Diseño e Implementación de un Motor de Análisis Léxico Genérico de código fuente para VPL

Proyecto realizado por Parreño Barbuzano, David

Tutorizado por:

Rodríguez del Pino, Juan Carlos

Trabajo Fin de Grado del Grado de Ingeniería Informática

Diciembre de 2022

Agradecimientos

Este proyecto no habría sido posible sin mi tutor, Juan Carlos Rodríguez del Pino, que siempre sacó tiempo y esfuerzo para resolver dudas, dar consejos, y apoyarme durante el desarrollo de la aplicación, manual y memoria.

También agradezco enormemente el apoyo y motivación que me dió mi familia, en especial mis padres, que siempre estuvieron ahí para darme los ánimos que necesitaba, incluso en los peores momentos.

Resumen

Virtual Programming Lab (VPL) es un módulo para la plataforma de aprendizaje Moodle que ofrece utilidades desde dicho entorno educativo para la administración y desarrollo de tareas de programación. De entre las muchas características que tiene, una de las más destacables es la búsqueda de similitud entre ficheros de código fuente. Actualmente, VPL proporciona esta funcionalidad a partir de analizadores léxicos basados en autómatas específicos.

Este proyecto pretende reducir los altos costos de creación de los analizadores mediante la adaptación de un motor genérico de interpretación de gramáticas que permita crear analizadores para cada lenguaje de programación de una forma más sencilla y fiable.

El entorno de programación en el que se llevará a cabo este trabajo de fin de título utilizará herramientas orientadas al desarrollo de aplicaciones, a la realización de pruebas unitarias y pruebas de comportamiento de la aplicación.

Abstract

Virtual Programming Lab (VPL) is a module for the Moodle learning platform that offers utilities from this educational environment for the administration and development of programming tasks. Among the many features it has, one of the most notable is the search for similarity between source code files. Currently, VPL provides this functionality from lexical analyzers based on specific automata.

This project aims to reduce the high costs of creating parsers by adapting a generic grammar interpretation engine that allows creating parsers for each programming language in a simpler and more reliable way.

The programming environment in which this final degree project will be carried out will use tools oriented to at application development, unit testing and application behavior testing.

Contenido

Índice de figuras

Índice de tablas

Índice de algoritmos

Capítulo 1

Introducción

"No es raro encontrar copias de hombres de consideración, y la mayor parte de las personas, como sucede con los cuadros, tienen mayor aprecio por las copias que por los originales"

Friedrich Nietzsche

1.1. Motivación

Desde tiempos inmemoriales, el plagio ha sido un problema para muchos ámbitos, como el académico o el científico, que ha perjudicado la calidad, imagen, e incluso importancia de cientos de obras. Con la creación de Internet, y la evolución creciente de las nuevas tecnologías informáticas, las copias ilícitas en el software son cada vez más frecuentes y difíciles de detectar. No obstante, si hay algo que caracteriza al ser humano, es su capacidad de encontrar soluciones a un problema. Con el avance de la computación, y el desarrollo de algoritmos de búsqueda de similaridad eficientes, han surgido múltiples sistemas capaces de detectar plagios en cualquier publicación o trabajo original.

La gravedad del plagio, y mi pasión por la computación, provocó en mí la necesidad de contribuir de alguna forma en la solución de este problema. Supe con certeza cómo podría colaborar en el momento en el que descubrí que VPL, la aplicación que he usado en muchas ocasiones durante mi recorrido universitario para las tareas de programación, también dispone

de una funcionalidad, conocida como búsqueda de similaridad, con la que se puede detectar copias en el código fuente de varios estudiantes. El inconveniente de este sistema es que no es fácilmente extensible, lo que significa que, añadir en la búsqueda de similaridad soporte para nuevos lenguajes de programación, supone un alto coste del tiempo y el esfuerzo requerido por los desarrolladores.

La motivación de este proyecto surge de las necesidades actuales que tiene el módulo Virtual Programming Lab (VPL) de mejorar el proceso de tokenización (procesamiento léxico en el que se obtiene una serie de unidades léxicas llamadas ”tokens”) de su búsqueda de similaridad (comparación de dos o más ficheros escritos en un lenguaje con el fin de detectar plagio) para así poder disponer de más lenguajes de programación en los que se pueda detectar plagio en el código fuente de las actividades de programación de varios estudiantes.

1.2. Descripción del problema

Según Niklas Johansson y Anton Löfgren, la extensibilidad es ”la capacidad de un sistema de poder extenderse con nuevas funcionalidades con efectos mínimos o nulos en su estructura interna y flujo de datos” [1]. En la historia de la programación, se han visto casos de proyectos software que, por no integrar la extensibilidad en su diseño, acaban siendo abandonados por sus creadores. Este suceso se debe principalmente a que, como el código no se ha desarrollado considerando la extensibilidad, un simple cambio puede traer consigo mucho esfuerzo por parte del equipo de desarrollo, además de altos costes económicos que hacen imposible el mantenimiento del software. Asimismo, la no inclusión de la extensibilidad en el código fuente no solo dificulta la evolución del mismo, sino que también aumenta el riesgo de que aparezcan errores durante el desarrollo del software.

Si bien la extensibilidad se aplica en muchos componentes que conforman a VPL, no está presente o no se explota adecuadamente en algunos de ellos. Uno de estos casos problemáticos se encuentra en su búsqueda de la similaridad, concretamente en una de las tareas que se llevan a cabo en su fase de troceado. Dicho proceso se conoce como análisis léxico (tokenización), que, en el ámbito de la computación, se define como la transformación del texto legible en tokens, es decir, en ”segmentos de un texto identificados como unidades significativas para el

análisis del texto" [2]. El principal inconveniente de la tokenización de VPL es que está compuesto por analizadores léxicos basados en autómatas específicos, lo cual significa que, para cada lenguaje de programación, se necesita desarrollar en el código fuente un tokenizador independiente. Las desventajas de esta limitación, unido a la falta de documentación, dificulta enormente la incorporación de nuevos lenguajes, así como el mantenimiento y mejora de los ya disponibles en el propio módulo.

1.3. Objetivos

El presente trabajo de fin de título pretende abordar el problema de la tokenización para la búsqueda de similaridad en el módulo de VPL de Moodle a partir del desarrollo de un motor genérico de análisis léxico, basado en máquinas de estados. Dicho motor sigue las reglas especificadas en un fichero JSON que determinan como hacer el análisis léxico (tokenización). Esta separación (reglas/código de análisis) permite realizar el análisis léxico de diferentes lenguajes de programación definiendo las reglas correspondientes de cada uno. Gracias a este sistema, se consigue que el código tenga una capa de abstracción potencial que separa la codificación del motor de la definición de las reglas que especifican el léxico.

Además del desarrollo del motor, con este proyecto se propone una API con la que el usuario pueda declarar el conjunto de reglas que el motor utiliza en su ejecución. Asimismo, la API que se ofrece se ha documentado en un manual de usuario en la que se incluye toda la información necesaria sobre el motor de análisis léxico.

Finalmente, la siguiente figura representa con un ejemplo conceptual los resultados que el motor de análisis léxico devolverá tras la tokenización de un fichero codificado en el lenguaje de programación C.

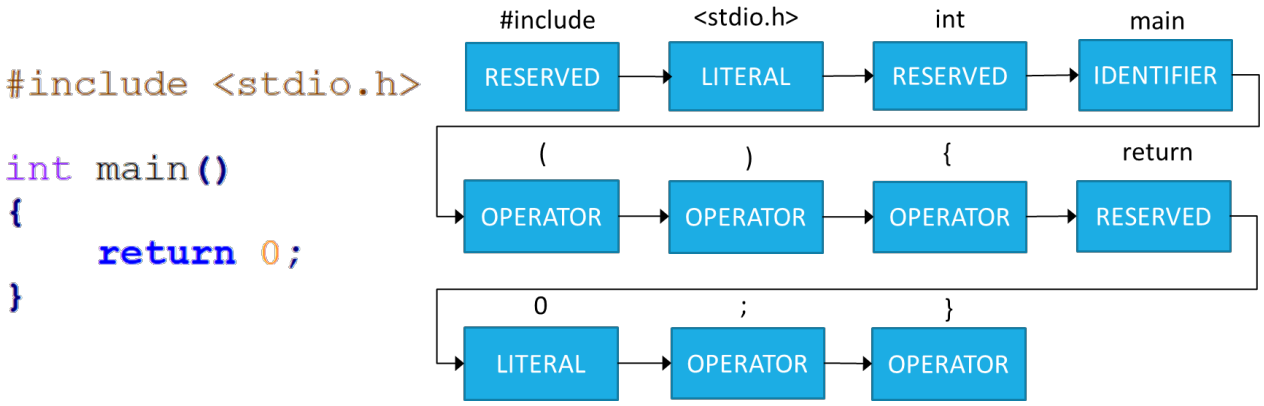

Figura 1.1: Ejemplo del funcionamiento de VPLT

1.4. Metodología

El código fuente del motor de análisis léxico planteado en este proyecto, es una adaptación del tokenizador del editor Ace [3], que integra el funcionamiento del editor en base a las restricciones impuestas por el módulo de Moodle VPL. A continuación, se explicará con mayor detalle el procedimiento que se realizó durante el desarrollo de este proyecto.

Antes de comenzar con la codificación, se llevó a cabo un estudio inicial tanto del software del editor de código responsable del proceso de tokenización, como del formato de los ficheros de reglas empleados por el tokenizador. Con este examen, se descartaron las partes que no eran interesantes para el presente proyecto, o que podían llegar a dificultar el desarrollo del motor de análisis léxico.

Tras el estudio preliminar del tokenizador, se especificó el diseño de los ficheros de reglas que emplearía el motor durante su ejecución. En este caso, dichos ficheros están definidos a partir del lenguaje JSON, y tienen una estructura similar a la de los ficheros de reglas de Ace.

Una vez que se definió el diseño de los ficheros de reglas, se empezó con el desarrollo del motor de análisis léxico. Esta fase intermedia fue la que más tiempo requirió, pues no sólo se implementó el código fuente del motor, sino que también se añadió un gran lote de pruebas que se encargan de validar el funcionamiento de la aplicación.

Finalmente, el presente trabajo de fin de título termina con la redacción de esta memoria

y el manual de usuario del motor de análisis léxico. Cabe mencionar que, próximamente, el manual de usuario estará disponible en la página web oficial de VPL. Asimismo, el código del motor se incorporará en una futura versión de VPL, mejorando así las posibilidades de funcionamiento del módulo para las miles de instalaciones de Moodle que lo usan en todo el mundo.

Capítulo 2

Estado del arte

"Antes que toda otra cosa la preparación es la clave para el éxito."

Alexander Graham Bell

La información que se recoge en el presente capítulo propone, por un lado, una aproximación conceptual de las técnicas, métodos, y teorías fundamentales en los que se inspira la aplicación desarrollada, y por otro lado, una lista de los antecedentes de este proyecto que ofrecen otras alternativas a la que se explica en este trabajo de fin de título.

2.1. Aproximación conceptual

En este apartado, se introduce una aproximación general de los conceptos básicos que fundamentan el diseño del motor de análisis léxico propuesto en este trabajo fin de titulo. Los conceptos que se tratan aquí son esenciales para la comprensión del funcionamiento del motor, así como de otros aspectos tales como su estructura o modelo en el que se sustenta.

2.1.1. Procesamiento léxico

Según el diccionario de la Real Academia Española, el léxico se define como el "vocabulario, conjunto de las palabras de un idioma, o de las que pertenecen al uso de una región, a una

actividad determinada, a un campo semántico dado, etc" [4]. Los vocablos de una lengua se registran en lo que se conoce como diccionarios, y la lexicografía es la disciplina especializada en la creación y análisis de este tipo de documentos. En la Ingeniería Informática, el léxico se ha tratado como un concepto que, junto a la sintaxis y la semántica, ha servido de inspiración para el desarrollo de sistemas de procesamiento de lenguajes de programación. En la actualidad, estos sistemas se organizan en dos grupos diferentes conocidos como compiladores e intérpretes. En este caso, el procesamiento léxico es un proceso de los sistemas de procesamiento de lenguajes de programación que se utiliza generalmente como fase inicial

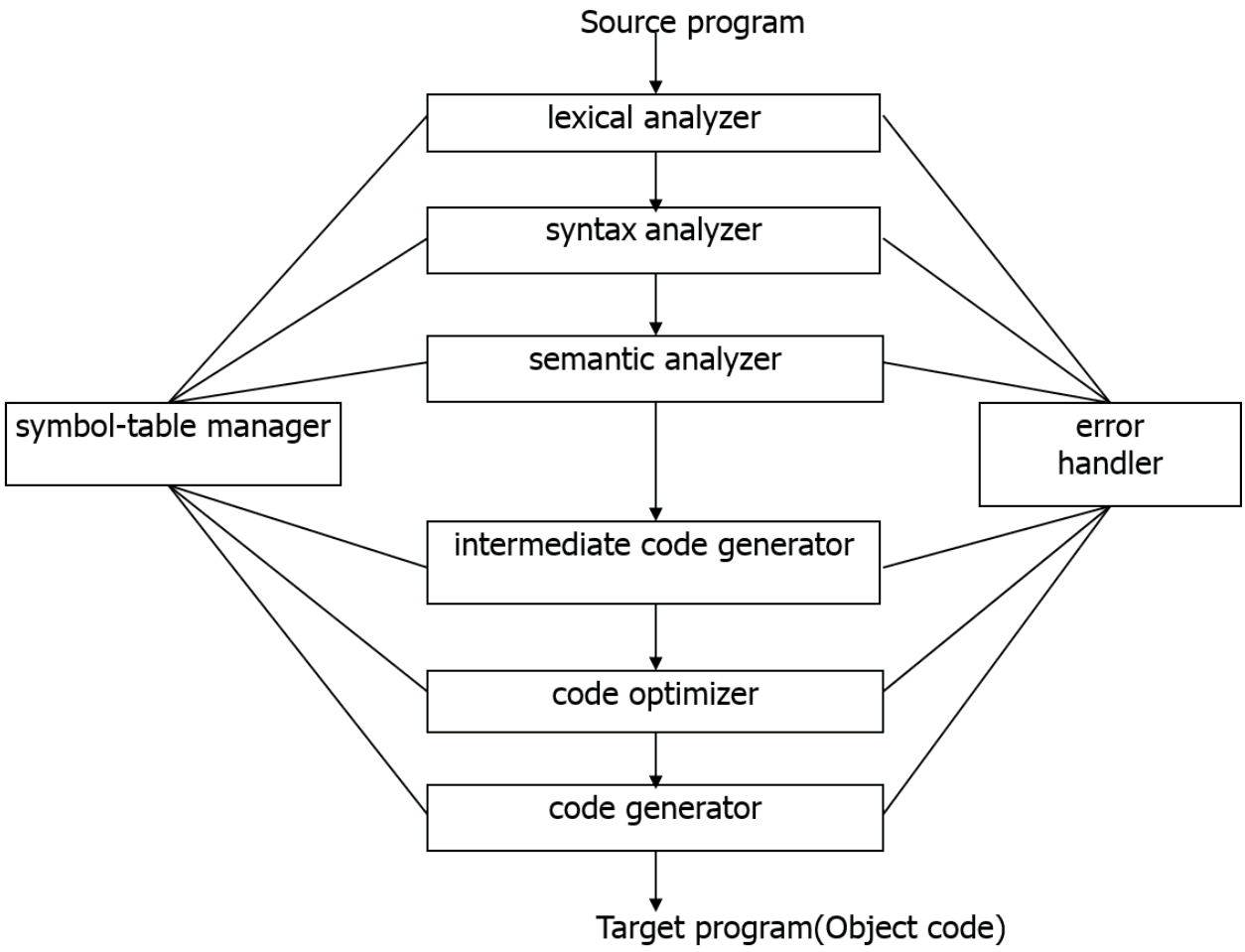

Figura 2.1: Fases de funcionamiento de los compiladores [5]

El procesamiento léxico, también conocido como tokenización, o análisis léxico, es el proceso en el que, partiendo de un fichero proporcionado en la entrada y del léxico de un lenguaje, se

genera una serie de símbolos o unidades mínimas que representan el léxico de cada fragmento del texto. A este tipo de símbolos se les conoce como ”tokens”, y pueden entenderse como ”secuencias de caracteres que tienen un significado colectivo” [6].

Como se comentó en el anterior párrafo, los analizadores léxicos ejercen su función en base a la definición léxica del lenguaje de programación que se considera en la tokenización. En general, la mayor parte de los analizadores léxicos representan el léxico por medio de reglas. Estas reglas determinan las condiciones que han de cumplirse para que un fragmento de texto esté asociado con un tipo de token determinado. En general, dichas condiciones suelen representarse por medio de unos patrones conocidos como expresiones regulares. En la figura 2.2, se presenta un modelo de caja negra del funcionamiento del procesamiento léxico.

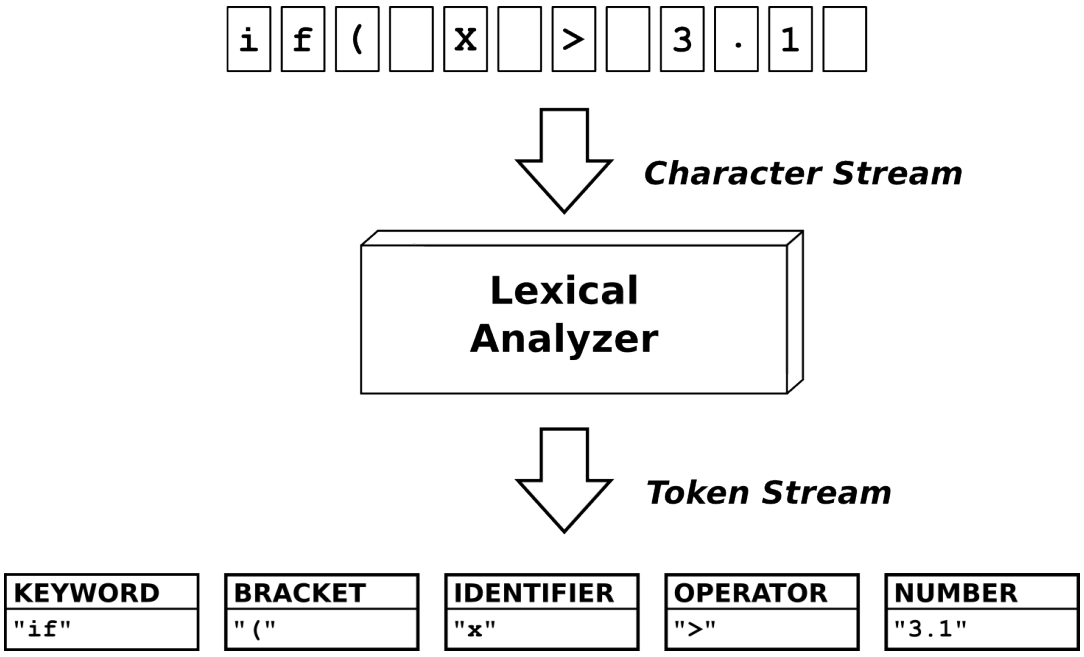

Figura 2.2: Funcionamiento conceptual de los analizadores léxicos [7]

En términos generales, el procesamiento léxico puede entenderse como la búsqueda de tokens por medio de reglas que usan expresiones regulares y que se recorren hasta que todos los fragmentos de texto hayan sido asociados con un token. El recorrido de dichas reglas depende completamente del algoritmo de búsqueda y estructura de datos empleada. Si bien existen muchas alternativas, una de las maneras de organizar las reglas de los analizadores léxicos

es mediante máquinas de estados, concretamente los autómatas finitos deterministas o no deterministas.

2.1.2. Teoría de autómatas

En la Ingeniería Informática, se entiende la teoría de autómatas como la agrupación de los modelos matemáticos, conocidos como autómatas, que se encargan de formalizar a manera de algoritmos las soluciones con los que se abordan los problemas computacionales. La teoría de autómatas suele utilizarse en la teoría del lenguaje formal como un recurso con el que proporcionar la representación finita de la gramática de un lenguaje formal. En la actualidad, esta teoría tiene aplicaciones en una gran variedad de ámbitos, tales como el procesamiento de texto, la compilación, la verificación formal, la inteligencia artificial, entre otros.

