**Herramientas**

**para el análisis de DART3**

**Luimarco Daniel Santiago Carrascal Diaz**

**Asesor**

**Nicolas Cardozo Álvarez**

**Facultad de Ingeniería, Departamento de Sistemas y Computación,**

**UNIVERSIDAD DE LOS ANDES**

**04 de junio de 2024**

**Bogotá D.C. Colombia.**

**Tabla de contenido**

[Resumen 3](#_Toc170248474)

[1. Introducción 4](#_Toc170248475)

[1.1. Descripción del Problema 4](#_Toc170248476)

[1.2. Descripción de los Componentes del Analizador 4](#_Toc170248477)

[2. Antecedentes 7](#_Toc170248478)

[3. Identificación del problema y de su importancia 7](#_Toc170248479)

[4. Implementación de la actualización de la gramática de DART 9](#_Toc170248480)

[4.1. Recolección de Información 9](#_Toc170248481)

[4.2. Alternativas de diseño 10](#_Toc170248482)

**[4.2.1. Diseño Basado en ANTLR](#_Toc170248483)**  [10](#_Toc170248483)

**[4.2.2. Diseño Basado en JavaCC](#_Toc170248484)**  [11](#_Toc170248484)

**[4.2.3. Diseño Basado en PEG.js](#_Toc170248485)**  [12](#_Toc170248485)

[4.3. Implementación 13](#_Toc170248486)

[4.4. Diseño e Implementación de la Gramática 14](#_Toc170248487)

[5. Validación 15](#_Toc170248488)

[6. Resultados 19](#_Toc170248489)

[7. Discusión 42](#_Toc170248490)

[8. Conclusiones 43](#_Toc170248491)

[9. Trabajo futuro 44](#_Toc170248492)

[Referencias: 45](#_Toc170248493)

[Apéndices 45](#_Toc170248494)

## **Resumen**

La actualización de la semántica de DART de la especificación 2.0 a la versión 3.03 representa un paso significativo hacia la modernización y optimización del lenguaje. Las mejoras en la consistencia, rendimiento y manejo de programación asíncrona, junto con nuevas características de seguridad y funcionalidad. Estas modificaciones posicionan a DART como una opción para el desarrollo de aplicaciones modernas. La compatibilidad hacia atrás y las herramientas de migración aseguran una transición suave, permitiendo a los desarrolladores aprovechar las nuevas capacidades sin interrupciones significativas.

Sin embargo, esta actualización presenta un problema: aunque existe la nueva especificación, todavía no se han desarrollado las herramientas necesarias para analizar programas que sigan esta nueva especificación.

Los resultados obtenidos en este proyecto se centran en la actualización de la gramática al documentar los cambios en la misma, agregando nuevas reglas al programa DART3. Εsta tesis aborda la necesidad de desarrollar herramientas que permitan analizar programas conforme a la nueva especificación.

# **Introducción**

Actualmente el análisis de programas usando DART se enfoca en la segunda versión de su gramática, pero los programas actuales se escriben bajo la especificación 3.0 de lenguaje, perdiendo la posibilidad de analizar las nuevas funcionalidades del lenguaje, puesto que no existen herramientas de análisis para la nueva especificación.

## **Descripción del Problema**

El objetivo de este trabajo es actualizar las herramientas de análisis de programas para parsear programas siguiendo la nueva especificación del lenguaje, DART 3. Actualmente, el análisis de programas usando DART se enfoca en la segunda versión de su gramática, pero los programas actuales se escriben bajo la especificación 3.0 del lenguaje. Esto implica la pérdida de la posibilidad de analizar las nuevas funcionalidades del lenguaje, ya que no existen herramientas de análisis para la nueva especificación.

Para realizar el proceso de actualización de la gramática, se revisaron las especificaciones de DART 3 y DART 2, identificando los cambios a las reglas existentes y las reglas agregadas dentro de la nueva especificación. El resultado de este proceso está en la documentación de los cambios y la generación de una gramática de DART 3 en ANTLR.

## **Descripción de los Componentes del Analizador**

Las mejoras en consistencia, rendimiento y manejo de programación asíncrona, junto con nuevas características de seguridad y funcionalidad, posicionan a DART como una opción para el desarrollo de aplicaciones modernas. DART 3 no solo optimiza y fortalece el lenguaje, sino que también mejora la experiencia de desarrollo, haciendo que sea más robusto y adecuado para enfrentar los desafíos actuales y futuros en el desarrollo de software. Es crucial que se desarrollen y actualicen herramientas de análisis para que los desarrolladores puedan aprovechar al máximo estas mejoras sin enfrentar obstáculos en la evaluación y depuración de sus programas. No obstante, la falta de herramientas de análisis actualizadas puede dificultar la adopción plena de estas mejoras. La compatibilidad hacia atrás y las herramientas de migración aseguran una transición suave, permitiendo a los desarrolladores aprovechar las nuevas capacidades sin interrupciones significativas. Sin embargo, sin herramientas adecuadas para la nueva especificación, el proceso de análisis y depuración de los programas de DART puede ser complejo.

El análisis de los programas se divide entre el análisis léxico dividido entre el LEXER y el PARSER.. EL LEXER es responsable de la primera etapa del análisis del código fuente. Su función principal es convertir el código fuente a una secuencia de tokens. Los tokens son unidades léxicas que representan elementos básicos del lenguaje de programación, como palabras clave, identificadores, literales, operadores y signos de puntuacióny elimina espacios en blanco y comentarios, ya que no son necesarios para el análisis sintáctico

El proceso de análisis léxico incluye los siguientes pasos:

1. Lectura del código fuente: El lexer lee el código fuente carácter por carácter.
2. Identificación de patrones: Utiliza reglas léxicas (definidas por la gramática) para agrupar secuencias de caracteres en tokens.
3. Generación de tokens: Cada secuencia de caracteres reconocida como un patrón válido se convierte en un token. Por ejemplo, en la instrucción async => 42;, los tokens serían:

async, =>, 42, y ;. Donde async es una palabra clave o identificador. => es un operador de flecha (arrow function). 42 es un valor literal numérico. ; es un punto y coma para terminar la instrucción.

El PARSER: también conocido como analizador sintáctico, toma la secuencia de tokens generada por el lexer y la organiza en una estructura jerárquica conocida como árbol de análisis sintáctico (parse tree, PT) o árbol de sintaxis abstracta (abstract syntax tree, AST). Esta estructura representa del código fuente.

El proceso de análisis sintáctico incluye los siguientes pasos:

1. Entrada de tokens: El parser recibe la secuencia de tokens del lexer.
2. Verificación de la gramática: Utiliza reglas gramaticales (definidas generalmente por una gramática formal) para verificar que la secuencia de tokens este bien estructurada de acuerdo con la sintaxis del lenguaje de programación.
3. Construcción del árbol sintáctico: Organiza los tokens en un árbol que refleja la estructura jerárquica del código fuente. Por ejemplo, para la expresión int x = 10; el árbol sintácticose muestra a continuación..



