los eventos de los botones ejecutan las funciones correspondientes, que los errores son propagados correctamente al usuario, y que los datos entre componentes fluyen de forma sincronizada.

Estas pruebas se realizaron accediendo a varios de los scripts usados para las pruebas del parser y creando alguno nuevo desde la aplicación de forma manual para evaluar el comportamiento en escenarios realistas de uso. Algunos casos incluyeron la ejecución de programas grandes con

múltiples objetos y transformaciones, importación de archivos, modificaciones en tiempo de ejecución y validación de funcionalidades encadenadas.

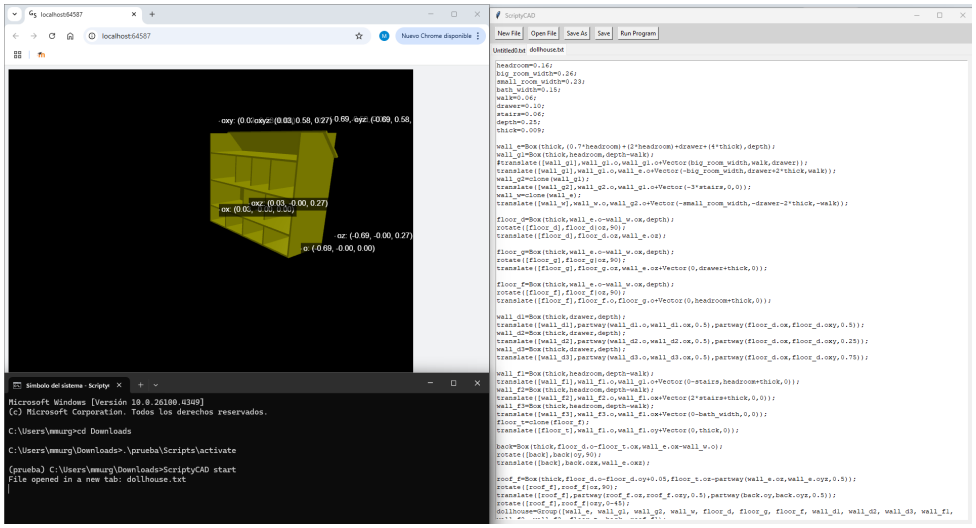

Figura 13: Prueba completa de la aplicación usando el fichero dollhouse.txt

El resultado de estas pruebas permitió confirmar que la arquitectura modular de la aplicación facilita la integración progresiva de funcionalidades, manteniendo la estabilidad general del sistema.

6.3. Pruebas de sistema de la Aplicación

Las pruebas de sistema tienen como objetivo verificar el comportamiento de la aplicación como un todo, comprobando que todos los módulos (lexer, parser, motor de ejecución, motor gráfico e interfaz) interactúan correctamente entre sí y ofrecen una experiencia de usuario coherente, eficiente y libre de errores críticos.

Estas pruebas buscan garantizar que el sistema no solo cumple con los requisitos funcionales establecidos, sino que también responde adecuadamente en distintos entornos y escenarios. Para ello, se han realizado pruebas enfocadas en aspectos como la portabilidad, usabilidad y escalabilidad del sistema.

6.3.1. Pruebas de portabilidad

Dado que la aplicación ha sido desarrollada en Python, se ha probado su ejecución en distintos sistemas operativos (Windows 10/11 y Ubuntu 22.04) utilizando versiones compatibles del intérprete (Python 3.9 o superior). Estas pruebas buscan garantizar que el sistema puede ejecutarse sin modificaciones ni errores críticos en diferentes entornos de usuario.

Inicialmente, se había contemplado distribuir la aplicación mediante un archivo comprimido `.zip` acompañado de un script `.sh` de instalación manual. Sin embargo, esta solución implicaba una configuración más compleja, ya que, a pesar de que el script instalaba también las dependencias, constaba de bastantes pasos y era más propenso a errores.

