UNIVERSIDAD CENTROAMERICANA

JOSÉ SIMEÓN CAÑAS

INTRODUCCION DE UN LENGUAJE DE DOMINIO ESPECÍFICO ENFOCADO EN COMPONENTES WEB COMPILANDO HACIA DISTINTOS FRAMEWORKS

TRABAJO DE GRADUACIÓN PREPARADO PARA LA FACULTAD DE INGENIERÍA Y ARQUITECTURA

PARA OPTAR AL GRADO DE

INGENIERO INFORMÁTICO

POR:

YURY ALEJANDRO RIVERA QUINTANILLA

MAYO 2024

ANTIGUO CUSCATLÁN, EL SALVADOR, C.A

RECTOR

MARIO ERNESTO CORNEJO MENA, S. J.

SECRETARIA GENERAL

LIDIA GABRIELA BOLAÑOS TEODORO

DECANO DE LA FACULTAD DE INGENIERÍA Y ARQUITECTURA

CARLOS ERNESTO RIVAS CERNA

DIRECTOR DE LA CARRERA DE INGENIERÍA INFORMÁTICA

JOSÉ ENMANUEL AMAYA ARAUJO

DIRECTOR DEL TRABAJO

JOSÉ ENMANUEL AMAYA ARAUJO

LECTOR

RONALDO ARMANDO CANIZALES TURCIOS

RESUMEN

Las tecnologías web han avanzado mucho desde su invención, producto de una siempre creciente demanda de características nuevas y más complejas debido a las exigencias de los usuarios de la web moderna. Muchas de estas mejoras fueron utilizadas para crear abstracciones cada vez de mayor complejidad y diferentes entre ellas, por esa razón muchas de las tecnologías web que existen hoy, como React o Vue son incompatibles unas con otras, a pesar de que ambas se ejecutan en un lenguaje llamado JavaScript.

Desde entonces se han inventado muchas soluciones distintas al problema de generar interfaz gráfica para el usuario de manera dinámica, esto ha fragmentado el estado actual del desarrollo web y principalmente afecta a los autores de librerías. Para ellos se ha vuelto una tarea muy difícil el portar sus librerías de una tecnología a la otra sin la necesidad de volver a programar nuevamente.

Por esa razón el enfoque de este trabajo es crear una alternativa que, en lugar de construir aplicaciones completas, únicamente genere la parte de interfaz gráfica de manera automática. Esto permite a los desarrolladores portar sus librerías a las distintas tecnologías disponibles. Esto se logrará haciendo uso de un lenguaje de dominio específico que se desarrollará utilizando varias técnicas y teorías que se van describiendo a lo largo de la obra.

En el primer capítulo se brinda el contexto sobre la evolución de la web, como tecnologías clave, cuáles son sus funciones y como estas han ido incorporando con otras prácticas del desarrollo web. Se mencionan hitos como librerías y otras tecnologías cuya revolucionaria visión del uso de JavaScript supuso un antes y un después en el desarrollo web. Además, se describen las limitaciones y objetivos de la investigación.

En el capítulo dos se describen generalidades sobre los lenguajes de programación, esto incluye cuestiones como sus elementos, diseño, paradigmas y detalles técnicos como la implementación de tipado en el lenguaje, fundamentos como la explicación de gramáticas y como se relacionan con la forma en que un programador escribe código en un lenguaje dado.

Después, en el capítulo tres se explorarán las técnicas y teorías necesarias para poder desarrollar un lenguaje de programación que incorpore las características que necesitaremos para crear un lenguaje que permita incorporar HTML, CSS y scripting de manera fluida. Esto incluye mucho sobre el análisis y procesamiento de archivos de texto planos y todas las distintas etapas que permiten a partir de este, terminar con código que se ejecute en la computadora e interactúe con el usuario.

Durante el cuarto capítulo se exploran más a detalle que son los lenguajes de dominio específico, sus principales diferencias con los lenguajes de propósito general. También se exploran los principios de diseño, es decir las directrices que se deben tener en cuenta al desarrollar un lenguaje de dominio específico, así como sus distintos tipos. De ambos tipos, lenguajes de dominio específico internos o externos se enfoca más en el segundo, ya que este enfoque es más flexible al momento de crear un lenguaje personalizado.

Se concluye con el capítulo 5 la síntesis de la obra, la integración de las técnicas y las teorías para empezar a definir lo que se convertirá en el lenguaje Haibt, sus características y expresividad sintáctica. Además, se trabajará en un compilador que es la pieza de software que permitirá pasar de un archivo Haibt en alguno de las tecnologías propuestas como React o Vue. Esta conversión tratará de ser un programa que sea evaluado antes de ser compilado, esto significa que el compilador podrá interpretar el código y deliberar si el código es correcto o no. Esto significa que el programa se ejecutara correctamente una vez haya sido compilado. En este capítulo se aborda más a detalle cómo se incorporan las distintas técnicas en este único programa y como la entrada en texto plano pasa por distintas fases, transformándose en varias estructuras que eventualmente podrán ser traducidas a código Typescript, el lenguaje de destino.

Finalmente, en el sexto capítulo se realiza una retrospectiva sobre el programa resultante, enfocándonos en las conclusiones que se obtuvieron producto de incorporar los conocimientos en un programa tan complejo como lo puede ser un compilador. También es pertinente mencionar cosas que se pudieron apreciar durante la definición de la gramática y el entendimiento más profundo que se obtuvo de como los lenguajes de programación son procesados por sus respectivos compiladores.

La otra parte importante del sexto capítulo son las recomendaciones, el resultado tiene margen para muchas mejoras, con algunas cosas ya en su lugar y funcionando es más fácil observar el gran esquema de las cosas y como el trabajo resultante podría ser mejorado. Muchas de las recomendaciones provienen del hecho que la gramática se fue trabajando junto al compilador lo que resulto en algunas limitaciones con respecto a la expresividad del lenguaje.

Otras de las áreas de mejora del compilador son con respecto al análisis, ya que capta los errores más comunes sin embargo aún existe bastante probabilidad de que el usuario termine pudiendo compilar programas que son incorrectos y que al momento de ejecutarlos no funcionaran. El lenguaje ya es de tipado estático por lo que ya ofrece suficiente información para poder agregar técnicas de análisis estático más avanzadas que pueda captar de forma más completa la amplia gama de posibles errores puede crear el usuario al momento de escribir programas.

ÍNDICE

ANEXOS

ÍNDICE DE FIGURAS

SIGLAS

3AC:	Tree Address Code (Código de tres direcciones)
API:	Application Programing interface (interfaz de programación para aplicaciones)
AJAX:	Asynchronous JavaScript And XML (XML y JavaScript Asíncrono)
AST:	Abstract Syntax Tree (Árbol de Sintaxis Abstracto)
DSL:	Domain Specific Language (Lenguaje Específico del Dominio)
DLL:	Dynamic Linked Library (Liberia Dinámicamente enlazada)
IDE:	Integrated Development Environment (entorno de desarrollo integrado)
HTML:	HyperText Markup Language (Lenguaje de Marcas de Hipertexto)
CSS:	Cascading Style Sheets (Hojas de Estilo en Cascada)
FQDN:	Fully Qualify Domain Name (Nombre de Dominio Completo)
DOM:	Document Object Model (Modelo de Objetos de Documento)
JSON:	JavaScript Object Notation (Notación de Objetos de JavaScript)
XML:	Extensible Markup Language (Lenguaje de Marcado Extensible)
JSX:	JavaScript XML
UI:	User Interface (Interfaz de Usuario)
SFC:	Single File Components (Componente de Archivo Único)
SCSS:	Sassy Cascading Style Sheets (Hojas de Estilo en Cascada Atrevidas)
SDK:	Software Development Kits (Kits de Desarrollo de Software)
HTTP:	HyperText Transfer Protocol (Protocolo de Transporte de Hiper Texto)

NOMENCLATURA

C#-P#-N#-L#-A#:	Número de Componentes, Propiedades, Nodos, Niveles, Atributos respectivamente
N#-L#:	Número Nodos y Niveles C se asume 1, P y A se asumen cero.

CAPITULO 1. INTRODUCCION

1.1 Contexto de la investigación

En sus inicios, la tecnología web consistía solo en HTML (HyperText Markup Language), un lenguaje de marcado diseñado para organizar información textual en Internet, otorgándole capacidades muy novedosas y útiles. Algunas de las características consisten en poder estructurar la web haciendo uso de etiquetas, estas son los bloques básicos de HTML que permiten indicarle al navegador cómo debe verse y comportarse cada parte que conforma el documento. Las etiquetas también permiten la incorporación de multimedia al documento, como las imágenes o la posibilidad de enlazar otros documentos a través de hipervínculos. Para poder ir de un documento a otro de forma sencilla (Crowder & Crowder, 2008).

El sistema de etiquetas en HTML ofrece una forma accesible de estructurar y componer documentos informáticos sin saber programar. Esta facilidad de uso de HTML propició su rápida popularización, atrayendo a una creciente comunidad de usuarios que adoptaron este nuevo formato para organizar la información y el contenido. La idea de escribir etiquetas que actúen como pequeños bloques con los que ensamblar el documento resultó ser muy atractiva para los usuarios que mantienen cada vez más contenido que ahora se encontraba disponible en páginas HTML.

En respuesta a la necesidad de una mejor presentación del documento, se introdujo CSS (Cascading Style Sheets). Esto proporcionaba una revolucionaria capacidad para dar estilo a los documentos HTML, separando así el contenido del documento de la presentación. Esto permite diseños con colores y estructuras que no solo mejoran la experiencia visual, sino que también facilitan la navegación del lector a través del contenido. CSS tenía la flexibilidad para declarar estilos que se apliquen a grupos de elementos que cumplan algún criterio específico con respecto a su ubicación relativa en el documento sin la necesidad de escribir múltiples veces los mismos estilos (Olsson, 2014).

Las cosas que CSS permitió modificar en sus primeras versiones fueron los espaciados, gracias a la introducción del modelo de caja se volvió posible incluir separaciones entre elementos, esto ayudaba bastante a evitar páginas con el contenido demasiado concentrado en ciertos lugares. Otra cosa útil es que estos contendores pudieran modificarse para cambiar sus propiedades, como el largo o el ancho, y colocar colores al texto o al fondo de la página para personalizar el documento. Además, entre sus funciones estéticas, CSS permite colocar elementos de manera relativa, lo que resulta en documentos más legibles y organizados y que además se adaptan mejor a los distintos tamaños de pantallas. Durante la década de los

90, en respuesta a una creciente demanda por parte de los usuarios, surgió la necesidad de dotar a los documentos HTML de mayor interactividad. En aquel entonces, HTML demostraba ser extremadamente limitado para gestionar las acciones del usuario. Algunos de los navegadores web de la época, entre ellos Netscape, probaron implementar un sistema de scripting que permitiera la creación de páginas web con contenido más dinámico e interactivo. Este sistema, que eventualmente tomaría el nombre JavaScript, se erigió como el último pilar fundamental en el desarrollo web. Su objetivo principal era convertirse en la norma mediante la cual los programadores podían infundir interactividad a sus documentos, prescindiendo de la dependencia directa del navegador utilizado por el usuario (Frisbie, 2023).

Durante la inclusión inicial de JavaScript, algunas características destacadas eran que se ejecutaba en la computadora del usuario, por lo que no dependía exclusivamente de las interacciones con el servidor para poder realizar acciones, lo que minimizaba los viajes completos de ida y vuelta para actualizar elementos del documento HTML. También se introdujo la capacidad de manipular el DOM (Document Object Model), lo que permitía la creación, eliminación y modificación de elementos en la página web. Esta funcionalidad dinamizó el comportamiento de las páginas y minimizó la necesidad de que el usuario recargara en repetidas ocasiones la página solo para observar los cambios ocurridos luego de alguna interacción como hacer clic en un botón.

Durante la década de los 2000, esos 3 estándares de la web HTML, CSS y JavaScript continuaron adaptando nuevas características, mejorando o reemplazando otras y añadiendo compatibilidad con los nuevos dispositivos que iban surgiendo. En la primera mitad de la década de los 2000 el uso de JavaScript aumentó significativamente para manejar la funcionalidad de la página del lado del usuario junto al incremento de funcionalidades y características que agregan los navegadores al lenguaje, esto derivó en aplicaciones cada vez más interactivas, complejas y con problemas de compatibilidad entre navegadores, esto frustraba a los desarrolladores, ya que algunas funciones muy utilizadas como modificar elementos HTML, animaciones, estilos dinámicos y manejar la interacción del usuario se volvieron muy engorrosas.

Los desarrolladores del momento debían procurar compatibilidad de navegadores, puesto que ya habían aparecido navegadores que se utilizan en la actualidad como Mozilla Firefox y Google Chrome. La razón tenía que ver principalmente con que la implementación de ciertas funciones comunes difería, ya sea en los tipos de parámetros, funciones con nombres diferentes, y esto generaba un entorno donde algunas páginas web funcionaban y se veían bien en algunos navegadores, pero no en otros. Adicionalmente, algunas características eran exclusivas de ciertos navegadores, por lo que algunas páginas no se podían usar en toda su extensión o directamente se rompían, lo que también resultaba frustrante para los usuarios.

1.1.1 Librerías

Esta situación de incompatibilidad y funcionalidades del navegador con nombres muy largos de usar, sin mencionar su diseño poco eficiente al momento de escribir código propiciaron la aparición de librerías. Las librerías son un conjunto de funcionalidades que se utilizan para agilizar tareas para las cuales estas fueron específicamente diseñadas. En líneas generales las librerías son creadas con el propósito de facilitar procesos a través de abstracciones o de la escritura previa de código que se utiliza frecuentemente.

Las librerías no son un concepto introducido o creado por JavaScript, no obstante, fue en el periodo de mitad de la década de los 2000 donde aparecieron librerías que se volvieron muy populares e influyentes en la historia del desarrollo web. Estas librerías ofrecían unas funcionalidades que eran difíciles de gestionar directamente con JavaScript o que este no poseía una forma nativa de hacerlo. Dichas funcionalidades incluían manejo de consultas AJAX (Asynchronous JavaScript And XML) para la interacción a posteriori con servidores, manipulación del DOM y gestión de eventos, creación de interfaces gráficas o incorporación de patrones de programación.

Fue hasta en 2006 que JQuery apareció, esta es sin duda una de las librerías que sería más influyente en el desarrollo web. JQuery se destacó principalmente por su facilidad de uso y sintaxis amena, facilitando alguno de los puntos más dolorosos a la hora de trabajar con JavaScript como lo son la manipulación del DOM y a homogeneizar las funciones disponibles en cada navegador (Nicholus, 2016).

Otra de las características novedosas que incluía JQuery fue la habilidad de expandir su ya amplia cantidad de funcionalidades con nuevas a través de plugin en librerías de terceros. Estos plugins cumplían numerosas funciones, entre las más comunes destacaban animaciones de elementos en la página web, llamadas AJAX a servidores y su respectivo manejo de la comunicación por HTTP (HyperText Transfer Protocol), interactividad del DOM y creación de interfaz gráfica junto a librerías de CSS como Bootstrap.

JQuery aporto mucho al espacio de interfaces gráficas programáticas con JavaScript, es decir crear las interfaces gráficas en tiempo de ejecución de acuerdo con un programa y reaccionando a ventos como clic a botones por parte del usuario. Sin embargo, aún se tenía que programar todas estas interacciones de manera manual lo que probaba ser una tarea repetitiva y que consumía tiempo. Además, crear interfaces de esta manera era muy flexible debido a todas las interacciones que se podían programar, pero tenían el costo de ser tardadas de implementar debido a que mantener en sincronía lo que el usuario veía con los datos del programa es particularmente complicado.

1.1.2 Librerías de UI

Entre los años 2010 y 2012 surgieron varias librerías que se especializaban en la creación de interfaces gráficas, estas librerías como AngularJS (Bampakos & Deeleman, 2023) y React introdujeron conceptos muy revolucionarios que ayudaron a lo que eventualmente se convertiría en el patrón de componentes.

El concepto fundamental bajo el que trabajaban estas librerías y muchas otras que se sumaron después fue el de componentes, definido como una unidad de código reutilizable que encapsula en cómo debe de comportarse, verse e interactuar con el usuario. Estos componentes son elementos que basta con definirlos una única vez y pueden ser reutilizados en muchos lugares (Rippon, 2023). Para poder lograr esta flexibilidad los componentes internamente manejan 3 cosas importantes.

Lo primero que permite que un componente pueda reutilizarse es la parametrización con propiedades. Las propiedades de un componente son piezas de datos o valores que le indican como debe comportarse, desde indicar el color de un elemento, un texto a mostrar en pantalla o un número que indique algún estado. Las propiedades permiten pasar datos desde fuera y que este reaccione de manera distinta, pero determinística dependiendo de los valores que se use, adaptándose así al contexto donde es usado.

Lo segundo que conforma un componente es el uso de un estado local. En desarrollo web se le denomina estado al conjunto de datos que describen como se encuentra el componente en un momento dado. Esta información es vital para poder reconstruir el componente y mostrárselo al usuario de siempre en sincronía con esto datos. La incorporación de los mecanismos automáticos de estas librerías para actualizar la interfaz gráfica supuso un incremento en la productividad de los desarrolladores, ya que ahora solo debían enfocarse en la información que cambia dinámicamente y estas librerías se encargaran de actualizar la UI (User Interface) de manera automática minimizando los errores con respecto a una implementación manual.

El tercer elemento de los componentes es la plantilla, esta describe el HTML que cada componente debe generar y que será visto por el usuario en su navegador. La parte crucial es que esa plantilla no necesariamente es estática, sino que puede mostrar valores dinámicos ya sea de las propiedades o del estado utilizando interpolación. La interpolación es la técnica de buscar un patrón predeterminado y luego sustituirlo por el valor real de que representa. Otra importante función de la plantilla es el enlace de datos, esto significa que cuando un dato cambia en la interfaz, este cambio se reporta y actualiza los valores en el programa, a través de esos dos mecanismos se puede mantener en sintonía la interfaz con el código que la genera de una forma más sencilla de entender y con mucho menos código y trabajo manual.

1.1.3 Frameworks

Las librerías de UI demostraron optimizar la forma en que se escribía software que renderizara y mantuviera en sincronía la interfaz gráfica con el resto del programa, lo hacían bien, pero no solucionaban los desafíos del desarrollo de aplicaciones basadas en componentes. Ciertos patrones se empezaron a generalizar entre las aplicaciones web, esto resultaba en la implementación múltiple de las mismas características en diversas aplicaciones (Bampakos & Deeleman, 2023).

Para reducir la carga de los desarrolladores y la necesidad de implementar las mismas características se añadió un nuevo nivel de abstracción, los frameworks. Los frameworks al igual que las librerías se enfocan en un dominio, es decir, realizar una tarea en concreto, como por ejemplo la administración de interfaces gráficas. La deferencia radica en que los frameworks son soluciones más completas a ese problema porque no solo proporcionan las abstracciones y código que sirve para la ejecución de la tarea en cuestión.

La completitud de un framework proviene de ser una colección de técnicas, código y abstracciones que evitan el manejo directo del funcionamiento a bajo nivel del programa y en su lugar exponen una forma declarativa para que los desarrolladores puedan implementar sus aplicaciones. Por lo general los distintos frameworks aportan múltiples librerías que manejan aspectos diferentes de una aplicación, pero que funcionan en conjunto para ofrecer una experiencia coherente y funcional sin necesidad de configuración o programación adicional. En ese sentido un frameworks se puede pensar como el pegamento que mantiene todas las piezas unidas y en su lugar.

Con la incorporación de los frameworks en el panorama del desarrollo web y dado que todos poseen prácticamente las mismas características, el enfoque de los programadores paso de decidir como implementar todas las funcionalidades deseadas en la aplicación a escoger un framework. La elección del framework se basa en parámetros como el soporte de la comunidad, la documentación y la incorporación de algún tipo de mecanismos que ayude con la orquestación de múltiples componentes en aplicaciones con cada vez más complejidad y cantidad de funciones.

Algunos de los frameworks más populares fueron Angular (diferente de la librería AngularJS) y Vue, estos poseían implementaciones nativas de navegación, manejo de rutas, inyección de dependencias, separación de la aplicación en módulos e integración con CSS. Otros igualmente populares como React no poseían tantas características, pero todos tenían alguna forma de manejar la reactividad, es decir los cambios en el estado del componente derivado de las acciones de un usuario o evento que ocurre en el programa.

1.1.4 JavaScript

JavaScript fue el primer lenguaje de programación para la web, en este se han creado incontables librerías, frameworks y aun incluso más aplicaciones. Aposto por ser una versión más simplificada de lenguajes de programación que eran populares en la época como Java o C (Frisbie, 2023). JavaScript tomaba inspiración de dichos lenguajes, principalmente de su sintaxis, que es la forma en que se escribe el código, pero simplificando mucho el cómo estos funcionan, algunos de las diferencias importantes son.

Es un lenguaje interpretado, lo que significa que el navegador interpreta el código y actúa según las acciones descritas en el script; por otro lado, lenguajes como Java necesitaban compilarse, lo que significa que el código fuente no es lo que se ejecuta, sino la representación binaria de las instrucciones equivalente. Estas instrucciones binarias son el resultado del proceso de compilación.

Tampoco tenía un sistema de tipado estático, significa que las estructuras donde se guardaba la información podían cambiar de forma diferente sin validación. Lo que originalmente permite la flexibilidad para la creación rápida de scripts demostraba ser un dolor de cabeza al intentar gestionar en aplicaciones con códigos fuente cada vez más extensos.

Otra razón por la que trabajar en JavaScript resultaba confuso es que posee un sistema de coerción de tipos, que permite a JavaScript convertir automáticamente datos de un tipo a otro, por lo que el resultado de algunas operaciones variaba según el orden de los factores, generando dificultades para comprender la operación precisa del programa.

Independientemente de sus defectos, JavaScript fue mejorando con el paso del tiempo, agregando características más consistentes y robusteciéndose para poder adaptarse mejor al desarrollo web del momento. Dichos cambios en la implementación se encuentran bajo el estándar ECMAScript que los navegadores incorporan en sus nuevas versiones (Frisbie, 2023).