Tabla

Descripción generada automáticamente

Para validar la corrección de la nueva especificación de la gramática se buscaron distintos fragmentos de programas de Dart que explotaran el uso de las nuevas reglas gramaticales y compararlas con las existentes en DART2 y así hallar que cambió a partir de las ramificaciones de los árboles. Los resultados que se obtuvieron son los árboles de sintaxis abstracta para la comparación de DART3 con respecto a DART2.

# **Antecedentes**

**Nuevas Características y Funcionalidades**

DART 3 introduce varias nuevas características que amplían las capacidades del lenguaje, tales como:

* **Null Safety:** Mejora la seguridad y reduce los errores relacionados con referencias nulas.
* **Mejores Librerías:** Librerías estándar mejoradas para manejar colecciones, fechas, y otras funcionalidades comunes.
* **Soporte para Interoperabilidad:** Mejor integración con otras plataformas y lenguajes, facilitando el desarrollo multiplataforma.

Si la secuencia de tokens no cumple con las reglas gramaticales, el parser generará errores sintácticos, indicando que el código fuente tiene errores de sintaxis.

# **Identificación del problema y de su importancia**

El análisis de código es una técnica esencial para identificar problemas, asegurar la calidad del código en los lenguajes de programación, incluido DART. Sin embargo, existen ciertos desafíos y limitaciones que pueden hacer que el análisis de código sea difícil o imposible en algunos casos.

Las posibles limitaciones del análisis de código incluyen la complejidad del código, que puede dificultar el análisis automático, especialmente cuando el código tiene demasiadas ramas y condiciones (complejidad ciclomática) o está mal organizado (código espagueti).

Otra limitación es la naturaleza del lenguaje de programación. Algunas características del lenguaje, como la metaprogramación, pueden ser difíciles de analizar, y las constantes actualizaciones y nuevas versiones del lenguaje pueden introducir cambios que las herramientas aún no soportan.

Las herramientas de análisis también tienen limitaciones. No todas pueden detectar todos los tipos de problemas, y pueden generar falsos positivos o negativos. Además, el análisis de código dinámico es complicado porque el código que se genera o modifica en tiempo de ejecución, o que interactúa con dependencias externas, puede ser difícil de analizar estáticamente.

La escalabilidad es otra limitación. En proyectos grandes, el análisis de código puede ser lento y requerir muchos recursos computacionales, lo que puede ser un desafío en el ciclo de integración continua.

El contexto y los requisitos del proyecto también limitan el análisis. La falta de entendimiento del dominio de aplicación puede llevar a interpretaciones incorrectas, y los requisitos no funcionales, como la usabilidad y el rendimiento, son difíciles de evaluar solo con análisis estático.

El análisis de código heredado u obsoleto presenta sus propias limitaciones, ya que el código antiguo que no sigue las mejores prácticas modernas y la falta de documentación adecuada complican el análisis.

Finalmente, la variabilidad en la calidad del código entre desarrolladores afecta la facilidad de análisis, y la revisión manual sigue siendo necesaria para complementar el análisis automático, lo cual es intensivo en tiempo y recursos.

La imposibilidad de analizar código de manera efectiva en DART 3 tiene un impacto significativo en la calidad y mantenimiento del software. Sin un análisis adecuado, los desarrolladores pueden pasar por alto errores críticos y vulnerabilidades de seguridad que podrían haberse detectado y corregido en las etapas tempranas del desarrollo. Esto no solo aumenta el costo y el esfuerzo necesarios para solucionar problemas en fases posteriores, sino que también compromete la confiabilidad y la estabilidad del software. Además, la falta de análisis reduce la capacidad de mantener un código limpio y eficiente, afectando negativamente la colaboración en equipos grandes y la escalabilidad del proyecto.

* Principio del formulario
* Final del formulario

En resumen, el análisis de código es crucial para asegurar un desarrollo de software de alta calidad, seguro y eficiente, y su ausencia puede tener consecuencias graves para cualquier proyecto.

# **Implementación de la actualización de la gramática de DART**

A continuación, se detallan los aspectos que se tuvieron en cuenta para la implementación de la actualización de la gramática DART.

## **Recolección de Información**

Para llevar a cabo un análisis detallado del lenguaje Dart y registrar los cambios encontrados, se recurrió a fuentes de documentación específicas y se anotaron en una bitácora. Este proceso fue facilitado por el uso de un laboratorio virtual de la organización ANTLR, una herramienta intuitiva y fácil de usar. ANTLR fue fundamental para la generación del analizador sintáctico basado en la gramática de Dart 3, lo que permitió un análisis más preciso del código. La automatización de las pruebas de la información gramatical garantizó la consistencia y exactitud del análisis, mientras que la generación de programas de ejemplo se apoyó en la documentación oficial de Dart 3, asegurando la validez y aplicabilidad de los resultados obtenidos.

## **Alternativas de diseño**

### **4.2.1. Diseño Basado en ANTLR**

ANTLR 4 (Another Tool for Language Recognition)[1] es una poderosa herramienta utilizada para definir y procesar gramáticas de programación. Permite la creación de analizadores léxicos y sintácticos para interpretar y manipular lenguajes de programación. ANTLR utiliza gramáticas específicas para describir la estructura de un lenguaje de programación. Una vez definidas, estas gramáticas se compilan en un conjunto de clases en el lenguaje de destino elegido (por ejemplo, Java, Python, JavaScript). ANTLR genera automáticamente el código necesario para el análisis léxico y sintáctico, permitiendo la interpretación o compilación del código fuente escrito en el lenguaje definido. El proceso de análisis del lenguaje Dart comienza con la definición de una gramática, la cual describe las reglas sintácticas del lenguaje. Esta gramática es luego compilada utilizando ANTLR, que genera el código fuente correspondiente. Posteriormente, el código generado se utiliza para realizar el análisis léxico y sintáctico, procesando el texto de entrada, identificando tokens y verificando su estructura de acuerdo con las reglas establecidas en la gramática.

**Ventajas:**

* **Flexibilidad y Potencia:** Soporta gramáticas complejas y múltiples lenguajes de destino.
* **Amplia Comunidad y Documentación:** Facilita la resolución de problemas y la implementación de soluciones avanzadas.
* **Herramientas de Depuración:** Incluye un conjunto robusto de herramientas para visualizar y depurar gramáticas.

**Desventajas:**

* **Complejidad:** Puede ser complicado para usuarios principiantes.
* **Rendimiento:** Puede ser menos eficiente en comparación con algunas alternativas más ligeras.

### **4.2.2. Diseño Basado en JavaCC**

JavaCC (Java Compiler Compiler)[2] es una herramienta que permite la definición y el procesamiento de gramáticas en el lenguaje de programación Java. Es utilizada para construir analizadores sintácticos y generadores de código. JavaCC procesa una descripción de la gramática escrita en un archivo. jj y genera un conjunto de clases Java que implementan un analizador sintáctico. Estas clases pueden ser utilizadas para analizar y validar la estructura de un programa escrito en el lenguaje definido por la gramática.

Definición de Gramática: Se crea un archivo .jj que contiene las reglas de la gramática.

Generación de Código: JavaCC procesa el archivo .jj y genera código Java.

Análisis Léxico y Sintáctico: El código generado analiza el texto de entrada y valida su estructura.

**Ventajas:**

* **Integración con Java:** Ideal para proyectos basados en Java.
* **Simplicidad:** Más sencillo que ANTLR para gramáticas menos complejas.

