Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo de Fin de Grado

Procesador de Lenguajes Genérico para un Sistema Auto-corrector de Prácticas

Autor: Andrés Ollero Morales

Tutor: José Luis Fuertes Castro

Madrid, junio 2024

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo de Fin de Grado
Grado en Ingeniería Informática
Título: Procesador de Lenguajes Genérico para un Sistema Auto-corrector de Prácticas
Junio 2024

Autor: Andrés Ollero Morales

Tutor:
José Luis Fuertes Castro
Lenguajes y Sistemas Informáticos e Ingeniería de Software
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

DRACO (Dinámica de Refuerzo del Aprendizaje de COmpiladores) es un sistema informático que comenzó su desarrollo en 2013. El sistema trata de mejorar la calidad de la enseñanza por medio de la gamificación en las asignaturas de Procesadores de Lenguajes y Traductores de Lenguajes.

DRACO permite a los alumnos acceder a una interfaz con actividades que sirven para repasar los conocimientos dados en clase, y al resolverse una actividad los alumnos ganan puntos para ascender dentro del *ranking* de DRACO. DRACO también ofrece a los profesores una interfaz para poder consultar las estadísticas generales del sistema y las estadísticas individuales de cada alumno.

Otra funcionalidad que tiene DRACO es comprobar los diferentes módulos de las prácticas de las asignaturas. Esta funcionalidad será en la que se centre este Trabajo de Fin de Grado. La práctica de Procesadores de Lenguajes consiste en implementar la parte de análisis de un compilador, que está formado por el Analizador léxico, el Analizador sintáctico y el Analizador semántico. La práctica de Traductores de Lenguajes consiste en implementar la parte de síntesis de un compilador, que está formado por el Generador de Código Intermedio y el Generador de Código Objeto.

La interfaz del profesor de DRACO contiene una serie de apartados para poder definir lenguajes que se usarán en las asignaturas. Estos apartados se presentan como formularios que le van pidiendo al profesor información hasta completar la especificación del lenguaje. Para la realización de este Trabajo no solo se implementó el módulo de síntesis, también se realizaron mejoras y ampliaciones a las páginas de especificación del lenguaje.

El módulo de comprobación de prácticas de DRACO solo comprueba las prácticas de Procesadores de Lenguajes, generando una necesidad por parte de los alumnos de crear un sistema que compruebe las prácticas de Traductores de Lenguajes. Este Trabajo de Fin de Grado se encargará de implementar la parte de análisis de un compilador genérico que se adapte al lenguaje que escoja el profesor para las prácticas.

Abstract

DRACO (Dinámica de Refuerzo del Aprendizaje de COmpiladores) is a computer system that began development in 2013. The system aims to improve the quality of teaching through gamification in the subjects of Procesadores de Lenguajes and Traductores de Lenguajes. DRACO allows students to access an interface with activities that help review the knowledge given in class. By solving an activity, students earn points to climb the DRACO ranking. DRACO also offers teachers an interface to view each student's general system statistics and individual statistics.

Another functionality of DRACO is checking the course assignments' different modules. This functionality will be the focus of this final degree project. The Procesadores de Lenguajes assignment involves implementing the analysis part of a compiler, which consists of the Lexical Analyzer, the Syntactic Analyzer, and the Semantic Analyzer. The Traductores de Lenguajes assignment involves implementing the synthesis part of a compiler, which consists of the Intermediate Code Generator and the Object Code Generator.

The teacher's interface in DRACO contains a series of sections defining languages used in the subjects. These sections are presented as forms that ask the teacher for information until the language specification is complete. For this project, not only was the synthesis module implemented, but also improvements and expansions were made to the language specification pages.

The practice-checking module of DRACO only checks the assignments for Procesadores de Lenguajes, creating a need for students to have a system that checks the assignments for Traductores de Lenguajes. This final degree project will implement the analysis part of a generic compiler that adapts to the language chosen by the teacher for the assignments.

Tabla de contenido

1 Introducción 1
2 Estado de la cuestión 4
2.1 Desarrollo web 4
2.1.1 HTML 4
2.1.2 CSS 6
2.1.3 PHP 8
2.1.4 SQL 10
2.2 Compiladores 10
2.2.1 Análisis Léxico 11
2.2.2 Análisis Sintáctico 12
2.2.3 Análisis Semántico 12
2.2.4 Tabla de Símbolos 12
2.2.5 Generador de código intermedio 13
2.2.6 Generador de código objeto 13
2.3 DRACO 13
3 Planteamiento del problema 15
3.1 Introducción 15
3.1.1 Propósito 15
3.1.2 Ámbito del sistema 15
3.1.3 Definiciones y abreviaturas 15
3.1.3.1 Definiciones 15
3.1.3.2 Abreviaturas 16
3.2 Descripción general 16
3.2.1 Perspectiva del producto 16
3.2.2 Funciones del sistema 16
3.2.2.1 Analizador léxico 17
3.2.2.2 Analizador sintáctico 17
3.2.2.3 Analizador semántico 17
3.2.2.4 Tabla de símbolos 17
3.2.2.5 Configuración del lenguaje 17
3.3 Requisitos específicos 17
3.3.1 Requisitos funcionales 17
3.3.1.1 Analizador léxico 17
3.3.1.2 Tabla de símbolos 18
3.3.1.3 Analizador sintáctico 19
3.3.1.4 Analizador semántico 19
3.3.1.5 Configuración del lenguaje 19
3.3.2 Requisitos de interfaces externas 20
3.3.2.1 Interfaz de subida de prácticas 20
3.3.2.2 Interfaz de configuración de las actividades de DRACO 20
3.3.2.3 Base de datos de DRACO 20
4 Solución 21

4.1 Diseño 21
4.1.1 Analizador léxico 25
4.1.1.1 Algoritmo del Analizador léxico 26
4.1.1.2 Algoritmo de Análisis del fichero y devolución de tokens 26
4.1.1.3 Reconocimiento de comentarios 27
4.1.1.4 Reconocimiento de cadenas y de caracteres 27
4.1.1.5 Reconocimiento de números 28
4.1.1.6 Reconocimiento de símbolos 29
4.1.1.7 Reconocimiento de palabras reservadas 30
4.1.1.8 Reconocimiento de identificadores 31
4.1.1.9 Reconocimiento de saltos de línea 31
4.1.2 Tabla de símbolos 32
4.1.3 Analizador sintáctico 32
4.1.3.1 Importación de la Gramática y aumento de la Gramática 32
4.1.3.2 Ítems 32
4.1.3.3 Cierre de un conjunto de ítems 32
4.1.3.4 Función Goto de un conjunto de ítems y un símbolo de la Gramática 33
4.1.3.5 Creación de la colección canónica 33
4.1.3.6 First y Follow de un símbolo gramatical 33
4.1.3.7 Algoritmo de creación de la tabla Acción y Goto 34
4.1.3.8 Algoritmo de análisis sintáctico 34
4.1.4 Analizador semántico 35
4.1.5 Configuración del sistema 35
4.2 Implementación 37
4.2.1 Analizador léxico 37
4.2.1.1 Clase token 37
4.2.1.2 Clase AnLex 38
4.2.1.3 Algoritmo de análisis de la clase AnLex 39
4.2.1.4 Funciones auxiliares utilizadas en el Analizador léxico 40
4.2.1.4.1 Función valueOf 40
4.2.1.4.2 Función getTokens 40
4.2.1.4.3 Función inRegex 40
4.2.2 Tabla de símbolos 40
4.2.2.1 Clase Key 40
4.2.2.2 Clase SymbolTable 41
4.2.3 Analizador sintáctico 41
4.2.3.1 Clase AnSin 41
4.2.3.2 Clase Item y clase ItemGroup 41
4.2.3.3 Clase ColeccionCanonica 42
4.2.3.4 Clase TablaAG 42
4.2.3.5 Algoritmos necesarios para el análisis 42
4.3 Pruebas 43
4.3.1 Uso del IDE y el depurador 43
4.3.2 Uso del log de PHP 44

4.3.3 Pruebas realizadas 45
4.3.3.1 Pruebas unitarias 45
4.3.3.2 Pruebas de integración 45
5 Resultados y conclusiones 46
6 Análisis de impacto 48
6.1 Impacto social 48
6.2 Impacto personal 48
6.3 ODS 48
6.3.1 ODS 4: Educación de calidad 48
6.3.2 ODS 8: Trabajo decente y crecimiento económico 48
6.3.3 ODS 9: Industria, innovación e infraestructuras 48
7 Futuras líneas de trabajo 50
8 Bibliografía 52

1 Introducción

El objetivo de este Trabajo de Fin de Grado consiste en ampliar las funciones de DRACO (Dinámica de Refuerzo del Aprendizaje de COmpiladores). DRACO tiene como objetivo ser la web de apoyo para las asignaturas de Procesadores de Lenguajes y Traductores de Lenguajes, que son impartidas en la Escuela Técnica Superior de Ingenieros Informáticos. El sistema fue creado en 2013 y se mantiene en desarrollo gracias a la colaboración de estudiantes que trabajan como becarios, voluntarios o realizan sus Prácticum o Trabajos de Fin de Grado en DRACO.

Entre las funcionalidades de este sistema están diferentes actividades, como en las que el alumno debe escoger entre varias opciones, ordenar opciones de la manera correcta o preguntas de respuesta corta. La puntuación obtenida en estas actividades se traslada a un *ranking* que ordena a los alumnos según la cantidad de puntos que han obtenido, al final del curso. Esta puntuación se ve reflejada en un porcentaje de la nota final. Además, con estos puntos se pueden comprar monedas llamadas DRACOins, que pueden ser obtenidas de esta manera o encontrando códigos llamados DRACOdes que están repartidos por la Escuela, y que pueden dar puntos o monedas.

Las actividades tienen un tiempo límite. Si el alumno no hace una actividad antes de la fecha límite no podrá obtener los puntos de dicha actividad. Los alumnos también pueden subir de nivel haciendo actividades, lo que les permite acceder a más actividades para obtener más puntos. A continuación, en la ilustración 1, se muestra la interfaz de DRACO que contiene el *ranking* de los alumnos.

Ilustración 1: Página de clasificación de DRACO.

El sistema se encarga también de corregir los distintos módulos de las prácticas de Procesadores de Lenguajes. Para esto se deben corregir los cuatro módulos de la práctica, el analizador léxico, sintáctico, semántico y la tabla de símbolos. Estos módulos se corrigen con actividades en las que el alumno elige qué funcionalidades optativas va a implementar en su práctica y los nombres que va a poner a diferentes elementos de la práctica, como pueden ser los nombres de los *tokens* del analizador léxico, la definición de terminales del analizador sintáctico y la definición de tipos del analizador semántico. En la ilustración 2, se muestra la parte de la interfaz del profesor de DRACO que se encarga de gestionar las actividades de las prácticas.

Las especificaciones del lenguaje se definen al principio del año en el apartado de administración del lenguaje de DRACO, con múltiples páginas donde el profesor debe definir el comportamiento del lenguaje.

Las pantallas de definición del lenguaje que hay actualmente son:

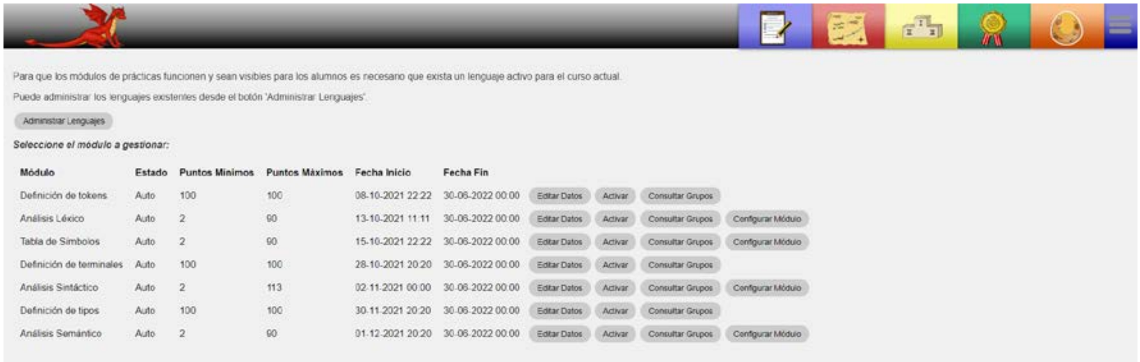

Ilustración 2: Página de gestión de las actividades de la práctica en DRACO.

- Definición de elementos del lenguaje: En esta pantalla se declaran los elementos que tiene el lenguaje, como pueden ser las palabras reservadas, los operadores, los identificadores, las constantes y los comentarios.
- Asignación de los códigos de *token*: En esta pantalla se declara la estructura de los *tokens*, el nombre a mostrar, el código asignado a cada *token*, si tiene atributo y si este atributo es fijo.
- Propiedades de los operadores: En esta pantalla se define la cardinalidad, la asociatividad, la prioridad y el tipo de operador (preoperador, postoperador…).
- Roles de los elementos del lenguaje: Esta pantalla recopila la información acerca de si un lexema es un operando, un operador u otra estructura, también se declara qué *tokens* marcan el fin de una sentencia.
- Características de los elementos del rol 'Otros': Si en la anterior pantalla se ha seleccionado un *token* y se ha dicho que es otra estructura, en esta pantalla se define qué función tiene dentro del lenguaje, y si pertenece a una estructura o no., En el caso de que sí pertenezca a una estructura, se debe decir si está en el interior, a continuación o si comienza una nueva estructura.
- Propiedades sobre las estructuras de bloque: En esta pantalla se definen los *tokens* que dan inicio al campo de condición de cada estructura.
- *Tokens* de inicio y fin de las estructuras de bloque: En esta pantalla se definen los *tokens* que dan inicio y fin a las estructuras de bloque y si hay una pareja adicional para declarar el principio y el final de una estructura de bloque.
- Pareja de *tokens* adicional de inicio y fin de las estructuras de bloque: Si en la pantalla anterior se ha marcado la opción de añadir una pareja de tokens adicionales en esta pantalla se deberán definir los *tokens* adicionales de inicio y fin.
- *Tokens* de tipos de datos: En este apartado se deben configurar los *tokens* que indican un tipo variable, un tipo parámetros o un tipo de valor devuelto
- Descripción de los *tokens* de tipos de datos: En esta pantalla se debe definir para cada lexema el tipo al que pertenece y el tamaño que tiene.
- Configuración de declaraciones de variables: En esta pantalla se debe definir cómo se declaran variables locales y globales. Se puede indicar en esta pantalla si la declaración de las variables globales es igual a la de las variables locales. Para cada declaración, se deben definir los *tokens* que la conforman y si se puede repetir el mismo lexema para declarar varias variables.
- Configuración de identificadores no declarados: En esta pantalla se define si el lenguaje admite identificadores no declarados y si, en el caso

de admitirlos, se comportan como una variable local o una global y cuál es el tipo de datos que se le asigna a los identificadores no declarados. En esta pantalla además de define si el lenguaje tiene diferentes modos de paso de parámetros y su estructura en el caso de que si hubiera diferentes modos.

- Configuración de la declaración de parámetros: En esta pantalla se definen las diferentes maneras que tiene el lenguaje para declarar un parámetro dentro de una función.
- Configuración de la declaración de funciones: En esta pantalla se definen las diferentes maneras que tiene el lenguaje de declarar las funciones y la estructura que tienen.

En este Trabajo de Fin de Grado se va a implementar la primera parte del corrector de las prácticas de Traductores de Lenguajes, en la que se evalúan los módulos de Generación de Código Intermedio y de Generación de Código Objeto, sin embargo, para poder hacer el corrector de estos módulos se debe comparar el fichero generado por un alumno con el fichero generado por el sistema. Para esto se debe implementar la primera parte del compilador que deberá adaptarse a los cambios que haya en el lenguaje para no tener que hacer un compilador específico del lenguaje cada año.

La información que está en el sistema actualmente es insuficiente para desarrollar un compilador funcional para el lenguaje. Para esto se deberán añadir y modificar algunas páginas al módulo de definición del lenguaje, así como modificar la base de datos de DRACO para poder almacenar la información adicional que se introduzca en las páginas de definición del lenguaje.

Para el desarrollo de este Trabajo será necesario conocer varias tecnologías de desarrollo web, concretamente PHP, CSS y HTML. También será necesario gestionar bases de datos y poder realizar consultas complejas, para esta tarea será necesario conocer MySQL, que es el software utilizado para gestionar las bases de datos del sistema. También será necesario tener conocimientos acerca de cómo funciona un compilador en todas sus fases, ya que una decisión de diseño en un módulo puede afectar a los módulos posteriores.

2 Estado de la cuestión

En este capítulo se abordarán los conocimientos y tecnologías fundamentales para comprender el desarrollo del Trabajo. Se expondrán los principios teóricos y las herramientas prácticas y cómo se usan en conjunto.

2.1 Desarrollo web

El desarrollo web se define como el proceso de creación de un sitio web; esto incluye aspectos como la estructuración del contenido, el diseño gráfico y la accesibilidad. Este término se comenzó a usar cuando en 1990, Tim Berners-Lee desarrolló el primer navegador web, al que llamó WorldWideWeb [1], las primeras páginas web contenían texto plano y enlaces a otros sitios web; las imágenes y los videos no eran soportados hasta que en 1993 se lanzó Mosaic, que permitió navegar por la web de forma gráfica. Posteriormente, en 1995 surgió Internet Explorer, que venía por defecto instalado en los ordenadores con Windows 95 [2] y permitía a los usuarios navegar por múltiples páginas web a la vez.

Durante esa década, además, se introdujeron varias tecnologías de desarrollo como JavaScript y Flash, que permiten a los sitios ser interactivos y mostrar animaciones. También surgió la necesidad de crear entornos de desarrollo web, como DreamWeaver, que daba la posibilidad a los usuarios de crear páginas web.

En 2004 se creó la Web 2.0 [3], que fue una manera distinta de entender la web, integrando al usuario en el funcionamiento de las páginas web, no solo como consumidor del contenido. Algunos ejemplos de este enfoque son las redes sociales o las wikis, en las que los usuarios crean contenido.

Durante la década de los 2000 se hizo más popular la creación de páginas web, ya que con el concepto de web 2.0 se comenzaron a crear *blogs* que explicaban cómo desarrollar una página web o tecnologías como Blogger, que permitía a los usuarios desarrollar una página web sin tener conocimientos de programación [4].

En esa misma década surgió el concepto de Web 3.0, que se diferenciaba de la Web 2.0 por el uso de los datos proporcionados por el usuario, haciendo uso de estos para darle una experiencia personalizada a cada usuario [5]. En estos últimos años, con la aparición de tecnologías como Big Data y Blockchain, o el desarrollo de tecnologías ya existentes como la Inteligencia Artificial, han hecho que este enfoque sea el más usado para desarrollar nuevos proyectos.