Muchos de los cambios y mejoras incorporados al lenguaje provenían de patrones en otros lenguajes de programación que tenían más madurez y otras provenían directamente de algunos de sus competidos. Existieron múltiples alternativas que trataban de mejorarlo, estas proveían una sintaxis adicional o parecía a la regular, pero corrigiendo aquellas partes donde JavaScript flaqueaba, a partir de este script con mejoras se podía compilar a código JavaScript y ejecutarlo en el navegador web. Existieron muchos scripts personalizados basándose en JavaScript, pero sin duda el de mayor influencia fue TypeScript.

1.1.5 TypeScript

TypeScript es un superconjunto de JavaScript, eso implica que todo código escrito en JavaScript es a su vez código TypeScript válido, pero el caso contrario no es cierto. La razón de esto es que TypeScript no crea una sintaxis novedosa diferente a la de JavaScript, sino que en su lugar la expande con constructos y definiciones más elegantes para adornar el lenguaje con información de los tipos de datos (Rippon, 2023).

La forma en que TypeScript logra esto es integrando anotaciones de tipos en las funciones y variables, esto le proporciona metainformación al compilador sobre la forma de la información y así puede evaluar de manera precisa si el valor de se ha asignado a una variable o parámetro corresponde con el tipo de valor declarado por el programado. Estas evaluaciones de corrección se llevan a cabo durante la compilación y si se encuentran errores estos son reportados al usuario, ayudando así a la detección temprana de errores.

TypeScript también expandía el lenguaje añadiendo estructuras como las clases, que son formas de incorporar datos y funciones en una sola unidad de código encapsulada, las clases son la unidad fundamental para poder trabajar con orientación a objetos. Las clases incorporan también un sistema de visibilidad, similar al de Java con los modificadores de visibilidad, permitiendo así a los desarrolladores ocultar detalles de la implementación las clases si así les parecía conveniente (Bampakos & Deeleman, 2023).

JavaScript no poseía ninguna forma de poder delimitar los alcances de variables o funciones por lo que era común que mientras más grande se volvía la base de código, más posibilidades existían de causar un conflicto debido a dos elementos llamados de la misma forma. Para evitar esto TypeScript introdujo el concepto de nombres de espacio permitiendo así evitar estos conflictos siempre y cuando dichos elementos se encontraran en nombre de espacio distinto. Esto promovía la reutilización y organización del código.

Estas características que ofrecía el lenguaje les parecieron muy atractivas a los desarrolladores, rápidamente se encontró que mejoraba bastante la experiencia de desarrollo, esto se vio en su nivel de adopción que comenzó a crecer rápidamente, al punto que era incluso el lenguaje por defecto de algunos frameworks, como el caso de Angular. Otros frameworks también empezaron a ver las ventajas que ofrecía y cambiaron sus bases de código para incorporar TypeScript al desarrollo del framework como tal y al mismo tiempo permitirles a los programadores que usaban el frameworks habilitar sus proyectos de manera opcional el uso de TypeScript.

1.1.6 NodeJS

Por muchos años JavaScript se podía ejecutar únicamente dentro de un navegador web, eso cambio a mediados del año 2009 con la salida de NodeJS. Este era básicamente el motor de JavaScript que se ejecuta dentro de los navegadores basados en Chromium para procesar JavaScript. Lo que ocurrió fue que se aisló esta parte llamada V8 en su propio programa y se convirtió en el primer entorno de ejecución de JavaScript por fuera de un navegador web (Buckler, 2022).

La introducción de NodeJS permitía ejecutar scripts de JavaScript desde la terminal de una computadora y que estos tuvieran accesos a distintas API (Application Programing Interface) y funcionalidades del sistema operativo que simplemente no tiene equivalente en el navegador. Algunas de estas capacidades nuevas eran la lectura y escritura de archivos, manejo de código de red, intercomunicación de procesos y carga de librerías nativas generadas en lenguajes como C/C++ (Casciaro & Mammino, 2020).

Con estas nuevas capacidades en JavaScript gracias a NodeJS empezaron a surgir nuevos tipos de aplicaciones, las "Build tools" o herramientas de construcción que buscan automatizar de forma más sencilla y eficiente los varios procesos que los desarrolladores han estado implementando para optimizar sus aplicaciones como la minificación de contenido, concatenación de archivos, post procesamiento de CSS entre otros (Pillora, 2014).

Una de las herramientas más populares fue Grunt, este introdujo el desarrollo frontend moderno, ya que más que simplemente automatizar las tareas repetitivas estandarizó la forma de describir y programar tareas ayudando así a una replicabilidad y adaptabilidad. Además, Grunt permite integrar otras tareas como compilación, optimización de imágenes, code splitting y varias otras técnicas que le permiten al desarrollador aplicaciones más sofisticadas y modernas en el ambiente competitivo de la web.

Las herramientas de construcción resultaron muy útiles gracias a la automatización y replicabilidad de sus resultados (Pillora, 2014), NPM un programa desplegado junto a NodeJS permitía la administración y resolución de dependencias de una manera muy práctica, bastan un par de comando en la terminal y ya se tenía integrada la librería junto al resto del código. Esta facilidad de incorporación de librerías fue lo que facilito la popularización de las librerías de UI y junto a las herramientas de construcción se podían iniciar proyectos en minutos con las configuraciones adecuadas que al final del proceso generaban los archivos de JavaScript que se enviarían eventualmente al navegador de los usuarios. Hacer las cosas de esta manera significaba menos tiempo resolviendo problemas entre librerías y más tiempo programando la aplicación.

1.1.7 Plantillas declarativas

Entre los años 2010 y 2013 empezaron a popularizarse los frameworks como React o Angular con un aporte trascendental, poder definir la interfaz gráfica de HTML de manera declarativa a través del uso de una versión extendida de XML (Extensible Markup Language) llamada JSX (JavaScript XML) en React o usando Angular templates en el caso de Angular. Estas plantillas permitían crear enlaces de datos entre el código de la aplicación y el código de UI, evitando escribir el código que maneja las interacciones de bajo nivel gracias a los compiladores que preprocesan las plantillas y generan el código de manera automática.

Esto facilitó mucho el desarrollo de aplicaciones para los desarrolladores, ya que elimina la necesidad de manejar elementos del DOM manualmente o de administrar colecciones individuales en la página. Con este nuevo paradigma de componentes en las plantillas era mucho más eficiente dividir la aplicación en bloques con tareas específicas y aislados que lidiar con toda la página al momento de hacer actualizaciones en el contenido de esta.

Los frameworks introdujeron muchos beneficios al hacer el trabajo duro por los desarrolladores, sin embargo, para poder llegar a este grado de sofisticación muchos necesitaban de herramientas de construcción cada vez con más configuración. En el caso particular de React (izquierda) como se muestra en Figura 1 necesita de un proceso de compilación para poder traducir la sintaxis de JSX en llamadas a la API de React en JavaScript (derecha), que es algo que el navegador web sí puede ejecutar (Rippon, 2023).

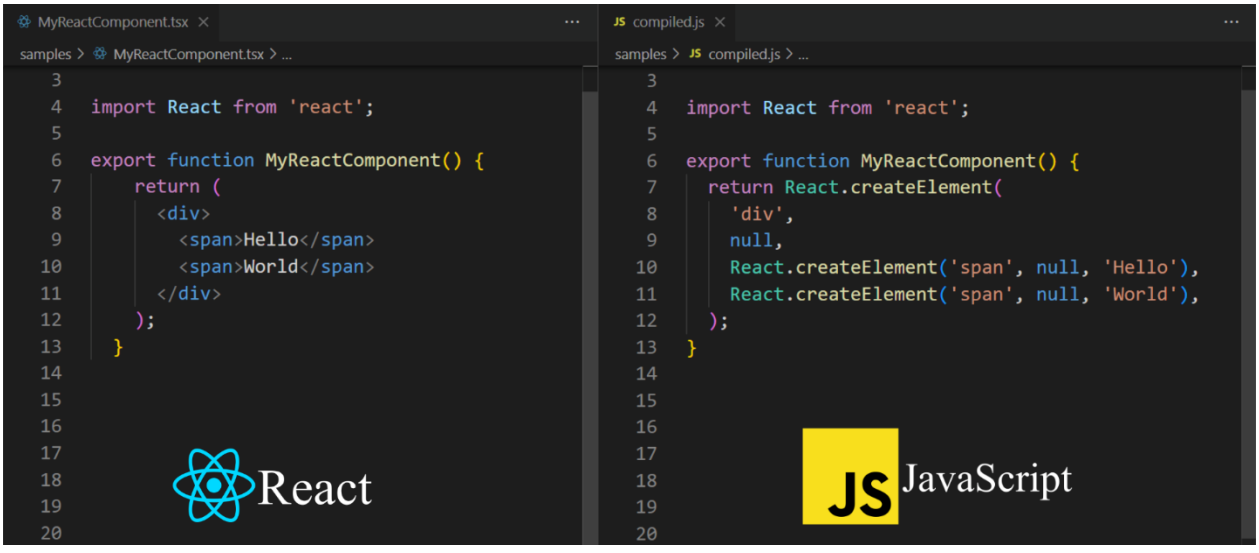

Figura 1 Código React compilado Fuente: Creación propia

Esto significa que el código fuente que se escribe en React no se puede ejecutar directamente en el navegador, sino que debe pasar por al menos una fase de compilación para poder llegar a un código JavaScript que sí puede interpretar el Navegador. Aunque el ejemplo utilizado es React el resto de frameworks posee el mismo problema intrínseco al sustituir JavaScript imperativo, especificando instrucción por instrucción que debe hacerse en favor de alguna sintaxis personalizada y declarativa.

1.1.8 Toolchains

Para mitigar esta desventaja muchos de los creadores de los frameworks también crearon las Toolchains o cadena de herramientas. Estas actúan en conjunto para poder llevar el código del frameworks de su forma inicial hasta su resultado final en archivos de JavaScript. Desde la terminal se percibían como un único proceso transparente al usuario. La idea fundamental era ofrecer herramientas de poca configuración que los usuarios de los frameworks podían utilizar de manera sencilla y los autores de estas cadenas de herramientas podían hacer cambios en el proceso de construcción de la aplicación tras de escenas sin que los usuarios necesitaran saber estos detalles.

Uno de los programas más importantes dentro de la toolchain es el bundler, ya que es este el que inicia todo el proceso y organiza a los demás programas para que se ejecuten de manera ordenada y correcta produciendo así el resultado esperado. Uno de los blunders más utilizados es WebPack (Rippon, 2023), este forma parte de 'CreateReactApp' la toolchain oficial de React por muchos años y se volvió tan popular debido a su fácil personalización y adaptabilidad gracias al uso de plugins que se podían combinar para obtener múltiples transformaciones, similar al caso de los task runners, pero tienen el agregado que le permite empaquetar módulos, analizar dependencias y hacer treeshaking para optimizar el resultado final.

El ecosistema de los frameworks y toolchains evolucionó muy rápido y caótico debido a que diferentes herramientas que surgieron muy cercanas en el tiempo. Múltiples estándares surgieron y empezaron a competir entre sí. Particularmente el tipo de desarrollador más afectado eran los autores de librerías para interfaces gráficas porque para obtener los mismos resultados visuales deben implementar básicamente el mismo código, pero con la sintaxis particular de cada framework. Otro ejemplo muy frustrante es la incompatibilidad entre distintas versiones de módulos de JavaScript porque algunos programas requieren la presencia de módulos commonJS y otros soportaban ESNext o ambos (Casciaro & Mammino, 2020). Los autores de librerías deben lidiar no solo con las particularidades de cada framework sino también con las de las herramientas que los sustentan. Otro punto importante es la falta de documentación clara para crear librerías, ya que cada framework se enfoca más en los desarrolladores de aplicaciones no de librerías.

1.2 Justificación de la investigación

El desarrollo de aplicaciones web modernas es un ámbito muy confuso, ya que existen multitud de opciones para escoger al momento de desarrollar una aplicación. Lo que originalmente hubiese sido únicamente escribir HTML, CSS y JavaScript se ha inundado con tantas opciones para los desarrolladores, que estos suelen sentirse perdidos frente a las abundantes decisiones de qué tecnologías conformarán sus aplicaciones.

Para los desarrolladores tantas elecciones son complicadas de administrar. Es ahí donde este proyecto pretende simplificar las cosas ofreciendo una sintaxis unifica para poder definir la interfaz gráfica de los componentes y estilos CSS con los que se debe mostrar en la página en el lenguaje de dominio específico llamado Haibt. Además, la incorporación de una sintaxis personalizada permitiría reducir la verbosidad de JavaScript o de las plantillas personalizadas de los distintos frameworks.

Esto ayudaría agilizando el desarrollo de librería de UI, ya que si se pretende llevar la misma interfaz gráfica a distintas plataformas basta con declararlo una sola vez. Existen otros casos de uso que, aunque menos frecuentes son igual de problemáticos, este es el caso de las aplicaciones que por alguna razón están migrando de un framework a otro, poseen partes programadas en distintos frameworks o los SDKs que por lo general se buscan distribuir para ser utilizados en múltiples plataformas.

Por esta razón al implementar un DSL (Domain Specific Languaje) que pueda abstraer la dependencia directa de un framework y que al mismo tiempo el compilador del lenguaje pueda reducir la necesidad de escoger y configurar varias técnicas detrás de la creación de librerías estas se podrían disponer en múltiples frameworks ofreciendo las mismas características y que se mantenga homogéneo. Para los autores esto significa que ya no deben preocuparse por los detalles configuración, particularmente esto beneficiaria a los más inexpertos.

1.3 Alcances de la investigación

Se definirá un lenguaje de programación personalizado, diseñando su sintaxis de tal forma que sea sencillo incorporar las tres partes de la web, estos siendo la estructura, decoración y scripting. Este lenguaje se clasificará como lenguaje de dominio específico, siendo su dominio la creación de componentes para interfaz gráfica en la web. El lenguaje se enfocará en poder definir la estructura HTML de una página web y usar estilos CSS de forma conjunta.

También se incorporará una forma de poder definir estructura en sus datos, por eso se incorporará un sistema de tipos en el lenguaje. Esto habilitará a los desarrolladores a poder definir estructuras de datos según las necesidades de sus programas. La revisión de este tipado será de manera estática y estructural lo que significa que se determina la corrección de una estructura de datos basada en los miembros que la componen al momento de compilarlo.

El lenguaje tendrá también construcciones específicas como la declaración de componentes, la declaración de estilos CSS y la definición de variables, esto implica que poseen una sintaxis dedicada. Otros aspectos como las plantillas de los componentes también tendrán su sintaxis dedicada incluyendo funciones básicas como el renderizado condicional e interpolación de expresiones.

Se propondrá un compilador que pueda tomar código fuente de este lenguaje y haga uso de un parser generado con el cual se construirán todas las estructuras que dicho compilador necesite para validar el programa. Este compilador contara con dos módulos que permitan la generación de código TypeScript adaptado para incorporar una salida a código de React y otra a VueJS que son dos de las tecnologías más populares para la creación de aplicaciones web.

Las expectativas de los desarrolladores con respecto a lenguajes de programación normalmente incluyen funcionalidades que mejoran la experiencia como autocompletado, sintaxis coloreada, auto formato y otras formas de integración con el IDE. Sin embargo, estas no son funciones que formen parte de un proceso de compilación. En su lugar se implementará un compilador que procese directamente los archivos de código fuente. Todas las demás son características ayudan a la productividad de los desarrolladores, pero no son funciones vitales del compilador así que serán omitidas.

1.4 Limitaciones

La gramática del lenguaje será diseñada para ofrecer formas expresivas de declarar las distintas estructuras de este, sin embargo, el enorme bagaje de los desarrolladores en lenguajes como JavaScript importa bastante. Por esta razón la sintaxis debe evitar alejarse mucho de una gramática típica de un lenguaje de programación. De otra forma es posible que los desarrolladores tengan problemas aprendiendo el lenguaje.

Por una razón similar se optará por utilizar autogenerar el parser y lexer, dado debido a que la gramática pasará por varios cambios sustanciales antes establecerse, esto evita volver realizar el mismo trabajo.

Lamentablemente esto también significa que hay poco control sobre como estos elementos autogenerados funcionan y esto podría limitar la flexibilidad para incorporar características o manejar errores durante el proceso de análisis en el compilador.

Dado que se deben implementar como mínimo dos plugins para poder generar código en dos frameworks y comprobar que en efecto Haibt puede ser traducido a múltiples de estos, se presenta la limitante de no tener expertos para cada uno. Por esta razón es posible que los plugins generen código no óptimo o desfasado, debido a que cada framework se mueve a su propio paso incorporando o cambiando características.

Por último, el tiempo para desarrollar la gramática, el compilador y los plugins fue bastante justo dado el tamaño del equipo. La cantidad de tiempo disponible impacto en el número de experimentos que se podía probar para una idea y las características que se implementaron en el compilador.

1.5 Objetivo general

- Implementar un lenguaje de dominio específico (DSL) que unifique elementos del proceso de codificación como interfaz gráfica, estilos CSS e interactividad con scripts en la creación de componentes web en un archivo de texto que sea compilado y procesado para generar código en múltiples frameworks conservando su funcionalidad.

1.6 Objetivos específicos

- Definir el lenguaje Haibt como otra opción para los autores de librerías que necesiten portar sus creaciones a múltiples frameworks, incorporando características que permitan el uso de estilos, HTML y scripting de manera fluida.
- Formular una gramática de tipo LL 1 de tal forma que el lenguaje compilado sea relativamente fácil de analizar, pero manteniendo una sintaxis similar a lo esperado en los lenguajes modernos.
- Incorporar un sistema de tipado estático dentro de Haibt que habilite un análisis más exhaustivo del código, el tipado será estructural para poder adecuarse a la naturaleza de JavaScript.
- Desarrollar plugins que permita la generación automática de código tanto para React como para Vue a partir de su representación intermedia con características como interpolación de expresiones y renderizado condicional en el compilador de Haibt.

1.7 Antecedentes

1.7.1 Mitosis

Existe un proyecto de código abierto muy similar en cuanto a funcionalidad, se conoce como Mitosis. El concepto de Mitosis es el mismo, utilizar una definición única de componente y a partir de esta generar el código equivalente en más de una docena de proyectos distintos, la mayoría son frameworks de JavaScript, pero otros son tan diversos como una salida a archivos de Figma que pueden ser vistos y editados por un diseñador. El proyecto comenzó a mediados del año 2020 y ha tenido un desarrollo activo desde entonces.

Mientras que su propósito es el mismo que el de este proyecto, Mitosis aborda el problema de manera muy diferente. Para empezar, Mitosis utiliza JavaScript como su lenguaje de programación, por lo que los desarrolladores no tienen que aprender un lenguaje nuevo. Esto significa que utiliza la misma sintaxis en su entrada y en su salida. Mitosis básicamente realiza un análisis del código JavaScript y basándonos en este determina qué propiedades, variables e interfaz el componente implementa. Tras este análisis, Mitosis genera un archivo JSON (JavaScript Object Notation) donde se describen estas características del componente. Una vez este archivo JSON es generado, ya es posible procesarlo con el compilador de Mitosis y generar el código esperado en la salida especificada de alguna de las soportadas (GitHub, 2/09/2023). EL flujo de las entradas ocurre como indica la figura 2.

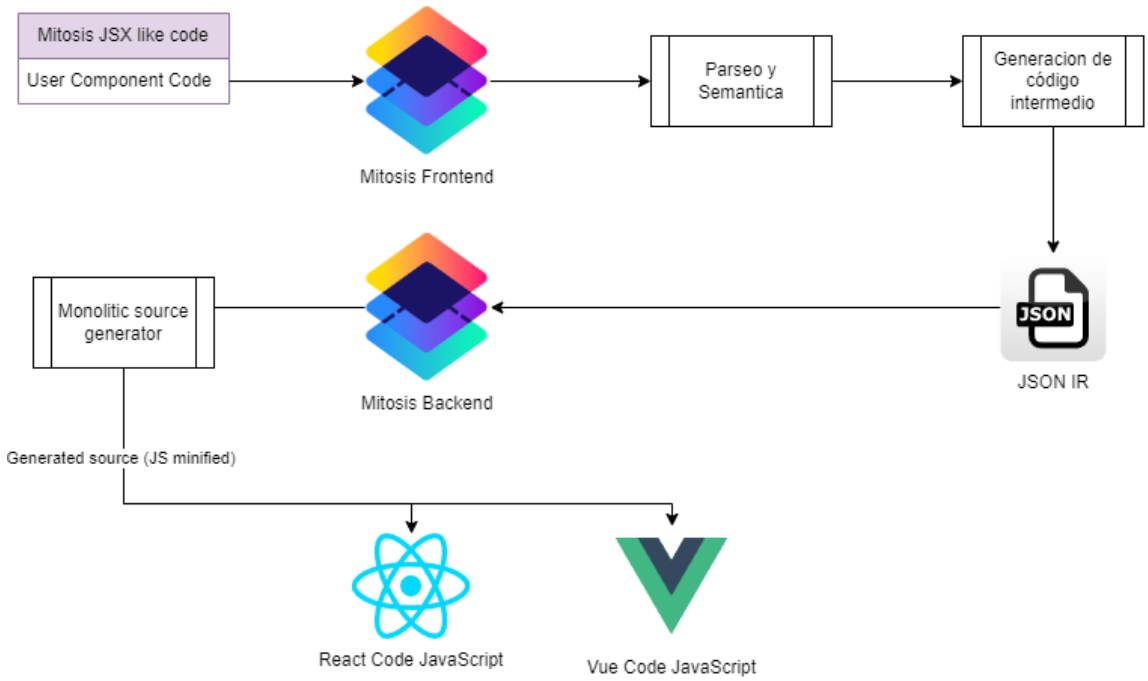

Figura 2. Proceso compilación Mitosis Fuente: Creación propia

La diferencia clave con Haibt es que este utiliza su propio lenguaje y sintaxis para poder definir estos elementos de manera más clara y entendible a los desarrolladores. Al ser un lenguaje precisamente diseñado para poder identificar las partes del componente de manera fácil con una mirada rápida.

Otra diferencia clave es que Mitosis utiliza una sintaxis inspirada en React y muchos de los mecanismos que React utiliza se definen de manera similar. Esto podría también introducir algunas limitantes en el sentido que se apega demasiado a React, lo que significa que hereda sus bondades, pero también sus fallos. Por su lado, Haibt al desarrollar su sintaxis propia, trata solamente de tomar lo mejor en términos de usabilidad y características de cosas que otros frameworks aparte de React, han introducido y fueron adicciones bienvenidas por parte de la comunidad de desarrolladores. Se muestra el flujo de entradas y salidas para el compilador de Haibt en la Figura 3.