**Desventajas:**

* **Menos Flexible:** Menos capacidad para soportar gramáticas extremadamente complejas.
* **Comunidad y Documentación:** Menos soporte en comparación con ANTLR.

### **4.2.3. Diseño Basado en PEG.js**

PEG.js (Parsing Expression Grammar for JavaScript)[3] es una herramienta para definir y procesar gramáticas en aplicaciones JavaScript y Node.js. Es utilizada para construir analizadores sintácticos basados en expresiones de gramática. PEG.js permite definir gramáticas utilizando una sintaxis declarativa y fácil de aprender. Una vez definida la gramática, PEG.js genera un analizador sintáctico en JavaScript que puede ser utilizado para analizar y validar el texto de entrada.

Definición de Gramática: Se escribe una gramática utilizando la sintaxis de PEG.js.

Generación de Analizador: PEG.js procesa la gramática y genera un analizador en JavaScript.

Análisis Sintáctico: El analizador generado procesa el texto de entrada y verifica su conformidad con las reglas de la gramática**.**

**Ventajas:**

* **Fácil de Usar**: Sintaxis declarativa y fácil de aprender.
* **Integración con JavaScript**: Ideal para aplicaciones web y de Node.js**.**

**Desventajas:**

* **Rendimiento**: Puede ser menos eficiente para gramáticas muy grandes.
* **Flexibilidad**: Menos capacidad para gramáticas extremadamente complejas**.**

Si se requiere una gramática muy compleja y flexible, ANTLR es la mejor opción. Por lo tanto, es la alternativa que se eligió debido a su capacidad para manejar gramáticas complejas, su amplia comunidad y la disponibilidad de herramientas de depuración avanzadas.

## **Implementación**

La solución que se implemento fue la gramática de DART 3, completamente funcional para analizar el código.(nota al pie de página)

En el proceso de implementación se realizó un diagnóstico detallado de las modificaciones necesarias, identificando los cambios específicos que se deberían realizar.

Una vez definidos, estos cambios se escribieron en el lenguaje ANTRL, asegurándose que las modificaciones estuviesen correctamente codificadas en el lenguaje utilizado para la gramática y el análisis.

Durante este proceso se identificó la falta de nuevas palabras reservadas del lenguaje, necesarias para las modificaciones sintácticas. En total se agregaron 5 nuevas palabras y símbolos.[8]

Se añadieron 16 reglas faltantes en el analizador sintáctico, actualizando las reglas de análisis para incorporar las nuevas modificaciones y vocabulario añadido.

Posteriormente, se llevaron a cabo pruebas en el laboratorio de ANTLR, donde se verificó que las modificaciones funcionaran correctamente y se ajustaran a los requisitos esperados.

Finalmente, se revisaron los nodos adicionales y las ramificaciones que cambiaron debido a las modificaciones, asegurando que todos los elementos adicionales y sus implicaciones estuviesen correctamente integrados y funcionando como se esperaba.

Este proceso asegura una implementación completa y precisa de las modificaciones en el sistema.

## **Diseño e Implementación de la Gramática**

Se desarrolló una gramática que cubriera la sintaxis específica de DART 3, incluyendo esas estructuras de control, declaraciones, expresiones y otras construcciones del lenguaje.

Se utilizo la herramienta de generación de analizadores ANTLR para convertir la gramática en un analizador sintáctico y así poder procesar archivos de código DART.

En resumen, la implementación de esta gramática nos permitirá usar herramientas automatizadas para generar el analizador ahorrando tiempo y reduciendo errores humanos en la implementación manual de la misma.

# **Validación**

Para cada una de las especificaciones de la gramática se generaron arboles de sintaxis abstracta los cuales al construirse permitieron revisar las diferencias entre las reglas de la versión anterior y las reglas de la versión actual, fue así como estas pruebas nos permitieron de validar los resultados.

**functionBody**: Se eliminó la opción de definir cuerpos de función con **NATIVE\_**, simplificando las opciones a cuerpos de función asincrónicos o sincrónicos y bloques de código regulares.

**declaration**: Se simplificó esta regla para estructurar mejor las declaraciones, como constructores y firmas de funciones, eliminando o reorganizando algunos modificadores como **ABSTRACT\_**.

**initializerExpression** y **typeNotVoid**: Se expandieron para aceptar más tipos de expresiones y tipos, incluyendo expresiones asignables y lanzamiento de excepciones (**throwExpression**), así como la adición de cualificadores opcionales.

**identifier** y **typeIdentifier**: Se reorganizaron para separar identificadores incorporados y otros tipos de identificadores, haciendo la gramática más modular y fácil de manejar.

**libraryDeclaration** y **partDeclaration**: Estos cambios reflejan una estructura más organizada para las declaraciones de bibliotecas y partes de código, permitiendo una mejor integración de etiquetas y metadatos.

**partDeclaration**: se añadió EOF al final de la definición partDeclaration. Esto asegura que después de partHeader y cualquier cantidad de metadata topLevelDeclaration, el analizador espera el final del archivo. Este cambio probablemente se realizó para garantizar que no haya elementos adicionales después de esta declaración en la entrada analizada.

**typeNotVoid**: Aquí, se agregó QU? a la alternativa typeNotVoidNotFunction. Esto permite que la opción typeNotVoidNotFunction también tenga un sufijo opcional QU, proporcionando más flexibilidad en la definición de typeNotVoid.

**typeNotFunction**: En este cambio, se añadió QU? a typeNotVoidNotFunction en la definición de typeNotFunction. Esto permite que typeNotVoidNotFunction pueda tener un sufijo opcional QU dentro de typeNotFunction, nuevamente proporcionando mayor flexibilidad en la gramática.

**typeNotVoidnotFunction**: Aquí, se eliminó QU? de la alternativa FUNCTION\_. Este cambio puede haber sido hecho para restringir la definición de typeNotVoidNotFunction, asegurando que FUNCTION\_ no pueda tener un sufijo opcional QU.

**typeIdentifierNotType**: Define un identificador que no es un tipo. Puede ser un identificador general (IDENTIFIER), otro identificador no tipo (otherIdentifierNotType) o la palabra clave DYNAMIC\_.

**builtInIdentifier**: Enumera una serie de identificadores predefinidos en el lenguaje, como ABSTRACT\_, AS\_, DYNAMIC\_, EXPORT\_, etc. Estos son identificadores con un significado específico en el contexto del lenguaje.

**otherIdentifierNotType**: Define otros identificadores que no son tipos, como ASYNC\_, HIDE\_, OF\_, ON\_, etc.

**otherIdentifier**: Amplía otherIdentifierNotType al incluir también TYPEDEF\_.

**identifierNoDollar**: Define un identificador que no contiene el símbolo $, utilizando identifierStartNoDollar y identifierPartNoDollar.

**reservedWord**: Enumera todas las palabras reservadas del lenguaje como ASSERT\_, BREAK\_, CASE\_, CATCH\_, CLASS\_, etc. Estas palabras tienen un significado especial y no pueden usarse como identificadores.

**singleLineComment**: Define un comentario de una sola línea que comienza con // y termina con un salto de línea opcional.

