Tecnológico de Costa Rica

Escuela de Ingeniería en Computación

Programa de Maestría en Computación

Método para convertir código fuente escrito en diversos lenguajes de programación a un lenguaje universal

Tesis para optar por el grado de *Magíster Scientiae* en Computación, con énfasis en Ciencias de la Computación

Jason Leitón Jiménez
Luis Alonso Barboza Artavia

Profesor Asesor:
Ignacio Trejos Zelaya

Asesor Científico:
Antonio González Torres

Cartago, octubre del 2021

Instituto Tecnológico de Costa Rica
Escuela de Computación
Unidad de Posgrados

ACTA DE APROBACION DE TESIS

MÉTODO PARA CONVERTIR CÓDIGO FUENTE ESCRITO EN DIVERSOS LENGUAJES DE PROGRAMACIÓN A UN LENGUAJE UNIVERSAL

Por: Barboza Artavia Luis Alonso y Leitón Jiménez Jason Josué

TRIBUNAL EXAMINADOR

IGNACIO TREJOS ZELAYA (FIRMA)
Firmado digitalmente por IGNACIO TREJOS ZELAYA (FIRMA)
Fecha: 2021.10.29 17:39:41 -06'00'

MSc. Ignacio Trejos Zelaya
Profesor Asesor

ANTONIO GONZALEZ TORRES (FIRMA)
Firmado digitalmente por ANTONIO GONZALEZ TORRES (FIRMA)
Fecha: 2021.10.29 16:39:06 -06'00'

Dr. Antonio González Torres
Profesor Lector

MARCO ANTONIO HERNANDEZ VASQUEZ (FIRMA)
Firmado digitalmente por MARCO ANTONIO HERNANDEZ VASQUEZ (FIRMA)
Fecha: 2021.10.29 16:30:39 -06'00'

Ing. Marco A. Hernández Vásquez
Lector Externo

LILIANA SANCHO CHAVARRIA (FIRMA)
Firmado digitalmente por LILIANA SANCHO CHAVARRIA (FIRMA)
Fecha: 2021.11.01 13:53:33 -06'00'

Dra. Lilliana Sancho Chavarría
Coordinadora
Unidad de Posgrados, Escuela de Computación

SINAES POST-GRADO ACREDITADO COSTA RICA

29 de octubre, 2021

Agradecimientos

Damos infinitas gracias a Dios por permitirnos concluir un proceso más en nuestra formación académica, guiándonos para tomar las mejores decisiones, tanto en asuntos académicos como personales.

Agradecemos a todos los miembros de nuestras familias por habernos brindado su apoyo y respaldo durante el proceso de maestría en el Tecnológico de Costa Rica. A compañeros y amigos que nos han extendido la mano cuando lo hemos necesitado.

Externamos un profundo agradecimiento a los profesores Antonio González Torres e Ignacio Trejos Zelaya por su gran guía y apoyo en la realización de este trabajo, así como sus consejos para el área profesional de los autores.

Jason Leitón Jiménez, Luis Alonso Barboza Artavia

Cartago, 4 de noviembre de 2021

Resumen

Durante el desarrollo y mantenimiento de software se requiere analizar el código fuente de los programas para determinar factores de calidad. Sin embargo, la gran cantidad de lenguajes de programación dificultan su procesamiento, debido a que difieren gramaticalmente entre ellos (árboles de sintaxis diferentes). Por ello se propone diseñar un método que sea capaz de traducir código Java, C# y RPG a un lenguaje universal, el cual se verifica de manera automática con el fin de asegurar que todos los elementos sintácticos fueron traducidos.

Se presenta la arquitectura básica para entender el diseño del marco de trabajo desarrollado en esta investigación. Su funcionamiento se basa en la obtención de datos provenientes de los archivos de código, el mapeo de dichos datos a la estructura genérica y, por último, la presentación en formato JSON del árbol de sintaxis abstracta genérico. El árbol de sintaxis abstracta genérico se rige por cierta estructura general, por lo que se detallan los diagramas BNF de las principales sentencias, por ejemplo, clases, métodos, paquetes e importaciones.

Se muestra la ejecución del método con proyectos en los lenguajes de programación Java, C# y RPG. De esta manera, se valida el funcionamiento tanto del método que traduce un lenguaje específico a uno genérico, así como el proceso de verificar que todos los elementos sintácticos del lenguaje específico hayan sido mapeados a esta estructura.

Palabras clave: MOF, AST, GAST, SAST, validador, BNF, metadato

Abstract

During software development and maintenance, it is necessary to analyze programs' source code to determine quality factors. However, there are programming languages that make the process difficult, since they have different types of grammar (diversity of syntax). Therefore, the goal of this research is to design a method capable of translating source codes written in programming languages such as Java, C# and RPG into an universal language. This method must be verified automatically to ensure that all syntactic elements were translated.

The basic architecture is presented to understand the framework designed in this research. Its functionality is based on data obtained from the source code files, the mapping of these data to a generic structure, and the presentation of the generic structure in JSON format. The generic abstract syntax language is ruled by a general structure, therefore, BNF diagrams for the main statements are presented in detail; for example: classes, methods, packages and imports.

The results of applying the method are presented with projects written in programming languages such as Java, C# and RPG. The method's operation is validated: it is able to translate a specific language into a generic one, and a tool helps verify that all syntactic elements from a specific programming language have been mapped to the generic structure.

Keywords: MOF, AST, GAST, SAST, validator, BNF, metadata.

Índice general

Índice de figuras

Capítulo 1

Introducción

Los programadores con frecuencia necesitan desarrollar o modificar sistemas escritos en distintos lenguajes de programación, debido a que es común que estos hayan sido implementados utilizando una combinación de lenguajes. Sin embargo, los desarrolladores se deben apoyar en diversas herramientas, porque la mayoría de estas fueron diseñadas para apoyar un lenguaje particular [54].

En este contexto, el proceso de medición y aseguramiento de la calidad de software sufre de limitaciones, tales como la imposibilidad de analizar múltiples lenguajes con una misma herramienta, pues, por lo general, tales herramientas son capaces de obtener la información solo para un lenguaje en particular [4]. Por esta razón, se requiere crear analizadores separados, para cada lenguaje, utilizando herramientas como SonarQube y Moose.

Para analizar el software escrito en diferentes lenguajes de programación, se requiere abordar de manera independiente cada uno de ellos, debido a las características específicas de sus estructuras gramaticales. Además, implementar un método de análisis particular agrega complejidad, debido a que se debe realizar un análisis específico para cada uno de los lenguajes.

La complejidad mencionada radica en las actividades relacionadas con el diseño, implementación, reutilización y mantenimiento de las aplicaciones encargadas de analizar el código fuente. Por este motivo, se considera que la transformación del código fuente de cada lenguaje de programación específico en un lenguaje universal (sintaxis universal equivalente al código original) puede facilitar el diseño de diversos métodos para analizar código fuente en distintos lenguajes de programación. Esto implica que su implementación considere la sintaxis de un lenguaje genérico en lugar de la sintaxis de cada lenguaje específico (por ejemplo, RPG, C#, Java). Este proceso implica modelar el código fuente de manera independiente al lenguaje y paradigma de programación que utilice.

Este trabajo de investigación propone una metodología para transformar de manera automática la sintaxis de lenguajes particulares hacia una sintaxis genérica equivalente. Esta metodología se valida mediante la transformación de código escrito en Java, C#, y RPG de diferentes sistemas en su sintaxis genérica equivalente.

Dicha transformación a una sintaxis genérica tiene como objetivo adaptarse a la amplia gama de lenguajes que existen en el mercado informático. Lo anterior, debido a que los programas pueden estar escritos en diferentes lenguajes de programación de acuerdo con las restricciones, las necesidades y el estilo del desarrollador. Por esto, una estrategia de solución es crear un único lenguaje para analizar un sólo tipo sintaxis, la cual sería genérica para cualquier programa, independientemente de las características y propiedades con las que fue escrito.

1.1 Antecedentes

El análisis y la refactorización de código fuente son actividades que se relacionan con los diferentes estándares que buscan representar los datos más significativos de los programas, con el fin de comprender las relaciones y datos de fondo que posee esta información. Además, los modelos utilizados proporcionan una base consolidada para examinar de manera profunda cada línea de código, lo cual permite que la implementación de un lenguaje universal no inicie desde cero.

Dos de los estándares para definir meta-modelos, cuya especificación es pública y sin ánimos de lucro, son MOF y FAMIX [21] [51], los cuales se detallan en la siguiente subsección.

Una aplicación cuyo fin se relaciona con este proyecto es SonarQube. Sin embargo, la manera de implementación de sus funcionalidades difiere en varios aspectos con las que se propone en este documento.

1.1.1 MOF (Meta Object Facility)

Este estándar fue creado por el OMG. La idea fundamental de MOF es permitir la creación y el mantenimiento de especificaciones de modelos, con el fin de obtener interoperabilidad y portabilidad entre las diferentes plataformas de hardware/software, independientemente de la arquitectura de éstas [27].

MOF también proporciona una base sólida para la definición de un meta modelo y se basa en la simplificación de las capacidades de modelado de UML2. MOF brinda todos los medios necesarios para especificar el modelo y agrega capacidades básicas para la gestión de modelos en general, incluidos los identificadores, etiquetas genéricas, operaciones reflexivas, las cuales se pueden aplicar independientemente del meta modelo [28].

MOF proporciona un marco de trabajo abierto e independiente de cualquier plataforma, lo que facilita la interoperabilidad de los sistemas impulsados por modelos y meta datos. Por ejemplo las aplicaciones que utilizan MOF, generalmente incluyen herramientas de modelado, desarrollo, almacenamiento y recuperación de datos, algunos de ellos cuentan con repositorios de información [28].

Por su parte, MOF ha contribuido significativamente a los principios básicos de la MDA

(Model Driven Architecture). Basándose en el modelado establecido por UML, MOF introdujo el concepto de meta modelos formales así como los modelos independientes de plataforma (PIM).

El trabajo que se presenta en este documento se basa en el núcleo MOF, el cual se detalla en la siguiente sección. La especificación completa del estándar MOF se puede encontrar en [52].

1.1.2 FAMIX

Es un estándar que busca aplicar la reingeniería de los datos con el fin de crear algún mecanismo para el intercambio de información entre distintos productos de software, los cuales pueden ser ejecutados en diferentes plataformas. FAMIX se crea como una alternativa a los modelos MOF y UML, ya que estos no son capaces de estandarizar la interoperabilidad de programas que utilizan herramientas de ingeniería inversa [34].

FAMIX se centra en el problema de la ingeniería de ida y vuelta, la cual consiste en sincronizar dos o más elementos de software diferentes, pero relacionados entre sí. Los tres pilares de esta área son [15]:

- Recuperación arquitectónica, donde el código se produce en diagramas de trabajo a través ingeniería inversa. Comúnmente se denomina como modelo específico de plataforma (PSM), el cual se ajusta a un determinado modelo, por ejemplo, el AST de Java que es una estructura para modelar la sintaxis de únicamente este lenguaje.
- La extracción del modelo independiente de plataforma (PIM), el cual se centra en aumentar el nivel de abstracción de los modelos específicos de plataformas, por ejemplo un AST genérico que modele varias sintaxis de diferentes lenguajes.
- Refactorización de código, generalmente incluye restricciones especificadas en el lenguaje OCL (Object Constraint Language).

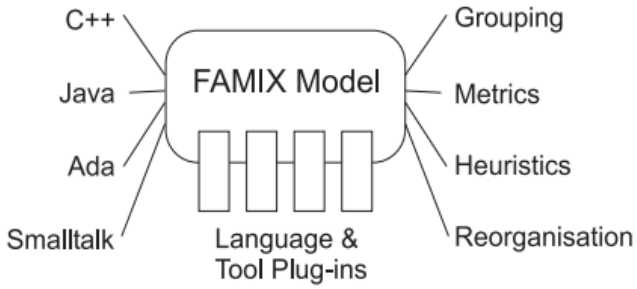


Figura 1.1: Conceptualización del estándar Famix [16].

La idea fundamental de este estándar es tomar los datos de programas escritos en diferentes lenguajes para obtener información específica del código, en la figura 1.1 se muestra el esquema básico de FAMIX.

La especificación completa de este estándar la puede encontrar en [16].

1.1.3 SonarQube

SonarQube es un *framework* que proporciona los mecanismos para crear y verificar nuevas gramáticas y parsers. Esta herramienta produce un árbol de sintaxis abstracta (AST) y permite la definición de reglas para la detección de fallas y el cálculo de métricas de calidad, a partir del recorrido del AST [10].

SonarQube, es una herramienta para la inspección continua, un proceso que hace que el análisis de la calidad del código y la presentación de informes sean una parte integral del ciclo de vida del desarrollo de software. Se enfoca en el análisis de defectos basados en reglas, además la estrecha integración de la compilación dan como resultado una calidad de código mejorada sin interrumpir el flujo de trabajo del desarrollador [10].

Cabe resaltar que la versión completa de SonarQube tiene un costo elevado. Además, las reglas sintácticas para una nueva gramática deben ser escritas por el desarrollador, esto disminuye la flexibilidad de la aplicación porque se requiere cierto grado de experiencia para completar este tipo de tareas.

1.2 Planteamiento del problema

La calidad del software se relaciona con el producto y el proceso de desarrollo [32], y es un factor que diferencia a las organizaciones [56]. La norma ISO/IEC 9126 Ingeniería de software - Calidad del producto, de la Organización Internacional de Normalización (ISO), especifica un modelo de calidad que contempla la funcionalidad, mantenibilidad, rendimiento, escalabilidad, usabilidad y portabilidad de los sistemas [32, 23]. Esta norma fue publicada en 1991 y actualizada en el 2001. La norma ha sido actualizada y extendida por la la familia ISO/IEC 25000, la cual tiene como objetivo principal la creación de un *framework* para evaluar la calidad del producto de software, para lograr mayor confiabilidad en este [57].

Las herramientas de análisis de código fuente facilitan estudiar algunos factores de calidad, diseño y mantenibilidad con base en los resultados de métricas que guían al desarrollador en todo el ciclo del software. Sin embargo, la obtención de métricas es compleja cuando se utilizan diferentes lenguajes de programación, por las peculiaridades de cada uno, que requieren ser abordadas de manera particular. Por esto, la transformación sintáctica de cada lenguaje en una sintaxis genérica equivalente a la original es una alternativa flexible para analizar los programas escritos en diferentes lenguajes de programación. El problema fundamental de esta investigación radica en efectuar el planteamiento de un método eficaz para convertir un lenguaje de programación específico en otro universal con una sintaxis genérica.

Debido a la variedad de opciones en el análisis de un programa, no existe un único modelo que relacione todos los factores que intervienen en el desarrollo del software. Por ello, es necesario proporcionar cada detalle del código. Por estas razones, la estructura del lenguaje genérico debe contemplar todos los aspectos sintácticos para que el programa pueda ser representado de manera total y con todos los detalles. Esto tiene como fin que la estructura genérica brinde la misma información que el código original.

Esta estructura genérica debe tomar en consideración los cambios que ocurren en los lenguajes de programación, con el fin de facilitar el mantenimiento y la escalabilidad en caso de que se requiera agregar nuevos lenguajes de programación o modificar los existentes. Las posibles soluciones son las siguientes:

Crear tantos analizadores como lenguajes existan: Este enfoque presenta una limitación cuando se quiera agregar nuevos lenguajes de programación, ya que las métricas y demás técnicas de análisis se deben adaptar para este nuevo lenguaje. El problema se hace más grande en caso de desear agregar una nueva métrica porque se debe añadir a cada uno de los analizadores.

Crear un analizador para un lenguaje genérico: se debe realizar la transformación de un lenguaje de programación en específico a una estructura genérica ya establecida. Con este enfoque, el trabajo de agregar o modificar métricas en el análisis se facilitará, puesto que están hechas para una estructura genérica y no para cada lenguaje.

Actualmente se cuenta con una amplia diversidad de lenguajes de programación, los cuales poseen diferentes sintaxis y pertenecen a distintos paradigmas. Esto plantea dificultades para analizar los programas, ya que es necesario crear un analizador específico para cada gramática de programación. Por lo anterior, se considera necesaria la existencia de un lenguaje universal con el fin de tener un único analizador de software en lugar de n analizadores para n lenguajes de programación. Esto da origen al siguiente problema.

Pregunta de investigación: ¿Cómo definir un método para efectuar la transformación automática de diversos lenguajes de programación en un AST generalizado con una sintaxis universal?

1.3 Justificación del problema

Generalmente, las organizaciones no realizan sus programas en el mismo lenguaje de programación [42], por lo que los desarrolladores de códigos deben ser lo suficientemente flexibles para comprender y actualizar cualquiera de estos. Por esta razón es que en este trabajo se propone crear un lenguaje universal con el fin de mapear los elementos de cada lenguaje en específico (e.g. Java, C# y RPG) a uno con gramática universal. Esto proporcionará a los programadores facilidades en sus labores, ya que el análisis se enfoca en una única gramática.

Cabe resaltar que los desarrolladores son los encargados de realizar el mantenimiento y la evolución del software. Dichas labores se caracterizan por su alto costo y lentitud de implementación. Todo software útil y exitoso atiende las solicitudes generadas por los usuarios para cambios (evolución) y mejoras (mantenimiento) en el tiempo [8]. Debido a este tipo de tareas es que se debe analizar el impacto de los cambios sobre el software existente y determinar cuál es la repercusión en su calidad. Este análisis se facilita si se tiene una sola sintaxis porque los métodos utilizados se pueden homogeneizar para cualquier lenguaje de programación. Esto implica un beneficio para la compañía con respecto de una reducción de costo y tiempo en tareas de actualización y mantenimiento de software, que pueden alcanzar montos iguales o superiores a los del desarrollo propiamente [33], lo que significa un mejor aprovechamiento de sus recursos.

Los lenguajes de programación tienen diferentes comportamientos de acuerdo con el paradigma al cual pertenezcan y por esta razón es que el análisis de los programas se complica al no tener uniformidad en los datos; por ejemplo, C++ posee herencia múltiple, pero Java no [1]. Estas diferencias hacen que en la toma de decisiones no se pueda definir parámetros homogéneos para todos los lenguajes por igual. Por otro lado, al existir una única manera de representar los datos, se podría abarcar un análisis mejor planteado, ya que se proporciona los datos con una misma estructura, lo que implica realizar comparaciones directas entre dos o más códigos.

1.4 Hipótesis

Hipótesis 1: La creación de un árbol de sintaxis abstracto genérico permite representar todos los elementos sintácticos de diversos lenguajes de programación, tales como RPG, C# y Java.

Hipótesis 2: La creación de un lenguaje universal permite homogeneizar el análisis de los elementos y las estructuras gramaticales de los programas escritos en distintos lenguajes de programación, tales como RPG, C# y Java.

1.5 Objetivos

Esta sección presenta el objetivo general y los objetivos específicos de este trabajo de investigación.

1.5.1 Objetivo general

Diseñar una metodología para transformar de forma automática la sintaxis de lenguajes particulares en una sintaxis genérica equivalente, utilizando los árboles de sintaxis abstracta (AST) de cada uno de estos.

1.5.2 Objetivos específicos

- Definir una estructura de datos para representar la sintaxis de cada lenguaje de programación en una sintaxis universal, modelando cada elemento gramatical.
- Validar el método de transformación de la sintaxis de diferentes lenguajes de programación en una sintaxis universal mediante una prueba de concepto, con el fin de verificar el modelado de la gramática.
- Validar de manera automática la equivalencia entre el AST del lenguaje universal y el AST de un lenguaje específico, con el fin de asegurar la existencia de todos los elementos gramaticales del lenguaje específico en el AST genérico.
- Crear la especificación BNF del lenguaje universal, utilizando gramáticas independientes del contexto, con el fin de formalizar la sintaxis de este.

1.6 Alcances, entregables y limitaciones del proyecto

1.6.1 Alcances y limitaciones

Los experimentos planteados se delimitan únicamente a los lenguajes C#, Java y RPG. Los programas de prueba se tomaron directamente desde los repositorios de GitHub, los cuales son públicos y compilados sin ningún tipo de errores.

El código universal se almacenará de manera local y utilizando un servidor de MongoDB, para reducir costos económicos de *hosting*.

1.6.2 Productos de investigación

Los productos de la investigación reportada en esta tesis fueron los siguientes:

- Código fuente del marco de trabajo para transformar un programa en Java, RPG, C# al lenguaje universal, el cual debe ser documentado con Javadoc para facilitar su mantenimiento y escalabilidad.
- Documentación técnica para que los futuros desarrolladores puedan comprender el trabajo realizado y tengan la posibilidad de agregar los módulos correspondientes a la traducción de otros lenguajes (Python, C++, VB, entre otros). Esto incluye los diagramas de clases, componentes e instalación.
- Diagramas BNF de las principales estructuras del lenguaje universal (clases, métodos, invocaciones de métodos, estructuras de control).
- Código fuente del verificador automatizado de estructuras, con su respectiva documentación técnica.

- Interfaz de la base de datos con MongoDB, así como la arquitectura para realizar consultas sobre datos sintácticos del lenguaje universal.
- Un artículo científico en inglés, dos meses después de la defensa de este trabajo, con el fin de publicarlo en alguna conferencia pertinente.

Una de las restricciones del proyecto de investigación fue que solo se debía utilizar herramientas de código abierto que no impliquen ninguna erogación por parte del proyecto de investigación AVIB.

1.7 Impacto, profundidad y originalidad del proyecto

En esta sección se brinda el contexto del impacto que tiene desarrollar un proyecto de esta naturaleza y con el alcance propuesto. También se menciona lo específico y profundo que suele ser el tema de análisis de código, así como la originalidad en el ámbito académico e industrial.

La sociedad cada vez más involucra la tecnología en cada uno de sus aspectos cotidianos, fenómeno que ha provocado que las empresas dedicadas al desarrollo de hardware y software aumenten sus estándares de calidad, lo que ha implicado que los desarrolladores de software requieran herramientas para el análisis de los programas que realizan. Este proyecto busca que los desarrolladores de software posean un método para el análisis de código fuente escritos en más de un lenguaje de programación, lo que beneficia a la empresa, ya que brinda la versatilidad y proporciona una interfaz para agregar más lenguajes – según se haga necesario. El impacto directo se reflejaría en factores económicos y de recurso humano, ya que el proceso de análisis de código fuente se realizaría de manera automatizada e independientemente del lenguaje de programación con el que trabaje la empresa. Además, el recurso humano aumentaría la productividad, ya que no se invertiría tanto tiempo en analizar millones de líneas de código de manera manual.

Por otra parte, el aporte a los científicos que se dedican al análisis de código sería en la reducción de tiempo para ejecutar experimentos, ya que podrían plantear un mismo experimento para ser realizado sobre programas escritos en múltiples lenguajes de programación.

La originalidad de este proyecto radica en la simplicidad para añadir lenguajes de programación a un *framework*, con el fin de que los programas escritos en dicho lenguaje puedan ser analizados utilizando los mismos métodos o métricas ya existentes. Asimismo, en caso de requerir una nueva métrica o un nueva variedad de análisis, no se debe crear una para cada lenguaje de programación, ya que el *framework* trabajará con un único lenguaje y es el universal, por lo que las métricas se trabajarán para este lenguaje. Además, los resultados se verifican con un validador automatizado, lo cual da un grado de confianza que otras herramientas no ofrecen.

Capítulo 2

Marco teórico

Este capítulo tiene como objetivo orientar al lector en los conceptos necesarios para comprender el diseño y el análisis tanto de la implementación del software (*framework*) como el diseño de los experimentos planteados y sus resultados. Se profundizará en aspectos que son fundamentales para desarrollar el proyecto como, por ejemplo, teoría de lenguajes (AST y BNF), análisis de código fuente, extracción de meta-datos (datos de los distintos programas). Además, se analizarán las herramientas desarrolladas por terceros y que son necesarias para ejecutar el *framework*.

Algunos autores, como Nadkarni [43] y Hiom [30] mencionan que la definición básica de meta-datos es los "datos acerca de los datos" (*data about data*). Los meta-datos son los adjetivos que describen los datos primarios de un sistema determinado, el acoplamiento entre éstos, en la mayoría de los casos, son vitales para el correcto funcionamiento del software, sin embargo no siempre es así; por ejemplo, una página web puede incluir meta-datos opcionales que especifiquen el lenguaje de programación en el que está escrito, las herramientas que se utilizaron y dónde se puede obtener más información [43]. La importancia de los meta-datos en el proyecto radica en que se debe extraer la información necesaria del código fuente de los programas. Los datos de base provienen del código fuente y corresponden a los elementos sintácticos y sus relaciones, mientras que la información corresponde a todo el proyecto de programación.