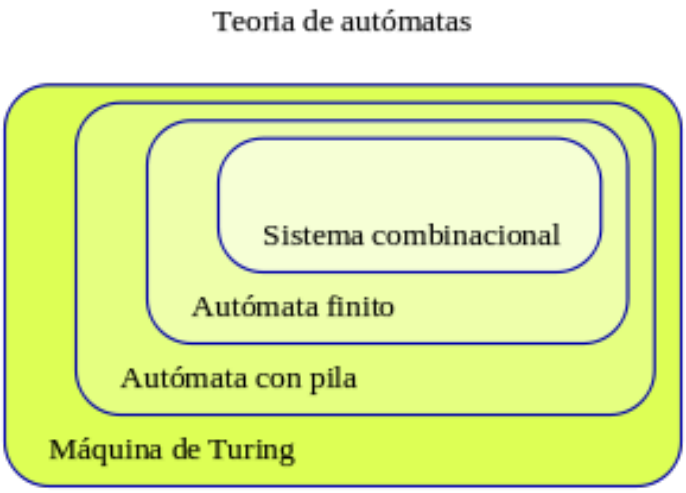

Figura 2.3: Tipos de máquinas de la teoría de autómatas [8]

Desde un punto de vista general, los autómatas se definen como máquinas abstractas que llevan a cabo un proceso o algoritmo de manera independiente y automática. Su estructura se conforma por una serie de estados, que representan cada fase o situación del algoritmo, y transiciones, que indican las conexiones o relaciones de los estados, y que se activan según el resultado de una función matemática concocida como función de transición. Con estos componentes, los autómatas pueden representar fácilmente el flujo de cualquier tipo de algo-

ritmo. Cabe mencionar que, generalmente, los autómatas se representan gráficamente como un grafo en el que los estados son círculos, y las transiciones flechas de una dirección que conectan un estado con otro. En la siguiente figura, se puede observar un ejemplo gráfico de un tipo de autómata, conocido como autómata finito determinista, que define un lenguaje formal en el que sólo se admite la codificación binaria.

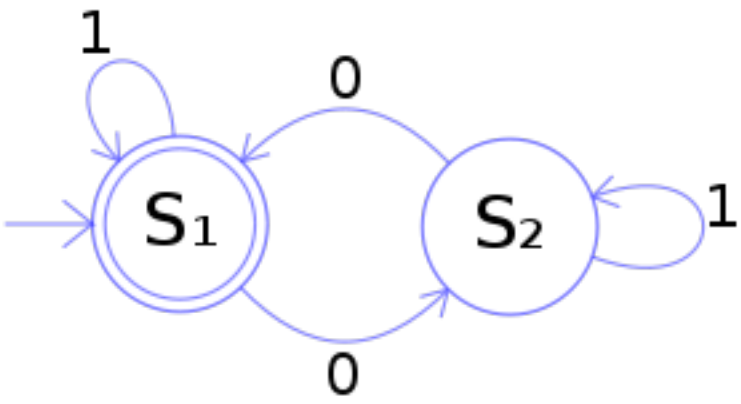

Figura 2.4: Ejemplo de un autómata de código binario [8]

Tal y como se ilustra en la figura 2.3, los modelos que se plantean en la teoría de autómatas se distribuyen en cuatro grandes tipos. Las clases que se proponen en esta clasificación se han organizado considerando aspectos tales como la capacidad de procesamiento, estructura, y diseño de los autómatas que los conforman. En los siguientes apartados, se explican cada uno de estos tipos de autómatas sin entrar en demasiados detalles.

2.1.2.1. Máquinas de estado finitos

Las máquinas de estado finitos, conocidas también por el nombre de autómatas finitos, son modelos computacionales cuya estructura se conforma de una serie de estados finitos, un alfabeto, una función de transición, y unas transiciones. En este tipo de sistemas, es estrictamente necesario que entre los estados exista un estado inicial, y un conjunto de estados finales. Con estos componentes, el autómata finito tratará durante su funcionamiento de trazar un recorrido desde el estado inicial hasta alguno de los estados finales, recorriendo en el proceso las transiciones de los estados por los que pasa.

Los autómatas finitos se clasifican en dos grupos bien diferenciados: autómatas finitos deterministas (AFD), y los no deterministas (AFND). Los primeros se caracterizan por tener en cada estado como máximo una transición por cada símbolo del alfabeto especificado, mientras que los segundos permiten que en un estado existan varias transiciones para el mismo símbolo. En la siguiente figura, se observa una comparación de ambos tipos de autómatas.

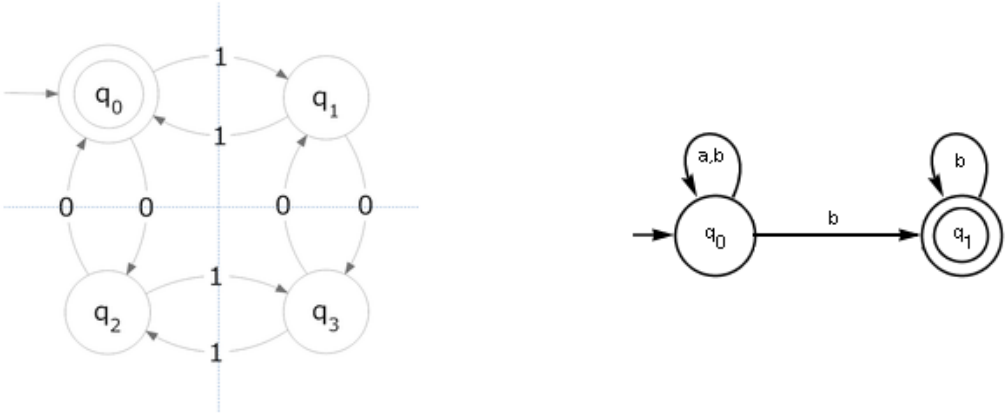

Figura 2.5: Comparativa de AFD y AFND [9]

2.1.2.2. Autómatas con pila

Los autómatas con pila son una clase particular de autómata finito que incorpora en su especificación una memoria llamada pila, que se emplea para el almacenaje y extracción de los símbolos procesados por el autómata. Este nuevo componente posibilita la definición de autómatas cuyas funciones de transición se formulan a partir del símbolo actual y con los que anteriormente se procesaron. Asimismo, los autómatas con pila admiten, gracias a su memoria, lenguajes formales más complejos que los de los autómatas finitos.

Al igual que ocurre con los autómatas que extiende, los autómatas con pila se clasifican en autómatas deterministas (APD), y no deterministas (APND). Las definiciones de cada tipo son equivalentes a los de los finitos, salvo en el hecho de que ambos tienen una pila.

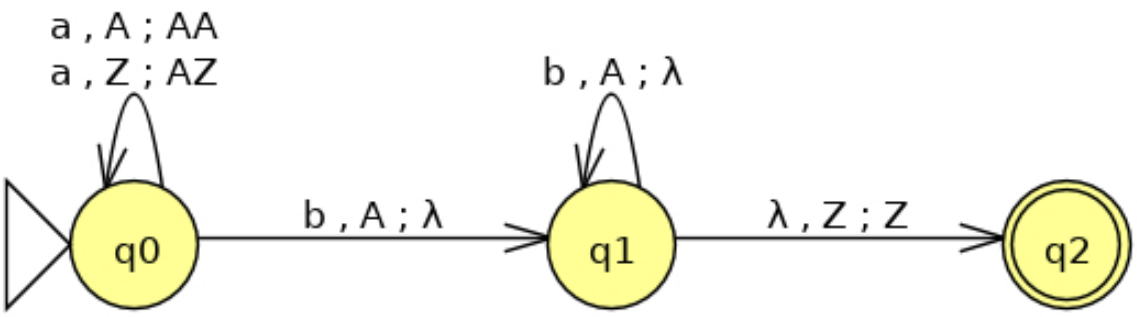

Figura 2.6: Ejemplo de un autómata con pila [10]

2.1.2.3. Máquina de Turing

Una máquina de Turing es una sistema abstracto que modeliza el concepto de algoritmo a partir de un cabezal conocido como unidad de control que realiza operaciones de lectura y escritura sobre una cinta infinita que almacena símbolos. Como ocurre con los otros tipos de autómatas, la máquina de Turing dispone de un conjunto de estados finitos, un alfabeto, y una función de transición. En este caso, las transiciones se entienden como las acciones del cabezal en las que se lee o escribe en la cinta un símbolo del alfabeto. Asimismo, el cambio de un estado se realiza siempre que el cabezal haga una acción y se mueva sobre la cinta hacia la izquierda o derecha.

A diferencia del resto, la máquina de Turing es un autómata con un alto poder computacional, que puede realizar cualquier cálculo que pueda ejecutarse en un computador. En la actualidad, la máquina de Turing, junto a otros sistemas, ha sido uno de los pilares fundamentales en los que sustenta el diseño y desarrollo de los computadores.

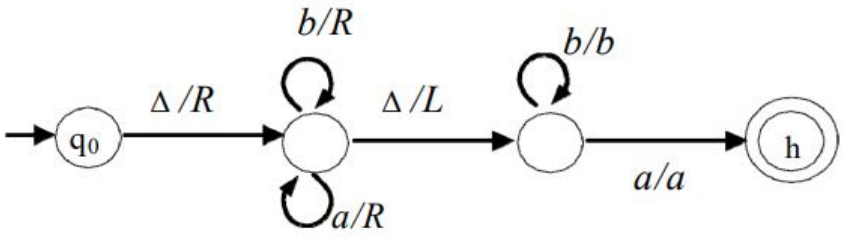

Figura 2.7: Ejemplo de una máquina de Turing [11]

```
funcion maquina_turing(C, UC) donde C es la cinta, y UC el cabezal
    asociar UC con el estado inicial Q_ini
    apuntar UC a la casilla Sq de C

    mientras el estado actual Q no sea el fin Q_fin
        leer el caracter S de Sq
        cambiar al estado Q', o mantenerse en Q
        escribir el caracter S' en Sq, o mantener S
        mover UC hacia la casilla izquierda o derecha
        Sq = casilla izquierda Sq_i | casilla derecha Sq_d
    fin mientras
fin funcion
```

Algoritmo 2.1: Funcionamiento de la máquina de Turing

2.1.3. Búsqueda de similitud de VPL

Toda aplicación, orientada al ámbito académico, que disponga de herramientas que faciliten la evaluación de tareas o actividades de aprendizaje, se enfrenta de manera directa al problema del plagio. Como ya se ha comentado anteriormente, la resolución de esta vulnerabilidad requiere de la construcción de sistemas informáticos que detecten las copias ilícitas mediante una serie de técnicas y algoritmos. A priori, el proceso de comparación del contenido de varios ficheros es una solución trivial que no presenta ninguna complejidad. El verdadero problema está en que, en la gran mayoría de los casos, el plagio se consigue ocultar a partir de modificaciones en el contenido que no aportan ningún tipo de información nueva. Todo esto provoca que los algoritmos que se desarrollen para la detección del plagio incorporen mecanismos con los que ignorar los cambios no significativos. En el caso del módulo VPL, este mecanismo se conoce como búsqueda de similitud.

La búsqueda de similitud es una funcionalidad del módulo VPL que permite detectar copias ilícitas en el código fuente de las entregas de los alumnos. Se trata de un ”sistema en el que las comparaciones no se ven afectadas por las posibles alteraciones que se realicen” [12].

Según Juan Carlos Rodriguez del Pino, creador del módulo VPL, la búsqueda de similitud es un proceso complejo que lleva a cabo la comparación del contenido de los ficheros con la mínima información posible de los mismos, es decir, con lo que se conoce en el módulo VPL como ”firma del fichero”.

En VPL, la búsqueda de similitud está dividida en tres pasos diferentes: troceado, comparación, y agrupamientos. ”El troceado es un paso previo en el proceso para obtener una firma normalizada de cada uno de los ficheros. La fase de comparación obtendrá la similitud entre todos los ficheros. Después de la comparación, el paso agrupamientos encontrará los grupos de ficheros similares” [12].

Figura 2.8: Fases de la búsqueda de similitud [12]

Tal y como se puede observar en la figura 2.8, las fases que se realizan en la búsqueda de similitud está compuestas a su vez por un conjunto de tareas. Para el presente trabajo de fin de título, la única que interesa es la primera de la fase de troceado, es decir, el análisis léxico, por ser el componente que el motor de análisis léxico remplaza. El análisis léxico en VPL se encarga de obtener, a partir del léxico de un lenguaje, un conjunto de símbolos. ”Cada símbolo representa un elemento del lenguaje de programación como: comentario, identificador, palabra reservada, operador, literal, etc” [12].

2.2. Antecedentes

En el siguiente apartado, se exploran algunos de los generadores de análisis léxico que se desarrollaron antes del motor propuesto en este trabajo de fin de título. Los antecedentes que se documentan aquí han servido como punto de inspiración, fundamentación, y estudio de muchos otros proyectos actuales. Por todo esto, la información concentrada en este apartado ilustra aspectos relevantes sobre el diseño, estructura, y funcionamiento de dichos sistemas.

Si bien en el mercado actual existen muchas más alternativas de generadores léxicos, en este apartado se ha optado por incluir sólo algunas de las más representativas. Cada una de estas propuestas se diferencia del resto en su estrategia, modelo, aportación, entre otros aspectos.

2.2.1. Generador Lex

Lex es un programa cuyo objetivo principal consiste en la generación de sistemas capaces de llevar a cabo el procesamiento léxico de un conjunto de caracteres proporcionados en la entrada. Se trata de un generador escrito en C cuyos analizadores léxicos están conformados por expresiones que se utilizan para la identificación léxica en un texto. Cabe mencionar que, ”el reconocimiento de las expresiones se realiza mediante un autómata finito determinista generado por Lex” [13].

Los sistemas que se generan en Lex requieren de la especificación de un conjunto de expresiones regulares que se emplean como patrones de búsqueda que posibilitan la asociación de los caracteres de entrada con el léxico del lenguaje. Las expresiones pueden ir acompañadas de código que se encarga de llevar a cabo cierta acción en el momento en el que una expresión coincida con la entrada.

Actualmente, existen en el mercado múltiples versiones de Lex que amplian las funcionalidades del generador u ofrecen alternativas para otros lenguajes de programación distintos a C. De entre todas estas versiones, Flex es una de las más conocidas, ya que es un programa gratuito que lleva a cabo la misma función que su predecesor Lex.

2.2.2. Generador RE/flex

Aunque Flex está diseñado para desarrollar sistemas en C, dispone de una opción que permite que los analizadores léxicos puedan producirse con el lenguaje C++. Sin embargo, el generador de análisis léxico con C++ "tiene errores y no funciona muy bien" [14]. Debido a esto, en el mercado han surgido múltiples alternativas a Flex que ofrecen un mejor soporte para C++. En este caso, se presenta el generador de análisis léxico RE/flex.

En pocas palabras, RE/flex es un programa que "ofrece muchas funciones nuevas y útiles en comparación con Flex, incluida la compatibilidad con Unicode, anclajes de sangría, límites de palabras, cuantificadores perezosos (no codiciosos, repeticiones perezosas) y opciones de ajuste del rendimiento. El software RE/flex también incluye una biblioteca de expresiones regulares independiente muy rápida para C++" [15].

2.2.3. Generador re2c

Al igual que ocurre en Lex, re2c es un generador para C y Go que permite la construcción de analizadores léxicos por medio de expresiones regulares que se modelan en autómatas finitos deterministas. A grandes rasgos, se trata de un generador fundamentado en las mismas bases que Flex. Lo que hace diferente a re2c del resto es su optimización y eficiencia: los autómatas que se crean se codifican no mediante una tabla de expresiones, sino con la definición de condiciones y comparaciones que marcan el flujo del sistema.

Tal y como dice el autor, re2c "no solo produce escáneres que son más rápidos que los creados por otros generadores de escáneres, sino que, sorprendentemente, también suelen ser más pequeños" [16]. Parar lograr esto, re2c incorpora una serie de métodos de optimización para la simplificación de los autómatas generados, así como la aplicación de mecanismos en el código que proporcionen mayor rapidez y menor peso.

Capítulo 3

Competencias y aportaciones

En este tercer capítulo, se explican cada una de las competencias específicas cubiertas, así como las contribuciones de este trabajo de fin de título para la plataforma virtual de aprendizaje Moodle, y en general, para el ámbito de la educación y la computación. La información se ha organizado en dos apartados, uno para las competencias y otro para las aportaciones del trabajo, y se han redactado con el objetivo de, por un lado, demostrar al lector el perfil profesional del autor, y por otro lado, el gran valor que tiene este trabajo para la comunidad de Moodle y, concretamente, para los usuarios de VPL.

3.1. Competencias

Para este trabajo de fin de título, se alcanzaron dos competencias específicas, una de ellas común para todos los campos de la Informática, la CII01, y la otra para la Computación, la CP02. En los siguientes apartados, se explica cada una de estas competencias, y se justifica además su relación con este trabajo.

3.1.1. Código CII01

Según el reglamento de la ULPGC [17], CII01 se define como la ”capacidad para diseñar, desarrollar, seleccionar y evaluar aplicaciones y sistemas informáticos, asegurando su fiabilidad, seguridad y calidad, conforme a principios éticos y a la legislación y normativa vigente”.

El desarrollo del proyecto supuso el desarrollo de una aplicación informática. Mediante las herramientas PHPUnit y Behat, el código fuente pudo validarse a partir de pruebas unitarias y de comportamiento, garantizando de esta forma una aplicación segura, robusta, y eficiente. Todo esto demuestra el hecho de que CII01 es una de las competencias que se abordan en este proyecto.

3.1.2. Código CP02

Tal y como se dice en el reglamento de la ULPGC, CP02 es la "capacidad para conocer los fundamentos teóricos de los lenguajes de programación y las técnicas de procesamiento léxico, sintáctico y semánticas asociadas, y saber aplicarlas para la creación, diseño y procesamiento de lenguajes.".

Como bien se ha explicado anteriormente, el objetivo principal de este trabajo es desarrollar un motor de análisis léxico que sustituya el actual sistema de tokenización de la búsqueda de similaridad de VPL. La competencia CP02 es una de las cubiertas por este proyecto porque incluye en su especificación las técnicas de procesamiento léxico, y dicha técnica es en la que se basa el funcionamiento del motor desarrollado.

3.2. Aportaciones del trabajo

En este trabajo se ofrece una alternativa al actual tokenizador del módulo VPL en el que se sustituye el procesamiento léxico independiente de cada lenguaje por un motor de análisis léxico genérico. Esta propuesta supone para VPL un antes y después para la búsqueda de similaridad, ya que el motor se ha llevado a cabo considerando la extensibilidad.

Desde un punto de vista funcional, el motor propuesto no supone ninguna diferencia con su predecesor: en pos de evitar incompatibilidades con los otros componentes de la búsqueda de similaridad, los resultados generados son los mismos que los que se obtienen mediante los tokenizadores de VPL. Sin embargo, con la inclusión de este nuevo sistema, se consigue simplificar enormemente la adición o modificación de los analizadores léxicos, que ahora no se codifican de forma independiente, sino que se declaran reglas que definen el léxico.

Finalmente, el motor de análisis léxico planteado en este trabajo también es un producto de valor para la plataforma Moodle o para otros programas similares, ya que proporciona a la comunidad los cimientos necesarios para la construcción de nuevos sistemas informáticos que perfeccionen el funcionamiento del motor o lo adapten para la resolución de otros problemas.

Capítulo 4

Diseño del motor

"La mejor estructura no garantizará los resultados ni el rendimiento. Pero la estructura equivocada es una garantía de fracaso."

Peter Ferdinand Drucker

En este capítulo, se recoge información detallada sobre el diseño del motor de análisis léxico propuesto en el presente trabajo de fin de título. En la explicación, se incluyen apartados que documentan aspectos tales como el funcionamiento del motor, y el formato de los ficheros de reglas.

4.1. Introducción

Como se ha explicado en capítulos anteriores, el motor de análisis léxico, que en este proyecto se conoce como Virtual Programming Lab Tokenizer (VPLT), es un sistema informático cuya finalidad principal es la obtención, por medio del procesamiento léxico, de los tokens asociados con un fichero proporcionado en la entrada. Para lograr este objetivo, es necesario que el motor disponga de una serie de reglas en las que se defina el léxico del lenguaje en el que está escrito dicho fichero. Sin embargo, la especificación de las reglas no es sufuciente para que el motor ya pueda funcionar, pues también es necesario proponer un modelo en el que se estructuren estas reglas y se plantee de qué manera se utilizarán en el motor. En

este caso, el motor VPLT utiliza como modelo computacional un tipo de máquina de estados conocida como autómata finito determinista.