**multiLineComment**: Define un comentario de múltiples líneas que comienza con /\* y termina con \*/. Puede contener otros comentarios de múltiples líneas o cualquier carácter que no sea \*/.

**singleLineString**: Define una cadena de texto de una sola línea, incluyendo varias variantes:

rawSingleLineString

singleLineStringSQBeginEnd

singleLineStringSQBeginMid expr (singleLineStringSQMidMid expr)\* singleLineStringSQMidEnd

singleLineStringDQBeginEnd

singleLineStringDQBeginMid expr (singleLineStringDQMidMid expr)\* singleLineStringDQMidEnd

**rawSingleLineString**: Define una cadena de texto sin procesar que comienza con r seguido de comillas simples o dobles, sin permitir caracteres específicos como salto de línea.

**stringContentCommon**: Define el contenido común de las cadenas de texto, excluyendo ciertos caracteres como comillas y signos de dólar, pero permitiendo secuencias de escape y interpolaciones simples.

**stringContentSQ**: Amplía stringContentCommon permitiendo también comillas dobles dentro de una cadena de comillas simples.

**singleLineStringSQBeginEnd**: Define una cadena de texto de comillas simples que comienza y termina con comillas simples, conteniendo stringContentSQ.

singleLineStringSQBeginMid, singleLineStringSQMidMid, singleLineStringSQMidEnd: Definen las partes de una cadena de texto de comillas simples que incluye interpolación de expresiones con ${}.

**stringContentDQ**: Similar a stringContentCommon, pero permite comillas simples dentro de una cadena de comillas dobles.

**singleLineStringDQBeginEnd**: Define una cadena de texto de comillas dobles que comienza y termina con comillas dobles, conteniendo stringContentDQ.

singleLineStringDQBeginMid, singleLineStringDQMidMid, singleLineStringDQMidEnd: Definen las partes de una cadena de texto de comillas dobles que incluye interpolación de expresiones con ${}.

**whitespace**: Define espacios en blanco que pueden incluir tabulaciones, espacios y saltos de línea.

Estos ajustes ayudan a clarificar y optimizar la definición y análisis sintáctico del lenguaje, lo que puede conducir a un análisis de código más eficiente y menos propenso a errores. Estas modificaciones suelen responder a la necesidad de adaptar el lenguaje a nuevos requisitos, mejorar la legibilidad del código o simplificar el compilador o el intérprete del lenguaje.

# **Resultados**

La regla anterior **functionBody** permitía ambigüedades con NATIVE\_. La nueva definición simplifica las opciones y clarifica el código, a continuación, un par de ejemplos para esta regla, presentare tres tipos de casos: uno que representa una eliminación de elementos del lenguaje, otro que se representa una novedad o algún elemento que se agregó y otro que representa algún que no cambió con la instrucción async=>42 se genera tanto para DART 2 como para DART3 el siguiente AST.

|  |  |
| --- | --- |
| **DART 2** | **DART 3** |
| **functionBody**  **: NATIVE\_ stringLiteral? SC**  **| ASYNC\_? EG expr SC**  **| ( ASYNC\_ ST? | SYNC\_ ST)? block**  **;** | **functionBody**  **: ASYNC\_? EG expr SC**  **| ( ASYNC\_ ST? | SYNC\_ ST)? block**  **;** |

Tabla

Descripción generada automáticamente

**Ejemplo regla functionBody**

Future<int> fetchData() async => await Future.delayed(Duration(seconds: 2), () => 100);

La función fetchData es una función asíncrona que retorna un Future<int>. Utiliza un cuerpo de expresión, definido mediante la sintaxis de flecha (=>), lo que permite una definición concisa para funciones que contienen una única expresión. Al analizar esta función en el contexto de la regla de producción functionBody actualizada, observamos cómo cada componente de la regla se aplica de manera coherente.

En esta función, la palabra clave async indica que la función es asíncrona, alineándose con la parte opcional ASYNC\_? de la regla. La sintaxis de flecha (=>) corresponde a EG, que define el inicio del cuerpo de la expresión. La expresión en sí, await Future.delayed(Duration(seconds: 2), () => 100), es la parte que se evalúa y devuelve, cumpliendo con el componente expr. Finalmente, el punto y coma (;) al final de la línea corresponde a SC, que finaliza la instrucción.

Este diseño simplificado elimina ambigüedades presentes en versiones anteriores, como el manejo de la palabra reservada NATIVE\_, y clarifica la estructura del cuerpo de la función. Al adoptar esta definición más precisa, el analizador sintáctico puede interpretar correctamente la función fetchData, asegurando que las herramientas de análisis desarrolladas para Dart 3 manejen adecuadamente tanto cuerpos de expresión como de bloque. Esto no solo mejora la precisión del análisis, sino que también facilita la incorporación de nuevas características del lenguaje, como mejoras en la programación asíncrona, proporcionando una base sólida para el desarrollo de aplicaciones modernas y eficientes.

**Contextualización de la Regla functionBody en la Gramática de Dart**

En la gramática de Dart, la regla functionBody juega un papel crucial al definir cómo se estructura y se interpreta el cuerpo de una función. En Dart 2, esta regla permitía ciertas ambigüedades, especialmente al manejar casos específicos como el uso de la palabra reservada NATIVE\_. Estas ambigüedades podían generar conflictos durante el análisis sintáctico, dificultando la correcta interpretación del código fuente por parte del analizador. Por lo tanto, era necesario revisar y actualizar esta regla para eliminar dichas ambigüedades y mejorar la precisión del análisis.

**Justificación de la Actualización de la Regla functionBody**

La actualización de la regla functionBody se motivó por la necesidad de simplificar las opciones disponibles y clarificar la estructura del código. Al eliminar las ambigüedades asociadas con NATIVE\_, se logró una definición más precisa y menos propensa a errores. Además, esta simplificación facilita la incorporación de nuevas características introducidas en Dart 3, como mejoras en la programación asíncrona y en la seguridad del lenguaje. Con una gramática más clara y directa, las herramientas de análisis desarrolladas pueden interpretar de manera más eficiente y fiable los cuerpos de las funciones, asegurando una mejor compatibilidad y rendimiento en proyectos modernos.

Adicionalmente, en Dart 2, un function body define la lógica que ejecuta una función al ser llamada y puede ser una expresión o un bloque de código. Una expresión utiliza la sintaxis de "flecha" (=>) para funciones simples de una sola línea, como int add(int a, int b) => a + b;, mientras que un bloque de código, encerrado entre llaves {}, se usa para funciones más complejas que requieren múltiples declaraciones, por ejemplo, int add(int a, int b) { return a + b; }. Ambos permiten procesar los parámetros de entrada y devolver un resultado.

**functionBody DART2**

native "gg";

async => 42;

async {

await Future.delayed(Duration(seconds: 1));

return 42;

}

{print('Hello, World!’);}

En Dart 3, un function body sigue definiendo la lógica de ejecución de una función, pero se han introducido mejoras significativas para aumentar la flexibilidad y la claridad del código. Además de las expresiones de una sola línea con la sintaxis de "flecha" (=>) y los bloques de código entre llaves {}, Dart 3 incorpora soporte mejorado para funciones asincrónicas y generadores, permitiendo una sintaxis más limpia y eficiente para manejar operaciones asíncronas y flujos de datos. Por ejemplo, la declaración de funciones async y async\* es más intuitiva y optimizada, facilitando la escritura de código no bloqueante y concurrente. Estas mejoras no solo hacen que el código sea más legible y mantenible, sino que también mejoran el rendimiento y la capacidad de respuesta de las aplicaciones, haciendo de Dart 3 una actualización robusta y eficiente para el desarrollo moderno.