2.1.1 HTML

HTML o HyperText Markup Language surge en 1993 como el componente que define el significado y la estructura de un sitio web [6]. HTML usa etiquetas para etiquetar el texto, por ejemplo, una etiqueta que se usa para definir texto plano es “`<p>`”.

Una etiqueta de HTML se compone de 3 elementos, el *tag* de apertura, en el que se pueden introducir atributos, el contenido, que contiene lo que se quiere mostrar al usuario y el *tag* de cierre, para indicar el final del contenido.

HTML 1.0 fue la primera versión del lenguaje de marcas lanzado en 1993 por Tim Berners-Lee. Esta versión contenía un conjunto de etiquetas básico que permitía mostrar texto y vínculos a otras páginas [7]. El 1 de octubre de 1994,

Tim Berners-Lee y el MIT crearon el consorcio W3 (W3C) para definir estándares para Internet. En la ilustración 3 se muestra un ejemplo de cómo usar una etiqueta HTML.

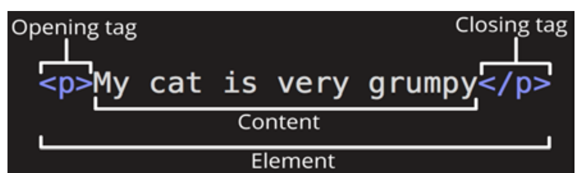

Ilustración 3: Ejemplo de la estructura de una marca HTML.

En 1995 se lanzó HTML 2.0, que era una mejora de la versión anterior, dando soporte a formularios y *scripts*. Esta versión fue el primer estándar definido por un organismo oficial, llamado IETF (Internet Engineering Task Force) [8]

HTML 3.2 fue publicado en 1997, respaldado por el W3C, que añadía soporte para tablas y marcos. Posteriormente, ese mismo año, se lanzó la cuarta versión de HTML, que añadía soporte para hojas de estilo externas y marcos en línea. En 1999, W3C se volcó en el desarrollo de XHTML, que se lanzó en el año 2000 y se basó en la sintaxis de XML [9].

HTML 5 surgió como una extensión de HTML 4.01 en el año 2014: Es la versión más utilizada actualmente y fue publicado por el WHATWG (Web Hypertext Application Technology Working Group).

Para la realización de este Trabajo se ha hecho uso de la versión 5 de HTML, de forma que para todas las páginas HTML se ha seguido una estructura que divide el código en dos elementos, `<head>` y `<body>`.

Dentro de la etiqueta `<head>` se sitúan los datos que no están expuestos directamente en el contenido de la página web. Esto se hace con etiquetas `<meta>` para definir atributos como el conjunto de caracteres a usar, etiquetas `<link>` que hacen referencia a recursos que se usan para dar forma a la página web, como la hoja de estilos, usando el atributo "`rel`" para definir cuál es la función del fichero al que se está referenciando y la etiqueta `<title>`, para definir el título que se mostrará en la pestaña del navegador. A continuación, se muestra un ejemplo de cómo usar la etiqueta <meta> y sus contenidos.

```
<head>
      <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
      <title>Definición de un Lenguaje | Gamif PDL</title>
      <link rel="stylesheet" type="text/css"
         href="../../css/admin.css" media="screen" />
      <link rel="shortcut icon" href="../../images/favicon.ico"
         type="image/x-icon">
      <link rel="icon" href="../../images/favicon.ico" type="image/x-icon">
</head>
```

En la sección `<body>` se incluye la estructura y el contenido de la página web, usando elementos como `<section>` o `<div>`, para separar el contenido de la página según sus características o funcionalidad. Para conseguir más modularidad dentro de la página, en estos elementos de separación se puede usar el atributo "*class*", que identifica de manera diferente a los elementos que tengan este atributo. A continuación, se muestra un ejemplo del uso de la etiqueta `<div>`.

```
<div class="red-panda">
      <picture>
            <source media="(max-width: 600px)" srcset="red-panda600.jpg" />
```

```
        <source media="(min-width: 601px)" srcset="red-panda1200.jpg" />
        <img src="red-panda1200.jpg" alt="PANDA">
    </picture>
</div>
```

Para mostrar contenido dentro del elemento <body> se usan etiquetas como:

- <p> para definir el contenido de los párrafos; dentro de los párrafos se pueden incluir etiquetas para destacar el texto, como <em> para mostrar el texto con énfasis, <strong> para mostrar el texto resaltado y <br>, para incluir un salto de línea.
- <h1> se usa para mostrar el contenido del título de la página. Esta etiqueta tiene variantes, que van desde <h1> a <h6>, representando distintos niveles de profundidad en cuanto a la organización del contenido de la página.
- <a> para definir los vínculos que tiene esta página usando su atributo "href".
- y <ul> definen listas ordenadas y no ordenadas, respectivamente. Dentro de estas etiquetas se usan otras llamadas *list item*, <li>, que definen los elementos de la lista que los contiene.
- <table> indica que se va a definir una tabla, usando <thead> para definir la fila encabezado de la tabla y <tbody> para definir la estructura de las siguientes filas de la tabla. Estos elementos a su vez contienen la etiqueta <tr>, que define una fila de la tabla y <th> y <td>, que representan las celdas de la tabla, siendo <th> una celda de encabezado y <td> una celda de datos.
- <form> se usa para definir un formulario que puede rellenar el usuario con datos. Con el atributo "*action*" y "*method*" se pueden procesar los datos introducidos en él con tecnologías como PHP y JavaScript. Dentro de los formularios puede haber elementos que contengan la etiqueta <input>, que es el elemento encargado de permitir al usuario introducir datos, que se pueden mostrar en forma de campo de texto, de c*heckbox*, etc., según lo que se introduzca en el atributo "*type*".

2.1.2 CSS

CSS o *Cascading Style Sheets* es una tecnología que permite dar estilo a los elementos de HTML. Nació en 1996 desarrollado por el W3C como solución a la demanda de los usuarios de sitios web más atractivos [10].

En diciembre de 1996 se publicó CSS1 [11], la primera especificación de CSS, recomendada por el W3C para desarrollar páginas web. Usando esta especificación, se podían dar propiedades como énfasis y espaciado a los párrafos, modificar los márgenes y los bordes del documento, así como la posición de la mayoría de los elementos.

CSS2 fue publicado en mayo de 1998 como una ampliación al conjunto de operaciones de CSS1. Añadió propiedades como la profundidad de los elementos en la página web, permitiendo el súper-posicionamiento de los elementos, nuevas maneras de posicionar los elementos, como posicionamiento absoluto o relativo y nuevas propiedades de los textos.

CSS3 se comenzó a desarrollar en el momento en que CSS2 se publicó como recomendación por el W3C. La diferencia principal con el resto de los niveles de CSS es que CSS3 no se presentaba como una colección extensiva de operaciones, sino que estaba dividido en diferentes documentos llamados módulos, que se centran en una característica específica del lenguaje. En 2011,

el W3C recomendó CSS3 para el desarrollo de páginas web, y actualmente cada módulo se desarrolla de manera independiente con un nivel propio de especificación.

Para la realización de este Trabajo se ha utilizado la versión 3 de CSS. La sintaxis de CSS se basa en tres elementos: el selector, que se utiliza para identificar los elementos HTML a los que se quiere aplicar el estilo, la propiedad que se quiere cambiar del elemento y el valor de la propiedad que es elegida para cambiar el estilo del elemento. En la ilustración 4 se muestra un ejemplo de la sintaxis mencionada anteriormente.

Ilustración 4: Ejemplo de uso de CSS3

En CSS se puede elegir qué elemento escoger de varias formas:

- Se pueden escoger todas las instancias de un tipo de elemento; por ejemplo, poner "*p*" en el selector selecciona todos los elementos HTML de este tipo sin importar su clase o ubicación
- Se pueden escoger según la clase o el identificador (que son atributos que se le pueden añadir a todas las etiquetas HTML). Para escoger elementos según su identificador se escribiría # y el nombre del identificador en el selector de CSS y si se quisiera escoger un elemento según la clase se utilizaría seguido del nombre de la clase.

En CSS también se pueden seleccionar elementos según si tienen declarado un atributo en concreto (por ejemplo, poner `img[src]` escogería todos los elementos HTML del tipo imagen que tuvieran declarado el atributo "*src*") y se pueden seleccionar elementos según el estado en el que están (por ejemplo, `a:hover` en el selector haría que se aplicase el estilo descrito en esa regla a todas las *anchors* que en ese momento tengan el cursor encima).

Para estructurar una página HTML con CSS se suele seguir un modelo llamado "modelo de caja" [12], que se basa en que todo elemento en HTML está rodeado de una caja. Estas cajas están conformadas por cuatro elementos:

- El contenido de la caja, que es la caja más interna y la zona en la que se muestra el contenido. En el código CSS se puede cambiar usando los atributos *height* y *width*.
- El relleno o *padding* de la caja, que es el espacio en blanco alrededor del contenido. Dentro del código CSS se puede cambiar usando el atributo *padding*.
- El borde de la caja, que envuelve la caja de contenido y el *padding*. En el código CSS es posible modificar su tamaño y estilo usando el atributo "*border*".
- El margen de la caja, que es la capa más externa y sirve como espacio en blanco entre esta caja y otros elementos. Este elemento se puede modificar usando el atributo "*margin*".

2.1.3 PHP

PHP es un lenguaje de programación que sucedió a un producto llamado PHP Tools, creado por Rasmus Lerdorf en 1994, que eran un conjunto de programas escritos en C para contar cuánta gente había visitado su página personal en la que exponía su CV. Posteriormente a este conjunto de *scripts* lo llamaría "*Personal Home Page Tools*". El uso de esta tecnología hizo que el lenguaje se desarrollase durante ese año, añadiendo soporte para conectarse a una base de datos, entre otras funcionalidades. Esto hizo que PHP Tools se usase como una herramienta para hacer páginas web dinámicas [13].

En junio de 1995, se liberó el código de PHP Tools, haciendo el proyecto un proyecto de código abierto, en el que los desarrolladores podían hacer cambios, añadir funcionalidad o arreglar *bugs*.

En septiembre de 1995, PHP cambió de nombre a FI, "*Forms Interpreter*". Esta implementación contiene algunas características propias de las versiones de PHP más actuales, como la inclusión de código HTML, sin embargo, esta sintaxis hacía difícil el desarrollo web. Entonces, en octubre de 1995, se lanzó "*Personal Home Page Construction Kit*", que era un cambio completo a todo el código, haciendo que la sintaxis se pareciera mucho a lenguajes como C o Perl.

En abril de 1996, se publicó "PHP/FI" como una versión beta que transformó PHP de un conjunto de *scripts* a un lenguaje de programación que se podría usar solo, tenía soporte para conexión a bases de datos, *cookies* y funciones definidas por el usuario.

En noviembre de 1997 se publicó PHP/FI de forma oficial, al mismo tiempo en el que se estaba cambiando el analizador sintáctico del lenguaje. En este momento PHP tenía una limitación principal, estaba siendo desarrollado por una sola persona, lo que hacía que el desarrollo del lenguaje fuera más lento.

En junio de 1998 se presentó PHP 3.0 "*Hypertext Preprocessor*" como sucesor de PHP/FI 2.0, que fue desarrollado por Rasmus Lerdorf y otras dos personas, Andi Gutmans y Zeev Suraski, que se sumaron al desarrollo de PHP tras hablar con Rasmus Lerdorf. PHP 3.0 fue un éxito debido a que era un lenguaje muy extensible, ya que permitía a los usuarios acceder a bases de datos, API y protocolos usando únicamente PHP.

A finales de ese mismo año, después de que PHP fuera lanzado, se comenzó un cambio al código de PHP, usando un nuevo mecanismo, llamado "*Zend engine*", que añadió un sistema de *parser* más avanzado.

En julio de 2004 se anunció la salida de PHP 5 [14], que daba soporte a un nuevo modelo de objetos gracias a la actualización de la "*Zend engine*". En este momento el equipo de desarrollo de PHP contaba con docenas de desarrolladores, ya que PHP era un lenguaje muy usado para el desarrollo web en ese momento.

La versión 6 de PHP no se lanzó nunca; esta consistía en dar soporte a los caracteres Unicode desde el núcleo de PHP, lo que traía la capacidad de usar *emojis* y funciones de manejo de cadenas complejas. El proyecto fue oficialmente cancelado en 2012 y las mejoras que añadía esta versión se fueron añadiendo a las versiones menores de PHP 5 [15].

PHP 7 se lanzó en 2015 con mejoras de rendimiento importantes respecto a sus versiones anteriores, volviendo a situar a PHP entre los lenguajes más usados. Entre las nuevas funcionalidades que incluye PHP 7 se destaca la mejora en la

velocidad y el uso de memoria, así como la inclusión del tipo de retorno de las funciones y la posibilidad de declarar clases anónimas.

El 26 de noviembre de 2020 se lanzó PHP 8, que añadió el tipo *enum* y la posibilidad de declarar números en octal, así como variables de tipo *final* y atributos de función de solo lectura.

La versión de PHP usada para desarrollar este proyecto es PHP 7.4, que usa *Zend Engine 3*, clases anónimas, declaración del tipo de retorno de las funciones y tipos escalares.

PHP es un lenguaje que se ejecuta en el servidor, al contrario que otros lenguajes como JavaScript. PHP genera la página HTML en el servidor y la envía al usuario sin que el usuario tenga acceso a ninguna de las variables ni a los *scripts* que están situados en el servidor. En la ilustración 5 se muestra el funcionamiento interno de PHP.

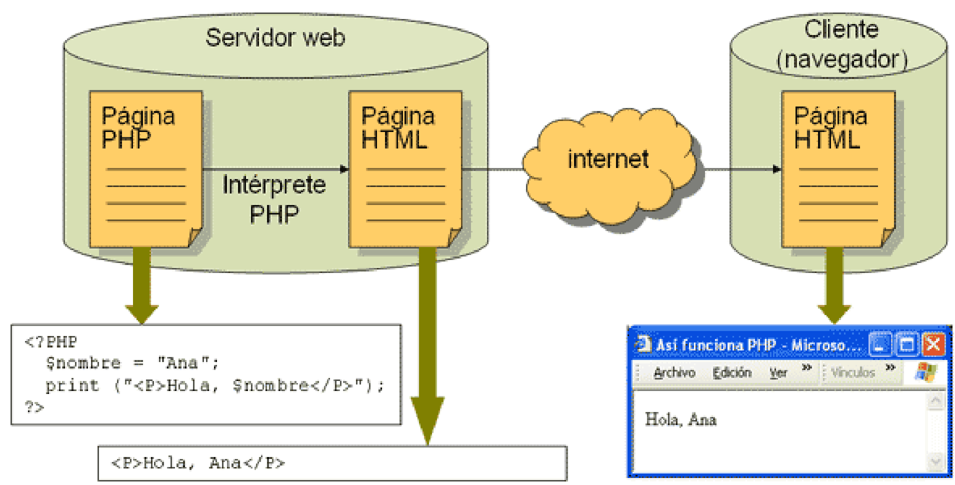

Ilustración 5: Funcionamiento interno de PHP

Para comenzar a escribir un programa en PHP, se debe iniciar el *script* con "`<?php`" y cerrarlo con "`?>`". Dentro de estas marcas se incluye todo el código en PHP. Algunos ejemplos de código en PHP se muestran a continuación.

```
<?php
	$hello = "Hello world";
	echo $hello;
?>
```

En el fragmento anterior de código, las variables se inicializan sin especificar de qué tipo son y deben comenzar por el carácter "$". Para imprimir texto en el navegador se usan instrucciones como `echo` o `print`.

```
<?php
echo "<p>This is a paragraph</p>
		<ul>
			<li>This is a list</li>
			<li>Second element</li>
		</ul>"
?>
```

En el fragmento anterior de código, se imprime código HTML usando la instrucción echo. Esta instrucción tiene incluida la interpretación de código HTML y junto a la instrucción `print`, hacen posible que el script genere una

página HTML a la que es posible aplicar estilos usando CSS, generando una página web estática.

2.1.4 SQL

SQL es un lenguaje de programación creado en 1970 para crear y administrar bases de datos relacionales. La primera versión de SQL se llamaba "SEQUEL" (*Structured English Query Language*) y fue creada por IBM. No fue hasta 1981 que se pasaría a llamar SQL, cuando lo lanzaron como parte de un sistema de administración de bases de datos [16].

Después de convertirse en el lenguaje estándar para interactuar con bases de datos relacionales, el *American National Standards Institute* (ANSI) publicó los estándares oficiales de SQL en 1986 y 1987.

SQL es un lenguaje de administración de bases de datos relacionales basado en tablas, las cuales se pueden crear, modificar, añadir filas y eliminarlas, etc. Esto se hace usando una sintaxis basada en palabras reservadas a través de una consulta denominada *query*.

```
SELECT players.name
FROM players
INNER JOIN players_teams
ON player_id = players.id
INNER JOIN teams
ON team_id = teams.id
WHERE teams.name = "Lakers"
GROUP BY shoe_contract_worth
HAVING shoe_contract_worth > 0
ORDER BY shoe_contract_worth DESC
LIMIT 1;
```

Ilustración 6: Estructura de una query de SQL

En la ilustración 6 se muestra una *query* SQL que está ordenada por el orden en el que se ejecutan las instrucciones. En primer lugar, se ejecuta la sentencia `SELECT` y `FROM`, que se encarga de seleccionar las tablas y los atributos de la tabla que interesa devolver en la *query*. Después, se ejecutan las sentencias `JOIN`, que juntan unas tablas con otras según un campo que tengan en común. `SQL` permite unir una tabla con otra de varias maneras usando versiones alternativas de la sentencia `JOIN`. Posteriormente se aplica un filtro a los resultados usando las sentencias `WHERE`, que se encarga de mostrar los elementos que cumplan con el filtro aplicado en la sentencia; por ejemplo, en la ilustración 4, "`WHERE teams.name = "Lakers"`" solo escogería aquellas columnas que tengan como valor "Lakers" en la fila teams.name. Después se agrupan los resultados usando la sentencia `GROUP BY`, que agrupa los resultados según un campo específico de la tabla resultante. `HAVING` filtra estas agrupaciones aplicando un filtro a los grupos que se han formado. Y, por último, se selecciona cómo se van a mostrar los resultados usando las sentencias `ORDER BY` y `LIMIT`. `ORDER BY` especifica el orden en el que se van a mostrar los resultados de la *query* y `LIMIT` limita el número de resultados que se van a mostrar en el conjunto de resultados.

2.2 Compiladores

Un compilador es un programa que se encarga de traducir un programa escrito en un lenguaje de alto nivel (como C o Java) a un lenguaje de bajo nivel (como ensamblador o código máquina). Durante el proceso de compilación se debe informar de cualquier error que se detecte en cualquiera de las fases.