En VPLT, el léxico de un lenguaje de programación se estructura en una serie de estados únicos que se identifican y diferencian del resto por medio de un nombre descriptivo. Cada uno de estos estados dispone de un conjunto de reglas que conforman en su totalidad una parte de la definición léxica. Las reglas de un estado se procesan secuencialmente en el mismo orden en el que se declararon, lo que significa que, para que el motor utilice todas las reglas de un estado, es necesario definir primero las reglas más específicas, y luego las generales. Cabe mencionar que, la especificación de la relación entre estados se concreta mediante las propias reglas. La figura 4.1 muestra una representación simplificada del modelo planteado.

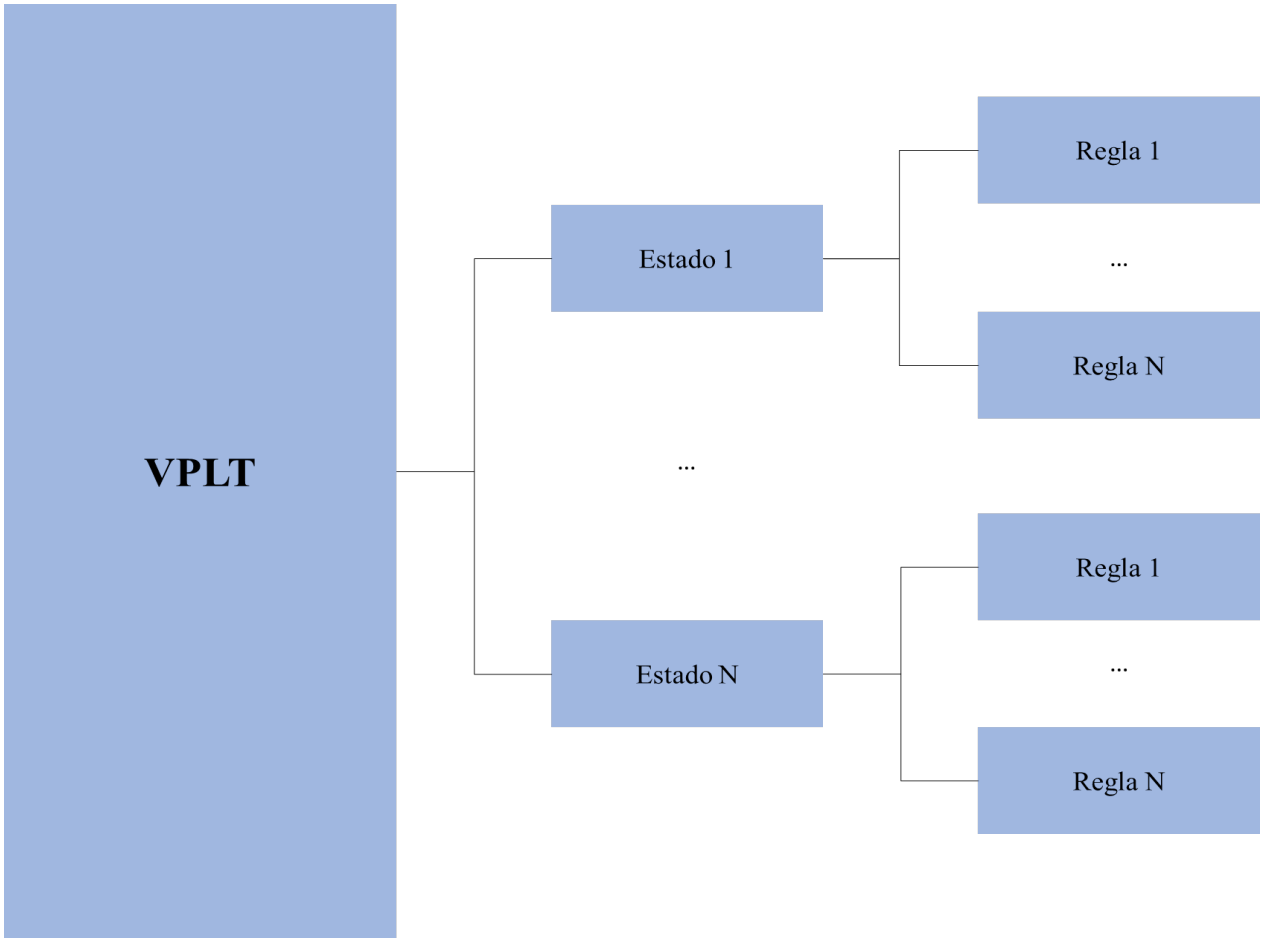

Figura 4.1: Modelo computacional del motor VPLT

4.2. Funcionamiento del motor

En este apartado se ilustra con todo lujo de detalles las diferentes fases y procesos que se llevan a cabo durante el funcionamiento del motor de análisis léxico. En este caso, la explicación se ha estructurado en tres subapartados, los cuales incluyen, por un lado, una serie de figuras que muestran el flujo de ejecución de cada fase, y por otro lado, pseudoalgoritmos de ciertas operaciones fundamentales empleadas por el motor.

4.2.1. Preprocesamiento

Antes de que el motor de análisis léxico comience con el proceso de tokenización, es primordial llevar a cabo un preprocesamiento en el que se preparen las reglas y se configure el motor a partir de una serie de opciones de configuración. Durante esta primera fase, se garantiza que tanto los ficheros de entrada como las reglas y opciones especificadas puedan emplearse para el posterior procesamiento léxico.

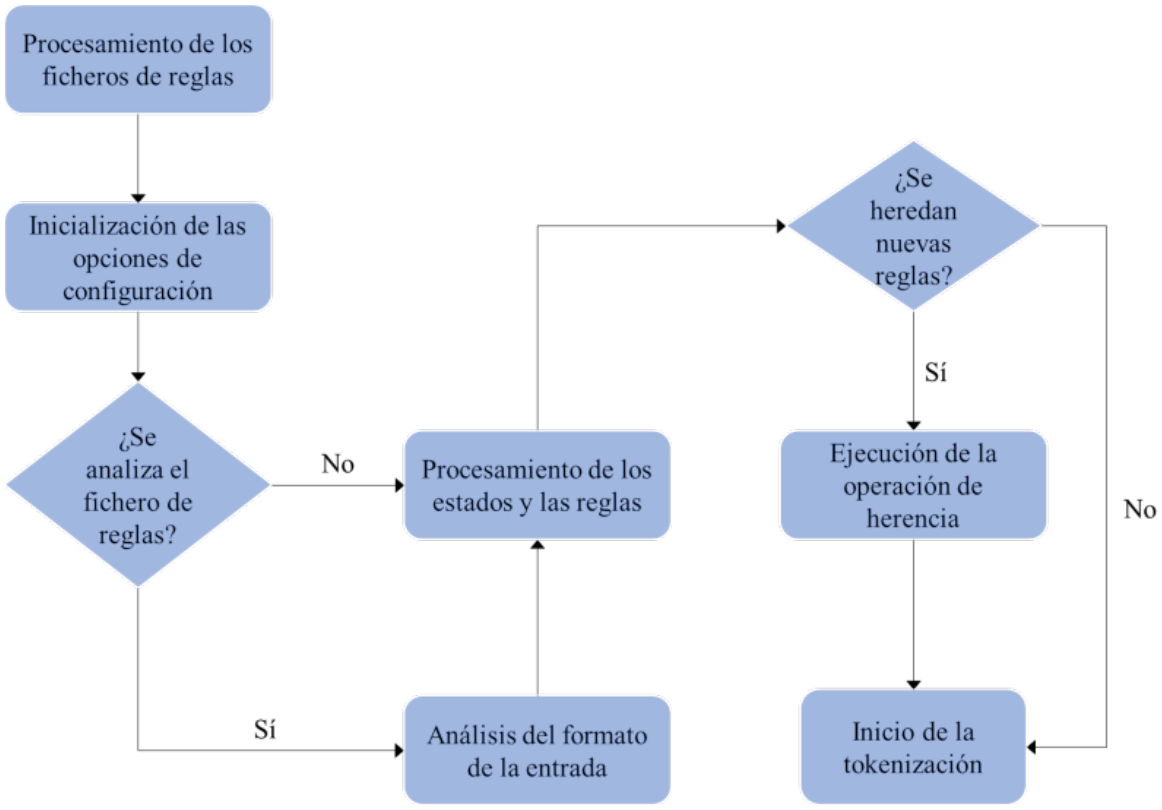

Figura 4.2: Diagrama de flujo del preprocesamiento

Como era de esperar, el preprocesamiento es una fase que está organizada en una serie de procesos o tareas que se ejecutan secuencialmente y que están ordenados de tal manera que se eviten conflictos entre ellos y se agilice la ejecución del motor. La figura 4.2 muestra un diagrama sencillo en el que se sintetiza el flujo de ejecución de la fase de preprocesamiento.

4.2.1.1. Procesamiento de los ficheros de entrada

Al comienzo del preprocesamiento, VPLT realiza una serie de pruebas con las que se asegura que la entrada suministrada, es decir, los ficheros de reglas y el archivo en el que se realiza el análisis léxico, esté, en términos generales, utilizable para la tokenización. Esencialmente, el motor verifica que los archivos proporcionados existan, que estén escritos en el formato JSON, y que sus nombres se definan a partir del sufijo "tokenizer_rules". En caso de que no se produzcan errores durante las pruebas, lo siguiente que hace el motor VPLT es procesar y almacenar los ficheros de entrada, descartando en el proceso todos los comentarios que hayan escritos en ellos.

4.2.1.2. Inicialización de opciones de configuración

La segunda tarea que se lleva a cabo durante el preprocesamiento consiste en la inicialización de las opciones de configuración que están definidas en el fichero de reglas, y que repercuten directamente en las reglas y en el comportamiento del propio motor. Si bien existen muchas opciones útiles, las más importantes son aquellas que afectan directamente al preprocesamiento y son críticas para el programa. Como era de esperar, estas opciones se inicializan antes de cualquier otra, y son "check_rules", "inherit_rules", y "override_tokens". En el apartado 4.4.3 de este documento se explica con mayor detalle cada una de estas opciones.

4.2.1.3. Análisis del formato de la entrada

En esta parte del preprocesamiento, VPLT aplica un análisis profundo en el que se comprueba la estructura y contenido del fichero de reglas proporcionado en la entrada. A partir de este estudio, el motor se asegura de que el formato de los ficheros de reglas especificado se declare adecuadamente tanto en su léxico, como en su sintaxis y semántica. Además, este sistema no sólo beneficia al motor de análisis léxico, sino también a los propios desarrolladores, que

podrán detectar fácilmente los errores de definición de los ficheros de reglas.

Sin embargo, el análisis del formato de la entrada es una tarea secundaria que puede desactivarse por medio de la opción de configuración "check_rules". El motivo de que este proceso sea opcional se entiende en el momento en el que aparecen ficheros de reglas definitivos que no vayan a editarse por mucho tiempo, y que ya se hayan comprobado previamente. En estos casos, lo conveniente sería deshabilitar el análisis del formato, pues ejecutarlo no aportaría nada y supondría un malgasto absurdo de tiempo.

4.2.1.4. Procesamiento de los estados y las reglas

Tras haber finalizado con las verificaciones iniciales, el siguiente paso del motor de análisis léxico consiste en el procesamiento de los estados y las reglas léxicas definidas en el fichero de reglas. En este momento, VPLT acondiciona la información procesada con el objetivo de agilizar ciertas operaciones usadas en la fase de tokenización como por ejemplo la búsqueda de los estados y las reglas. Es importante mencionar que el procesamiento realizado en este punto no considera la operación de herencia.

4.2.1.5. Ejecución de la operación de herencia

En VPLT, se puede modificar los estados y reglas de un fichero de forma estática o dinámica. La opción estática es la más sencilla de utilizar, ya que, para poder aplicarla, basta con editar el contenido de los ficheros de reglas. El problema de esta alternativa es que, a larga, puede llegar a ser muy díficil y costoso la modificación o comprensión de los ficheros de reglas. Con el objetivo de resolver esto, se implementó la opción dinámica, que utiliza una operación llamada "herencia" con la que se reusan los estados y reglas de otros ficheros, y que se emplea al final del preprocesamiento siempre que se haya activado con la opción "inherit_rules".

En pocas palabras, la operación de herencia es un mecanismo que se encarga de añadir en un fichero de reglas la definición léxica de otro fichero, así como de sobreescribir ciertas opciones heredables. En este proceso no se admite la herencia múltiple de varios ficheros de reglas, y tampoco se incluyen durante su ejecución aquellas reglas y estados que se repitan. En los siguientes pseudoalgoritmos, se incluye las funciones fundamentales que se emplean durante

la ejecución de la operación de herencia, y que definen la forma en la que VPLT compara los estados y las reglas de un fichero.

```
funcion heredar_estado(S1, S2) donde el estado S2 hereda de S1
    si S1 y S2 tienen el mismo nombre
        para cada regla R1 de S1
            si R1 no aparece en S2
                incluir R1 en S2 por el final
            fin si
        fin para
    fin si
fin funcion
```

Algoritmo 4.1: Herencia entre dos estados

```
funcion comparar_regla(R1, R2) donde R1 y R2 son reglas
    si R1 y R2 estan en el mismo estado S y tienen N opciones
        para cada opcion O1 de R1
            si O1 no aparece en R2
                devolver que R1 y R2 no son iguales
            fin si
        fin para

        devolver que R1 y R2 son iguales
    sino
        devolver que R1 y R2 no son iguales
    fin si
fin funcion
```

Algoritmo 4.2: Comparación de dos reglas

4.2.2. Tokenización

Una vez que se haya finalizado el preprocesamiento inicial, el siguiente paso que se hace en el motor de análisis léxico es lo que se conoce como tokenización o procesamiento léxico. La finalidad principal de este proceso es la obtención de los tokens o unidades léxicas que definen cada fragmento o partes de un texto que está escrito en un lenguaje de programación determinado. Gracias a esta funcionalidad, VPLT puede clasificar cada cadena de caracteres de un texto en base a una definición léxica, que, como ya se ha explicado en otros apartados, se especifica por medio de reglas.

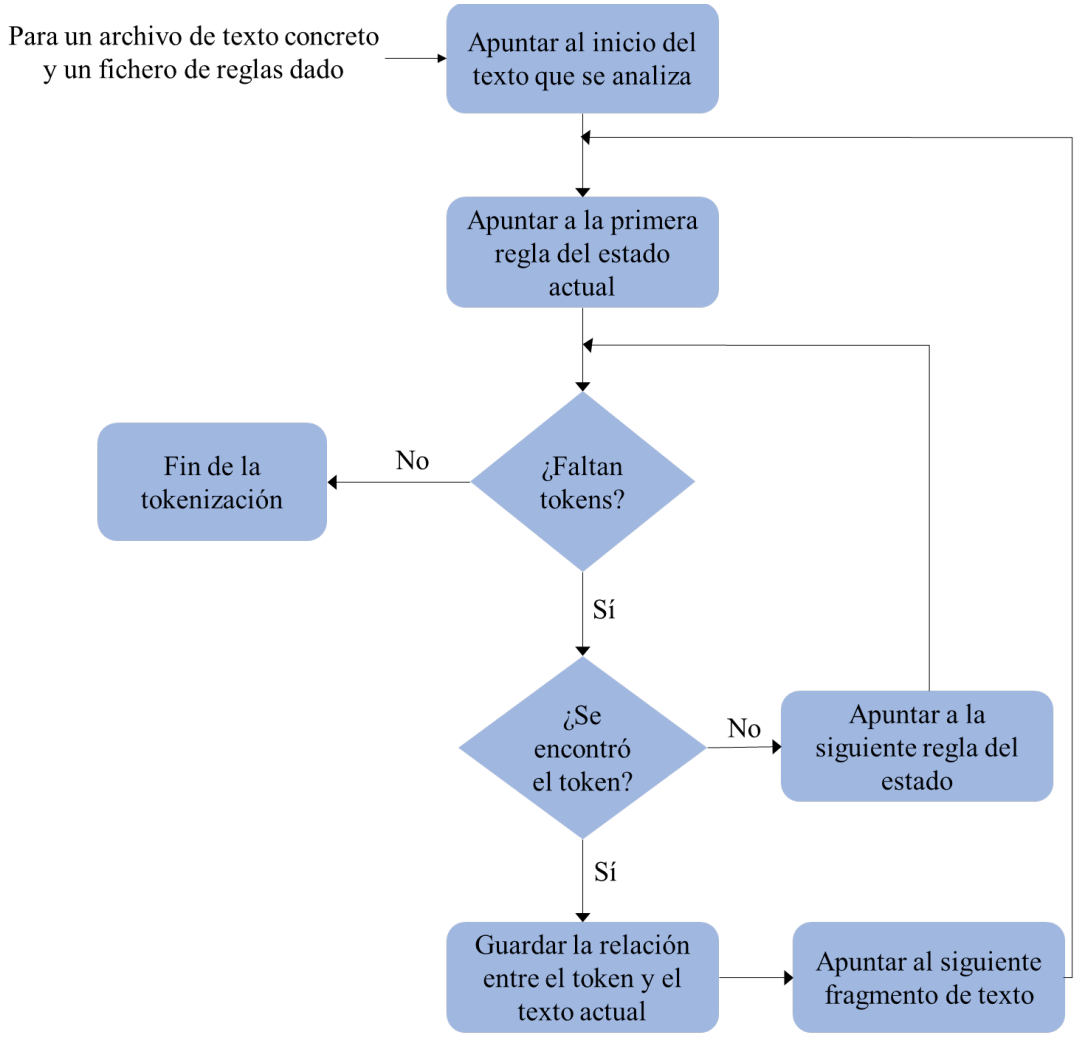

Figura 4.3: Búsqueda de un token sin cambiar de estado

En realidad, la tokenización desarrollada en VPLT puede entenderse como un proceso iterativo en el que se busca, desde un autómata finito determinista conformado por estados con reglas, el token asociado con el texto actual. En cada iteración, el motor recorre de manera descendente las reglas de un estado hasta que encuentra en alguna de ellas el token correspondiente, o alguna opción que provoque su terminación.

El motor de análisis léxico VPLT realiza la tokenización por medio de patrones de búsqueda. Este tipo de patrones se conocen como expresiones regulares, y son el mecanismo fundamental con el que se identifican los tokens de un texto. En la figura 4.3, se puede observar un diagrama de flujo básico que clarifica la aplicación de las expresiones regulares en el motor VPLT para cuando la búsqueda sólo se lleva a cabo en un único estado.

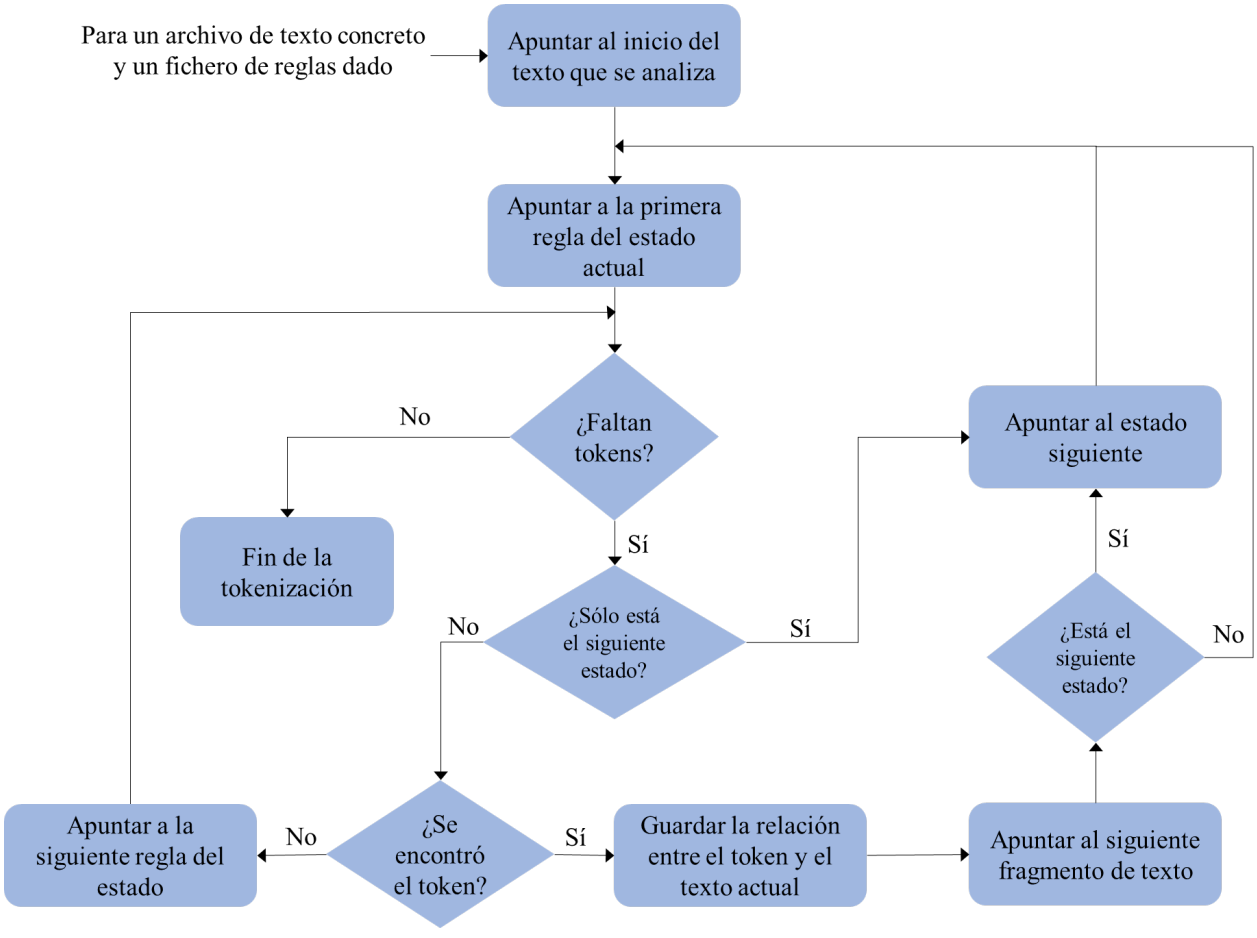

Figura 4.4: Búsqueda de un token con posibilidad de cambiar de estado

Tal y como se comentó en apartados anteriores, el motor VPLT utiliza un autómata finito determinista como modelo con el que estructurar en una serie de estados las reglas que se emplean para la tokenización. Esto significa que, durante el funcionamiento del motor, la regla que identifica un token concreto podría encontrarse en cualquier estado, no sólo por el que se comienza la búsqueda. Teniendo en cuenta este hecho, es lógico pensar que en el motor se necesita de algún mecanismo con el que definir el flujo de la búsqueda en otros estados. En el motor VPLT, se aborda este problema a partir de una opción llamada ”next” con la que se especifica el estado por el que comienza la siguiente iteración. Para más información sobre esta opción, vaya a la sección 4.4.3 del apartado de los ficheros de reglas.