**functionBody DART3**

async => 42;

|  |  |
| --- | --- |
| DART 2 | DART 3 |
|  |  |

|  |  |
| --- | --- |
| La nueva regla **initializerExpression** ahora permite expresiones asignables con operadores, lo que amplía la flexibilidad en la inicialización.DART 2 | DART 3 |
|  | **initializerExpression**  **: assignableExpression assignmentOperator expr**  **| conditionalExpression**  **| cascade**  **| throwExpression    ;** |

void main() {

var greeting = "Hello World";

print(greeting);

}

A continuación, el árbol que genera la instrucción anterior en DART 3.

Tabla

Descripción generada automáticamente

**Ejemplo regla No. 2 (Initializer Expression)**

void main() {

var greeting = StringBuffer("Hello")

..write(" World")

..write("!");

print(greeting);

}

Pensamiento durante 8 segundos

Un ejemplo que ilustra la alternativa de **cascade** en la regla initializerExpression y genera un “Hello World” es el siguiente:

dart

Copiar código

void main() {

var greeting = StringBuffer("Hello")

..write(" World")

..write("!");

print(greeting);

}

En este programa, la variable greeting se inicializa mediante la creación de un objeto de tipo StringBuffer con el texto inicial "Hello" y, a continuación, se aplica una secuencia de llamadas encadenadas gracias al operador de cascada (..). Este operador permite realizar sucesivas operaciones sobre la misma instancia de StringBuffer sin necesidad de asignarla repetidamente. En cada llamada a write, se agregan nuevas partes del mensaje hasta formar la cadena final "Hello World!", la cual se imprime en la consola. Este patrón encaja con la producción de la gramática que contempla la construcción de una cascade como expresión de inicialización

La inclusión de throwExpression en initializerExpression permite manejar excepciones directamente en expresiones iniciales, mejorando la claridad y control de errores.

En Dart, el error **typeNotFunction** ocurre cuando se intenta invocar algo que no es una función como si lo fuera. Este error se presenta al intentar llamar a una variable, objeto o tipo que no tiene capacidad de ejecutarse como función, como un entero o una referencia a un método sin paréntesis. Para evitar este error, es importante asegurarse de que cualquier elemento invocado sea efectivamente una función o método, y de diferenciar correctamente entre propiedades y métodos. Comprender esta regla permite escribir código más robusto y evitar errores comunes en Dart.

**typeNotFunction DART2**

double? measureTemperature() {

return null; // Example: No temperature measurement available

}

void printMessage(String message) {

print(message);

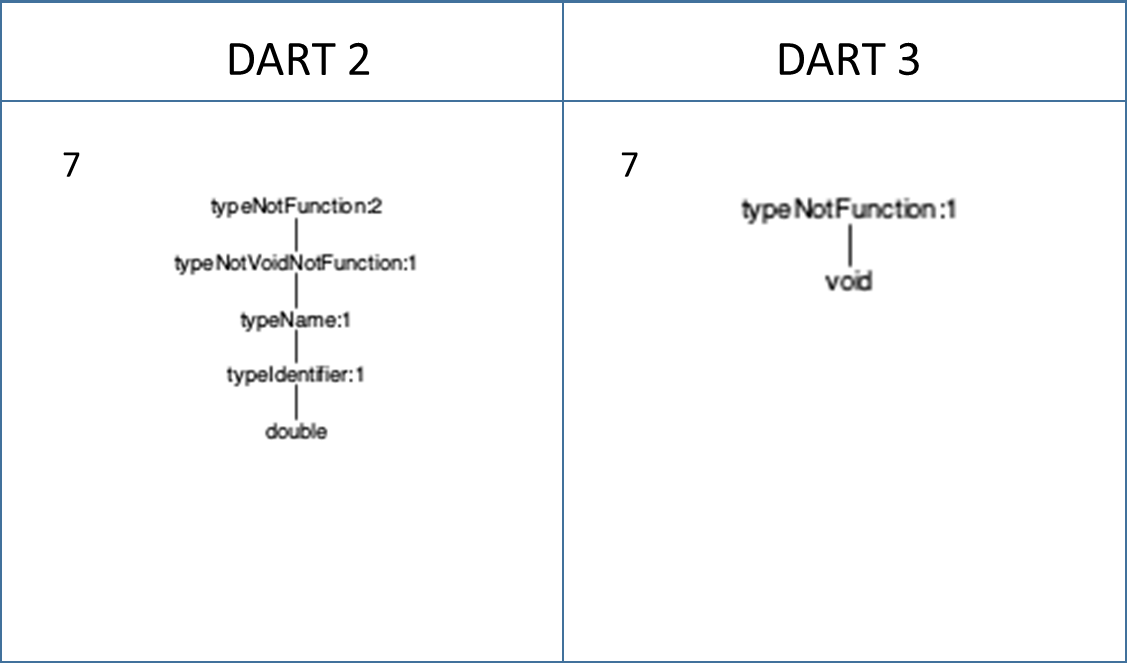
}

En Dart 3, la regla typeNotFunction sigue previniendo la invocación de objetos no funcionales como si fueran funciones, pero se beneficia de una mayor rigurosidad en la tipificación y mejoras en el manejo de tipos de funciones. Estas mejoras permiten detectar errores como typeNotFunction de manera más temprana y precisa, reduciendo la probabilidad de intentos incorrectos de invocar variables o propiedades no funcionales.

**typeNotFunction DART3**

async => 42;

Tabla

Descripción generada automáticamente

La regla **type\_not\_void** en Dart 2 asegura que todas las funciones, métodos y getters devuelvan un valor de tipo específico y no void, salvo cuando no se pretende devolver nada. Esto fomenta una programación más segura y explícita, al evitar ambigüedades y errores potenciales cuando una función inesperadamente no devuelve un valor. Al aplicar esta regla, los desarrolladores deben pensar cuidadosamente sobre el tipo de datos que sus funciones deben devolver, resultando en un código más robusto y fácil de mantener. Implementar type\_not\_void ayuda a detectar errores en tiempo de desarrollo, mejorando la eficiencia del proceso y manteniendo la integridad del tipo en las aplicaciones.

**typeNotVoid DART2**

int myInteger;

User myUser;

List<String> myStringList;

Map<String, int> myMap;

double? myNullableDouble;

**DART 3**

En Dart 3, la regla type\_not\_void ha sido mejorada para proporcionar aún más seguridad y precisión en la gestión de tipos de retorno. A diferencia de su predecesor en Dart 2, Dart 3 introduce una verificación más estricta y detallada, que no solo asegura que las funciones, métodos y getters devuelvan un tipo específico y no void, sino que también incorpora análisis en tiempo de compilación para detectar inconsistencias y potenciales errores de tipo con mayor eficacia. Este cambio fortalece la capacidad del compilador para identificar funciones que deben devolver valores específicos y ayuda a los desarrolladores a mantener un código aún más robusto y libre de errores, mejorando así la calidad y la confiabilidad de las aplicaciones escritas en Dart.

