# Descripción del sistema

La Figura 1 muestra el diagrama de la arquitectura de nuestro sistema. A continuación, describimos someramente sus elementos para posteriormente profundizar en los mismos. La entrada del sistema es un conjunto de programas Java obtenidos tanto de GitHub como de dos asignaturas de primer año de programación de un Grado en Ingeniería Informática del Software de la Universidad de Oviedo (Capítulo 4.1), para utilizar código escrito por principiantes y expertos. La salida es una colección de los patrones sintácticos más frecuentemente utilizados, su relación con el nivel de experiencia del programador y un informe acerca de los patrones atípicos detectados.

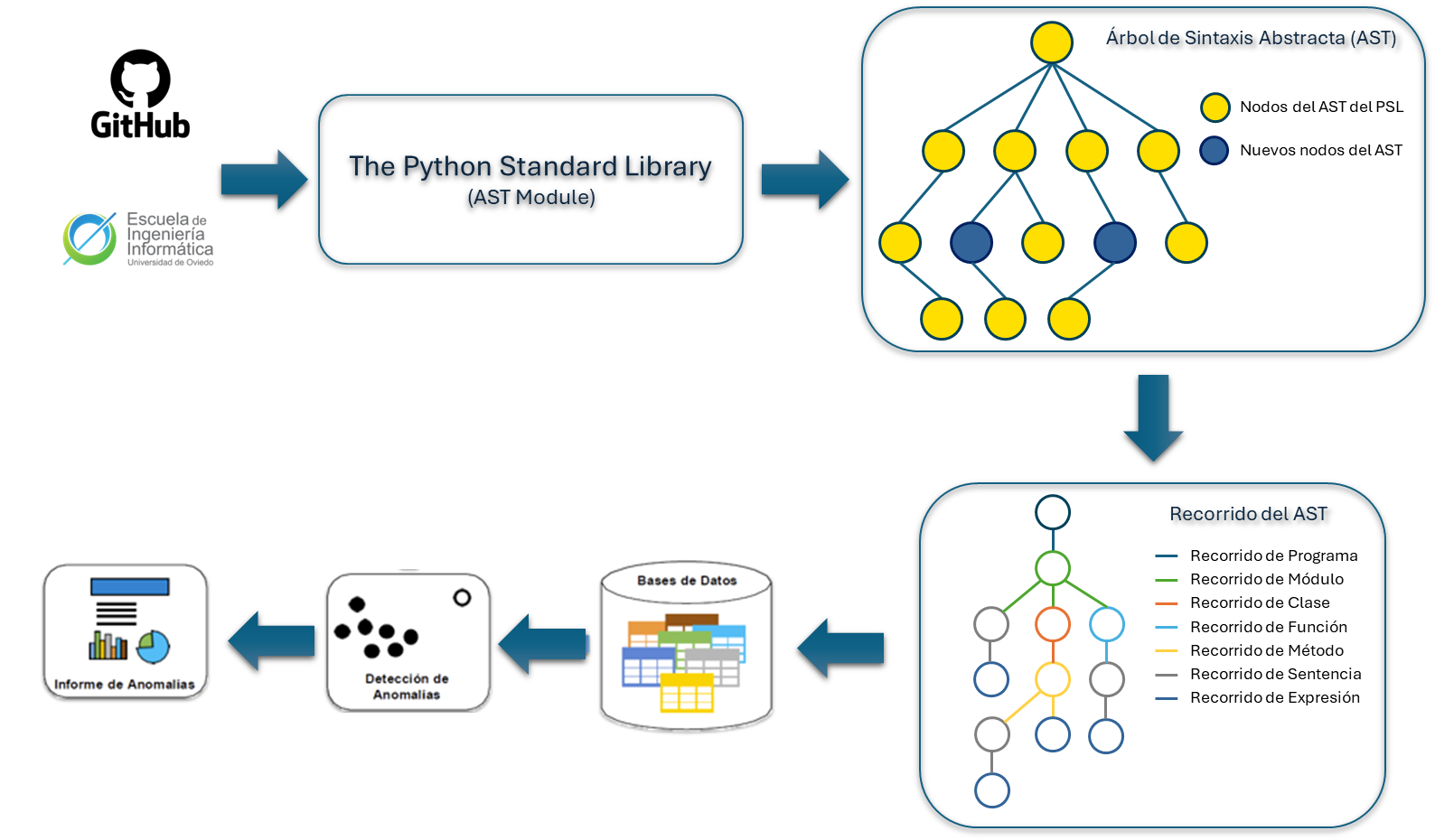


Figura 1: Arquitectura del sistema propuesto para la extracción de información sintáctica.

Lo primero que hacemos es modificar el compilador de Java del OpenJDK, aumentando la información sintáctica ofrecida por éste. Para ello, rediseñamos su AST, añadiendo nuevos nodos y relaciones, proporcionando así una información mucho más detallada que la ofrecida por el compilador original (Capítulo 3.1).