Si bien con el modelo de la figura 4.4 se puede indicar al motor de análisis léxico, en cada momento y bajo ciertas condiciones, en qué estados se va a llevar a cabo la tokenización, no se aborda el caso particular en el que no se encuentra un token, pero sí que se presupone cuál podría ser. Para resolver este problema, en el motor VPLT se incluye la opción ”default_token” (sección 4.4.3 del apartado de los ficheros de reglas) con la que se puede definir, para un fragmento de texto dado, el tipo de token que le corresponde.

Finalmente, el diagrama de flujo propuesto en la figura 4.5 es la representación completa del proceso de tokenización que se realiza durante el funcionamiento del motor de análisis léxico VPLT. En esta figura, se han considerado todas las tareas, procesos, y comprobaciones, que se efectúan en el procesamiento léxico.

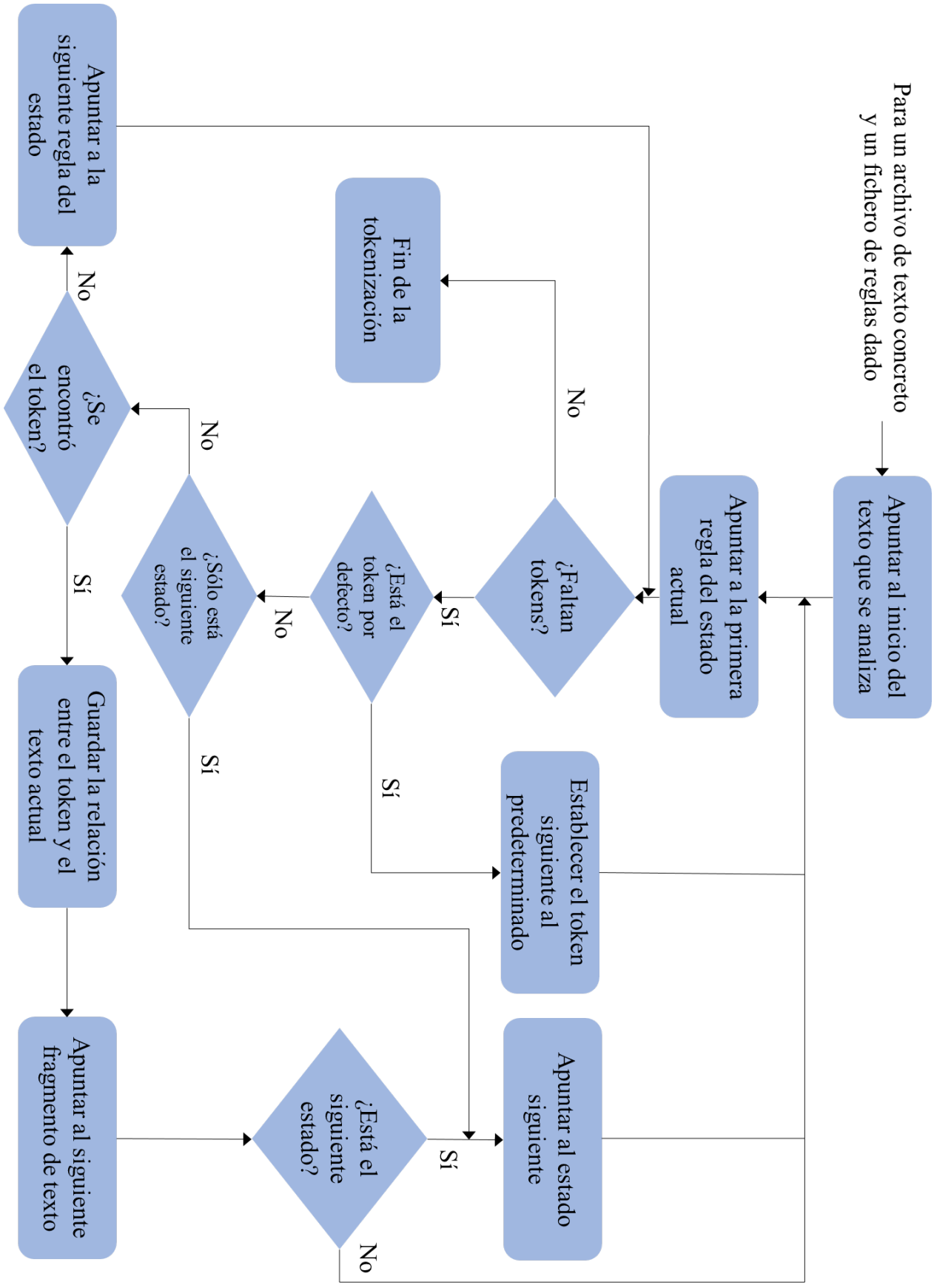

Figura 4.5: Funcionamiento completo de la tokenización

4.2.3. Adaptación del resultado

En esta última fase, el motor de análisis léxico VPLT lleva a cabo un proceso en el que se ajusta o adapta el contenido y estructura de los tokens obtenidos en la tokenización con el objetivo de que éstos puedan emplearse en la fase de troceado de la búsqueda de similaridad de VPL. El procedimiento que se realiza en esta fase de adaptación puede verse como un lote de comprobaciones con las que se descartan aquellos tokens que no aporten información léxica o que tengan en sus atributos valores inválidos.

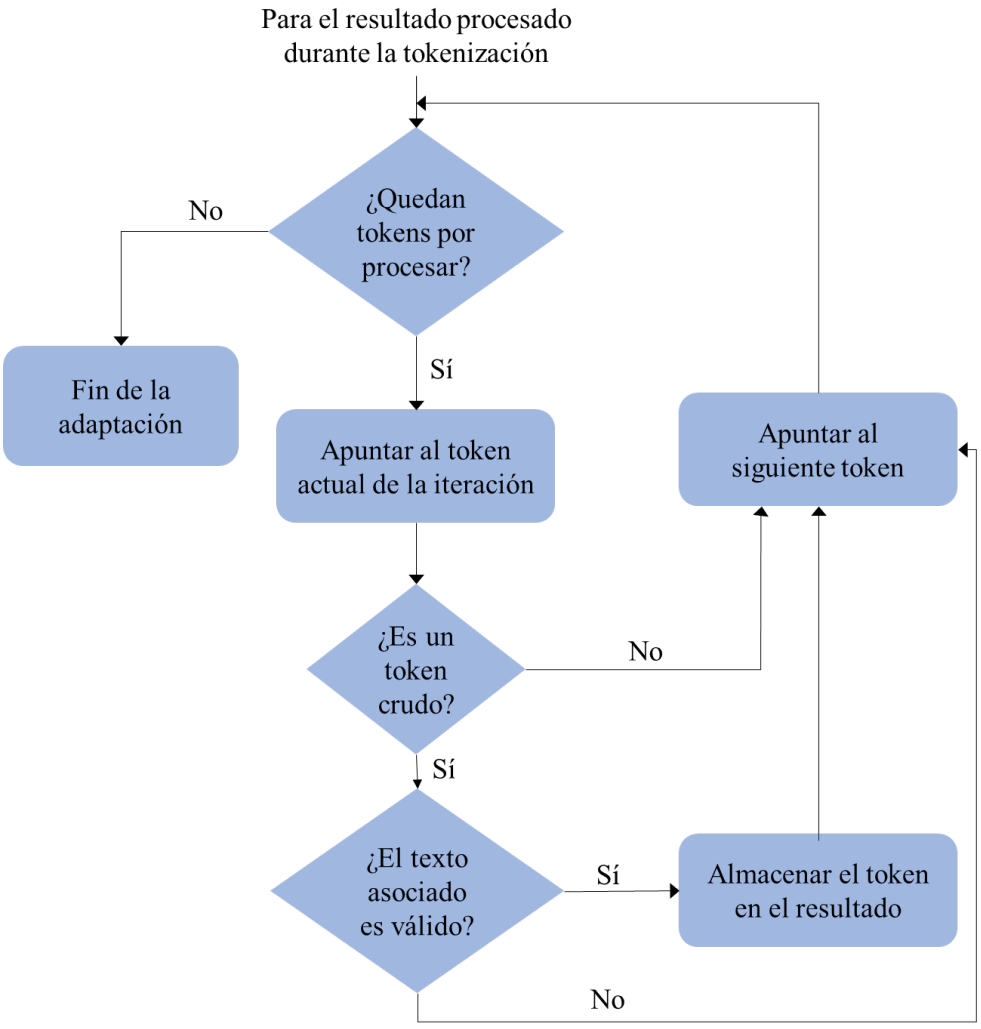

Figura 4.6: Diagrama de flujo de la adaptación del resultado

Tal y como se puede observar en el diagrama de flujo de la figura 4.6, el proceso de adaptación del resultado es una fase en la que se proporcionan aquellos tokens que realmente contengan información útil del léxico detectado. Para ello, es estrictamente necesario llevar a cabo un proceso con el que determinar cuáles son los tokens inválidos que deben descartarse para el resultado final. A partir de esto, el motor podrá proporcionar resultados correctos semejantes a los que se obtienen con los antiguos analizadores léxicos del módulo VPL. A continuación, se explicará con mayor detalle cada una de estas comprobaciones.

Por un lado, con la primera evaluación de la fase de adaptación del resultado, el motor de análisis léxico VPLT verifica el valor del tipo de un token. Las condiciones con las que se determinan esta validez se sustentan en un listado predefinido que especifica los tipos de tokens que son admitidos por el motor. Por definición, cada tipo admitido debe declararse en dicho listado mediante una relación en la que se asocia el nombre del tipo con uno de los tipos que internamente manejan los analizadores léxicos del módulo VPL. Cabe mencionar que, el listado de tokens admitidos puede modificarse dinámicamente por el desarrollador por medio de la opción "override_tokens", que se explica más adelante en la sección de los ficheros de reglas 4.4.

Por otro lado, la segunda tarea que lleva a cabo el motor de análisis léxico VPLT durante la ejecución de esta última fase consiste en comprobar que el valor del texto que se ha relacionado con un token determinado sea válido. En pocas palabras, el motor admite un texto de un token que se haya definido como mínimo con un carácter distinto al espacio en blanco. Dicho de otra manera, VPLT no admite aquellos tokens cuyo texto no tenga caracteres, o sólo esté conformado por espacios en blanco.

4.3. Ficheros de reglas

En este apartado, se recoge toda la documentación acerca de los ficheros de reglas, que son el recurso con el que los desarrolladores pueden especificar las reglas que son procesadas por el motor para llevar a cabo el procesamiento léxico o tokenización. La explicación se ha organizado en varios subapartados en los que se recoge información acerca del formato y estructura empleada en este tipo de ficheros, y qué parámetros de configuración hay disponibles para su especificación.

4.3.1. Introducción

El léxico de un lenguaje de programación se define mediante una serie de reglas que permiten detectar un token determinado mediante expresiones regulares. En VPLT, las reglas asociadas al léxico de un lenguaje se declaran en ficheros JSON especiales, conocidos como ”ficheros de reglas”, los cuales mantienen una estructura concreta, y cuyo contenido se declara siguiendo unas restricciones dadas.

```
{
   "name": "c-tokenizer",
   "extension": [ ".c", ".h" ],
   "states": {
      "start": [
         {
            "token": "comment.line",
            "regex": "\\/\\/"
         }
      ]
   }
}
```

Algoritmo 4.3: Ejemplo de un fichero de reglas

4.3.2. Estructura

En cierto modo, los ficheros de reglas no son más que archivos cuyo contenido se estructura en base a las exigencias del formato JSON. Lo interesante de este tipo de ficheros es que, en cualquier circunstancia, su contenido representa las reglas léxicas de un lenguaje. Esto exige que la información mostrada se declare necesariamente con sólo ciertos atributos que el motor reconoce y utiliza durante el procesamiento léxico. A este tipo de atributos se le conoce en VPLT como opciones, y son las empleadas para tanto la construcción de las reglas, como para la personalización del tokenizador asociado.

Además de las opciones, los ficheros de reglas pueden incluir en su especificación comentarios de una o varias líneas que, aunque no se van a utilizar por VPLT, sí que pueden llegar a ser útiles para los desarrolladores para informar acerca de alguna cuestión sobre las reglas. Es importante mencionar que los comentarios pueden definirse en cualquier parte del fichero.

```
{
   // Esto es un comentario
   "check_rules": false,

   /*
   Esto es otro comentario
   pero que abarca mayor espacio
   */
   "states": {
      [...]
   }
}
```

Algoritmo 4.4: Comentarios de los ficheros de reglas

El motor VPLT clasifica las opciones según su ámbito de actuación, su accesibilidad para otros ficheros de reglas, y requerimiento de su declaración. El listado que se muestra a continuación explica con mayor detalle cada tipo de opción.

- **Opciones generales:** se utilizan esencialmente para establecer la configuración del tokenizador y la del lenguaje de programación analizado. Este tipo de opciones también se pueden llegar a emplear para la modificación de las reglas empleadas durante el análisis léxico.
  - **Opciones privadas:** afectan únicamente al fichero de reglas que las ha declarado, lo que significa que ningún otro fichero externo puede llegar a añadirlas en su configuración. En general, las opciones privadas se utilizan para la configuración sensible o crítica del sistema.
  - **Opciones heredables:** permiten que la configuración se aplique en otros ficheros de reglas que hayan indicado que quieren integrar enteramente dicha configuración en su tokenizador.
- **Opciones para reglas:** se emplean tanto para especificar las reglas léxicas, como para definir, en caso de que se haya encontrado cierto token, el siguiente estado por el que se vuelve a comenzar el procesamiento léxico.
- **Opción "states":** es una opción heredable, que siempre debe declarase, que contiene el conjunto de estados y reglas que se emplean en el procesamiento léxico. Esta opción indexa los estados por medio de sus nombres, que son únicos.

En cuanto a la localización de las opciones anteriores en el fichero de reglas, todas ellas se declaran dentro de un objeto JSON. Las opciones generales siempre se especifican al principio del objeto, justo antes de la opción "states", en la que, dentro de cada estado, hay otros objetos JSON que representan con opciones de reglas el léxico del lenguaje.

```
{
   "general_option_1": "value_1",
   "general_option_2": "value_2",
   "states": {
      "state_1": [
         {
            "rule_option_1": "value_3",
```

```
            "rule_option_2": "value_4"
        }
    ]
  }
}
```

Algoritmo 4.5: Estructura de los ficheros de reglas

4.3.3. Opciones

En este apartado se explican cada una de las opciones que, hasta la fecha, admite el motor de análisis léxico VPLT en la especificación de los ficheros de reglas. En la explicación se incluye la definición de la opción, su aportación para el motor VPLT, y ejemplos ilustrativos que muestran cómo utilizarlo. La tabla 4.1 muestra un resumen sencillo de cada una de las opciones disponibles en el motor VPLT.

Opciones	**Nombre**	**Accesibilidad**	**Requerido**	**Tipo de Dato**
Generales	name	Privado	No	String
	extension	Privado	No	String o contenedor de strings
	check_rules	Privado	No	Booleano
	inherit_rules	Privado	No	String
	override_tokens	Heredable	No	Objeto de strings
	max_token_count	Privado	No	Número Natural
States		Heredable	Sí	Contenedor de estados
Reglas	token	Heredable	Sí	String o contenedor de strings
	regex	Heredable	Sí	String
	default_token	Heredable	No	String
	next	Heredable	No	String

Tabla 4.1: Resumen de las opciones para los ficheros de reglas

4.3.3.1. Opción ”name”

Permite especificar el nombre descriptivo o identificador del analizador léxico asociado con un fichero de reglas dado. En principio, no es necesario especificar un valor concreto para esta opción, pues de forma predeterminada se define por el motor como ”default”, pero sí que se recomienda usarlo. La razón de esto se debe al hecho de que, por un lado, la búsqueda de similaridad aplica esta opción durante el proceso de troceado, y por otro lado, en los mensajes de error del motor VPLT se indica el tokenizador que falló por medio de ”name”.

```
{
   "name": "c-tokenizer",
   [...]
}
```

Algoritmo 4.6: Ejemplo de la opción ”name”

En caso de que se quiera emplear adecuadamente la opción ”name” en un fichero de reglas, es necesario que en el nombre se indique el nombre del lenguaje, seguido del sufijo ”-tokenizer”. Esta limitación se debe a una cuestión de compatibilidad del motor con VPL.

4.3.3.2. Opción ”extension”

Con esta opción, se pueden definir las extensiones del lenguaje de programación asociado con un fichero de reglas dado. Tal y como ocurre en ”name”, la opción ”extension” no es indispensable, pero permite añadir una capa de seguridad mayor en el motor VPLT con la que asegurar que el fichero del que se obtiene los tokens está realmente definido con la extensión adecuada. Ahora bien, en caso de que se quiera omitir la comprobación de las extensiones, la opción ”extension”, o no se define, o se especifica mediante su valor por defecto ”no-ext”.

La declaración de ”extension” puede hacerse de dos maneras diferentes: con una ristra de caracteres, o con un contenedor de strings. El primer método está pensado para aquellos casos en los que sólo se quiere incluir una extensión, mientras que el segundo se utiliza para cuando hay múltiples extensiones. En ambos casos, las extensiones deben declararse con un

punto al principio, salvo que se use su valor predeterminado, que sólo está admitido para cuando se define la opción con una única ristra.

```
{
   "extension": ".c",
   [...]
}
```

Algoritmo 4.7: Ejemplo de la opción ”extension” con un sóla extensión

```
{
   "extension": [ ".c", ".h" ],
   [...]
}
```

Algoritmo 4.8: Ejemplo de la opción ”extension” con muchas extensiones

4.3.3.3. Opción ”check_rules”

”check_rules” es una opción booleana con la que se activa o desactiva el proceso previo de análisis y verificación que realiza el motor VPLT para asegurar que un fichero de reglas se ha definido según el formato y restricciones establecidas. Por defecto, esta opción está activada, pero es recomendable desactivarla para los ficheros de reglas completos que no vayan a cambiarse por mucho tiempo, ya que así se disminuye la carga de trabajo del programa y se agiliza y mejora su ejecución.

```
{
   "check_rules": true,
   [...]
}
```

Algoritmo 4.9: Ejemplo de la opción ”check_rules”

4.3.3.4. Opción ”inherit_rules”

Con esta opción, se especifica la ruta relativa del fichero de reglas del que se extrae todas sus reglas y opciones heredables para luego incorporarlas adecuadamente en el fichero actual. En VPLT, este proceso de extracción se le conoce como herencia, que, en pocas palabras, consiste en la inclusión de todos los estados y las reglas de otro fichero de reglas que no se hayan definido en las reglas actuales, así como las opciones que se hayan marcado como opciones heredables. Gracias a la incorporación de esta opción, se consigue que haya una mayor modularización de las reglas, lo que supone menos líneas de texto.