Se refiere como meta-datos a las relaciones de fondo (por ejemplo, qué elemento está dentro del alcance de otro) y datos primitivos (por ejemplo los identificadores) que existen entre los diferentes elementos de un código fuente. La idea fundamental es analizar y abstraer la información más relevante de cada línea de código, es decir, tomar datos acerca de los datos (código fuente).

El análisis de los datos se ha vuelto fundamental para refactorizar información y crear aplicaciones portátiles. Un ejemplo de meta-dato es la cantidad de bytes que necesita una variable (meta-dato) acorde con su tipo y representación, en el caso de INT se sabe que utiliza 4 bytes (en el lenguaje Java) [43].

Los meta-datos son críticos para todos los aspectos de la interoperabilidad dentro de

cualquier ambiente heterogéneo (ejecutado por distintas plataformas), ya que son los que brindan las relaciones entre los elementos del código, las cuales son reutilizadas por diferentes plataformas (interoperabilidad) para ejecutar funciones equivalentes. De hecho, esta información es el principal medio por el cual la interoperabilidad es lograda.

2.1 Notación BNF

La notación BNF es un ejemplo conocido de una meta sintaxis, es decir, una que describe sintácticamente un lenguaje de programación. Usando BNF es posible especificar qué secuencias de símbolos constituyen un programa sintácticamente válido en un lenguaje dado. La semántica, es decir, lo que significan estas cadenas de símbolos válidas debe especificarse por separado [39].

Una unidad sintáctica es aquella cuyos valores son cadenas de símbolos elegidos entre los símbolos permitidos en el lenguaje dado. En BNF, las variables meta lingüísticas están entre corchetes, < >, para mayor claridad y para distinguirlas de los símbolos en el lenguaje propiamente, que se llaman símbolos terminales o solo terminales . El símbolo ::= se utiliza para indicar la equivalencia meta lingüística; una barra vertical (|) se usa para indicar que se debe hacer una elección entre los ítems así indicados. La concatenación (poner símbolos en una secuencia) se indica simplemente al yuxtaponer los elementos por concatenar. El siguiente ejemplo indica que un dígito puede tomar los valores del $0-9$:

- <digito>::=1|2|3|4|5|6|7|8|9|0

La barra vertical toma un significado de elección que es equivalente al comportamiento de la compuerta OR (en circuitos digitales) o instrucciones como **if** o **switch** (en lenguajes de programación).

Las reglas generales y más ejemplos del estándar BNF se pueden encontrar en el trabajo de Adrian Farrel [24].

2.2 AST

La manera en que se obtienen los datos de un código fuente es por medio de su Árbol de Sintaxis Abstracta (AST), el cual representa todos los elementos sintácticos de un programa escrito en un lenguaje de programación. Esta estructura de datos se centra en las reglas gramaticales esenciales en el que está escrito un determinado software en lugar de los elementos como llaves o puntos y comas que delimitan las instrucciones en algunos lenguajes [7].

El AST es una representación formal de la estructura sintáctica del software, a partir de las técnicas de análisis aplicadas sobre la sintaxis concreta (toma todos los tokens) [13].

La construcción del árbol generalmente implica el uso de tecnologías de análisis sintáctico, pero los AST también se pueden construir por medio de una generación o proceso de derivación de alguna otra especificación, como lo pueden ser esquemas de XSD [28]. Las estructuras del AST permiten expresar las relaciones de composición con otros elementos del lenguaje y proporcionan un medio para expresar un conjunto de propiedades derivadas y asociadas con cada componente del lenguaje de programación [28].

El AST es un árbol jerárquico, con los elementos de los enunciados de programación divididos en partes llamadas nodos [7]. Por ejemplo, un árbol para una comando condicional tiene las reglas para las variables que cuelgan del operador requerido. En la figura 2.1 se muestra un ejemplo sencillo de un AST de una sentencia **if**. Nótese que los círculos con los operadores (==,>) son los nodos de la condición mientras que los operandos son las hojas de dichos nodos.

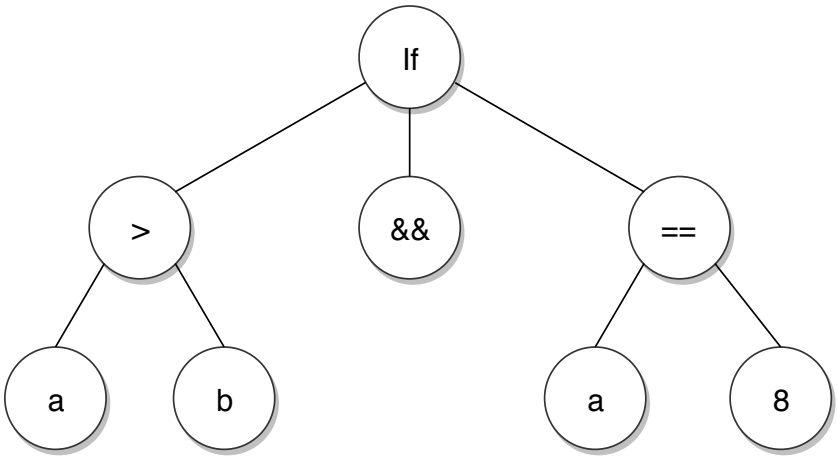


Figura 2.1: Ejemplo de un AST para una sentencia "if".

Formalmente, un árbol de sintaxis abstracta (AST) es un árbol finito, etiquetado y dirigido, donde los nodos internos están etiquetados por operadores, y los nodos hoja representan los operandos. Se debe utilizar un analizador sintáctico como intermedio entre un árbol de análisis sintáctico (estructura con todos los elementos sintácticos) y un AST. Este último típicamente se usa como insumo del compilador o como representación interna de un programa mientras se optimiza (instrucciones fuera de orden) y a partir de la cual se realiza la generación de código binario. El rango de todas las estructuras posibles es descrito por la sintaxis abstracta. Un AST difiere de un árbol de análisis (que contiene cada detalle del código) pues omite nodos y bordes (hojas) para las reglas de sintaxis que no afectan la semántica del programa, por ejemplo, elementos que terminan expresiones, en el caso de Java es el punto y coma [52].

Algunas de las propiedades más útiles de los AST son: [52]

1. Puede ser invertible, debido a que permite la reconstrucción de la sintaxis a partir de la estructura de datos.

2. Puede ser escalable con el análisis de flujo de datos, análisis axiomático (razonamiento

formal respecto de los programas) y análisis de denotación (enfoque semántico para dar significado a los programas).

3. Se puede utilizar como entrada de datos en el cálculo de métricas de ingeniería de software.
4. Puede ser mapeado o trasformado en otros modelos.
5. Puede ser utilizado para realizar consultas, utilizando algún mecanismo como QVT (Query View Transformation).

Este trabajo se enfocará en las propiedades 3, 4 y 5, ya que lo que se requiere es obtener métricas de código a partir de una estructura sintáctica neutral.

En el desarrollo de esta investigación se interactúa con diferentes AST tomados directamente del código fuente de los distintos programas por analizar con el fin de mapearlos a un árbol sintáctico genérico. Por esta razón, es necesario obtener las diferencias entre dos estructuras de datos, con el fin de evaluar si toda la información de cada elemento sintáctico está presente en el AST generalizado. Los árboles de sintaxis abstracta, al ser una representación del código, pueden ser utilizados para mostrar información de cambios en dos estructuras de una manera detallada [25]. De esta forma, uno de los usos que comúnmente se da es en el control de plagio o versiones. Esto aprovecha que el árbol presenta los nodos ordenados y etiquetados, gracias a lo cual se realiza un análisis para buscar las hojas con valores y con ello obtener los datos [31]. Una característica que ofrecen estos árboles son elementos sintácticos asociados al código fuente, lo cual facilita el filtrado de información que concuerde en la comparación [35]. Puede ocurrir un caso donde un nodo no se encuentre por completo en el segundo árbol, creando una diferencia de tipo ausencia. Por otro lado, también puede suceder donde una de las hojas presenta un cambio en algún valor, o que puede ser considerado como una desigualdad entre los dos elementos sintácticos [63].

2.3 Lenguaje universal

Un lenguaje universal consiste en una sintaxis que es capaz de representar otras sintaxis más simples, esto implica que debe poseer una gramática generaliza, con el fin de representar la mayor cantidad posible de lenguajes de programación. Para este trabajo en específico, se requiere que las sintaxis de Java, C# y RPG puedan ser representadas de manera equivalente en el lenguaje universal.

Ante la existencia de tantos lenguajes de programación, realizar un análisis para cada uno conlleva un extenso trabajo, pues se deben tomar todos los posibles casos que presenta el paradigma específico para conformar las reglas. Una opción que puede aprovechar las características de los árboles sintácticos es el metalenguaje. Un metalenguaje, según el diccionario de Cambridge, es una forma especializada de un lenguaje o un conjunto de símbolos utilizados para describir una estructura de (otro) lenguaje [17].

Este concepto se puede aplicar para describir un lenguaje de programación. Para lograrlo, los componentes principales que poseen la información necesaria se abstraen a una estructura general con el fin de encapsularlos en un lenguaje genérico. El resultado final de dicha estructura servirá como intermediario para futuras acciones que se deseen realizar sobre el código fuente como, por ejemplo, obtención de métricas y análisis de clones. Una ventaja de utilizar este enfoque es que permite la reutilización y la creación de una solución general para soportar múltiples lenguajes de programación [61].

El lenguaje universal que se propone también posee un AST, que para efectos de este documentos se le nombrará como Meta Árbol de Sintaxis Abstracta, conocido en inglés como Abstract Syntax Tree Meta (ASTM), el cual tiene la capacidad de generar un árbol sintáctico para mostrar todos los componentes en una forma jerárquica. En este caso, es una estructura estándar desarrollada por OMG como un complemento para el modelado de árboles abstractos sintácticos [50]. Una característica que posee es el mapeo que, según la guía para el ASTM, es la representación fidedigna de un código escrito en cualquier lenguaje de programación. También se pueden adjuntar semánticas de bajo nivel del *software* producidas por una delimitación en el análisis.

Una estructura de este tipo está compuesta por un elemento genérico y uno específico, así como de su relación. En el caso del primero, llamado Árbol Genérico de Sintaxis Abstracta del Meta Modelo (GASTM), es una representación del código fuente sin estar ligado a un lenguaje de programación en específico, debido a que contiene los elementos comunes en forma de metatipos. Este elemento sirve como base para definir el dominio con las características propias que tenga el lenguaje de programación. El Árbol Específico de Sintaxis Abstracta del Meta Modelo (SASTM) corresponde a la construcción haciendo uso de los casos especiales que presente el lenguaje [14]. En caso de no representar un árbol y que sea consistente con el estándar MOF ni los modelos anteriores, se utiliza el Árbol Propietario de Sintaxis Abstracta del Meta Modelo (PASTM) que tiene un carácter de posesión, debido a que son modelos específicos de un vendedor. Por tal razón, si una persona desea brindar un PASTM, se debe transformar a una mezcla de GASTM y SASTM [47].

El ASTM puede representar múltiples lenguajes de programación. Algunos ejemplos son Ada, ensamblador, C, C#, COBOL, FORTRAN, Java, SQL. Alcanzar dicha representación se logra con las técnicas para definir el GASTM y SASTM con respecto del lenguaje de programación. De esta manera, permite extenderse a más lenguajes desarrollando un modelo de dominio específico para cada uno [47].

2.4 Control de calidad

La validación y la verificación son dos conceptos asociados al control de calidad del software. La validación es el proceso para determinar el grado de exactitud de la representación de un modelo en instancias de trabajo en la manera que se va a usar. Mientras que la

verificación es el método para precisar que la implementación de un modelo representa correctamente la descripción conceptual de un desarrollador, así como la solución al modelo [46]. De esta manera, un programa se va a someter a múltiples actividades y pruebas con el fin de establecer si cumple con los requerimientos. Lo que se busca es que un producto de *software* sea capaz de ejecutarse correctamente según la lista de funciones y además, que se pueda utilizar de manera precisa en un periodo largo de tiempo [60].

En proyectos con muchos componentes, el control de calidad debe estar presente en cada una de las fases. De esta manera, el resultado de los elementos debe ser correcto según los requerimientos y, a su vez, servirá como entrada para que el siguiente procedimiento realice los diferentes cálculos con una entrada correcta, conforme a la especificación del componente. En estos casos, el bloque debe probarse y validarse con el fin de detectar imprecisiones [26]. Para este proyecto, se requiere una validación porque la estructura generada por un módulo es el insumo para otro componente, por lo que es necesario garantizar que el procedimiento se realizó de acuerdo con los requerimientos planteados.

En los sistemas con múltiples componentes, cada elemento presenta características distintas, por lo que una prueba general no abarca la totalidad de las propiedades. De tal manera, se deben realizar ajustes para crear pruebas específicas y cubrir la totalidad del proyecto, mediante pruebas de integración y del sistema [26].

En la actualidad, la calidad en los productos y servicios tiene un rol esencial, por esto el Instituto de Ingenieros Eléctricos y Electrónicos (IEEE) creó un estándar para los sistemas llamado IEEE-1012-2016 donde busca abarcar los temas de verificación y validación. Este estándar busca determinar que los procesos de validación y verificación establezcan que los productos desarrollados de una actividad cumplen con los requerimientos y si el resultado del proyecto satisface el uso esperado. Para lograr esta función, se pueden realizar análisis, evaluaciones, revisiones, inspecciones y pruebas de productos y procesos [22].

Para la verificación, el estándar establece que el proceso genere una evidencia objetiva de que los productos [22] :

- Cumplen los requerimientos en aspectos de correctitud, completitud, consistencia y exactitud para todas las actividades dentro del ciclo de vida.
- Satisfacen los estándares, prácticas y convenciones durante el ciclo de vida.
- Completan de manera exitosa el ciclo de vida de cada actividad.

En el caso de la validación, la aplicación del estándar permite dar evidencia de que el producto [22]:

- Satisface los requerimientos del sistema asociados al producto al final del ciclo de vida de cada actividad.
- Resuelve de manera correcta el problema.

- Satisface las necesidades del usuario en el ambiente operacional.

Estos procesos se deben ejecutar a través del desarrollo del producto en cada una de los fases de trabajo. Con esto, además de encontrar posibles errores, ayuda a determinar el cumplimiento de los requerimientos indicando si están correctos, completos, precisos, consistentes y si se pueden probar. Algunos beneficios de realizar esta práctica según el estándar de la IEEE son [22]:

- Facilitar la detección y corrección temprana de anomalías.
- Mejorar la visión en el manejo de riesgos de procesos y productos.
- Soportar los procesos en el ciclo de vida para asegurar el cumplimiento de los requerimientos del sistema de software, así como del cronograma de trabajo y del presupuesto.
- Proporcionar una revisión temprana de rendimiento.
- Brindar evidencia objetiva de conformidad para dar soporte a procesos formales de certificación.
- Mejorar los productos de los procesos de adquisición, suministro, desarrollo y mantenimiento.
- Soportar actividades de mejoramiento de procesos.

2.5 Herramientas de desarrollo utilizadas

En esta sección se describen las herramientas computacionales que se utilizan en la implementación del marco de trabajo, las cuales facilitan los distintos procesos que llevan a cabo las funciones necesarias. Cabe destacar que todas las herramientas son de uso libre.

2.5.1 Maven

Maven es una herramienta de gestión y portabilidad de proyectos de software. Con base en el concepto de un modelo de objeto de proyecto (POM), Maven puede gestionar la compilación, los informes, las dependencias y la documentación de un proyecto a partir de una pieza central de información (código) [38].

Maven utiliza un arquetipo, el cual es un patrón o modelo con el que se pueden desarrollar tareas de un mismo tipo. Se podría hacer una analogía con una plantilla parametrizada con la cual los desarrolladores podrían basarse para organizar el código de una aplicación. La principal ventaja de esto es la estandarización en la estructura de los proyectos.

Maven busca que los desarrolladores de Java posean un ambiente de desarrollo para cualquier proyecto que se desarrolle en dicho lenguaje.

Los principales objetivos de Maven son [38]:

- Realizar el proceso de construcción y despliegue de aplicaciones de una manera sencilla, ya que el proceso de compilación es transparente para el desarrollador.

- Proporcionar información de alta calidad sobre el software, como lo es la versión, autores, bibliotecas, dependencias entre otros. Esto ocurre gracias a la capacidad que tiene para generar la documentación del código.

- Proporcionar mejores guías para ejecutar buenas prácticas de programación, ya que provee cierta estructura que guía al desarrollador en la jerarquía de los archivos.

- Permite una migración transparente y provee un mejor ambiente para agregar nuevas características debido a que la estructura de los archivos es independiente del IDE con el que se trabaja.

2.5.2 MapStruct

MapStruct es un generador de código que simplifica la implementación de asignaciones entre datos de diferentes objetos. El código de mapeo generado utiliza invocaciones de métodos simples y, por ello, es rápido, seguro y fácil de entender [37].

Las aplicaciones de varias capas a menudo requieren mapear entre diferentes modelos de objetos (por ejemplo, entidades y DTO) [37]. Escribir el código de mapeo es una tarea tediosa y propensa a errores. MapStruct busca simplificar este trabajo automatizándolo tanto como sea posible.

A diferencia de otros marcos de mapeo, MapStruct genera mapeos de *bean* (clase simple, reutilizable y con cierta estructura definida, por ejemplo, los métodos constructor, set, get y atributos) en tiempo de compilación [12], lo que garantiza un alto rendimiento, permite una rápida retroalimentación del desarrollador y una minuciosa comprobación de errores [37].

MapStruct es un procesador de anotaciones que se conecta al compilador de Java y se puede usar en compilaciones de línea de comandos Maven, Gradle, entre otros. MapStruct usa valores predeterminados razonables, pero toma otra ruta en el flujo de información cuando se trata de configurar o implementar un comportamiento especial [37]. En la figura 2.2 se muestra un ejemplo de las anotaciones de MapStruct como lo son “@Mappings” y “@Mapper”.

```java
@Mapper
public interface CompilationMapper {
    CompilationMapper INSTANCE= Mappers.getMapper(CompilationMapper.class);
    @Mappings({@Mapping(target = "path", ignore = true),
               @Mapping(target = "fragments", ignore = true),
               @Mapping(target = "language", ignore = true),
               @Mapping(source="tipos",target = "opensScope.declOrDefn"),
               @Mapping(source="imports",target="imports"),
               @Mapping(source="compilation.package",target="gPackage")})
```

Figura 2.2: Ejemplo de mapeo con Mapstruct.

2.5.3 Gson

Gson es una biblioteca de Java que convierte los objetos de Java en su representación JSON. También se puede usar para convertir una cadena JSON a un objeto Java equivalente [29]. Los principales objetivos de Gson en la conversión de datos son [19]:

- Proporcionar mecanismos fáciles de usar como toString() y constructor para convertir Java a JSON y viceversa.
- Permitir la conversión de objetos preexistentes no modificables a JSON.
- Permitir representaciones personalizadas para objetos.
- Generar salida JSON compacta y legible.

2.6 Clones de código

Los clones de código se definen como fragmentos de código similares en los archivos fuentes de un software. Si dos o más fragmentos de código son exactamente los mismos sin tener en cuenta los comentarios e identación, estos fragmentos de código se denominan clones exactos o clones de Tipo 1. Los clones de Tipo 2 son códigos sintácticamente similares. En general, se crean a partir de clones Tipo 1 debido al cambio de nombre de los identificadores o al cambio de datos de tipos. Los clones de Tipo 3 se crean principalmente debido a adiciones, eliminaciones o modificaciones de líneas de código en clones de Tipo 1 o Tipo 2, sin embargo, mantienen una estructura y semántica similar [40].

2.7 Trabajos relacionados

La transformación de código fuente entre lenguajes es necesaria en la industria del desarrollo de software, debido a que la mayoría de compañías requieren que sus programas sean ejecutados en múltiples plataformas [6], para que la flexibilidad y compatibilidad del software sea mayor. El trabajo realizado por Bastidas y Pérez [6] muestra que una de

las maneras más comunes de realizar transformaciones de código fuente es por medio de transpiladores, cuyo trabajo consiste en el procesamiento de sintaxis, mapeos lineales y generación de nuevo código. Los transpiladores son de suma importancia porque permiten desarrollar y analizar software destinado a múltiples plataformas, con la ventaja de escribir una única vez el código [53, 6]. Además, brindan mayor flexibilidad a los desarrolladores, al solucionar problemas en múltiples lenguajes.

En el marco de trabajo propuesto por O'Hara [49] se realizan los siguientes pasos para transformar código fuente.

1. Mediante una gramática, se analiza el código fuente lo cual produce un Árbol Semántico de Programa (PST). Esta estructura es similar al AST, pero incluye alguna información de los elementos tal como una referencia entre las declaraciones de variables (ocurrencias de definición) y sus usos (ocurrencias aplicadas).

2. Se genera un segundo PST de manera que todas las referencias al lenguaje objetivo son escondidas mediante interfaces.

3. Con un programa con la gramática para el lenguaje objetivo, se lee el nuevo PST y genera un archivo fuente.

Para efectuar la transformación, los autores definieron clases para implementar interfaces a fin de crear clases, métodos, expresiones y declaraciones. Este planteamiento modular se realizó pensando en la escalabilidad. El autor desarrolló tres experimentos para mostrar las capacidades.

La primera transformación es de BNF a una gramática. Los programas de gramática no representan nodos terminales en su gramática como otros lenguajes de programación. El flujo que sigue es analizar una gramática BNF, transformarla a la gramática del programa en Java y usar ese programa generado para volver a analizar la gramática original. El segundo experimento consiste en pasar de COBOL a Python y finalmente, el tercero es de Delphi a C#.

Kijin An, Na Meng y Eli Tilevich [3] proponen un enfoque de desarrollo de software multiplataforma que utiliza mapeos y transformaciones lineales, de manera similar a la presentada en este trabajo. Los autores proponen un diseño para automatizar la traducción de aplicaciones de un lenguaje a otro. Esta propuesta se basa en definir reglas de traducción, las cuales forman y asignan las equivalencias entre los diferentes elementos sintácticos.

La idea fundamental consiste en contar los códigos escritos en diferentes lenguajes, con el fin de identificar los bloques que son sintácticamente equivalentes (similitud de cadenas). Una vez identificados estos bloques, se crean los AST de ambos lenguajes de programación. Esto permite aprovechar el dominio minimalista de la correspondencia de los lenguajes, por ejemplo, los operadores y palabras reservadas.

Posteriormente se realiza el alineamiento de los nodos del AST para inferir tanto la sintaxis como las reglas de asignación. Esto permite almacenar las reglas como plantillas de cadena

y con ello traducir el código que se desea. Las pruebas que se realizaron para validar el diseño arrojaron que la implementación tiene un 76% de precisión, lo que hace para esta investigación insuficiente porque se busca traducir la totalidad de las sentencias de un programa. Sin embargo, los autores consideran que la herramienta ayuda a la conversión y que supera a otras como j2swift. Cabe resaltar que según las pruebas mostradas en el trabajo Kijin An, Na Meng y Eli Tilevich [3] los elementos sintácticos más frecuentes son los que se muestran en la tabla 2.1.

Posición	**Elemento sintáctico**
1	Declaraciones de variables locales
2	Sentencias condicionadas
3	Declaraciones de importaciones
4	Asignaciones
5	Declaraciones de cuerpo de clase
6	Retornos
7	Expresiones
8	Declaraciones de sentencias

Tabla 2.1: Elementos más comunes en programas escritos en Java

La investigación desarrollada por Lachaux, Roziere, Chanussot y Lample [36], consiste en un transpilador, cuyo objetivo es traducir de un lenguaje a otro. Según los autores, el uso más común es migrar de un lenguaje de programación obsoleto a uno actualizado. Se propone aprovechar los enfoques de traducción automática sin supervisión para entrenar un transcompilador basado en aprendizaje no supervisado.