El proceso de compilación se divide en dos fases: la de análisis, formada por el análisis léxico, sintáctico y semántico, y la de síntesis, formada por el generador de código intermedio y el generador de código objeto. Después de la fase de generación de código intermedio se pueden aplicar algoritmos de optimización de código. Durante todas las fases se consulta o se añade información a un módulo del compilador encargado de gestionar la información acerca de las variables del programa, llamado tabla de símbolos [17]. Este proceso está representado a continuación en la ilustración 7.

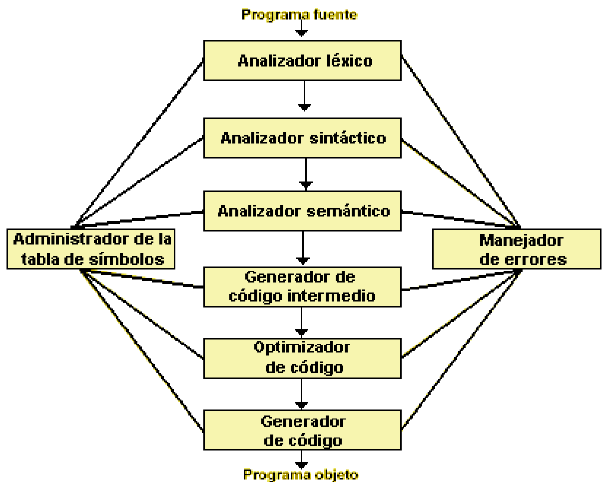

Ilustración 7: Fases de un compilador

2.2.1 Análisis Léxico

El análisis léxico es la parte del compilador que se encarga de leer el programa fuente y generar *tokens* que serán procesados por los módulos posteriores del compilador. Este proceso se implementa usando un Autómata Finito Determinista o AFD, que define la estructura de todos los *tokens* posibles de un lenguaje [17]. Este autómata se construye a partir de una gramática regular que representa la estructura de los elementos del lenguaje. Por ejemplo, para un autómata que puede reconocer números en hexadecimal de Java, la expresión regular y el autómata usados podrían ser los siguientes.

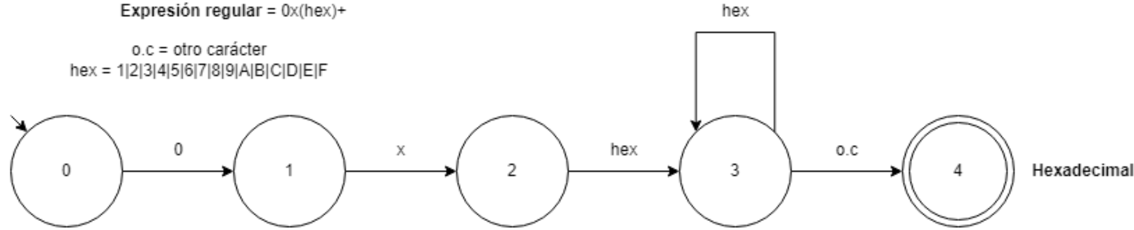

Ilustración 8: AFD de un lenguaje que solo acepta números hexadecimales

En la ilustración 8 se muestra una representación de un AFD formado por estados y transiciones, siendo los estados representaciones del estado del autómata y las transiciones las posibilidades de cambio que tiene un estado. Los estados pueden ser iniciales, que es donde se empieza a procesar el lexema leído, intermedios y finales o de aceptación, que es donde se emite el *token* resultante. Solo se emite un *token* si el conjunto de caracteres procesados por el autómata cumple con la gramática regular definida por el autómata; en otro caso, se tendría que emitir un error léxico.

Cada *token* en un lenguaje viene representado por dos campos, código y atributo. El código sirve para identificar el tipo de *token* que se ha detectado y el atributo sirve para dar información adicional acerca del *token* que se está

tratando. Esta información adicional podría ser un puntero a una posición en la tabla de símbolos, el valor de un número o puede estar vacío.

2.2.2 Análisis Sintáctico

El análisis sintáctico es el módulo del compilador encargado de construir un árbol sintáctico a partir de la gramática del lenguaje y los *tokens* recibidos del analizador léxico.

Las gramáticas están conformadas por símbolos Terminales, No Terminales, Reglas de producción y el Símbolo Inicial. Los símbolos no terminales son aquellos que pueden producir otros símbolos; los símbolos terminales son los tokens que hay en el lenguaje; las reglas de producción se usan para construir el árbol sintáctico y definen las producciones de los símbolos no terminales; y el símbolo inicial es aquel con el que se empieza a construir el árbol sintáctico [17].

El analizador sintáctico funciona consumiendo los *tokens* hasta que se termina el fichero fuente. Si se encuentra una situación no válida, se enviaría un error al usuario.

Algunos errores que pueden surgir durante el análisis sintáctico de un programa pueden ser abrir un corchete dentro de una función y cerrarlo fuera de la declaración de la función, no poner un punto y coma al final de cada sentencia o recibir un token en un momento que no debería ser posible.

Hay dos tipos de análisis sintáctico, aquel que construye el árbol desde las hojas hasta la raíz (ascendente) y el que construye el árbol desde la raíz hasta las hojas (descendente). Ambos se basan en una gramática que debe ser o bien LL para los analizadores descendentes o LR para los analizadores ascendentes para poder utilizar las producciones.

2.2.3 Análisis Semántico

La fase de análisis semántico de un programa consiste en hacer comprobaciones semánticas para asegurar la consistencia del archivo fuente. También introduce información en la tabla de símbolos y puede detectar errores semánticos en el programa [17].

Para poder realizar el análisis semántico de un programa de un lenguaje se define un Esquema de Traducción, que incluye unas acciones llamadas acciones semánticas que se sitúan entre los símbolos de las reglas sintácticas. Estas acciones semánticas hacen uso de atributos de los símbolos gramaticales [17].

El analizador semántico puede detectar errores como asignaciones de tipo erróneas (intentar asignar un valor de tipo entero a una variable de tipo carácter), o detectar un uso incorrecto de las operaciones definidas por el lenguaje (sumar un entero con una cadena), o detectar fallos en las funciones (llamar a una función con un número de parámetros erróneo o devolver un tipo diferente al definido por la función).

2.2.4 Tabla de Símbolos

La tabla de símbolos es un módulo del compilador que se encarga de almacenar la información relativa a los nombres de variables y funciones (identificadores) del programa. Durante el análisis léxico se introducen los lexemas y durante el análisis semántico se introduce la información relativa al tipo de las variables y su desplazamiento, la cantidad de parámetros de una función y su tipo de retorno y de las constantes se almacena su tipo, etc.

La tabla de símbolos tiene la capacidad de detectar el entorno en el que se declara una variable, es decir, si está declarada una variable como global y otra como local, al recibir una referencia a la tabla de símbolos debe ser capaz de reconocer a qué variable se refiere [17].

Las operaciones que debe tener el módulo de la tabla de símbolos para que sea funcional son ser capaz de añadir, modificar y destruir entradas, devolver o modificar la información relativa a un identificador y crear y destruir la tabla de símbolos activa.

2.2.5 Generador de código intermedio

Este módulo del compilador cumple con la función de generar código en un lenguaje intermedio para poder traducirlo posteriormente al ensamblador específico de cada lenguaje [17].

El objetivo de este módulo es usar toda la información recopilada en los anteriores módulos para traducirla a un código que se asemeje más al código ensamblador.

El generador de código intermedio se encarga de generar variables temporales que se usan en las operaciones o en las expresiones de un lenguaje, emitir etiquetas a las que otras operaciones pueden saltar, facilitando así la creación de funciones o condicionales y de diseñar el Registro de Activación de una función [17].

El Registro de Activación de una función consiste en una zona de memoria donde se guardarán los valores para los parámetros, las variables temporales y locales de la función y para la información del estado del procesador antes de la llamada a la función, entre otras cosas.

El código intermedio puede ser representado de varias formas, por ejemplo, se puede representar usando cuartetos estructurados de la siguiente manera:

```
(operador, operando 1, operando 2, resultado)
```

En este ejemplo, el campo operador se refiere a la operación que se va a usar en la instrucción, los operandos se refieren a los elementos que se van a utilizar en la operación, que pueden ser variables o constantes y el resultado es la posición de memoria en la que se almacena el resultado de la operación.

Según las especificaciones del lenguaje, puede haber cuartetos en los que alguno de los operandos no se utilice, o incluso que no se utilice ninguno de los operandos o solo se emita una etiqueta.

2.2.6 Generador de código objeto

El generador de código objeto se encarga de convertir el código intermedio a código ensamblador. Esto se hace mediante una plantilla de traducción en la que para cada cuarteto se crea un código equivalente en el ensamblador de las máquinas en las que el lenguaje se quiera ejecutar [17].

2.3 DRACO

DRACO (Dinámica de Refuerzo del Aprendizaje de Compiladores) es una plataforma online que tiene como objetivo gamificar la enseñanza de las asignaturas de Traductores de Lenguajes y Procesadores de Lenguajes impartidas en la Escuela Técnica Superior de Ingenieros Informáticos de la Universidad Politécnica de Madrid.

DRACO ofrece diferentes herramientas a los profesores para apoyarse durante el curso: tiene la capacidad de corregir los distintos módulos de las prácticas de Procesadores de Lenguajes, poner ejercicios interactivos para ayudar a los

alumnos a repasar la asignatura, tiene incorporado un sistema de clasificación por puntos que se consiguen resolviendo actividades, ya sea corregir un módulo de la práctica o resolver los ejercicios interactivos.

La interfaz de la plataforma está dividida en dos módulos, aquel al que tiene acceso el profesor, con opciones de administración y visualización de datos, y la interfaz del alumno, que consiste principalmente en menús para visualizar los datos individuales, para acceder a las actividades y para acceder a la clasificación de puntos entre los alumnos.

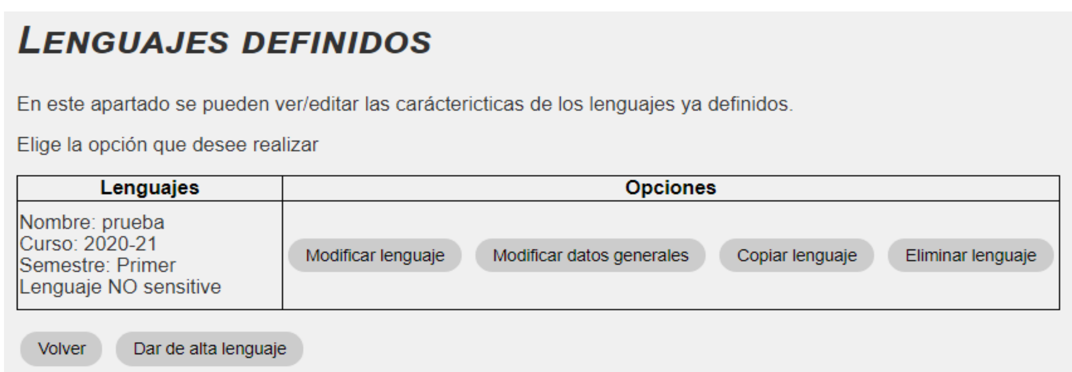

Ilustración 9: Interfaz del profesor para la administración de los lenguajes disponibles

Las prácticas de ambas asignaturas giran alrededor de un lenguaje definido al principio del curso y para que DRACO pueda ejecutar los comprobadores de las prácticas, el profesor responsable del sistema debe introducir la información necesaria acerca del lenguaje. En la ilustración 9, se muestra la interfaz que permite modificar los lenguajes definidos. Entre otras opciones, se deben rellenar las palabras reservadas que tenga el lenguaje, los operadores y su precedencia, la estructura que tienen los comentarios, las constantes y los identificadores, los tipos disponibles del lenguaje, la estructura que tienen las sentencias complejas como bucles y condicionales, cómo se declaran los identificadores y las funciones, cuáles son los errores que pueden surgir en los diferentes módulos y se debe especificar también que tokens debe implementar el alumno de forma obligatoria u opcional.

3 Planteamiento del problema

3.1 Introducción

En este apartado se realizará una especificación de requisitos software para las mejoras de la plataforma DRACO que se van a implementar en este proyecto. Esta especificación se desarrolla basándose en el estándar ISO/IEC/IEEE 29148 [18].

3.1.1 Propósito

El propósito de este apartado es definir los requisitos de las ampliaciones realizadas en el sistema DRACO. Al finalizar este Trabajo, la mayoría de los requisitos especificados serán implementados en el sistema, permitiendo al alumnado corregir automáticamente las prácticas de la asignatura de TDL.

3.1.2 Ámbito del sistema

Con este trabajo se busca añadir al sistema la posibilidad de corregir también las prácticas de TDL. Para ello se deberá implementar un analizador genérico que implemente los módulos de análisis léxico, tabla de símbolos, análisis sintáctico y análisis semántico. La parte de síntesis del sistema será abordada por otro Trabajo de Fin de Máster, ya que la carga de trabajo del sistema completo es muy alta y ha habido que separarla en tareas más pequeñas. La parte de la interacción con el alumno y la configuración de los comprobadores por el profesor será desarrollada por otro Trabajo de Fin de Grado.

Para poder realizar este sistema se parte de una serie de páginas de configuración que permiten definir las especificaciones del lenguaje. Sin embargo, la información disponible actualmente es insuficiente para poder crear un analizador genérico.

3.1.3 Definiciones y abreviaturas

En este apartado se presentan los conceptos necesarios para poder comprender la especificación de requisitos.

3.1.3.1 Definiciones

Término	**Definición**
Axioma	El símbolo con el que comienza el procesado del lenguaje
Compilador	Programa que tiene como objetivo devolver un fichero con código ensamblador o máquina equivalente a un fichero fuente. Está formado por seis módulos: el análisis léxico, el análisis sintáctico, el análisis semántico, el generador de código intermedio, el generador de código objeto y la tabla de símbolos.
DRACO	Sistema de gamificación para las asignaturas de Procesadores de Lenguajes y Traductores de Lenguajes
Gramática	Define la estructura de un lenguaje. Está formado por símbolos terminales, símbolos no terminales, el axioma y las reglas de producción. En un compilador se usan gramáticas de tipo 2 y de tipo 3 [19]

Término	**Definición**
Ítem	Una regla que está siendo procesada por el analizador sintáctico, y que tiene un atributo extra que indica por qué símbolo va el procesado de la regla
Parse	Serie de reglas utilizadas durante el procesado del fichero
Profesor	Docente que se encarga de impartir las asignaturas de Procesadores de Lenguajes y Traductores de Lenguajes y que actúa también como administrador de DRACO
Regla de producción, regla, producción	Define cómo se forman los distintos elementos de un lenguaje.
Símbolos no terminales	Conjunto de símbolos que derivan en otros símbolos no terminales o terminales
Símbolos terminales	Conjunto de símbolos que representan elementos finales del lenguaje
Token	Unidad mínima de información que tiene el lenguaje para separar los elementos léxicos del mismo

3.1.3.2 Abreviaturas

Término	**Definición**
CSS	Cascading Style Sheets
DRACO	Dinámica de Refuerzo del Aprendizaje de Compiladores
HTML	HyperText Markup Language
PDL	Procesadores de Lenguajes
PHP	Hypertext Preprocessor
TDL	Traductores de Lenguajes

3.2 Descripción general

Esta sección provee una descripción general de los cambios y las mejoras a añadir en el sistema.

3.2.1 Perspectiva del producto

El sistema se ampliará en el módulo de corrección de prácticas de DRACO, y se añadirá la funcionalidad de poder corregir las prácticas de TDL.

3.2.2 Funciones del sistema

Las funciones se pueden dividir en cinco módulos:

- Analizador léxico
- Analizador sintáctico
- Analizador semántico

- Tabla de símbolos
- Configuración del lenguaje

En los siguientes apartados, se explicarán estos módulos y su funcionamiento dentro del sistema.

3.2.2.1 Analizador léxico

El primer módulo del compilador genérico es el análisis léxico, que sacará información de las páginas de configuración del lenguaje. Este módulo deberá poder recibir un fichero fuente e ir analizándolo extrayendo los *tokens* que haya definido el lenguaje.

3.2.2.2 Analizador sintáctico

El segundo módulo del compilador genérico deberá poder construir un árbol sintáctico a partir de la gramática y el fichero fuente. El Analizador sintáctico devolverá el *parse*.

3.2.2.3 Analizador semántico

El tercer módulo del compilador genérico deberá poder realizar comprobaciones para confirmar la consistencia del significado del programa. Principalmente se deberán extraer la información sobre los identificadores del programa y la consistencia de las operaciones.

La información obtenida acerca de los identificadores será almacenada en la Tabla de símbolos.

3.2.2.4 Tabla de símbolos

La Tabla de símbolos es un módulo del compilador que almacena información relativa a los identificadores, ya sean una función o una variable. La Tabla de símbolos está compuesta por una entrada por cada identificador. Además, el sistema deberá poder gestionar varias tablas de símbolos; entonces, se deberá poder crear y destruir una tabla de símbolos. Habrá una Tabla de símbolos global que almacene la información de las funciones y las variables globales más otra tabla local que almacene información sobre los identificadores locales.

3.2.2.5 Configuración del lenguaje

Para poder realizar el compilador genérico se deberá ampliar la información acerca del lenguaje que se recoge en el sistema, ya que la que está disponible actualmente es insuficiente para configurar los nuevos módulos.

3.3 Requisitos específicos

A continuación, se presentan todos los requisitos que deberá tener implementado el sistema cuando se finalice el módulo de análisis.

3.3.1 Requisitos funcionales

En primer lugar, se presentan los requisitos funcionales separados en los cinco grupos identificados en la descripción general.

3.3.1.1 Analizador léxico

Requisito 1.1: El Analizador léxico deberá poder recibir la información del lenguaje que está guardada en la base de datos y deberá almacenar esta información de manera que el programa pueda aplicar un filtro para cada tipo de *token* definido por el lenguaje.

Requisito 1.2: El Analizador léxico recibe como parámetro el fichero fuente a leer.

Requisito 1.3: El Analizador léxico tendrá una función que itera sobre el fichero recibido por parámetro hasta que encuentre un *token*. Entonces, devolverá el *token* encontrado, indicando su código y su atributo.

Requisito 1.4: Si el Analizador léxico encuentra un error o un elemento que no encaja con ninguna de las descripciones dadas, notificará un error y en las próximas llamadas al analizador, se continuará desde el último carácter leído.

Requisito 1.5: El Analizador léxico deberá poder reconocer cuándo comienza un comentario en el fichero dada la estructura que tenga un comentario en la definición del lenguaje.

Requisito 1.6: El Analizador léxico deberá poder reconocer si el salto de línea es un *token* dentro del lenguaje y emitir el *token* en el caso de que si lo sea.