Finalmente, se optó por emplear la herramienta `setuptools`, una utilidad estándar en el ecosistema Python para empaquetar y distribuir software. Esta elección permitió:

- Automatizar la instalación de dependencias definidas en el fichero `setup.py`.
- Facilitar la creación de paquetes reutilizables y actualizables desde repositorios remotos.
- Garantizar compatibilidad multiplataforma sin intervención manual del usuario.

La instalación completa puede realizarse de forma sencilla desde el terminal, ejecutando:

```
pip install git+https://mmg122@bitbucket.org/mmg122/tfg-manuel.git
```

Este comando instala la aplicación directamente desde el repositorio remoto, siempre que el sistema tenga instalado Python y las librerías `setuptools` y `wheel`. Estas últimas pueden instalarse fácilmente ejecutando:

```
pip install setuptools wheel
```

Una vez instalada, la aplicación puede ejecutarse en cualquier entorno compatible sin necesidad de configuración adicional usando el comando

```
ScriptyCAD start
```

Las pruebas demostraron que el proceso es robusto y accesible para usuarios con conocimientos básicos de Python.

Durante las pruebas, se verificó:

- La correcta instalación y funcionamiento de todas las dependencias (SLY, VPython, Tkinter, etc.).
- La ejecución sin errores del intérprete DSL y su integración con el sistema de visualización 3D.
- La apertura y guardado de archivos en sistemas de ficheros con diferentes convenciones (ej. rutas con espacios o caracteres especiales).

En conclusión, se considera que la aplicación es portable y accesible, cumpliendo con los criterios de instalación sencilla y compatibilidad multiplataforma.

6.3.2. Pruebas de usabilidad

Aunque la interfaz es sencilla y sirve solo como prototipo inicial, se ha verificado su comprensión por parte de usuarios cercanos, incluidos familiares y amigos:

- Se realizaron pruebas con 3 usuarios que no conocían previamente el lenguaje.
- Se les facilitó una guía rápida y ejemplos de código.
- Los usuarios lograron escribir scripts simples, modificar parámetros y visualizar resultados correctamente.

Se identificaron mejoras futuras como la incorporación de autocompletado, resaltado de sintaxis y mensajes de error más descriptivos.

6.3.3. Pruebas de escalabilidad

Se evaluó el comportamiento del sistema al ejecutar scripts grandes, con más de 200 instrucciones, múltiples objetos y transformaciones, el script usado fue pruebas_escalabilidad.txt. No se identificaron errores en el procesamiento, aunque la representación 3D mostró ligeros retardos, esperables por la carga visual.

6.4. Pruebas de aceptación de la Aplicación

El lenguaje y la aplicación han sido validados por el usuario final (Esteban Stafford, Co-Director del TFG), verificando que:

- El DSL cumple con los requisitos de expresividad deseados.
- Se pueden definir y visualizar modelos 3D complejos a través del lenguaje.
- La interfaz proporciona un entorno mínimo funcional para cargar, editar y ejecutar scripts.

Aunque se han identificado áreas de mejora, se considera que el objetivo principal del proyecto —crear un lenguaje de dominio específico para modelado geométrico con interpretación visual— ha sido alcanzado satisfactoriamente.

7 Conclusiones

7.1. Conclusión

A lo largo de este trabajo se ha diseñado e implementado un Lenguaje Específico de Dominio (DSL) orientado a la creación y manipulación de objetos tridimensionales mediante una sintaxis simple e intuitiva. Este lenguaje permite a los usuarios definir construcciones geométricas, transformarlas y visualizar el resultado en tiempo real, todo ello integrado dentro de una aplicación interactiva escrita en Python.

Durante el desarrollo, se han abordado distintas fases fundamentales en la construcción de un lenguaje: desde el análisis léxico y sintáctico, utilizando la librería SLY, hasta la interpretación semántica de las instrucciones. Asimismo, se ha proporcionado una interfaz gráfica intuitiva que facilita la escritura y ejecución de código en este lenguaje, permitiendo su uso incluso a usuarios sin experiencia previa en lenguajes de programación general.