El modelo desarrollado se entrena con código fuente de proyectos de GitHub, los cuales deben estar abiertos. El método se basa exclusivamente en código fuente monolingüe, además se crea y se lanza un conjunto de pruebas paralelas. Cabe resaltar que dicha investigación opera de manera correcta con lenguajes de un nivel de abstracción similar, debido a que los AST tienen cierta equivalencia entre ellos. La idea de realizar la investigación en torno al aprendizaje no supervisado fue impulsada porque las reglas para modificar el AST de un lenguaje de programación suelen ser complicadas en caso de que no se conozca el lenguaje de origen y el de destino, además, traducir de lenguajes no tipados como Python a tipados como C++ aumenta la dificultad, porque se deben de inferir los tipos de las variables y funciones.

La tesis realizada por Andersson [5] es otro trabajo que se relaciona con el presente proyecto. El autor propone una transformación de código mediante la representación de un archivo XML. El objetivo principal de la investigación desarrollada es recibir como entrada código HTML y transformarlo en C. Para *parsear* el código HTML utilizan una biblioteca de terceros, posteriormente diseña un archivo XSD donde se proporciona la estructura que debe tener el "programa" para validar la gramática. El código se genera por medio de mensajes de tipo CAN, los cuales poseen la información para indicar el

elemento que se debe generar en el lenguaje C. Estos mensajes se almacenan en una lista para que el parser pueda acceder a su contenido cada vez que se encuentra un nodo en el archivo XML. Cabe destacar que la flexibilidad de analizar otro tipo de archivo se pierde, ya que no todos los lenguajes de programación cuentan con un XSD.

Un trabajo que también utiliza XML es el realizado por Teduh et al. [18]. En este caso, los autores buscan cómo pasar de un pseudocódigo a código fuente, en específico C++. Para realizar esto, crean un modelo intermedio que consiste en un meta-modelo en el cual buscan representar el pseudocódigo de una manera más ordenada. Mediante una jerarquía de modelos, tanto el pseudocódigo como el modelo intermedio se encuentran en un modelo independiente de la plataforma. En contraste, el código fuente es un modelo específico a la plataforma. El modelo se encuentra XML por lo que se puede indicar el esquema de XML del meta-modelo. Una estructura es utilizada para representar el meta-modelo. El núcleo lo definieron como "especificación" por lo que a partir de ahí empieza a crecer el problema. Para realizar la transformación, se debe seguir una plantilla la cual define una gramática para guiar el proceso. La herramienta obtiene un árbol de parseo, mediante ANTLR, del pseudocódigo para traducirlo a un modelo intermedio en XML. Con una herramienta que utiliza hoja de estilos XSL se realiza traducción del modelo intermedio al código C++.

Por otro lado, la investigación desarrollada por Shettym Saldanha y Thippeswamy [58] da origen a CRUST, cuyo nombre se debe a que es un transpilador que convierte programas escritos en C/C++ en código Rust, el cual es un lenguaje creado por Mozilla y que se ha vuelto cada vez más popular en la industria de la informática. En el proceso de la conversión se utiliza una metodología que los autores denominan *Nano-Parser*. Estos consisten en analizadores sintácticos pequeños, cuya función es limitada y tienen como objetivo trabajar en conjunto con otros analizadores, cada uno es diseñado para analizar una gramática específica y posteriormente ser orquestados por un *Master Parser*, quien es el que realiza las llamadas a los distintos *Nano-Parsers* y con ello procesar una entrada de texto compleja. La arquitectura utilizada por CRUST se basa en dos bloques, el primero consiste en un módulo encargado de analizar la sintaxis y el segundo consiste en el generador de código. El procesamiento sintáctico se realiza por medio de los *Nano-Parsers*, mientras que la generación de código se realiza con una función de emparejamiento cuando una expresión regular reconozca algún patrón válido que corresponda a código Rust. Este enfoque tienen ciertas limitantes como lo son generar*parsers* para todos los lenguajes que se desean analizar y la dificultad de generar las expresiones regulares para distintos lenguajes de programación, como por ejemplo RPG.

Otra investigación que está relacionada con transcompiladores es "gLua", la cual fue desarrollada por Lei Mu [41] y su principal objetivo fue desarrollar una herramienta que tradujera de un lenguaje de programación a otro, ya que según los autores es deseable reutilizar y optimizar código fuente. Los autores proporcionan un marco de trabajo con el propósito de compilar código fuente utilizando el lenguaje Lua, lo cual es una clara diferencia con la investigación que se propone en este documento, ya que en este caso el análisis se enfoca en la sintaxis. Sin embargo, la metodología utilizada es similar a la que se desarrolla en este trabajo.

En la figura 2.3 se muestra la arquitectura que se plantea en [41], la cual presenta una representación intermedia (IR, por el inglés *Intermediate Representation*) con el fin de compilar la IR de acuerdo con el lenguaje de programación que se desea, por ejemplo, si se desea Java, se utilizaría JVM, en caso en caso de que sea C, se utilizaría GCC.

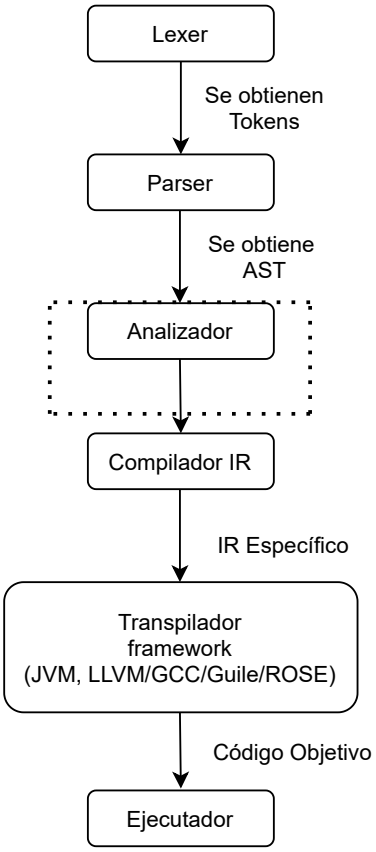


Figura 2.3: Arquitectura del transcompilador de gLua [41]

Cabe destacar que en el artículo se menciona que si alguna característica del código fuente no se puede modelar con Lua, se opta por tratar de emular su comportamiento, sin embargo, algunos casos se tiene que que adecuar el código para que se pueda cumplir con representación en el IR. Un ejemplo de lo anterior es la sentencia "continue", la cual se representa aplicando refactorización de código o intercambiando el orden de las sentencias.

También existe la investigación realizada por John Bradbury [9], la cual consiste en la creación de un lenguaje llamado JASMINT, con el fin de transformar un lenguaje de programación en otro. El transpilador que se propone está compuesto por dos módulos de software, el primero es el transpilador de C++ y el segundo corresponde al transpilador de Python.

JASMINT es un lenguaje, cuya gramática está disponible en ANTLR y puede ser utilizado como una biblioteca para crear los proyectos de programación y posteriormente utilizar los transpiladores. Esta biblioteca proporciona las funciones de AST como el parseo, además posee una interfaz gráfica sencilla para que el desarrollador interactúe, ya que se necesita configurar, ejecutar y serializar los archivos. Lo anterior con el fin de utilizar el

transpilador de Python o C++, cuyo insumo es el AST serializado y la salida es el código equivalente, ya sea C++ o Python.

También existen otros trabajos relacionados con la generación de código que utilizan el enfoque de aprendizaje profundo (*Deep Learning*). Tal es el caso del que desarrollan Ying y Neubig [64], quienes proponen una arquitectura de neuronas utilizando decodificadores y encodificadores para generar código a partir de un AST. El encodificador utiliza una red LSTM (*Long Short Term Memory*) para computar un conjunto de palabras, mientras que el decodificador utiliza una RNN (*Recurrent Neuronal Network*) para modelar el proceso de generación secuencial de un AST definido. Por esta razón una secuencia de *tokens* puede ser interpretada con el desarrollo de la RNN en cada uno de los pasos de tiempo. Cabe resaltar que la manera en que se mantiene el rastreo de una secuencia en la RNN es por medio de una función de probabilidad. Además de que la generación de tokens se realiza por medio de un vocabulario predefinido o directamente copiado desde la entrada de un lenguaje. La investigación tiene como objetivo generar un AST aplicando acciones desde un modelo gramatical (Red Neuronal).

Por otro lado, autores como Rabinovich, et al. [55] introducen el concepto de redes de sintaxis abstracta, con el objetivo de producir mapeos no estructurados en la generación de código. En estas redes las salidas se representan como AST y se construyen mediante un decodificador (*Decoder*) con una estructura modular, la cual se determina dinámicamente y de forma paralela a la estructura del árbol de salida. Los autores utilizaron el *benchmark* "HEARTHSTONE" para la generación de código, los resultados arrojaron un 79.2% BLEU (Bilingual Evaluation Understudy) y un 22.7% de precisión de coincidencias exactas. El modelo utiliza una arquitectura de codificador-decodificador jerárquica. Este enfoque permite estructurar el decodificador como una colección de módulos recursivos entre sí. Estos módulos corresponden a los elementos gramaticales del AST. El codificador utiliza una LSTM bidireccional para incrustar cada componente, los cuales son aplicados sobre la entrada de cada pasa para el proceso de decodificación.

Una alternativa para realizar la traducción a otros lenguajes es por medio de Traducción automática estadística (SMT por sus siglas en inglés). La investigación propuesta por Oda, et al. [48] está enfocada en pasar código fuente a pseudocódigo. Para ello utilizan SMT para traducir de Python a lenguaje más natural para las personas. El proceso que los autores establecen es tomar el archivo y revisar palabra por palabra con el fin de determinar cuál sería la salida más probable de acuerdo con el modelo definido. Luego, por medio de árboles sintácticos se define la estructura que tiene el código con el fin de realizar ajustes, en caso de ser necesarios para adaptarse al lenguaje del pseudocódigo. Los marcos de trabajo utilizados por los autores tienen un enfoque estadístico por lo que deben entrenar los modelos con el fin de que se puedan describir nuevas reglas de traducción. Como último paso del método, se tiene una evaluación automática que calcula la similitud de las traducciones realizadas con referencias creadas por humanos. Este método aunque es automático y rápido, no garantiza que el resultado sea correcto en términos semánticos. Para contrarrestar esta problemática, los autores realizan una evaluación humana para determinar qué tan aceptable es el producto final.

Similar a este último método, el trabajo de Nguyen, et al. [45] utiliza SMT para transformar desde Java para Android hacia C# para Windows Phone. La idea principal es hacer uso de SMT para inferir reglas de traducción de códigos ya migrados, por lo que no sería necesario definir de manera manual otras reglas. Para generar el resultado, los autores dividen el procedimiento en tres fases. Primero realizan un entrenamiento haciendo uso de los ASTs del código fuente y producen secuencias con anotaciones. Luego, se continúa con el entrenamiento, pero con elementos semánticos para finalizar con lexemas. Estas fases se combinan para procesar el código fuente final en el lenguaje de programación C#.

La investigación propuesta por Drissi, et al. [20] utiliza un modelo de codificación y decodificación de árbol a árbol. La idea central es tomar la estructura jerárquica que presentan los programas. Primero, se realiza una codificación de la entrada por medio de una lista de símbolos que se utilizan para determinar el más probable de ese conjunto. Se utiliza un entrenamiento para mejorar el proceso de codificación. Luego, se continúa con una etapa de decodificación donde se crean los nodos desde la raíz y se va generando cada hijo en el lenguaje de programación meta.

Siguiendo la misma idea, el trabajo realizado por Chen, et al. [11] utiliza también redes neuronales. De igual manera, presentan codificación y decodificación de árbol a árbol. Los autores plantean un mecanismo que cuando el decodificador expande un objetivo que no es terminal, el mecanismo ubica la porción del árbol en el árbol fuente para guiar la expansión del decodificador. Una propiedad que se destaca es que el proceso de traducción es modular.

Como primer paso para poder desarrollar el traductor, se requiere que los árboles sean binarios. La razón por la que realizaron esta tarea, es que los árboles pueden tener múltiples ramas, pero manejar árboles que son binarios pueden ser más efectivos. Como codificador se tiene una red LSTM para árboles con el fin de calcular los espacios para todo el árbol del código fuente y todos los subárboles. Se usan nodos y cada uno tiene un hijo izquierdo y uno derecho. Por su parte, el decodificador se encarga de generar el árbol objetivo iniciando por una raíz. Al principio copia el primer estado LSTM y lo usa como raíz. Luego, haciendo uso de una cola de todos los nodos que pueden expandirse, permite realizar recursión para expandir cada uno de sus nodos.

En otro uso de las redes neuronales, los autores Urim Roy, et al. [2] se enfocaron en generar código en un contexto dado de un programa y una parte de dicho programa que lo denominan p. De esta manera se predice p siempre y cuando sea un subárbol válido dentro del árbol abstracto sintáctico.

Una característica es que realizan un modelo de lenguaje estructural (SLM), el cual estima la probabilidad del AST del programa mediante la descomposición en un producto de probabilidades condicionales en su nodos. Para lograr esto, realizaron una red neuronal que calcula esas probabilidades tomando en cuenta todos los caminos del AST para llegar al nodo objetivo. Para codificar los caminos del AST, se utiliza la representación de vector. Cada uno de los nodos está compuesto por un par con el tipo de nodo y un índice entre los nodos en el mismo nivel. Cuando se tiene un grupo de caminos al padre del nodo

objetivo, la meta es representarlo como un vector con el fin de predecir ese nodo padre. Mediante una función de agregación con parámetros del conjunto de caminos, el camino a la raíz y el índice del nodo que actualmente se está prediciendo. Para predecir se utiliza una función *softmax* en el nodo.

Capítulo 3

Transformación de código específico a universal

En este capítulo se detalla la metodología de la investigación y el diseño del método desarrollado. El método consiste en la implementación de los experimentos para probar las hipótesis planteadas en el capítulo 1.

El método diseñado tiene como objetivo analizar de manera sintáctica el código fuente. Sin embargo, también proporciona escalabilidad para ejecutar análisis semánticos en los lenguajes que así lo requieran.

Este proyecto se desarrolló utilizando un proceso incremental que consiste de cuatro fases, como lo muestra la figura 3.1:

1. En la etapa de investigación y análisis se determina cuáles otras soluciones se han implementado para problemas similares, posteriormente se realizaron reuniones con los líderes técnicos y se discutió sobre la viabilidad y sobre cuál opción podría ser la mejor base para la solución.
2. El diseño del *framework* se planteó utilizando como base la literatura existente.
3. La implementación es la etapa donde se desarrolla el código de extracción, mapeo y generación de datos.
4. Cuando una funcionalidad se terminaba de implementar, se debían realizar las pruebas funcionales correspondientes en la fase de verificación.

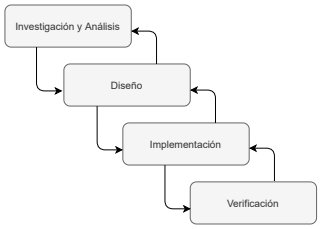


Figura 3.1: Metodología utilizada en el proyecto

Esta metodología permite adaptarse al cambio y corregir los errores que se presentan durante el tiempo de desarrollo del proyecto, porque la verificación en cada ciclo brinda la posibilidad de recibir retroalimentación y mejorar funcionalidades.

3.1 Diseño general del *framework*

En la figura 3.2 se muestra el diseño de la arquitectura general del *framework*, en la cual se puede observar que las entradas que recibe son los AST de los lenguajes específicos. Es decir, cada tipo de AST específico depende del lenguaje de programación fuente (RPG, Java o C#). Esto implica que se debe proporcionar un mecanismo que lee el código fuente almacenado en un repositorio o de manera local y extraer el AST correspondiente para cada uno de los lenguajes.

Cuando el *framework* toma el AST específico (SAST) necesita asociar sus elementos sintácticos a la estructura de datos genérica que está basada en el estándar MOF, que a su vez utiliza los conceptos teóricos de la especificación de un meta-modelo para describir para representar lenguajes KDM y ADM. Una vez concluido el mapeo, se procede con la validación, cuya función es validar que todos los elementos del código original están mapeados correctamente. Por último, se procede con la presentación del lenguaje universal proveniente del árbol de sintaxis abstracto genérico (GAST).

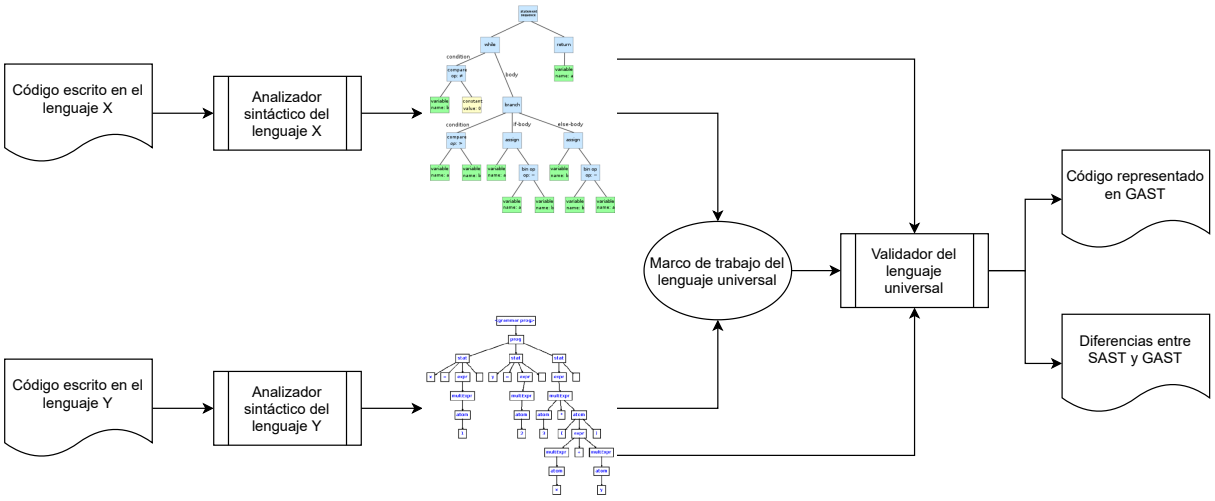


Figura 3.2: Arquitectura general del transformador de lenguaje específico a universal.

La figura 3.2 muestra las cuatro funciones principales que realiza el *framework*, las cuales se presentan en la figura 3.3 y se detallan a continuación:

1. Establecer la ruta donde se encuentran los archivos del código de programación que se transformarán. Cabe destacar que los archivos pueden estar en algún repositorio remoto de GIT o de manera local.

2. Elegir el lenguaje de los programas por analizar, ya sea Java, RPG o C#.

3. Trasformar el código deseado al lenguaje universal, con su respectiva validación.

4. Obtener el código del lenguaje universal.

La razón de que las funciones sean básicas y concretas es porque el marco de trabajo está orientado al procesamiento y no a realizar transacciones y es por esto la interacción con el usuario es poca. Una de las funciones del marco de trabajo es que el desarrollador proporcione únicamente la ruta de la raíz del proyecto, y que se logre un mejor flujo de la información al evitar que el usuario interactúe con el software.

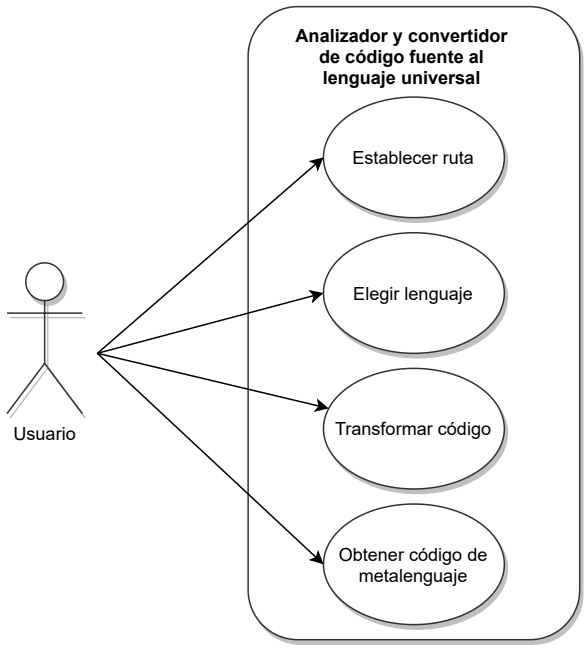


Figura 3.3: Diagrama de funcionalidades por parte del usuario.

En el proyecto se utilizan algunas herramientas desarrolladas por terceros para facilitar algunas funciones, como:

- Manipulación de cadenas.
- Consultas y recorridos del AST.
- Mapeo de los objetos.
- Administración de la jerarquía de archivos.
- Extracción de información de los archivos del código fuente.
- Almacenamiento de documentos en la base de datos.

En la figura 3.4 se muestra la interacción del sistema con las diferentes bibliotecas: Gson, MapStruct, Driver de MondoDV, ANTLR y JDT. Estas distribuciones son de uso libre.

Estas bibliotecas son incorporadas por medio de Maven, el cual facilita el proceso de descarga y administración de todas las dependencias y así brinda portabilidad entre plataformas. En la tabla 3.1 se muestra la función que cumple cada una de ellas dentro del marco de trabajo.

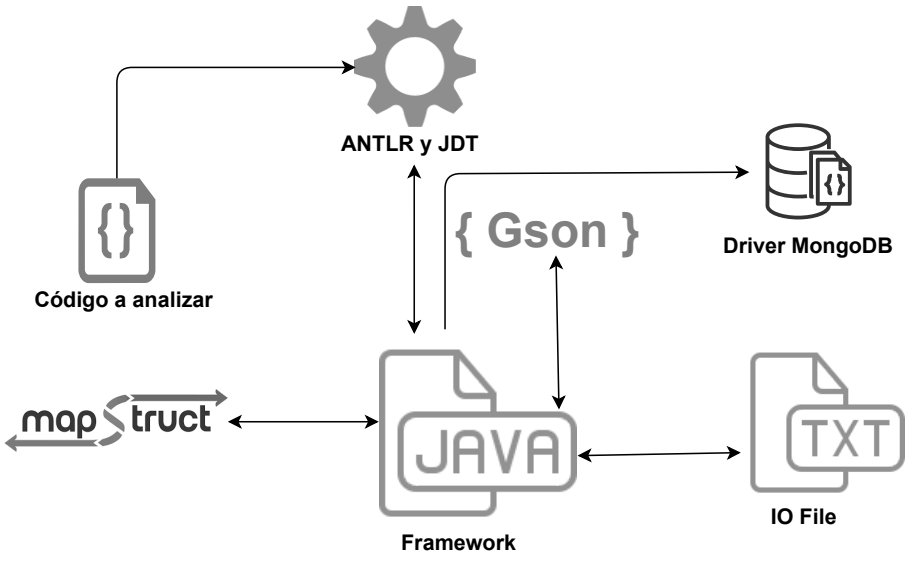


Figura 3.4: Arquitectura general del marco de trabajo.

La arquitectura mostrada en la figura 3.4 permite identificar las cuatro etapas que caracterizan los diferentes procesos de ejecución del software desarrollado, y que se detallan en la tabla 3.2. Por otra parte, la figura 3.5 muestra la secuencia de las etapas en el flujo de la información, así como los resultados (datos de entrada y salida) en sus interfaces. El resultado de una fase es la interfaz con la otra, por ejemplo, la etapa de mapeo necesita el AST proveniente de la etapa donde se extrae la información de los archivos fuente.

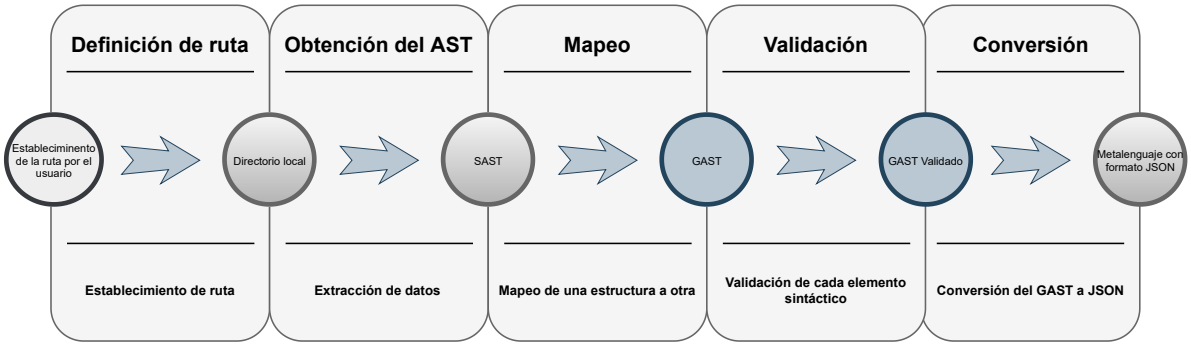


Figura 3.5: Etapas en el flujo de la información.