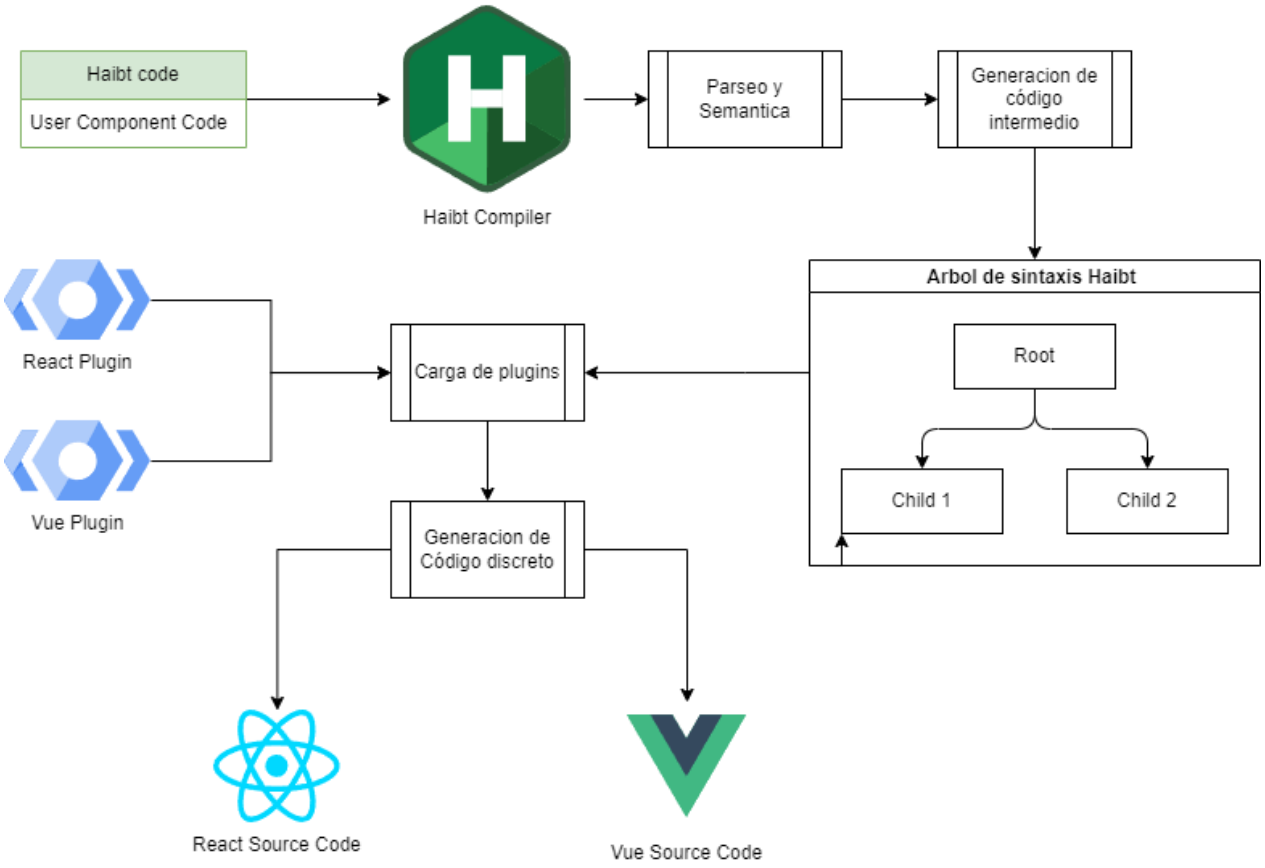

Figura 3. Proceso de compilación Haibt Fuente: Creación propia

Aparte de Mitosis, no se tiene conocimiento de ningún otro proyecto que tenga un abordamiento parecido. La mayoría de los proyectos en el ecosistema tratan de ser un reemplazo de otros proyectos existentes en lugar de proporcionar una unificación que simplifique el desarrollo de aplicaciones web.

1.7.2 Componentes web

Otra tecnología que podría ser utilizada para fines parecidos es el estándar de componentes web. Este estándar aglomera varias API en el navegador que pueden ser utilizadas desde JavaScript. Con estas API se puede modelar y definir componentes con encapsulamiento, CSS y descripción de la interfaz usando HTML, de manera similar al resto de frameworks (Web Components 18/09/2023).

La diferencia principal entre utilizar componentes web y un framework como React es que los componentes web se ejecutan de manera nativa en el navegador, eso significa que no hay ningún código adicional o intermedio ejecutándose. Basta con definir el componente web en JavaScript y siempre que este se ejecute en un navegador moderno que soporte componentes web funcionara. En el caso del resto de frameworks, estos envían siempre algo conocido como el runtime. Este es el código encargado de varias tareas, entre ellas ofrecer funcionalidad común, inicializar la aplicación y orquestar los componentes definidos por el usuario para que funcionen de la manera esperada.

Estos pueden ser utilizados desde cualquier código JavaScript, esto permite que los componentes web se utilicen desde dentro de otros frameworks o en aplicaciones que no poseen ninguno, ya que no necesitan ningún runtime. Así es posible que una aplicación hecha en React, pueda utilizar componentes web.

Lamentablemente, los componentes web se incorporaron relativamente tarde dentro de los procesos de desarrollo web, por esto muchas de las tecnologías y frameworks ya existentes no se integran de manera sencilla o directamente no funcionan con estos. Algunas de estas tecnologías que no se relacionan particularmente bien con los componentes web son los componentes de servidor, que son otro tipo de componentes, pero su funcionalidad ocurre en el lado del servidor. Naturalmente, como los componentes web solo funcionan en el navegador, al ejecutarse, estos en otro entorno como NodeJS estos no funcionan.

Otras técnicas como la hidratación, que consisten en restaurar el estado del componente como si este se hubiera ejecutado en el navegador desde el inicio y no previamente en un servidor, no funcionan particularmente bien. En general, si la inicialización del componente o en alguno se ejecute partes de este fuera del navegador, es muy probable que la funcionalidad se rompa. Adicionalmente, los componentes web interactúan directamente con el DOM y la mayoría de frameworks de JavaScript necesitan algún tipo de abstracción o control adicional sobre este, así que para incorporarlos siempre se necesita de código adicional que funcione a manera de un envoltorio o interfaz de compatibilidad entre el framework y el componente web.

CAPÍTULO 2. ASPECTOS GENERALES DE UN LENGUAJE DE PROGRAMACÓN

2.1 Qué es un lenguaje de programación

Los lenguajes de programación son una serie de instrucciones que se escriben en un archivo llamado código fuente y que luego, de una forma u otra, son ejecutados en la computadora. Estas instrucciones, al igual que en el lenguaje natural, poseen forma, es decir, solo hay ciertos patrones de instrucciones que son correctos. Estos patrones determinados se llaman constructos y son lo que lo convierte en un lenguaje formal. Estas instrucciones también tienen un significado que normalmente se traduce en que el programa realice algún cálculo o cambio en un sistema informático.

2.2 Componentes de un lenguaje de programación

Los lenguajes de programación se componen de elementos clave que definen la escritura y ejecución del código. Entre estos se encuentra la gramática, que establece la estructura o sintaxis del código; la semántica, que define el significado del código; los tipos de datos, que clasifican y determinan las propiedades de la información manejada; y las estructuras de control, que guían el flujo del programa. Además, estos lenguajes cuentan con componentes externos que influyen en el comportamiento del código. Uno de ellos es el compilador, que traduce el código de programación a instrucciones directas para la máquina, alternativamente, es el intérprete, que lee y ejecuta el código de manera directa.

2.2.1 Gramática

Como lenguajes formales, los lenguajes de programación tienen una estructura claramente definida, similar a los lenguajes naturales, conocida como gramática. Esta gramática es la responsable de determinar si una secuencia de instrucciones está correctamente formada dentro del lenguaje, ya que establece todos los patrones válidos para organizar dichas instrucciones. Si una secuencia de instrucciones no se acopla a dicho patrón, entonces es sintácticamente inválida.

Existen cuatro tipos de gramática, clasificada con la Jerarquía de Chomsky, donde se agrupan las gramáticas según su complejidad y restricciones (Aho, Lam, Sethi, & Ullman, 2007). Estas son:

Gramáticas Recursivamente Enumerarles (Tipo 0): Son las menos restrictivas y pueden generar cualquier lenguaje que una máquina de Turing sea capaz de reconocer. No tienen restricciones específicas en sus reglas, lo que les permite una gran flexibilidad en la generación de lenguajes.

Gramáticas Sensibles al Contexto (Tipo 1): Estas gramáticas tienen reglas que dependen del contexto en el que se encuentran los símbolos. Es decir, la aplicación de una regla puede variar según los símbolos que la rodean. Esto les permite ser más expresivas que las gramáticas de niveles inferiores, pero también más complejas.

Gramáticas Libres de Contexto (Tipo 2): Son más restrictivas que los tipos anteriores. En estas gramáticas, las reglas se aplican a los símbolos de manera independiente del contexto en el que se encuentren. Son muy utilizadas en la programación y en el análisis de lenguajes naturales, ya que proporcionan un buen balance entre complejidad y capacidad expresiva.

Gramáticas Regulares (Tipo 3): Son las más simples en la jerarquía. Sus reglas producen un símbolo terminal seguido o precedido por un símbolo no terminal. Estas gramáticas son fundamentales en el diseño de expresiones regulares y autómatas finitos, y son adecuadas para describir estructuras de lenguaje simples.

Durante el desarrollo de la gramática para un lenguaje de programación, es común optar por una gramática libre de contexto. Esta elección se debe a que las gramáticas libres de contexto ofrecen un equilibrio ideal entre expresividad en el diseño del lenguaje y sencillez en su análisis. Una de sus ventajas principales es que, para analizar una regla, solo se necesita considerar esa regla en sí misma, sin la necesidad de tomar en cuenta los símbolos adyacentes, a diferencia de lo que sucede con las gramáticas sensibles al contexto. Este enfoque facilita la creación de lenguajes que tienen una estructura flexible y expresiva, pero que al mismo tiempo no resultan excesivamente complicados para ser analizados. Esta característica es fundamental para diseñar lenguajes de programación eficientes y accesibles.

2.2.2 Semántica

A diferencia de la gramática, que se centra en la forma de las instrucciones en los lenguajes de programación, la semántica se ocupa del significado de cada una de estas instrucciones. La semántica determina cómo se ejecutan las instrucciones y cómo se manipulan los datos en el programa (Fowler, 2010). En otras palabras, la semántica se enfoca en el funcionamiento interno del programa, abarcando aspectos como los alcances de las variables, el manejo de errores, el valor resultante de las expresiones y, en términos más amplios, la

corrección y el comportamiento esperado del programa. Este enfoque asegura que no solo se escriba un código que se ajuste a la estructura sintáctica del lenguaje, sino que también se ejecute de manera lógica y coherente con los objetivos del programa.

Por ejemplo, la frase “Un árbol duermen rojo” es sintácticamente correcta en español, ya que sigue la estructura gramatical adecuada. Sin embargo, carece de significado lógico, lo que la hace semánticamente incoherente. De manera similar, en un lenguaje de programación, no todas las instrucciones que están bien formadas desde el punto de vista sintáctico son necesariamente válidas en términos semánticos. Por ello, es crucial diseñar un lenguaje de programación que sea consistente tanto en su sintaxis como en su semántica. Esto asegura que el lenguaje no solo permite escribir instrucciones que se ajusten a sus reglas formales, sino que también describa con precisión las intenciones del programador y ejecute las operaciones de manera lógica y coherente.

2.2.3 Tipos de datos

Todos los programas informáticos se diseñan para realizar operaciones con el fin de procesar datos, convertir unos datos en otros y ejecutar operaciones aritméticas. Fundamentalmente, estos datos se almacenan en algún tipo de memoria dentro de una computadora, y se guardan exclusivamente en formato binario. Sin embargo, es la forma en que estos datos binarios son interpretados por el programa lo que establece diferentes categorías de tipos de datos. Así, dependiendo de cómo se interprete un bloque de memoria, este puede representar un texto, un número, un valor verdadero o falso, una fecha, entre otros (Bampakos & Deeleman, 2023).

Debido a esta diversidad, es esencial que un lenguaje de programación permita el manejo adecuado de estos datos, así como la conversión de un tipo a otro. Además, la mayoría de los lenguajes de programación facilitan la combinación de datos primitivos, como los mencionados anteriormente, en estructuras más complejas conocidas como estructuras de datos. Por ejemplo, un usuario puede ser definido por una combinación de datos primitivos, como el texto de su nombre y el número que representa su edad. Esta capacidad de manipular y combinar diversos tipos de datos es fundamental para crear programas más sofisticados.

Otro aspecto crucial en la programación es el mecanismo que permite la recuperación y almacenamiento de datos bajo demanda, comúnmente conocido como variables. Estas variables se clasifican en dos tipos principales: variables normales y constantes.

Variables normales: Son las más flexibles y comunes. Permiten almacenar un valor y modificarlo o sobrescribirlo en cualquier momento durante la ejecución del programa. Esta capacidad de cambio hace que las variables normales sean herramientas esenciales para el manejo dinámico de datos, permitiendo almacenar y modificar información según sea necesario, como se muestra en Figura 4.

```
Const edad = 16;
```

Figura 4. JavaScript declarar una variable, Fuente: Creación propia

Constantes: A diferencia de las variables normales, una constante almacena un valor que se define en el momento de su creación y no puede ser alterado posteriormente. Las constantes son útiles cuando se necesita un valor que no debe cambiar a lo largo del tiempo de ejecución del programa, como, por ejemplo, el valor de pi (π) en cálculos matemáticos, como en se hace en Figura 5.

```
const PI = 3.1416;
```

Figura 5. JavaScript declarar una constante, Fuente: Creación propia

Las variables, tanto normales como constantes, son elementos fundamentales en los lenguajes de programación debido a la gran flexibilidad y funcionalidad que ofrecen. Permiten a los programadores trabajar eficientemente con datos en diferentes contextos, ya sea para almacenar valores temporales, mantener registros, realizar cálculos o manejar estados dentro de un programa.

2.2.4 Operadores

Así como el álgebra utiliza símbolos especiales para representar sus operaciones básicas, como el '+' para la suma, '/' para la división, o '-' para la resta. Los lenguajes de programación también emplean símbolos o palabras especiales que indican la ejecución de una operación sobre los datos. Estos símbolos, conocidos como operadores, son fundamentales para la construcción de programas.

Los operadores en programación no solo incluyen los aritméticos para realizar cálculos matemáticos básicos, sino también, por ejemplo, en JavaScript, operadores lógicos como '&&' (y) y '||' (o), que son cruciales para implementar decisiones basadas en múltiples condiciones. Además, existe un operador de asignación '=' que permite asignar el valor de un dato a una variable. La habilidad para utilizar estos operadores y crear expresiones es una necesidad primordial en la programación.

Permite a los desarrolladores implementar lógica en sus programas y combinar diferentes operaciones en expresiones más complejas, es decir, se pueden concatenar unas con otras para ensamblarlas. Los operadores tienen orden de precedencia, por lo que es importante respetarlos; de otra forma, las expresiones podrían ejecutarse en un orden no esperado.

2.2.5 Estructuras

Los lenguajes de programación están dotados de estructuras de control que les permiten realizar funciones esenciales, tales como la repetición, agrupamiento y selección sobre un conjunto de instrucciones. Algunas de estas son:

Estructuras de Control Condicional: Permiten ejecutar ciertas instrucciones solo cuando se cumplen determinadas condiciones. La estructura “if-else” es un ejemplo clave, donde el programa toma un camino si la condición es verdadera y otro camino si es falsa. (Aho, Lam, Sethi, & Ullman, 2007). Esta capacidad de toma de decisiones basada en condiciones (como valores de variables, entradas del usuario o resultados de operaciones) es crucial para la lógica de los programas que deben responder de manera dinámica a diferentes situaciones.

```
If (esCierto) {
  console.log ('Esto es cierto');
} else {
  console.log ('Esto es falso');
}
```

Figura 6. If-else JavaScript, Fuente: Creación propia

En Figura 6, el fragmento de código ilustra cómo un programa escrito en JavaScript puede ejecutar alternativamente entre los mensajes “Esto es cierto” o “Esto es falso”, dependiendo del valor que la variable esCierto posea al momento de la evaluación.

Estructuras de Repetición (Bucles): Permiten ejecutar un grupo de instrucciones repetidamente mientras se cumpla una condición específica. Los bucles más comunes son el “for” y el “while”. El bucle “for” es útil para iterar sobre una secuencia (como una lista o un rango de números), mientras que el “while” se utiliza para repetir un bloque de código mientras una condición determinada sea verdadera. Estos bucles son

esenciales para tareas como procesar listas de datos, realizar cálculos iterativos o automatizar procesos repetitivos.

```
For (let x = 0; x < 100; x++) {
  console.log (`Este es el ${x} mensaje`);
}
```

Figura 7. Bucle for en JavaScript, Fuente: Creación propia

En Figura 7, el fragmento de código permite ejecutar la instrucción para imprimir un mensaje 100 veces sin necesidad de escribir las 100 instrucciones que serían necesarias de otra forma.

Además, existen estructuras más avanzadas como las funciones que permiten agrupar una secuencia instrucciones en bloques reutilizables y otras más sofisticadas como las clases que permiten agrupar datos y funciones en un prototipo reutilizable, aunque estas últimas no están presentes en todos los lenguajes. Estas estructuras son fundamentales para la creación de programas flexibles y potentes.

2.3 Sistemas de Tipos

Los sistemas de tipos en los lenguajes de programación actúan como mecanismos de seguridad que previenen el uso indebido o no permitido de ciertas operaciones. Esto es posible porque los tipos de datos tienen definidas las operaciones o conversiones válidas entre ellos (Fowler, 2010). La efectividad de estos sistemas de tipos y el momento en que se aplica la verificación pueden variar, lo que las categoriza en:

Según el momento de la verificación:

- Tipado estático: los tipos de todas las variables se determinan en tiempo de compilación. Esto significa que el tipo de cada variable se conoce y se verifica antes de que el programa se ejecute, lo que puede ayudar a prevenir errores en el tiempo de ejecución. Lenguajes como C++ y Java utilizan tipado estático.

- Tipado dinámico: el tipado dinámico determina los tipos de las variables en tiempo de ejecución. Es decir, recopila información sobre los tipos de las variables y resultados de las expresiones en tiempo real, mientras el programa se ejecuta en la computadora. Python y JavaScript son ejemplos de lenguajes con tipado dinámico.

Según la rigidez:

- Los lenguajes con tipado fuerte imponen restricciones estrictas en las interacciones entre diferentes tipos de datos, minimizando las conversiones implícitas de tipos. Esto reduce los errores no intencionados, pero puede requerir más código para la conversión explícita de tipos.

- Los lenguajes con tipado débil permiten más flexibilidad en las conversiones de tipos, a menudo de manera implícita, lo que puede resultar en comportamientos inesperados si no se maneja cuidadosamente.

2.4 Paradigmas de programación

Los paradigmas de programación son un conjunto de reglas, prácticas y procedimientos que dictan cómo un problema de computación debe ser modelado y adaptado. Los paradigmas dictan la pauta bajo la que se escriben los programas y esto afecta su estilo de programación. Cabe destacar que un lenguaje de programación puede soportar la escritura de múltiples paradigmas a la vez. Algunos de los paradigmas más usados son:

Imperativo: Bajo este tipo de paradigmas el programa se crea como una serie de instrucciones que comienza y termina, puede poseer subrutinas, pero lo importante es que bajo este paradigma todo y absolutamente todo tiene una definición y el programador se centra en escribir exactamente lo que tiene que pasar y como.

Declarativo: En este paradigma, la clave es poder describir qué debe hacerse, pero no como. Las definiciones del cómo son abstraídas del programa, lo que permite que el programador únicamente debe enfocarse en dictar lo que debe hacerse y cuando.

Funcional: En esencia, fundamenta los lenguajes de programación en principios matemáticos, empleando diversos conceptos. Por ejemplo, se postula que todos los procedimientos deben ser funciones idempotentes, lo que implica que su resultado será determinado exclusivamente por sus entradas, sin importar cuántas veces se ejecuten. Este logro se alcanza al eliminar el estado compartido y priorizar el uso de datos inmutables. En consecuencia, cualquier modificación requerida implica la creación o clonación de nuevos elementos. Aquellas instancias que no cumplen con esta pureza son consideradas como efectos colaterales que el paradigma procura mitigar con diligencia.

Orientado a Objetos: Este paradigma tiene varias propiedades, entre las más destacables está la encapsulación que permite exponer u ocultar parte de los datos del programa dependiendo de si son relevantes para otras secciones del programa o no. Posee el concepto de herencia donde las propiedades prototípicas y métodos de un objeto pueden ser heredados a otro tipo de objeto. Otra propiedad interesante es que los objetos se pueden comunicar entre ellos con “mensajes”. Estos mensajes pueden ser de cualquier tipo, como enviar texto, números o ejecutar acciones.

2.5 Intérpretes y compiladores

Existe una división clara en los lenguajes de programación que los divide en dos tipos según su forma de ejercitarse en la computadora. Por un lado, los lenguajes compilados, que requieren de un programa externo conocido como compilador. El compilador se encarga de traducir el código fuente, escrito en el lenguaje de programación, a un lenguaje objetivo, que suele ser código binario o de máquina. Y, por otro lado, los lenguajes interpretados, a diferencia de los compilados, no pasan por un proceso de compilación antes de su ejecución. En su lugar, un programa llamado intérprete lee y ejecuta el código fuente directamente. El intérprete se encarga de realizar las acciones equivalentes a cada línea o bloque de código en tiempo real. Normalmente a los programas escritos en lenguajes interpretados se les llama scripts.

CAPÍTULO 3. TÉCNICAS PARA EL DESAROLLO DE LENGUAJES DE PROGRAMACIÓN

3.1 Sobre el desarrollo de Lenguajes de programación

El desarrollo de un lenguaje de programación es una tarea bastante compleja y amplia, pero en líneas generales se recomienda agrupar las tareas que llevan desde el procesamiento del código fuente hasta la ejecución del código en alguna plataforma en dos grandes bloques, la fase analítica o frontend y la parte de síntesis o backend (Aho, Lam, Sethi, & Ullman, 2007).