**typeNotVoid DART3**

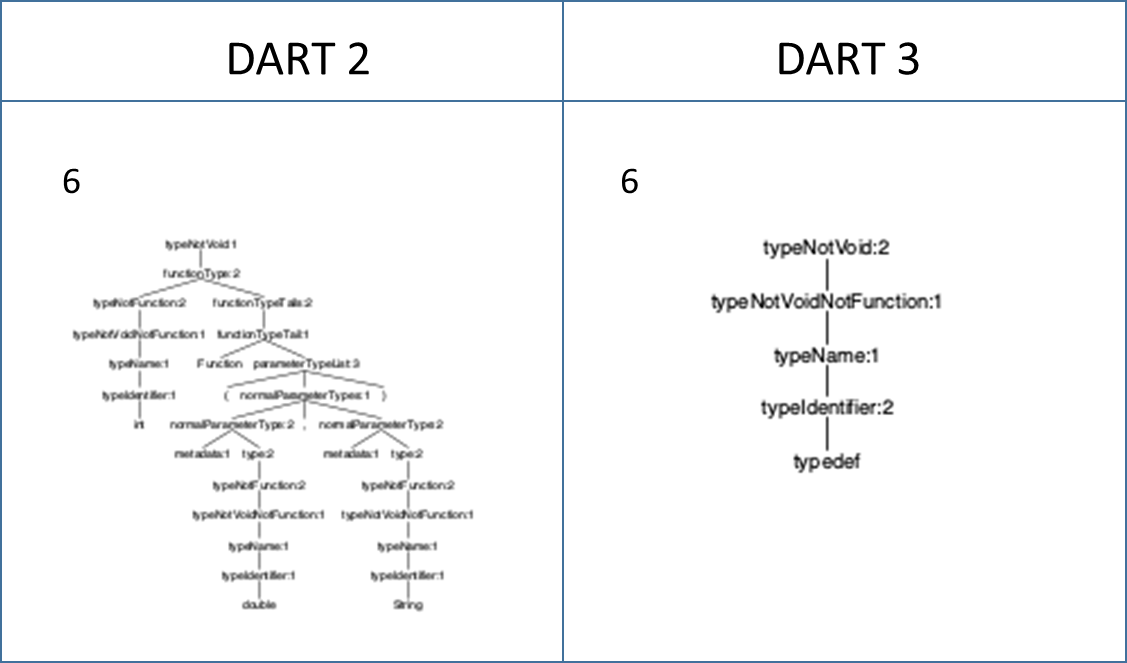
List<String> myList = [];

typedef IntToString = String Function(int);

String? myString;

Compare<int> compareIntegers = (int a, int b) => a - b;

Tabla

Descripción generada automáticamente con confianza media 

En Dart 2, la declaración de bibliotecas se realiza utilizando la palabra clave library, seguida del nombre de la biblioteca. Esta declaración debe colocarse al principio del archivo y permite organizar y encapsular el código en módulos reutilizables. Por ejemplo, library my\_library; establece una biblioteca llamada my\_library, y se puede usar junto con directivas como import y export para gestionar dependencias y exposiciones de símbolos. Esta estructura ayuda a mantener el código ordenado y modular, facilitando la colaboración y el mantenimiento.

**libraryDeclaration DART2**

library my\_awesome\_library;

export ‘utils.dart';

part ‘part\_of\_library.dart';

final String description;

@MyMetadata('This is a top-level function')

void myTopLevelFunction() {

print('This is a top-level function in the library.');

}

@MyMetadata('This is a top-level class')

class MyTopLevelClass {

void doSomething() {

print('Doing something in the top-level class.');

}

}

class PartOfLibraryClass {

void doPartOfLibraryThing() {

print('Doing something in the part of the library.');

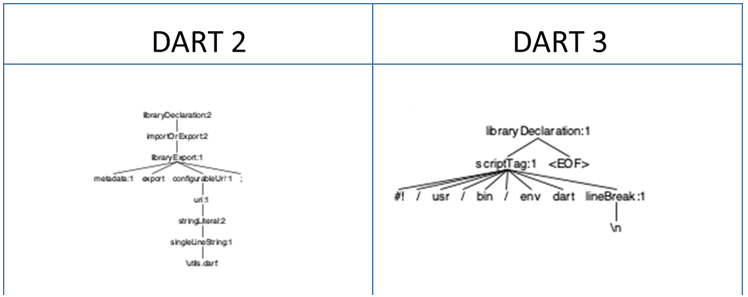
}

}

En Dart 3, aunque la declaración de bibliotecas sigue siendo similar a Dart 2 en cuanto al uso de la palabra clave library para definir el nombre de la biblioteca al inicio del archivo, se han introducido mejoras significativas en la gestión y resolución de dependencias. Dart 3 permite una integración más profunda con sistemas de paquetes y módulos, mejorando la capacidad para resolver y gestionar versiones de bibliotecas de manera más eficiente. Además, se han incorporado herramientas de análisis estático más avanzadas que facilitan la detección de conflictos y optimizan la organización del código, promoviendo un desarrollo más ágil y con menos errores.

**libraryDeclarationDART3**

#!/usr/bin/env dart



En Dart 2, una declaration (declaración) introduce nuevas entidades como variables, funciones, clases o bibliotecas dentro del programa. Estas declaraciones permiten asignar nombres y definir comportamientos o valores. Por ejemplo, int count = 10; es una declaración de variable, void greet() { print('Hello'); } es una declaración de función, y class Person { String name; int age; } es una declaración de clase. Estas estructuras son esenciales para la organización del código.

**declaration DART2**

abstract static late Map<String, dynamic> data = {};

static late Map<String, dynamic> data = {};

final int x, y;

external const Point(this.x, this.y);

external factory Shape();

final String name;

external Logger(this.name);

external static int get maxScore;

external static set config(Map<String, dynamic> config);

external static int add(int a, int b);

final int numerator, denominator;

const Fraction(this.numerator, this.denominator);

external Fraction operator +(Fraction other);

static const double pi = 3.141592653589793;

static final String apiUrl = “https://api.example.com";

static void initialize() {

instance = Singleton.\_internal();

}

static late final Singleton instance;

covariant late final String name;

void setName(String newName) {

name = newName;

}

covariant late String type = ‘Unknown';

void setType(String newType) {

type = newType;

}

late final String username;

void initialize(String name, int userAge) {

username = name;

}

late String connectionString;

void connect(String connectionString) {

this.connectionString = connectionString;

}

double x, y;

Point(this.x, this.y);

factory Point.alongXAxis(double x) = Point;

final double x, y;

const ImmutablePoint(this.x, this.y);

double width, height;

Rectangle(this.width, this.height);

Rectangle.square(double side) : this(side, side);

En Dart 3, declaration (declaraciones) ha mejorado significativamente. Además del mismo concepto básico, Dart 3 ofrece un análisis estático más avanzado y una integración completa de la seguridad de null (null safety), lo que permite detectar y prevenir errores en tiempo de compilación con mayor precisión. La mejora en la gestión de tipos asegura que el código sea más robusto y menos propenso a errores, facilitando así una codificación más segura y clara.