Biblioteca	Función
JDT	Es un *plug-in* de Java que permite desarrollar todo tipo de aplicación en este lenguaje. Es utilizado para tomar la información de los archivos .java que se quieren analizar. Esta herramienta retorna al programa el AST del código Java correspondiente a cada uno de los archivos .java.
ANTLR	Corresponde a una biblioteca que extrae gramáticas de distintos lenguajes de programación, posee la gramática de C# y RPG, por lo que genera el AST de cada uno de estos lenguajes.
MapStruct	Es una herramienta que realiza mapeos lineales entre dos objetos. Estos mapeos pueden ser encadenados, debido a que posee la propiedad de efectuar procesamiento recursivo. Se encarga de mapear todos los atributos del AST de Java, C# y RPG al GAST por medio de una interfaz. Le retorna al programa principal la raíz del GAST con los valores mapeados.
Gson	Esta biblioteca es la que se encarga de la manipulación y conversión de clases a JSON. El retorno es una cadena con formato JSON, la cual tiene la información del GAST.
Driver de MongoDB	Corresponde al API para conectar el *framework* desarrollado con el servidor de la base de datos de Mongo.

Tabla 3.1: Funciones de las bibliotecas utilizadas en el proyecto.

Etapa	Proceso
Definición de ruta	Se refiere a que el usuario establece la ruta donde están los archivos que desea analizar.
Obtención del AST	Con el directorio raíz establecido, se realiza la extracción de los datos de todos los archivos contenidos en dicho directorio. JDT realiza un parseo de línea por línea para obtener el AST de cada documento.
Mapeo	Una vez obtenido el AST de Java se procede al mapeo de cada elemento del mismo con los datos del GAST. Este proceso se realiza por medio de MapStruct, el cual devuelve la raíz del GAST.
Conversión	Esta etapa consiste en convertir la estructura de datos del GAST en un JSON, teniendo en cuenta que ambos son jerárquicos.

Tabla 3.2: Etapas y procesos en el flujo de información.

3.2 Diseño lógico del *framework*

En esta sección se detallan los aspectos de diseño del software desarrollado, mediante diagramas de clases, de paquetes y notación BNF.

La figura 3.6 muestra el diagrama de paquetes de alto nivel que representa el diseño

lógico del *framework*. El paquete principal corresponde "ASTMCore". Este se divide en tres componentes: semántico, sintáctico y de recurso. El *framework* realiza un análisis sintáctico, por lo que este paquete debe definir las declaraciones, definiciones, directivas, expresiones, instrucciones de control y tipos. Cada uno de estos componentes se encuentra en un paquete independiente. Las próximas sub-secciones detallan el contenido de cada paquete.

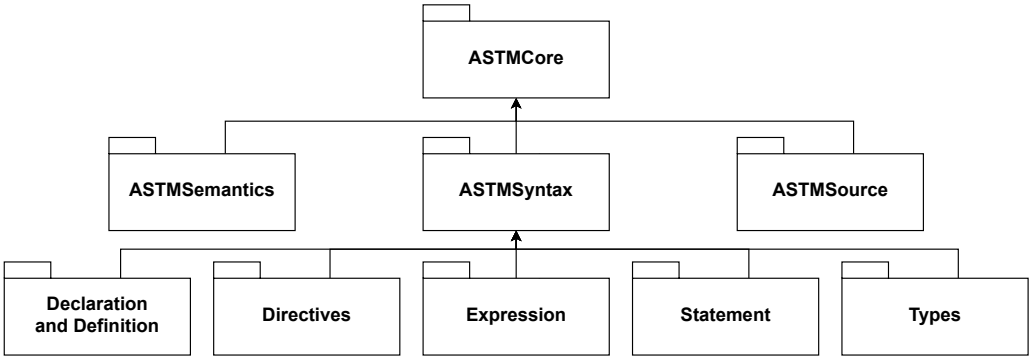


Figura 3.6: Diagrama de paquetes de alto nivel del diseño lógico del *framework*.

3.2.1 Paquete semántico

Este proyecto no se enfoca en aspectos semánticos, sin embargo, algunos elementos son necesarios para establecer relaciones entre elementos sintácticos, como por ejemplo el alcance de una variable dentro de un bloque de código.

Se realiza el modelado de diferentes alcances de los elementos sintácticos con el fin de identificar el espacio del código donde un componente es válido, por ejemplo, una variable definida dentro de un "if" no es válida en la sentencia "else" del mismo. Otro aspecto relevante es la recursividad que debe de existir en la clase "Scope" (encargada de modelar el alcance de un componente), esto permite que hayan sentencias dentro de otras, tantas como sea necesario, tal es el caso de los condicionales y ciclos anidados.

Cabe destacar que los archivos fuentes se modelan por el atributo "files" de la clase "Project", el cual corresponde a las unidades de compilación (estructura que contiene todos los elementos sintácticos del código que se desea analizar) que corresponden a cada clase o cada archivo de código del proyecto, en el caso de Java son los .java, en el caso de RPG son los .rpgle, mientras que el caso de de C# son los .cs.

3.2.2 Paquete de recurso

El foco de atención en este paquete debe recaer sobre la clase "CompilationUnit" que, como se mencionó anteriormente, contiene los elementos del código correspondientes a los archivos fuente. En esta clase existen atributos como lenguaje, paquete, alcance y lista de importaciones, los cuales representan la estructura básica de un código orientado a objetos.

Las definiciones de clases y enumerados son modeladas como estructuras que están dentro del alcance de dicho programa por lo que son representadas por una clase "Scope".

Las demás clases ofrecen una abstracción de los archivos fuente que permiten implementar funcionalidades como: localización del archivo, localización de líneas de código en el archivo y referencias a archivos externos.

3.2.3 Paquete sintáctico

Debido a que este proyecto está orientado a reconocer las sintaxis de lenguajes ya existentes, este paquete es el que contiene mayor cantidad de clases, debido a que es el encargado de modelar las distintas formas de escritura de los lenguajes de programación. Todos los diagramas de clases se pueden encontrar en [52], en este trabajo se mencionan los principales de ellos para entender el fundamento teórico y de diseño que hace posible el modelado genérico de algunas gramáticas.

Este paquete lo conforman 5 subtipos de paquetes, los cuales se observan en la figura 3.6 y se detallan a continuación:

1. Declaraciones y definiciones: Permite modelar todos los elementos que se pueden declarar o definir, como las variables, clases, métodos, enumerados entre otros.
2. Directivas: Se refiere a elementos de pre-procesamiento. Debido al alcance del proyecto no se utiliza en la traducción del código.
3. Expresiones: Hace posible la representación de cualquier expresión válida para un lenguaje de programación, como es el caso de las asignaciones, llamadas a métodos, creación de objetos, operaciones aritméticas, entre otros.
4. Sentencias: Modela el uso de las estructuras de control de flujo y de instrucciones básicas, por ejemplo, while, for, switch, if, return break entre otros.
5. Tipos: Modela todos los tipos de datos que existen y los que se pueden crear, como las clases, *integers*, *boolean* y *array*.

3.2.4 Paquete de declaraciones y definiciones

La figura 3.7 se muestra la estructura básica de este paquete, en donde hay dos tipos de objetos, declaración y definición, los cuales pueden ser de diferentes clases, entre ellas, las variables, funciones y enumeraciones. Cabe señalar que el Objeto "DataDefinition" tiene un atributo llamado "initialValue" que se utiliza para asignar valores iniciales cuando se realiza alguna definición, por ejemplo, en el siguiente código:

```
1 int a;
2 double b=21;
```

En la primera línea se muestra una declaración sencilla de una variable entera llamada “a”. Sin embargo, la segunda línea (double b=2) corresponde a una definición de tipo “double” con un valor inicial de 21. Por esta razón es que es necesario el atributo “initialValue”, y que sea de tipo Expression porque los valores pueden contener varios componentes, como sumas, restas, multiplicaciones y divisiones. También cabe destacar, la relación numérica que tiene este dato, la cual es 0 ó 1, ya que puede ser inicializado o no .

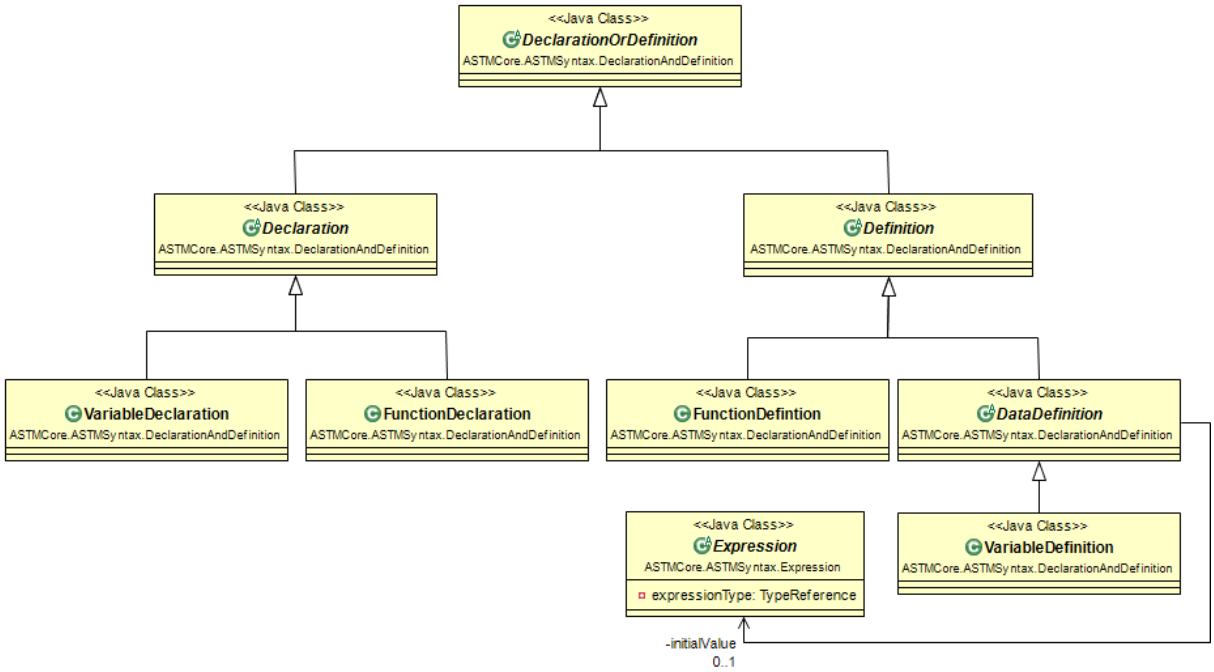


Figura 3.7: Base estructural del paquete de declaraciones y definiciones.

3.2.5 Paquete de expresiones

Este paquete se encarga de modelar las instrucciones que son compuestas (más de un elemento) y que tienen algún tipo de relación con otras, por ejemplo, las asignaciones se pueden identificar como expresiones binarias donde existe un operando izquierdo, el operador = y el operando derecho.

Un aspecto que se debe resaltar es la recursión que debe existir para un elemento que contenga expresiones, ya que algún componente puede tener más de una expresión y así sucesivamente. En la figura 3.8 se muestra un ejemplo de este caso con una instrucción que involucra expresiones compuestas.

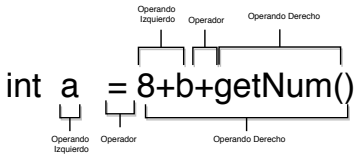


Figura 3.8: Ejemplo de expresiones dentro de una expresión.

En la figura 3.8 el operando derecho es también un componente de tipo "Expression", ya que es un conjunto de sumas las cuales son modeladas como expresiones. Es por este motivo que la recursividad en este paquete es necesaria, ya que se puede tener un número indeterminado de expresiones anidadas (sub-expresiones). En las figuras 3.9 y 3.10 se muestra las principales clases que componen este paquete. Se puede observar los diferentes tipos de expresiones y cómo implementan la recursión en cada caso.

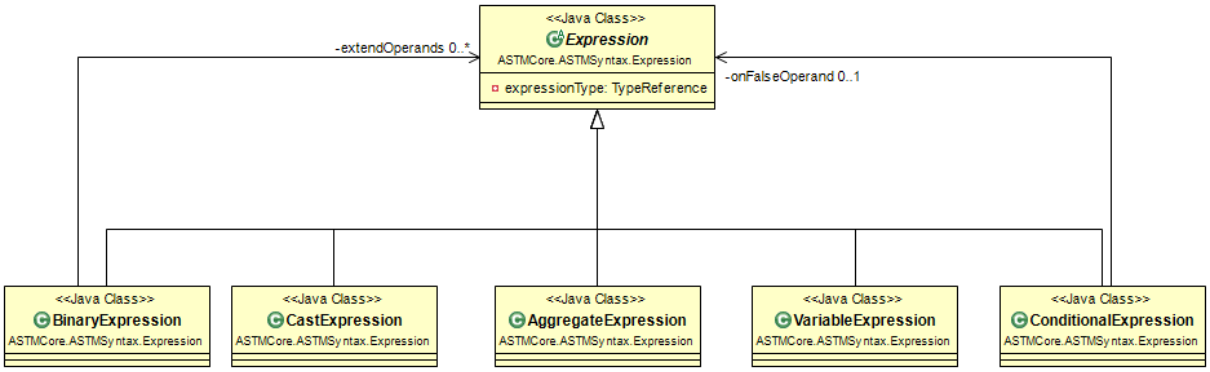


Figura 3.9: Principales clases del paquete "Expression".

La recursión aparece en la mayoría de las clases mostradas en las figuras 3.9 y 3.10, ya que todas ellas tienen un atributo de tipo "Expression".

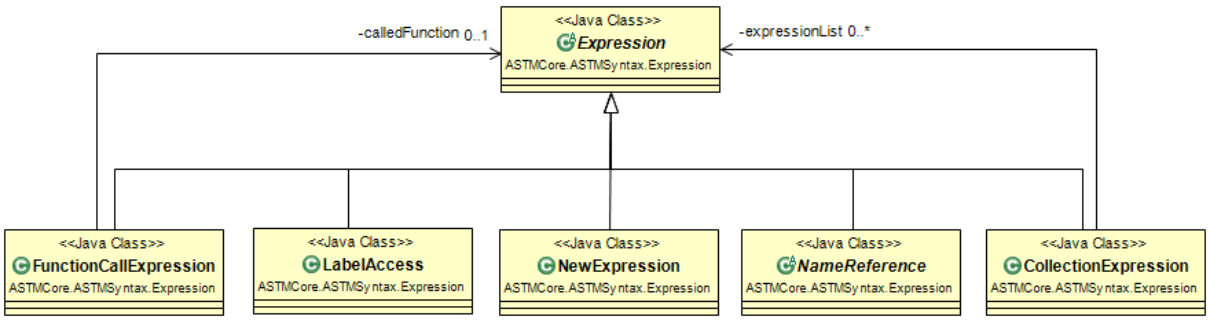


Figura 3.10: Continuación de la arquitectura del paquete "Expression".

Las expresiones, generalmente, poseen operandos y operadores, por lo que es necesario modelar este tipo de comportamiento. En la figura 3.11 se muestra las clases básicas que conforman el modelado de expresiones binarias, sin embargo, el GAST puede contener subárboles de otros tipos.

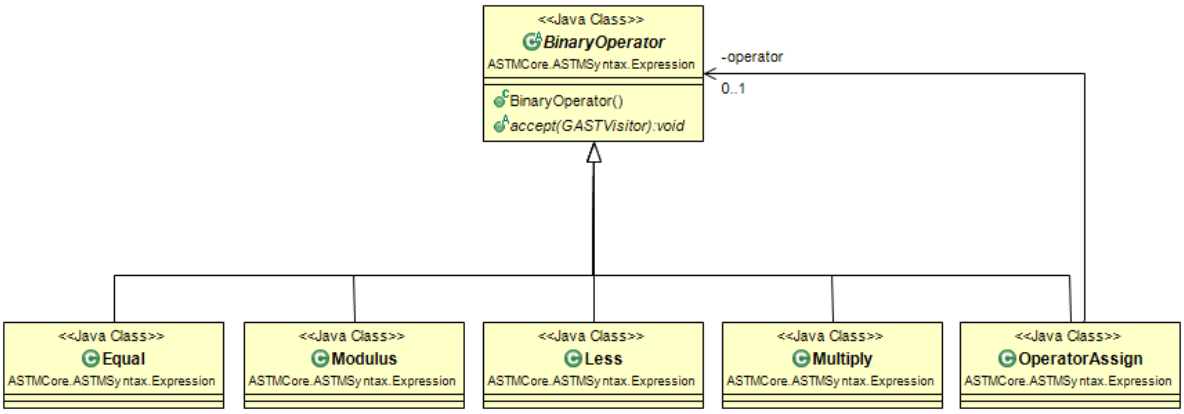


Figura 3.11: Clases básicas para las expresiones binarias.

3.2.6 Paquete de instrucciones de control (Statement)

Se encarga de modelar toda instrucción que modifique directa o indirectamente el flujo de información, algunos ejemplos son: while, if, for, switch, return y break.

En la figura 3.12 se muestra las diferentes sentencias que se pueden modelar con este estándar, cabe recalcar que existe la necesidad de realizar de alguna manera la recursión de sentencias debido a que una sentencia puede tener otras en su interior. En este caso la recursión se modela con un atributo de tipo "Statement" como se observa en las figuras 3.12 y 3.13. Por ejemplo la clase "IfStatement" tiene los atributos "thenBody" y "elseBody" (Statement) porque ambos también son sentencias que a su vez tienen dentro de su alcance otros objetos del mismo tipo.

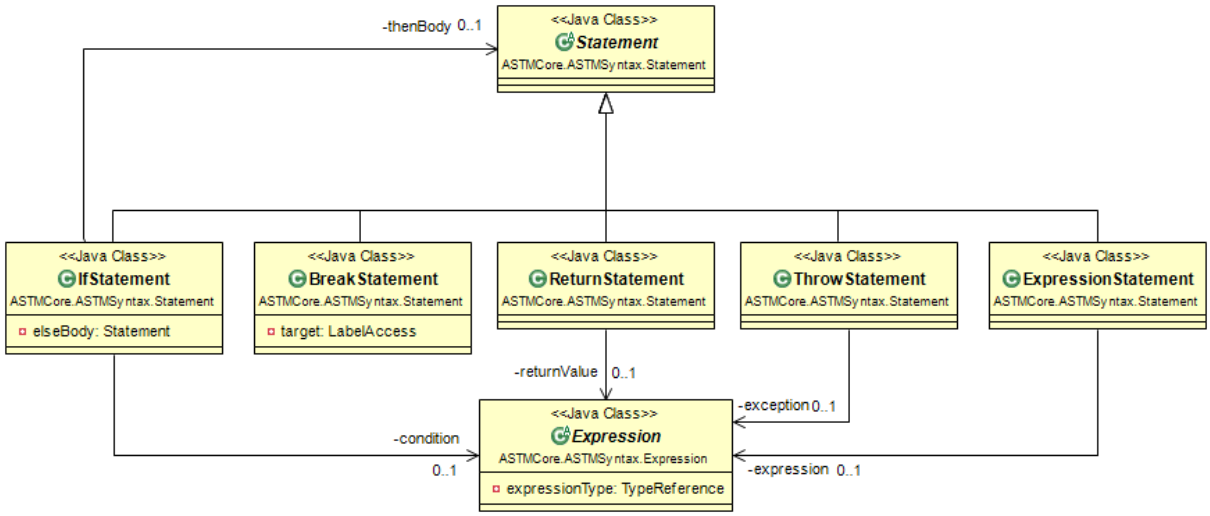


Figura 3.12: Principales clases del paquete "Statement".

Generalmente las instrucciones de control involucran alguna expresión que indica la forma como controlarán el flujo, por esta razón es que las clases "IfStatement" y "ReturnS-

tatement" (mostradas en la figura 3.12) poseen una relación de asociación con la clase "Expression".

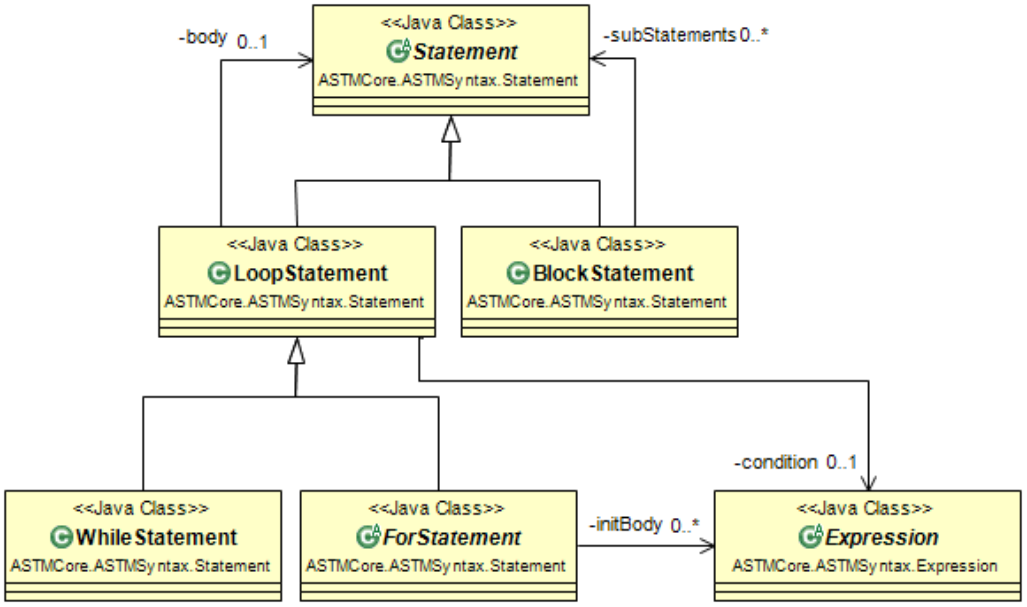


Figura 3.13: Continuación de la arquitectura del paquete "Statement".

Este paquete se relaciona con el de expresiones debido a que la mayoría de las sentencias tienen en su estructura algún componente que es una expresión como, por ejemplo, los condicionales. Lo anterior se puede observar en la figura 3.13 donde las sentencias de ciclos tienen atributos de tipo "Expression".

3.2.7 Paquete de tipos

El rol de este paquete es esencial para el proyecto, ya que indica los tipos tanto de los datos primitivos como los que se pueden crear, además modela la clase de elemento sintáctico. Por ejemplo, cuando se hace una declaración de variable se debe indicar el tipo o cuando se crea una expresión también se debe indicar su clasificación.

En la figura 3.14 se puede observar los subtipos del objeto "Type", "DataType" es una clase abstracta que modela la clasificación de las estructuras de datos, por ejemplo, enumerados, clases, arreglos, identificadores, parámetros entre otros. La figura 3.15 muestra la expansión de "DataType", la cual contiene una estructura que permite dividir cada elemento en subtipos, como es el caso de "FormalParameterType". El diagrama completo de este paquete se puede encontrar [52].

El modelado que se realiza para una clase, que a su vez representa un objeto, es por medio de "AggregateType", la cual es una clase abstracta que tiene como subtipo a "ClassType", ésta última es la que se encarga de representar una abstracción del mundo real.

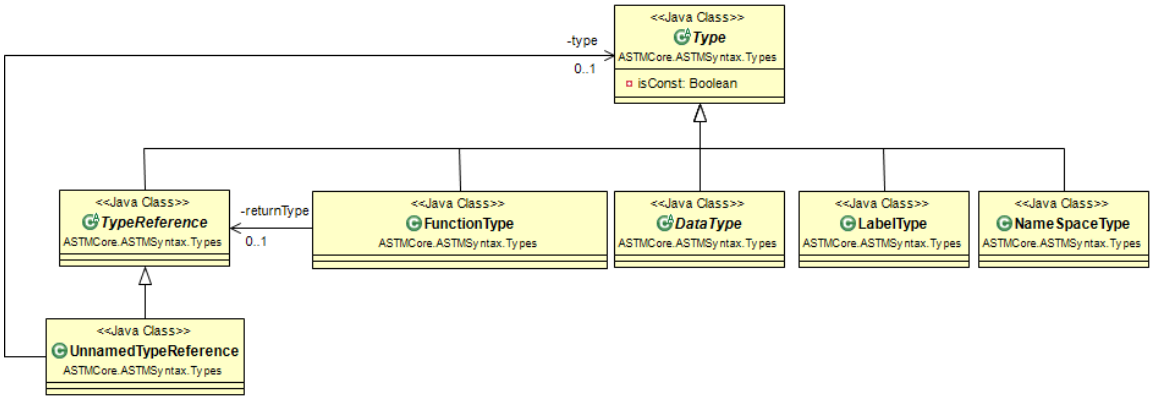


Figura 3.14: Principales clases del paquete "Type".