Durante las etapas de análisis, las tareas consisten básicamente en procesar cada una de las partes del código fuente y transformarlas en una representación alternativa del código con la que sea más fácil de trabajar que hacerlo directamente sobre el texto del programa. Eventualmente, el resultado de esta parte se convierte en un parse tree o árbol sintáctico.

La segunda etapa se centra en trabajar sobre el árbol sintáctico y transformarlo, es decir, modificar su estructura con distintos fines. Los más comunes son resolver expresiones que son constantes en tiempo de compilación, optimización de código y generación de código.

Entre cada etapa existe un número de subetapas que aportan algo nuevo al análisis, por lo general la estructura interna de la etapa de análisis para un compilador o un intérprete consta de la etapa donde se analizan las palabras, luego una donde se analiza la sintaxis y finalmente la semántica ante de llegar a un estado de código intermedio que sea utilizable.

3.2 Notación sobre árboles

Los árboles son estructuras de datos muy prominentes dentro de las ciencias de la computación debido a las muchas propiedades que poseen, así como los algoritmos que los emplean. Debido a su amplio uso, existe una nomenclatura que se usa de manera muy común para referirse a las partes que componen la estructura (Aho, Lam, Sethi, & Ullman, 2007). Los árboles en informática constan de las siguientes partes:

Raíz: Es el elemento inicial del árbol, a partir de este se generan las conexiones a todos los demás elementos del árbol, el nodo raíz puede tener hijos, pero no puede tener padre.

Nodos: Son cada uno de los elementos individuales que componen el árbol, los nodos poseen la característica de almacenar un valor y permiten navegar hacia niveles inferiores del árbol ya que posee un puntero a cada uno de sus hijos.

Niveles: hace referencia a la cantidad de generaciones que hay que navegar desde la raíz hasta un nodo en particular, así los hijos de la raíz están en nivel 2, los nietos en nivel 3 y así sucesivamente.

Hojas: Es el nombre especial que reciben los nodos que se ubican en el último nivel del árbol, es decir el más profundo

En la Figura 8 se puede observar cómo se conectan las distintas partes de un árbol. Por lo general la estructura de presenta de manera invertida por lo que la raíz casi siempre termina siendo el nivel más alto en el diagrama.

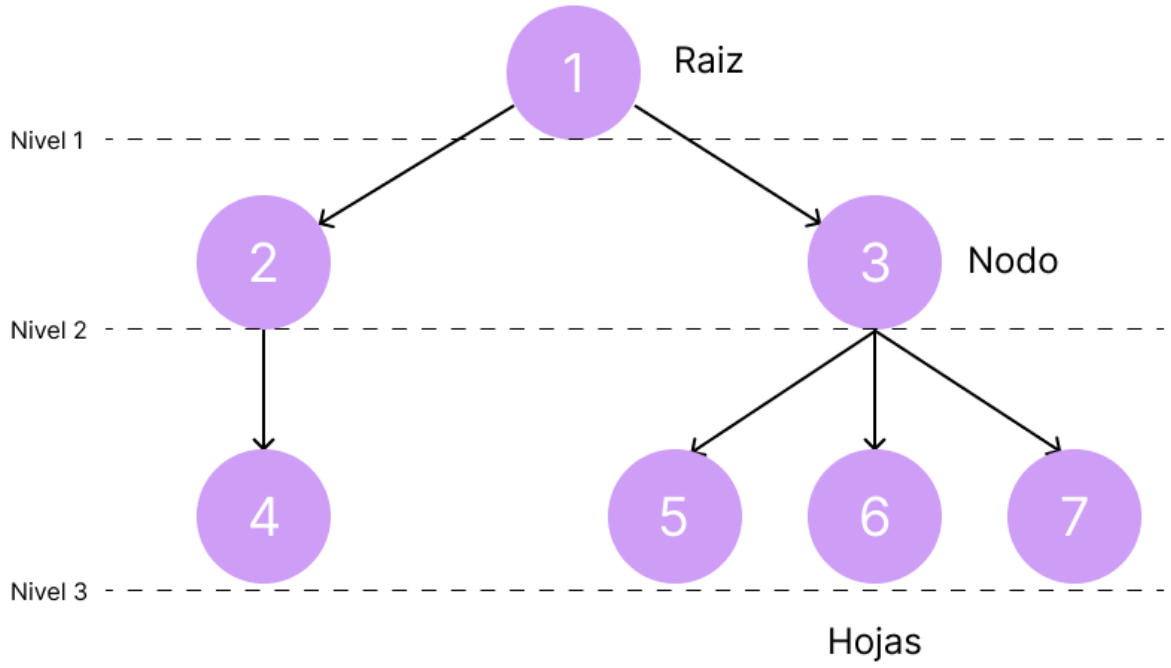

Figura 8. Diagrama de un árbol, Fuente: Creación propia

3.3 Técnicas para el análisis de un lenguaje de programación

Para un análisis eficiente sobre el texto de un programa en algún lenguaje de programación, lo primero que debe hacerse es definir la gramática del lenguaje, ya que a partir de esta el resto de los elementos en las siguientes etapas quedan comprometidos porque se moldean a la especificación de la gramática.

Como ya se explicó anteriormente, las gramáticas libres de contexto son el tipo de gramáticas que suelen emplearse al momento de desarrollar la sintaxis para los lenguajes de programación. De manera formal, las gramáticas libres de contexto, en adelante llamadas únicamente gramáticas, se definen con cuatro elementos.

El primer elemento es un conjunto finito de símbolos terminales, algunas veces llamados **tokens**. Estos son los símbolos elementales del lenguaje, es decir, las unidades mínimas con significado (Fowler, 2010). El análogo de esto para un lenguaje natural serían las palabras.

El segundo elemento es un conjunto de símbolos no terminales, estas son secuencias de símbolos terminales dispuestos en patrones que hay que definir.

El tercer elemento sería un conjunto de **producciones** que son una secuencia de instrucciones que indican cómo transformar cada uno de los símbolos no terminados. A la izquierda está la cabeza, o el nombre del símbolo no terminal, y a la derecha, separado por una flecha, se indica la secuencia de símbolos terminales-no terminales en los que se puede transformar. Finalmente, hay que designar alguno de los símbolos no terminales como el punto de inicio de la gramática.

3.3.1 Nomenclatura de Gramáticas

Existe una notación bastante popular para anotar gramáticas libres de contexto llamada Backus-Naur Form o BNF (Aho, Lam, Sethi, & Ullman, 2007). Esta notación permite describir fácilmente la estructura jerárquica de los lenguajes de programación. Para anotar gramáticas con esta notación se colocan únicamente símbolos no terminales a la izquierda, luego le sigue una flecha y a la derecha se colocan las producciones de esa regla.

En la Figura 9 se define una gramática que sirve para definir operaciones aritméticas sencillas, únicamente cubriremos la suma y la multiplicación.

$$E \rightarrow T\ E'$$
$$E' \rightarrow +\ T\ E'$$
$$E' \rightarrow \varepsilon$$
$$T \rightarrow F\ T'$$
$$T' \rightarrow *\ F\ T'$$
$$T' \rightarrow \varepsilon$$
$$F \rightarrow (\ E\ )$$
$$F \rightarrow número$$

Figura 9. Gramática de ejemplo, Fuente: Princeton (09/09/2023)

En la gramática anterior tenemos una gramática con 9 reglas, en esa gramática los símbolos terminales son los operadores '+' para la suma, '*' para la multiplicación, 'número' que se refiere a un número cualquiera y '(' junto a ')' que permiten agrupar expresiones. Son considerados símbolos terminales por que no existe ninguna regla de producción que desglose a esos símbolos en más símbolos no terminales. En otras palabras, los símbolos terminales ya no poseen más definiciones por lo que son un valor literal.

La gramática de ejemplo también utiliza los símbolos no terminales E que significa expresión, T, que hace referencia al término en una expresión de suma y F que es para indicar un factor dentro de las expresiones de multiplicación. En este ejemplo en particular las reglas primas, es decir E' y T' se refieren a la siguiente regla que se deriva de E o T respectivamente.

Finalmente, el terminal ε hace referencia a una palabra 'vacía' lo que significa que es cualquier texto con longitud cero. Se suele utilizar este término para poder crear reglas de producción donde algunos elementos sean opcionales.

Por ejemplo, la expresión E' tiene dos alternativas una donde se continúa con el símbolo '+' indicando que existe un segundo término para la suma y la otra donde no existe un '+' por lo que no es necesario seguir derivando la expresión a partir de ahí.

Para explicar mejor la gramática anterior usaremos un ejemplo muy sencillo, la figura 10:

$$2 * 1 + 3$$

Figura 10. Expansión de una expresión en la gramática del ejemplo parte 1, Fuente: Creación propia

La forma en que la gramática de ejemplo aborda esa expresión es primero definiendo que todo es una expresión como se indica en Figura 11.

$$E = 2 * 1 + 3$$

Figura 11. Expansión de una expresión en la gramática del ejemplo parte 2, Fuente: Creación propia

ahora hay que ver cada elemento de manera individual y se desglosa como en Figura 12.

$$E = T + T = 2 * 1 + 3$$

Figura 12. Expansión de una expresión en la gramática del ejemplo parte 3, Fuente: Creación propia

El desglose anterior ha convertido la expresión en la suma de dos términos, pero debemos seguir aplicando las reglas de producción hasta llegar a únicamente símbolos terminales como en Figura 13.

$$E = [F * F] + F = 2 * 1 + 3$$

Figura 13. Expansión de una expresión en la gramática del ejemplo parte 4, Fuente: Creación propia

En el paso anterior descompusimos la regla T por su producción, es decir la multiplicación de dos factores y el término T se ha transformado en un factor independiente como se indica en Figura 14.

$$E = [número * número] + número = 2 * 1 + 3$$

Figura 14. Expansión de una expresión en la gramática del ejemplo parte 5, Fuente: Creación propia

Finalmente, los factores se reducen a la última regla de producciones que es '*número*' y así termina el proceso de la gramática. El proceso anterior es únicamente con fines ilustrativos, ya que las técnicas usadas para que una computadora pueda analizar gramáticas son diferentes. Con el ejemplo anterior podemos ver cómo las gramáticas pueden definir en este caso un lenguaje sencillo para expresar operaciones aritméticas.

3.3.2 Árboles de parseo

Las gramáticas trabajan fundamentalmente tomando el texto de entrada y repetidamente aplicando producciones hasta que se pueda derivar un símbolo terminal con la entrada actual en el texto. Este proceso de descender sobre las reglas de producción naturalmente lleva a la formación de estructuras de árbol. Estos árboles que se van formando tras finalizar la lectura de todo el texto y de haber aplicado las reglas de producción hasta únicamente poder derivar terminales se llaman árboles de parseo.

Un ejemplo de cómo se ve el árbol de parseo usando la gramática de ejemplo en la Figura 15.

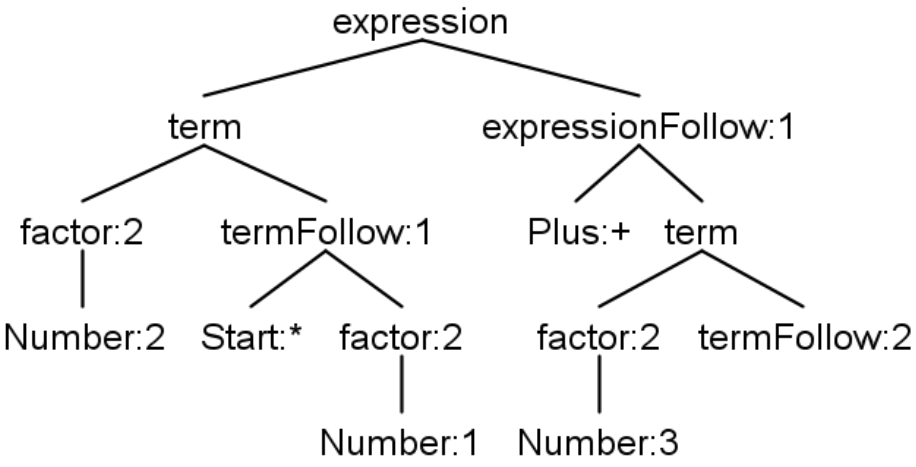

Figura 15. Árbol de parseo de una expresión aritmética, Fuente: Creación propia

En el árbol de parseo para la expresión '2 * 1 + 3' podemos ver la forma de la estructura. Del diagrama podemos observar en primer lugar que la regla de entrada, en este caso la de E o expresión se encuentra en la raíz del árbol, lo siguiente es que todos lo no terminales siempre terminan en las hojas del árbol y, por último, pero no menos importante está el hecho que por la forma en que la gramática fue diseñada, la precedencia de operadores se preserva, en este caso de la multiplicación sobre la suma. Esto es así porque todas las operaciones de multiplicación siempre están más cerca de los terminales que la suma por lo que al momento de hacer más análisis sobre este árbol, esas operaciones siempre se mantendrán más cerca y se evaluarán antes.

3.4 Técnicas para el análisis de unas gramáticas

Cuando la gramática para el lenguaje se encuentra definida aún falta la encontrar una forma que nos permita pasar de texto plano a un árbol de parseo. Para cualquier lenguaje existe un parser, un programa que lo convierte a un árbol de parseo que como mucho puede procesarlo en O(n3) donde n es el número de terminales. Pero el enfoque de bruto es muy lento para poder crear parsers efectivos (Aho, Lam, Sethi, & Ullman, 2007).

En muchos compiladores comerciales suelen usar una técnica llamada descenso recursivo con alguna de las siguientes variantes. Top-down parsing donde el árbol se comienza a construir desde la raíz hacia las hojas y Bottom-Up que comienza a ensamblar el árbol desde las hojas hacia la raíz. Al pararse usando la técnica

de Bottom-Up se pueden crear parsers que pueden manejar gramáticas con esquemas más complicados. Sin embargo, en este capítulo nos centraremos en la técnica de descenso recursivo de izquierda a derecha o más bien conocido como LL debido a que es más fácil implementarlos de manera manual eficientemente.

3.4.1 Parseo Top-Down

Funciona leyendo texto de la entrada y procesando de izquierda a derecha mientras ensambla el árbol de parseo de la raíz hasta las hojas. Supongamos nuevamente la entrada 'número * número + número'. En la gramática ejemplo de la sección anterior la forma de ensamblar el árbol es como en Figura 16.

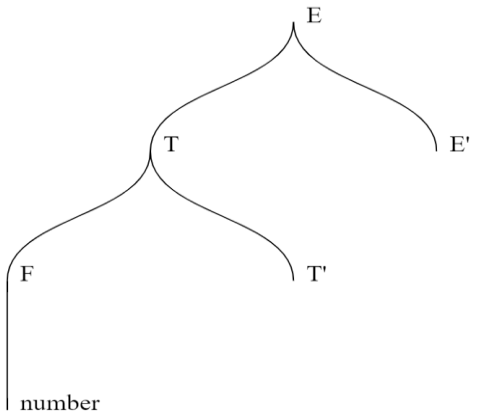

Figura 16. Parseo de una expresión aritmética parte 1, Fuente: Creación propia

El parser va descendiendo en las reglas de producción hasta que llega a un símbolo terminal en este caso un número como se aprecia en la Figura 17.

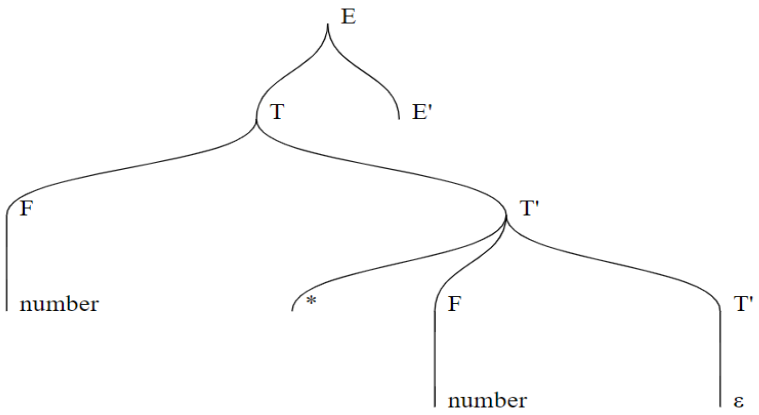

Figura 17. Parseo de una expresión aritmética parte 2, Fuente: Creación propia

Una vez que termina con esa sección del árbol regresa hacia la regla T y procede nuevamente a seguir aplicando las distintas reglas de producción hasta que vuelve a llegar a los símbolos terminales de cada sección como indica la Figura 18.

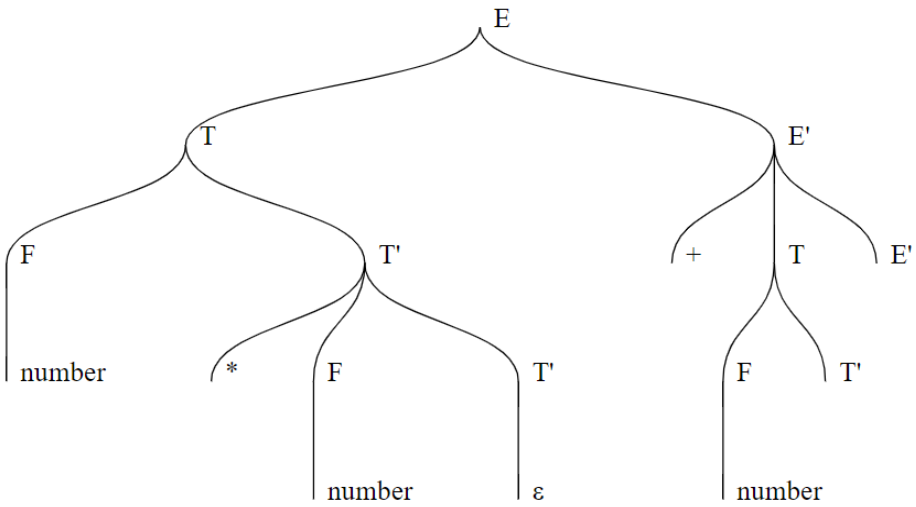

Figura 18. Parseo de una expresión aritmética parte 3, Fuente: Creación propia

Eventualmente termina con todo el subárbol izquierdo regresa nuevamente a la raíz y vuelve a aplicar lo mismo, pero al lado derecho. El árbol de parseo terminado se vería como en Figura 19:

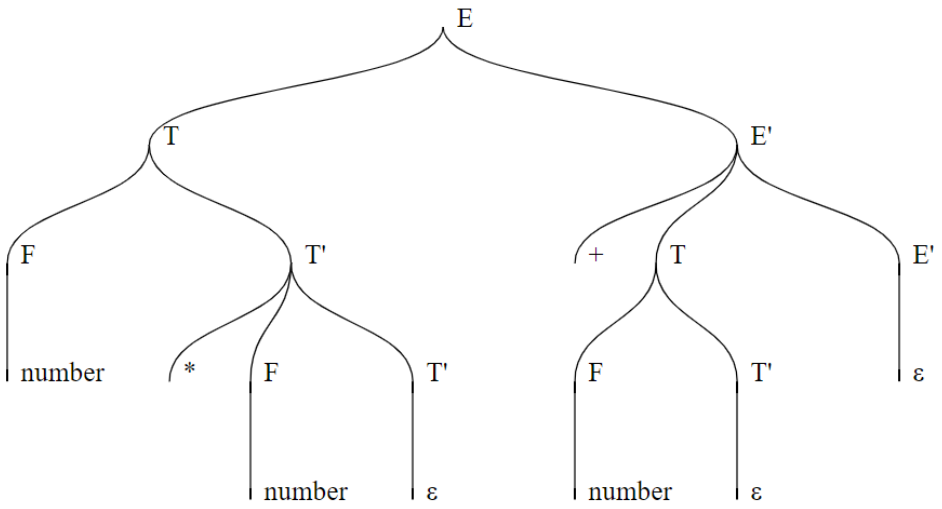

Figura 19. Parseo de una expresión aritmética parte 4, Fuente: Creación propia

Esta simulación nos lleva a dos conclusiones clave. En primer lugar, la estructura del árbol generada por el analizador requiere que todas las reglas de producción empiecen con un símbolo terminal o, en caso de comenzar con un no terminal, que las producciones subsiguientes de este no terminal inicien también con un símbolo no terminal. La segunda es que un símbolo no terminal no puede ser el primero en su propia regla de producción, en otras palabras, no puede ser recursivo por el lado izquierdo dado que esto causaría un bucle infinito (Aho, Lam, Sethi, & Ullman, 2007).

3.4.2 Parseo Top-Down predictivo

Una consideración importante con los parsers que hemos usado hasta el momento es que si un símbolo no terminal tiene más de una regla de producción el parser podría elegir de manera equivocada por lo que solo habría dos alternativas, fallar el proceso de parseo debido a esta ambigüedad o implementar backTracking que en esencia es regresar un paso en el proceso de parseo y probar con las producciones alternativas hasta que encontremos una que satisfaga la entrada.

Una forma de lidiar con este problema es la implementación de un parser predictivo, esto quiere decir que el parser puede determinar de manera inequívoca el flujo que debe tomar para seguir paseando el resto de la entrada. Esto se logra haciendo un lookahead o adelantando tokens de la entrada para poder tomar la decisión, gracias a que las gramáticas LL no son recursivas por la izquierda quiere decir que todas las reglas de producción para cada uno de los no terminales se pueden determinar viendo cuáles son sus secuencias de inicio usando los símbolos terminales únicamente. Para poder implementar un parser predictivo primero debemos analizar nuestra gramática y obtener dos listas de valores.

La primera lista es el conjunto FIRST de cada no terminal. Este conjunto es la lista de todos aquellos símbolos terminales que aparecen en cada una de las derivaciones posibles de un no terminal. Se llama así porque efectivamente es la lista del primer no terminal que aparece en cada una de las posibles derivaciones de un no terminal (Aho, Lam, Sethi, & Ullman, 2007).

Existen algunas reglas para calcular el conjunto FIRST de una producción A denotado por FIRST(A):

- Si A → ε donde ε es la cadena vacía; añadir ε a FIRST(A).
- Si A → α donde α es un terminal; añadir α a FIRST(A).
- Si A → β donde β es un no terminal; añadir FIRST (β) a FIRST(A).
- Si A → $\alpha\beta$ donde α y β son no terminales y FIRST (α) incluye ε; añadir FIRST (α) a FIRST(A) excluyendo ε y además agregar FIRST (β) a FIRST(A).