El lenguaje ha demostrado ser funcional, expresivo y flexible para tareas de modelado geométrico basado en operaciones de geometría constructiva, transformaciones afines y agrupaciones de objetos. Además, se ha verificado el correcto funcionamiento del sistema mediante pruebas unitarias e integradas, abarcando tanto el análisis del código como la visualización de los modelos generados.

Este proyecto ha supuesto un desafío, especialmente en lo referente al desarrollo y depuración del analizador sintáctico (Parser), como se detalla en la sección 4.2. Al tratarse de la primera ocasión en la que abordaba la implementación de un parser optimizado sin recurrir a la generación explícita de un árbol de sintaxis abstracta (AST), el proceso requirió una inversión considerable de tiempo y esfuerzo. No obstante, alcanzar una versión final completamente funcional ha resultado altamente satisfactorio.

En conclusión, la elección de este tema como Trabajo de Fin de Grado ha sido una decisión acertada. A lo largo del desarrollo del proyecto, he adquirido un conocimiento profundo y extenso sobre los aspectos clave implicados en la creación de un lenguaje de programación desde cero, incluyendo el diseño léxico, sintáctico, semántico y su integración en una herramienta práctica. Esta experiencia me ha permitido consolidar competencias técnicas avanzadas y comprender de forma integral los retos y trabajo que supone construir un nuevo lenguaje de programación.

7.2. Trabajo Futuro

Aunque el sistema desarrollado cumple con los objetivos iniciales, se han identificado múltiples vías de mejora y ampliación que podrían enriquecer el proyecto:

- **Extensión del lenguaje:** Se podrían incluir nuevas primitivas geométricas ya definidas en VPython como conos o pirámides, así como nuevas operaciones booleanas como intersección y diferencia.

- **Control de errores más robusto:** Mejorar el sistema de gestión de errores léxicos y sintácticos con mensajes más detallados para ayudar al usuario a depurar sus scripts, y mecanismos de recuperación que permitan seguir interpretando el código parcialmente correcto.
- **Mejora de la interfaz:** La interfaz actual, como se ha mencionado anteriormente, presenta un diseño sencillo, cuyo propósito principal ha sido establecer las bases para el desarrollo futuro de una interfaz más interactiva. Asimismo, ha servido como entorno de pruebas funcionales para el lenguaje diseñado.

  En esta línea, existen diversas propuestas en el ámbito de la interacción persona-computador, como las expuestas en el artículo de González Gonzalez et al. [2023], entre las que destaca la programación bidireccional. Esta técnica permitiría al usuario realizar transformaciones directamente sobre los objetos en la ventana de visualización, sin necesidad de depender exclusivamente de comandos textuales. De esta forma, se facilitaría una interacción más fluida y dinámica, haciendo el sistema más accesible e intuitivo para un mayor número de usuarios.
- **Entorno colaborativo o basado en web:** Migrar la aplicación a una plataforma web permitiría su uso desde el navegador, fomentando el acceso remoto y la colaboración entre usuarios.

En resumen, el presente trabajo establece una base sólida sobre la que construir un lenguaje más potente, con mayores capacidades de modelado y con un entorno de desarrollo cada vez más completo y accesible.

Bibliografía

A. V. Aho, M. S. Lam, R. Sethi, y J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Boston, 2nd edition, 2006.

J. F. Gonzalez, D. Kieken, T. Pietrzak, A. Girouard, y G. Casiez. Introducing bidirectional programming in constructive solid geometry-based cad. In *Proceedings of the 2023 ACM Symposium on Spatial User Interaction*, SUI '23, page 1–12. ACM, Oct. 2023.

O. Micolini, L. Ventre, F. Facundo, y A. David. *SISTEMA DE LOCALIZACIÓN DE PERSONAS MEDIANTE BLUETOOTH*. Universidad Nacional de Córdoba, 04 2022.