La clase "CostructedType" de la figura 3.15 es abstracta, ya que es la encargada de modelar los tipos de datos que son compuestos, como es el caso de los arreglos, estructuras y punteros. La clase "PrimitiveType" también es abstracta debido que de ella derivan todos tipos primitivos existentes, como por ejemplo "int", "double", "char", "float", "byte" entre otros, los cuales tiene un identificador que corresponde con el tipo, en el caso de int, el identificador es "int".

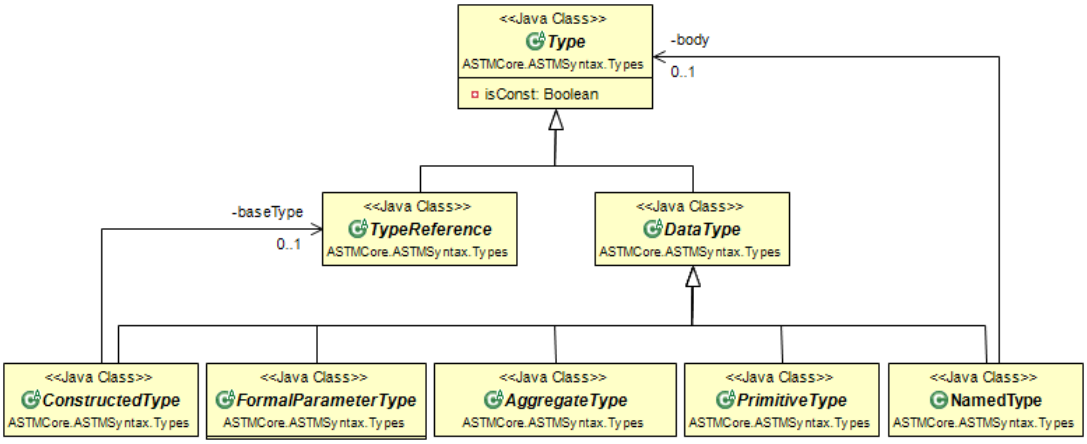


Figura 3.15: Continuación de la arquitectura del paquete "Type".

3.2.8 Clases de los paquetes API, ASTJAVA y JavaMappings

Estos paquetes están compuestos por una única clase, la cual cumple funciones específicas de acuerdo con el rol que tienen dentro del proyecto, cabe resaltar que las clases son semejantes para cada uno de los lenguajes. A continuación se detalla la función de cada paquete, en el caso de Java.

API: Se encarga de crear una interfaz que sirve como intermediario entre el SAST y el GAST, pues MapStruct no es capaz de mapear objetos muy complejos que dependan de otros con clases abstractas.

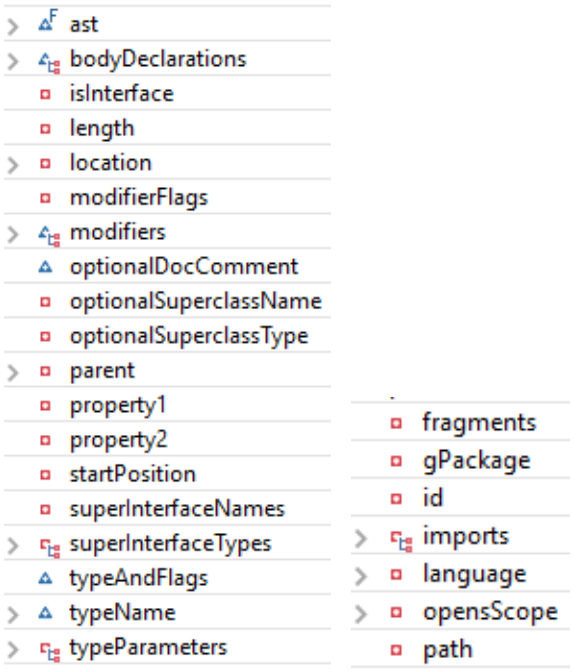
ASTJava: En este paquete es donde inicia el flujo de la información, ya que es aquí donde se obtienen todos los archivos y se extraen sus datos.

JavaMappings: Se encarga de alojar los mapeos de uno a uno entre el SAST y GAST, los cuales son escritos utilizando MapStruct.

3.3 Validador del GAST

El diseño de la herramienta que compara las estructuras debe tomar el GAST y el AST, para recorrerlos en su totalidad con el fin de determinar diferencias. Un aspecto para considerar es que si presentan nombres distintos en los nodos, así como información propia de cada estructura, esta debe ser ignorada porque son datos fuera del dominio sintáctico del código fuente. Por ejemplo, en un AST de Java pueden encontrarse los comentarios del código fuente.

Como se observa en la figura 3.16, las dos estructuras mostradas presentan múltiples nodos desde la raíz. Para el caso del AST (figura 3.16a) hay elementos que son propios de esa estructura que no se encontrarán en el GAST. Esto sucede porque el AST tendrá información adicional a los componentes sintácticos. De esta manera, se debe realizar un estudio de los nodos que contengan detalles sintácticos del código fuente al comparar con el GAST. Además, es necesario un estudio para determinar cuál es el nodo equivalente entre ambas estructuras porque pueden diferir en el nombre.



(a) Raíz del AST. (b) Raíz del GAST.

Figura 3.16: Visualización del nodo raíz.

3.3.1 Arquitectura del validador

La figura 3.17 muestra el diseño de la arquitectura del validador, con los flujos de los diferentes datos. Se puede observar que el programa presenta como entradas los árboles de sintaxis abstracta del lenguaje universal y del lenguaje específico.

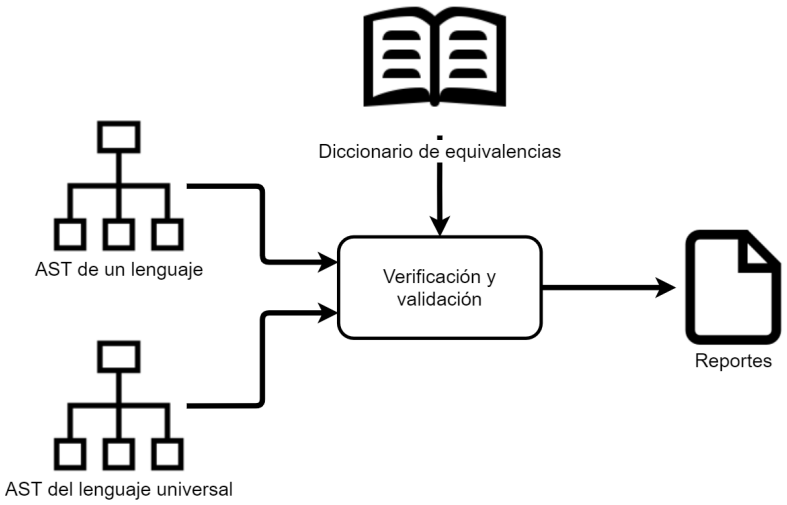


Figura 3.17: Validador: Diseño de la arquitectura a un alto nivel.

El proceso de validación y verificación es una tarea que se realiza automáticamente al finalizar el mapeo del lenguaje específico. Por esta razón, no hay interacción humana para indicar al validador qué debe comparar en las dos estructuras. El elemento que ayuda en esta tarea es un diccionario que brinda al validador las entradas con las equivalencias de nombres para relacionar las dos estructuras. Con estos datos, el validador determina el nodo equivalente del GAST con respecto del AST y obtiene los nombres de las hojas para verificar que la información contenida es equivalente.

El propósito de esta fase en el proyecto es brindar un visto bueno al desarrollador para indicar que la transformación de un lenguaje de programación a un lenguaje universal es correcta y todos los elementos que se encuentran en el GAST están presentes en el AST original. La labor se va a ejecutar en cada uno de los archivos que previamente han sido analizados por la herramienta y los resultados de esta fase serán las entradas para el proceso de verificación. Esto aprovecha los recursos calculados porque el paso previo a la verificación consiste en la generación del GAST con la información del AST.

La última fase es la generación de reportes. El listado consiste en la dirección del archivo, el valor en el AST y el valor en el GAST. El propósito es informar a los desarrolladores del proyecto las diferencias que posee la estructura del GAST con respecto del AST. En este informe se detalla cada elemento analizado y se presenta cualquier diferencia en la información. El reporte de diferencias debe indicar de la manera más detallada los errores presentes en los árboles analizados. El propósito es asegurar que la información que será enviada por medio del GAST cumpla con los requerimientos que, en este caso, consiste en crear un mapeo exacto de un lenguaje de programación al lenguaje universal.

El proyecto utiliza varias herramientas para facilitar las tareas que se deben ejecutar, por ejemplo, la manipulación de cadenas. La figura 3.18 muestra la relación del programa con los diferentes componentes que interactúan con el sistema. El programa recibe como insumo un documento con las relaciones entre el AST y GAST, así como el GAST generado en la etapa anterior. Por su parte, el validador generará un archivo con las diferencias encontradas. Estas relaciones se observan en la figura.

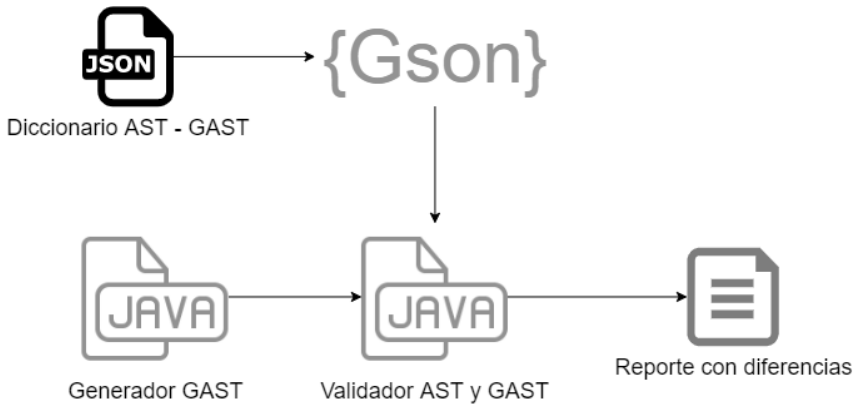


Figura 3.18: Arquitectura general del proyecto.

La única biblioteca con la que tiene una asociación es Gson, que es de uso libre. Su función es manipular el formato JSON del diccionario con las equivalencias de los diferentes métodos que se encuentran tanto en el GAST como en el AST. La biblioteca tiene como entrada un archivo en formato JSON el cual detalla las equivalencias según el lenguaje de programación del código fuente analizado.

El siguiente componente corresponde a un módulo llamado "Convertidor y analizador de código fuente al lenguaje universal" y tiene la función de generar el GAST a partir de un lenguaje determinado. La conexión que presenta es que el elemento contiene tanto el GAST como el AST del código fuente que está siendo analizado, por lo que facilita enviar las dos estructuras como un objeto al validador. Al ser objetos, los atributos pueden ser accedidos mediante los métodos que el paradigma permite con el fin de obtener un valor en específico.

El último actor corresponde al módulo de reportes. Este elemento va a recibir toda la información referente a las diferencias encontradas entre las dos estructuras. De esta manera el archivo generado contendrá los datos del nodo en el árbol que fue detectado, con el fin de indicar la ubicación del error.

3.3.2 Obtención de datos en el AST y GAST

El validador debe comparar el AST y el GAST. El AST, al ser un árbol sintáctico específico, es generado en el proceso del mapeo al GAST. Esta estructura será tratada como un objeto porque la programación de este proyecto está escrita en Java. Un objeto en este lenguaje presenta la ventaja que los atributos pueden ser recuperados mediante el llamado

de métodos asociados al objeto.

El lenguaje de programación Java permite utilizar reflexión. Esta característica obtiene el valor de un atributo realizando un llamado a un método perteneciente al objeto. De esta manera, mediante la ejecución de métodos sobre los nodos del AST se puede conocer la composición de cada nodo de la estructura. Al utilizar la raíz del AST como punto inicial, se puede recorrer todo el árbol sintáctico. La figura 3.19 muestra los atributos asociados a la raíz del AST. Cada uno de estos elementos debe ser accedido con el fin de compararlo con su equivalente en el GAST.

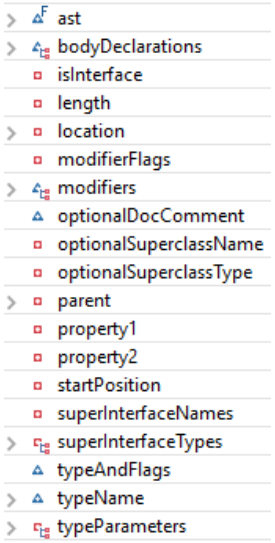


Figura 3.19: Estructura AST para una clase.

Cuando se ejecuta una llamada de un método perteneciente al objeto, el resultado consiste en otro objeto que tendrá múltiples atributos. La idea es recorrer, de manera recursiva, la estructura hasta llegar a un nodo hoja donde se presente el valor correspondiente a un elemento sintáctico tal como se muestra en la figura 3.20.

Figura 3.20: Nodo terminal del AST.

En este caso, este elemento no presenta más nodos con información correspondiente al valor de una variable, por lo que consiste el nivel más profundo del árbol para este atributo. El "1" será la valor que se comparará en el GAST para determinar que en esa otra estructura se encuentre el mismo valor.

En el caso del GAST se aplica el mismo principio porque está basado en el estándar MOF. Este estándar establece que para cada elemento existe un método para obtener y escribir valores (*getters* y *setters*). De igual manera que con el AST, en esta estructura se puede hacer uso de reflexión. Al aplicar una invocación de un método al objeto, se puede obtener

un nuevo nodo para ser analizado. La raíz para esta estructura se encuentra en la unidad de compilación y esto se debe a que es el mismo árbol, pero transformado a los nombres propios que presenta el metalenguaje. La figura 3.21 presenta el resultado de aplicar un método para obtener la información de una clase.

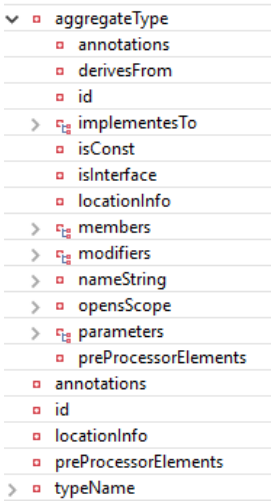


Figura 3.21: Estructura GAST para una clase.

En algunos casos, como en la figura 3.21, sucede que el nodo presenta información única para el metalenguaje. Por ejemplo, para caracterizar un nodo, es necesario indicar el tipo. Por tal razón, el metalenguaje incluye un nivel que indica que el nodo representa una clase. En la figura 3.21 esta excepción se observa en el atributo "typeName". Por lo tanto, hay ciertas características que no estarán en el AST de un lenguaje de programación.

3.3.3 Reflexión en los árboles sintácticos

En las sub-secciones anteriores se detalló la forma en que los datos pueden ser extraídos de las dos estructuras haciendo uso de la reflexión brindada por Java. Este procedimiento puede ser aplicado en los objetos para obtener los atributos o métodos pertenecientes a dicho objeto.

Los métodos presentan una ventaja sobre los atributos, ya que tanto el AST como el GAST presentan métodos públicos para obtener los valores. En el caso de los campos, por seguridad deben estar privados para evitar que los valores puedan ser extraídos. Por tal razón, en caso de utilizar reflexión para obtener los atributos, se debe conceder un permiso para obtener de manera correcta el valor.

En el caso de los métodos, se deben obtener todos los nombres asociados que serán comparados. Para realizar este proceso se deben ejecutar varios pasos a fin de conocer los datos relacionados con el objeto. La figura 3.22 muestra un ejemplo de cómo obtener una lista de métodos asociados al objeto. Dentro del ciclo *for* se pueden realizar diversas acciones, como comparar el nombre para ejecutar alguna acción.

```
// Obtener de elementos de un objeto
import java.lang.reflect.Method;

public class ReflectionMethods {
    public void printMethods(Object node){
        Method[] methodsNode = node.getClass().getMethods();
        for (Method method : methodsNode){
            System.out.println("Nombre:" + method.getName());
            // Acciones de cada elemento
        }
    }
}
```

Figura 3.22: Ejemplo de reflexión para obtener los métodos y sus nombres.

Las excepciones deben agregarse en los métodos, a fin de asegurar el correcto funcionamiento y manejar errores al llamar un método no presente en ese objeto.

3.3.4 Algoritmo validador y verificador de las estructuras

El verificador y validador de los dos árboles sintácticos es recursivo, porque el resultado generado se vuelve a analizar en el mismo procedimiento hasta encontrar un valor. Esto es necesario porque se debe revisar el nodo correspondiente, buscar el método que se asocia y ejecutarlo para obtener un nuevo nodo. El final es cuando se detecta una hoja que contenga un valor para comparar con la otra estructura. La figura 3.23 muestra un diagrama de flujo con los diferentes pasos que los datos deben tomar para llegar al resultado.

En la figura 3.23 se observa cada fase que pasa un nodo. Al inicio, mediante el uso de reflexión se obtienen los métodos que presenta la estructura. Cada uno de estos elementos será comparado con los métodos que presenta el otro árbol sintáctico para llegar a una equivalencia. Una vez que la igualdad ha sido encontrada haciendo uso de un diccionario, se procede a identificar si el nodo corresponde a una hoja. Una hoja es el punto terminal que presenta el valor de la palabra clave del código fuente. Puede ser el nombre de una variable, el valor de una constante, un literal o los modificadores.

En los casos de los árboles sintácticos, la condición se verifica si el nodo presenta un campo de tipo valor. Una vez encontrado esto, se procede a capturar dicho valor para comparar los dos recibidos. Si los resultados son distintos, corresponderá crear una entrada para el informe que detalla el problema encontrado.

A los nodos que estén en medio de la estructura se les vuelven a aplicar los pasos con el fin de navegar a través del árbol, en términos de profundidad, para bajar niveles. Estos nodos

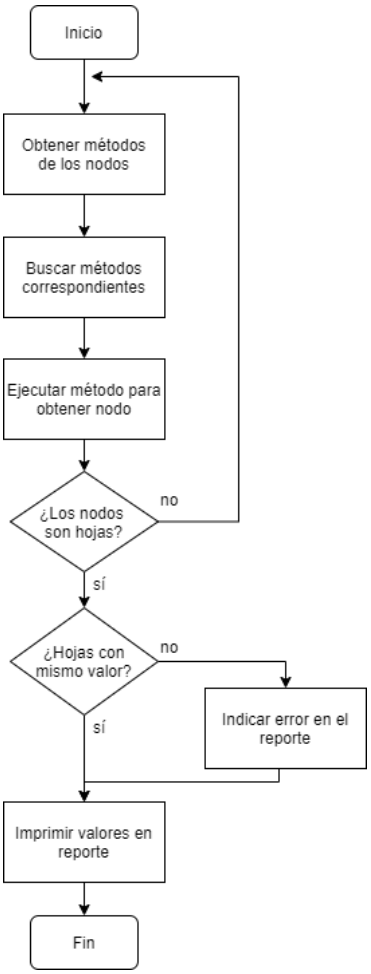


Figura 3.23: Diagrama de flujo del verificador.

pueden contener múltiples subárboles, por esto, se debe navegar por cada uno para llegar a las características del árbol generado a partir del nodo padre. El proceso se realiza por los dos árboles sintácticos en su totalidad de manera recursiva, por lo tanto, al terminar con un nodo, debe proceder al siguiente nodo y así de forma sucesiva hasta completar todos los nodos.

3.4 Diagramas BNF del Lenguaje Universal

Los diagramas BNF del lenguaje universal permiten entender mejor los componentes de su sintaxis, además brindan formalidad a su sintaxis. En esta sección se presentan los principales elementos, que corresponden a las instrucciones de paquete, métodos, y definición de clases.

3.4.1 Definición de un paquete e importación de archivos

En la figura 3.24 se muestra el diagrama con el cual se puede modelar la instrucción que indica el paquete de una clase, la instrucción está conformada por la palabra reservada "gpackage", mientras que en la figura 3.25 muestra la definición de un identificador. Por otro lado, en la figura 3.26 se muestra los componentes que tienen una importación de un archivo o biblioteca.

Tanto el "identifierName" de la figura 3.26 como el "nameSpace" de la figura 3.24 son cadenas de caracteres que se pueden modelar con el diagrama BNF mostrado en la figura 3.25. Los no terminales llamados "nameString", "directClassDerived", "directImplemens" y "typeName" poseen este mismo comportamiento, por lo que se tratan como cadenas.

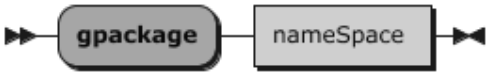


Figura 3.24: Diagrama BNF de la definición de un paquete.

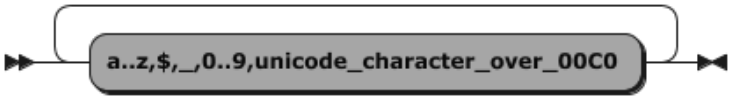


Figura 3.25: Diagrama BNF del identificador general.

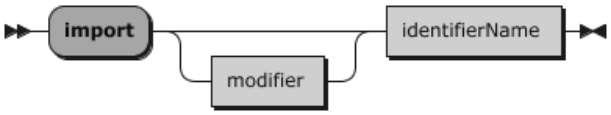


Figura 3.26: Diagrama BNF de la importación de una biblioteca o archivo.

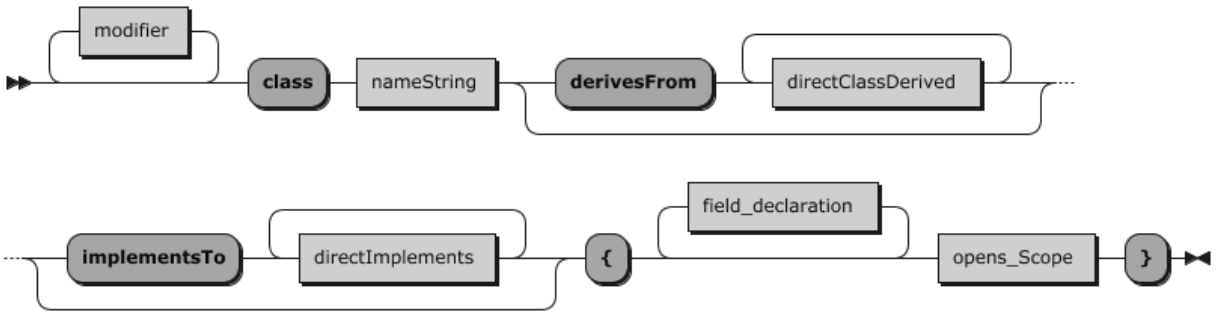


Figura 3.27: Diagrama BNF de la definición de una clase.

3.4.2 Definición de una clase

En la figura 3.27 se muestran los componentes que posee una instrucción para crear una clase. Nótese que la herencia está implementada para modelarla como herencia múltiple, aunque Java carezca de esta característica. De igual manera sucede con las implementaciones, las cuales pueden ser 0, 1 o más de una interfaz. El no terminal llamado "modifier", se encarga de modelar todos los modificadores como se muestra en la figura 3.28. Cabe resaltar que conforme se añadan lenguajes al marco de trabajo se escalará la gramática implementada.

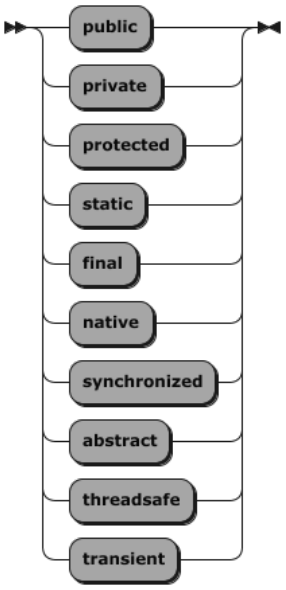


Figura 3.28: Diagrama BNF de modifier.