Requisito 1.7: El Analizador léxico deberá poder reconocer los *tokens* del tipo cadena según como esté definida su notación dentro del lenguaje.

Requisito 1.8: El Analizador léxico deberá poder reconocer los *tokens* del tipo carácter según como estén definidos dentro de la configuración del lenguaje. La estructura de estos *tokens* es igual a los tokens definidos en el requisito 1.7

Requisito 1.9: El Analizador léxico deberá poder reconocer los distintos formatos de números definidos por el lenguaje. Un número viene definido por tres parámetros, el conjunto de caracteres con los que puede comenzar el número, el conjunto de caracteres con los que puede terminar el número y el conjunto de caracteres que puede ir en medio del número.

Requisito 1.10: El Analizador léxico deberá emitir los *tokens* numéricos de tal manera que el atributo incluya el valor del número en decimal, independientemente de la base que tenga el número. Esto se hará recibiendo de la base de datos de DRACO el valor que corresponde a cada símbolo que no sea decimal.

Requisito 1.11: El Analizador léxico deberá poder reconocer los *tokens* que equivalgan a cualquier símbolo declarado por el lenguaje. Estos símbolos pueden estar formados por más de un carácter. Estos símbolos pueden ser operadores, signos de puntuación, etc.

Requisito 1.12: El Analizador léxico deberá poder reconocer los *tokens* que representan palabras reservadas del lenguaje. Estas palabras reservadas pueden estar compuestas de cualquier símbolo que no sea un espacio, según estén definidas en la configuración del lenguaje.

Requisito 1.13: El Analizador léxico deberá poder reconocer los *tokens* que representan identificadores en el lenguaje. La estructura de un identificador está definida en las páginas de configuración del sistema de la misma manera en la que están definidos los *tokens* del requisito 1.9.

Requisito 1.14: El Analizador léxico tendrá acceso a la Tabla de símbolos activa, en la que irá metiendo los identificadores nuevos que encuentre.

3.3.1.2 Tabla de símbolos

Requisito 2.1: La Tabla de símbolos deberá estar compuesta de entradas, una por cada identificador.

Requisito 2.2: Una Tabla de símbolos podrá crearse.

Requisito 2.3: Una Tabla de símbolos podrá destruirse.

Requisito 2.4: Las entradas de la Tabla de símbolos podrán tener los siguientes campos: nombre del identificador, desplazamiento, tipo, etiqueta, parámetros, tipo de los parámetros y valor de retorno. Cada entrada no tiene por qué tener información en todos los campos.

Requisito 2.5: Las entradas de la Tabla de símbolos se podrán crear y modificar.

Requisito 2.6: La información de las entradas de la Tabla de símbolos se pueden consultar o modificar accediendo directamente a cada entrada.

Requisito 2.7: Se puede consultar un identificador de la Tabla de símbolos; si se encuentra, se devuelve la entrada correspondiente al identificador.

3.3.1.3 Analizador sintáctico

Requisito 3.1: El Analizador sintáctico recibe como entrada la lista de *tokens* generado por el Analizador léxico.

Requisito 3.2: El Analizador sintáctico deberá tener acceso al analizador léxico, al que irá llamando cuando necesite un *token*.

Requisito 3.3: El Analizador sintáctico deberá usar la estrategia de análisis sintáctico ascendente LR. Por tanto, deberá crear las tablas de acción y *goto* a partir de la gramática sintáctica del lenguaje.

Requisito 3.4: El Analizador sintáctico deberá implementar las funciones *first*, *follow* y cierre de un conjunto de ítems, *goto* de un conjunto de ítems dado un símbolo y la creación de la colección canónica.

Requisito 3.5: El Analizador sintáctico deberá imprimir mensajes de error y seguir con el análisis cuando encuentre un *token* que no encaje con la estructura sintáctica del lenguaje.

Requisito 3.6: El Analizador sintáctico deberá devolver la lista de reglas usadas en la creación del árbol sintáctico teórico. Esto significa que cada vez que se reduzca por una regla, se notificará al analizador semántico y generador de código intermedio para que se ejecuten sus acciones semánticas.

3.3.1.4 Analizador semántico

Requisito 4.1: El Analizador semántico debe recibir las reglas utilizadas por el Analizador sintáctico.

Requisito 4.2: El Analizador semántico debe ejecutar las acciones semánticas que están asociadas a cada regla.

Requisito 4.3: El Analizador semántico debe poder acceder a la Tabla de símbolos para consultar la información de un identificador.

Requisito 4.4: El Analizador semántico debe poder modificar la información de un identificador en la Tabla de símbolos.

Requisito 4.5: En el caso de que el Analizador semántico encuentre un error durante el análisis, debe notificar el error y continuar ejecutándose.

3.3.1.5 Configuración del lenguaje

Requisito 5.1: Se deberá crear en la configuración del lenguaje una página para añadir el valor de los símbolos no decimales para cada plantilla de *token* numérico.

Requisito 5.2: Por cada página creada se deberá añadir una entrada en el índice usado para navegar por las páginas de configuración.

Requisito 5.3: Se deberá modificar la página de la definición del lenguaje para completar la información necesaria para reconocer la forma de los *tokens* numéricos e identificadores.

Requisito 5.4: Se deberá añadir una página para subir la gramática del lenguaje activo para poder enviarlo al módulo de análisis sintáctico.

Requisito 5.5: Se deberá añadir una página en el que se muestren todas las reglas de la gramática para dar información acerca del tipo que son (Sentencias de control, operadores, funciones, declaraciones y operandos).

Requisito 5.6: Para cada sentencia de control se deberá especificar cuál es su comportamiento para poder generar las reglas semánticas correspondientes a la regla.

Requisito 5.7: Para cada regla que contenga un operador se deberá poder especificar cuáles son los tipos de datos de los operandos de un operador usando una página de la configuración del lenguaje para las reglas que correspondan a una operación.

Requisito 5.8: Para cada regla que contenga un operador se deberá poder especificar el tipo de salida que produce la operación, esto se hará usando una página de la configuración del lenguaje para las reglas que correspondan a una operación.

Requisito 5.9: Para cada regla que corresponda a la declaración de una función se deberá especificar qué campos son el identificador de la función, qué campos corresponden a los parámetros de la función y qué partes de la regla son el cuerpo de la función. Esto se realizará en una página de la configuración del lenguaje para las reglas que correspondan a la declaración de una función.

Requisito 5.10: Para cada regla que corresponda a una declaración se deberá especificar el símbolo que corresponde al tipo del identificador que se está declarando (si las variables solo admiten un tipo de datos); esto se hará en una página de la configuración que corresponderá a las reglas que sean declaraciones.

3.3.2 Requisitos de interfaces externas

3.3.2.1 Interfaz de subida de prácticas

El sistema recibirá las solicitudes de procesamiento desde una interfaz que verán los alumnos para poder corregir sus prácticas, para acceder a esta página será necesario un navegador ya que el sistema estará alojado en un servidor en línea. Esta parte es objeto de otro TFG.

3.3.2.2 Interfaz de configuración de las actividades de DRACO

Para que el profesor pueda configurar el valor que tienen las correcciones, cuando se abre la actividad que se encargará de corregir las prácticas y los ficheros de ejemplo que serán utilizados en las correcciones será necesaria una página de configuración de actividades, en la que se podrá activar la actividad de corrección de la práctica de TDL. Esta parte es objeto de otro TFG.

3.3.2.3 Base de datos de DRACO

El sistema recogerá la información de la especificación del lenguaje mediante una conexión a la base de datos de DRACO. Será imprescindible por lo tanto el acceso a internet para poder modificar el lenguaje.

4 Solución

4.1 Diseño

En este apartado se explicará el diseño de las funciones del sistema. Para ayudar a entender cómo se ha desarrollado el sistema, a continuación, se explicará la estructura de la base de datos usada por el sistema y el contenido de las diferentes tablas. En la ilustración 10 se representan las relaciones entre las tablas de la base de datos *draco_lenguaje*.

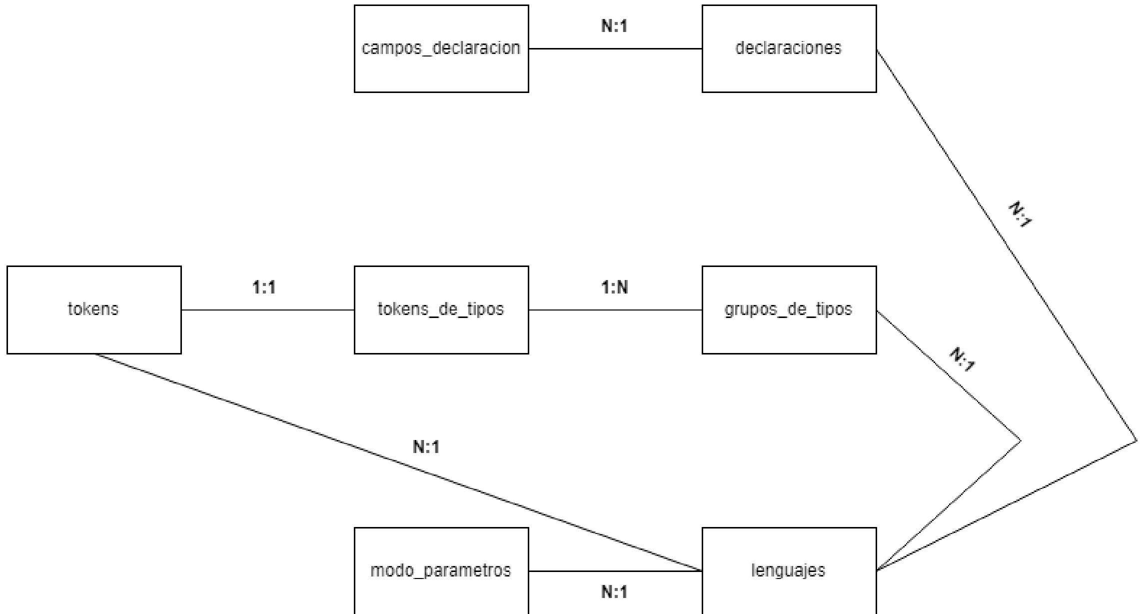

Ilustración 10: Estructura de la base de datos *draco_lenguaje* y las relaciones entre sus tablas.

La tabla *lenguajes* representa los lenguajes definidos por el sistema, de los cuales sólo puede haber uno activo.

Campo	**Descripción**
id_lenguaje	Identificador del lenguaje
nombre	Nombre del lenguaje
year	Año de creación del lenguaje
asignatura	Asignatura en la que se va a usar el lenguaje; puede ser PDL o TDL
semestre	Semestre en el que se va a usar el lenguaje
esSensitive	Valor que indica si el lenguaje es *case sensitive*
dec_global_eq_local	Valor que indica si los identificadores globales se pueden declarar de la misma manera que los locales
undec_ids	Valor que indica si el lenguaje permite identificadores no declarados

Campo	**Descripción**
undec_ids_ts	Valor que indica si los identificadores no declarados actúan como variables globales o locales
undec_ids_type	Valor que indica el tipo de los identificadores no declarados
param_pass_mode	Valor que indica si el lenguaje tiene varios modos de paso de parámetros

La tabla *modo_parámetros* almacena los distintos modos de paso de parámetros.

Campo	**Descripción**
param_mode_id	Identificador del modo de paso de parámetros
language_id	Identificador del lenguaje
mode_name	Nombre del modo de paso de parámetros
token_id	Identificador del *token* asociado al modo de paso de parámetros

La tabla *grupos_de_tipos* almacena los grupos de los tipos de datos existentes en el lenguaje.

Campo	**Descripción**
language_id	Identificador del lenguaje
dtg_id	Identificador del grupo de datos
dtt_id	Identificador del *token* de tipo de datos
dtg_desc	Descripción del grupo de tipos de datos

La tabla *tokens_de_tipos* almacena los *tokens* que representan algún tipo de dato en el lenguaje.

Campo	**Descripción**
dtt_id	Identificador del tipo de datos
dtt_desc	Descripción del tipo de datos
token_id	Identificador del *token* asociado al tipo de datos
despl	Desplazamiento o tamaño del tipo de datos

La tabla *tokens* almacena la información relativa a todos los *tokens* del lenguaje.

Campo	**Descripción**
idToken	Identificador del *token*
id_lenguaje	Identificador del lenguaje al que pertenece el *token*
lexema	Lexema que representa al *token*
codigoToken	Código del *token*
nombreTokenAlum	Nombre del *token*
tipo	Tipo del *token*
cardinalidad	Cardinalidad del *token* si es un operador
asociatividad	Asociatividad del *token* si es un operador
prioridad	Prioridad del *token* si es un operador
preOperador	Valor que indica si el *token* es un pre-operador
esOperando	Valor que indica si el *token* es un operando
esOperador	Valor que indica si el *token* es un operador
esOtros	Valor que indica si el *token* pertenece al grupo de *tokens* Otros
esFinSentencia	Valor que indica si el *token* representa el final de sentencia
estructura	Estructuras a las que está asociado el *token*
enInteriorEstructura	Valor que indica si el *token* está en el interior de una estructura
aContinuacionEstructura	Valor que indica si el *token* está a continuación de una estructura
tieneCondicion	Valor que indica si la estructura que representa el *token* tiene campo condición
combina	Valor que indica si los *tokens* de inicio y fin de la estructura representada por el *token* son variables
tokInicioEstructura	*Token* que representa el inicio de la estructura representada por el *token*

Campo	**Descripción**
tokFinEstructura	*Token* que representa el final de la estructura representada por el *token*
pareja	Valor que indica si la estructura representada por el token tiene una pareja adicional de *tokens* de inicio y fin de estructura
tokInicioEstructura1	*Token* adicional que representa el inicio de la estructura representada por el *token*
tokFinEstructura1	*Token* adicional que representa el final de la estructura representada por el *token*
esObligatorio	Valor que indica si los alumnos deben implementar este *token* obligatoriamente
esOpcional	Valor que indica si los alumnos no deben implementar este *token* obligatoriamente
opc1ObligatoriasGrupos	Valor de texto que indica a qué grupo de *tokens* que tienen que implementar los alumnos de forma obligatoria pertenece este *token*
opc2ObligatoriasGrupos	Valor de texto que indica a qué grupo de *tokens* que tienen que implementar los alumnos de forma obligatoria pertenece este *token*
opc3ObligatoriasGrupos	Valor de texto que indica a qué grupo de *tokens* que tienen que implementar los alumnos de forma obligatoria pertenece este token
opc1OpcionalesGrupos	Valor de texto que indica a qué grupo de *tokens* que tienen que implementar los alumnos de forma opcional pertenece este *token*
opc2OpcionalesGrupos	Valor de texto que indica a qué grupo de *tokens* que tienen que implementar los alumnos de forma opcional pertenece este *token*
opc3OpcionalesGrupos	Valor de texto que indica a qué grupo de *tokens* que tienen que implementar los alumnos de forma opcional pertenece este *token*
yaAsignada	Valor que indica a qué agrupación de *tokens* pertenece este *token*

Campo	Descripción
especificoPorGrupos	Valor que indica a qué grupo de *tokens* pertenece el *token*, este tipo de grupo representa la parte específica de la práctica de cada grupo
asignacion	Sin uso
atributoVariable	Valor que indica si el *token* tiene un atributo variable
atributoFijo	Valor que indica si el atributo del *token* tiene un valor fijo

La tabla *declaraciones* almacena las diferentes maneras de declarar funciones, parámetros y variables.

Campo	Descripción
language_id	Identificador del lenguaje
dec_id	Identificador de la declaración
dec_type	Tipo de declaración
name	Nombre de la declaración
ref	Referencia de la declaración usada por la tabla *campos_declaracion* para relacionar *tokens* y declaraciones

La tabla *campos_declaracion* almacena los campos que tiene cada declaración de la tabla *declaraciones*.

Campo	Descripción
dfield_id	Identificador del campo de declaración
elem_id	Identificador del *token* que representa el campo de la declaración
elem_type	Tipo del campo de la declaración
repeated	Valor que indica si el campo se repite
optional	Valor que indica si el campo es opcional en la declaración
orden	Valor que indica el orden en el que van los campos de una declaración
func_marker	Sin uso

Campo	Descripción
dec_ref	Referencia a la declaración a la que pertenece el campo de declaración

4.1.1 Analizador léxico

En esta sección se explicará qué decisiones de diseño se han tomado para poder implementar el analizador léxico. Se explicará cómo se han cumplido los requisitos del 1.1 al 1.14 para implementar el algoritmo del analizador léxico, explicando el flujo de ejecución del sistema.

4.1.1.1 Algoritmo del Analizador léxico

El algoritmo del Analizador léxico se compone de tres fases. Primero se inicializa el analizador con el fichero a analizar y se consulta la base de datos para recibir la estructura de todos los *tokens* a reconocer. Después de la inicialización se puede llamar al Analizador léxico hasta que no queden *tokens* por reconocer en el fichero fuente. Por último, cuando se llega al final del archivo, se envía el token EOF para indicar que se ha terminado de analizar el fichero. El algoritmo representado en la ilustración 11 cumple los requisitos 1.1, 1.2 y 1.3.

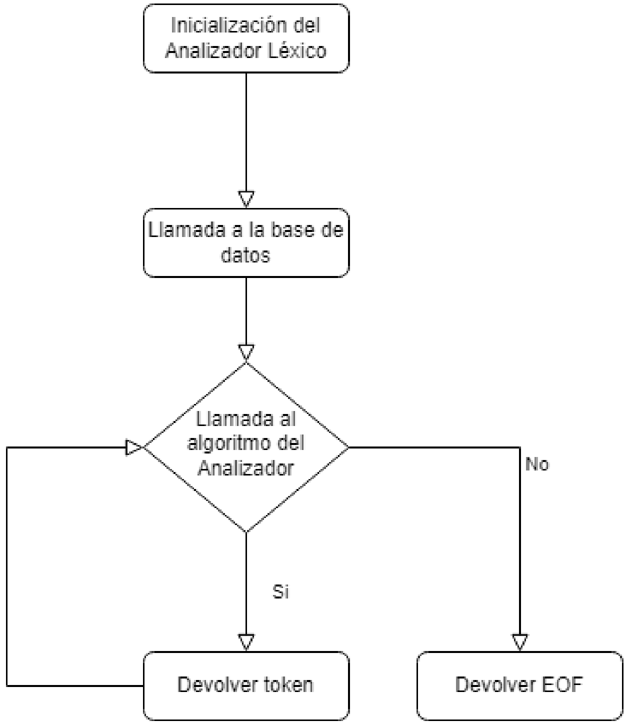

Ilustración 11: Algoritmo del Analizador léxico.