En cuanto a la especificación, ”inherit_rules” admite rutas relativas de la localización actual del fichero de reglas correspondiente. Ahora bien, en ningún caso debe incluir al final la extensión JSON del fichero, tal y como se observa en el ejemplo siguiente.

```
{
   "inherit_rules": "rules/c_tokenizer_rules",
   [...]
}
```

Algoritmo 4.10: Ejemplo de la opción ”inherit_rules”

4.3.3.5. Opción ”override_tokens”

En el motor VPLT, la mayor parte de las reglas construidas están asociadas a un tipo de token concreto. La especificación de este identificador está limitado por un repertorio predefinido de tipos, los cuales están recogidos en este documento en el apéndice A. Aunque el listado de tipos es amplio y completo, puede que esta restricción suponga un inconveniente para aquellos ficheros de reglas más complejos. La opción ”override_tokens” trata de resolver este problema.

En pocas palabras, ”override_tokens” permite la inclusión y modificación de tipos de tokens a partir del repertorio original de los mismos. Dicho de otra manera, cada tipo de token planteado en la opción estará directamente relacionado con otro tipo que ya se ha definido

en la opción, o que se admite por defecto por el motor de ánalisis léxico.

```
{
   "override_tokens": {
      "keyword.control.c": "keyword.control",
      "reserved.c": "vpl_reserved"
   },
   [...]
}
```

Algoritmo 4.11: Ejemplo de la opción ”override_tokens”

Respecto a los tipos de tokens, es importante mencionar que su construcción se ha desarrollado a partir de unos tipos generales llamados ”tokens crudos”, que están asociados con los que emplean los tokenizadores antiguos de VPL. Estos tipos se diferencian del resto en su nomenclatura: los nombres deben empezar siempre con el prefijo ”vpl_”, seguido del nombre descriptivo del token. La tabla B.11 recogida en el apéndice A del presente documento muestra una explicación de cada uno de estos tipos de tokens.

Si bien la mayor parte de los tipos de tokens que se incluyen se emplean para el procesamiento léxico, puede que hayan ciertos casos en los que se quiera ignorar alguno de ellos. La opción ”override_tokens” cubre esta necesidad a partir del tipo ”vpl_null”, con el que se indica los tokens que no deben incluirse en los resultados finales del análisis léxico. Cabe mencionar que también se puede usar el valor vacío para ignorar tipos de tokens.

```
{
   "override_tokens": {
      "comment.c": "vpl_null",
      "comment": ""
   },
   [...]
}
```

Algoritmo 4.12: Ejemplo de ”vpl_null” en la opción ”override_tokens”

4.3.3.6. Opción "max_token_count"

Especifica el número máximo de tokens que puede generar el motor de análisis léxico durante su ejecución. En caso de que se haya alcanzado este límite, los siguientes tokens que se encuentren serán de tipo "overflow", y no estarán agrupados adecuadamente. Esta limitación se utiliza como una medida de detección de bucles infinitos que se hayan creado por una mala definición de las reglas léxicas. El límite definido en esta opción nunca puede ser inferior o igual a cero, y por defecto es 2000.

```
{
   "max_token_count": 2000,
   [...]
}
```

Algoritmo 4.13: Ejemplo de la opción "max_token_count"

4.3.3.7. Opción "states"

Sin lugar a dudas, la opción "states" es la más importante de todas, pues es la que contiene todos los estados con las reglas que definen el léxico del lenguaje de programación. Cada uno de los estados está indexado en "states" por medio de su nombre, que es único y no puede repetirse en ningún otro estado, tanto en el propio fichero de reglas como en los que se hayan heredado con la opción "inherit_rules".

```
{
   "states": {
      "start": [
         [...]
      ],
      [...]
   }
}
```

Algoritmo 4.14: Ejemplo de la opción "states"

La principal diferencia de esta opción frente a las demás es que, en todos los ficheros de reglas, siempre se tiene que declarar la opción ”states”, sin importar la circunstancia. Asimismo, es necesario que en ”states” se defina por lo menos el estado ”start”, que es por el que se realiza la búsqueda de los tokens al principio del proceso de tokenización.

4.3.3.8. Opción ”token”

Esta opción indica el tipo de unidad léxica de la regla en la que se ha definido. Como ya se comentó en apartados anteriores, los tipos de tokens están limitados por un repertorio predefinido, pero puede ampliarse o modificarse dicho repertorio por medio de la opción ”override_tokens”. Para ver una lista completa de todos los tipos de tokens que admite el motor VPLT, vaya al apéndice A.

```
{
   "states": {
      "start": [
         {
            "token": "storage.type",
            "regex": "int"
         }
      ]
   }
}
```

Algoritmo 4.15: Ejemplo de la opción ”token”

Es aspectos generales, la finalidad principal de ”token” es informar sobre el tipo de unidad léxica que se encontró durante el procesamiento léxico del motor. Esta detección se hace por medio de expresiones regulares, que se definen en la opción ”regex”, y son la clave para diferenciar las reglas que tengan el mismo tipo de token. Todo esto explica el motivo de que en el diseño del motor se haya decidido crear una relación de dependencia mutua entre ambas opciones, en la que no puede declararse una opción sin la otra.

Tal y como se observa en el cuadro 4.1, la opción ”token” puede definirse tanto en forma de

ristra como con un contenedor de las mismas. Para más información acerca de la segunda alternativa, vaya al apartado siguiente en el que se habla de la opción ”regex”.

4.3.3.9. Opción ”regex”

La opción ”regex” especifica la expresión regular que se emplea durante el funcionamiento del motor para detectar los tokens del fichero de texto suministrado. Al igual que ocurre con ”token”, esta opción no puede declararse de manera individual, pues no proporcionaría al motor en ese caso ningún tipo de información útil.

```
{
   "states": {
      "start": [
         {
            "token": "keyword.control",
            "regex": "if|else"
         }
      ]
   }
}
```

Algoritmo 4.16: Ejemplo de la opción ”regex”

Por cuestiones del desarrollo, las expresiones regulares se procesan en VPLT introduciendo al principio y al final de éstas el carácter ”/”. Este hecho supone que, si se especifica en la expresión regular dicho carácter, puedan llegar a producirse errores que impidan un correcto análisis léxico. Para evitar este grave problema, es necesario utilizar el carácter de escape junto a los carácteres ”/”, tal y como se observa en el siguiente algoritmo.

```
{
   "states": {
      "start": [
         {
```

```
            "token": "comment",
            "regex": "\\/\\/"
        }
    ]
  }
}
```

Algoritmo 4.17: Ejemplo de cómo escapar el carácter ”/” en la opción ”regex”

Por último, la característica más destacable de la opción ”regex” es, sin lugar a dudas, la posibilidad de definir expresiones regulares a partir de subexpresiones o grupos. Para ello, no sólo se delimitan los grupos en la expresión por medio de paréntesis, sino que además en la opción ”token” deben haber en un contenedor tantos tokens como subexpresiones haya. Cabe mencionar que, debido a esta funcionalidad, los paréntesis que no vayan usarse para crear grupos deben declararse al principio de la forma ”(?:”.

```
{
  "states": {
    "start": [
      {
        "token": [ "keyword.storage", "text", "identifier" ],
        "regex": "(int|float|long|double)(\\s+)(.+)"
      }
    ]
  }
}
```

Algoritmo 4.18: Ejemplo de subexpresiones regulares de la opción ”regex”

4.3.3.10. Opción ”next”

”next” permite especificar el estado por el que se comenzará a buscar el siguiente token. Esta opción puede usarse junto a ”token” y ”regex”, o sin ninguna otra opción. La diferencia entre

estas dos alternativas es que, en la primera, se activa la opción siempre que la regla haya detectado la unidad léxica, mientras que la segunda se aplica para cuando se haya alcanzado la regla en la que se define.

```
{
   "states": {
      "start": [
         {
            "token": "comment.line",
            "regex": "\\/\\/$",
            "next": "start"
         },
         {
            "next": "start"
         }
      ]
   }
}
```

Algoritmo 4.19: Ejemplo de la opción ”next”

4.3.3.11. Opción ”default_token”

Durante el procesamiento léxico, es posible que ocurra un caso excepcional en el que no se encuentra un token, pero que igualmente se tenga información suficiente como para presuponer cuál podría ser. En estas circunstancias, el motor de análisis léxico dispone de una opción, conocida como ”default_token”, con la que se indica el tipo predeterminado al que se asociará el texto en caso de que las reglas anteriores no lo hayan hecho ya.

```
{
   "states": {
      [...],
      "multipleLineComment": [
```

```
            {
                "token": "comment.block",
                "regex": "\\*\\/",
                "next": "start"
            },
            {
                "default_token": "comment.block"
            }
        ]
    }
}
```

Algoritmo 4.20: Ejemplo de la opción ”default_token”

La opción ”default_token” es un caso particular que sólo puede incluirse en aquellas reglas que no tengan otras opciones tales como ”token”, ”regex”, o ”next”. El motivo de esto es por el hecho de que dicha opción provoca la terminación forzosa de la iteración actual sin necesidad de disponer de expresiones regulares u otro tipo de información. Además, cuando la opción ”default_token” se activa, el siguiente estado siempre será el actual, y no habrá forma de cambiar esta condición.

Para emplear adecuadamente la opción ”default_token”, lo ideal es que ésta siempre se declare en la última regla del estado para que así todas sus reglas pueda aprovecharse en algún momento de la tokenización. En caso de que ”default_token” estuviese en alguna regla anterior a la última, las siguientes no podrían explorarse, pues ”default_token” no requiere de ninguna condición para su activación más allá de que en la búsqueda se llegue a la regla en la que está definida.

Capítulo 5

Desarrollo

En este capítulo, se recoge información detallada sobre el desarrollo del motor de análisis léxico propuesto en el presente trabajo de fin de título. En la explicación, se incluyen apartados que documentan aspectos tales como las herramientas empledas para el desarrollo del proyecto, así como las diferentes fases en las que se dividió el desarrollo.

5.1. Herramientas usadas

En este apartado, se muestra un listado detallado de todas las herramientas empleadas durante el desarrollo del motor de análisis léxico. Para cada caso, se explicará sus aportaciones y los motivos de su uso, y, en algunos casos, las razones de haberlo escogido frente a otras herramientas alternativas. Cabe mencionar que, al final de cada apartado, se incluye una descripción técnica más completa sobre dicha herramienta.

5.1.1. Ace

Ace es un editor de código fuente que proporciona desde la web un entorno virtual en el que pueden colaborar varios desarrolladores al mismo tiempo. Una de las características que distingue a Ace de otros editores de código es su capacidad de poder integrarse fácilmente en cualquier página web o aplicación escrita en JavaScript.

Figura 5.1: Editor de código Ace [3]

5.1.1.1. Uso en el proyecto

Al igual que sucede en otras aplicaciones, el código fuente de Ace está organizado en una serie de componentes o módulos, los cuales se encargan de cubrir cada una de las características del editor. El desarrollo del motor de análisis léxico planteado en este proyecto está basado en uno de estos componentes, el tokenizador, por ser un sistema para el procesamiento léxico que resuelve el mismo problema que el propio motor. El hecho de ser este tokenizador la fuente de inspiración del software desarrollado en este proyecto, hace que Ace se convierta en una herramienta esencial para el mismo.

5.1.1.2. Detalles

Ace destaca por ser un editor de código fuente eficiente, rápido, y ligero, que se puede usar en cualquier sitio web de la misma manera que se hace con las librerías JavaScript en HTML. Tal y como se dice en su página web oficial, Ace es el sucesor del obsoleto editor Bespin, y es el editor que se utiliza en el IDE Cloud9. Entre todas las características que tiene, las más interesantes son el resaltado de sintaxis, el indentado automático, el plegado de código, la personalización de la apariencia del editor por medio de temas, entre otros.

En este trabajo de fin de título, no se ha considerado a Ace para la edición de código fuente, sino como el punto de partida inicial para el desarrollo del motor de análisis léxico VPLT. Los ficheros consultados en este proyecto se encuentran en el repositorio de Github de Ace, cuyo enlace es el siguiente: https://github.com/ajaxorg/ace.

5.1.2. PHP

PHP: Hypertext Preprocessor [18] es un lenguaje de programación de propósito general, multiparadigma, y de código abierto, que se usa especialmente para el desarrollo web. Destaca por ser un lenguaje que permite gestionar dinámicamente los servidores y bases de datos de una aplicación web, asegurando un buen rendimiento, seguridad, y fiabilidad.

Figura 5.2: Lenguaje de programación PHP [19]

5.1.2.1. Uso en el proyecto

Tal y como se ha explicado anteriormente, el motor de análisis léxico que se explica en este documento es una herramienta interna del Virtual Programming Lab (VPL) [20]. Dicha herramienta es, a su vez, un módulo de Moodle [21], y tanto Moodle como VPL están escritos en PHP. Teniendo en cuenta todo esto, es lógico pensar que el motor necesite estar escrito en PHP, ya que utilizar otro lenguaje de programación supondría desarrollar igualmente software que traduzca el código a PHP.

5.1.2.2. Detalles

Una de las características más destacables de PHP es que está orientado al servidor. Esto significa que, desde el cliente, no se tiene acceso al código fuente escrito en PHP, y tampoco se dispone de una comunicación directa con los servidores web. Gracias a esta ventaja, los desarrolladores pueden incorporar nuevas funcionalidades para sus sitios web que utilicen datos privados sin que ésto comprometa su seguridad.

PHP es un lenguaje de programación orientado a objetos (POO), disponible en la mayoría de los servidores web, con una amplia documentación y una larga lista de herramientas o

frameworks que facilitan enormente el desarrollo web. Para más información sobre el lenguaje, acceda al manual en español desde el enlace https://www.php.net/manual/es/.

Respecto a la versión empleada, el código fuente se ha desarrollado usando la nueva versión 8 de PHP. Sin embargo, el código escrito en el trabajo es compatible con PHP 7.4.3, que es la que exige como requisito Moodle 4.0.

5.1.3. Moodle

Moodle es una aplicación diseñada para ofrecer a los usuarios un sistema seguro, fiable, y robusto, para crear en Internet espacios de enseñanza personalizados. En el ámbito de la educación, la plataforma Moodle es una utilidad muy popular siendo, en su ámbito, la herramienta de código abierto más usada con más de 170000 instalaciones en todo el mundo (información obtenida de https://stats.moodle.org/).

Figura 5.3: Plataforma de aprendizaje Moodle [21]

5.1.3.1. Uso en el proyecto

Moodle se caracteriza por ser un sistema muy personalizable en el que se pueden incluir nuevas utilidades o módulos que añaden más funcionalidades y mejoran la calidad de los cursos creados mediante esta plataforma. El motor de análisis léxico desarrollado forma parte de uno de estos módulos, el Virtual Programming Lab (VPL). Este hecho convierte a Moodle en el pilar fundamental en el que se sustenta todo el proyecto, pues es el espacio de trabajo en el que se utiliza y ejecuta el motor.

5.1.3.2. Detalles

En este proyecto, se ha utilizado Moodle 4.0, que se diferencia de su predecesor por tener una interfaz gráfica más amigable, sencilla, que mejora la experiencia de los usuarios. Ahora

bien, el motivo de que se haya preferido el Moodle 4.0 frente a versiones anteriores es para poder usar las características que proporciona la versión de PHP 8.0.

5.1.4. VPL

Virtual Programming Lab (VPL) es un módulo de Moodle orientado al manejo y desarrollo de actividades educativas de programación desde un espacio virtual de la red. Proporciona a los usuarios la edición, ejecución, y evaluación de programas escritos en varios lenguajes de programación tales como C, C++, Go, Java, o Fortran.

Figura 5.4: Módulo de Moodle VPL [20]

5.1.4.1. Uso en el proyecto

De entre todas las posibilidades que ofrece VPL, la búsqueda de similaridad es, para este proyecto, una de las más interesantes. Como el motor de análisis léxico ha sido desarrollado para remplazar en VPL la antigua manera de realizar la tokenización en la búsqueda de similaridad, es razonable que éste sea una herramienta obligatoria para el desarrollo de este proyecto.

5.1.4.2. Detalles

VPL es una aplicación diseñada para cubrir las necesidades tanto de los profesores como la de los alumnos. Por un lado, los docentes tienen toda la información sobre los envíos realizados, pueden evaluar una tarea de programación de forma manual o por medio de pruebas automáticas, y pueden detectar plagios en las actividades mediante la búsqueda de similaridad. Por otro lado, los estudiantes pueden utilizar un IDE sencillo con el que ejecutar, depurar, o incluso evaluar el código fuente desarrollado. Para más información sobre el módulo, vea el manual oficial de VPL [22].

Respecto a la versión empleada, el motor de análisis léxico se ha incluido en el módulo en una nueva versión llamada VPL 3.5.0++, que está basada en la antigua versión VPL 3.5.0+. Dentro de VPL 3.5.0++, el motor tiene un registro de versiones independiente en el que se puede conocer cada uno de los cambios aplicados en el proyecto. Para más información, vea el manual oficial del motor VPLT en https://github.com/losedavidpb/tokenizer-doc_vpl.

5.1.5. PHPUnit

PHPUnit [23] es un marco de trabajo del grupo de frameworks de pruebas xUnit [24] con la que se prueden desarrollar pruebas unitarias en PHP. Con esta herramienta se puede verificar fácilmente por medio de lo que se conoce como aserciones que cada uno de los componentes que conforman una aplicación funcionen según lo esperado.

Figura 5.5: Framework de pruebas PHPUnit [23]

5.1.5.1. Uso en el proyecto

Actualmente, existen en el mercado multitud de frameworks alternativos a PHPUnit con los que también se pueden desarrollar pruebas unitarias o incluso, otro tipo de pruebas como los tests funcionales o de aceptación. No obstante, se ha optado por utilizar PHPUnit no sólo por ser el framework más popular sino también por ser el que se utiliza en VPL para el desarrollo de pruebas unitarias.

5.1.5.2. Detalles

En PHPUnit, cada prueba unitaria se considera un bloque de código que verifica que un componente del programa en una situación dada funcione adecuadamente. Los tests se definen mediante aserciones, que son hechos o reglas que deben comprobarse, por lo que una prueba no se considera exitosa si alguna de las aserciones no se cumple. Existe una larga lista de

aserciones disponibles en PHPUnit, y están documentadas en https://phpunit.readthedocs.io/es/latest/assertions.html.

Para este proyecto, se ha utilizado la versión 9.5.13 de PHPUnit para el desarrollo de las pruebas unitarias. Cabe destacar que PHPUnit se ha utilizado junto a la métrica de cobertura del código fuente, con la que se puede detectar qué partes del código no se han comprobado.

5.1.6. Behat

Behat [25] es un framework de código abierto para PHP que facilita el desarrollo de pruebas orientadas al comportamiento (BDD). Se trata de una herramienta sencilla que utiliza el lenguaje natural para definir pruebas o escenarios en los que se especifica cómo debe ser el flujo de ejecución del programa, es decir, cómo debe actuar.

Figura 5.6: Framework de pruebas Behat [25]

5.1.6.1. Uso en el proyecto

Tal y como sucede con PHPUnit, Behat no es el único framework de pruebas orientadas al comportamiento para PHP que existe. De hecho, hay muchas otras posibilidades, en ciertos casos, mejores, que podrían emplearse. Ahora bien, las pruebas de comportamiento desarrolladas para el motor de análisis léxico de este proyecto son pocas y sencillas, y VPL usa Behat para sus pruebas BDD. Todo esto implica que, si se cambiase la tecnología, se perdería tiempo y supondría un mayor coste que si se optase por mantener el framework actual.

Este proyecto ha aprovechado Behat para probar la interfaz y el flujo de ejecución de la aplicación en aquellos casos en los que se emplea el motor de análisis léxico VPLT. De esta forma, se garantiza que los resultados de la búsqueda de similaridad que visualiza un profesor son adecuados y son los que se esperaban.

5.1.6.2. Detalles

Behat es un marco de trabajo, basado en Cucumber [26], cuyas pruebas se expresan con el lenguaje de programación Gherkin. Los casos de uso se definen en lo que se conoce como features, y éstos, a su vez, están formados por una serie de escenarios, que son el listado de sucesos o hechos que deben ocurrir durante el flujo del programa.