3.4.3 Definición de método

La instrucción para la creación o definición de un método se muestra en figura 3.29, en la cual se puede observar todos los componentes que conforman la estructura de la instrucción. Un detalle que cabe resaltar es que el método puede tener 0, 1 o más de un parámetro, así como la posibilidad de poseer instrucciones dentro de él (*body*) o no.

La expansión del nodo "formalParameter" se muestra en la figura 3.30, donde se puede observar que está compuesto por el tipo del parámetro y su nombre, como se puede apreciar está la posibilidad de que sea un vector.

El código que se utilizó para generar los diagramas en notación BNF se puede observar en la sección de los apéndices, además, el *framework* utilizado para generar los diagramas fue *Railroad Diagram Generator*.

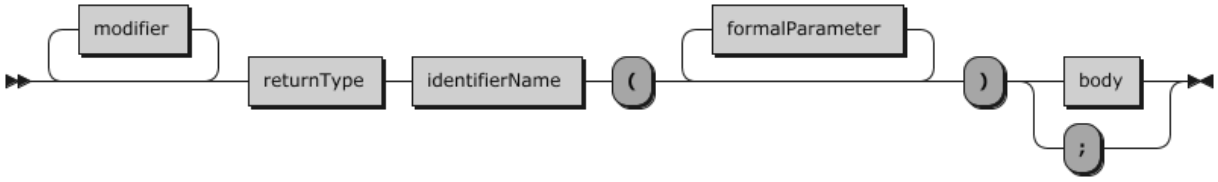


Figura 3.29: BNF de la definición de un método en una clase.

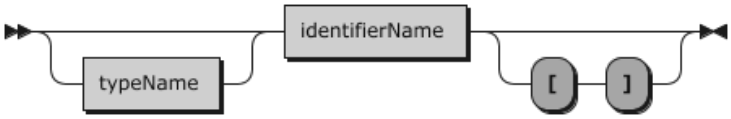


Figura 3.30: BNF de un parámetro formal de un método.

Capítulo 4

Experimentos, resultados y análisis

En este capítulo se presentan los experimentos propuestos, así como los resultados de su ejecución. Hay una sección dedicada a cada experimento.

4.1 Experimento 1: Mapear los SAST al GAST

Este experimento tiene por fin probar la hipótesis que plantea que la creación de un árbol de sintaxis abstracta genérico permite representar todos los elementos sintácticos de diversos lenguajes de programación tales como RPG, C# y Java. A continuación se detalla y describe el objetivo asociado con esta prueba.

Objetivo: Comprobar que todos los elementos sintácticos de diversos lenguajes de programación, como RPG, C# y Java, se pueden representar en una misma estructura de datos, mediante mapeos lineales entre el SAST y el GAST.

Descripción: Se requiere probar la utilidad del GAST mediante el mapeo de cualquier código de entrada, independiente de su extensión y elementos gramaticales. Este experimento permite verificar si todos los elementos sintácticos de Java, RPG y C# pueden ser mapeados y representados de manera correcta y completa en el GAST. Además, brinda información sobre la cantidad de recursos necesarios para llevar a cabo el procesamiento de tales datos. La validación entre el GAST y el SAST se efectúa de manera automática, debido a la cantidad de información que es necesario procesar. Esto implica que se verifica el procesamiento para garantizar la calidad de la representación y que se están procesando todos los datos. A continuación se definen los pasos por seguir para este experimento:

1. Se seleccionaron proyectos cuyo tamaño en el código sea diferente para cada uno de los lenguajes. En el caso de Java se utilizaron tres proyectos de código abierto: Arduino el cual contiene 279 archivos con extensión “java”, Patrones de diseño implementados en Java con un total de 1478 archivos escritos en este

lenguaje de programación y JDT que tiene 8365 archivos. Para C# se utilizaron tres proyectos de código abierto: ANTLR v4 el cual contiene 188 archivos con extensión "cs", ShareX con un total de 742 archivos escritos en este lenguaje de programación y Maui que tiene 6336 archivos. Por último, el lenguaje de programación RPG es comúnmente utilizado en aplicaciones comerciales. Por lo tanto, para realizar el análisis de este lenguaje de programación, se utilizaron 92 archivos proporcionados por un profesor del Tecnológico de Costa Rica para realizar pruebas del analizador de código.

2. Se descargaron y analizaron cada uno de los proyectos anteriores desde un mismo programa.

3. Se generó el GAST para cada uno de los proyectos y con ello se generó el archivo con los datos de salida.

4. Se verificó que todos los elementos del SAST fueran mapeados correctamente en el GAST, utilizando un validador automático, cuyo funcionamiento se basa en la comparación de los AST.

5. Se generó el informe con las diferencias entre el lenguaje genérico y el código original, cuando existían.

6. Se generaron las estadísticas de tiempos de ejecución de los proyectos.

4.1.1 Resultados del Experimento 1

El objetivo de este experimento era comprobar la factibilidad de representar todos los elementos sintácticos de varios lenguajes de programación, como RPG, C# y Java en una sola estructura de datos genérica. Para realizar este experimento se utilizó un computador portátil con un procesador i5 de quinta generación y 8GB de memoria RAM.

A continuación se detallan los resultados de los pasos descritos anteriormente para los tres lenguajes de programación.

Análisis de código fuente escrito en Java

Los proyectos utilizados son el insumo para el analizador y verificador con el fin de comparar los tiempos de ejecución para cumplir con el segundo paso del experimento.

Una vez ingresados los archivos al analizador se generan los GAST de cada uno. Por ejemplo, el documento perteneciente al proyecto Arduino llamado `ConsoleOutputStream.java` tiene el fragmento de código fuente mostrado en la figura 4.1.

```
public void setAttibutes(SimpleAttributeSet attributes) {
    this.attributes = attributes;
}
```

Figura 4.1: Fragmento de código fuente del archivo `ConsoleOutputStream.java`.

Una vez que se genera el GAST para los archivos fuentes del proyecto, se verifica que toda la información está contenida en la estructura. Este proceso se realizó de manera manual. En la figura 4.2 se puede observar la verificación de los argumentos y el tipo de retorno del fragmento de código fuente. El método *setAttibutes* retorna un valor de tipo *void* el cual se detalla en el GAST en la etiqueta "returnType". Dentro de esta etiqueta se encuentra el valor "void" lo que indica que el tipo de retorno se mapeó de manera correcta. Seguidamente se encuentran los modificadores de la función. En este caso se tiene uno que indica que el método es público. Este elemento se coloca en la etiqueta "modifier" la cual tiene el valor "public". Por último, la etiqueta "identifierName" define que la función tiene el nombre "setAttibutes".

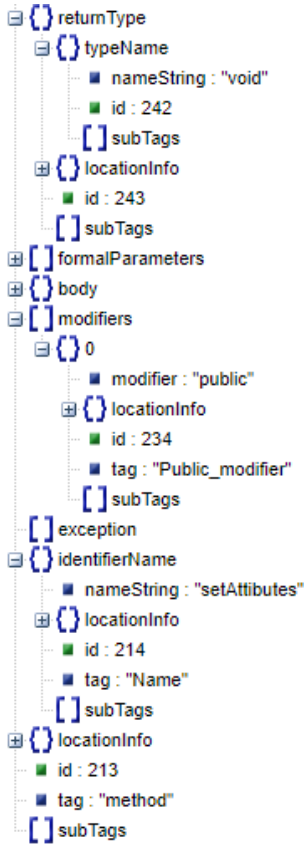


Figura 4.2: Porción del GAST que representa los parámetros y tipo de retorno del código fuente de la figura 4.1.

Continuando con el análisis del fragmento de código fuente, se debe validar que el argumento que recibe la función y la asignación de la variable *attributes*. El otro segmento del GAST

que representa estos componentes se visualiza en la figura 4.3. En esta figura se muestra que la etiqueta "formalParameters" contiene una lista de todos los parámetros de la función. Para este ejemplo, sólo hay un parámetro. Dentro de este elemento, la etiqueta "identifierName" indica que el nombre es "attributes" el cual coincide con el del código fuente mostrado anteriormente. Es importante recalcar que la etiqueta "definitionType" muestra que el parámetro es de tipo "SimpleAttributeSet".

Finalmente, el cuerpo del método corresponde a la asignación. La expresión contiene un operador, operando izquierdo y operando derecho. El operador izquierdo consiste de una variable llamada "attributes".

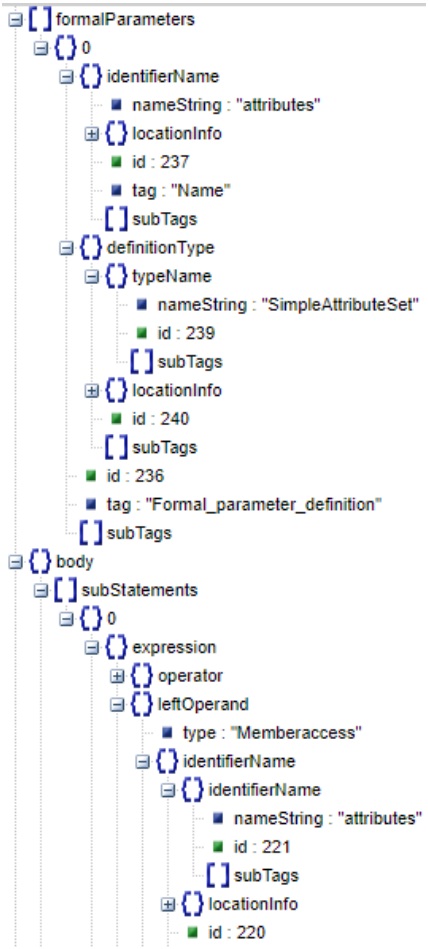


Figura 4.3: Porción del GAST que representa el nombre de la función y los modificadores del código fuente de la figura 4.1.

De esta manera, se verifica manualmente que todos los elementos sintácticos para definir la función *setAttibutes* en el archivo `ConsoleOutputStream.java` se encuentran el GAST.

Este proceso de verificación se hace de manera automática mediante la herramienta de validación. En este caso, la herramienta no encontró diferencias. La figura 4.4 muestra los elementos analizados en el mismo nivel para ambas estructuras. La figura 4.5 contiene el fragmento de código que se está verificando. Como se observa en la figura 4.4, ambas

estructuras detectaron que la variable de tipo “PrintStream” llamada “printStream” es final y privada. De igual forma, la variable de tipo “boolean” llamada “newLinePrinted” es de tipo privada y volátil.

```
GAST private AST private
GAST final AST final
GAST printStream AST printStream
GAST PrintStream AST PrintStream

GAST private AST private
GAST volatile AST volatile
GAST newLinePrinted AST newLinePrinted
GAST boolean AST boolean
```

Figura 4.4: Diferencias encontradas entre el SAST y GAST del código fuente del archivo `ConsoleOutputStream.java`.

```
private final PrintStream printStream;
private volatile boolean newLinePrinted;
```

Figura 4.5: Fragmento de código fuente del archivo `ConsoleOutputStream.java`.

La parte final del experimento para el lenguaje Java consiste en comparar los tiempos de ejecución de los proyectos descritos anteriormente. Para ello, la tabla 4.1 muestra los tiempos de ejecución para los tres proyectos elegidos para el experimento. En esta tabla se incluye el tiempo de ejecución con verificación y sin verificación.

Nombre proyecto	Tiempo con verificador (s)	Tiempo sin verificador (s)
Arduino	15.130469	9.223224
Patrones de diseño	47.121012	19.832072
JDT	646.253487	97.544804

Tabla 4.1: Tiempos de ejecución del análisis de código fuente escrito en Java.

Análisis de código fuente escrito en C#

Los proyectos mencionados anteriormente sirvieron de insumo para el analizador y verificador que comparó los tiempos de ejecución para cumplir con el segundo paso del experimento.

Una vez ingresados los archivos al analizador se generan los GAST de cada uno. Por ejemplo, el documento perteneciente al proyecto ANTLR llamado `Transition.cs` tiene el fragmento de código fuente mostrado en la figura 4.6.

```
public abstract bool Matches(int symbol, int minVocabSymbol,
int maxVocabSymbol);
```

Figura 4.6: Fragmento de código fuente del archivo `Transition.cs`.

Una vez que se genera el GAST para los archivos fuentes del proyecto, se verifica que toda la información está contenida en la estructura. Realizando este proceso de manera manual, se puede observar en la figura 4.7 la verificación de los argumentos y el tipo de retorno del fragmento de código fuente. El método *Matches* retorna un valor de tipo booleano, el cual se detalla en el GAST dentro de la etiqueta "returnType". Dentro de esta etiqueta se encuentra el valor "bool" lo que indica que el tipo de retorno se mapeó de manera correcta. Seguidamente se encuentran los parámetros de la función, los cuales consisten de tres variables de tipo entero. Estos elementos se colocan en la etiqueta "formalParameters". El primer parámetro, mediante las etiquetas "identifierName" y "definitionType" definen que este elemento lleva como nombre "symbol" y es de tipo "int" respectivamente. Los otros dos parámetros se muestran definidos mediante los elementos 1 y 2 contenidos en "formalParameters". Por cuestiones de espacio, no se expanden las ramas en la figura.

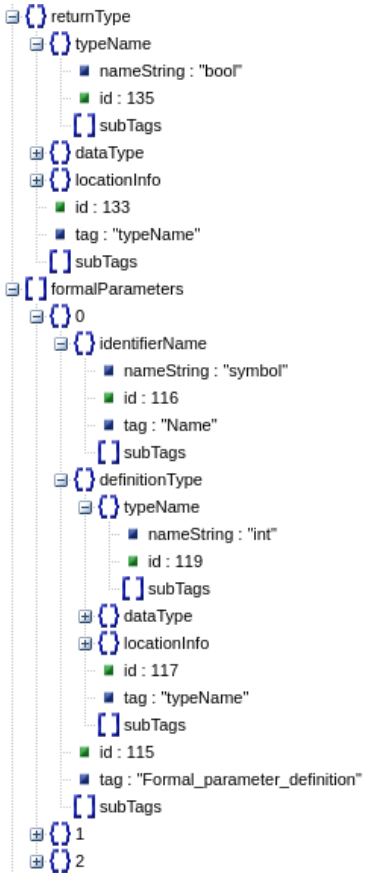


Figura 4.7: Porción del GAST que representa los parámetros y tipo de retorno del código fuente de la figura 4.6.

Continuando con el análisis del fragmento de código fuente, se valida que el nombre y los modificadores son los correctos. El otro segmento del GAST que representa estos componentes se presenta en la figura 4.8. En esta figura se muestra que la etiqueta "modifiers" tiene dos valores: público y abstracto. Estos valores coinciden con los modificadores presentados en el fragmento del código fuente. Por último, la etiqueta "identifierName" indica que el nombre de la función es "Matches".

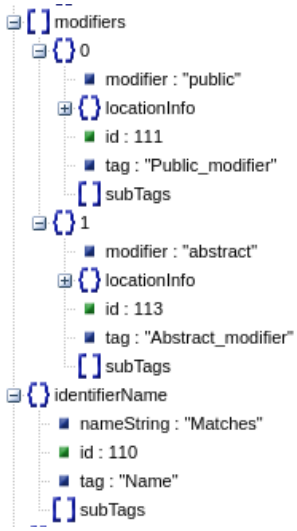


Figura 4.8: Porción del GAST que representa el nombre de la función y los modificadores del código fuente de la figura 4.6.

De esta manera, se verifica manualmente que todos los elementos sintácticos para definir la función *Matches* en el archivo `Transition.cs` se encuentran el GAST.

Este proceso de verificación se realiza de manera automática mediante la herramienta de validación. La figura 4.9 muestra el documento generado a partir de las diferencias encontradas luego de comparar el GAST y SAST. Como se observa, en cada una de las ocurrencias se indica el valor esperado y el obtenido. Además, para este caso se da un detalle adicional que una variable es nula. Estos datos son insumo para el generador del GAST para corregir el proceso de mapeo.

```
Expected: Antlr4, Got: eje : variable name not mapped right
Expected: namespace declaration id, Got: null : namespace declaration id
not mapped
Expected: namespace declarations, Got: null : namespace declarations
not mapped
Expected: , Got: null : warning, null object in programScope

In file: /home/luis/Desktop/ASTin/Transition.cs
```

Figura 4.9: Diferencias encontradas entre el SAST y GAST del código fuente del archivo `Transition.cs`.

Como parte final del experimento para el lenguaje C# se compararon los tiempos de ejecución de los proyectos descritos anteriormente. La tabla 4.2 muestra los tiempos de ejecución para los tres proyectos elegidos para el experimento. En esta tabla se incluye el tiempo de ejecución con verificación y sin verificación.

Nombre proyecto	Tiempo con verificador (s)	Tiempo sin verificador (s)
ANTLR v4	26.404405	14.216345
ShareX	15.059357	8.070384
Maui	2390.123577	1118.999027

Tabla 4.2: Tiempos de ejecución del análisis de código fuente escrito en C#.

Análisis de código fuente escrito en RPG

Los archivos descritos anteriormente sirvieron de insumo para el analizador y verificador que comparó los tiempos de ejecución para cumplir con el segundo paso del experimento.

Una vez ingresados los archivos al analizador se generaron los GAST de cada uno. Por ejemplo, el documento llamado `RTVDDSSRCR.rpgle` tiene el fragmento de código fuente mostrado en la figura 4.10.

```
C                   If        JoinFlag<>'Y'
C                             And HierFlag<>'Y'
C                   If        APBof<>*Blanks
C                   ExSr      GetPF
```

Figura 4.10: Fragmento de código fuente del archivo `RTVDDSSRCR.rpgle`.

Una vez que se generó el GAST para los archivos fuentes del proyecto, se verificó que toda la información está contenida en la estructura. Este proceso se realiza de manera manual y en la figura 4.11 se puede observar la verificación de los argumentos y el tipo de retorno del fragmento de código fuente. La sentencia condicional tiene dos términos que deben ser evaluados para continuar con las siguientes instrucciones. En el lenguaje RPG se utiliza el comando "AND" para separar ambos elementos. La figura muestra que la condición contiene un operador con valor "&&" lo que equivale al "AND". Por lo tanto, el operador izquierdo contiene que la variable "JoinFlag" debe ser diferente a la variable "Y". El operador derecho tiene la información de la segunda condición.

Al continuar con el análisis del fragmento de código fuente, se validaron las instrucciones dentro del condicional el cual consiste en otro condicional. En la figura 4.12 se muestra que la etiqueta "thenBody" que representa todas aquellas instrucciones que deberían ejecutarse si se cumple con la condición. Al haber otro condicional dentro, se repite la estructura previamente explicada. En este caso se debe validar que la variable "APBof"

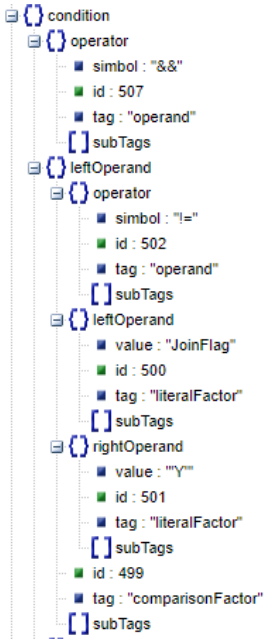


Figura 4.11: Porción del GAST que representa los parámetros y tipo de retorno del código fuente de la figura 4.10.

sea diferente de "*Blanks". Posterior a este condicional, se tiene una instrucción en RPG llamada "Exsr" la cual invoca una subrutina. Al no existir un elemento equivalente en lenguajes como Java o C#, se modela como una llamada a una función.

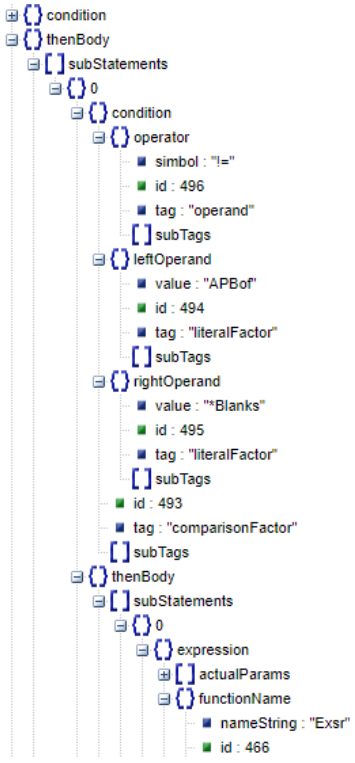


Figura 4.12: Porción del GAST que representa el nombre del cuerpo del condicional del código fuente de la figura 4.10.

De esta manera, se verificó manualmente que todos los elementos sintácticos del fragmento de código fuente en el archivo `RTVDDSSRCR.rpgle` se encuentran el GAST.

Este proceso de verificación se hizo de manera automática mediante la herramienta de validación. Para el caso de este ejemplo, el validador no encontró diferencias. Por lo que la figura 4.13 muestra valores de hojas de la línea de código mostrado en la figura 4.14 correspondiente a las dos estructuras que fueron comparados.

```
GAST: KeyWord AST: KeyWord
GAST: 'PFILE(' AST: 'PFILE('
GAST: '/' AST: '/'
GAST: ')' AST: ')'
GAST: APBof AST: APBof
GAST: *Blanks AST: *Blanks
```

Figura 4.13: Diferencias encontradas entre el SAST y GAST del código fuente del archivo `RTVDDSSRCR.rpgle`.

```
C       Eval    KeyWord='PFILE('+ChkLib(APBol)+'/'+
C                 %Trim(APBof)+')
```

Figura 4.14: Fragmento de código fuente del archivo `RTVDDSSRCR.rpgle`.

Como parte final del experimento para el lenguaje RPG se compararon los tiempos de ejecución de los proyectos descritos anteriormente. La tabla 4.3 muestra los tiempos de ejecución para el conjunto de archivos que fueron proporcionados para el experimento. En esta tabla se incluye el tiempo de ejecución con verificación y sin verificación.

Tiempo con verificador (s)	**Tiempo sin verificador (s)**
78.308761	42.406874

Tabla 4.3: Tiempos de ejecución del análisis de código fuente escrito en RPG.

4.2 Experimento 2: Homogeneizar el análisis

Este experimento consiste en realizar el análisis del código fuente con el fin de probar la hipótesis que plantea si la creación de un lenguaje universal permite homogeneizar el análisis de los elementos y las estructuras gramaticales de los programas escritos en distintos lenguajes de programación, tales como RPG, C# y Java. A continuación se detalla y describe el objetivo asociado con esta prueba

Objetivo: Comprobar que el análisis de los elementos y las estructuras de los programas escritos en diversos lenguajes de programación se puede efectuar de forma homogénea con el uso de un árbol de sintaxis abstracta genérico.

Descripción: Este experimento consiste en tomar dos programas con funcionalidad equivalente escritos en diferentes lenguajes de programación, generar sus SAST y verificar si el mapeo de la totalidad de sus instrucciones producen el mismo GAST. Para efectuar este experimento se implementará una base de datos que almacenará el lenguaje genérico (GAST) de cada programa. Luego, se consulta el GAST de cada uno de estos programas para verificar si se obtiene el mismo resultado. A continuación se definen los pasos que se sigue para efectuar este experimento:

1. Se desarrolló dos programas con funcionalidad equivalente, uno escrito en Java y otro en C#.
2. Se creó el SAST de cada código, uno correspondiente al programa en Java y otro correspondiente al programa en C#.
3. Se analizó y se creó ambos SAST para generar sus GAST y los archivos de salida con el lenguaje genérico.
4. Se verificó la similitud entre los GAST generados, tanto para Java como C#.
5. Se instaló y se configuró el servidor de la base de datos.
6. Se implementó la interfaz para que el analizador de código se conecte al servidor.
7. Se realizaron las consultas sobre los GAST para verificar la equivalencia de los códigos generados.
8. Se proporcionó un mecanismo para introducir la consulta que se desea realizar.
9. Se retornó el resultado de la base de datos en un archivo con el fin presentarlo visualmente utilizando algún *framework* en línea.