4.1.1.2 Algoritmo de Análisis del fichero y devolución de tokens

Para analizar el fichero y devolver los *tokens*, se recogen una serie de caracteres del fichero y se compara con los tipos de *token* que puede ser. Se hace de manera ordenada para que no haya conflictos y se reconozcan todos los *tokens* bien. Las fases en las que se divide el análisis son las siguientes:

- Se comprueba si es un comentario; en el caso de que lo sea, se salta el comentario y se continúa la ejecución.
- Se comprueba si es una cadena; en el caso de que lo sea, se devuelve el *token string* y se termina la ejecución.

- Se comprueba si es un *token* de tipo carácter; en el caso de que lo sea, se devuelve el *token* carácter y se termina la ejecución.
- Se comprueba si es un *token* numérico; en el caso de que lo sea, se emite el *token* número con el valor en decimal como atributo y se termina la ejecución.
- Se comprueba si es un símbolo; en el caso de que lo sea, se emite el *token* correspondiente al símbolo y se termina la ejecución.
- Se comprueba si es una palabra reservada; en el caso de que lo sea, se emite el *token* correspondiente a la palabra reservada y se termina la ejecución.
- Se comprueba si es un identificador; en el caso de que lo sea, se devuelve el *token* identificador tras insertar el identificador en la Tabla de símbolos; después, se termina la ejecución.
- Si está activada la opción de salto de línea en el lenguaje, en el caso de que se encuentre el *token* salto de línea se emite y se termina la ejecución.
- Si se encuentra un error, se saltan los caracteres que han dado lugar al error, se notifica y se continua la ejecución hasta enviar un *token*.

Este algoritmo de análisis cumple los requisitos del 1.4 al 1.14.

4.1.1.3 Reconocimiento de comentarios

Los comentarios se definen dentro de DRACO en la interfaz de definición del lenguaje. Los comentarios están formados por un inicio de comentario y un final de comentario. Para reconocerlo, se comprueba si los caracteres leídos son iguales a los del inicio del comentario; después se debe leer el cuerpo del comentario, comprobando siempre si el carácter que se está leyendo es igual al fin de comentario. Cuando se termine de leer el comentario se continúa la ejecución del algoritmo sin devolver ningún *token*. Con el algoritmo representado en la ilustración 12 se cumple el requisito 1.5.

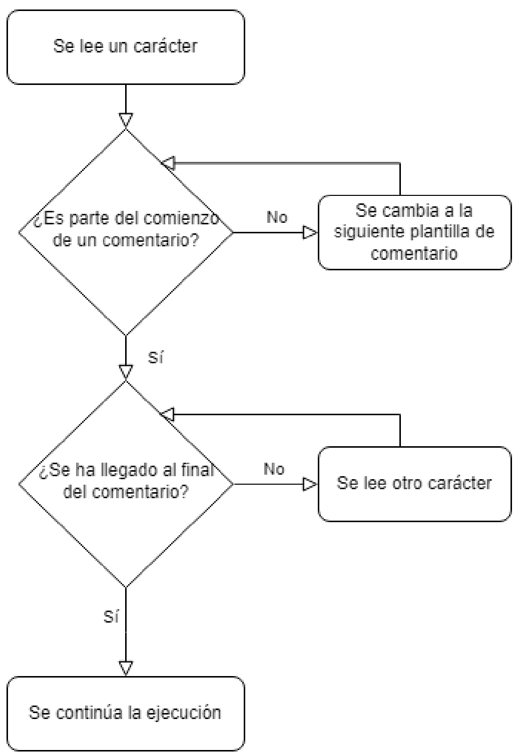

Ilustración 12: Diagrama de flujo para el reconocimiento de los comentarios.

4.1.1.4 Reconocimiento de cadenas y de caracteres

La estructura de las cadenas y los caracteres se definen en la interfaz de DRACO en la página de definición del lenguaje, cuyo apartado está representado en la ilustración 13. Como comienzan de la misma forma que terminan, se define el carácter de comienzo y solo pueden comenzar por un carácter; para reconocer ambos *tokens* se debe comprobar que el carácter leído coincide con el de comienzo de la cadena o el carácter. Este algoritmo, representado en la ilustración 14, cumple el requisito 1.7 y 1.8.

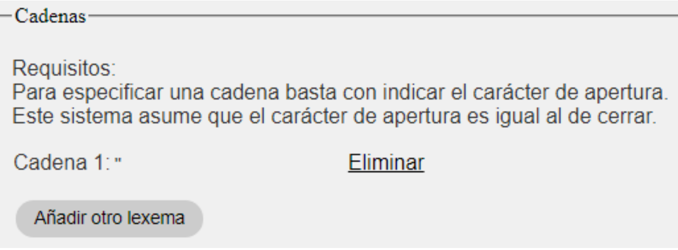

Ilustración 13: Interfaz de DRACO de la declaración de las cadenas.

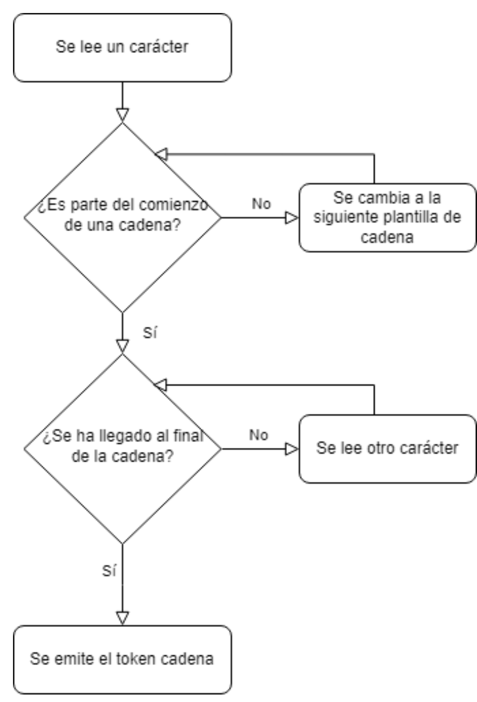

Ilustración 14: Diagrama de flujo del reconocimiento de cadenas.

4.1.1.5 Reconocimiento de números

La estructura de los *tokens* numéricos se define en la interfaz de DRACO en la página de definición del lenguaje. Su estructura se divide en tres partes, el inicio, el medio y el final. La parte del medio se puede repetir indefinidamente. Para reconocer un *token* numérico, debe coincidir la primera parte del número con el carácter leído; después se deben repetir los caracteres del medio del *token* hasta que se llega a los caracteres del final, o en el caso de que coincidan los del medio con los del final, hasta que se llega a un carácter que no está representado en la estructura del *token*. Durante el diseño del algoritmo que reconoce los *tokens* numéricos específicos del lenguaje activo se observó que los campos que había disponibles eran insuficientes, ya que según el tipo de número que se diseñase, por ejemplo el hexadecimal, si introducías la cadena 0x la reconocería como un número. Se concluyó que había que añadir un campo más a la base de datos para indicar si el campo del medio de la estructura del

token es obligatorio y añadir la posibilidad de indicar en la definición del lenguaje esta opción.

El *token* tendría que devolverse usando el valor del número en decimal, por lo tanto, se tendría que convertir el número estando en la base que sea a decimal. Se decidió que tendría que añadirse una página adicional para añadir el valor que tendrá cada dígito según la base.

La ilustración 15, diagrama de flujo del reconocimiento de tokens numéricos, representa el algoritmo utilizado para poder reconocer la estructura definida por la página de DRACO de definición del lenguaje.

Con estos cambios se cumplirían los requisitos 1.9 y 1.10.

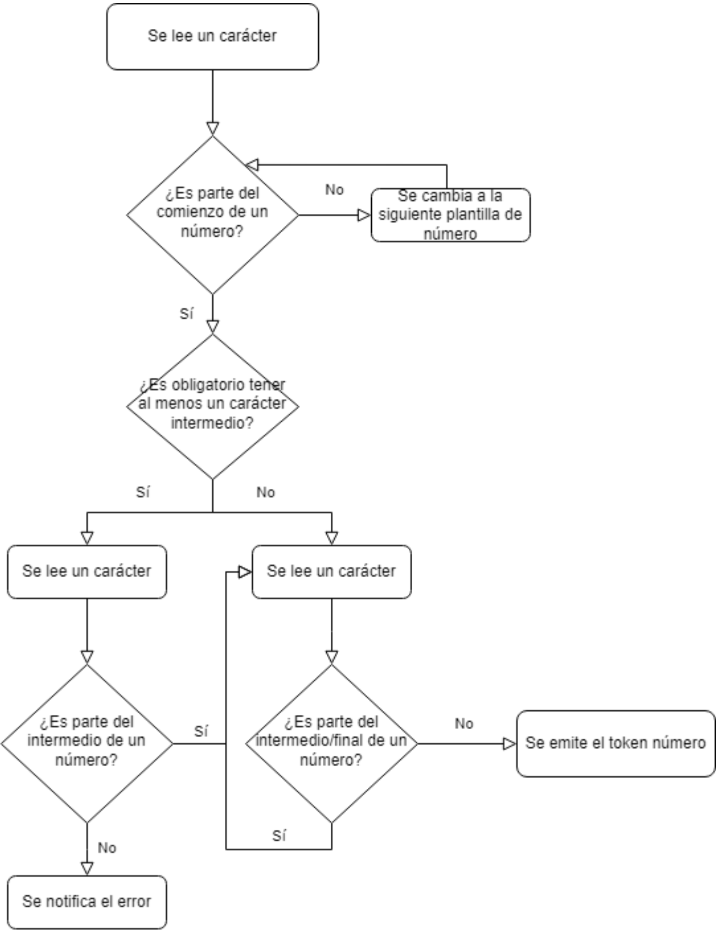

Ilustración 15: Diagrama de flujo del reconocimiento de tokens numéricos.

4.1.1.6 Reconocimiento de símbolos

Para reconocer los símbolos definidos por el lenguaje primero habría que sacar de la base de datos los *tokens* que representen símbolos. Los símbolos pueden tener diferentes tamaños y pueden estar compuestos de cualquier carácter, por lo tanto, el algoritmo usado para reconocer símbolos va reconociendo caracteres del fichero fuente y los compara con los diferentes símbolos que hay en el sistema, eliminando los que no son parecidos a la cadena leída; repitiendo este paso, se llega al símbolo del lenguaje más grande que está representado en el fichero fuente. Este algoritmo cumpliría el requisito 1.11.

A continuación, en la ilustración 16, se muestra una representación del algoritmo utilizado para reconocer símbolos del lenguaje.

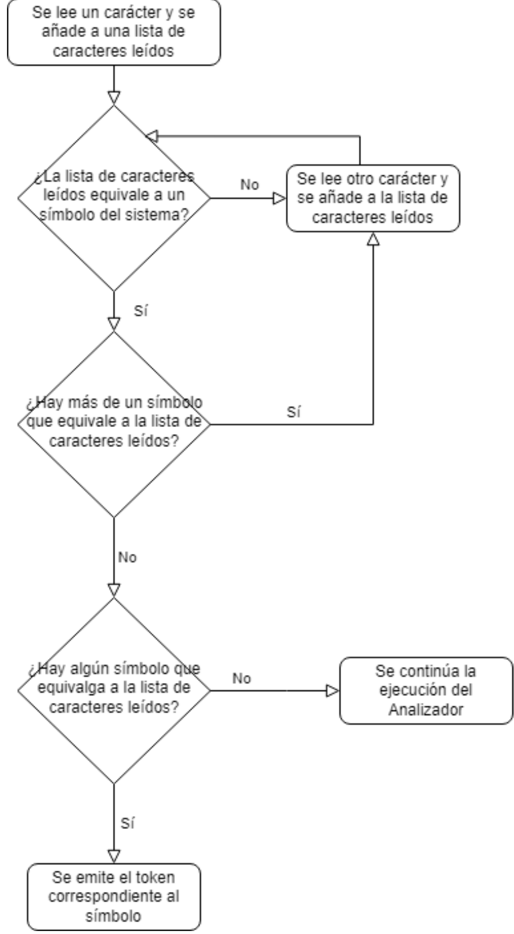

Ilustración 16: Diagrama de flujo del reconocimiento de los símbolos.

4.1.1.7 Reconocimiento de palabras reservadas

Las palabras reservadas se definen en la página de definición del lenguaje de DRACO. para reconocer los *tokens* del tipo palabra reservada se recogerán todos los caracteres que hay desde del último *token* leído hasta el último delimitador y se compararán con las palabras reservadas del lenguaje. Si la cadena leída coincide con alguna palabra reservada del lenguaje se emitirá el *token* correspondiente. Con este algoritmo se cumplirá el requisito 1.12. En la ilustración 17 se muestra la parte de la interfaz de definición del lenguaje de DRACO usada para definir las palabras reservadas del lenguaje.

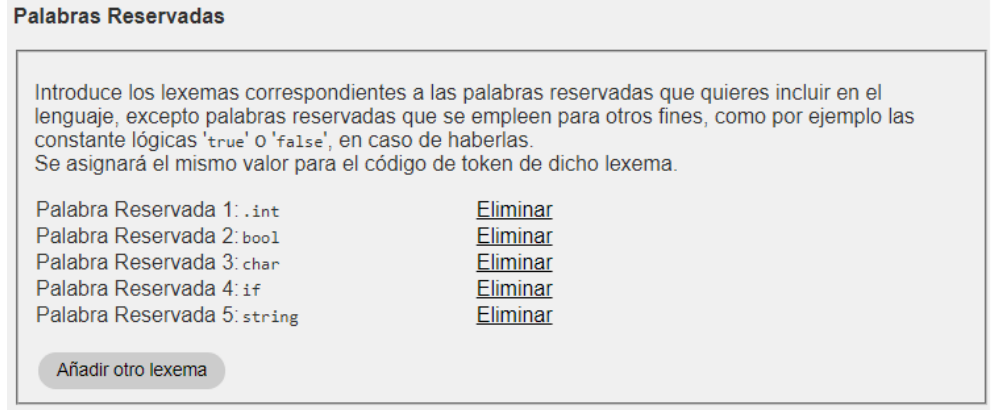

Ilustración 17: Interfaz de DRACO de la declaración de las palabras reservadas.

4.1.1.8 Reconocimiento de identificadores

Los identificadores tienen la misma estructura que los números, con la diferencia de que solo puede haber un tipo de identificador en el lenguaje. Para reconocerlos se aplicará el mismo algoritmo que el usado para los números, incluyendo el campo de si es obligatorio usar la parte del medio de la estructura, ya que si el lenguaje tuviera *tokens* identificadores que comenzasen por ejemplo por “$” no sería válido un identificador que fuera solo un “$”.

Cuando se emita un *token* del tipo identificador, además se debe añadir a la Tabla de símbolos activa si no está presente en la tabla; si está presente en la Tabla de símbolos, se emitirá el mismo *token* que se emitió la primera vez que se reconoció el mismo identificador. Con este algoritmo se cumplirían los requisitos 1.13 y 1.14. A continuación se muestra en la ilustración 18 el diagrama de flujo que usará el programa para reconocer *tokens* de tipo identificador.

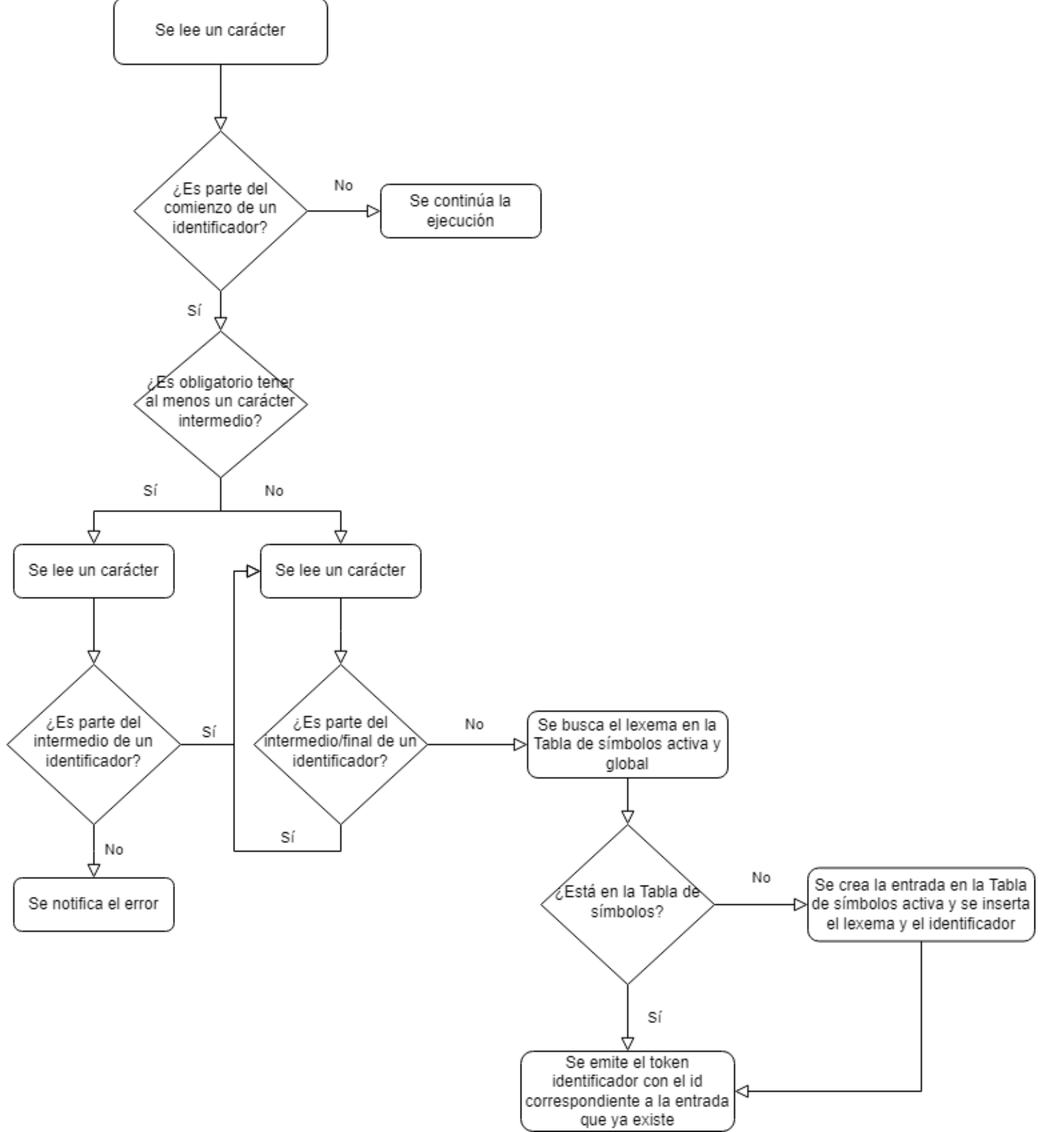

Ilustración 18: Diagrama de flujo para el reconocimiento de los identificadores.