Establecemos una clasificación de los distintos tipos de subárboles con estructura común, encontrando los siguientes 7 tipos de construcciones sintácticas: programa (compuesto como una colección de tipos definidos en él), definición de tipo (clase, interfaz, enumerado o registro), definición de campo, definición de método, sentencia (o instrucción), expresión y tipo. Se definen 7 algoritmos de recorrido de estos tipos de subárboles, guardando la información de cada construcción sintáctica en 7 tablas distintas (Capítulo 3.3).

Posteriormente, los datos de las tablas son procesados, combinados y filtrados para crear los *datasets* que se utilizarán como entrada de los algoritmos de extracción de reglas y detección de anomalías. Si las anomalías son debidas a errores de medición (entradas que no debían haber sido consideradas) se eliminan; en caso contrario, se incluyen en un informe de anomalías y se consideran en las reglas de asociación (Capítulo 3.7). Los patrones sintácticos son finalmente generados por el sistema.

## Modificación del compilador de Java

El primer paso consiste en implementar una versión modificada del compilador de Java para obtener información sintáctica con un mayor nivel de detalle de los nodos de los AST generados por el compilador de Java. Hemos utilizado el API que ofrece el compilador del OpenJDK para extender su implementación mediante *plug-ins* [15]. La API del compilador de Java incluye varias interfaces y clases que pueden extenderse para modificar los AST generados durante la fase de análisis sintáctico.

Para cada fichero Java, el compilador realiza un análisis sintáctico del mismo y genera el AST original. Una vez creado el AST, se recorre para sustituirlo por otro con información más detallada sobre la estructura sintáctica del programa. Recorremos el AST original mediante el patrón de diseño *Visitor* [20].

A modo de ejemplo, todas las expresiones binarias en Java son representadas en el OpenJDK como instancias de BinaryTree, dificultando el conocimiento del tipo concreto de expresión que representa. Para dar más información acerca de la expresión, hemos añadido nodos que representan expresiones aritméticas (Arithmetic), de comparación (Comparison), lógicas (Logical) o a nivel de bits (Bitwise), entre otras (Capítulo 3.2). De este modo, se obtiene una versión con información más detallada de las construcciones sintácticas utilizadas por el programador.

## Nueva estructura del AST

El diseño del AST de OpenJDK está formado por un total de 55 clases [22]. Nosotros añadimos 56 nuevas clases y campos, tal y como se muestra en la Figura 2, para facilitar el minado de patrones sintácticos.

Nuestro diseño crea nuevas generalizaciones de árboles para definiciones (variable, clase, interfaz, método, etc.) y tipos (entero, array, referencia, cadena de caracteres, etc.). También especializa varias clases pertenecientes a las construcciones sintácticas de sentencia y expresión. A modo de ejemplo, OpenJDK modela cualquier expresión utilizada como una sentencia mediante una instancia de su clase ExpressionStatement. Esta clase no distingue construcciones sintácticas como “a=a+1” y “++a”. No obstante, la primera suele ser más habitual entre los programadores noveles, pudiéndose detectar como patrón común a este tipo de programadores. La segunda es más comúnmente utilizada por programadores expertos. Por ello, nuestro diseño del AST incluye los siguientes tipos de nodos para las expresiones que también pueden utilizarse como sentencias: Assignment, MethodInvocation, NewClass, {Post,Pre}fix{De,In}crementUnary y CompoundAssignment.

Lo mismo sucede cuando declaramos un valor o un literal. Nuestro sistema puede, de forma relativamente sencilla, conocer su tipo y añadir mayor especificación que si guardáramos Binary o Literal. Para el fragmento de código “if(a > 0 && b < 3) {…}”, la condición dentro del if será tratada como un LogicalBinary. Por otro lado, a > 0 y b < 3 se definen como ComparisonBinary. Definimos las siguientes clases para clasificar una expresión literal: CharLiteral, DoubleLiteral, FloatLiteral, IntLiteral, LongLiteral, ShortLiteral, Stringliteral o Nullliteral.

Además de las nuevas clases, nuestro diseño también añade nuevos campos para almacenar información sintáctica de interés. Por ejemplo, nuestros nodos del AST incorporan información relativa al rol que éstos juegan en su construcción sintáctica padre. Así, una asignación puede jugar dos roles distintos si su padre es una sentencia while: ser la condición del while o una sentencia de su cuerpo. La información que denota cuándo una asignación se utiliza como condición es significativa, por ejemplo, para detectar programadores no principiantes (éstos rara vez usan asignaciones en las condiciones de while). Estas modificaciones nos permiten ampliar el nivel de detalle en comparación con la versión original, donde solamente conoceríamos la clase que define a un nodo y las que definen a sus hijos.

Además de esto, nuestro sistema introduce una nueva categoría de construcción sintáctica para modelar los tipos. Aunque la inferencia de tipos se realiza en un análisis posterior dentro de la compilación de un programa, a nivel sintáctico es posible distinguir aquellas categorías escritas directamente por el programador en las declaraciones, pudiendo así crear subASTs como *ArrayType*, *Reference*, *GenericType*, *String*, *Char*, *Int* o *UnionType*, entre otros.