P. O. Ohlbrock, P. D'acunto, J.-P. Jasienski, y C. Fivet. Constraint-driven design with combinatorial equilibrium modelling. In *Proceedings of IASS annual symposia*, volume 2017, pages 1–10. International Association for Shell and Spatial Structures (IASS), 2017.

A. A. G. Requicha. Representation of rigid solids: Theory, methods, and systems. *ACM Computing Surveys (CSUR)*, 12(4):437–464, 1980.

Q. Zou, H.-Y. Feng y S. Gao. Variational direct modeling: A framework towards integration of parametric modeling and direct modeling in cad. *Computer-Aided Design*, 157:103465, Apr. 2023.

A Apéndice 1: Análisis Léxico (Lexer) Ampliación

```
1
2       # Token regexs
3       NUMBER = r'(\d)+(\.\d+)?'
4       STRING = r'"([^"\\]|\\"|\\n|\\t)*"'
5       ARROW = r'->'
6       PLUS =r'\+'
7       LESS =r'\-'
8       MUL = r'\*'
9       DIV = '/'
10      LE = r'<='
11      LT = r'<'
12      GE = r'>='
13      GT = r'>'
14      EQ = r'=='
15      NE = r'!='
16      ASSIGN = r'='
17      DOT = r'\.'
18      AND = r'&&'
19      OR = r'\|\|'
20      literals = {',', ';', '{', '}', '(', ')', '[', ']', '|'}
21
22      @_("#.*")
23      def Comentario(self, t):
24          pass
25
26      @_(r'[a-zA-Z_][a-zA-Z0-9_]*')
27      def ID(self, t):
28          keywords = { 'if', 'else', 'return', 'as', 'import', 'define',
                'while', 'and', 'or'}
29          if t.value.lower() in keywords:
30              t.type = t.value.upper()
31          elif t.value.lower() in ['true', 'false']:
32              t.type = 'BOOLEAN'
33          return t
```

Código A.1: Expresiones regulares que definen cada token.

Las expresiones regulares mostradas en la Figura A.1 definen los distintos tokens reconocidos por el analizador léxico. La mayoría de operadores y símbolos (‘PLUS‘, ‘LESS‘, ‘MUL‘, ‘DIV‘, etc.) son definidos de manera directa mediante expresiones simples.

Los tokens `NUMBER`, `STRING` e `ID` requieren expresiones regulares más elaboradas:

- **NUMBER**: El token `NUMBER` permite reconocer tanto números enteros como números reales (decimales positivos). Su expresión regular es:

```
NUMBER = r'(\d)+(\.\d+)?'
```

Esta expresión se compone de dos partes:

- `(\d)+` indica uno o más dígitos (0–9), capturando así la parte entera del número.
- `(\.\d+)?` es un grupo opcional (indicado por el `?`) que permite capturar un punto decimal seguido de uno o más dígitos. Esto habilita la detección de números reales (por ejemplo, `3.14`, `0.5`).

Este token no considera números negativos, ya que el signo `-` se trata como un operador separado y no forma parte del literal numérico, los números negativos se definen en el parser mediante una regla gramatical propia que se puede ver en el apéndice B. Tampoco reconoce notación científica (e.g., `1e5`), ya que no es necesaria dentro del dominio del lenguaje desarrollado.

- **STRING**: Explicado en la Sección 4.1.1.
- **ID**: El token `ID` (identificador) representa nombres de variables, funciones o clases definidos por el usuario. Su expresión regular es:

```
ID = r'[a-zA-Z_][a-zA-Z0-9_]*'
```

Esto define que:

- Un identificador debe comenzar por una letra (mayúscula o minúscula) o un guion bajo `_`.
- A partir del segundo carácter, puede contener letras, dígitos o guiones bajos.

Ejemplos válidos incluyen `objeto`, `_variable1`, `Transform3D`, `x1_y2`.