La segunda lista es el conjunto FOLLOW, que básicamente es una lista de todos los terminales posibles que pueden aparecer después de una regla de producción. Las reglas para calcular FOLLOW(A) son:

- $, donde $ es el fin de la entrada se añade automáticamente al FOLLOW(A) si esta es la regla inicial de la gramática.
- Si A → αBβ donde α y β son no terminales; añadir FIRST (β) a los elementos de FOLLOW (B) exceptuando ε.
- Si A → αBβ donde α y β son no terminales y FIRST (β) contiene ε; añadir a FOLLOW (A) a FOLLOW(B)

Estas dos listas de conjuntos son cruciales para la implementación de un parser predictivo LL ya que le permiten poder anticiparse al inicio y fin de las reglas solamente con ver cierta cantidad de tokens adelante y no todo el programa. El tipo de parser que discutiremos en la sección de la implementación será de tipo LL 1 es decir, recursivo descendente, de izquierda a derecha, y predictivo con 1 símbolo.

3.5 Análisis lexicográfico

Raramente, un parser LL puede trabajar directamente con la entrada del texto de un programa esto ocurre porque los caracteres individuales no necesariamente son reconocidos por la gramática. Así como en español no es común que una única letra tenga significado, en el parse de gramáticas LL tampoco es común que se pueda asociar un único carácter a un terminal, aparte de que es poco práctico leer carácter por carácter como entrada. Por eso existe un paso previo al análisis sintáctico o parser, esto es el escaneo, análisis lexicográfico o tokenización.

Escaneo, análisis lexicográfico o tokenización, consiste en ir escaneando el archivo de texto con el código fuente, un carácter a la vez e irlos agrupando en las unidades mínimas con significado para el lenguaje de programación. Similar a cuando una persona lee, esta empieza a agrupar cara letra en el texto hasta que puede formar una palabra con significado, en el caso del escaneo, este agrupa letras individuales hasta encontrar un patrón conocido de los distintos símbolos en el alfabeto del lenguaje y los agrupa en una unidad llamada **token** y repite este proceso hasta llegar al final del archivo. Así, por ejemplo, en lugar de que el parser tenga que lidiar con la complejidad de la entrada ‘1’, ‘2’, ‘3’ significa el número 123 o 12 y luego un 3 o cualquier combinación de estos, simplemente se puede enfocar en el tipo de token en la entrada sin importar su valor literal o la cantidad de caracteres que lo conforman.

Al final del proceso, lo que originalmente era un archivo de texto ya se ha transformado en un flujo de tokens individuales que describen el tipo de token que se ha analizado sin detallar cada una de las cadenas que lo conforman (Fowler, 2010). En caso de que algún carácter no pertenezca al alfabeto del lenguaje entonces esta etapa provoca un fallo. La parte de compilador encargada de esta etapa se llama lexer.

3.6 Análisis sintáctico

Durante esta etapa se procesa el flujo de tokens creados en la etapa anterior, la principal función del analizador sintáctico o parser es ir ensamblando estos tokens en la estructura que indique la gramática, estos tokens se van procesador y ensamblando en el árbol de parseo que es una representación abstracta para organizar el código y poder trabajar de manera programática en su análisis y transformación en la siguiente etapa.

Los errores ocurridos durante esta etapa se les llama errores de sintaxis, esto, indica que la secuencia de tokens en la entrada no concuerda con ninguna de las estructuras gramaticales definidas, por lo que el parser inicialmente no sabría cómo avanzar. Sin embargo, es posible que el parser se recupere del fallo usando alguna estrategia. La idea de implementar una estrategia de recuperación es poder seguir parseando el resto del flujo de tokens, aunque una sección en específico no pueda ser procesada. Una de las estrategias más comunes es la del uso de tokens de sincronización, como por ejemplo "}" o ";" que normalmente son el último token en un bloque de código o una oración respectivamente (Aho, Lam, Sethi, & Ullman, 2007).
La idea de esta estrategia es bastante sencilla, una vez detectado un error de sintaxis el parser simplemente va consumiendo e ignorando todos los demás errores hasta que llega a uno de estos tokens de sincronización y continua como si el error nunca hubiera ocurrido, esto le permite seguir procesando tokens aun si un error se presenta en el patrón de los tokens de entrada.

3.7 Tabla de símbolos

Al finalizar la construcción de los árboles de parseo, se necesita inicializar una estructura llamada tabla de símbolos en esta se coloca la mayor cantidad de información relevante sobre los símbolos del programa. Los símbolos del programa hacen referencia a todas aquellas estructuras o constructos que el usuario haya declarado o utilizado en el programa, así por ejemplo se puede almacenar información sobre donde se crea una variable, el alcance de esta, su tipo y además su mutabilidad, es decir si puede alterarse su valor después de la creación. Las variables no son los únicos símbolos en un programa, otras estructuras comunes como la declaración de tipos o clases también califican. Dependiendo el tipo de símbolo se puede almacenar

información como su ubicación en memoria, métodos, tipo, tipos de retornos, lista de parámetros si es una función, su nombre, donde fue creado y donde está siendo usado.

Con toda esta información el compilador puede tomar decisiones para garantizar que un programa sea correcto y se pueda ejecutar, también le proporciona insumos para validar que los símbolos declarados o usados por el usuario están siendo utilizados de la manera esperada.

Hay dos maneras comunes de implementar tablas de símbolos, la primera es crear una única tabla donde se almacenarán los símbolos de los programas, normalmente se implementan con una estructura llamada mapa o hashtable y la forma de buscar un símbolo en concreto es a través de su **FQDN** (Fully Qualify Domain Name) o nombre de dominio completo. La segunda opción es una estructura anidada de tablas de símbolos más pequeñas que se limitan a únicamente los símbolos dentro de su alcance (Aho, Lam, Sethi, & Ullman, 2007). La ventaja de anidar las tablas de símbolos es que la información sobre el alcance de cada símbolo es un reflejo de la estructura jerarquizada de las tablas, así todas las variables de una tabla superior están disponibles para una tabla hija, pero los símbolos declarados en la tabla hija no se puede acceder desde el padre. Esto es así debido nuevamente a la estructura jerárquica de los programas. La desventaja es que es más complejo de implementar y de buscar símbolos ya que hay que hacer búsquedas de manera recursiva.

Por otro lado, si se implementa usando únicamente una tabla, las búsquedas son sencillas, sin embargo, la información sobre el alcance de los símbolos debe de guardarse como información adicional o en una estructura adicional lo que complica la definición del alcance de los símbolos.

3.8 Análisis semántico

El análisis semántico en un compilador implica asignar significado a las instrucciones del programa original. En este punto, el compilador, que ya tiene un árbol de análisis, identifica estructuras clave como creación de variables y declaraciones de funciones. Además, en esta etapa, se utiliza la tabla de símbolos para almacenar y consultar información sobre los símbolos del programa, lo que facilita la validación del alcance de las variables y la coherencia de los tipos de datos en las asignaciones.

Como el lenguaje de programación que se diseñará será fuertemente tipado uno de los procesos más trascendentales que ocurre en esta esta es el **type checking**. Mediante este proceso el compilador puede crear una lista de tipos tanto de los nativos como de los creados por el programa, con esta lista de tipos y sus propiedades el compilador puede revisar la exactitud de un programa al interpretar las instrucciones y

los valores que generará en tipo de ejecución. Además de eso le permite determinar si se puede ejecutar conversiones implícitas entre tipos y de manera general si se usan únicamente las propiedades que sí existen en ese tipo. Estos análisis son de manera estática ya que se realizan únicamente con las instrucciones escritas del programa y ayudan a saber si los contratos declarados en las distintas partes del programa son respetados y utilizados de manera correcta tal cual fueron definidos.

3.9 Generación de código intermedio

Una vez que se ha validado el programa y que la semántica de este es correcta, se puede pasar a la siguiente fase, esta es la generación de código intermedio. Para la generación de código intermedio existen dos opciones populares, la primera es llamada código de tres direcciones o 3AC (three address code) (Aho, Lam, Sethi, & Ullman, 2007), donde básicamente se reescribe el código del árbol de parseo en instrucciones más simples que se acercan bastante al lenguaje de máquina o ensamblador, se llama así porque la mayoría de las instrucciones posee únicamente 3 operandos lo que lo vuelve fácil de procesar, aunque sea texto.

```
Const a = b + c * d - e;
```

Figura 20. Asignación del valor de una expresión en JavaScript, Fuente: Creación propia

El código JavaScript anterior, ver Figura 20, asigna el resultado de una expresión aritmética en una variable se podría representar como 3AC como muestra la Figura 21.

```
T1 = c * d
t2 = b + t1
t3 = t2 - e
a = t3
```

Figura 21. Asignación del valor de una expresión en 3AC, Fuente: Creación propia

Otra forma de representar código intermedio en el proceso de compilación es mediante el uso de Árboles de Sintaxis Abstracta (ASTs, por sus siglas en inglés). Aunque los ASTs guardan cierta similitud con los árboles de parseo, también presentan diferencias significativas, principalmente en el tipo de información que contienen.

La principal diferencia radica en el nivel de abstracción. Un AST omite muchos de los detalles sintácticos específicos del lenguaje de programación, como el uso de paréntesis, palabras clave, y símbolos de sincronización como comas o puntos y comas. Estos elementos, aunque esenciales para la estructuración

correcta del código en su forma original, no son necesarios para el análisis o la optimización del código durante la compilación.

En un árbol de sintaxis para una gramática LL 1 existe el inconveniente que el árbol se vuelve excesivamente profundo, ya que las reglas deben definirse de tal manera que no sean ambiguas, con un único token es bastante común que los árboles de sintaxis se aniden múltiples niveles de manera un poco redundante. Una ventaja de usar un AST en lugar que directamente el árbol de sintaxis es que se pueden simplificar varias de estas reglas anidadas lo que produce árboles más planos y fáciles de analizar.

3.10 Generación de código

Al finalizar todos los análisis y optimizaciones finalmente es momento de generar el código equivalente en otro lenguaje. Por lo general los lenguajes compilados generan código de máquina, conocido como ensamblador, que básicamente son las instrucciones más sencillas que una máquina puede ejecutar al romper instrucciones más complejas de un lenguaje más expresivo en varias instrucciones de otro lenguaje más limitado. Ejemplo de esto es el lenguaje C a ensamblador.

Existen otras salidas como por ejemplo código binario que se ejecuta en máquinas virtuales, es decir son utilizados por un programa que simula ser una máquina más compleja pero que en realidad no existe. A este tipo de salida se le conoce como Bytecode y dos ejemplos de esto son C# o Java.

Finalmente existe otra salida que es código fuente en otro lenguaje de alto nivel diferente. En este caso al pasar de Lenguaje Haib a JavaScript técnicamente se categoriza como transpiración, pero para efectos prácticos nos referimos a la salida de la generación de código como ‘compilado’.

CAPÍTULO 4. LENGUAJES DE DOMINIO ESPECÍFICO

4.1 Qué es un DSL

Los lenguajes de dominio específico (DSL) son lenguajes de programación con la característica que han sido diseñados y optimizados para un dominio en concreto. Un dominio se refiere a un ámbito específico de estudio o un campo de conocimiento en particular como la matemática, la física y la biología.

4.2 Tipos de DSL

Existen dos tipos de DSL, los internos que toman como base un lenguaje de propósito general y limitan su sintaxis y estilo para poder adaptar el lenguaje general al dominio. Todos los DSL internos son realmente código válido en su lenguaje anfitrión, pero al tener reglas adicionales da la impresión de ser un lenguaje diferente. Un ejemplo de un DSL interno es Ruby y su framework Rails (Fowler, 2010).

El otro tipo de DSL son los externos, la diferencia crucial es que los externos tienen una sintaxis propia y por tanto también se debe de implementar un compilador o intérprete que permita su ejecución en una computadora. Esto también implica que se crea un proceso completo de análisis para poder traducir esta sintaxis personalizada y poder ser ejecutado en una computadora (Fowler, 2010).

4.3 Razones para implementar un DSL

La implementación de un DSL es una tarea compleja y de esfuerzos considerables por lo que vale la pena preguntarse qué razones existen para implementar un lenguaje especializado si ya existen muchos lenguajes generales.

Como los DSL son especializados en las tareas del dominio una consecuencia de esto es el aumento de productividad (Fowler, 2010). Esto se debe al efecto que provoca una sintaxis que facilita la abstracción de los problemas y tareas repetitivas del dominio. Una sintaxis más adecuada también trae otros efectos beneficiosos para la productividad como la necesidad de escribir menos código que no solo vuelve los programas más legibles y fáciles de entender lo que significa o hacen sino también reducen la probabilidad de escribir un programa erróneo. Otra importante cualidad de la sintaxis optimizada de los DSL es que facilitan hacer programas de la forma correcta y vuelve extremadamente difícil escribir un código que se salga de las directrices del diseño del DSL, en otras palabras, dificulta que se cometan errores.

Normalmente, durante un proceso de desarrollo de software son los expertos del dominio quienes definen y utilizan características que se listan como requerimientos de un sistema informático y luego estos requerimientos se les transfieren a los desarrolladores quienes son los responsables de implementarlo en algún lenguaje de programación. Lamentablemente, esta forma de ejecutar un proceso de desarrollo software ha probado ser particularmente ineficiente en la fase donde se transfiere los requerimientos a los desarrolladores y estos en implementar lo requerido, esto se debe a que los expertos del dominio tienen su propia terminología y notaciones, con las cuales muchas veces dan por sentado que los desarrolladores comprenden con precisión y en el otro sentido los programadores también tiene su lenguaje extremadamente técnico que es extraordinariamente difícil de explicar o traducir al lenguaje del dominio. Esta dinámica de dos grupos habla diferentes lenguajes es lo que produce estas ineficiencias.

Una forma de reducir esto es con un DSL ya que un DSL puede implementar palabras y conceptos propios del dominio dentro del lenguaje de programación. Esto no significa que los expertos del dominio puedan programar en el DSL necesariamente pero sí supone una mejora en la claridad cuando se intenta leer. El simple hecho que los expertos puedan leer código DSL ya mejora la velocidad y la claridad de la comunicación además de dotar a los expertos con una comprensión más profunda de su sistema informático que les permita encontrar errores dentro de los programas (Fowler, 2010).

Debido a la tendencia a la abstracción de los DSL, este tipo de lenguaje es muy bueno para la síntesis de programas, programas que, escritos en un lenguaje de propósito general, serían más difíciles de comprender en comparación. Como los DSL tienden a incorporar palabras del domino dentro de su sintaxis propicia la construcción de programas más compactos y claros. Estas abstracciones del lenguaje pueden tomar la forma de funciones utilizadas comúnmente incorporadas al lenguaje, estructuras nativas que el lenguaje maneja internamente para eliminar o reducir la escritura manual de código como inicialización o combinación de datos, palabras clave que cambian la forma en que el programa opera sobre los datos sin necesidad de escribir manualmente todas las instrucciones o sintaxis que actúan como atajos para operaciones utilizadas en conjunto de manare repetitiva.

4.4 Principios de Diseño de los DSL

Claridad: Durante el diseño de un DSL el enfoque es sobre la claridad del lenguaje. Supuesto un usuario típico del lenguaje la meta es que este pueda entender el significado de las sentencias del programa con fluidez, independiente de si este es un programador o experto del dominio. Esto también implica que las pueda leer y comprender lo más rápido posible, por esta razón debe evitarse el código de ceremonia.

El código de ceremonia es todo aquel código que no influye directamente en la lógica de un programa sino cuya función es inicializar o habilitar la funcionalidad básica de otra entidad. Normalmente esto ocurre con código que expone mucha configuración inicial antes de llegar al código que verdaderamente modifica el comportamiento del programa. A este tipo de código normalmente se le conoce con el nombre de boilerplate y un ejemplo muy famoso es Java.

```
public class Main {
    public static void main(String[] args) {
        System.out.print("Hello");
    }
}
```

figura 22. Java boilerplate, Fuente: Creación propia

El código de la figura 22 muestra cómo se debe escribir mucho código de Java para ejecutar una única instrucción. En ese ejemplo la clase Main no sirve para nada excepto habilitar la creación de una función que actúa como punto de entrada para el programa.

Experimental: Muy probablemente las primeras versiones de un DSL recién definido no funcionen, ya sea por rechazo de los expertos del dominio o del equipo de desarrollo. En este aspecto es importante probar siempre ideas nuevas, proponer múltiples alternativas para observar las reacciones de la gente. Básicamente a través de un proceso de rechazo y adaptación se llegará a un lenguaje más útil para los usuarios.

Familiaridad: Es muy común que en las áreas donde se trata de introducir un DSL ya exista algún lenguaje de programación general y jerga técnica que utilice los expertos del dominio. Es importante incorporar estos elementos en el DSL para producir un lenguaje que se adecue a las necesidades de los usuarios. La incorporación de estos elementos en el DSL genera un proceso de adaptación más orgánico por parte de los expertos del dominio, dado que muchos de los conceptos que utilizan ahora tienen una representación dentro del lenguaje.

Consistencia: Se debe evitar llevar el DSL demasiado cerca del espectro de un lenguaje natural, esto es porque el lenguaje se llena mucho de reglas sintácticas inconsistente que dificultan entender el significado de las sentencias. Después de todo un DSL sigue siendo un lenguaje de programación y su uso debe sentirse como tal. Esto es importante porque la precisión que expresan los lenguajes de programación en su semántica frente a un lenguaje natural es muy superior.

4.5 Casos de uso de un DSL

Los DSL pueden utilizarse en muchos ámbitos, sin embargo, donde hacen un aporte de manera más contundente es en aquellos donde se debe de codificar el conocimiento de los expertos del dominio en algún sistema informático. Esto se debe a que en este tipo de ámbitos las reglas del dominio son bastante cambiantes y al intentar seguirles el paso utilizando procesos de desarrollos de software tradicionales, la complejidad del software aumenta de forma desproporcionada al igual que la calidad de este se ve comprometida.

Por esa razón en lugar de que los desarrolladores traten de mantener el sistema al corriente es mucho más eficiente permitirles una forma de escritura o expresión de las reglas del dominó a sus expertos, de esta manera los expertos no tienen que preocuparse que los pequeños detalles de sus reglas se pierdan entre en la comunicación con los desarrolladores o que se implementen de manera incorrecta en el sistema informático resultante. En el sentido inverso, esto también quiere decir que los desarrolladores del sistema no deben de preocuparse de entender ni de mantener las reglas del dominio como carga cognitiva ya que no se convierte en responsabilidad suya el mantener los detalles del dominio, en su lugar los desarrolladores pueden enfocarse de manera única en que el DSL permite expresar de manera sencilla y entendible las reglas generales del dominio.

Los DLS permiten generar esta dinámica única donde los expertos pueden detallar las especificaciones en el sistema por ellos mismos y el sistema reacciona de manera dinámica a los cambios que los expertos solicitan. Esta dinámica permite que los softwares se puedan desarrollar de manera más eficiente al reducir la necesidad de viajes de ida y vuelta entre los expertos del dominio y los desarrolladores. Además de claramente ser más responsivo al momento de realizar cambios.

Otras de las ventajas de implementar DSL es que los expertos son dados de herramientas que pueden utilizar para componer y combinar en maneras complejas permitiéndoles así expresar reglas que quizás no son parte del lenguaje base pero que ellos pueden implementar por sí mismo, incorporando así más funcionalidad de la que se creyó posible inicialmente. Esto tiende a conocerse como la expansibilidad del lenguaje, esto quiere decir que el lenguaje puede obtener e incrementar las funcionalidades que se le fueron conferidas en su diseño original pero que pueden ser construidas a partir de sus elementos base.

CAPÍTULO 5. IMPLEMENTANDO UN LENGUAJE DE DOMINIO ESPECÍFICO

5.1 Definiendo el lenguaje

Al momento de implementar un DSL (Domain Specific Languaje) es necesario definir ciertas directrices. Esto con el fin de garantizar que el lenguaje es coherente, no posee una curva de aprendizaje muy difícil, tiene un propósito y un dominio en el que es nativo.

5.1.1 Definiendo el nombre del lenguaje

Lo primero que debemos hacer es darle un nombre al lenguaje esto hará que el lenguaje sea reconocible e identificable. Escoger nombre para un lenguaje de programación es algo arbitrario así que básicamente puede tener cualquier nombre. Un esquema común para el bautizo de lenguajes de programación es con una letra que representa una nota musical en inglés, como todos los lenguajes C que son equivalentes a un Do. Otro esquema es utilizar el nombre de algún referente dentro del área de la computación, matemáticas o física y es precisamente el esquema que usaremos.

El DSL se llamará **Haibt** en honor a Lois Haibt. Ella fue la única mujer dentro del equipo original de 10 personas que desarrollaron FORTRAN un lenguaje de programación conocido como uno de los primeros lenguajes de alto nivel exitosos. Ayudó a analizar el flujo de ejecución de partes importantes de los programas de FORTRAN, así como el desarrollo del primer analizador sintáctico para el lenguaje.

5.1.2 Definiendo el paradigma del lenguaje

El objetivo de Haibt es únicamente generar interfaz gráfica con el patrón de componentes en la web, no tiene intención de generar aplicaciones completas, sino únicamente su interfaz gráfica. Por esta razón utilizar un paradigma de programación como estructural o funcional no son realmente candidatos que puedan integrarse de una manera natural en un DSL. Otros paradigmas como el orientado a objetos y declarativo están mejor adaptados para la tarea de definir interfaces gráficas, sin embargo, se deben de hacer ciertos cambios para poder adecuarlos a las necesidades de Haibt.

En la actualidad con los frameworks más populares de interfaces gráficas se utiliza el patrón de componentes. Un componente web es una unidad reutilizable que encapsula el código JavaScript que le da funcionalidad, el HTML (HyperText Markup Languaje) que define la estructura de la interfaz gráfica en la

página web y los estilos CSS (Cascading Style Sheets) que define la estética del componente. Otra característica importante de los componentes es que como indica su nombre, se pueden componer, unos dentro de otros o simplemente ensamblarlos al lado de otros como piezas de lego. Esto crea estructuras jerárquicas conocidas como árboles de componentes, donde habrá componentes que sean padres de otros, hermanos o hijos.

Como la estructura es jerárquica esto significa que el flujo de información y mensajes puede darse solamente de hacia arriba o hacia abajo. El flujo de información de arriba hacia abajo se conoce como propiedades o atributos del componente. Estos son parámetros o estructuras de datos que el componente necesita para poder funcionar o cambiar su comportamiento según sea necesario.