Una de las características más destacables de Behat con respecto a otras herramientas BDD, es que las pruebas se expresan mediante el lenguaje natural, lo que hace que sea mucho más legible para los desarrolladores, fácil de utilizar y de aprender.

5.1.7. Visual Studio Code

Visual Studio Code [27] es un editor de código gratuito y multiplataforma, que ofrece soporte para muchos lenguajes de programación, entre ellos PHP. Se caracteriza por disponer de coloreado de sintaxis, autocompletado, y la ejecución y depuración de programas.

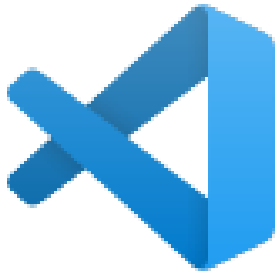

Figura 5.7: Editor de código Visual Studio Code [27]

5.1.7.1. Uso en el proyecto

En este caso, Visual Studio Code no es indispensable ni tampoco la única opción disponible. En el mercado hay muchos IDEs y editores de código que ofrecen los mismos o incluso más servicios que Visual Studio Code. La decisión de usar este editor ha sido completamente personal, y porque además fue el recomendado por el tutor de este trabajo de fin de título.

5.1.7.2. Detalles

Una de las ventajas más relevantes de Visual Studio Code es el hecho de que permite a los desarrolladores crear extensiones que incluyan más funcionalidades, las cuales pueden abarcar desde temas que modifican la apariencia del editor, hasta software de soporte para nuevos lenguajes o lenguajes de programación ya mantenidos en el editor. Esta característica hace que Visual Studio Code sea escalable, lo que supone que haya en el editor cada vez mejor soporte para cualquier lenguaje de programación.

5.1.8. Git

Git [28] es un sistema de control de versiones con el que se puede mantener un control constante de las modificaciones realizadas en un programa. Esta herramienta es de código abierto, ofrece una alta rapidez y eficiencia, y suele utilizarse en los casos en los que en una aplicación grande hay varias personas colaborando.

Figura 5.8: Sistema de control de versiones Git [28]

5.1.8.1. Uso en el proyecto

Aunque Git suele ser mucho más útil para los casos en los que trabajan uno o varios equipos de desarrollo en un mismo proyecto software, también puede ser ideal para cuando sólo hay un desarrollador colaborando en una aplicación. El hecho de que Git sea una especie de registro de los cambios y evolución del software hace que se pueda utilizar como medida de seguridad del código fuente. Por todo esto, se ha optado por utilizar en este proyecto Git como una herramienta con la que guardar las diferentes versiones del motor sin que ello pueda llegar a comprometer los cambios hechos.

5.1.8.2. Detalles

Git se diferencia del resto, entre otras cosas, por su modelo de ramas, y por ser un software rápido, que ocupa muy poco en memoria, y que tiene un espacio llamado índice en el que se pueden verificar los cambios realizados antes de publicarlos. La documentación de Git ofrece información mucho más detallada sobre este tema en https://git-scm.com/about.

Además de Git, también se ha usado Github [29] como plataforma en la que alojar el código fuente de las aplicaciones en repositorios. Para este proyecto, hay dos repositorios públicados en Github, uno para el motor VPLT (https://github.com/losedavidpb/moodle-mod_vpl/tree/v3.5.0++), y otro con su manual de usuario (https://github.com/losedavidpb/tokenizer-doc_vpl).

5.1.9. Docker

Docker [30] es un producto software gratuito y de código abierto que ofrece una capa de abstracción para la virtualización de contenedores software. Dichos contenedores ejecutan software de forma aislada, asegurando de esta manera que un error o un fallo de seguridad en el mismo no afecte al resto del sistema.

Figura 5.9: Contenedores para despliegue de programas Docker [31]

5.1.9.1. Uso en el proyecto

Para poder colaborar en el módulo VPL, es indispensable preparar el entorno de programación con todas las aplicaciones necesarias para la ejecución, depuración, y verificación del código fuente. Con el fin de agilizar esta fase inicial del desarrollo, se ha optado por utilizar Docker. Asimismo, se ha preferido esta aplicación frente a otras por el simple hecho de que en VPL la preparación del entorno se hace a partir de Docker.

5.1.9.2. Detalles

En este proyecto, los contenedores e imágenes de Docker que se emplearon para preparar el entorno de programación se instalaron en una distribución Linux virtual. Cabe mencionar que, también se utilizó Docker Desktop [32], que es una aplicación que facilita el manejo de Docker desde máquinas Windows o Linux.

5.1.10. Windows Subsystem for Linux

Windows Subsystem for Linux (WSL) [33] es una aplicación para máquinas Windows con la que se puede ejecutar un entorno Linux desde el propio sistema operativo sin la necesidad de utilizar máquinas virtuales que tenga instalada alguna distribución Linux.

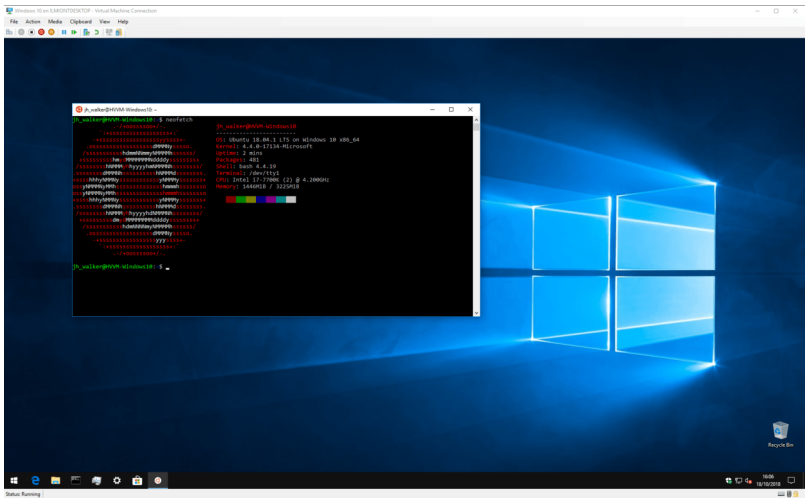

Figura 5.10: Terminal de Windows Subsystem for Linux [34]

5.1.10.1. Uso en el proyecto

El motor de análisis léxico VPLT se desarrolló desde una máquina Windows. No obstante, para la ejecución del código fuente, y la preparación del entorno de programación se optó por un entorno virtual de Linux creado a partir de WSL. Los motivos de esta elección son, por un lado, porque las máquinas virtuales pueden llegar a ser lentas y tediosas en su uso, y por otro lado, porque la instalación de WSL y la preparación del entorno es rápido, sencillo, y el software descargado ocupa muy poco espacio en memoria.

5.1.10.2. Detalles

Para el desarrollo del motor, se ha empleado WSL 2 que tiene instalado la distribución Ubuntu versión 22.04.1 LTS. La mayoría de herramientas explicadas en este apartado, tales como Moodle, Docker, entre otros, se prepararon desde este entorno virtual, así como el código fuente del módulo VPL y la del motor VPLT.

5.1.11. Sphinx

Sphinx [35] es una herramienta que se encarga de generar documentación relacionada con la programación traduciendo un conjunto de ficheros de texto planos escritos con lenguajes de marcado tales como reStructuredText o con Markdown.

Figura 5.11: Generador de documentos Sphinx [36]

5.1.11.1. Uso en el proyecto

Los manuales de usuario pueden realizarse a partir de múltiples herramientas. De entre todas las disponibles, se ha optado para este proyecto por Sphinx y reStructuredText, no sólo por su facilidad de uso sino también por la salida que se genera, que presenta el contenido de manera clara, organizada, y con una interfaz agradable para cualquier dispositivo electrónico.

5.1.11.2. Detalles

Respecto al funcionamiento de Sphinx, este generador emplea ficheros escritos en reStructuredText o Markdown para crear documentación que se representa por medio de ficheros HTML. Lo notable de este proceso es que el contenido generado tiene un diseño concreto, que se adapta en base a las necesidades de los propios usuarios. En Sphinx, este diseño se conoce como plantillas o templates, y el generador ya ofrece una gran variedad de estas. Cabe mencionar que, las plantillas están escritas en HTML, CSS, y JavaScript, y se pueden modificar o añadir nuevos templates.

5.1.12. TeX Live

TeX Live [37] es una distribución de código abierto, disponible para múltiples plataformas, con la que se puede utilizar LaTeXy TEXpara generar electrónicamente libros, trabajos académicos, o artículos científicos, entre otros tipos de documentos.

Figura 5.12: Distribución de LaTeX TeX Live [38]

5.1.12.1. Uso en el proyecto

TeX Live y LaTeX son las herramientas esenciales con las que se ha redactado este trabajo de fin de título. Bien es cierto que existen otras alternativas diseñadas para crear documentos electrónicos, pero en este caso se ha preferido utilizar un tipo de herramienta que emplea ficheros de texto no formateados. Por último, es importante aclarar que ha usado TeX Live en vez de otra distribución por una cuestión meramente personal.

5.1.12.2. Detalles

Por un lado, LaTeX es un lenguaje de marcado diseñado para la preparación de documentos electrónicos. A diferencia de otras herramientas, en LaTeX se codifica la información con ficheros de texto plano que no están formateados. Para que el contenido tenga una estructura ordenada y un estilo concreto, en LaTeX los ficheros usan etiquetas de marcado, como ocurre con otros lenguajes de marcado como Mardown. Cabe mencionar que LaTeX no es más que una macro herramienta del sistema tipográfico TeX.

Por otro lado, TeX Live es una aplicación que proporciona soporte y una serie de utilidades para LaTeX y TeX. Actualmente, esta distribución es un estándar para muchas distribuciones Linux, y es el sucesor moderno y actualizado de TeTex.

5.2. Preparativos previos

Antes de comenzar con el desarrollo del motor de análisis léxico, es estrictamente necesario llevar a cabo un estudio preliminar del marco teórico en el que se sustenta el proyecto, así como un proceso de preparación del entorno de programación y herramientas empleadas para la implementación del software.

Con el estudio preliminar se adquirieron conocimientos suficientes sobre los generadores de análisis léxico, así como de los modelos, técnicas, estrategias, y métodos que pueden emplearse para la resolución del problema. Para cada caso, se analizaron sus características, ventajas, y debilidades, además del costo computacional que supone su implementación e integración en VPL. Asimismo, en el estudio también se analizó el código fuente del tokenizador desarrollado en el editor de Ace. De esta forma, se pudo tener información con la que desarrollar una motor simple compatible con VPL que utiliza el modelo planteado en Ace como fuente principal de inspiración.

Después del estudio teórico, se seleccionaron las herramientas y tecnologías empleadas para el desarrollo del motor de análisis léxico VPLT. Debido al hecho de que el motor desarrollado es un componente interno del módulo VPL, muchas de las herramientas se escogieron, por un lado, para mantener la compatibilidad entre ambos programas, y, por otro lado, para

evitar trabajo innecesario. Respecto al resto de herramientas, los criterios que se consideraron fueron, principalmente, el nivel de familiarización del autor con dicha herramienta, y su facilidad de uso.

Respecto al entorno de programación, el motor de análisis léxico se desarrolló desde una máquina con el sistema operativo Windows 10, en la que se instaló ciertos programas como Visual Studio Code, Windows Subsystem for Linux, Docker Desktop, Sphinx, y TeX Live. El resto de aplicaciones que se emplearon se descargaron e instalaron a partir del entorno virtual de Linux Ubuntu 22.04.1 LTS, que se preparó en WSL 2. Estas herramientas son, concretamente, PHP, Moodle, VPL, Docker, Git, Behat, y PHPUnit. Para más información sobre todas estas herramientas, vaya al apartado anterior en el que se explica sus características, y sus aportaciones para el presente proyecto.

5.3. Implementación del motor

El código fuente del motor de análisis léxico VPLT se organiza en una serie de ficheros, escritos en PHP, que se encargan de cubrir los diferentes procesos que se llevan a cabo durante el funcionamiento del programa. En general, cada archivo del motor VPLT describe uno de sus componentes por medio de la orientación a objetos (POO), en la que se definen los datos por medio de plantillas conocidas como ”clases”. Asimismo, el código escrito mantiene un estilo predefino por Moodle, trata de ser lo más sencillo y modular posible, y aplica las características de PHP 8.0 para ofrecer un sistema eficiente y seguro.

La mayor parte de los ficheros de VPLT proporcionan al módulo un componente o clase nueva. Sin embargo, hay que tener en cuenta que, al ser el motor VPLT una parte interna de VPL, existen ciertos archivos que ya existían en el módulo, y que sólo se han modificado para que sean compatibles con el motor. También, hay ciertos cambios de estos archivos que no modifican el contenido del código, pero que añaden una capa adicional al programa.

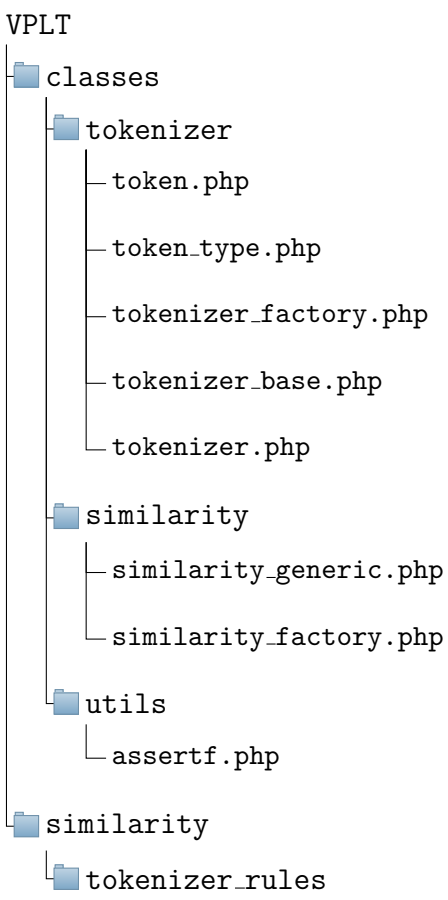

Figura 5.13: Estructura/Ubicación de ficheros del proyecto

Tal y como se puede observar en la figura 5.13, los ficheros de código fuente del motor VPLT se distribuyen en cuatro carpetas diferentes. Cada una de éstas almacena únicamente aquellos ficheros que están directamente relacionados con una parte concreta del desarrollo. En pocas palabras, "tokenizer" contiene los ficheros que se emplean para el funcionamiento de VPLT, "similarity" tiene los que permiten que VPLT pueda emplearse como herramienta de tokenización para la búsqueda de similitud de VPL, "utils" es la carpeta con funcionalidades utilizables en cualquier parte del motor o del módulo, y en la carpeta "tokenizer_rules" se encuentran todos los ficheros de reglas admitidos por el motor VPLT.

En los siguientes apartados, se incluye un listado de los ficheros que componen al motor de análisis léxico VPLT. En cada caso, se explica tanto el propósito del fichero, como el contenido del mismo (propiedades y funciones de la clase). Cabe mencionar que la explicación no considera las partes particulares que son privadas para el resto del software, sino sólo las que son accesibles y utilizables externamente. A mayores, aquellos archivos que no se hayan incluído en el listado no se tuvieron en cuenta porque no sufrieron cambios notables.

token.php

En el fichero "token.php", se dispone de una clase que define la estructura de datos que representa a una unidad léxica. Esta clase codifica un token con tres propiedades públicas distintas: su tipo, valor, y número de línea. El primero se refiere al nombre descriptivo que

identifica al token, el segundo a la cadena de caracteres que se asoció con una expresión regular al mismo, y el tercero al número de línea en el que se encontró.

En este caso, el fichero "token.php" ya existía en el módulo VPL. Lo único que se ha hecho es adaptar la clase para que sea compatible tanto para el motor como para VPL. En el siguiente listado, se incluye un listado de las funciones disponibles en dicha clase:

- **equals_to:** comprueba si dos tokens son iguales en sus propiedades.
- **hash:** calcula a partir de las propiedades del token su código hash.
- **show:** muestra por el canal de salida las propiedades del token.

token_type.php

El fichero "token_type.php" representa en una clase los tipos de tokens que admite la búsqueda de similitud de VPL. Con esta clase, el motor VPLT lleva a cabo un proceso de adaptación en el que se modifican los tipos de tokens del motor por los incluídos en "token_type.php". Gracias a esto, VPLT es compatible con VPL, y puede emplearse en la tokenización de la búsqueda de similitud. Cabe mencionar que, "token_type.php" es un fichero que ya existía en VPL, que se ha modificado para que el motor pueda usarlo. Para más información sobre los tipos de tokens de VPL, vaya al cuadro B.11, en el que se listan los tokens crudos de VPLT compatibles con los de VPL.

tokenizer_factory.php

En VPL, existe un componente que se encarga de construir automáticamente los tokenizadores que se emplean en la búsqueda de similitud. En VPLT, se ha trasladado dicho componente a la clase del fichero "tokenizer_factory.php". El proceso de construcción se ha cambiado para que VPLT pueda utilizarse en el módulo VPL. En pocas palabras, la clase desarrollada en este fichero trata de crear tokenizadores de dos maneras. En la primera, el sistema intenta definir el analizador por medio de VPLT. Para ello, busca en la carpeta "tokenizer_rules" un fichero de reglas determinado. En caso de que no lo encuentre, la construcción se realiza con los sistemas léxicos antiguos de VPL.

El fichero "tokenizer_factory.php" lleva a cabo el proceso de construcción por medio de la función pública "get". La llamada de esta función requiere de un único parámetro que especifique el nombre de un lenguaje de programación, que debe ser, si se quiere usar el motor VPLT, el mismo que se emplea en los nombres de los ficheros de reglas.

tokenizer_base.php

El fichero "tokenizer_base.php" incluye una clase global que especifica las propiedades y funciones generales que cualquier propuesta de motor de análisis léxico VPLT debe tener. En este caso, "tokenizer_base.php" podría entenderse como la definición inicial y abstracta de VPLT, que representa las características fundamentales de dicho motor. En los siguientes listados, se ofrece una explicación de las propiedades y funciones de este fichero.

Por un lado, las propiedades del fichero "tokenizer_base.php" son:

- **states:** estructura de datos que almacena el conjunto de estados que se especificaron en los ficheros de reglas que participan en la construcción del analizador léxico. En este listado, los estados se indexan por su nombre para así agilizar su búsqueda.
- **matchmappings**: mapeado de los estados declarados en los ficheros de reglas que se emplean para la creación del tokenizador. Este mapa de estados se utiliza en VPLT para poder acceder fácilmente a las reglas de los estados.
- **regexprs:** lista compacta de expresiones regulares encontradas en cada estado. Este atributo comprime las expresiones de las reglas de un estado con el operador OR.
- **tokens:** conjunto de tokens obtenidos en el último procesamiento léxico que se llevó a cabo en el motor VPLT. Para VPL, los tokens que se devuelven ya están adaptados para que sean compatibles para la búsqueda de similitud.

Por otro lado, las funciones del fichero "tokenizer_base.php" son:

- **get_tokens:** devuelve los tokens obtenidos en el último procesamiento léxico.
- **get_states:** devuelve los últimos estados procesados por el motor VPLT.
- **get_matchmappings:** devuelve el mapa de estados procesados anteriormente.

- **get_regexprs:** devuelve las expresiones regulares de cada estado procesado.
- **check_type:** comprueba si un valor está definido en un tipo de datos dado.
- **contains_rule:** verifica si una regla se encuentra en un estado concreto.
- **check_token:** comprueba si los tokens están incluidos en un listado dado.
- **remove_capturing_groups:** elimina de una expresión regular las subexpresiones.
- **get_token_array:** devuelve los tokens de un texto determinado a partir de subexpresiones especificadas en una expresión regular. Para la construcción de los tokens, se dispone además del número de línea y tipo de cada uno.

tokenizer.php

La codificación del funcionamiento principal del motor de análisis léxico se concentra principalmente en el fichero ”tokenizer.php”. En este archivo, se puede observar desde una punto de vista práctica todas las fases que participan en el funcionamiento del motor: preprocesamiento, tokenización, y adaptación del resultado. Asimismo, es en este fichero en el que se especifican los tokens adminitos por VPLT, así como los valores por defecto de cada opción disponible de los ficheros de reglas. Para más información sobre el proceso que sigue VPLT durante su ejecución, vaya al capítulo anterior.