Dentro del método correspondiente en el lexer, se realiza una comprobación posterior para verificar si el identificador coincide con una palabra clave (como `if`, `define`, etc.) o con un valor booleano (`true`, `false`). En dichos casos, se asigna el tipo de token correspondiente en mayúsculas. Si no coincide con ninguna palabra clave reservada, se considera un identificador común.

Esta verificación adicional es importante para distinguir entre identificadores definidos por el usuario y construcciones propias del lenguaje, manteniendo así la coherencia semántica durante el análisis sintáctico.

Los **comentarios** se definen con la expresión regular `#.*`, que indica que cualquier línea que comience por el carácter `#` y continúe con cualquier secuencia de caracteres hasta el final de la línea será ignorada por el lexer, al no generar tokens. Esta expresión hace uso del carácter `.` (cualquier carácter) y `*` (cero o más repeticiones), siendo ambos símbolos especiales dentro del contexto de expresiones regulares.

Finalmente, el conjunto de `literals` recoge todos los signos de puntuación usados por el lenguaje que no requieren una expresión regular compleja. En particular, el carácter `|` fue añadido como literal en fases más avanzadas del desarrollo, debido a su uso específico como operador auxiliar, tal y como se detalla en la Sección 4.2.3.

B Apéndice 2: Análisis Sintáctico (Parser) Ampliación

A continuación se detalla la gramática completa del lenguaje creado:

```
program      : orders

orders       : orders order
             | order

order        : statements ;
             | DEFINE ID ( arguments ) { statements ; } ;
             | DEFINE ID ( ) { statements ; } ;
             | DEFINE ID ( arguments ) { statements ; RETURN expr ; } ;
             | DEFINE ID ( ) { statements ; RETURN expr ; } ;
             | DEFINE ID ( arguments ) { RETURN expr ; } ;
             | DEFINE ID ( ) { RETURN expr ; } ;
             | IMPORT ID AS ID ;
             | IMPORT ID ;

arguments    : arguments , argument
             | argument

argument     : expr

statements   : statements ; statement
             | statement

statement    : assign
             | condStruct
             | loop
             | expr

assign       : ID ASSIGN expr

condStruct   : IF expr { statements ; } ELSE { statements ; }
             | IF expr { statements ; }

loop         : WHILE expr { statements ; }

expr         : expr ARROW expr
             | ID '|' ID
             | ID
```

```
              | invokeFunc
              | attribute
              | expr PLUS expr
              | expr LESS expr
              | expr MUL expr
              | expr DIV expr
              | expr EQ expr
              | expr NE expr
              | expr GT expr
              | expr GE expr
              | expr LT expr
              | expr LE expr
              | expr OR expr
              | expr AND expr
              | [ exprs ]
              | [ ]
              | NUMBER
              | STRING
              | BOOLEAN
              | LESS expr
              | ( expr )

invokeFunc    : ID ( arguments )
              | ID ( )

attribute     : ID DOT ID
              | ID DOT ID ( arguments )
              | ID DOT ID ( )
              | attribute DOT ID
              | attribute DOT ID ( arguments )
              | attribute DOT ID ( )

exprs         : exprs , expr
              | expr
```

Es importante destacar que, aunque un fragmento de código cumpla con la sintaxis establecida por la gramática, esto no garantiza que sea correcto desde un punto de vista semántico. Por ejemplo, en nuestro lenguaje, los números negativos se definen mediante la producción `LESS expr`. Sin embargo, dado que `expr` puede representar distintos tipos de valores, puede darse el caso de una combinación como `LESS STRING` que sea sintácticamente válida pero carezca de sentido, es decir, que semánticamente fuese incorrecta.

Por esta razón, las reglas susceptibles a este tipo de incoherencias incluyen controles semánticos específicos en el parser. En el caso de `LESS expr`, el parser genera un error si la evaluación final de `expr` no corresponde a un valor numérico (`NUMBER`). Para un análisis detallado de estos controles semánticos, se puede consultar el código fuente del parser en el archivo