Por el contrario, cuando el flujo de información ocurre de abajo hacia arriba estos se llaman eventos y funcionan a manera de notificaciones. Cuando ocurre una reacción dentro del componentes disparadas por algún evento ya sea causado por el usuario de la web o no, entonces se dispara una señal de que ha ocurrido. La función que se despacha se llama **listener** o escucha y se invoca con información relacionada con el evento como argumento.

Otra pieza importante del patrón de componentes es el ciclo de vida. Cada componente que se crea pasa por distintas etapas como el momento en el que se crea, se monta, lo que significa que se vuelve visible al usuario y entra dentro de la estructura, cada vez que se actualiza, cuando se desmonta y finalmente cuando se dispone del componente.

Usando estas tres ideas, el flujo de información bidireccional, la estructura jerarquizada y el ciclo de vida se han creado los pilares fundamentales del paradigma que el lenguaje utilizara. Este paradigma será bautizado como Orientado a componentes.

El paradigma orientado a componentes como se ha podido observar tiene bastante similitudes con el paradigma orientado a objetos. Si bien no existen objetos como un constructo o concepto del lenguaje son los componentes los que toman ese rol. Además, vemos que existe la comunicación entre elementos del sistema por medio de mensajes, ya sea por propiedades o por eventos. Existe encapsulamiento ya que el funcionamiento y data interna de los componentes no se revela a los demás. También existen algunas diferencias ya que por la forma fundamental de cómo operan los componentes podemos prescindir de la herencia y el polimorfismo.

Como se necesita reducir la ceremonia, es decir código de inicialización o protocolo en cuanto a creación y composición de la interfaz gráfica, optamos por un paradigma declarativo. Declarar la estructura de la interfaz gráfica con alguna variante de lenguaje de etiquetas como XML (Extensible Markup Languaje) no es una idea nueva, pero si ha probado ser la forma mejor adaptada. Debido a que declara la interfaz como un XML es una idea bastante común y utilizada, la integraremos dentro de nuestro paradigma.

5.1.3 Definiendo el sistema de tipos del lenguaje

Debido a que el resultado final de la compilación de Haibt será ejecutando JavaScript en el navegador y además este será parte de una librería de UI (User Interface) es imperativo que los desarrolladores que la utilicen tengan una experiencia de uso fácil y que la API (Application Programing Interface) de los componentes pueda ser descubierta de manera fácil sin necesidad de leer documentación excesiva para comprender qué funciones o parámetros posee un componente. Para cumplir con este objetivo emplearemos las ventajas de TypeScript.

Typescript un lenguaje de programación que es un superset de JavaScript, su principal ventaja es que se pueden anotar sus elementos con anotaciones de tipo lo que le da información adicional al compilador sobre las propiedades y los tipos de cada componente. Con la metainformación adicional sobre los tipos de los elementos en el programa se facilita que el usuario pueda explorar la API del componente utilizando la función de autocompletado de su IDE (Integrated Development Environment). Las anotaciones de tipo además agregan una capa extra de validación sobre el programa final que desarrolle el usuario.

Para que Haibt pueda generar estas anotaciones de tipos usando TypeScript necesariamente Haibt deberá ser también un lenguaje tipado. La intención principal de Haibt no es crear toda una nueva sintaxis completamente fuera de lo convencional. Haibt más bien tomara lo que se sabe que funciona bien en el paradigma actual de frameworks y hacer más fácil la creación de estos eliminando la complejidad sintáctica de JavaScript. Por esa razón la forma en que se declaran, anotan y utilizan los tipos será idéntica a la forma que lo hace TypeScript. La principal ventaja de hacer esto es que los usuarios que ya utilicen TypeScript ya no tendrán que reaprender cómo declarar y anotar los tipos en el lenguaje. Con toda la metainformación que obtenga el compilador gracias al sistema de tipos será más fácil generar programas en TypeScript que mantengan semántica parecida.

Con respecto a la rigidez del sistema de tipos, Typescript es un lenguaje fuertemente tipado, eso a su vez significa que Haibt heredara esos atributos. Esto significa que Haibt podrá definir tipos personalizados en

el lenguaje pero que las validaciones para corroborar asignación de un tipo a otro se ejecutarán de manera estructura. El sistema de tipado estructura, es decir validar la estructura de los objetos en lugar de sus nombres es un mejor enfoque al lidiar con la naturaleza de JavaScript. Además, como las validaciones de tipos existen al momento de compilar será posible realizar análisis estático sobre el código para asegurar que el programa presente una menor cantidad de errores por acceder a miembros no existente de los objetos dinámicos al momento de ejecutarse.

5.1.4 Definiendo tipos de datos del lenguaje

Los tipos de datos en Haibt funcionaran igual que lo hacen en JavaScript/TypeScript, en esos lenguajes existen tipos de datos primitivos, es decir que están definidos directamente por el lenguaje. Estos son las cadenas de texto, los booleanos, los números enteros o flotantes y literales como nulo o indefinido. Como son tipos nativos tanto en Haibt como en JavaScript la traducción de estos será inyectiva.

Otros tipos de datos como los objetos planos se podrán definir en Haibt usando una definición de tipo. Es la misma idea que en Typescript, por lo que se pueden crear estos objetos planos como constructos del lenguaje. Eso significa que habrá una sintaxis dedicada para crear objetos planos.

5.1.5 Definiendo estructuras del lenguaje

Las entidades más utilizadas al momento de desarrollar interfaz gráfica con los frameworks de JavaScript modernos son 3. Los componentes, los estilos CSS y el Scripting, como son utilizados de forma muy frecuente tiene mucho sentido optimizar el lenguaje para poder definir estas estructuras de manera rápida.

Para los componentes definiremos una estructura con el mismo nombre. Para declarar un componente en Haibt bastará con indicar que algo es un componente con la palabra clave **component** seguido del nombre de dicho componente. Posteriormente dentro del componente se definirán las propiedades, eventos y markup del mismo. Esto prácticamente convertiría a los componentes en un análogo de las clases en otros lenguajes orientados a objetos.

De manera similar con respecto a los estilos CSS se definirá utilizando una palabra clave, **style** y dentro de ellos se escribirán todas las reglas CSS tal cual se escriben en un archivo CSS regular. La única diferencia es que también se podrán anidar clases como ocurre en otros dialectos tales como SCSS (Sassy Cascading

Style Sheets). Eso evita que el desarrollado deba escribir toda la secuencia de especificadores de CSS como se hace regularmente y que este se pueda escribir en bloque anidados para agilizar la escritura.

El scripting en forma de funciones o de lifecycle hooks no se implementará en la primera versión del compilador de Haibt sin embargo quedará definido en la gramática.

Con respecto a las estructuras de control como el if y el else estos se utilizarán como directiva dentro del markup. Una directiva es un atributo especial, cuya función no es pasar información como las propiedades. En su lugar, una directiva afecta a la forma en que se interpreta el markup. En el caso concreto del if es una directiva de tipo estructural, lo que significa que esta altera la estructura del markup resultante después de ser evaluada

5.1.6 Definiendo operadores del lenguaje

Los operadores no aportan ninguna forma de expresividad a la sintaxis, así que sus casos de uso se limitan a los mismos que ya tenían en JavaScript. Eso significa que un '+' en JavaScript servía para declarar la suma de dos operadores y nada más, en Haibt también actúa de la misma manera. Esto es así para los cuatro operadores aritméticos y también aplica al operador de asignación.

5.2 Implementando gramática del lenguaje

Ya que Haibt es un lenguaje con tipado estático similar a TypeScript se ha diseñado con una sintaxis que sea cercana a este, la razón es sencilla, TypeScript proporciona los beneficios de un lenguaje de tipado estático y además es uno de los más populares entre desarrolladores de todos los frameworks. TypeScript es un lenguaje prácticamente idéntico a JavaScript pero que tiene anotaciones de tipo para introducir tipado estático. Incorporar estos elementos de otros lenguajes populares para ciertos constructos como declarar variables o funciones debería de reducir la curva de aprendizaje para programadores que ya usen TypeScript. Esta gramática se basará en la implementación de la gramática para C de ANTLR (GitHub, 20/09/2023)

5.2.1 Tipos de token en la gramática

Si bien todas las palabras que pueden formarse utilizando el alfabeto de lenguaje se pueden generalizar llamándose tokens, no todos los tokens son iguales al momento de implementar una gramática. Los tokens poseen distintos tipos que suelen asociarse para poder separarlos en categorías y para que el parser no deba

de preocuparse de que tipo de token un valor concreto sea. Algunas de las categorías más comunes de tokens son las siguientes:

Operadores, estos son por lo general caracteres provenientes de las matemáticas cuya función más prominente es indicar donde deben ejecutarse ciertas operaciones como suma, multiplicación o asignación por mencionar algunas. Eso no quiere decir que los operadores están limitados a un único carácter, pues de necesitar un operador podría ser incluso un palabra o secuencia de palabras, sin embargo, esto no es práctico y utilizar un único carácter es una forma concisa y clara de utilizar operadores.

Identificadores, estos son palabras que se utilizan como nombres y sirven por lo general para identificar elementos del lenguaje de programación como funciones o variables. Los identificadores también sirven para hacer referencia a elementos ya existentes en el programa para poder usarlos en conjunto con otros, usarlos dentro de otros elementos o usarlos para crear más elementos. Por lo general la forma en que los identificadores son implementados son alias para algún lugar en la memoria del programa, eso permite modificar ya sea lo que contienen o a donde apuntan por medio de la instrucción de asignación.

Palabras clave, las palabras reservadas o palabras clave son palabras en específico que el lenguaje utiliza con fines concretos. Se denominan así porque esas palabras en particular no pueden utilizarse como identificadores ni nombres para ningún otro tipo de token dentro del lenguaje.
Por lo regular las palabras claves se utilizan para declarar estructuras claves dentro del lenguaje de programación, ejemplos de estos son las palabras reservadas para control de flujo como ‘if’, ‘for’ y ‘else’. En muchos lenguajes estas son palabras reservadas y sirven como el inicio de la declaración de estructuras condicionales o de repetición. Estas palabras clave suelen ser el inicio de la regla y suelen ser bastante localizadas es decir solo se utilizan bajo un contexto específico y sencillas donde son el único lugar que se utilizan. En ese sentido son palabras exclusivas para cierto tipo de sentencias.

Otro tipo de palabras reservadas como ‘class’ o ‘function’ suelen ser el preludio hacia bloques de código, el propósito de estas palabras claves entonces es marcar el inicio de estas estructuras que pueden abarcar muchas otras sentencias como parte de su definición. Otras palabras como ‘const’ o ‘let’ se utilizan para declarar variables dentro del lenguaje y así existen otras palabras reservadas que ayudarán a agilizar ciertas tareas dentro de cada lenguaje, otra forma de entender las palabras reservadas es que son sintaxis dedicadas para realizar tareas comunes dentro de los lenguajes y que de otra forma no serían tan fácil.

Los números literales también son otra parte del lenguaje que tienen sus tokens específicos, esto es porque muchos programas necesitan realizar cálculos con números o utilizar valores arbitrarios de números como banderas o enums que tienen un significado único dentro de la lógica del programa. Por esta razón los lenguajes de programación tienen formas de poder indicar un valor concreto de un número, así por ejemplo el carácter en texto '5' de hecho representa el valor 5 aritmético. Los números literales tiene múltiples usos dentro de los programas, aunque alguno de los más comunes es, instruir ejecución un ciclo cierta cantidad de veces, definir valores de las constantes, comparar valores con estos.

Los números literales suelen clasificarse según su escala de presión, los que representan solamente números enteros conocidos como 'integers' del inglés, números flotantes 'floats' que suelen tener 32 bits de precisión, los 'double' que utilizan 64 bits y así otros tipos de números con más precisión. Esta distinción con respecto a la precisión tiende a afectar a lenguajes de programación con manejo de memoria más estricto, en el caso de JavaScript y por consiguiente de Haibt no hay necesidad de distinguir entre la precisión de los números ya que esa complejidad es manejada por el navegador.

Los textos literales o 'strings' son otros tipos de tokens cuya función es representar texto, pero no como parte del lenguaje de programaciones sino literalmente transcribir el mensaje al programa. Los strings suelen manejarse dentro de los lenguajes de programación generalmente entre comillas dobles o simples. Este tipo de tokens suelen usarse para mostrar mensajes al usuario sobre el estado o acciones del programa, para comparar entrada de teclado con ciertos valores, comprobar si un patrón se encuentra dentro de otro mensaje y varios casos de uso más.

Otros valores literales también suelen implementarse como tokens ya que representan valores o tipos nativos, lo que quiere decir que no se pueden definir en función a otro tipo, sino que son parte del lenguaje. Ejemplos de estos son palabras como 'void' o 'null' que suelen usarse para indicar que una función no retorna nada o que en el caso de 'null' una variable se encuentra vacía o nula. Al utilizar TypeScript también existen tokens especiales para los tipos de datos nativos como 'number', 'string' y 'bool'. Otros tipos de valores constantes son los datos de tipo booleano ya que existe un token para el valor literal 'true', y otro para el valor literal 'false' en prácticamente todos los lenguajes de programación.

La puntuación del lenguaje de programación también necesita la creación de tokens específicos. Dentro de la categoría de puntuación para un lenguaje de programación se encuentran caracteres como '{' y '}' que suelen utilizarse para agrupar sentencias en grupos conocidos como bloques de código o definir las sentencias que pertenecen a una función en otra estructura llamada cuerpo de la función.

Caracteres como la coma ‘,’ sirve para separar elementos en listas o miembros de un objeto, los caracteres ‘[‘ y ‘]’ suelen utilizarse en situaciones donde se necesita acceder a un elemento específico de una lista de valores, esto se llama indexación y otro de los más comunes es el ‘;’ que sirve para delimitar el fin de una sentencia ya que estas pueden abarcar cantidades arbitrarias de palabras o líneas.

El propósito de este tipo de tokens suele ser de dos tipos, como sincronización para permitir que el parser del programa sea más fácil y como definición del alcance en el caso de los bloques de código. Esto permite a los programadores poder tener un código con estructura visual la cual también representa fielmente su estructura en el programa final lo que permite un entendimiento más completo de la forma que los programas operan al momento de su ejecución.

5.2.2 Implementando gramática del lenguaje

Con las directrices principales del lenguaje, ser un DSL, tipado estático, gramática LL 1 y paradigma orientado a componentes ya se puede establecer los elementos específicos que conforman la gramática del lenguaje. La gramática se creó en base a la gramática del lenguaje C para los elementos comunes como las expresiones de todo tipo, lógicas y aritméticas (lista completa de producciones gramaticales en Anexo A). Dado que la gramática es LL 1 lo primero será crear una regla que actúe de raíz o inicio, en este caso la llamaremos “program” y la definiremos como se indica en la Figura 23:

```
program: module EOF;
```

Figura 23. Ejemplo de programa, Fuente: Creación propia

La regla gramatical program se deriva en otra regla llamada module seguida del fin del archivo. Por su parte la regla module será la que utilicemos para poder definir componentes o estilos CSS dentro de nuestro programa una o múltiples veces. Como se necesita repetir una cantidad indeterminada de modules entonces se define como muestra la Figura 24.

```
Module: (component | style | typeDef) module |;
```

Figura 24. Definición de module, Fuente: Creación propia

Esto significa que un module en la gramática es cualquier secuencia de un componente, un estilo o una definición de tipo seguida opcionalmente de otro módulo.

5.2.3 Implementando gramática de un componente

Como el lenguaje se fundamenta en la creación de componentes web entonces lo primero será definir la forma en la que los componentes serán declarados por el programador. Para poder diferenciar exactamente qué tipo de módulo se va a declarar agregaremos la palabra clave **component** siempre antes del nombre de cada componente, y entre llaves se escribirán todas aquellas sentencias de código que describan las propiedades, eventos y maquetado del componente.

Los componentes son la parte más importante del lenguaje Haibt por lo que también tendrán bastantes estructuras sintácticas para ellos. Como describimos anteriormente los componentes necesitan piezas de información llamadas propiedades que son pasados a estos como parámetros. En otros frameworks anotar que estructuras son una propiedad es un trabajo un tanto engorroso. Como todos los frameworks son creados sobre JavaScript dependen de la sintaxis y los mecanismos que este tenga. Por ejemplo, React necesita que se envíen como los argumentos de un constructor o que sean los argumentos de una función. Otros como Vue necesitan que se declaren objetos con la forma, tipos y nombres de cada propiedad lo que es un trabajo manual ya que no existe nada que lo automatice.

Para reducir el trabajo manual y que sea más fácil añadir, quitar o modificar propiedades de los componentes, añadiremos otra palabra clave, **prop** que se coloca al inicio de la declaración de cada propiedad. Esa palabra clave le indica al compilador que lo que sigue será una propiedad y tendrá comportamientos específicos. Por ejemplo, todas las propiedades son públicamente accesibles lo que quiere decir que se pueden asignar desde fuera del componente. Otra característica que tendrán las propiedades es que no se podrán modificar desde dentro del componente, ya que ningún framework soporta modificación a través de propiedades, únicamente de eventos, por lo que el compilador reforzar el uso correcto estableciendo a las propiedades como variables que son de solo lectura, es decir no se puede alterar su valor a través de una asignación directa.

Dado que Haibt es un lenguaje tipado, también necesitamos una forma de declarar los tipos de datos que estas propiedades tomarán. Para esto tomaremos la sintaxis de TypeScript como ejemplo. En TypeScript para declarar el tipo de una variable se escribe ‘:’ dos puntos justo después del nombre de la variable y luego una palabra clave como ‘string’ para cadenas de text, ‘number’ para números o un identificador que sea el nombre de algún tipo declarado previamente por el programado. Otra importante decisión con respecto a la declaración de propiedades es la capacidad de decidir si estas tendrán un valor por defecto, lo que las convierte en opcionales. Esto es importante dado que los componentes tienen un número arbitrario de

propiedades no sería cómodo ni eficiente hacer que un programador pase todas y cada una de las propiedades cada vez que necesite usar ese componente. Para mitigar esto la sintaxis permitirá la inicialización de propiedades al momento de su creación. Esto tiene el efecto de asignar un valor por defecto de tal forma que no siempre tenga que estar definido desde fuera.

Colocando todo esto junto podríamos definir la declaración e inicialización de una propiedad con la siguiente sintaxis indicada en Figura 25.

```
propDeclaration: Prop Identifier Colon varType initValue SemiColon;
```

Figura 25. Definición de Prop Declaration, Fuente: Creación propia

Donde initValue será otra regla que permita colocar el operador de asignación '=' seguido de una expresión que retorne el valor inicial de la propiedad.

La creación de variables y constantes es una característica necesaria para poder crear programas sofisticados además de ser vitales para la existencia de muchos algoritmos. En Haibt se podrán definir variables o constantes utilizando las palabras clave **var** o **val** respectivamente. Lo siguiente será una anotación de tipo utilizando ':' y el identificador del tipo y finalmente la inicialización opcional. Similar a las propiedades cualquier cosa que se declare con valor no se podrá modificar después de su creación, pero contrariamente a las propiedades las constantes no son visibles por fuera del componente por lo que no se puede saber su valor. En el caso de las variables estas siguen sin poder ser visibles desde fuera del componente y además se puede alterar su valor después de ser creadas. La única excepción notable a esto es intentar asignar un nuevo valor a una variable desde el método render que es donde se imprime el HTML hacia la página web.

Similar a la implementación de las propiedades las variables y constantes pueden ser inicializadas utilizando el operador de asignación '=' y luego escribir una expresión que le asigne su valor inicial. Para el caso particular de las variables ya no es necesario indicar nuevamente que son variables utilizando la palabra reservada **var** al intentar asignar un valor nuevo, basta con indicar su nombre el operador de asignación '=' y luego la expresión con un nuevo valor.

Lo siguiente será una forma de poder definir el maquetado, es decir el HTML que permita crear la interfaz gráfica dentro del navegador web. Para lograr esto crearemos una función del ciclo de vida llamada **render**.

Las funciones de ciclo de vida son funciones que se ejecutan en momentos determinados dentro del esquema de ciclo de vida de los componentes. Este esquema tiene como mínimo una etapa de creación y una etapa de disposición o eliminación. Otras etapas dentro del ciclo de vida pueden incluir, pero no se limitan a cuando el componente se vuelve visible por primera vez, cuando sale de la vista del usuario y cada vez que algo se actualiza.

En este último caso el método render necesita ser ejecutado cada vez que se actualice alguna de sus dependencias. Con respecto a qué cosas califiquen como dependencias serían todas las variables que se utilicen para interpolar valores en la plantilla HTML o valores que se pasen como propiedades a componentes hijo. Afortunadamente esta complejidad de llevar seguimiento de las dependencias ya es una tarea que resuelven todos los frameworks de JavaScript, sin embargo, todos la manejan un poco diferente, es por esto por lo que vamos a abstraer ese mecanismo y dejaremos que el compilador implemente esa parte por nosotros. En lo que al programador respecta solamente deberá comprender que ese método se ejecuta múltiples veces, en concreto cada vez que una dependencia cambie.

Lo que se definirá dentro del método render será directamente el HTML que servirá para mostrar en el navegador web. Para definir cómo se utilizará esta plantilla HTML utilizaremos la sintaxis de JSX (JavaScript XML) que ya es bastante usada tanto en Vue como en React. Esta sintaxis es cómoda para desarrolladores gracias a que permite pasar propiedades como atributos de HTML para tributos de tipo string entre comillas y adicionalmente permite hacer algo llamado **binding**. El binding consta de asociar un valor o la referencia a un valor en un atributo HTML, esto sirve para poder enviar por propiedades valores que no son booleanos o strings. Por ejemplo, enviar un número o una variable que almacene un número requería de binding, pero también cosas más complejas como objetos de JavaScript, esto incluye objetos planos y funciones e incluso otros componentes.

```
<button disabled id=”button” aria-valuemax={5} onClick={onCLick}>
  Button Text
</button>
```

Figura 26. Ejemplo botón JSX, Fuente: Creación propia

En el fragmento de Figura 26, el código JSX se puede observar cómo los valores tipo booleanos y los de texto se pueden asignar de la misma forma en la que se hace con HTML plano, pero otros valores que requieren tipos distintos como número en ‘aria-valuemax’ usan la sintaxis de binding que involucra abrir con ‘{‘, luego escribir la expresión que se desea vincular y luego terminar con ‘}’.