El fichero ”tokenizer.php” estructura las fases del funcionamiento del motor en una serie de funciones, las cuales son mayoritariamente de acceso privado. Sin entrar en demasiados detalles, el motor dispone de funciones que se encargan de inicializar las opciones de los ficheros de reglas, cargar el fichero JSON y quitar los comentarios, comprobar que los estados y reglas son válidos, entre otros. En cada una de estas funciones, VPLT realiza una gran cantidad de verificaciones con las que asegurar que los datos suministrados son correctos y adecuados para la tokenización. En los listados siguientes, se explican las propiedades y funciones del fichero ”tokenizer.php”. Es importante mencionar que, para el caso de las propiedades, sólo se han incluído las más importantes para la comprensión del fichero.

Por un lado, las propiedades de ”tokenizer.php” son:

- **name:** nombre del tokenizador que se construye.
- **extension:** lista de extensiones del lenguaje procesado.
- **checkrules:** verifica si se quiere analizar los ficheros de reglas.
- **inheritrules:** ruta del fichero de reglas que se hereda.
- **setcheckrules:** asigna "checkrules" sin considerar el fichero de reglas.
- **rawoverridetokens:** tokens redefinidos con la opción "override_tokens".
- **maxtokencount:** número máximo de tokens admitidos para la tokenización.
- **availabletokens:** tokens adminitos por defecto por el motor VPLT.

Por otro lado, las funciones de "tokenizer.php" son:

- **get_override_tokens:** devuelve la lista final de los tokens admitidos.
- **get_raw_override_tokens:** devuelve el valor de la opción "override_tokens".
- **get_max_token_count:** devuelve el valor de la opción "max_token_count".
- **set_max_token_count:** asigna un valor para la opción "max_token_count".
- **parse:** realiza la tokenización y devuelve los tokens obtenidos. En este caso, la función admite que se le pase como parámetro el nombre del fichero, o un contenedor con su contenido ya procesado.
- **get_all_tokens:** devuelve los tokens obtenidos en cada línea de un fichero. En esta función, no se lleva a cabo la adaptación del resultado, por lo que no es adecuado emplearlo para la búsqueda de similitud.
- **get_line_tokens:** devuelve los token encontrados en la línea pasada como parámetro. Al igual que ocurre en "get_all_tokens", la función "get_line_tokens" no incluye en su código la adaptación del resultado.

similarity_generic.php

Para VPL, cada lenguaje de programación que soporte la búsqueda de similitud dispone obligatoriamente de una clase específica que se encarga de la construcción de un tokenizador. Este hecho en VPLT supone que, cada vez que se añada un nuevo lenguaje, se tendrá que definir un fichero con este tipo de clase. Para solucionar este problema, VPLT ofrece en "similarity_generic.php" una alternativa que permite crear clases de este tipo genéricas.

En "similarity_generic.php", la clases de similitud se construyen a partir de una cadena de caracteres que codifica el nombre de un lenguaje de programación. A partir de esta información, y con la clase de "tokenizer_factory.php", se asocia un número identificador y un tokenizador al lenguaje en cuestión. Cabe mencionar que, a diferencia de otros casos, el fichero "similarity_generic.php" es un elemento adicional que no existía en el módulo VPL.

Por un lado, las propiedades definidas en "similarity_generic.php" son:

- **tokenizerclass:** cadena de caracteres que representa el nombre descriptivo de un lenguaje de programación determinado. Como era de esperar, el valor de esta propiedad debe ser la misma que se especifica en el nombre del fichero de reglas correspondiente.
- **typenumber:** valor numérico entero que identifica a la clase de similitud que se construye. En este caso, la asignación es incremental, lo que significa que, cada vez que se crea una nueva clase, el valor de su tipo será el número siguiente al anterior.

Por otro lado, las funciones públicas de "similarity_generic.php" son:

- **get_type:** devuelve el identificador asociado con la clase de similitud actual.
- **get_tokenizer:** devuelve el tokenizador del lenguaje de programación dado que se obtiene a partir de la clase del fichero "tokenizer_factory.php".

similarity_factory.php

En el fichero "similarity_factory.php" se dispone de una clase que se encarga de construir automáticamente una instancia de la clase de similitud de un lenguaje de programación. Al

igual que ocurre en ”tokenizer_factory.php”, en VPL ya se había creado una clase similar. La diferencia de esta versión es que, en el proceso de construcción, aparece una nueva clase genérica que se emplea para los lenguajes que no tienen ninguna clase específica para la búsqueda de similitud.

El proceso de construcción realizado en ”similarity_factory.php” consta de una serie de pasos. Primero, se intenta crear la instancia mediante una clase específica situada en el espacio de nombres ”mod_vpl/similarity”. En caso de que no esté dicha clase, el proceso trata de crearla con la clase genérica de ”similarity_generic.php”. Finalmente, si ”similarity_generic” no tiene soporte para el lenguaje, lo último que se hace es intentar construir la clase de similitud mediante la clase antigua de VPL.

Al igual que ocurre en ”tokenizer_factory.php”, la función que se encarga de construir las clases de similitud se conoce por el nombre ”get”. En este caso, sólo es necesario incluir un parámetro que indique el nombre del lenguaje que se tokeniza.

assertf.php

El fichero ”assertf.php” es una utilidad adicional del motor VPLT que proporciona una serie de funciones con las que incluir aserciones que muestran los errores en mensajes presentados con una buena apariencia, y que pueden personalizarse fácilmente. En este caso, se disponen de las siguientes funciones:

- **get_error:** devuelve una cadena de caracteres formateada que representa un mensaje de error que informa sobre la causa del fallo y el fichero que lo provocó.
- **assert:** verifica si se cumple una condición determinada. En caso de que el resultado no sea existoso, la función muestra una excepción con un mensaje de error definido en la función ”get_error”.
- **showerr:** verifica una condición, y muestra por el canal de salida un mensaje de error en caso de que dicha condición no se cumpla.

5.4. Pruebas automáticas del programa

Antiguamente, la corrección de errores en el software era una tarea que tenía poca relevancia en el desarrollo de una aplicación, y que además pertenecía a las últimas fases del proyecto. En la mayoría de los casos, los programas que se desarrollaban con esta estrategia acababan siendo abandonados por sus creadores debido a sus costos elevados de su mantenimiento y mejora. Estos problemas provocaron que muchos proyectos acabasen optando por emplear durante todo el proceso del desarrollo, una serie de técnicas y herramientas con las que se puede evaluar y verificar el código fuente. En Ingeniería Informática, el conjunto de estas técnicas y utilidades se las conoce como pruebas de software o "software testing".

Las pruebas de software son la agrupación de los procesos y técnicas con las que se puede revisar y analizar las funcionalidades incorporadas en el software con el fin de informar sobre la calidad y fidelidad de un producto. A partir de estas pruebas, se puede lograr reducir los costos del desarrollo, optimizar el rendimiento de la aplicación, y, en definitiva, corregir la mayor parte de los errores del código.

Como era de esperar, las pruebas de software se ha utilizado durante el desarrollo del motor VPLT como recurso fundamental con el que verificar que los resultados devueltos por el programa son correctos. Concretamente, se han definido un lote de pruebas unitarias y otras basadas en el comportamiento. En las siguientes secciones se explica con mayor profundidad cada uno de éstos. Cabe mencionar que las pruebas no han sido la única herramienta empleada para la verificación de resultados, sino que también se ha incorporado código con el que contrastar los resultados del motor con respecto a los que devuelve VPL.

5.4.1. Pruebas unitarias

Se entiende como prueba unitaria como el conjunto de comprobaciones que verifican el funcionamiento de un componente individual de una aplicación. Dicho de otra forma, las pruebas unitarias son el mecanismo con el que se puede evaluar que la ejecución de una pequeña parte del código se hace según lo esperado. Este tipo de pruebas son ideales para cuando se ha construído el software por medio de un conjunto de módulos, que se caracterizan por ser

remplazables y se encargan de cubrir cada uno una parte de la funcionalidad del programa.

En el motor VPLT, se han desarrollado las pruebas unitarias a partir de un framework conocido como PHPUnit. Con esta herramienta, se realizaron pruebas que comprueban, para una gran variedad de casos o situaciones posibles, todas las partes que conforman el motor de análisis léxico. En la figura 5.14, se puede observar desde una consola de comandos que las pruebas unitarias desarrolladas en el motor ofrecen resultados satisfactorios.

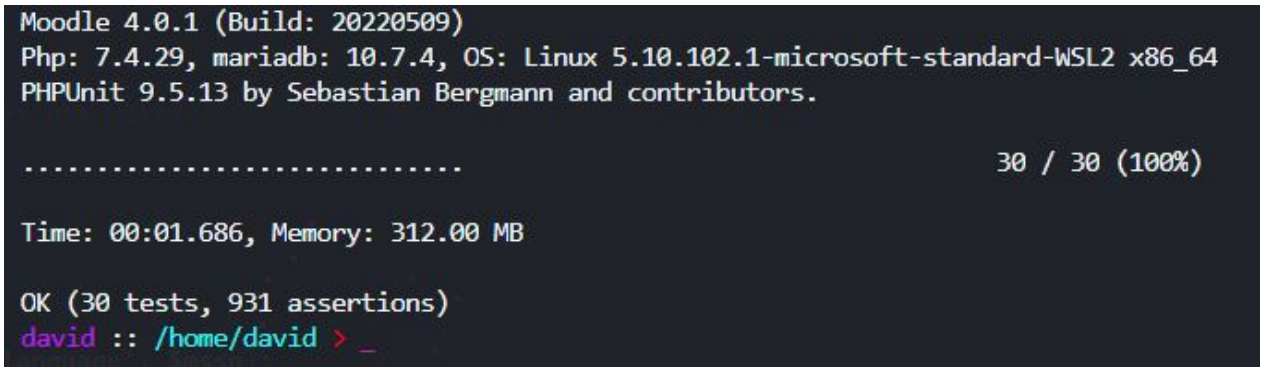

Figura 5.14: Resultados de las pruebas unitarias

Una de las características más interesantes del framework PHPUnit es la posibilidad de incoporar en las pruebas unitarias un tipo de métrica conocida como cobertura de código o ”code coverage”. Esta métrica mide el grado en que se ha comprobado el código de un producto software, por lo que, en cierto modo, también sirve para determinar la calidad de las pruebas desarrolladas. En el caso de VPLT, la inclusión de la cobertura de código ha facilitado enormemente el desarrollo de las pruebas unitarias. Tal y como se observa en la figura 5.15, las pruebas del motor VPLT han llegado a tener una cobertura de código con valores satisfactorios superiores al 95 %.

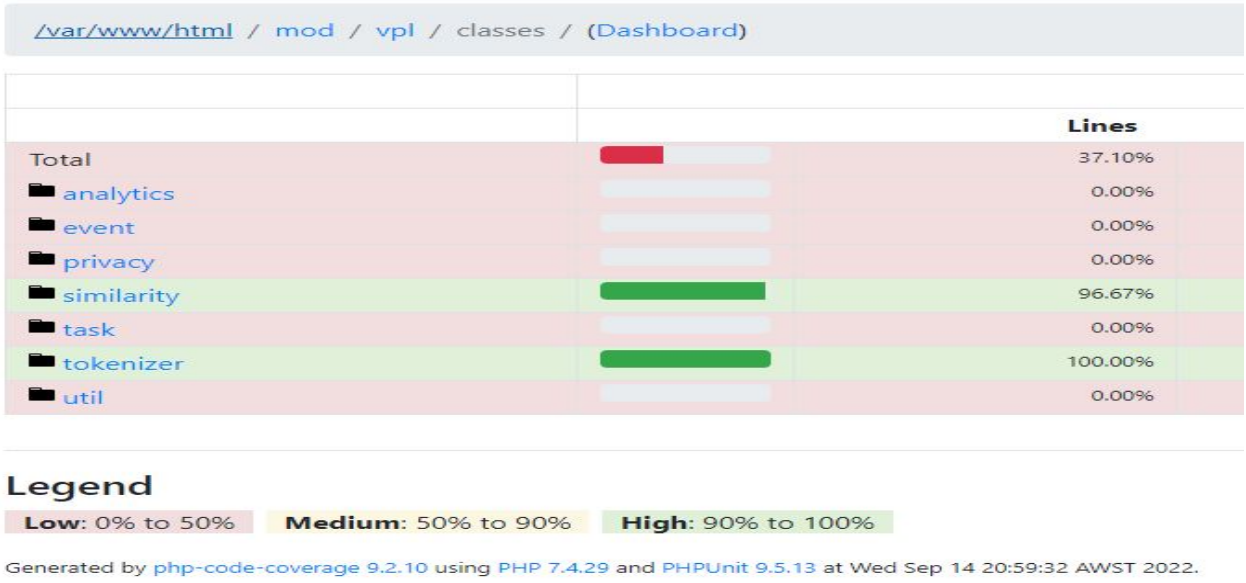

Figura 5.15: Resultados de la cobertura de las pruebas unitarias

5.4.2. Pruebas de comportamiento

Una prueba de comportamiento se define como la descripción en lenguaje natural de una serie de sucesos que deben ocurrir en un momento concreto durante la ejecución de una aplicación. Este tipo de pruebas están principalmente orientadas tanto al usuario como al propio comportamiento del sistema, y suelen emplearse no sólo por los desarrolladores del software, sino también por el resto del equipo.

Esencialmente, la razón principal de que VPLT disponga de pruebas de comportamiento se debe a que se necesita de alguna herramienta que verifique la parte de la interfaz de usuario del módulo VPL en la que esté involucrada el motor de análisis léxico. Asimismo, con estas pruebas de comportamiento se puede comprobar, en aspectos generales, si VPL procesa correctamente los resultados devueltos por el motor. Cabe mencionar que, la definición de este tipo de pruebas se llevó a cabo por medio del framework Behat.

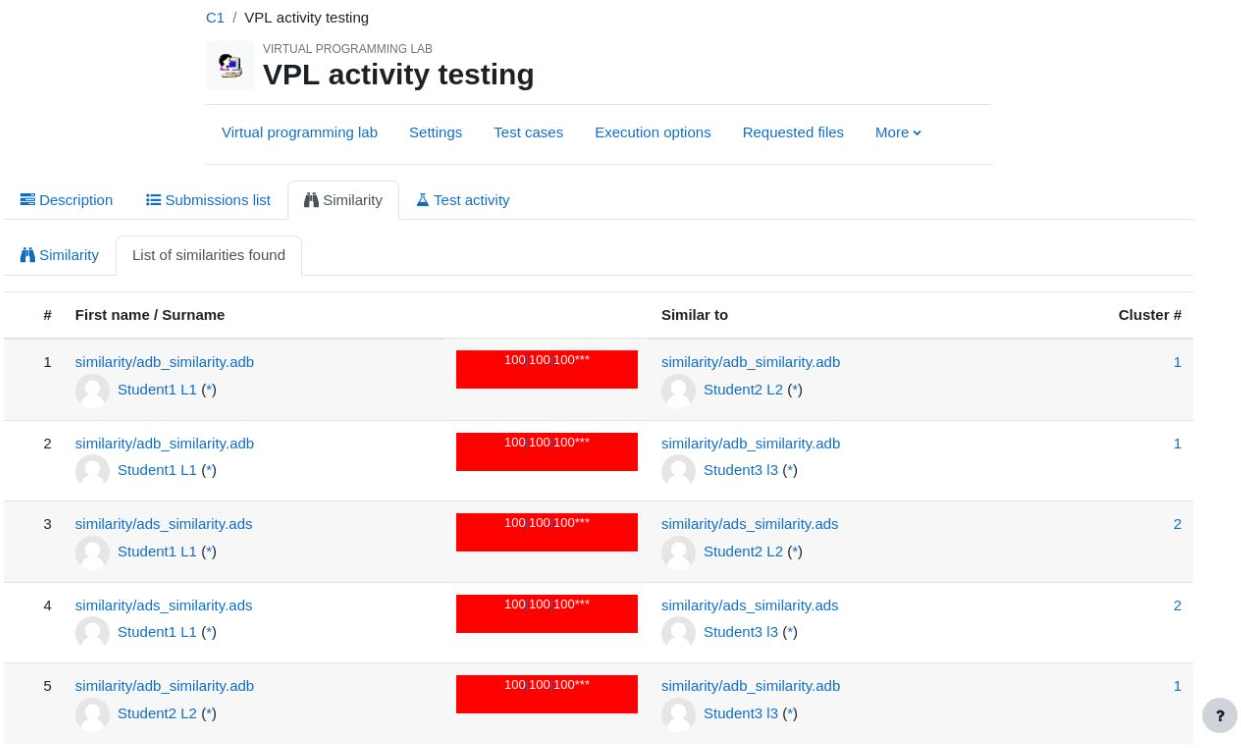

Figura 5.16: Ejemplo de la visualización de las pruebas de comportamiento

5.4.3. Evaluación con los resultados de VPL

Como ya se ha comentado en repetidas ocasiones, el objetivo principal del motor de análisis léxico VPLT es proporcionar un sistema que sustituya los analizadores léxicos de la búsqueda de similaridad del módulo VPL. Para que este remplazo pueda hacerse, los resultados devueltos por VPLT deben ser los mismos que se devuelven actualmente por los tokenizadores de VPL. Esto implica que, una manera de comprobar que VPLT funciona correctamente, es mediante la comparación de los resultados de ambos sistemas. Este tipo de verificación se ha utilizado en VPLT para todos los lenguajes de programación que dispongan de soporte en VPL.

Capítulo 6

Conclusiones y trabajos futuros

En este capítulo se ofrece, por un lado, una síntesis de los resultados finales conseguidos mediante el desarrollo de este trabajo de fin de título, y por otro lado, el listado de cambios que se podrían incorporar al proyecto en un futuro. Las conclusiones recogidas sirven como resumen con el que se engloban cada una de las cuestiones y observaciones detalladas en este documento, y también proporciona algunas de las ideas que se podrían explorar para la mejora y el avance del sistema propuesto.

6.1. Conclusiones

Con este trabajo de fin de título se ha conseguido desarrollar un motor de análisis léxico que sustituya los analizadores léxicos independientes de la búsqueda de similaridad de VPL. Este sistema es capaz de llevar a cabo procesamiento léxico mediante una máquina de estados que se define a partir de un conjunto de reglas especificadas en ficheros JSON, y permite una mayor extensibilidad de la tokenización incorporada en el módulo mencionado.

Si bien con el motor desarrollado no se pueden definir gramáticas de lenguajes muy complejas que necesiten de funcionalidades avanzadas, sí que es un sistema inicial que sirve como punto de partida para futuras actualizaciones del motor o nuevas aplicaciones de tokenización que traten el problema del plagio u otras cuestiones.

6.2. Trabajos futuros

Como ocurre con cualquier otro producto software, el motor de análisis léxico que se ha desarrollado en este proyecto podría tener una serie de carencias que, por falta de tiempo, no se han podido cubrir. A continuación, se proponen algunas de las mejoras que podrían hacerse para incrementar las funcionalidades del motor:

- Como primer punto, sería interesante **mejorar todos los ficheros de reglas incorporados en el motor** para ofrecer al usuario resultados completos y precisos que coincidan con los devueltos por el tokenizador actual de VPL.
- La **definición de nuevos ficheros de reglas** para los lenguajes de programación soportados por VPL, para así sustituir por completo los tokenizadores que se emplean actualmente en las búsquedas de similaridad.
- La **incorporación de una pila** en el modelo conceptual del motor, que actualmente está basado en los autómatas finitos deterministas, para que se puedan definir léxico avanzado de ciertos lenguajes que requiera de memoria.
- Cabría plantearse **proponer nuevos sistemas de detección de reglas infinitas** que ofrezcan un planteamiento diferente al procedimiento actual. Con este cambio, se conseguiría una mejoría en el rendimiento y velocidad, y un menor mantenimiento de los ficheros de reglas, que, a fecha de este proyecto, tienen que actualizarse cada vez que aparecen casos que superen el número de tokens permitidos.
- Por último, un gran avance sería disponer de una función en el motor que admita **heredar un grupo de reglas de un fichero**, ya que así se podría plantear la herencia múltiple en la que se extraiga de cada fichero de reglas sólo la parte que interese para cada caso concreto.

Anexo A

Lenguajes soportados

En este anexo, se ofrece una lista de los lenguajes de programación que están actualmente soportados en el motor de análisis léxico VPLT. La información presentada en este listado incluye para cada lenguaje una explicación breve sobre el mismo, una tabla con datos sobre la especificación técnica de dicho lenguaje en VPLT, y finalmente la razón de su incorporación en el motor y sus limitaciones léxicas.

Ada

Ada es un lenguaje de programación orientado a objetos e imperativo cuyas características lo convierten en un lenguaje flexible, sencillo, y adecuado para el desarrollo de sistemas críticos que requieran de una alta seguridad y fiabilidad.