**declaration DART3**

static late Map<String, dynamic> data = {};

|  |  |
| --- | --- |
| **DART 2** | **DART 3** |
|  |  |

En DART 2, un identifier es un nombre utilizado para identificar una variable, función, clase, o cualquier otra entidad definida en el código. Los identificadores deben comenzar con una letra o un guion bajo (\_) y pueden contener letras, dígitos y guiones bajos. Por ejemplo, myVariable, \_hiddenFunction, y ClassName son identificadores válidos. Los identificadores son fundamentales para referenciar y manipular entidades dentro del programa.identifier **dart2.**

int myVariable = 42;

external factory() {

return this;

}

await Future.delayed(Duration(seconds: 1));

En DART 3, los identifiers (identificadores) siguen las mismas reglas básicas que en Dart 2, pero con mejoras en la detección de errores y la claridad del código. Dart 3 ofrece una mejor integración con herramientas de análisis estático que ayudan a evitar colisiones de nombres y otros problemas relacionados con los identificadores, haciendo el código más comprensible y seguro desde el punto de vista sintáctico.identifier dart3.

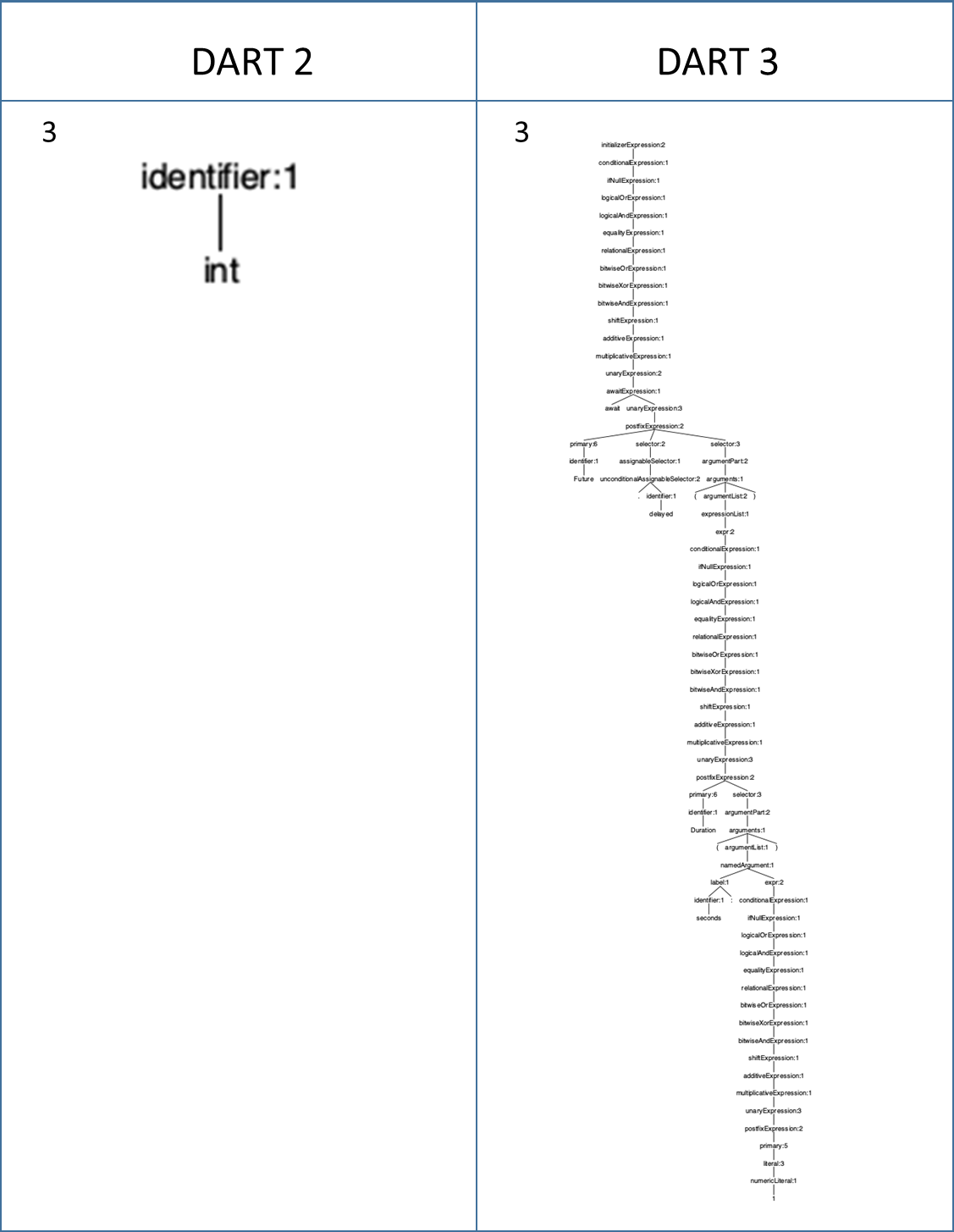
await Future.delayed(Duration(seconds: 1));

external factory() {

return this;

}

Esquemático

Descripción generada automáticamente con confianza media

En Dart 2, un type identifier es un nombre que se utiliza para identificar tipos de datos como clases, interfaces, y tipos definidos por el usuario. Por ejemplo, en la declaración String name;, String es un type identifier. Los type identifiers permiten al compilador entender y verificar los tipos de datos que se están utilizando en el código, ayudando a prevenir errores de tipo.ASYNC asyncOperation;

HIDE hiddenValue;

OF mySet;

ON myStream;

SHOW myList;

SYNC syncFunction;

AWAIT waitingFunction;

YIELD rangeFunction;

DYNAMIC dynamicType;

NATIVE nativeType;

FUNCTION functionType;

MyClass myInstance = MyClass(

() async {},

42,

{1, 2, 3},

Stream.fromIterable(['a', 'b', 'c']),

[3.14, 2.71],

() {},

Future.value(10),

(range(1, 3)),

'Dynamic can be anything',

'Native behavior',

() => print('Function type'),

);

En DART 3, los type identifiers mantienen la misma funcionalidad esencial, pero con un análisis de tipos más avanzado. Esto incluye mejoras en la inferencia de tipos y una mejor detección de inconsistencias de tipo durante la compilación. Estas mejoras permiten a los desarrolladores escribir código más preciso y ayudan a evitar errores tipográficos y conceptuales relacionados con el uso de tipos.