En Haibt usaremos la misma sintaxis de JSX añadiendo dos mejoras. En JSX por defecto se asigna true a las propiedades de tipo booleano con tan solo escribir el nombre de la propiedad. Sin embargo, esto solo funciona con propiedades booleanas, con propiedades de otro tipo se debe de escribir la sintaxis de data binding completa, aunque estas tengan el mismo nombre, esto es bastante común as que en Haibt se añadió la capacidad de justo hacer eso. Esto significa que el en JSX personalizada de Haibt se pueden asignar valores de variables a un atributo sin importar el tipo siempre que este tenga el mismo nombre utilizando una abreviatura.

```
<button {onCLick}>
    Button Text
</button>
```

Figura 27. Botón Haibt JSX, Fuente: Creación propia

En el ejemplo de Figura 27 agregamos la capacidad a JSX de poder asignar el valor de una variable a una propiedad del mismo nombre simplemente encerrando entre llaves.

La otra mejora tiene que ver con una limitante de JSX en React y es que React solamente puede renderizar un elemento a la vez por componente lo que quiere decir que, si deseamos mostrar dos botones en un solo componente, no es posible, a menos que se rodeen ambos botones con otro elemento HTML. Como no siempre es posible usar un elemento HTML para funciones como contenedor React introdujo el concepto de fragmento. El cual es un elemento del framework que no renderiza nada en la página web, simplemente se usa para sortear esta limitación. Esto resulta ser una verdadera molestia ya que muchos formateadores de código añaden automáticamente más añadir fragmentos al código JSX es una tarea que debe hacerse manual cada vez, por lo que después de cierto número de intentos se vuelve algo engorroso.
Otros frameworks como Vue o Angular no tienen esta limitación y Haibt tampoco debería tenerla por lo que de nuevo usaremos al compilador para que genere el código de fragmento de manera automática si es necesario.

El resto de las sintaxis de JSX trataremos de mantenerlo intacto ya que como se ha explicado anteriormente JSX es bastante popular entre desarrolladores web actualmente así que para reducir la curva de aprendizaje simplemente lo utilizaremos tal cual es y se añadirá las dos mejoras anteriores.

Por lo que al juntar toda la sintaxis para el método render se puede definir como la palabra clave **render** seguida de ‘(‘paréntesis de apertura y en medio podremos escribir el JSX necesario, indicar **undefined** es

decir no renderizar nada o simplemente dejarlos vacío con un ‘)’ paréntesis de cierre. Este será el único método de ciclo de vida que tenga una sintaxis distinguida. El resto serán simplemente identificadores los cuales el compilador validará si existen o no ya que el resto de los métodos ejecutarán JavaScript que será agregado por los desarrolladores.

```
Render: Render Oparen render Follow;
renderFollow: (Undefined | template) Cparen SemiColon;
```

Figura 28. Regla de producción para método render, Fuente: Creación propia

La Figura 28 respecta a template, esto sería cualquier JSX válido que puedan escribir dentro de los paréntesis. Incluyendo binding e interpolación.

5.2.4 Implementando gramática de tipos

Para implementar una gramática de tipos se pudo haber desarrollado una gramática más llamativa sin embargo el acercamiento de TypeScript para declarar tipos es bastante conciso y además permite expresar la complejidad de la naturaleza de tipado dinámico de JavaScript bastante bien. Por esa razón y que además es una notación de tipos bastante usada se implementara de la misma forma dentro del lenguaje Haibt, pero solo de manera básica. De la característica de tipos de TypeScript se implementará tipos con propiedades opcionales y la validación de los tipos será estructural.

```
Type User = {
  name: string;
  lastName: string;
  age: number;
}
```

Figura 29. De la declaración de un tipo en Haibt, Fuente: Creación propia

En Figura 29 se muestra otra característica de los tipos en Haibt al igual que en TypeScript es que la declaración de un tipo puede incluir una referencia a otro tipo, siempre y cuando no haya una dependencia circular.

5.2.5 Implementando gramática de estilos

Para evitar tener una gramática con muchas palabras reservadas o con muchas reglas de producción la implementación de la gramática de los estilos serán los más simplificada posible, para lograr esto se toman en cuenta las reglas más usadas de CSS y se trata de abstraer los elementos más utilizados hasta su forma indivisible. Una vez con una lista de elementos se trata de encajarlos en patrones para cada una de las reglas.

De forma más concreta podemos dividir la gramática de CSS en dos secciones, una donde se listan la combinación específica de todas las clases de CSS a la que aplican los estilos incluyendo todos los modificadores y otra dentro de llaves donde se declaran los estilos propiamente dichos.

En lo que a selectores respecta, estos están compuestos por identificadores, puntos '.' Y el signo mayor que '>'. La limitante con respecto a CSS normal es que los identificadores en CSS pueden contener guiones bajos '_' y guiones normales '-', pero en la gramática de Haibt ya definimos que los identificadores no pueden poseer guiones normales '-' ya que ese carácter se usa como el operador resta, por lo tanto, un identificador separado por guiones normales '-' se podría interpretar erróneamente como la resta de dos identificadores. Soportar ambos estilos de identificadores es un problema bastante complejo por lo que para mantener las cosas simples no se admitirán este tipo de identificadores en primer lugar. Como resultado la lista de modificadores que contenga guiones normales '-' en sus identificadores se tendrán que reescribir usando camelcase o guión bajo en su lugar. Por ejemplo, si la regla se llama originalmente 'button-class' en Haibt deberá llamarse 'butonClass' o 'button_class'.

Otro elemento que deseamos ignorar explícitamente para simplificar la gramática son los espacios entre los nombres de dos clases ya que en CSS normal las clases inician con un punto en su nombre y dependiendo de si existe un espacio entre un nombre de clase y el siguiente el efecto de los selectores pasa de ser clase dos dentro de la clase uno a un elemento con la clase uno y la clase dos simultáneamente. Dado que el analizador léxico que implementaremos ignora los espacios y construir una gramática que tome en cuenta cantidades arbitrarias de espacios también es bastante trabajoso se prefiere resolver este detalle de implementación durante el análisis semántico. Para corregir esto basta con ver los índices de inicio y final de cada token si incrementan en un solo carácter eso quiere decir que los nombres de clase están concatenados, si es más de uno entonces existe espacio entre ellos y deben procesarse de manera distinta. Con el resto de los elementos que pueden aparecer en la parte de lista de especificadores de CSS no existen más problemas.

Para la lista de reglas de CSS se tiene el mismo problema de los identificadores los cuales no pueden tener guiones normales en sus nombres, esto se resolverá de la misma forma que la lista de selectores, es decir forzando identificadores únicamente con letras y guión bajo al inicio. Las reglas de CSS se escriben como un identificador seguido del carácter ‘:’, seguido de una o varios de los siguientes posibles casos:

Un número que tenga algún valor con una unidad de forma opcional. Ejemplos de esto son los paddings, márgenes y tamaños de fuente. Pero también existen reglas que solo deben indicar un número sin necesidad de una unidad de medida. Estas validaciones se harán en el analizador semántico para reducir la complejidad de la gramática.

Un identificador, que puede ser alguno de los posibles valores que tomen reglas como **display** o **text-transform**. Nuevamente para no introducir estos valores específicos como palabras clave, se generalizó la gramática a cualquier identificador válido. Los valores válidos admitidos por cada regla se validarán de forma diferida en el análisis semántico.

Colores en código hexadecimal, estos si son procesados directamente en la sintaxis y al igual que en CSS para que los códigos CSS hexadecimales de colores sean válidos estos deben consistir en 3 caracteres o 6 caracteres que solamente utilizan números del cero al nueve y letras entre la ‘A’ y la ‘F’ sin distinción de mayúsculas. Para indicar el inicio de un color debe usarse el numeral ‘#’ al inicio seguido de los 3 o 6 caracteres hexadecimales que representan el color deseado.

Estilos anidados, en el uso de CSS normal es común que las listas de especificadores se vuelvan bastante larga debido a que a mayor especificidad por lo general el camino detallando la secuencia de selectores tiende a incrementar. Eso resulta normalmente en líneas sumamente largas describiendo los selectores. Para combatir con esta secuencia de especificadores siempre creciente han surgido varias alternativas a escribir CSS puro como SCSS que implementan un sistema donde los identificadores se pueden colocar unos dentro de las reglas de otros, esto es una sintaxis abreviada de escribir la lista completa y al momento de generar CSS plano nuevamente dado que es lo único que el navegador web soporta los pre procesadores anteriormente mencionados se encargan de resolver y concatenar toda la serie de especificadores. Naturalmente esto resulta en un código más compacto que al final es exactamente igual que lo que se hubiese escrito a mano por un desarrollador.

5.2.6 Implementando gramática expresiones

Los lenguajes de programación tienen un tipo de sentencias especiales llamadas expresiones. Las expresiones son formas de combinar elementos del lenguaje con operadores y al final de esa operación obtener un valor de resultado, este puede ser almacenado para uso posterior, inmediatamente usado o compuesto en una expresión aún más compleja (Aho, Lam, Sethi, & Ullman, 2007).

Existen expresiones de varios tipos, existen las de asignación donde se asigna un valor en una variable, en este caso el valor resultante de la expresión es el valor que se asignó a la variable. Otras expresiones son del tipo algebraico utilizando los simples operadores como la suma y la multiplicación. También existen expresiones de tipo lógico donde por lo general el resultado es un valor verdadero o falso, pero en el caso de JavaScript esto no siempre es así, además pueden existir expresiones unitarias, son todas aquellas expresiones que solo necesitan de un miembro y un operador, al contrario del resto de expresiones mencionadas hasta el momento que necesitan un miembro a la derecha y otro a la izquierda para poder operar.

Las expresiones no necesariamente involucran el uso de operadores ya que existen expresiones primarias como por ejemplo un identificado por sí solo o expresiones que se llaman expresiones literales que son todas aquellas expresiones que resuelven a un valor de token literal como los strings o números. Además, existen expresiones compuestas como cuando se indexa un valor en la lista, ya que lo que indexa la lista no necesariamente debe ser un número, sino que puede ser un identificador que apunta a un número o una llamada a función que retorna el número.

Las expresiones también tienen la capacidad de componerse y agruparse es decir que una expresión no está necesariamente limitada a un operador, sino que puede tener múltiples operadores al mismo tiempo y cada operador puede repetirse múltiples veces, así por ejemplo si uno desea sumar tres números no es necesario hacerlo en dos pasos, sino que puede hacerse en una sola expresión. Los operadores son también esos elementos que permiten componentes expresiones, por esa razón casi siempre se encuentran entre dos términos, los operandos, normalmente se suele referir a ellos como derecha e izquierda.

Cuando una expresión se compone de uno o más operadores esta se puede modelar como una forma de árbol donde el nodo central es el operador y en este se conectan dos, el operando de la izquierda y el de la derecha, como estos nodos a su vez puede ser expresiones esta estructura se replica de manera recursiva hasta que llegamos a las expresiones de tipo primario. Expresiones primarias son aquellas que son consideradas

expresiones por sí mismas como los identificadores y valores literales, por ejemplo. Sin embargo, no es factible modelar el árbol de expresiones únicamente teniendo en cuenta el orden en que se presentan escritas en el programa, sino que también es necesario tomar en cuenta la precedencia de operadores, así habrá operadores que siempre deben ser procesados antes que otros.

Por ejemplo, el operador de asignación es el de más baja precedencia dado que su naturaleza solo es asociar valores a identificadores. Por otro lado, expresiones como las multiplicativas siempre tendrán precedencia sobre la suma y la suma sobre la asignación. Esto es con el fin de proporcionar resultado consistente en las operaciones sobre las expresiones y no que estas dependan de la forma en que el programador las escriba. Por esa razón una de las formas más fáciles de asegurarse que las expresiones se traduzcan correctamente en el compilador es delegando esta responsabilidad al parse, esto quiere decir que este tipo de comportamiento de precedencia de unas operaciones sobre otras debe estar diseñado directamente en la gramática.

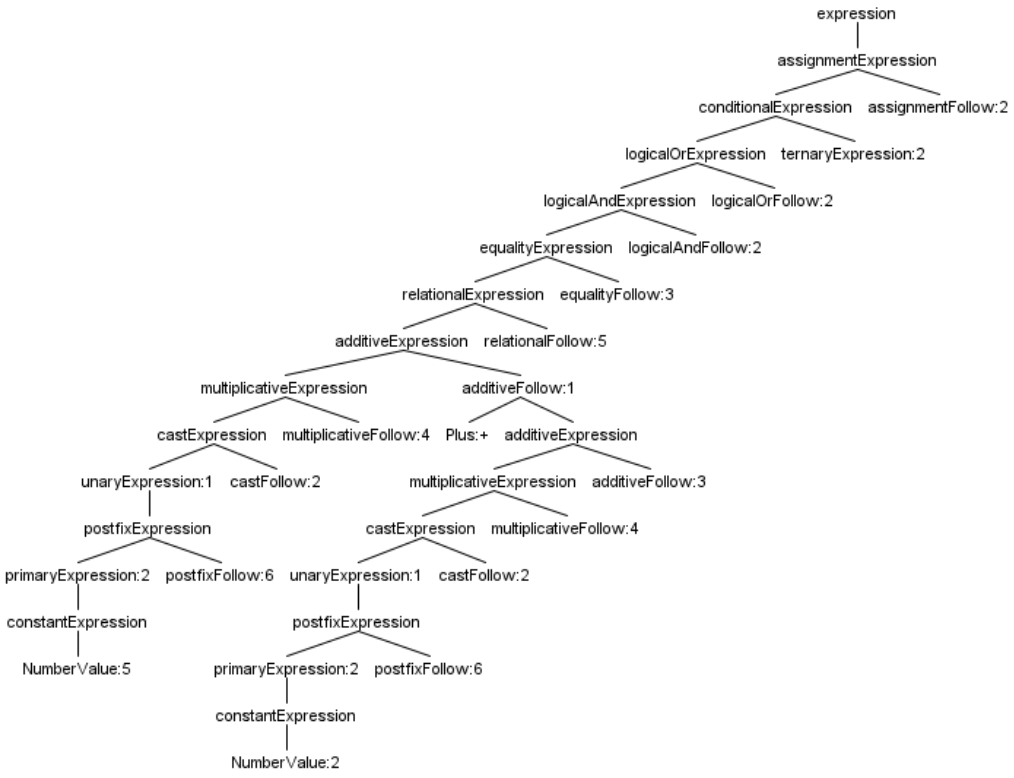

Figura 30. Árbol de expresiones en Haibt para la expresión 5 + 2, Fuente: Creación propia

En el árbol de sintaxis en Figura 30 se puede observar como las expresiones van ensamblando desde la más general llamada **expresión** hasta la más específica como lo pueden ser los valores literales 5 y 2 en la suma. Otro patrón importante es que la precedencia de operaciones se representa en el árbol como niveles de

jerarquía, así pues, las operaciones con mayor presidencial como las aritméticas se encuentran más abajo en el árbol que las operaciones con menor precedencia que se encuentran en los niveles superiores. La razón de este orden no intuitivo es que al momento de procesar el árbol estos se procesan desde los nodos más profundos hacia los más superficiales.

Habrá ciertos casos donde se necesite que algunas operaciones se procesen antes que otras y estas no sigan el orden de precedencia establecido, para ese tipo de casos existe el agrupamiento. El agrupamiento permite convertir lo que originalmente era parseado como dos expresiones, a una sola, afectando así el orden en que se evalúan. El único operador de agrupación en la gramática de Haibt son los paréntesis. Al agrupar las expresiones podemos influir en el parser para que también modifique la forma en que ensambla el árbol de sintaxis y por consiguiente del orden como se ejecutan en el programa final.

Por ejemplo, en el caso donde sea necesario realizar las sumas antes que una multiplicación, se podría rodear la suma con paréntesis y esto causaría que el parser deba resolver primero la expresión entre paréntesis, efectuado así un cambio en el orden de precedencia. Sin embargo, dado que la gramática es LL1 esta capacidad viene al costo de crear árboles de sintaxis extremadamente profundos. Una expresión relativamente simple como ‘(5 + 2) * 3’ genera el siguiente árbol.

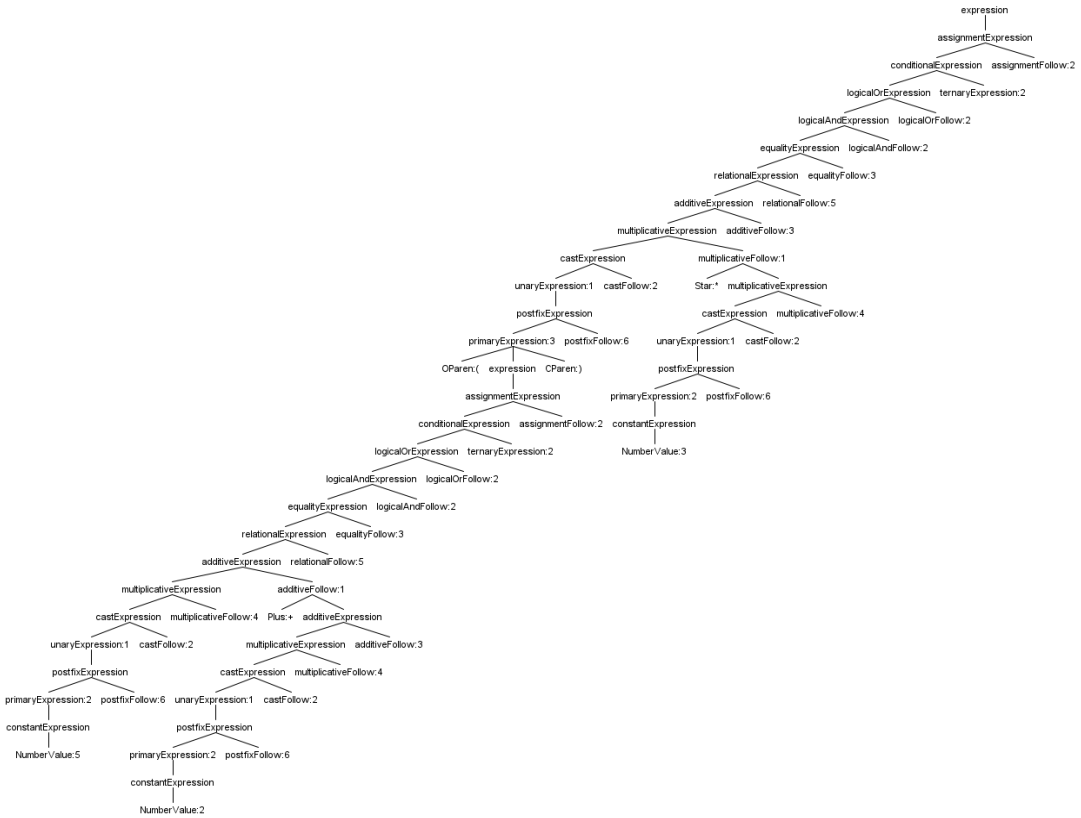

Figura 31. Árbol de expresiones en Haibt para la expresión (5 + 2) * 3, Fuente: Creación propia

Como ilustra Figura 31 dadas las características de la gramática las opciones para lidiar con esta complejidad son limitadas, particularmente porque muchos de los nodos resultan vacíos y realmente no poseen información. En su lugar optamos por simplificar este árbol en una siguiente etapa, en la representación intermedia.

5.3 Implementando representación intermedia

Una vez ensamblado el árbol sintáctico sin errores, ya es posible entrar en etapas más avanzadas del proceso de compilación. Hasta este punto lo que ha ocurrido es que pasamos el programa escrito por el desarrollador de una forma escrita a una estructura en la memoria de la computadora que podemos recorrer programáticamente e interpretar semánticamente las acciones que el programador ha descrito en las sentencias de su programa.

No constante, aunque el programa técnicamente puede ser analizado en el estado actual, es muy trabajoso de analizar debido a que el árbol de sintaxis como tal contienen varios elementos que no son relevantes al momento de estudiar la semántica de un programa. Algunos de estos elementos que podemos eliminar para simplificar el proceso de análisis semántico que es la etapa siguiente serían los tokens de puntuación. La puntuación es bastante útil durante el análisis de la sintaxis del programa esto se debe a que ayudan a marcar la estructura con los acentos que el programador haya definido, sin embargo, el hecho que exista un punto y coma al final de una sentencia realmente no significa nada. Por esta razón todos los tokens que actúan como separadores o sincronización y delimitadores se pueden eliminar del árbol sintáctico.

Otro grupo de elementos que pueden quitarse de manera segura del árbol sintáctico sin alterar la semántica del programa serían todas las producciones que terminan con la palabra vacía. Las producciones que terminan en palabra vacía son un mecanismo que se emplea en el parser para poder parsear la existencia de elementos opcionales. Si en el árbol de parseo la rama termina con la palabra vacía eso significa que la producción de ese subárbol nunca se desarrolló y por tanto es posible eliminarlo sin repercusiones.

5.3.1 Simplificación de expresiones

Los árboles de expresiones producidos por la gramática de Haibt son bastante profundos y tratar de analizar estos árboles innecesariamente profundos e intrincados no es muy eficiente. Debido a que el parser debe de mantener la precedencia de operadores, pero al momento de ensamblar el árbol no se conoce ni el fin ni la longitud de la expresión, siempre se asume que la expresión será de asignación, que es la más general.

A partir de esta expresión general, se comienza de descender recursivamente en las expresiones más específicas hasta que se llega a una expresión atómica, es decir poseen un único token. Es a partir de este punto que el parser regresa hasta el inicio de la expresión original llenado el resto de los elementos de las demás expresiones intermedias si estos existen.

Para evitar trabajo innecesario podemos ejecutar la simplificación de la siguiente manera. Para todas las expresiones que se compongan de un operador y dos operandos podemos examinar primero el operador izquierdo y simplificarlo si es posible, al terminar revisamos si esta expresión posee un miembro derecho, si el miembro derecho es la palabra vacía se concluye que no hay una expresión como operando derecho y por esa razón el resultado de la expresión depende únicamente del miembro izquierdo. Para efectuar esta simplificación habría que modificar el árbol de tal manera que la expresión original sea reemplazada por su operando izquierdo, reduciendo así la profundidad del árbol en un nivel. Este proceso se realizará de forma recursiva en pre-order para simplificar los nodos del más profundo al más superficial. El árbol resultante sería como mucho la expresión más específica que logre describir la operación que estaba indicada en el programa.