Característica	**Detalles**
Tokenizador	ada-tokenizer
Extensiones	ada, adb, ads
Compilador/Intérprete	GNAT (GNU Ada) http://www.gnu.org/software/gnat/
Similaridad antigua de VPLT	Sí
Versiones	Disponible Ada 2005 y Ada 2012. Ada 2005 es el compilador predeterminado.

Tabla A.1: Información técnica de Ada para VPLT

En este caso, se decidió ofrecer soporte del lenguaje Ada en el motor VPLT por una cuestión meramente personal. El autor del presente trabajo de fin de título está familiarizado con Ada y lo ha utilizado para el desarrollo de varios proyectos académicos que prueban las características del lenguaje de programación.

En principio, el motor de análisis léxico VPLT es capaz de llevar a cabo, para todos los casos posibles, el procesamiento léxico de ficheros escritos en el lenguaje de programación Ada. Ahora bien, esto no implica que exista soporte en el motor de los dialectos de Ada, tales como SPARK o el perfil de Ravenscar.

C

C es un lenguaje de programación compilado, imperativo y procedural que se diseñó con el propósito de ofrecer un lenguaje eficiente, completo, y ligero con el que crear fácilmente sistemas operativos robustos. Hoy en día, C ha sido la inspiración de múltiples lenguajes de programación modernos tales como Java, Python, C++, entre otros.

Característica	**Detalles**
Tokenizador	c-tokenizer
Extensiones	c, h
Compilador/Intérprete	GNU GCC https://gcc.gnu.org/
Similaridad antigua de VPLT	Sí
Versiones	Disponible ANSI-C, C99 y C11. La versión predeterminada empleada es la del propio compilador.

Tabla A.2: Información técnica de C para VPLT

Como ya se dijo anteriormente, C ha sido la influencia directa de muchos lenguajes de programación actuales. De hecho, tal ha sido su repercusión, que incluso el léxico, sintaxis, y semántica de estos lenguajes derivados se fundamenta en el del propio C. Sabiendo esto, es lógico pensar que la incorporación de C en VPLT puede llegar a suponer una gran ventaja, pues para añadir el soporte de los lenguajes que derivan de C, sólo se necesita definir reglas

para casos particulares que no se puedan heredar de las de C.

Sin embargo, el soporte del lenguaje C en el motor VPLT no está del todo completo y presenta una serie de limitaciones. De entre todas estas, la más relevante es que el sistema no es capaz de detectar macros muy complejas cuya definición esté en varias líneas de código. En estos casos, es probable que el motor no funcione adecuadamente, y devuelva resultados no sólo incoherentes para las macros, sino también para el resto de tokens.

C++

C++ es un lenguaje de programación de propósito general que presenta las mismas características que las de su predecesor C, pero que incorpora un nuevo paradigma conocido como programación orientada a objetos. En cierto modo, C++ sería como una versión mejorada del lenguaje C con características nuevas que facilitan el desarrollo de sistemas informáticos.

Característica	**Detalles**
Tokenizador	cpp-tokenizer
Extensiones	cc, cpp, C, c++, hxx, H
Compilador/Intérprete	GNU C++ https://gcc.gnu.org/projects/cxx-status.html
Similaridad antigua de VPLT	Sí
Versiones	Disponible C++98, C++11, C++14, y C++17. La versión predeterminada que se emplea en el motor de análisis léxico es la del propio compilador.

Tabla A.3: Información técnica de C++ para VPLT

Tal y como se comentó previamente, C++ es un lenguaje de programación que extiende las características de C a partir de la incorporación de otras funcionalidades. Para el motor VPLT, este hecho significa que el léxico de C++ es, en esencia, el mismo que el de su predecesor, pero con reglas adicionales. Todo esto justifica el motivo de que se haya incoporado en el motor VPLT el lenguaje C++, que por la operación de herencia sólo necesita la especificación

de estas nuevas reglas para estar totalmente completo.

Si bien la inclusión de las reglas de C en C++ es una estrategia con la que se facilita enormemente la especificación del léxico, también tiene una serie de inconvenientes. De entre todos estos, destaca su gran dependencia a las reglas que se heredan. Este problema provoca no sólo que exista durante la modificación de las reglas un mayor riesgo de que el léxico deje de ser correcto, sino que también los errores que habían en las reglas heredadas se propagen a las del propio lenguaje especificado. En definitiva, el soporte del motor VPLT del lenguaje C++ presenta los mismos problemas que ocurrían en C con las macros.

Fortran

Fortran es un lenguaje de programación de propósito general, imperativo, y compilado, que se emplea principalmente para el desarrollo de sistemas informáticos en los que se aplique, por encima de cualquier otro proceso, el cálculo numérico y la computación científica.

Característica	**Detalles**
Tokenizador	fortran-tokenizer
Extensiones	f, F
Compilador/Intérprete	GNU GCC Fortran https://gcc.gnu.org/fortran/
Similaridad antigua de VPLT	No

Tabla A.4: Información técnica de Fortran para VPLT

El motivo de que se haya optado por incoporar Fortran al motor de análisis léxico VPLT es por el hecho de que se trata de otro lenguaje que le es familiar al autor y con el que se puede emplear el motor sin reglas que se hayan heredado con la operación de herencia.

A diferencia de otros lenguajes de programación, Fortran no tiene soporte alguno en la búsqueda de similaridad del módulo de Moodle VPL. Esto implica que, para el desarrollo de su léxico en VPLT, no se tiene de ninguna referencia con la que verificar que el procesamiento léxico funcione según lo esperado, por lo que no se ha podido asegurar que el fichero de reglas

de Fortran defina correctamente el léxico del lenguaje. Luego, podría decirse que la limitación principal en este caso sería que VPLT no garantiza que el procesamiento léxico pueda hacerse.

Java

Java es un lenguaje de programación de propósito general, de alto nivel, compilado, y orientado a objetos con el que se puede llevar a cabo el desarrollo de código fuente seguro y fiable.

Característica	**Detalles**
Tokenizador	java-tokenizer
Extensiones	java
Compilador/Intérprete	Java https://www.oracle.com/java/ y Java Development Kit (JDK) https://openjdk.java.net/
Similaridad antigua de VPLT	Sí

Tabla A.5: Información técnica de Java para VPLT

En este caso, se decidió añadir soporte en el motor de análisis léxico VPLT para Java por ser un lenguaje muy empleado en el ámbito académico que presenta un léxico que puede definirse fácilmente a partir de las herramientas de las que se dispone para la definición de un fichero de reglas. Asimismo, el autor del presente trabajo de fin de título ha programado en este lenguaje en varias ocaciones y está familiarizado con su léxico y sintaxis.

Actualmente, las reglas de Java en el motor VPLT sólo cubren una pequeña parte del léxico del lenguaje. Este hecho se debe a que, para esta primera propuesta, se ha preferido plantear un léxico sencillo que sirva como demostración de que el motor de análisis léxico VPLT funciona correctamente.

Scheme

Scheme es uno de los dialectos disponibles de Lisp que se caracteriza por ser un lenguaje funcional e imperativo cuya filosofía es la obtención de la abstracción minimalista del código fuente por medio de la recursividad y con estructuras de datos sencillas.

Característica	**Detalles**
Tokenizador	scheme-tokenizer
Extensiones	scm, ss
Compilador/Intérprete	https://racket-lang.org/
Similaridad antigua de VPLT	Sí

Tabla A.6: Información técnica de Scheme para VPLT

En pocas palabras, la razón de que Scheme sea uno de los lenguajes de programación que se haya incorporado en el motor de análisis léxico VPLT se explica por su sencillez en su léxico. A diferencia de lo que ocurre con otros lenguajes, Scheme no tiene demasiados símbolos, lo que hace que la especificación en el motor VPLT suponga un muy bajo costo para su desarrollo y mantenimiento.

Para este caso, el fichero de reglas que contiene el léxico de Scheme es lo suficientemente completo como para abordar todos los aspectos léxicos del lenguaje de programación. En definitiva, Scheme en el motor léxico VPLT no presenta ninguna limitación ni problema, y en teoría, podría ya incluirse en el módulo VPL.

Anexo B

Tokens disponibles

El presente anexo muestra un listado documentado de todos los tipos de tokens admitidos por el motor de análisis léxico VPLT. Como este listado es muy grande, se ha dividido la información en una serie de cuadros según el tipo de token. Cabe mencionar que, las tablas recogidas en este apéndice contienen una explicación breve y sencilla para cada token.

Tokens para comentarios

Nombre	**Detalles**
comment	Cualquier tipo de comentario.
comment.line	Comentarios de una línea.
comment-line.double-slash	Comentarios de una línea representados con "//".
comment-line.double-dash	Comentarios de una línea representados con "–".
comment.line.number-sign	Comentarios de una línea representados con "#".
comment.line.percentage	Comentarios de una línea representados con "%".
comment.line.character	Comentarios de una línea con otros caracteres.
comment.block	Comentarios en bloque o de varias líneas.
comment.block.documentation	Comentarios de varias líneas para la documentación.

Tabla B.1: Tokens para comentarios disponibles

Tokens para constantes y literales

Nombre	Detalles
constant	Valores constantes, literales, entre otros.
constant.numeric	Constantes numéricas.
constant.character	Constantes de caracteres especiales.
constant.character.escape	Constantes de caracteres con un carácter de escape.
constant.language	Constantes especiales proporcionadas por el lenguaje de programación.
constant.language.escape	Constantes especiales proporcionadas por el lenguaje con un carácter de escape.
constant.other	Cualquier otro tipo de constante no contemplada.

Tabla B.2: Tokens para constantes disponibles

Tokens de etiquetas

Nombre	Detalles
entity	Palabras concretas de entidades para funciones, etiquetas, clases, secciones, entre otros.
entity.name	Nombres de las entidades.
entity.name.function	Nombres de funciones.
entity.name.type	Nombres de tipos declarados o clases.
entity.name.tag	Nombres de etiquetas de marcado.
entity.name.section	Nombres de secciones o encabezados.
entity.other	Otro tipo de entidades no contempladas.
entity.other.inherited-class	Declaración de la superclase o la clase de base.
entity.other.attribute-name	Nombre de un atributo de una etiqueta.

Tabla B.3: Tokens para etiquetas disponibles

Tokens para lenguaje de marcado

Nombre	Detalles
markup	Tokens para lenguajes de marcado.
markup.underline	Texto subrayado.
markup.unerline.link	Texto subrayado definido como enlaces.
markup.bold	Texto en negrita.
markup.heading	Encabezados.
markup.italic	Texto en itálico.
markup.list	Listas.
markup.list.numbered	Listas numeradas.
markup.list.unnumbered	Listas no numeradas.
markup.quote	Texto entrecomillado.
markup.raw	Texto escrito en crudo, como por ejemplo código fuente.
markup.other	Otro tipo de tokens para lenguajes de marcado.

Tabla B.4: Tokens para lenguajes de marcado

Tokens de palabras clave

Nombre	Detalles
keyword	Palabras reservadas para un propósito concreto.
keyword.control	Palabras clave reservadas para el control del código.
keyword.operator	Palabras clave reservadas para los operadores.
keyhword.other	Cualquier otro tipo de palabra clave.

Tabla B.5: Tokens para palabras clave disponibles

Almacenamiento de datos

Nombre	Detalles
storage	Tokens para el almacenaje de datos.
storage.type	Tipo de dato en el que se almacena.
storage.modifier	Modificador de acceso del dato almacenado.

Tabla B.6: Tokens para el almacenaje de datos disponibles

Tokens para cadenas de caracteres

Nombre	Detalles
string	Cadenas de caracteres o ristras.
string.quoted	Ristras entrecomilladas.
string.quoted.single	Ristras entrecomilladas con comillas simples.
string.quoted.double	Ristras entrecomilladas con comillas dobles.
string.quoted.triple	Ristras entrecomilladas con comillas triples.
string.quoted.other	Otro tipo de ristras entrecomilladas.
string.unquoted	Ristras no entrecomilladas.
string.interpolated	Ristras que forman expresiones que se evaluan.
string.regexp	Ristras que forman expresiones regulares.
string.other	Otro tipo de ristras.

Tabla B.7: Tokens para ristras disponibles

Tokens para soporte de frameworks o librerías

Nombre	Detalles
support	Proporcionados por frameworks o librerías externas.
suppport.function	Funciones de una librería o framework.
support.class	Clases de una librería o framework.
support.type	Tipos de una librería o framework.
support.constant	Constantes de una librería o framework.
support.variable	Variables de una librería o framework.
support.other	Otro tipo de soporte de una librería o framework.

Tabla B.8: Tokens para el soporte de frameworks o librerías

Tokens generales

Nombre	Detalles
meta	Para el marcado de una parte larga de un documento.
identifier	Nombre descriptivo genérico de un token.
punctuation	Símbolo de puntuación.
punctuation.separator	Símbolo de puntuación empleado como separador.
paren	Paréntesis de comienzo o de final.
paren.lparen	Paréntesis de comienzo ”(”.
paren.rparen	Paréntesis de final ”)”.
text	Cualquier fragmento de texto.

Tabla B.9: Tokens para tokens generales disponibles

Tokens de variables

Nombre	Detalles
variable	Variables definidas en el código fuente.
variable.parameter	Parámetro de una función o método.
variable.language	Variables especiales reservadas por el lenguaje de programación.
variable.other	Otro tipo de variables.

Tabla B.10: Tokens para variables disponibles

Tokens crudos

Nombre	Definición
vpl_identifier	Secuencia de caracteres que representan el identificador o nombre descriptivo de una variable, función, atributo, nombres de clases, entre otros.
vpl_literal	Reservado para constantes, y valores literales de cualquier tipo de dato como números, valores booleanos, cadenas de caracteres, entre otros.
vpl_operator	Asociado con los símbolos y signos que indican un operador del lenguaje de programación, como por ejemplo, sumas, restas, operador de módulo, entre otros.
vpl_reserved	Similar al tipo "vpl_identifier", pero que está destinado para las palabras reservadas del lenguaje, que sólo tienen un significado, y no pueden utilizarse para otros símbolos.
vpl_other	Cualquier otro tipo de token no contemplado.
vpl_null	Indica los tipos de tokens que se descartan en el proceso de análisis léxico o tokenización.

Tabla B.11: Tokens crudos disponibles

Anexo C

Manual de Usuario

Como ya se ha comentado anteriormente, el motor de análisis léxico VPLT es una herramienta que estará incorporada en futuras versiones del módulo de Moodle VPL. Su código y manual de usuario estarán disponibles respectivamente en el repositorio oficial de VPL, y en su página web. Para facilitar la búsqueda de estos recursos, en este anexo se han incluido todos los enlaces relevantes en los que se puede acceder al código y documentación de VPLT.

Es importante mencionar que puede que en este momento VPLT aún no esté incorporado de manera oficial en VPL. Para esos casos, se recomienda visitar los repositorios de Github en los que se encuentra el código fuente base del motor de análisis léxico.

Enlaces de VPL

- **Página web oficial:** https://vpl.dis.ulpgc.es/
- **Repositorio:** https://github.com/jcrodriguez-dis/moodle-mod_vpl

Enlaces de VPLT

- **Repositorio:** https://github.com/losedavidpb/moodle-mod_vpl/tree/v3.5.0++
- **Manual de usuario:** https://github.com/losedavidpb/tokenizer-doc_vpl

Bibliografía

[1] N. Johansson y A. Löfgren, «Designing for Extensibility: An action research study of maximizing extensibility by means of design principles,» Tesis de mtría., Universidad de Gothenburg, 2009.

[2] L. A. Mullen, K. Benoit, O. Keyes, D. Selivanov y J. Arnold, «Fast, consistent tokenization of natural language text,» *Journal of Open Source Software*, vol. 3, n.º 23, pág. 655, 2018.

[3] Ajax.org, *Ace - The High Performance Code Editor for the Web*, https://ace.c9.io/, 2022.

[4] Real Academia Española y Asociación de Academias de la Lengua Española, *Léxico*, https://dle.rae.es/l%C3%A9xico, 2022.

[5] Jacky, *Compilation Process Steps*, http://jackyhjj.blog.binusian.org/author/jackyhjj/, 2014.

[6] A. V. Aho, R. Sethi y J. D. Ullman, *Compiladores: principios, técnicas y herramientas*. Pearson Educación, 1990.

[7] M. Andrade, *Writing your own programming language and compiler with Python*, https://medium.com/@marcelogdeandrade/writing-your-own-programming-language-and-compiler-with-python-a468970ae6df, 2018.

[8] Wikipedia, *Teoría de autómatas*, https://es.wikipedia.org/wiki/Teor%C3%ADa_de_aut%C3%B3matas, 2022.

[9] Wikipedia, *Autómata finito*, https://es.wikipedia.org/wiki/Aut%C3%B3mata_finito, 2022.

[10] C. Dante, *Autómatas de pila*, https://cristian-dante94.blogspot.com/2017/07/automatas-de-pila.html, 2017.

[11] Escuela de Ingeniería Informática, ULPGC, «Máquinas de Turing,» 2019.

[12] Rodríguez del Pino, J.C., «Integridad académica en la docencia universitaria actual con énfasis en el plagio del código fuente: modelo, propuesta de intervención y herramientas,» Tesis doct., Universidad de Las Palmas de Gran Canaria, 2016.

[13] M. E. Lesk y E. Schmidt, *Lex: a lexical analyzer generator*. Bell Laboratories Murray Hill, NJ, 1990.

[14] J. Levine, *Flex & Bison: Text Processing Tools*. O'Reilly Media, Inc., 2009.

[15] R. van Engelen, *Constructing Fast Lexical Analyzers with RE/flex*, https://www.genivia.com/reflex.html, 2017.

[16] P. Bumbulis y D. D. Cowan, «RE2C: A More Versatile Scanner Generator,» *International Journal of Applied Engineering and Management Letters (IJAEML)*, vol. 2, n.º 1-4, 1993.

[17] ULPGC, *Objetivos y Competencias del GII*, https://www2.ulpgc.es/archivos/plan_estudios/4008_40/ObjetivosyCompetenciasdelGII.pdf, 2022.

[18] PHP Group, *PHP: Hypertext Preprocessor*, https://www.php.net/, 2022.

[19] Wikipedia, *PHP*, https://es.wikipedia.org/wiki/PHP, 2022.

[20] Rodríguez del Pino, J.C., *VPL - Virtual Programming Lab - Home*, https://vpl.dis.ulpgc.es/, 2022.

[21] The Moodle Project, *Moodle - Open-source learning platform*, https://moodle.org/?lang=es, 2022.

[22] Rodríguez del Pino, J.C., *Virtual Programming Lab for Moodle (VPL)*, https://vpl.dis.ulpgc.es/documentation/vpl-3.4.3+/, 2021.

[23] S. Bergmann, *PHPUnit - The PHP Testing Framework*, https://phpunit.de/, 2022.

[24] .NET Foundation, *xUnit.net*, https://xunit.net/, 2022.

[25] K. Kudryashov, *Behat - a php framework for autotesting your business expectations*, https://docs.behat.org/en/latest/, 2016.

[26] A. Hellesøy, J. Wilk, M. Wynne, G. Hnatiuk y M. Sassak, *BDD Testing & Collaboration Tools for Teams — Cucumber*, https://cucumber.io/, 2022.

[27] Microsoft, *Visual Studio Code - Code Editing. Redefined*, https://code.visualstudio.com/, 2022.

[28] L. Torvalds, *Git*, https://git-scm.com/, 2021.

[29] Github, Inc., *Github: Where the world builds software*, https://github.com/, 2022.

[30] Docker, Inc., *Home - Docker*, https://www.docker.com/, 2022.

[31] F. Pérez, *Para qué nos sirve el popular Docker*, https://www.latirus.com/blog/2020/09/27/para-que-nos-sirve-el-popular-docker/, 2020.

[32] Docker Inc., *Docker Desktop - Docker*, https://www.docker.com/products/docker-desktop/, 2022.

[33] Microsoft, *Windows Subsystem for Linux Documentation — Microsoft Docs*, https://docs.microsoft.com/en-us/windows/wsl/, 2022.

[34] J. Walker, *How to install Windows 10's Linux Subsystem on your PC*, https://www.onmsft.com/news/how-to-install-windows-10s-linux-subsystem-on-your-pc, 2018.

[35] G. Brandl, *Welcome - Sphinx documentation*, https://www.sphinx-doc.org/en/master/, 2022.

[36] G. Brandl, *Introducción a reStructuredText*, http://www.pythondoc.com/sphinx/rest.html, 2011.

[37] T. U. Group y K. Berry, *TeX Live - TeX Users Group*, https://tug.org/texlive/, 2022.

[38] Wikipedia, *TeX Live*, https://es.wikipedia.org/wiki/TeX_Live, 2020.