4.2.1 Resultados de Experimento 2

El análisis del código correspondiente al GAST se realizó serializando (guardar en disco y base de datos) y deserializando (procesar en memoria) los datos con las clases de cada elemento sintáctico con el fin de procesar los datos en memoria (donde son tratados como objetos). Este experimento utiliza el código representado en la figura

4.15 el cual es implementado en Java y C#, de forma que el mismo programa es escrito de dos maneras diferentes. Este experimento prueba que si se convierte los programas equivalentes a una sintaxis única, se puede analizar la sintaxis genérica en lugar de analizar la gramática específica de cada uno de los lenguajes.

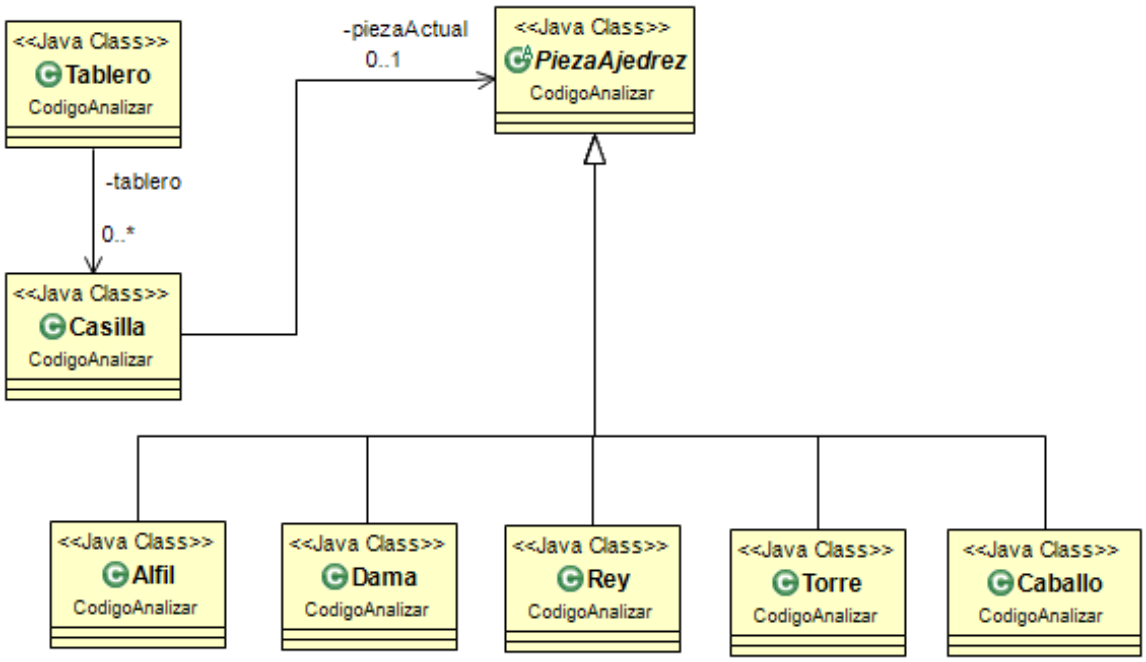
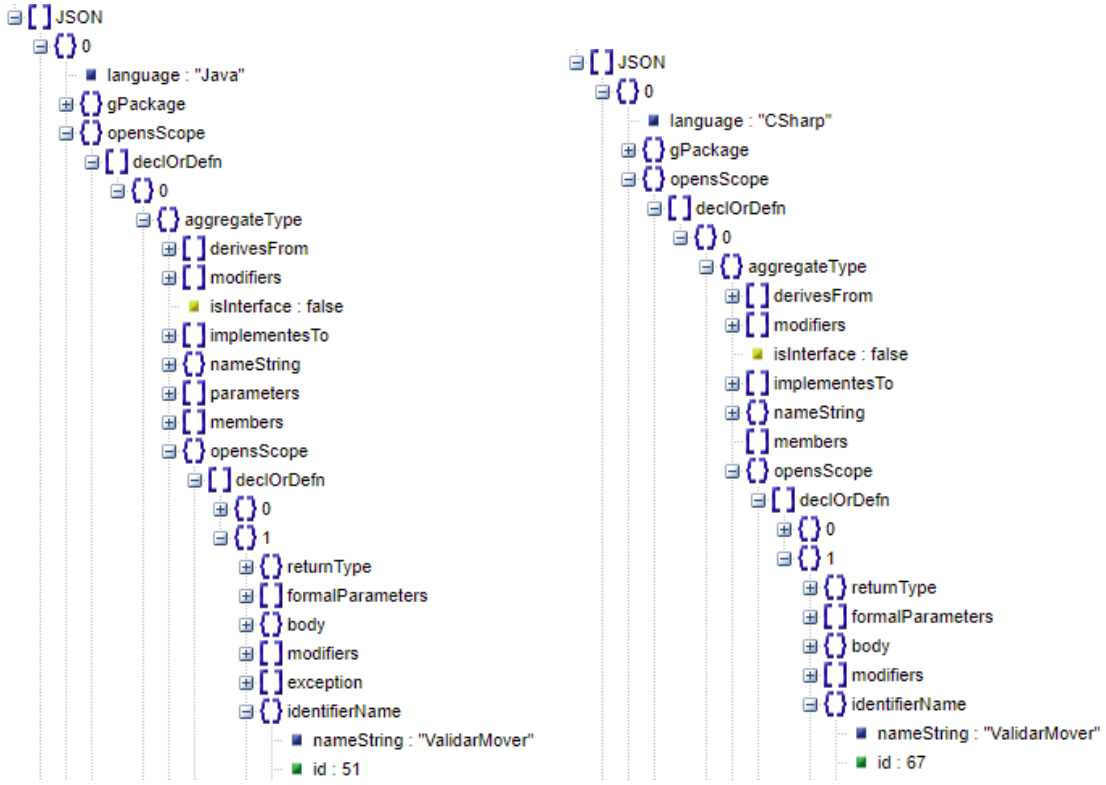


Figura 4.15: Representación UML del código utilizado en el experimento 2

La figura 4.15 muestra la existencia de herencia y asociación entre las clases. También se observan clases abstractas y el uso de arreglos de objetos. Este diseño permite analizar código fuente y obtener detalles como las llamadas a métodos, listado de clases de un proyecto, listado de métodos de una clase, complejidad ciclomática de un método, clones y número de líneas de código. Cabe resaltar que, para efectos de documentación, se presentarán de manera gráfica la obtención de clases, llamadas a métodos y detección de clones del código correspondiente al proyecto JDT. El código fuente y base de datos se encuentran en el siguiente enlace (link).

La fase inicial del experimento 2 consiste en transformar el código escrito en Java y C# en el lenguaje universal con el fin de identificar que el mapeo se está realizando de manera uniforme, independientemente del lenguaje. En la figura 4.16 se muestra el resultado de convertir el código representado en la figura 4.15 (C# y Java) al lenguaje universal.



(a) Código Java (b) Código C#

Figura 4.16: Comparación del código escrito en el lenguaje universal

Las diferencias que se observan entre la figuras 4.16a y 4.16b se atribuyen a las características específicas de los AST de Java y de C# debido a que los elementos sintácticos de ambos lenguajes son distintos. Por ejemplo, el atributo llamado "id", el cual tiene un valor de 51 en Java y un valor de 67 en C#, la razón de esto es porque la cantidad de nodos en el AST de C# es mayor que en el de Java, sin embargo, es únicamente un identificador para cada elemento en el GAST que no afecta en el análisis posterior.

Otra de las diferencias es la etiqueta "language", cuyo valor corresponde al nombre del lenguaje en el que fue escrito el código fuente, en este caso los lenguajes que se están analizando corresponden a Java y C# (CSharp).

El resultado mostrado en la figura 4.16 confirma que el transformador realiza el mismo mapeo de las instrucciones, independientemente del lenguaje en el que esté escrito el código fuente.

Una vez que se corrobora que el mapeo de los elementos sintácticos se hace de manera homogénea, se debe validar si la representación del GAST es apta para realizar análisis de código fuente. Para ello se analizó el lenguaje universal utilizando un módulo de software que detecta clones en los archivos de programación. Este analizador detectó todos los clones que posee el proyecto representado en la figura 4.15. Sin embargo, sólo se muestra un ejemplo de manera gráfica, porque el programa extractor de clones se puede obtener del Repositorio. En el Apéndice A A se muestra la tabla con los clones significativos (sin

contar set y get) encontrados en el proyecto mostrado en la figura 4.15.

La figura 4.17 muestra el código original de uno de los clones detectados con el módulo de software, mientras que la figura 4.18 presenta el mismo clon utilizando la representación del GAST. Cabe destacar que el programa que extrae clones opera sobre la sintaxis del lenguaje universal. En ambas imágenes se aprecia que ambos métodos poseen la misma estructura. Se concluye que lo único en que se diferencian es en el uso de la variable "y1" y "y2" (en la sentencia for), lo que implica que este trozo de código sea caracterizado como un clon de tipo 2.

El módulo de software [59] encargado de extraer los clones tiene la capacidad de detectar clones de tipo uno, dos y tres [62]. El experimento se realiza para encontrar clones del tipo dos.

```
public boolean ValidarLineasRectaHorizontalD(int x1, int y1, int x2, int y2) {
    boolean bandera=true;
    for (int i=x1; i<x2+1;i++) {
        if(tablero[i][y1].isEstaOcupada()) {
            bandera=false;
        }
        i++;
    }
    return bandera;
}

public boolean ValidarLineasRectaVertical(int x1, int y1, int x2, int y2) {
    boolean bandera=true;
    for (int i=y1; i<y2+1;i++) {
        if(tablero[i][y1].isEstaOcupada()) {
            bandera=false;
        }
        i++;
    }
    return bandera;
}
```

Figura 4.17: Clones de código en el lenguaje de Programación Java

Por otro lado, la figura 4.18a muestra el método llamado "ValidarLineasRectaHorizontalD", mientras que en la figura 4.18b, se presenta el método "ValidarLineasRectaVertical". Ambas imágenes muestran la estructura de cada uno de los métodos. Se puede observar que ambos módulos tienen la misma cantidad de parámetros, porque el atributo "fomalParameters" indica los parámetros que posee, en este caso son 4, lo cuales corresponden a otros objetos Json que detallan la información de cada uno de ellos, como el nombre y el tipo.

Otro atributo que se debe destacar es "subStatements", que contiene los objetos representados en formato Json correspondientes a las declaraciones que se encuentran en el cuerpo del método. En este caso, hay tres *statements*, los cuales son la variable *booleana*, el *for* y el *return*. El *for* contiene en su interior dos *statements*, el *if* y el incremento. De igual manera sucede con el *if*, ya que en su interior contiene un *statement* cuya función es la asignación binaria.

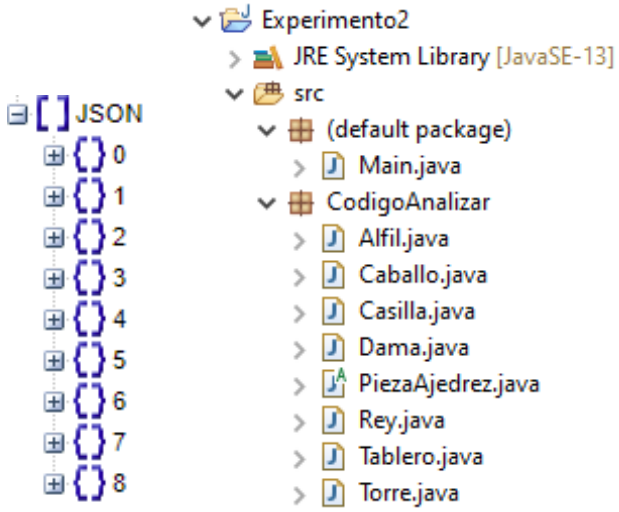
(a) Método ValidadLineasRectaHorizontalD (b) Método ValidadLineasRectaVertical

Figura 4.18: Visualización del clone en el GAST

Cuando el código fuente, escrito en algún lenguaje de programación específico, es transformado al lenguaje universal puede desearse extraer algunos datos para analizar la calidad y mantenibilidad del software. Por esta razón, el lenguaje universal debe soportar consultas de los desarrolladores para extraer información valiosa; por ejemplo, obtener las clases de un proyecto u obtener los métodos de una clase específica. La extracción de estos datos se ejemplifica en las figuras 4.19 y 4.21, donde se utiliza el proyecto mostrado en la figura 4.15 como entrada.

Para la visualización de las consultas se utilizó Neo4j , una base de datos orientada a grafos que permite mostrar gráficamente el resultado de las consultas realizadas sobre la sintaxis del lenguaje en específico. El motivo de utilizar Neo4j es porque permite una mejor interpretación de los resultados de manera visual, que, si se los muestra en texto, se vuelven ilegibles en proyectos de gran magnitud, como lo pueden ser JDT y Hadoop, donde la cantidad de clases que conforman el proyecto están en el orden de los miles. El software que se desarrolló permite la conexión a una base de datos ya establecida o inicializar una base de datos, con el fin de consultar los datos cuando se necesiten.

La figura 4.19a muestra la representación en objeto Json de todas las clases del proyecto, la cual corresponde al lenguaje universal. Por otro lado la figura 4.19b muestra los archivos .java que contiene el proyecto. Nótese que la cantidad de clases coincide en ambas imágenes.



(a) Clases representadas en el GAST

(b) Clases representadas en Java

Figura 4.19: Clases del proyecto de la figura 4.15

Si el lenguaje universal es almacenado en la base de datos, se puede realizar una consulta para obtener la totalidad de las clases dado un proyecto. Para el proyecto mostrado en la figura 4.19, se puede visualizar el resultado de la consulta sobre el lenguaje universal en la figura 4.20, en donde no hay relaciones entre nodos porque la consulta no lo permite.

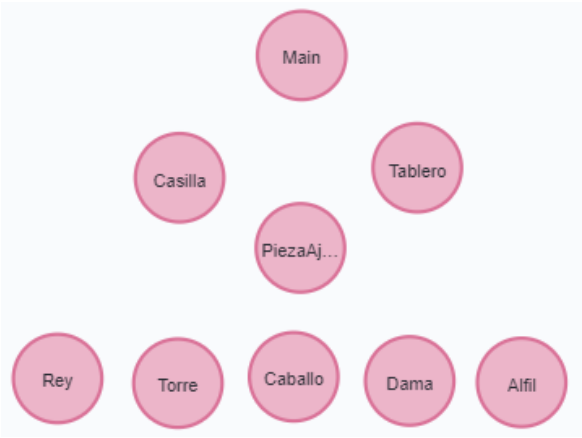
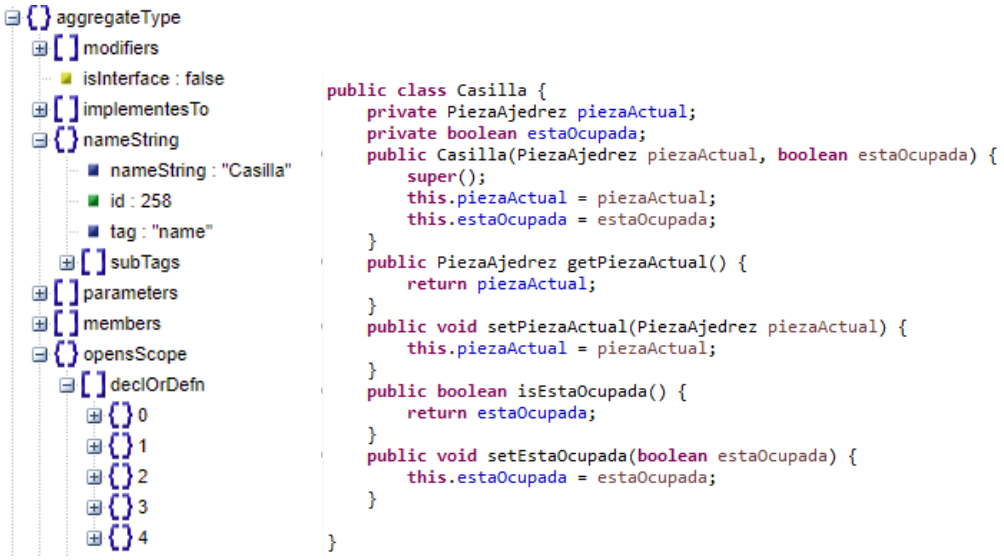


Figura 4.20: Resultado de la consulta sobre las clases de un proyecto en específico

La figura 4.21a muestra los métodos en representación del lenguaje universal contenidos en la clase “Casilla”, mientras que la figura 4.21b muestra el código original de dichos métodos.

Cabe destacar que la variable que contiene los métodos corresponde a "declOrDefn", la cual posee 5 objetos Json, cuyo significado es cada una de las operaciones de la clase.



(a) Métodos en el GAST (b) Métodos en Java

Figura 4.21: Métodos de la clase "Casilla"

En caso de que se requiera consultar directamente la base de datos sobre la sintaxis del lenguaje universal se debe proporcionar la firma de la clase a la cual se desea extraer los métodos, debido a que pueden existir en el proyecto archivos con el mismo nombre, pero pertenecientes a distintos paquetes. En caso de que exista más de una clase con el mismo nombre y la consulta sobre los métodos es ejecutada, la base de datos tomará ambas entidades y extraerá la información. La figura 4.22 muestra el resultado de los métodos pertenecientes a la clase "Casilla". Se puede observar que cada método tiene una relación de pertenencia (*OWNS_METHOD*) con la clase que lo contiene, con el fin de indicar que el método pertenece a una determinada clase.

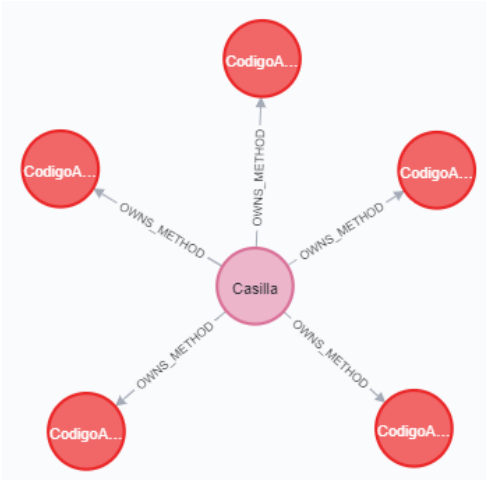


Figura 4.22: Resultado de la consulta sobre las clases de un proyecto en específico

Tanto las métricas de las clases como la de los métodos son utilizadas para las visualizaciones en tareas de mantenimiento del software.

Expuestos los resultados anteriores se evidencia que el lenguaje universal permite extraer métricas para el análisis del software. Sin embargo, es necesario realizar pruebas de estrés con proyectos de gran magnitud y extraer métricas como cantidad de líneas de código por método, llamadas entre métodos y complejidad ciclomática.

El proyecto que se eligió para hacer pruebas de estrés fue JDT, el cual está desarrollado en Java y posee 7544 clases. Para efectos de visualización se analizará uno de los paquetes más grandes de este proyecto. Para efectos de revisión más detallada, tanto el código de la base de datos como el de JDT están dentro del Repositorio.

El almacenamiento del proyecto JDT se realiza en la base de datos Neo4J. Debido a que, es orientada a grafos, permite visualizar las clases, métodos y relaciones de una mejor manera, así como realizar los cálculos de las métricas de forma sencilla. La figura 4.23 muestra la totalidad de las clases que posee el proyecto, en donde cada clase es representada por un punto.

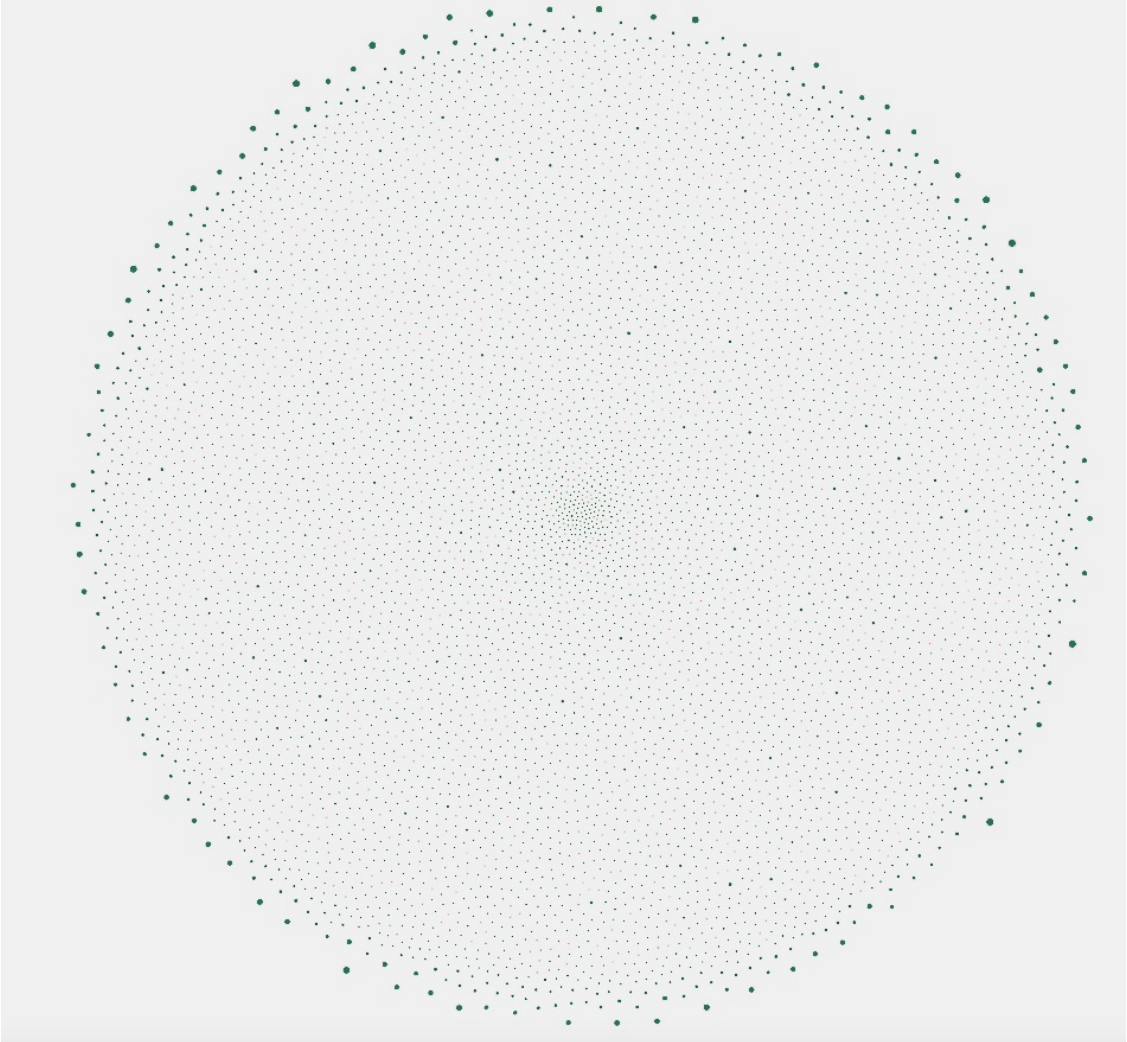


Figura 4.23: Clases del proyecto JDT (*org.eclipse.jdt*)

La figura 4.24 muestra todas las clases pertenecientes al paquete *org.eclipse.jdt* del proyecto JDT. Cada clase es representada por un círculo rojo, mientras que los métodos son los de color café claro, se puede notar que es una gran cantidad de clases que fueron trasformadas al lenguaje universal con el fin de realizar la visualización y análisis del código fuente.