4.1.1.9 Reconocimiento de saltos de línea

El *token* salto de línea se emite o no dependiendo de si el lenguaje tiene la opción “*token* salto de línea” activada. Esta opción se puede activar desde la ventana de definición del lenguaje de DRACO. Para reconocer este *token* se deben haber

intentado reconocer el resto de los *tokens* sin éxito. Si hay un salto de línea, a continuación, se emitirá el *token* salto de línea si el lenguaje lo tiene activado. Con esta opción se cumpliría el requisito 1.6. A continuación, en la ilustración 19, se muestra la parte de la página de definición del lenguaje que determina si el *token* salto de línea está definido.

Seleccionar si existe en este lenguaje
Opción CR ☑

Ilustración 19: Interfaz de DRACO de la opción salto de línea.

4.1.2 Tabla de símbolos

Para implementar una Tabla de símbolos genérica hay que poder almacenar todo tipo de entradas posibles en el lenguaje. En la versión actual de la definición del lenguaje de DRACO se pueden almacenar entradas de variables y de funciones. Las operaciones que debe hacer una Tabla de símbolos para ser funcional son las que están descritas en los requisitos 2.1 a 2.7.

La Tabla de símbolos será utilizada en el Analizador léxico para emitir los *tokens* identificador y el Analizador semántico se encargará de iniciar y destruir las Tablas locales que se crean cuando se entra a una función.

Puede haber dos instancias de Tabla de símbolos activas a la vez como máximo, la Tabla global y la Tabla local, que será sustituida por otra Tabla local cuando se inicie una nueva función.

4.1.3 Analizador sintáctico

El Analizador sintáctico será el módulo encargado de recibir los *tokens* del Analizador léxico. El tipo de analizador sintáctico que se ha decidido usar es el analizador ascendente, porque su gramática es más fácil de entender y su algoritmo es más sencillo de implementar. Entre otras, se crearán las funciones cierre de un conjunto de ítems, *goto* de un conjunto de ítems, aumento de la Gramática, creación de la colección canónica y la creación de la tabla acción y *goto*.

4.1.3.1 Importación de la Gramática y aumento de la Gramática

En el Analizador ascendente se usan las reglas de la Gramática del lenguaje y se necesita tenerlas disponibles en el código para poder trabajar con ellas. Para esto se usará el código de DRACO ya existente de análisis de Gramáticas, ya que devuelve una estructura que representa la Gramática del lenguaje, con los símbolos terminales y no terminales, el axioma y sus producciones.

Para comenzar el proceso de creación del analizador será necesario aumentar la Gramática, esto se hace sustituyendo el símbolo Axioma por un símbolo nuevo y añadir una regla de la siguiente manera: `NuevoAxioma -> Axioma`.

4.1.3.2 Ítems

Para implementar el algoritmo del analizador ascendente se debe usar una estructura que tenga los mismos componentes que una regla, el antecedente, los consecuentes y el número de regla; y un indicador que muestre por dónde va el análisis de la regla. Esta estructura que representa a un ítem LR(0) o, simplemente ítem, será utilizada para hacer el cierre de un conjunto de ítems y el algoritmo *goto*.

4.1.3.3 Cierre de un conjunto de ítems

El cierre de un conjunto de ítems es un algoritmo que toma un conjunto I de ítems de una gramática y devuelve otro conjunto de ítems aplicando el siguiente algoritmo [20], cumpliendo el requisito 3.4:

1. Se añaden todos los elementos del conjunto inicial al cierre de I.
2. Para cada ítem que tenga el indicador del análisis antes de un símbolo no terminal, se añaden todas las reglas que lo tengan como antecedente como un nuevo ítem cuyo indicador del análisis está antes del primer elemento del consecuente de la regla.
3. Se repite el paso 2 hasta que no se puedan añadir más símbolos al cierre.

4.1.3.4 Función *Goto* de un conjunto de ítems y un símbolo de la Gramática

El *goto* de un conjunto de ítems I y un símbolo X se define como el cierre del conjunto de todos los ítems que tengan el símbolo X después del indicador del análisis, siempre que el ítem con el indicador de análisis antes de X esté en I [20], cumpliendo el requisito 3.4.

4.1.3.5 Creación de la colección canónica

La colección canónica representa los estados del AFD que es capaz de reconocer todos los prefijos viables de la gramática. La colección canónica se hace usando la gramática aumentada del lenguaje del que se quiera construir la colección. Este algoritmo cumplirá el requisito 3.4. El algoritmo que se seguirá para construir la colección canónica es el siguiente [20]:

1. Se hace el cierre de la regla del axioma de la gramática aumentada.
2. Para cada conjunto de ítems en la colección y cada símbolo gramatical tal que el *goto* de ambos no este vacío y no esté en la colección, se añade el conjunto a la colección.
3. Se repite el paso 2 hasta que no se pueden añadir más elementos a la colección.

4.1.3.6 *First* y *Follow* de un símbolo gramatical

El *First* de un símbolo se define como el conjunto de símbolos terminales que pueden aparecer como primer símbolo terminal en las cadenas derivadas a partir de dicho símbolo. El *First* de un símbolo terminal es él mismo.

Para calcular el *First* de un símbolo no terminal se seguirá el siguiente algoritmo [21]:

1. Si existe una regla en la que el símbolo derive a la cadena vacía o lambda, se añadirá lambda al *First* del símbolo.
2. Para cada una de las reglas gramaticales del símbolo, se sigue el siguiente procedimiento, los elementos del consecuente de una regla se denotarán como $Y_1 \ldots Y_n$:
    a. Todos los elementos no nulos del *First* de Y_1 se añaden al *First* del símbolo.
    b. Si lambda pertenece al *First* de Y_1, se avanza en los símbolos de la regla.
    c. Se repite el procedimiento con el símbolo Y_{i+1}, si lambda no pertenece al *First* de Y_{i+1}, se termina la ejecución.

El *Follow* de un no terminal se define como el conjunto de terminales que pueden aparecer inmediatamente a la derecha del no terminal en alguna forma sentencial. Si el símbolo no terminal puede ser el símbolo de más a la derecha de alguna forma sentencial, entonces $ (el delimitador de la entrada) está en el *Follow* del símbolo.

Para calcular el *Follow* de un símbolo A se aplica el siguiente algoritmo [21]:

1. Si A es el axioma de la gramática, añadir $ al *Follow* de A.
2. Si existe una regla en la que A sea parte del consecuente y tenga después a un símbolo B, se añaden todos los elementos no nulos del *First* de B al First de A.

3. Si existe una regla en la que A esté al final o haya después un símbolo B que pueda derivar a lambda, todo lo que esté en el *Follow* de B se añade al *Follow* de A.

Estos algoritmos cumplen los requisitos 3.4.

4.1.3.7 Algoritmo de creación de la tabla Acción y Goto

Para crear la tabla de análisis se usará la gramática aumentada del lenguaje del que se quiera construir la tabla. Este algoritmo cumplirá los requisitos 3.4. El algoritmo para construir la tabla de análisis es el siguiente [20]:

1. Se construye la colección canónica de la gramática aumentada.
2. El estado i es el que se obtiene a partir del conjunto de ítems de la colección canónica I_i. Las acciones para cada estado se determinan de la siguiente manera:
    a. Si el indicador del análisis está situado antes de un símbolo terminal a y el goto del estado I_i con el símbolo terminal da lugar a otro estado j de la colección canónica, la acción del estado [i, a] será "desplazar j".
    b. Si el indicador de análisis está al final de una regla x, la acción a realizar será "reducir por x" para todos los terminales que pertenezcan al *Follow* del antecedente de la regla que se ha analizado; esta regla no se aplica al axioma de la gramática aumentada.
    c. Si el indicador de análisis está situado al final de la regla del axioma, la acción [i, $] será "aceptar".
3. Si hay más de una regla en alguna casilla, no es posible construir un Analizador Sintáctico mediante este algoritmo para la gramática dada.
4. Las transiciones del estado i se construyen para todos los no terminales A mediante la regla si goto(I_i,A) = I_j, entonces GOTO[i,A] = j.
5. Todas las casillas no definidas por las reglas 2 y 3 quedan en blanco y representan los casos de error.
6. El estado inicial del analizador es el que tenga el conjunto de ítems que contiene la regla del axioma con el indicador del análisis al comienzo de la regla.

4.1.3.8 Algoritmo de análisis sintáctico

Para poder analizar un fichero fuente se deberá introducir el delimitador *$* al final del todo (puede usarse el *token* EOF). Además, será necesario haber construido previamente la tabla de análisis para la gramática del lenguaje. Con este algoritmo se cumplirán los requisitos 3.1, 3.2, 3.3, 3.5 y 3.6.

Primero, la pila contendrá el estado inicial I_0 y la cadena de *tokens* de la entrada está pendiente de ser analizada. Se seguirá el siguiente algoritmo [20], que terminará al encontrar una acción de aceptar o de error.

1. Se pide un *token* "a" al Analizador léxico.
2. Siendo s el estado de la cima de la pila, se ejecutará una de las siguientes acciones:
    a. Si la acción de la tabla de análisis [s, a] es desplazar s', se mete el símbolo "a" en la pila y se mete el símbolo s' en la pila, se pide además un *token* al Analizador léxico.
    b. Si la acción de la tabla de análisis [s, a] es reducir por una regla, se sacan de la pila el doble de símbolos de los que hay en el consecuente de la regla, se mete el antecedente de la regla en la pila y siendo s' el estado que estaba en la cima de la pila se obtiene el *goto* de s' y el antecedente de la regla y se mete en la pila

   c. Si la acción es aceptar, se termina el análisis
   d. Si se detecta un error, se notifica y se continúa el análisis.
3. Se repite el paso 2 hasta que se encuentra un error o se llega a la acción de aceptar.

4.1.4 Analizador semántico

Para implementar las reglas semánticas del Analizador semántico se decidió dividir en grupos las reglas del lenguaje y cada grupo correspondería a un tipo de regla semántica. Por ejemplo, en el caso de que hubiera una regla que correspondiese a una operación como puede ser la suma, se habría definido previamente en la configuración del lenguaje qué tipo deben tener tanto los operandos como el resultado de la operación.

Para ejecutar estas reglas semánticas se unirá el algoritmo que se use para el Analizador semántico al Analizador sintáctico para que analicen el código simultáneamente.

Las especificaciones de las reglas se podrán consultar en la base de datos de DRACO, en las que se crearán tablas nuevas para almacenar la información que necesite cada tipo de regla para ser ejecutada. Estas reglas irán relacionadas con la regla que se quiere ejecutar en ese momento en el analizador sintáctico, ya que en el proceso de análisis se usa un indicador para ver hasta donde se ha analizado de una regla, cumpliendo con los requisitos 4.1 y 4.2.

El Analizador semántico accederá a la Tabla de Símbolos activa para poder introducir los datos de los tipos de las variables o símbolos procesados en ese momento, cumpliendo con el requisito 4.3 y 4.4.

Si el Analizador semántico encuentra un error, se notificará y se seguirá con el análisis cumpliendo con el requisito 4.5.

4.1.5 Configuración del sistema

Para completar la información necesaria para realizar los algoritmos de análisis será necesario añadir varias páginas de configuración del lenguaje y modificar la página de definición del lenguaje.

Para añadir información del Analizador léxico será necesario modificar la interfaz de la definición del lenguaje para añadir una opción en los identificadores y en los números para implementar el algoritmo descrito en el apartado de reconocimiento de *tokens* numéricos.

Se tendrá que añadir una página adicional para definir el valor de los símbolos de las bases de los números del sistema; la página tendrá la misma estructura que las páginas de DRACO y para cada tipo de número definido en el sistema se creará una sección con una lista de los elementos de las bases que no sean decimales o dig, ya que dig es una palabra reservada en DRACO que equivale a todos los números del 0 al 9. Al lado de cada símbolo habrá un elemento que permita introducir un valor en decimal y guardará en la base de datos el valor del símbolo y el símbolo al que corresponde; esto cumpliría los requisitos 5.3, 5.2 y 5.1.

Para realizar esta tarea se deberá añadir una nueva tabla en la base de datos de DRACO que tendrá la siguiente estructura.

Campo	**Descripción**
lexema	Lexema al que pertenece el símbolo

Campo	Descripción
símbolo	Símbolo al que se va a asignar un valor
valor	Valor del símbolo
id_lenguaje	Identificador del lenguaje al que pertenece el lexema

Con esta estructura se podrá acceder desde el código a la base de datos para añadir o consultar los datos del valor de cada símbolo. A continuación, en la figura 20, prototipo de la interfaz de DRACO para asignar el valor a la base de un token numérico, se muestra como estará estructurada la página de la asignación de los valores a los elementos de un *token* numérico, que estará formada por un formulario por cada tipo de número que esté declarado en el lenguaje. Cuando en la forma de construir un *token* numérico aparecen números decimales, se asume que el valor de estos números equivale a su valor en decimal.

WWW.DRACO.COM

OX/DIG B C D/DIG B C D

OX NULL

B 11

C 12

D 13

DIG A B C D/DIG A B C D/DIG A B C D

A 10

B 11

C 12

D 13

Ilustración 20: Prototipo de la interfaz de DRACO para asignar el valor a la base de un *token* numérico.

Para el Analizador sintáctico se deberá añadir una página adicional para que el profesor pueda subir la Gramática que será analizada. La página tendrá un campo de subida en el que se subirá la Gramática. Además, al subir la Gramática se comprobará si está formada correctamente para poder crear un Analizador ascendente basado en la Gramática. Con esta implementación se cumplirían los requisitos 5.2 y 5.4. A continuación se muestra el prototipo de interfaz propuesto para esta página en la ilustración 21, prototipo de la interfaz de DRACO para la subida de la Gramática del lenguaje.

Ilustración 21: Prototipo de la interfaz de DRACO para la subida de la Gramática del lenguaje.

Para el Analizador semántico también habrá varias páginas en la configuración del lenguaje en las que se podrá dar más información de los bloques de reglas, por ejemplo, se tendrá una página para las reglas que correspondan a operaciones en la que se podrá definir qué tipo de salida tiene la operación y que tipo deben tener las entradas de los operandos.

4.2 Implementación

Para la implementación del sistema se han usado las tecnologías HTML5, CSS3, PHP 7.4 y SQL. Para el entorno de desarrollo se ha usado PHPstorm 2024.1.4 y Wampserver 3.3.2. Para la gestión de la base de datos local se ha usado PHPMyAdmin, que ofrece una interfaz para interactuar con una base de datos desplegada en local, en la que se han volcado los archivos de inicialización de la base de datos de DRACO.

En este apartado se describirá la implementación de las funcionalidades del sistema.

4.2.1 Analizador léxico

Para implementar el Analizador léxico se creó una clase de PHP llamada AnLex, que contiene el algoritmo encargado de extraer un *token* del fichero fuente. A continuación, se explicará el proceso seguido para implementar esta clase y los componentes necesarios para su correcto funcionamiento.

4.2.1.1 Clase token

Para poder implementar el valor que devuelve el Analizador léxico tras analizar un fichero se creó una clase `Token`, que tiene como parámetros el tipo de *token* y el valor del *token*.

Estos dos parámetros podrán ser consultados por medio de funciones como `getType()` o `getValue()`, además, para poder representar estos *tokens* en caso de que haya algún error, se ha añadido un método `toString()`, que devuelve el valor de los campos del objeto en forma de cadena.

4.2.1.2 Clase AnLex

La clase AnLex es la encargada de consultar a la base de datos las características del lenguaje y almacenarlas en atributos que se usarán a lo largo del algoritmo. Los atributos más relevantes para crear la clase AnLex fueron los siguientes:

- $connection: Este atributo almacena la conexión a la base de datos que se va a usar para hacer las consultas.
- $arrayComentariosGeneral: Este atributo almacena la información necesaria para poder reconocer los comentarios dentro del fichero fuente. Los datos se almacenan como un vector en el que cada posición se almacena una forma de escribir comentarios; si no hay, se iniciará un vector vacío.
- $arrayCadenas: Este atributo almacena la información necesaria para poder reconocer cadenas y caracteres dentro de un fichero fuente. Los datos se almacenan como un vector de dos elementos que contiene un vector que contiene las formas de formar cadenas y otro vector que contiene las formas de formar caracteres. Las cadenas y los caracteres solo contienen el carácter de comienzo de una cadena, por lo tanto los componentes de estos vectores tendrán 3 elementos cada uno, el comienzo del *token* y el código de *token* dentro de la base de datos de DRACO.
- $saltoDeLinea: Este atributo es una variable lógica cuyo valor varía dependiendo de si el lenguaje tiene activado el *token* salto de línea.
- $simbolos: Este atributo contiene un vector con todos los *tokens* que pertenecen al conjunto de los símbolos del lenguaje; para cada símbolo se almacena el lexema del *token* y el código del *token*.
- $arrayDeclaracionNumeros: Este atributo contiene la información necesaria para formar números; tiene una estructura similar a las cadenas, con la diferencia de que cada forma de declarar un número contiene información sobre la estructura de la declaración, que se divide en inicio, intermedio y final, si es el intermedio obligatorio y el código de *token*.
- $arrayIdentificadores: Este atributo contiene la información necesaria para reconocer identificadores del lenguaje; tiene una estructura similar a los *tokens* numéricos, con la diferencia de que cada forma de declarar identificadores no tiene una base.
- $arrayPalabraReservada: Este atributo contiene un vector con todas las palabras reservadas del lenguaje. Se almacena para cada palabra reservada el lexema del *token* y el código del *token*.
- $stringFile: Este atributo almacena el fichero fuente en forma de cadena para mejorar la eficiencia, ya que acceder a un fichero abierto muchas veces consume más que leerlo una vez e ir iterando sobre un vector de caracteres. Al inicializar el objeto AnLex con el parámetro *path*, se almacena todo el texto que contiene el fichero descrito por el parámetro en el atributo y se añade un nulo al final para indicar que se ha acabado de leer el fichero y evitar fallos.
- $linecounter: Este atributo almacena la línea del fichero fuente por la que va el Analizador; siempre que se lee un salto de línea de la variable $stringfile, se aumenta en uno el contador de líneas.
- $pointer: Este atributo almacena la posición del vector por la que va el análisis.
- $token: Este atributo es el encargado de almacenar el *token* completo que será devuelto en la iteración del algoritmo.

- `$globalTable`: Este atributo es el encargado de almacenar la tabla de símbolos global de las variables del fichero fuente.

Estos atributos son inicializados en el constructor del Analizador y usados en el algoritmo de análisis.