**typeIdentifierDART3**

MyClass myInstance = MyClass(

() async {},

42,

{1, 2, 3},

Stream.fromIterable(['a', 'b', 'c']),

[3.14, 2.71],

() {},

Future.value(10),

(range(1, 3)),

'Dynamic can be anything',

'Native behavior',

() => print('Function type'),

);

MyClass instance = MyClass();

int myNumber = 5;

List<String> myList = [];

typedef IntToString = String Function(int);

IntToString converter = (int x) => x.toString();

typedef Comparator<T> = int Function(T a, T b);

ASYNC asyncOperation;

HIDE hiddenValue;

OF mySet;

ON myStream;

SHOW myList;

SYNC syncFunction;

AWAIT waitingFunction;

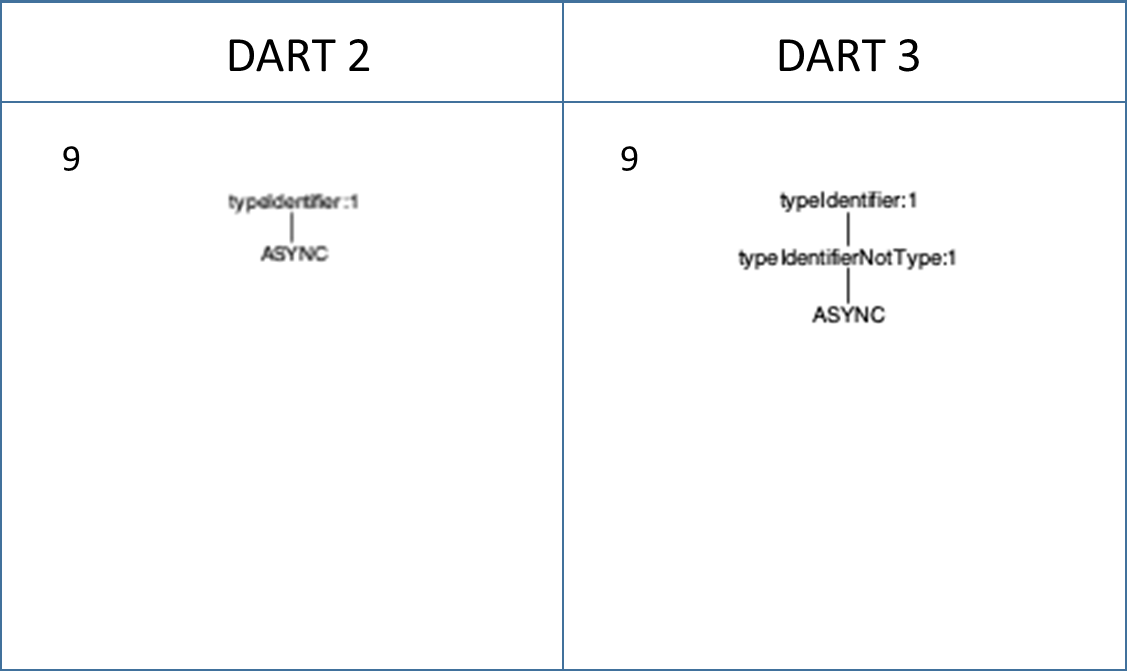
YIELD rangeFunction;

DYNAMIC dynamicType;

NATIVE nativeType;

FUNCTION functionType;

Tabla

Descripción generada automáticamente

En DART 2, una part declaration se utiliza para dividir un programa en varios archivos. Utilizando las directivas part y part of, puedes organizar tu código en múltiples archivos que se combinan en una única biblioteca. Por ejemplo, part 'src/helper.dart'; en un archivo principal y part of 'main.dart'; en el archivo secundario. Esto facilita la organización y el mantenimiento de proyectos grandes.

**partDeclaration DART2**

part of 'main\_library.dart';

En Dart 3, las part declarations siguen siendo utilizadas para dividir el código en múltiples archivos, pero con mejoras en la gestión de dependencias y la coherencia del código. Dart 3 proporciona una mejor integración con herramientas de construcción y análisis, permitiendo una organización de código más eficiente y evitando problemas comunes asociados con la referencia cruzada entre archivos. Estas mejoras simplifican la estructuración de proyectos y mejoran la claridad general del código.

**partDeclarationDART3**

|  |  |
| --- | --- |
| DART 2 | DART 3 |
|  |  |

**Ejemplo regla No. 16**

part of 'main\_library.dart';

# Este archivo forma parte de una biblioteca mayor definida en otro archivo Dart llamado main\_library.dart. La sentencia part of 'main\_library.dart'; indica que el contenido de este archivo debe considerarse como una extensión directa de la misma biblioteca. De esta forma, las funciones, variables y clases definidas aquí comparten el mismo espacio de nombres y pueden ser utilizadas de manera transparente dentro de main\_library.dart. Esta organización modular permite dividir el código en varios archivos, manteniendo una estructura más ordenada y fácil de mantener.**Conclusión**

La actualización de la semántica en DART 3 introduce mejoras significativas que impactan tanto en la facilidad de uso del lenguaje como en su eficiencia y capacidad para manejar tareas complejas.

**Mejora en la Consistencia y Claridad del Lenguaje**

Las actualizaciones semánticas en DART 3 buscan reducir ambigüedades y mejorar la consistencia en la interpretación del código. Esto se traduce en un lenguaje más intuitivo y accesible, lo que facilita su aprendizaje y uso diario. Las reglas de sintaxis y semántica más estrictas ayudan a evitar errores comunes y a asegurar que el código se comporte de manera predecible.

**Compatibilidad y Migración**

Aunque la actualización trae consigo muchas ventajas, DART 3 también se ha diseñado para ser compatible con versiones anteriores, facilitando la migración de proyectos existentes. Las herramientas de migración y el soporte continuo aseguran que los desarrolladores puedan aprovechar las nuevas características sin enfrentar obstáculos significativos.

# **Trabajo a futuro**

Para el futuro se propone agregarle características de user interface al proyecto con el fin de visualizar los cambios en el código de forma más intuitiva, esto se puede hacer a través de Flutter ya que es un lenguaje muy simple y que permite mayor cobertura de plataformas en formato híbrido, tanto para los dispositivos que usan iOS como para los que usan Android.

# 

# **Referencias:**

[1] "ANTLR - ANother Tool for Language Recognition," ANTLR. [Online]. Disponible: https://www.antlr.org/. [Accedido: 15-Feb-2024].

[2] "JavaCC - The most popular parser generator for use with Java applications," JavaCC. [Online]. Disponible: https://javacc.github.io/javacc/. [Accedido: 15-Feb-2024].

[3] B. Ford, "Parsing Expression Grammars: A Recognition-Based Syntactic Foundation," Massachusetts Institute of Technology, Cambridge, MA. [Online]. Disponible: https://bford.info/pub/lang/peg.pdf. [Accedido: 15-Feb-2024].

4"Documentation - DART." [Online]. Available: https://DART.dev/guides. \*\*

5."DART Programming Language Specification," DART, [Online]. Available: [https://DART.dev/guides/language/spec](https://dart.dev/guides/language/spec).

6. P. Leger, H. Fukuda, N. Cardozo, and D. San Martín, "Exploring a Self-Replication Algorithm to Flexibly Match Patterns," IEEE Access, vol. 12, pp. 13553-13566, Jan. 2024, doi: 10.1109/ACCESS.2024.3355319. \*\*\*\*

7. PEG.js: https://pegjs.org/

[8] "Dart 3 Overview," Lazebny. [Online]. Disponible: https://lazebny.io/dart3/. [Accedido: 28-Mar-2024].

## 

# **Apéndices**

Para más información sobre los resultados de esta investigación consulte el siguiente repositorio de propiedad pública en GitHub.

<https://github.com/LuimarcoCarrascalDiaz/UIClones>