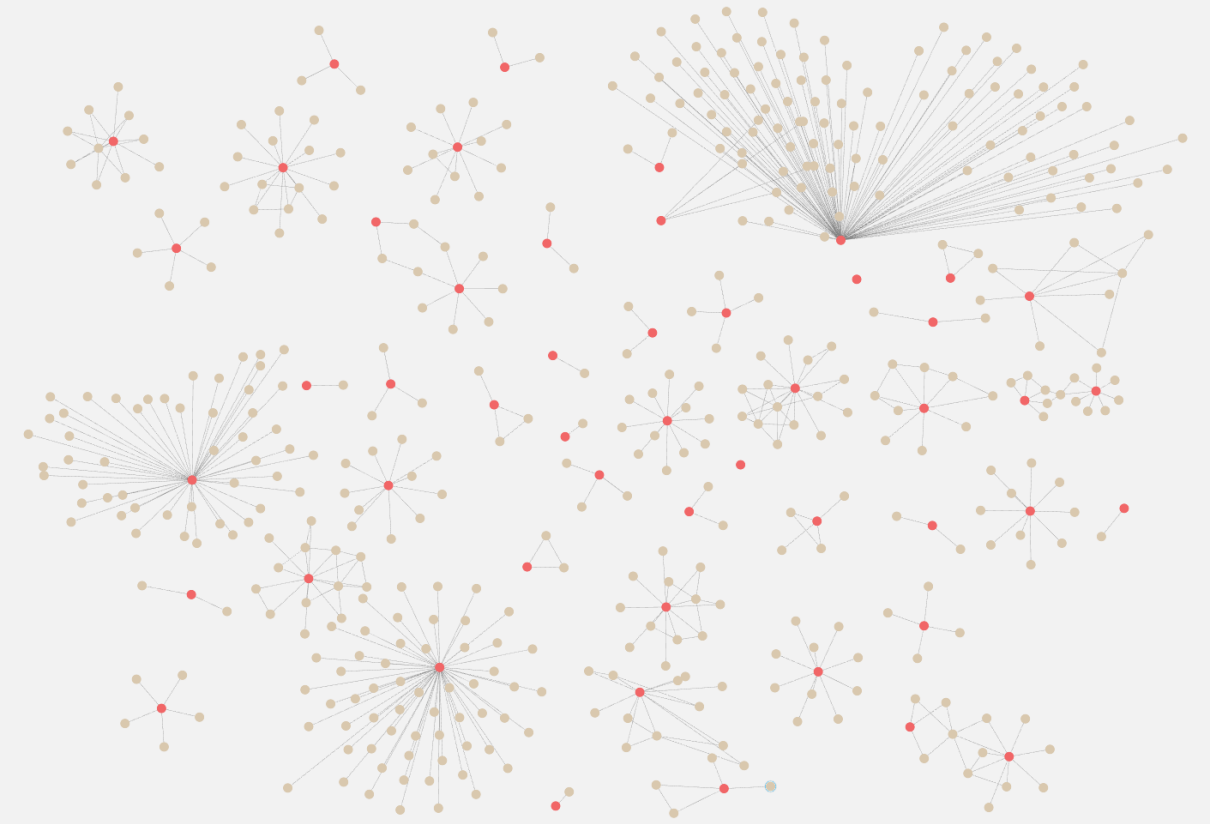


Figura 4.24: Clases y métodos del paquete *org.eclpse.jdt* del proyecto JDT

Debido a la cantidad de clases y métodos que contiene el paquete de JDT, el análisis se enfoca en un trozo de código (dos clases con sus respectivos métodos) para mostrar el estudio que se realiza sobre el lenguaje universal.

La métrica de obtener todos los métodos de todas las clases y crear las asociaciones entre cada uno de los métodos se puede observar en la figura 4.24. Se puede notar las llamadas entre métodos y con ello calcular las cadenas de nodos necesarias para obtener el acoplamiento indirecto [44] a lo largo del grafo. En la figura 4.25 se observa dos clases específicas, *JavaMethodFiltersTable* y *TypeFilterAdapter*, las cuales contienen métodos que realizan llamadas a otros métodos, lo que permite establecer dos tipos de relaciones. La primera es "CALLS", la cual indica que un método llama a otro dentro de su código. La segunda corresponde a "OWNS_METHODS" que indica la pertenencia del método a la clase.

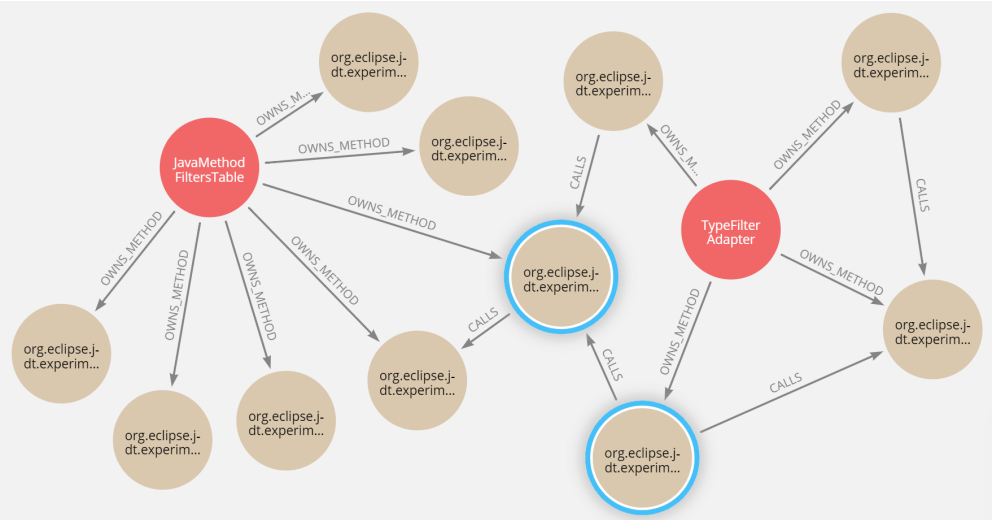


Figura 4.25: Dos clases específicas del proyecto JDT, con sus respectivos métodos y relaciones

Por otro lado, el análisis sobre el lenguaje universal permite obtener métricas analíticas y cuantitativas del software, por ejemplo, líneas de códico (loc), cantidad de métodos llamados (calls), cantidad de métodos que llaman a un método (calledby) y complejidad ciclomática (cyc). Las figuras 4.26a y 4.26b se muestran las métricas de los métodos con contorno azul mostrados en la figura 4.25. El trabajo realizado por Navas-Sú et al. [44] muestra la manera en que fueron calculados los valores de cada métrica.

Las métricas citadas son obtenidas analizando el lenguaje universal y utilizando la base de datos Neo4j para realizar las consultas sobre los grafos generados en el procesamiento sintáctico y semántico. Esto implica que la gramática del lenguaje universal permite tomar las métricas del software de manera equivalente a las que se extraen de la gramática del código original.

calledby	calls	cyc	fcyc	fhal
0	2	2	7	1180.967791

floc	fnom	hal	id	iscollapsed
22	4	33	Ajedrez208	0

ismethod	isrecursive	isscc	loc
1	0	0	2

method: doubleClicked(ListDialogField)

name: org.eclipse.jdt.experimental.ui.preferences.JavaMethodFiltersTable.TypeFilter

nom	rcyc	rhal	rloc	rnom
1	2	33	2	1

sccid: 0

(a) Estadísticas del método doubleClicked

Method

calledby	calls	cyc	fcyc	fhal
2	1	6	6	965.099408

floc	fnom	hal	id	iscollapsed
19	2	846.493217	Ajedrez215	0

ismethod	isrecursive	isscc	loc
1	0	0	18

method: doButtonPressed(int)

name: org.eclipse.jdt.experimental.ui.preferences.JavaMethodFiltersTable.doButt

nom	rcyc	rhal	rloc	rnom
1	7	1001.350425	21	3

(b) Estadísticas del método doButtonPressed

Figura 4.26: Estadísticas de los métodos resaltados en la figura 4.25

Capítulo 5

Conclusiones

5.1 Conclusiones

El GAST diseñado y desarrollado en este trabajo se basa en el estándar MOF 2.0, que no fue planteado para modelar lenguajes de programación . Hemos demostrado que el GAST permite la representación de los AST de lenguajes específicos tales como RPG, C# y Java. La manera en que está estructurado el GAST permite agregar sintaxis de lenguajes nuevos.

La transformación del AST específico de un lenguaje al GAST depende directamente de la estructura de dichos AST, lo que implica que el proceso del mapeo cambia con respecto del lenguaje específico. La biblioteca MapStruct proporciona un marco de trabajo que automatiza este proceso, lo cual permite realizar el mapeo de manera eficiente y sencilla de implementar.

El método implementado en este trabajo utiliza reflexión, lo que permitió extraer los valores tanto del AST para Java, C# y RPG, así como el GAST. La forma en que esto fue realizado posibilita la incorporación de nuevos métodos al diccionario con el fin de comparar nodos homólogos y evitar modificar el código fuente del comparador, sino ingresar parámetros para los análisis. Esto permite trabajar con objetos generales sin tener que realizar validación de tipos de datos, ni ejecutar transformaciones entre tipos de datos para recuperar valores.

El método de verificación automático asegura que cada archivo de un proyectos tiene sus elementos sintácticos almacenados en el GAST. Como se evidenció en los resultados, esta validación es una tarea costosa en tiempo, pero representa la ventaja de que otros módulos que utilicen el GAST no deban verificar la completitud de los elementos sintácticos.

La representación del lenguaje universal en formato JSON posibilitó una verificación transparente con respecto del AST de cada lenguaje (SAST), y por ende del GAST, ya que JSON posee una estructura jerárquica similar a los AST. Además, la ventaja de utilizar este formato es que brinda la posibilidad de almacenar el lenguaje universal tanto en una

base de datos orientada a documentos como en una base datos relacional.

La representación de la gramática del lenguaje universal en diagramas BNF permitió aprovechar el formalismo y validar a sintaxis de manera visual cuando se analiza cualquier lenguaje de programación.

El trabajo realizado permite crear un árbol de sintaxis abstracto genérico con todos los elementos sintácticos de los lenguajes de programación RPG, C# y Java. De esta manera se valida la primera hipótesis con los resultados mostrados en este documento. Esta hipótesis plantea que mediante una misma estructura se pueden representar los elementos sintácticos de un lenguaje de programación. Esto genera escalabilidad debido a que se puede agregar tantos lenguajes como se deseen.

El proceso de generar el árbol de sintaxis abstracto genérico se realiza mediante el mismo marco de trabajo. Esto valida la segunda hipótesis, al generalizar sobre los casos de prueba mostrados en las secciones anteriores. De esta manera, el marco de trabajo analiza diversos lenguajes de programación sin realizar modificaciones en el código fuente.

El proyecto genera un impacto positivo en la industria de la informática, debido a que permite analizar los programas independientemente del lenguaje en el que fueron escritos, brindando apoyo en el control de calidad y mantenimiento del software en las compañías. También facilita a los desarrolladores el entendimiento de los programas porque no es necesario que ellos conozcan con profundidad el lenguaje de programación en que fueron realizados los programas.

El GAST genera beneficios directos en las finanzas de proyectos informáticos, debido a que se disminuye el tiempo que los desarrolladores deben de invertir para entender el código hecho por otros y para brindar el mantenimiento adecuado al software, lo que implica que dicho tiempo se tomaría para otras actividades creativas, aumentando la productividad de la empresa.

5.2 Trabajo futuro

El trabajo futuro radica en la paralelización, en el nivel de hardware, del procesamiento de los datos, con el fin de abordar proyectos de programación de mayor tamaño y sin limitantes de recursos físicos como la memoria RAM. Uno de los diseños que se ha propuesto es un clúster compuesto por al menos cuatro computadoras, así como paralelizar, en el nivel de hilo, en cada nodo perteneciente a dicho clúster. Este diseño permitiría procesar código de mayor tamaño en menor cantidad de tiempo, lo que significará un menor tiempo de respuesta para el usuario.

También se desea incorporar nuevas gramáticas de lenguajes específicos para brindar una mayor versatilidad en el análisis y cálculo de métricas de un código de programación.

El proyecto del árbol de sintaxis abstracto genérico permite agregar tantos lenguajes de programación como sea necesario analizar. Esta estructura genérica está en la facultad

de soportar lenguajes de programación de diferentes paradigmas. Por esta razón, se pueden obtener los elementos sintácticos de diversos lenguajes de programación. Con esta estructura se puede analizar el código sin importar la sintaxis específica de cada lenguaje.

La estructura general permite realizar un proceso de generación de código fuente a partir del árbol de sintaxis abstracto genérico. Actualmente, se planea generar código fuente en los lenguajes de programación Java, C# y Python.

Por otra parte, el proyecto del GAST puede ser utilizado para analizar código correspondiente a algún *malware*, debido a que si se obtiene el código ensamblador del programa, este sería analizado automáticamente por el marco de trabajo. Esto implica que se habilita la detección temprana de riesgos que un malware genera en un sistema computacional.

También pueden desarrollarse investigaciones relacionadas con detectar *code smells*, las cuales pueden utilizar como base el GAST para apoyar el análisis de código de programación independientemente del lenguaje en el que esté escrito el software. Esto ayudaría a mejorar la calidad de los programas que se desarrollan tanto en industria como en la academia.

Bibliografía

[1] Fawzi Albalooshi and Amjad Mahmood. A comparative study on the effect of multiple inheritance mechanism in Java, C++, and Python on complexity and reusability of code. *International Journal of Advanced Computer Science and Applications*, 8(6):109–116, 2017.

[2] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models for any-code generation. *ICLR*, 2019.

[3] Kijin An, Na Meng, and Eli Tilevich. Automatic inference of Java-to-Swift translation rules for porting mobile applications. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, pages 180–190, 2018.

[4] Paul Anderson. The use and limitations of static-analysis tools to improve software quality. *CrossTalk: The Journal of Defense Software Engineering*, 21(6):18–21, 2008.

[5] Gustav Andersson. Translation of CAN bus XML messages to C source code. 2020.

[6] F Andrés Bastidas and María Pérez. A systematic review on transpiler usage for transaction-oriented applications. In *2018 IEEE Third Ecuador Technical Chapters Meeting (ETCM)*, pages 1–6. IEEE, 2018.

[7] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377. IEEE, 1998.

[8] Keith H Bennett and Václav T Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 73–87, 2000.

[9] John E Bradbury. JASMINT: Language to user-friendly ast with emphasis on translation. *https://bit.ly/3gLrHLe*, 2018.

[10] G Campbell and Patroklos P Papapetrou. *SonarQube in action*. Manning Publications Co., 2013.

[11] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. *arXiv preprint arXiv:1802.03691*, 2018.

[12] Antony Courtney. Frappé: Functional reactive programming in Java. In *International Symposium on Practical Aspects of Declarative Languages*, pages 29–44. Springer, 2001.

[13] Roger F Crew et al. Astlog: A language for examining abstract syntax trees. In *DSL*, volume 97, pages 18–18, 1997.

[14] Gaëtan Deltombe, Olivier Le Goaer, and Franck Barbier. Bridging KDM and ASTM for model-driven software modernization. In *SEKE*, pages 517–524, 2012.

[15] Serge Demeyer, Stéphane Ducasse, and Sander Tichelaar. Why FAMIX and not UML. In *Proceedings of UML'99*, volume 1723, 1999.

[16] Serge Demeyer, Sander Tichelaar, and Patrick Steyaert. Famix 2.0. *Date of visit: OCtober*, 2021.

[17] Cambridge Dictionary. Metalanguage — meaning in the Cambridge English Dictionary [online, visitado el 2019-03-26]. URL https://dictionary.cambridge.org/dictionary/english/metalanguage.

[18] Teduh Dirgahayu, Sheila Nurul Huda, Zainudin Zukhri, and Chanifah Indah Ratnasari. Automatic translation from pseudocode to source code: A conceptual-metamodel approach. In *2017 IEEE International Conference on Cybernetics and Computational Intelligence (CyberneticsCom)*, pages 122–128. IEEE, 2017.

[19] Josh Dreyfuss. The ultimate JSON library: JSON. simple vs GSON vs Jackson vs JSONP. *Takipi Blog*, 2015.

[20] Mehdi Drissi, Olivia Watkins, Aditya Khant, Vivaswat Ojha, Pedro Sandoval, Rakia Segev, Eric Weiner, and Robert Keller. Program language translation using a grammar-driven tree-to-tree model. *arXiv preprint arXiv:1807.01784*, 2018.

[21] Stéphane Ducasse, Nicolas Anquetil, Muhammad Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, and Tudor Girba. MSE and FAMIX 3.0: an interexchange format and source code model family. 2011.

[22] Systems Engineering, Standards Committee, C Sesc Committee, and Computer Society. *IEEE 1012 Standard for System and Software Verification and Validation*, volume 2012. 2012.

[23] Marcelo G Estayno, Gladys N Dapozo, Liliana Raquel Cuenca Pletsch, and Cristina L Greiner. Modelos y métricas para evaluar calidad de software. In *XI Workshop de Investigadores en Ciencias de la Computación*, 2009.

[24] Adrian Farrel. Routing Backus-Naur Form (RBNF): A syntax used to form encoding rules in various routing protocol specifications. *https://www.hjp.at/doc/rfc/rfc5511.html*, 2009.

[25] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[26] Jerry Gao, H-SJ Tsao, and Ye Wu. *Testing and quality assurance for component-based software*. Artech House Computing Library, 2003.

[27] Tracy Gardner, Catherine Griffin, Jana Koehler, and Rainer Hauser. A review of OMG MOF 2.0 Query/Views/Transformations submissions and recommendations towards the final standard. In *MetaModelling for MDA Workshop*, volume 13, page 41, 2003.

[28] Object Management Group. Meta Object Facility (MOF) specification. https://shortest.link/LYs, 2002.

[29] Gson. Gson guide, sep 2018. URL https://sites.google.com/site/gson/gson-user-guide.

[30] Debra Hiom. What is metadata? *Online Information Services in the Social Sciences*, pages 177–184, 2004, https://shortest.link/NwP.

[31] Kaifeng Huang, Bihuan Chen, Xin Peng, Daihong Zhou, Ying Wang, Yang Liu, and Wenyun Zhao. Cldiff: generating concise linked code differences. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 679–690. ACM, 2018.

[32] Emanuel Irrazábal and Javier Garzás. Análisis de métricas básicas y herramientas de código libre para medir la mantenibilidad. *REICIS. Revista Española de Innovación, Calidad e Ingeniería del Software*, 6(3), 2010.

[33] Capers Jones. The economics of software maintenance in the twenty first century. https://bit.ly/3BEKLmz, 2006.

[34] Pasi Kellokoski. Round-trip engineering. Master's thesis, University of Tampere, 2000.

[35] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *2006 13th Working Conference on Reverse Engineering*, pages 253–262. IEEE, 2006.

[36] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*, 2020.

[37] Mapstruct. Mapstruct Java Bean, jul 2018. URL https://maven.apache.org/.

[38] Apache Maven. Apache Maven Project, sep 2018. URL https://maven.apache.org/.

[39] Daniel D. McCracken and Edwin D. Reilly. Backus-Naur Form (BNF). In *Encyclopedia of Computer Science*, pages 129–131. John Wiley and Sons Ltd., Chichester, UK. URL http://dl.acm.org/citation.cfm?id=1074100.1074155.

[40] Manishankar Mondal, Chanchal K Roy, and Kevin A Schneider. A comparative study on the bug-proneness of different types of code clones. In *2015 IEEE International conference on software maintenance and evolution (ICSME)*, pages 91–100. IEEE, 2015.

[41] Lei Mu. gLua: A modern Lua transpiler in Scheme. 2019.

[42] Zaigham Mushtaq, Ghulam Rasool, and Balawal Shehzad. Multilingual source code analysis: A systematic literature review. *IEEE Access*, 5:11307–11336, 2017.

[43] Prakash M Nadkarni. What is metadata? In *Metadata-driven Software Systems in Biomedicine*, pages 1–16. Springer, 2011.

[44] José Navas-Sú and Antonio González-Torres. A method to extract indirect coupling and measure its complexity. In *2018 International Conference on Information Systems and Computer Science (INCISCOS)*, pages 186–192. IEEE, 2018.

[45] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. Migrating code with statistical machine translation. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 544–547, 2014.

[46] William L Oberkampf, Timothy G Trucano, and Charles Hirsch. Verification, validation, and predictive capability in computational engineering and physics. *Applied Mechanics Reviews*, 57(5):345–384, 2004.

[47] Object Management Group (OMG). Architecture-driven Modernization: Abstract Syntax Tree Metamodel (ASTM). (January), 2011.

[48] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura. Learning to generate pseudo-code from source code using statistical machine translation. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 574–584, 2015.

[49] Steven A O'Hara. Improving programming language transformation. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, pages 129–135. The Steering Committee of The World Congress in Computer Science, Computer, 2018.

[50] OMG. OMG Meta Object Facility (MOF) Core Specification - Version 2.4.1. (November), 2013. URL http://www.omg.org/spec/MOF/2.4.1/.

[51] OMG. About the Meta Object Facility Specification version 2.5.1 [online]. 2016 [visitado el 4 de noviembre de 2021]. URL https://www.omg.org/spec/MOF.

[52] QVT Omg. Meta Object Facility (MOF) 2.0 Query/View/Transformation specification. *Final Adopted Specification (November 2005)*, 2008.

[53] Mike Papadakis, Marcio Delamaro, and Yves Le Traon. Mitigating the effects of equivalent mutants with mutant classification strategies. *Science of Computer Programming*, 95:298–319, 2014.

[54] Benjamin C Pierce. *Types and programming languages*. MIT Press, 2002.

[55] Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535*, 2017.

[56] Stan Rifkin. Guest Editor's Introduction: Software Measurement. *IEEE Software*, 26(3):70–70, 2009.

[57] Paola Andrea Roa, Claribel Morales, and Patricia Gutiérrez. Norma ISO/IEC 25000. *Tecnología Investigación y Academia*, 3(2):27–33, 2015.

[58] Nishanth Shetty, Nikhil Saldanha, and MN Thippeswamy. Crust: A C/C++ to Rust transpiler using a "nano-parser methodology" to avoid C/C++ safety issues in legacy code. In *Emerging Research in Computing, Information, Communication and Applications*, pages 241–250. Springer, 2019.

[59] Marcelo Sánchez Solano. Clones detecting.

[60] Jeff Tian. *Software quality engineering: testing, quality assurance, and quantifiable improvement*. John Wiley & Sons, 2005.

[61] Eelco Visser. Meta-programming with concrete object syntax. In *International Conference on Generative Programming and Component Engineering*, pages 299–315. Springer, 2002.

[62] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 261–271. IEEE, 2020.

[63] Wuu Yang. Identifying syntactic differences between two programs. *Software: Practice and Experience*, 21(7):739–755, 1991.

[64] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*, 2017.

Apéndice A

Clones totales del Experimento 2

La tabla A.1 muestra los clones significativos (sin incluir set y get) encontrados por el detector de clones, utilizando como entrada el GAST (C# y Java). Cabe resaltar que el archivo donde se almacena esta información se encuentra en el Repositorio. Este archivo es auto generado por el detector de código y contiene todos los clones indicando los métodos y los archivos a los cuales pertenecen.

GAST Java	GAST C#	Método 1	Método 2
Si	Si	Inicializar	ValidarLineasRectaHorizontalD
Si	Si	Inicializar	ValidarLineasRectaVertical
Si	Si	ValidarDiagonalDerechaAbajo	ValidarDiagonalIzquierdaAbajo
Si	Si	ValidarDiagonalDerechaAbajo	ValidarLineasRectaHorizontalD
Si	Si	ValidarDiagonalDerechaAbajo	ValidarLineasRectaVertical
Si	Si	ValidarDiagonalIzquierdaAbajo	ValidarLineasRectaHorizontalD
Si	Si	ValidarDiagonalIzquierdaAbajo	ValidarLineasRectaVertical
Si	Si	ValidarLineasRectaHorizontalD	ValidarLineasRectaVertical
Si	Si	ValidarL	ValidarR
Si	Si	ValidarMover	EstablecerPieza

Tabla A.1: Clones encontrados en el proyecto de la figura 4.15, utilizando el GAST tanto de C# como de Java

Apéndice B

Código correspondiente a los principales diagramas BNF del Lenguaje Universal

```
package_name
      ::=
      identifier
      | ( package_name  identifier )
import
      ::=
      "import" (modifier)? identifierName

package_statement
      ::=
      "gpackage" nameSpace

identifier
      ::= "a..z,$,_,0..9,unicode_character_over_00C0"
      ↪  ("a..z,$,_,0..9,unicode_character_over_00C0")*

class_declaration
      ::=
      (modifier* )? "class" nameString
      ( "derivesFrom" directClassDerived (directClassDerived)* )?
      ( "implementsTo" directImplements ( directImplements )* )?
      "{" ( field_declaration (field_declaration)* )? opens_Scope"}"

method_declaration
      ::=
```

```
      ( modifier* )? returnType identifierName
      "(" ( (formalParameter*)? ) ")"
      ( body | ";" )

formalParameter
      ::=
      typeName? identifierName ( "[" "]" )?

modifier
      ::=
      "public"
      | "private"
      | "protected"
      | "static"
      | "final"
      | "native"
      | "synchronized"
      | "abstract"
      | "threadsafe"
      | "transient"
```