4.2.1.3 Algoritmo de análisis de la clase AnLex

El algoritmo usado en el Analizador léxico sigue un orden para detectar *tokens*. La función encargada de implementar este orden es `getToken()`, situada en la clase `AnLex`. El orden utilizado es el siguiente:

- Primero, se detecta si el siguiente carácter dentro del fichero fuente corresponde al comienzo de un carácter. Se detectan primero los comentarios para evitar detectar símbolos en los caracteres de apertura o identificadores y números dentro de las palabras de los comentarios; para cerrar un comentario deben leerse los caracteres correspondientes a la cadena de finalización de un comentario; si no se encuentra un final para el comentario, se envía un error y se continua con el análisis, aunque en este caso, si se deja un comentario sin cerrar no se podría continuar con el análisis, ya que no hay más *tokens* que leer, por lo tanto, cada vez que se vuelva a llamar a la función de análisis se devolverá que se ha acabado la lectura.
- Después de los caracteres se leen las cadenas y los caracteres en el mismo bloque de código, ya que su estructura es la misma y lo único que cambia es su código de *token*. Si no se encuentra el carácter que da fin a una cadena o a un carácter se devolverá el mismo error que los comentarios al Analizador, para poder continuar con el análisis.
- Para poder reconocer símbolos del sistema se ha usado un algoritmo que elige el símbolo de mayor tamaño en el caso de que se encuentre una situación en la que haya una serie de caracteres equivalentes a varios símbolos separados y a uno de mayor tamaño; esto se ha hecho usando una lista que inicialmente contiene a todos los símbolos. Después, se itera sobre la lista comparando el carácter actual con el carácter inicial de todos los símbolos, hasta que solo quede uno, ya que se irán eliminando de la lista los que no sean iguales a lo que se haya leído. Si al final la lista no contiene ningún símbolo, se continúa el análisis, y si se queda un símbolo en la lista, se emite el *token* correspondiente al símbolo.
- Después continúa la parte de los números, que consiste en reconocer el inicio de un *token* numérico dentro de la cadena que se está leyendo del fichero fuente. Si se detecta un *token* numérico, se continua la ejecución almacenando todos los caracteres del *token* hasta que se llega al final del *token*. Antes de emitir el *token* entero, se debe transformar la cadena leída a decimal, ya que el ensamblador usado en las prácticas solo admite números decimales. Esto se hace usando la base del número que está almacenado junto a la descripción de la estructura del *token* en la base de datos de DRACO.
- Para evitar detectar identificadores donde están palabras reservadas se intenta detectar antes si hay una palabra reservada. Esto se hace leyendo los caracteres equivalentes a la palabra reservada más grande, a menos que no queden suficientes caracteres, y se compara la cadena leída con cada palabra reservada del lenguaje; si no se detecta ninguna palabra reservada, se continúa con la detección de identificadores.
- Para detectar identificadores se utiliza el mismo algoritmo que el usado con los números, pero en vez de emitir un *token* entero se emite un *token*

identificador y en vez de usar la base de un número se comprueba si existe el lexema en la Tabla de símbolos. En el caso de que exista, se consulta el campo `id` de esa entrada y se emite un *token* identificador con el valor del campo `id` de la entrada detectada; en el caso contrario, se crea una nueva entrada en la Tabla de símbolos y se emite el *token* identificador con código identificador y el valor del campo `id` de la entrada.

- Para detectar saltos de línea se consulta el atributo `$saltoDeLinea`. En el caso de que esté activado, se emite el *token* salto de línea; en este fragmento de código también se aumenta el valor del contador que contiene el número de líneas leídas hasta el momento.
- Si no se detecta ninguno de los tipos de *token* definidos, se muestra un error y se continúa con el análisis saltando el carácter no reconocido.

4.2.1.4 Funciones auxiliares utilizadas en el Analizador léxico

Para poder implementar el Analizador léxico se han usado funciones auxiliares que realizan tareas que se repiten o que realizan una función concreta para mantener el código limpio. A continuación, se describen las funciones más relevantes para el Analizador.

4.2.1.4.1 Función `valueOf`

La función `valueOf()` recibe como parámetros una cadena que se quiere transformar a decimal, la estructura del *token* que se ha detectado y la conexión a la base de datos.

La función consulta el valor de cada símbolo de la base para poder calcular el valor decimal del número y lo devuelve en forma de entero.

4.2.1.4.2 Función `getTokens`

La función `getTokens()` consulta en la base de datos los *tokens* de la consulta sql que recibe como parámetro. La función asume que los *tokens* tendrán un campo lexema para meterlos en un vector de cadenas que será el valor devuelto.

4.2.1.4.3 Función `inRegex`

La función `inRegex()` recibe como parámetro un carácter y un conjunto de símbolos que pueden ser “let”, “dig” o cualquier letra o número. Eesta función se usa para saber si un carácter leído está dentro de la declaración de un *token*. “let” se refiere a cualquier letra de la “a” a la “z” y “dig” a cualquier número entre el 0 y el 9.

4.2.2 Tabla de símbolos

La Tabla de símbolos consiste en dos clases, una clase `Key` y una clase `SymbolTable`. A continuación, se describe la funcionalidad de ambas clases.

4.2.2.1 Clase Key

Esta clase cumple la función de una entrada en la Tabla de símbolos. Tiene como atributos el lexema del identificador, la posición en la Tabla según lo que ocupa cada tipo y el tipo del identificador.

En el caso de que las entradas resulten ser funciones, hay campos adicionales para completar la información, como el número de parámetros que admite, el tipo de los parámetros, el modo de paso de los parámetros, el tipo de retorno y la etiqueta de la función.

Para facilitar el acceso y la modificación de estos campos, se han definido funciones que permiten modificar y consultar cada uno de los campos del objeto, por ejemplo, para el atributo `$tipo` habría dos funciones, `getTipo()` y `setTipo()`.

4.2.2.2 Clase SymbolTable

Esta clase cumple la función de una Tabla de símbolos, tiene las funciones que están descritas en los requisitos, que son las funciones `addKey`, `getKeyById`, `getKeyByLexeme`. La Tabla de símbolos se destruye cuando se elimina la referencia y se construye al crearse el objeto. Además, cuando se hace cualquier método que comienza por `getKey`, se devuelve una entrada que se puede modificar.

4.2.3 Analizador sintáctico

Para la implementación del Analizador sintáctico se han usado los algoritmos definidos por el diseño, para facilitar esta implementación se han creado varias clases que representan estructuras de datos de los algoritmos de Análisis, como los *ítems* o la *colección canónica*. En esta sección se explicará cómo se han implementado los algoritmos de Análisis sintáctico y como se ha enlazado con los demás analizadores.

4.2.3.1 Clase AnSin

Para implementar el algoritmo de análisis se decidió crear una clase llamada `AnSin`, que se encarga de almacenar la información necesaria para llevar a cabo el análisis. Los atributos que tiene esta clase son:

- Una referencia a un objeto del tipo `AnLex`, para poder pedir *tokens* cada vez que se necesiten para continuar el análisis.
- Una referencia a un objeto del tipo `Token`, que es el último *token* que ha sido utilizado para el análisis.
- Una referencia a las Tablas de símbolos activas.
- Una referencia a un objeto del tipo `TablaAG`, para poder llevar a cabo el análisis.
- Para representar la pila del análisis se ha utilizado una referencia a la clase SplStack de PHP.

Dentro de la clase `AnSin`, se han definido dos funciones `Analizar()` y el constructor del objeto, a continuación se detalla el funcionamiento de ambas funciones:

- La función `Analizar()` se encarga de llevar a cabo el análisis usando el algoritmo descrito en el diseño.
- El constructor de la clase recibe como parámetro la ruta del fichero que se quiere analizar y se encarga de inicializar el Analizador léxico, crear la tabla acción y *goto*, crear la Tabla de símbolos global y iniciar la pila de análisis.

4.2.3.2 Clase Item y clase ItemGroup

Para poder representar los *ítems* descritos por el diseño, se creó una clase ítem. Esta clase tiene como atributos el antecedente de la regla representada por una cadena, los elementos del consecuente representados como una lista de cadenas y el punto por el que va el análisis de la regla representado por un entero.

Las funciones que contiene esta clase son:

- El constructor crea el array de elementos del consecuente inicialmente vacío, se inicializa la variable punto a cero y se recibe por parámetro el antecedente de la función.
- Hay dos funciones que se encargan de modificar y consultar el antecedente, estas son `setAnte()` y `getAnte()`. También hay dos funciones, `setPunto()` y `getPunto()`, que se encargan de consultar y modificar el atributo punto del objeto.

- La función `getSimboloPunto()`, que se encarga de devolver el símbolo que está despues del punto.
- La función `introducirElemConsec()`, que se encarga de meter un símbolo al final de la lista del consecuente.
- La función `sigElem()`, que hace que el punto avance dentro de la regla, apuntando al siguiente símbolo.
- La función `compareTo()`, que se encarga de comparar *ítems*, devuelve verdadero si los *ítems* son iguales y falso si no lo son.

La clase `ItemGroup` se creó para poder comparar grupos de *ítems*, esta clase tiene como atributo una lista de *ítems*, además de dos funciones, `setList()` y `getList()`, que se encargan de modificar y consultar el valor de la lista de *ítems*, el constructor, que se encarga de inicializar la lista vacía y una función que compara dos grupos de *ítems*, usando la función `compareTo()` de la clase `Item`, llamada `compareTo()`.

4.2.3.3 Clase ColeccionCanonica

Para poder implementar el algoritmo de creación de la colección canónica y almacenar su resultado se decidió crear una clase llamada `ColeccionCanonica`, que tiene como atributo la colección canónica de la gramática del lenguaje en forma de lista de `ItemGroup`. Para poder implementar este algoritmo se creó dentro de esta clase una función llamada `crearColeccion()`, que se encarga de crear la colección canónica para la gramática y almacenarla en el atributo de la clase, además, hay una función, `getColeccion()`, que permite consultar la colección canónica.

4.2.3.4 Clase TablaAG

Se creó una clase para representar la tabla acción y *goto* de la gramática, se decidió crear una clase llamada `TablaAG`, que tiene como atributos la tabla acción y *goto* de la gramática del lenguaje almacenado como un array de dos dimensiones y una lista que corresponde a los símbolos correspondientes a cada columna de la tabla. Para poder implementar este algoritmo se creó dentro de esta clase una función llamada `crearTabla()`, que se encarga de crear la tabla para la gramática y almacenarla en el atributo de la clase, además, hay una función, `getTabla()`, que permite consultar la tabla acción y *goto*.

En el caso que no se pueda crear la tabla porque hay dos o más entradas por celda se almacena un nulo dentro del atributo que almacena la tabla en el objeto.

4.2.3.5 Algoritmos necesarios para el análisis

Para poder realizar el análisis sintáctico son necesarias las funciones de cierre, *goto*, *first* y *follow*. Estas funciones están en un fichero separado de las clases, ya que estos algoritmos se usan en varios sitios, para poder usar estas funciones es necesario importar el fichero llamado *AnSinAux*.

4.3 Pruebas

En este apartado se describirá el tipo de pruebas que se han realizado en el sistema y con qué herramientas se ha podido probar el sistema.

4.3.1 Uso del IDE y el depurador

Durante el desarrollo del sistema se ha utilizado una función del entorno de desarrollo llamada *debugger*. En la figura 22, interfaz del *debugger* integrado en PHPstorm, se muestra la interfaz de usuario del entorno de desarrollo con un ejemplo de ejecución del Analizador léxico.

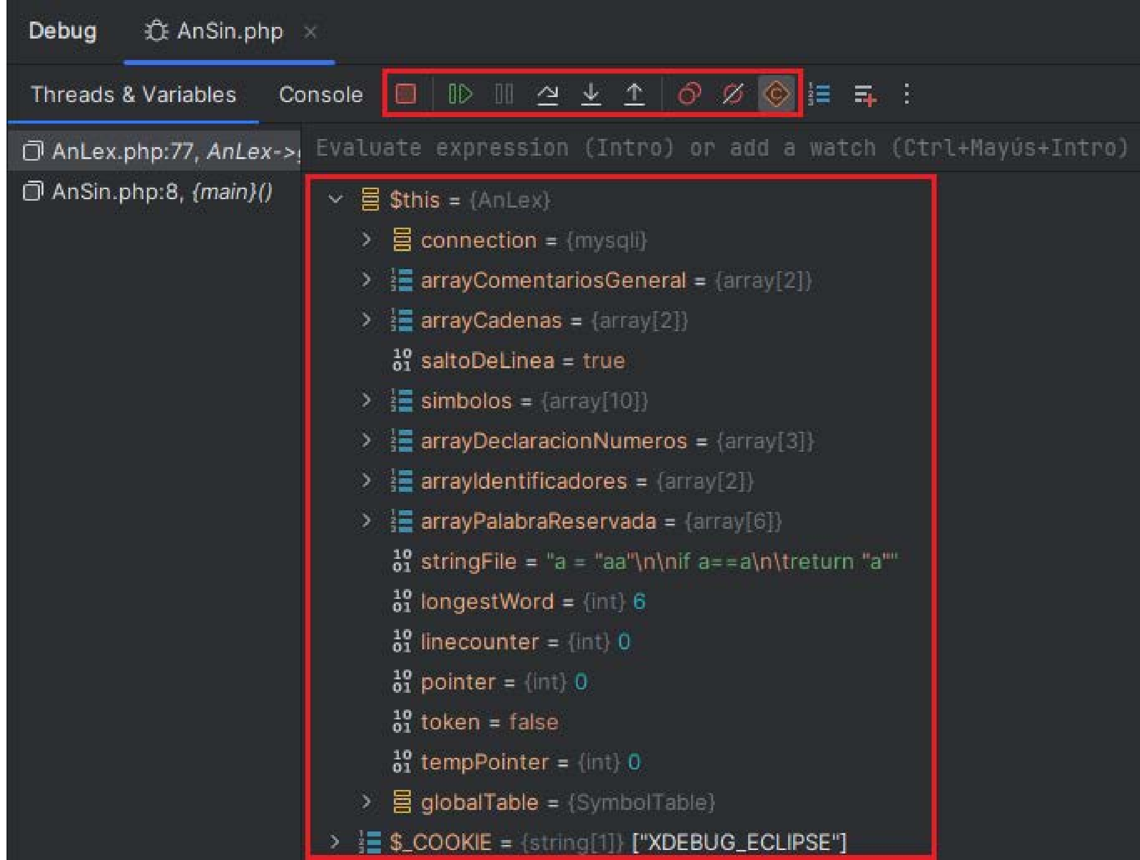

Ilustración 22: Interfaz del *debugger* integrado en PHPstorm.

Las funciones que ofrece esta herramienta se pueden ver en el menú del cuadro rojo superior. Las funciones de los botones, ordenadas de izquierda a derecha son:

- Parar la ejecución.
- Continuar la ejecución hasta llegar a un punto de ruptura. Un punto de ruptura es un punto en el código en el que se parará la ejecución del programa para ver el valor de las variables y el estado del programa en ese punto.
- Parar la ejecución en el caso de que se quede en un bucle infinito.
- Avanzar una instrucción del código.
- Avanzar una instrucción del código, con la diferencia de que si la siguiente instrucción es una función se continúa la ejecución a partir de la primera instrucción de la función.
- Salir de la función actual, avanzando la ejecución hasta que se salga de la función actual.
- Ver los puntos de ruptura.
- Desactivar los puntos de ruptura.
- Ver las constantes definidas por el usuario.

Además, para cada constante y variable de la ejecución se puede ver y modificar su valor en el momento en el que se haya detenido la ejecución; también se pueden crear variables y comprobar condiciones lógicas del sistema con la herramienta evaluación.

Para poder depurar páginas generadas con PHP, se ha usado la extensión Xdebug, mostrada en la ilustración 23, que permite enviar una señal al servidor PHP y al IDE de que se quiere depurar la página que se ha ejecutado.

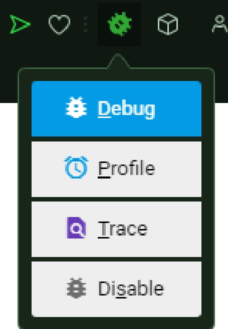

Ilustración 23: Interfaz de la extensión de depuración en el navegador Opera.

4.3.2 Uso del log de PHP

El servidor WAMPP que se usa para poder tener un servidor de PHP tiene un *log* de errores que permite tener un control más avanzado de los errores del *script* de PHP. Es especialmente útil para detectar en qué línea se provoca un bucle infinito o solucionar *warnings* que presenta el *script* pero no se muestran durante la ejecución.

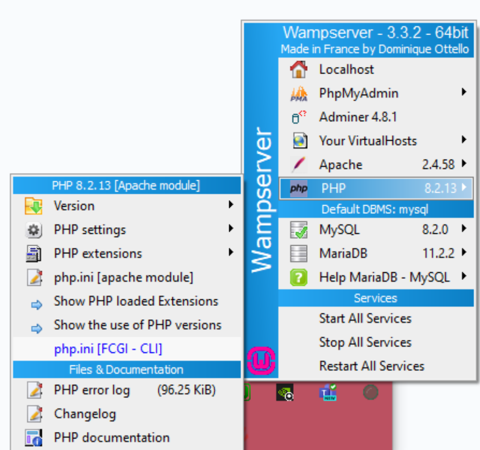

Ilustración 24: Interfaz de la barra de tareas de WAMPP.

Para poder abrir el *log* se puede ir a la extensión en la barra de tareas (en el caso de Windows) y en el menú que sale pulsar PHP, en la barra lateral que se genera sale como opción abrir el fichero de *log* de PHP e información acerca de su tamaño. Este proceso está representado en la ilustración 24.

```
[26-May-2024 09:05:22 CET] PHP Fatal error:  Uncaught TypeError: implode():
Argument #2 ($array) must be of type ?array, string given in
E:\wamp64\www\htdocs\Practicas\Configuracion\confi_opciones4.php:387

Stack trace:

#0:\wamp64\www\htdocs\Practicas\Configuracion\confi_opciones4.php(387):
implode(Array, ', ')

#1 {main}  thrown in
E:\wamp64\www\htdocs\Practicas\Configuracion\confi_opciones4.php on line 387
```

Este ejemplo de error da información acerca de qué función ha fallado y en qué línea.

4.3.3 Pruebas realizadas

Durante la implementación de las funcionalidades del sistema se han realizado dos tipos de pruebas para asegurar el buen funcionamiento de las funciones y clases.

4.3.3.1 Pruebas unitarias

Las pruebas unitarias consisten en aislar una característica del código y comprobar si su comportamiento es el deseado. este tipo de pruebas se han realizado durante la implementación del Analizador léxico. Para cada módulo del analizador, se ha separado el código en una función y se han comprobado los casos de error que no se cumplían. Además de detectar varios errores de implementación, esto llevó, por ejemplo, a hacer el cambio de añadir un campo a la tabla de los números en la página de definición del lenguaje.

Durante el desarrollo del Analizador sintáctico se probaron las diferentes funciones que se tenían que desarrollar para el algoritmo de Análisis ascendente, comprobando el comportamiento de las funciones *First*, *Follow*, cierre, *goto*, la clase ítem, la clase conjunto de ítems y la función colección canónica.

4.3.3.2 Pruebas de integración

Al terminar de implementar un módulo se debía comprobar si su funcionamiento era el esperado. En el caso del Analizador léxico se probaron diferentes archivos de entrada con casos límite para asegurar que el algoritmo usado puede soportar las características del lenguaje genérico definido por DRACO.

5 Resultados y conclusiones

El desarrollo del Trabajo de Fin de Grado consistió en mejorar y ampliar la funcionalidad del sistema web DRACO, concretamente el módulo del corrector de las prácticas, que permite a los alumnos corregir los diferentes módulos del trabajo de Procesadores de Lenguajes, enviando ficheros de prueba a los alumnos para que estos puedan probarlo en diferentes casos, dando puntos del sistema para los alumnos que consigan que su procesador tenga el comportamiento correcto. Para la asignatura de Traductores de Lenguajes este sistema no existía; esto generó una necesidad en los estudiantes que se podía resolver implementando un sistema parecido al que es usado en Procesadores de Lenguajes.

Debido a la complejidad del proyecto, se ha dividido el trabajo entre varios Trabajos de Fin de Grado y Trabajos de Fin de Máster. En este Trabajo de Fin de Grado se han diseñado las tres primeras partes del compilador, que forman la parte de análisis de un compilador junto a la tabla de símbolos, para que funcionen de manera genérica (para un lenguaje que pueda ser modificado por el profesor) y se han implementado las dos primeras partes, el Analizador léxico y el Analizador sintáctico, junto a la tabla de símbolos. No se ha podido implementar el último módulo, el Analizador semántico, debido a la complejidad de los algoritmos que se han tenido que diseñar e implementar en los otros módulos.

La principal diferencia y lo que hace más complicada la implementación de este procesador es la capacidad que tiene de procesar un lenguaje que puede cambiar sus características. Pueden cambiar los tipos que tiene definidos el lenguaje, los operadores, las estructuras de control o cómo se declaran las variables. Esta diferencia ha causado que durante el desarrollo del procesador se hayan encontrado partes del sistema que no podían ser implementadas de la manera tradicional como se haría con un procesador normal. Para la implementación del Analizador léxico, normalmente se usaría un Autómata Finito Determinista, pero al no saber a priori la estructura de los lexemas de los *tokens* del lenguaje, se analizaron los *tokens* que podían configurarse en las páginas de definición del lenguaje y se realizó un algoritmo que recibe la estructura de todos los *tokens* posibles del lenguaje y se adapta a ellos. De manera similar, el Analizador sintáctico se tendrá que basar en la gramática proporcionada por el profesor. Finalmente, el Analizador semántico tendrá que obtener información acerca de las distintas estructuras del lenguaje a través de un cuestionario que deba rellenar el profesor; con esta información, podrá realizar las comprobaciones necesarias. Para poder asegurar el correcto funcionamiento de todo el sistema, se ha tenido que asegurar que todos los Analizadores interactúan de manera correcta y no hay errores, ya que están constantemente intercambiando información entre ellos.

Con este sistema se podrá dar apoyo a los alumnos de Traductores de Lenguajes, permitiéndoles comprobar el funcionamiento de sus prácticas antes de la entrega final, ya que actualmente sólo hay una oportunidad para el estudiante de comprobar el funcionamiento con el profesor. Además, la creación de una actividad con fecha límite en DRACO para realizar la actividad de corrección incentivará el realizar la parte del sistema que se corrige para la fecha límite y no para la corrección final con el profesor, lo que permitirá a los alumnos tener tiempo para refinar el compilador y como resultado aumentar la nota que tendría el proyecto.

Durante la realización del Trabajo se han detectado múltiples limitaciones a la hora de definir un lenguaje. Para arreglar estas limitaciones se han hecho cambios a la interfaz de definición del lenguaje y se han añadido nuevas páginas que permiten al profesor añadir un lenguaje o modificarlo con más opciones y flexibilidad.

A un nivel más personal el hecho de participar en un proyecto tan grande y con tantos módulos, aunque al principio fuera complicado ubicar las cosas o comprender qué hacía exactamente un fichero, me ha ayudado a comprender la importancia de organizar los proyectos en carpetas separadas y ficheros y funciones con nombres descriptivos. Durante la carrera, los proyectos que se realizaban no tenían tanta complejidad como lo tiene DRACO y no era necesario pensar en que algún alumno podía continuar mi trabajo, lo que me ha hecho poner un esfuerzo en hacer el código más legible y con nombres descriptivos.

Nunca antes había programado usando herramientas como WampServer, que me ha ayudado a simular un entorno web en el que se puedan probar los programas que se estaban desarrollando. Además, aprendí a usar las herramientas de depuración que tenía el IDE integradas. Como los IDE y las buenas prácticas como depurar están muy extendidas en el sector laboral, considero que la experiencia usando estas herramientas serán muy enriquecedoras para mi futuro laboral.

Durante la carrera se mencionaron tecnologías como HTML, CSS, JavaScript y PHP, pero nunca se llegó a tener una asignatura o práctica que se centrase en el desarrollo web. Durante el desarrollo de este Trabajo pude aprender a programar en PHP y a usar HTML y CSS, lenguajes y herramientas en los que no tenía experiencia previa.

También se ha usado SQL para la realización del Trabajo, ya que es donde la mayoría de la configuración del lenguaje se almacena después de configurarlo en las páginas de configuración del lenguaje en DRACO. En la carrera se ha estudiado SQL, sin embargo, no había usado nunca una base de datos tan compleja.

Durante la realización de este Trabajo se han propuesto soluciones innovadoras para un proyecto que no se había realizado antes, lo que me ha hecho pensar en soluciones fuera de lo común y tener en cuenta todo el sistema antes de comenzar a implementar una funcionalidad. Además, como el sistema va a ser usado por mucha gente se hicieron muchas pruebas para ver si todo funcionaba correctamente antes de seguir desarrollando los siguientes módulos del sistema.

6 Análisis de impacto

En este capítulo se analizará el impacto de este Trabajo en diferentes aspectos, a continuación, se analizarán cada uno de ellos.

6.1 Impacto social

DRACO tiene un gran impacto en el entorno universitario, concretamente en las asignaturas de Procesadores de Lenguajes y Traductores de Lenguajes. DRACO ayuda a los estudiantes a mantenerse motivados mediante un sistema de clasificación que anima a los estudiantes a ser competitivos. Además, el sistema es usado como una herramienta de repaso usando actividades que a su vez dan puntos para el *ranking* cuando se completan bien. Además, el sistema ha sido desarrollado principalmente por alumnos de la Escuela en prácticas o realizando el Trabajo de Fin de Grado, lo que hace que se puedan integrar en un proyecto más grande que las prácticas que se hacen en la Universidad, dando una aproximación a lo que es estar en un entorno más profesional.

6.2 Impacto personal

Participar en el desarrollo de DRACO me ha hecho aprender tecnologías muy usadas en desarrollo web como pueden ser HTML, CSS o PHP. Además, aprendí a desplegar un entorno de desarrollo para poder hacer pruebas y simular un entorno real. Durante el desarrollo de este Trabajo pude aplicar varios de los conocimientos que adquirí en la carrera en un proyecto más profesional.

6.3 ODS

El Trabajo realizado se alinea con varios de los Objetivos de Desarrollo Sostenible [22] establecidos por las Naciones Unidas En este apartado se explicarán los objetivos más relevantes para el proyecto.

6.3.1 ODS 4: Educación de calidad

Para el objetivo 4 se cumplen las siguientes metas:

- 4.3: DRACO hace posible que los alumnos de la Escuela accedan al sistema independientemente de su género.
- 4.4: DRACO ayuda a los alumnos a conseguir las competencias necesarias para aprobar las asignaturas de Procesadores y Traductores de Lenguajes, lo que permite a los alumnos acceder a empleos más técnicos.

6.3.2 ODS 8: Trabajo decente y crecimiento económico

Para el objetivo 8 se cumplen las siguientes metas:

- 8.4: DRACO es un sistema informático que al contrario que los métodos de enseñanza convencionales, usa la web como medio para dar servicio a los alumnos, evitando el gasto de materiales como el papel.
- 8.5: DRACO trata de manera igualitaria a todos los usuarios; también se ha adaptado el sistema para que las personas con diferentes discapacidades puedan acceder a todo su contenido.
- 8.6: DRACO ayuda a los jóvenes a cursar los estudios y darles formación especializada.

6.3.3 ODS 9: Industria, innovación e infraestructuras

Para el objetivo 9 se cumplen las siguientes metas:

- 9.5: DRACO permite a los profesores desarrollar la asignatura de una manera en la que se gastan muchos menos recursos y permiten a los alumnos hacer varias tareas desde casa, ofreciendo una experiencia más innovadora que en otras asignaturas de la carrera.

- 9.c: DRACO se ha desarrollado teniendo en cuenta todo tipo de navegadores para que las personas con un ordenador sin JavaScript o con un buscador más antiguo puedan acceder al sistema. Todas las páginas de DRACO tienen su funcionalidad implementada en PHP, aunque alguna tiene la opción de poder usar JavaScript, que es la opción usada por defecto, pero en el caso de que el ordenador no tenga JavaScript, se mostrará una versión de la página en HTML estático.

7 Futuras líneas de trabajo

DRACO es un sistema web que tiene una gran cantidad de funcionalidades y da servicio a dos asignaturas de la carrera, aunque todavía tiene una gran cantidad de ramas por las que seguir desarrollando el proyecto. Por ejemplo, este proyecto ha sido una nueva iniciativa que se ha comenzado este año, por lo que a día de hoy aún hay varias líneas de trabajo que se podrían seguir en los próximos Trabajos de Fin de Grado o de Máster.

Respecto a la parte encargada del análisis en el corrector de prácticas de Traductores de Lenguajes se podría implementar el Analizador semántico del lenguaje siguiendo las pautas de diseño que se han definido en este Trabajo. Se tendrían que añadir varias páginas de configuración que permitan añadir qué rol tiene cada regla del lenguaje. Para poder saber qué reglas hay en un lenguaje se puede aprovechar la página ya implementada de la subida de la gramática y almacenar las reglas en una nueva tabla de la base de datos para poder añadir información como el rol de la regla e información adicional que se necesite según el tipo de regla que sea.

Por otra parte, habrá que integrar la parte de síntesis con la parte de análisis. Esto se podría hacer desde el Analizador sintáctico, en la parte en la que se genera el *parse*, se tendría que asegurar que el Generador de Código Intermedio obtenga la información de la Tabla de Símbolos de la misma forma que el Analizador semántico y que se pueda integrar en el código mediante las acciones que tenga definidas para el lenguaje. Por ejemplo, las acciones del Generador de Código intermedio se podrían meter en la pila del Analizador sintáctico y cada vez que el Analizador sintáctico utilice una regla deberá llamar al algoritmo del Generador de Código Intermedio.

También se podría implementar la actividad de DRACO que llama al compilador genérico. Se tendrían que definir qué tipo de archivos fuente son adecuados para probar el compilador de los alumnos y se tendría que crear un nuevo tipo de actividad. Por ejemplo, podría implementarse de tal manera que el sistema envíe un archivo fuente para ejecutar y que pida un número o una cadena que tenga como resultado el programa. Para comprobar si es correcto, se tendría que comparar el resultado del compilador del alumno y el de prueba.

Para poder comprobar de la misma manera las asignaturas de Procesadores de Lenguajes y Traductores de Lenguajes se podrían añadir las nuevas opciones de definición del lenguaje a los comprobadores de Procesadores, permitiendo al profesor añadir más posibilidades al lenguaje. Para esto se tendría que actualizar el código de los comprobadores para que tengan en cuenta la nueva información del lenguaje que se ha añadido. Por otra parte, se podría usar el compilador como comprobador para la asignatura de Procesadores, haciendo que solo ejecute hasta una fase del análisis o desactive otras. Por ejemplo, si se quisiera probar el analizador semántico se tendría que desactivar el Generador de Código Objeto y el Generador de Código Intermedio, pero esta solución tiene la dificultad de hacer que la salida del compilador pueda traducirse a la salida del compilador del alumno. Es decir, si los *tokens* tienen una estructura muy diferente, el Analizador léxico no se podría comprobar o daría error al no ser iguales, sin embargo, para el Analizador sintáctico y el semántico sí sería posible realizar estas comprobaciones. Para comprobar el Analizador sintáctico se tendría que añadir como entrada la gramática del alumno y para el semántico la implementación sería más sencilla, ya que solo se usa el fichero de la Tabla de Símbolos, que se podría devolver al finalizar el análisis de un fichero.

Por otra parte, se debería actualizar en DRACO la versión de PHP 7.4 a PHP 8.3 para tener disponibles las últimas funcionalidades del lenguaje. Para poder realizar este cambio se tendría que ver la especificación de los cambios, especialmente qué cosas se eliminan del lenguaje y ver si se pueden eliminar o cambiar por opciones más modernas o alternativas.

Mas allá de las funciones de DRACO que tienen que ver directamente con la corrección de las prácticas de las asignaturas, se podría añadir un *ranking* para las prácticas que hayan obtenido mejores resultados; así se fomentaría la competitividad entre los estudiantes y los motivaría a tener un compilador que pase todas las pruebas que les pueda poner DRACO.

También se podría añadir una pestaña en DRACO que consista en un Chatbot que permita a los alumnos preguntar dudas acerca del temario de las asignaturas. Esto se podría implementar gracias a los avances que ha habido recientemente en el ámbito de la IA. Ya hay varios modelos *open-source* que se podrían usar para darles el contexto necesario de la asignatura en el formato adecuado. La dificultad que tendría la creación de este sistema sería recopilar el material suficiente como para cubrir todo el temario de la asignatura y transformarlo a un formato que pueda reconocer el modelo; la accesibilidad de este sistema dependería del tipo de salida que se implemente.

Para añadir más elementos de gamificación al sistema, se podrían implementar cosméticos que se puedan comprar con puntos; sin embargo, los alumnos generalmente no gastan puntos por que es la métrica que se usa para el *ranking* de todos los estudiantes. Para solucionar esto se podrían usar las monedas existentes en el sistema, y que podrían usarse también para comprar cosméticos. Además, se tendría que implementar una especie de “tienda” en la que se puedan comprar cosméticos para el avatar o decoraciones para la barra del *ranking* entre alumnos. Los objetos que se compren también podrían ser coleccionables y que al comprarlos o desbloquearlos todos se consiga un logro en la página web. Estos objetos coleccionables podrían ser referencias a la historia de los compiladores o a la teoría de las asignaturas; esto podría ayudar a los alumnos a aumentar su interés por la asignatura y repasar la teoría de manera inconsciente al navegar por el sistema.

8 Bibliografía

[1] Joffre Carriel O. (2020). “Diseño web | Definición e historia” [Online]. Available: https://webcorp.ec/diseno-web-definicion-historia

[2] Miguel Rodríguez (2020). “Internet Explorer” [Online]. Available: https://www.seoenmexico.com.mx/blog/internet-explorer

[3] Editorial Etecé (19 de noviembre, 2023). “Web 2.0” [Online]. Available: https://concepto.de/web-2-0/

[4] Blogger. “Blogger” [Online]. Available: https://www.blogger.com/about/ [Último acceso: 14 Abril 2024]

[5] Telefónica. “¿Qué es la Web 3.0 y cuáles son sus características?” [Online]. Available: https://www.telefonica.com/es/sala-comunicacion/blog/que-es-la-web-3-0-y-cuales-son-sus-caracteristicas/ [Último acceso: 14 Abril 2024]

[6] MDN. “HTML: Lenguaje de etiquetas de hipertexto” [Online]. Available: https://developer.mozilla.org/es/docs/Web/HTML [Último acceso: 15 Abril 2024]

[7] Manual Web. “Historia HTML. Inicios” [Online]. Available: https://www.manualweb.net/html/historia-html-inicios/ [Último acceso: 15 Abril 2024]

[8] W3Schools. “HTML history” [Online]. Available: www.w3schools.in/html/history [Último acceso: 15 Abril 2024]

[9] Soluciones Inába. (18 de mayo, 2023) “Versiones de HTML” [Online]. Available: www.inabaweb.com/versiones-de-html-desde-html-1-0-hasta-html5

[10] Anupama Raj. (30 de octubre, 2023) “History of CSS: The Evolution of Web Design” [Online]. Available: https://www.almabetter.com/bytes/articles/history-of-css

[11] Amy E. Hissom, (26 de septiembre, 2011) “History of HTML and CSS” [Online]. Available: https://amyhissom.com/HTML5-CSS3/history.html

[12] MDN, “El modelo de caja” [Online]. Available: https://developer.mozilla.org/es/docs/Learn/CSS/Building_blocks/The_box_model [Último acceso: 16 Abril 2024]

[13] The PHP group, “History of PHP” [Online]. Available: https://www.php.net/manual/en/history.php.php [Último acceso: 16 Abril 2024]

[14] Tutorialspoint, “PHP - History” [Online]. Available: https://www.tutorialspoint.com/php/php_history.htm [Último acceso: 16 Abril 2024]

[15] Javier Gutiérrez Chamorro, “PHP 7 y lo que pasó con PHP 6” [Online]. Available: https://www.javiergutierrezchamorro.com/php-7-y-lo-que-paso-con-php-6/

[16] W3Schools, “SQL History” [Online]. Available: https://www.w3schools.in/sql/history

[17] A. V. Aho, M. S. Lam, R. Sethi y J. D. Ullman, Compiladores: principios, técnicas y herramientas, 2 ed., Pearson Educación, 2008.

[18] «IEEE STANDARD 29148-2011 - Systems and software engineering — Lifecycle,» 2011.

[19] Juan Manuel Cueva Lovelle, [Online]. Available: https://reflection.uniovi.es/ortin/publications/automata.pdf [Último acceso: 23 de mayo de 2024]

[20] DLSIIS, "Analizador Sintáctico Ascendente" [Online]. Available: https://dlsiis.fi.upm.es/procesadores/Documentos/AStLR.pdf

[21] DLSIIS, "First y Follow" [Online]. Available: https://dlsiis.fi.upm.es/procesadores/Documentos/FirstFollow.pdf

[22] "Objetivos y metas de desarrollo sostenible" [Online]. Available: https://www.un.org/sustainabledevelopment/es/objetivos-de-desarrollo-sostenible/

Este documento esta firmado por

	Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Sun Jun 30 00:16:52 CEST 2024
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)