Para las expresiones primarias y unitarias no existe ninguna simplificación por lo que el resultado es únicamente eliminar los tokens de puntuación y palabras clave innecesarias.

Las expresiones agrupadas en sí mismas no pueden ser simplificadas exceptuando el caso donde únicamente agrupen una y solo una expresión primaria o una expresión de agrupación. Para los demás casos las expresiones agrupadas se simplifican analizando cada uno de sus términos y simplificando los de manera individual.

Para mostrar el resultado de las simplificaciones se plantea el siguiente código fuente Haibt de ejemplo en Figura 32.

```
component MyComponent {
   prop text: string = "";
}
```

Figura 32 componente MyComponent en Haibt, Fuente: creación propia

El código de ejemplo en Figura 32 es un componente con una propiedad llamada text inicializada en un valor por defecto de un string vacío. El árbol de parseo para esa entrada es el que se muestra en Figura 33

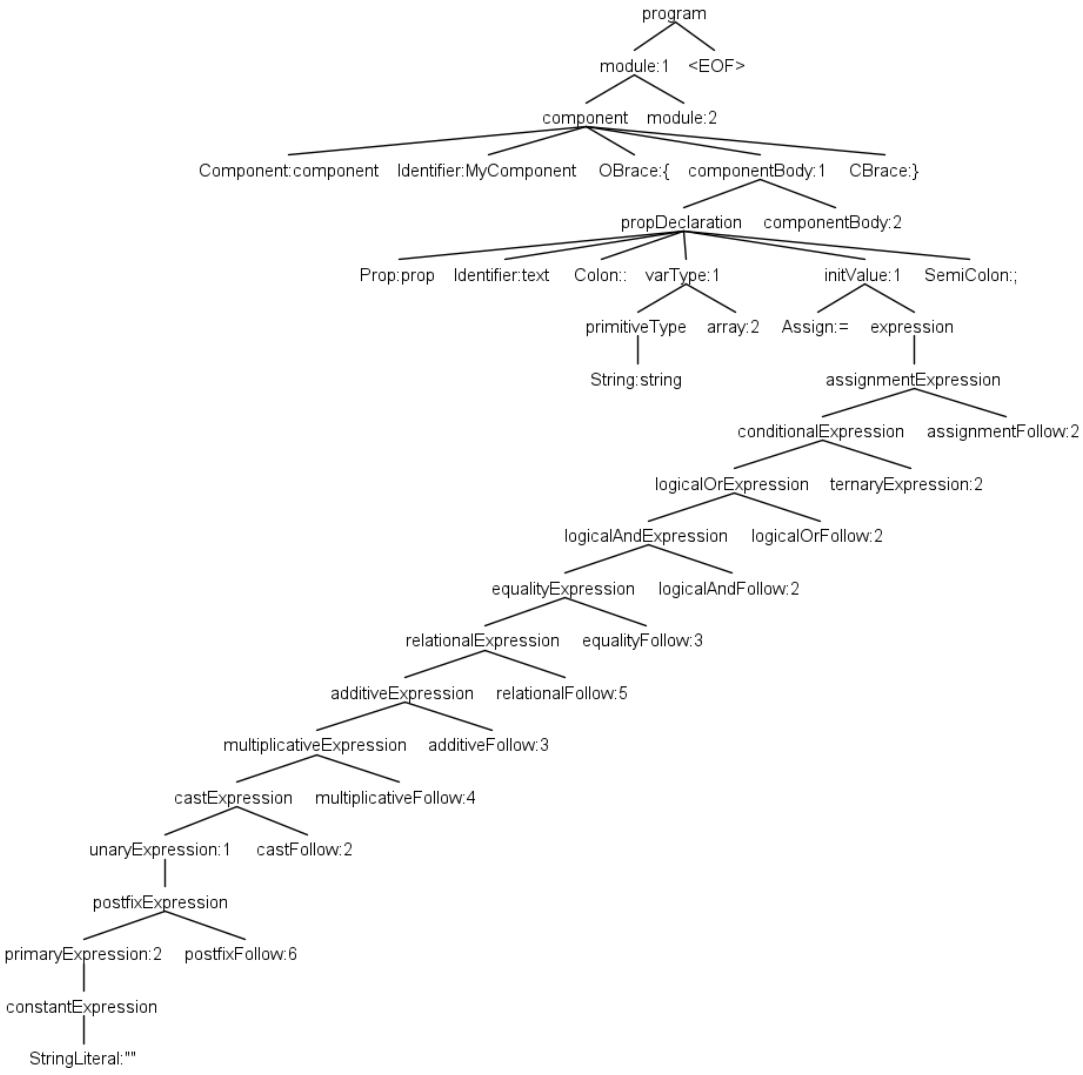

Figura 33. Árbol de parseo para componente MyComponent, Fuente: creación propia.

Como se puede observar el árbol de parseo ensambla todas y cada una de las producciones derivadas de las expresiones hasta llegar a las expresiones primarias, en este caso una expresión constante de tipo string. Luego de ejecutar el proceso de simplificación el árbol anterior toma una nueva forma, como se muestra en Figura 34.

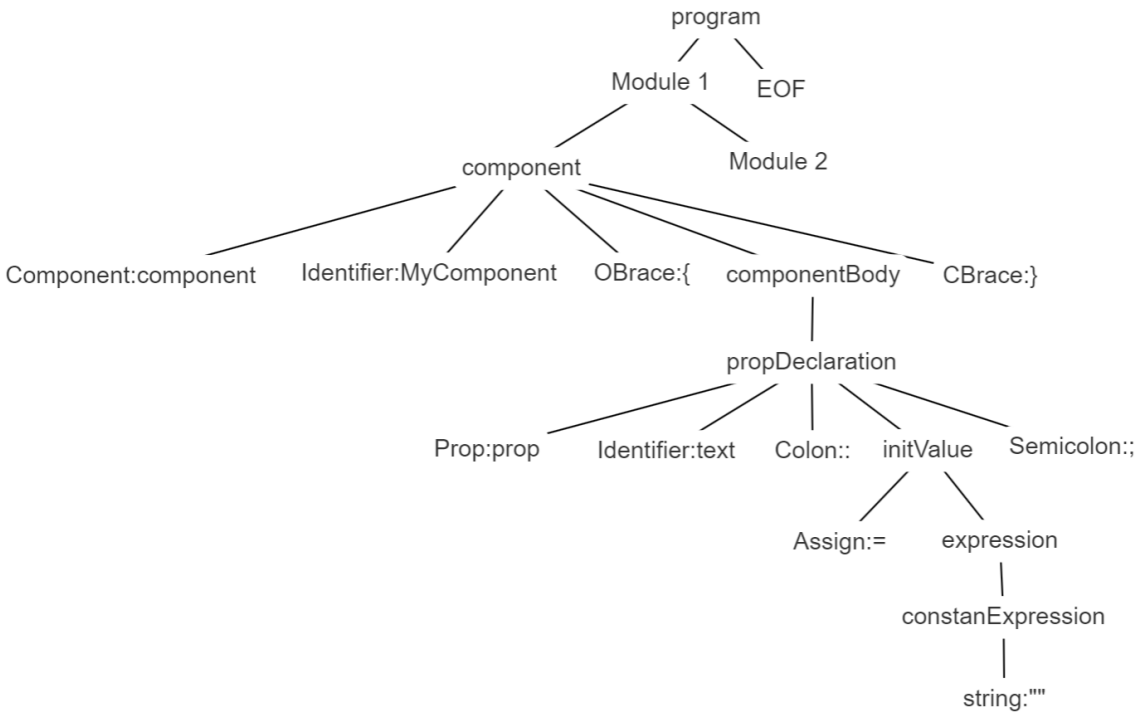

Figura 34. árbol de sintaxis abstracto simplificado de MyComponent, Fuente: creación propia.

La parte inicial del árbol ha quedado igual debido a que no es una expresión, sin embargo, en el nodo initValue se puede apreciar un severo recorte en el árbol de expresiones. Prácticamente el proceso de simplificación ha logrado recortar todos los tipos de expresiones intermedias y concluir que es una expresión constante y nada más.

Este proceso se puede aplicar a cualquier tipo de expresión. Las expresiones que resultan del uso de dos operandos y un operador como en el caso de las expresiones aritméticas o relacionales también pueden verse simplificadas. Esto se logra simplificando cada una de sus ramas, un ejemplo de simplificación para una expresión se desarrollará a partir del código mostrado en Figura 35.

```
component MyComponent {
    prop text: number = 5 + 2;
}
```

Figura 35. simplificación de una expresión aditiva, Fuente: creación propia.

El árbol de parseo para el código de la figura 35 es el que se muestra a continuación en la Figura 36.

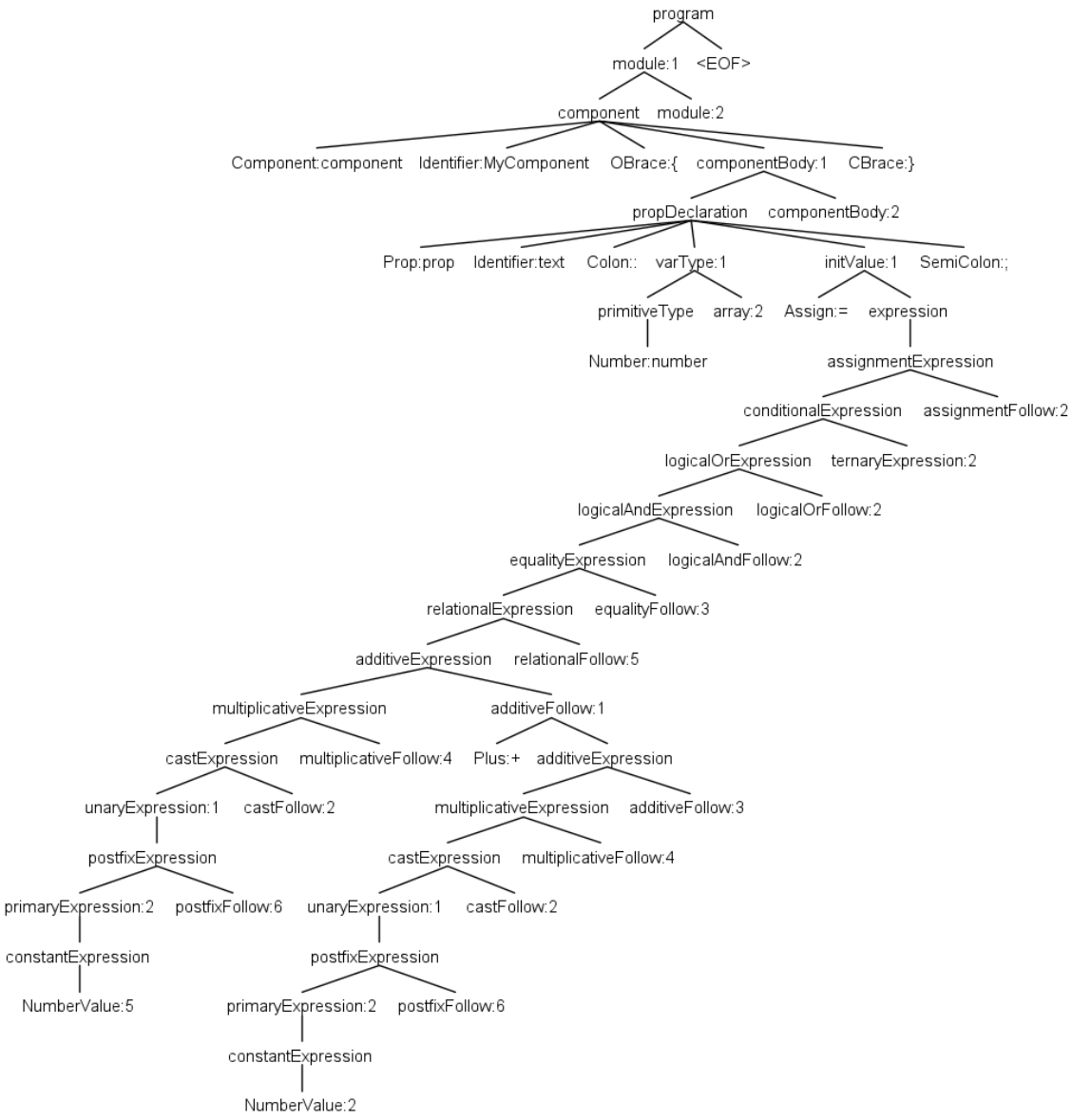

Figura 36 árbol de parseo para código de una expresión aditiva, Fuente: creación propia

Tras simplifica el árbol de parseo de la Figura 36 el resultado es como se muestra en la figura 37.

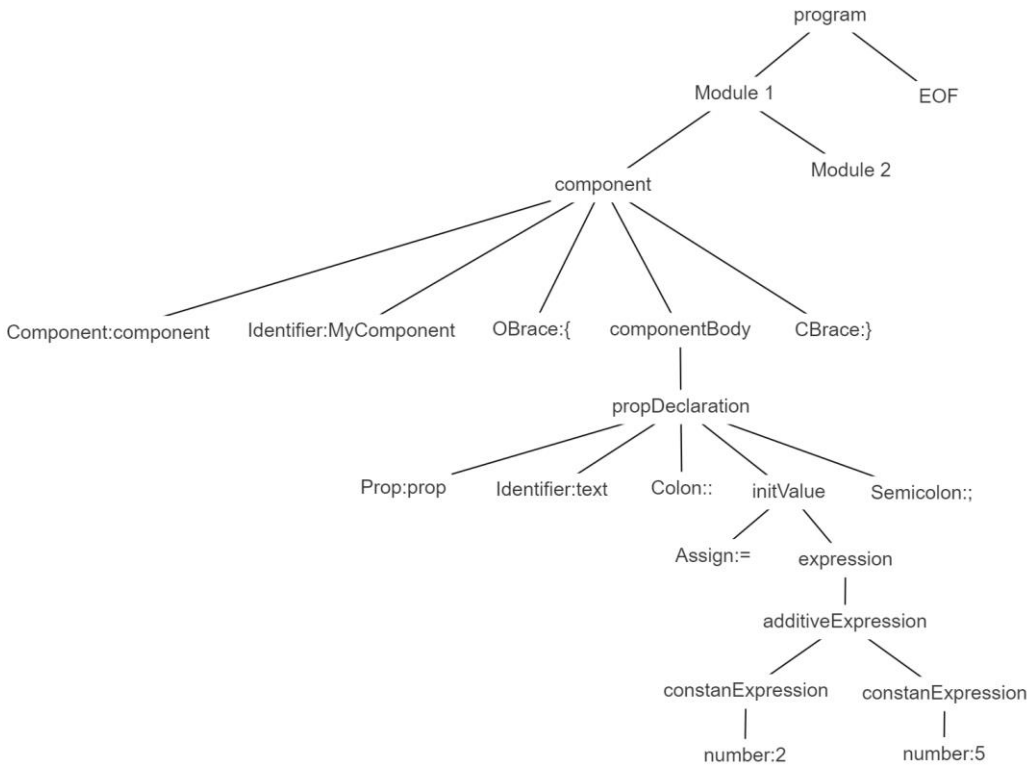

Figura 37 árbol de sintaxis abstracto para código de una expresión aditiva, Fuente: creación propia

Otro tipo de expresión que puede ser simplificada son las expresiones de tipo lógica. En el caso de los operadores lógicos AND y OR adicional a las técnicas de simplificación anteriores podemos realizar simplificaciones adicionales en base a las tablas de verdad. Es decir, por ejemplo, si una expresión AND posee un miembro que se sabe es falso, automáticamente podemos asumir que toda la expresión es falsa dada la naturaleza del operador. De la misma manera el operador OR también puede simplificarse comparando los valores de los operandos contra la tabla de verdad.

5.3.2 Visitando árbol de sintaxis

Existen múltiples formas de ejecutar la simplificación del árbol de sintaxis sin embargo queremos evitar tener que modificar directamente la estructura del árbol sintáctico ya que un error en la implementación podría llevar a errores en el programa muy difíciles de depurar y además se está comprometiendo la integridad del programa escrito originalmente. Para evitar este tipo de situación se implementará la simplificación del árbol de sintaxis utilizando el patrón de visitante. Este patrón consta básicamente de

recorrer el árbol de sintaxis original en pre-order y de manera paralela ir ensamblando un árbol simplificado en base a este. Este árbol adicional que ensamblamos en base al árbol de sintaxis original se le denomina árbol de sintaxis abstracto.

El patrón de visitante se implementa definiendo una interfaz de visitante, cada método de la interfaz está destinado a visitar o procesar una parte en concreto del árbol de sintaxis original. El tipo de regla que visitan es su contexto y es con esta información que el visitante debe tomar las decisiones de cómo simplificar o abstraer la información relevante para poder ensamblar el AST. Es un caso común que muchas reglas dependen de otras reglas por que el patrón de visitante genera un método por cada tipo de nodo en el árbol de sintaxis, por eso sin un nodo tiene como hijos otros nodos que no sabe simplificar, delega esa tarea a otro método de la interfaz que sí sepan procesar ese tipo de contexto en concreto. Todos los nodos de un contexto deben ser visitados y procesados, cuando este proceso termina entonces la función visitante ya puede retornar la fracción del AST que logró procesar. Cada una de estas fracciones del AST se van ensamblando de manera recursiva hasta llegar a la raíz del árbol y terminar.

Las ventajas de hacerlo de esta manera aparte de no modificar el árbol de sintaxis original son la separación de responsabilidades ya que cada método de visita solo se especializa en un tipo de nodo a la vez y delegan todos aquellos nodos que no sepan procesar. Otra ventaja es que el orden de visita puede ser decidido por el visitante, aunque lo más común es que se visite en el mismo orden que el que se usó para ensamblar el árbol de sintaxis en un primer lugar. Adicionalmente como las funciones están segregadas esto significa que se puede reutilizar cada método de visita, aunque el contexto del nodo cambie, dado que el tipo de retorno siempre debe ser el mismo.

Una vez se haya visitado todo el árbol las simplificaciones y transformaciones necesarias estarán hechas y el AST recién creado puede pasar a la siguiente etapa.

5.4 Implementando análisis semántico

La siguiente etapa importante en el procesamiento del programa que ya simplificamos es analizar su semántica, es decir que las cosas que el usuario ha descrito en sentencias realmente puedan ser ejecutadas por el lenguaje de destino o que sean algo que el compilador soporte traducir entre el lenguaje de entrada y el de destino.

5.4.1 Semántica de tipos

Dado que haibt es un lenguaje tipado lo primero que debe de realizar el compilador es registrar todos los tipos declarados por el usuario en el programa, ya que la declaración actúa como plantilla la cual usaremos para validar que los valores asignados a las variables corresponden con los tipos de datos declarados. Un proceso crucial durante el análisis estructural de los tipos es resolver las dependencias, es decir que tipos conforman a otros. Para lograr esto lo primero es registrar todos los nombres de los distintos tipos junto al nombre de cada propiedad, para cada propiedad también es necesario guardar información sobre qué tipo de dato se supone almacenará, en el caso de las propiedades que sean tipos de datos primitivas como los strings o los números esa información ya la conoce el compilador por lo que no es necesario realizar ninguna búsqueda. Las propiedades que no sean tipos de datos primitivos simplemente las marcaremos como una referencia hacia otro tipo.

Una vez realizado ese proceso lo siguiente será ensamblar una gráfica de dependencias entre tipos, esto nos permitirá saber si hay alguna definición de tipo que no pueda ser creada debido a que se define a sí misma de manera directa o circular. Si la referencia al nombre del tipo B se encuentra entonces esta se marca como dependencia del tipo A, naturalmente seguimos resolviendo las referencias hasta haber encontrado todos los tipos que A refiera directamente. Si un tipo no se encuentra en la lista entonces esto significa que hay un error en el programa porque ese tipo no está definido y la compilación debe ser abortada. Por otro lado, si logramos procesar todos los nodos en la lista y todas las dependencias de cada nodo entonces deberíamos de contar con un grafo de dependencias al final del proceso.

Al obtener el grafo de dependencias ya es posible resolver la estructura final de los tipos. Como estaremos validando la estructura de los tipos es muy infectivo tener que realizar todas las búsquedas de tipos en una lista utilizaremos el grado de dependencias como una estructura de momoizacion y poder resolver estas comparaciones más eficientemente. Lo único que debemos revisar en el grafo es que no existan dependencias circulares para eso podemos utilizar un algoritmo de ordenamiento topológico en grafos direccionados. Esto lo implementaremos utilizando DFS y haciendo seguimiento de los nodos visitados, si un nodo ya fue visitado por otra parte del DFS entonces existe un bucle y el grafo no es acíclico, en cuyo caso la única alternativa es abortar la compilación.

5.4.2 Semántica de expresiones

Para este punto ya tenemos registrados todos los tipos existentes en el programa y además tenemos una forma efectiva de validar si un objeto puede ser asignado a una variable de ese tipo. Para poder determinar si el uso de una expresión es correcto lo primero que se necesita es una forma de saber el tipo de retorno de una expresión, con esto lo que se intenta saber es si por ejemplo al sumar un número con otro la naturaleza del resultado es un número o es otro tipo de dato. Para poder determinar los tipos de expresiones hay que definir primero los tipos de las expresiones primarias. Para el caso de Haibt las expresiones literales numéricas siempre tendrán asociado el tipo número, las expresiones de texto siempre serán de tipo string, otros como los tokens para true, false y null también poseen tipos, booleano en caso de true y false y null es un tipo en sí mismo.

El siguiente caso es resolver los tipos de los identificadores, siempre designan nombre de elementos del lenguaje una forma de poder resolver el tipo de dato que representan es buscarlo en la tabla de símbolos, ya que es aquí donde hemos almacenado esa información.

A partir de los dos casos anteriores ya podemos resolver los tipos de expresiones más complejas.
Por ejemplo, todas las expresiones aritméticas necesitan dos números para poder operar, así que saber si son semánticamente correctas es relativamente sencillo, primero resuelve su operando izquierdo y su el resultado es de tipo numérico entonces está bien, por lo que se procede a la resolución del operando derecho, si este también resulta ser un tipo de dato numérico entonces la operación está definida y puede ejecutarse. La única excepción a esto es la suma ya que el operador '+' se encuentra sobrecargado lo que significa que tiene múltiples definiciones de datos con los que puede operar. Esto se debe a que la suma está definida para números, lo que es una conclusión esperada pero también para textos, el significado de sumar dos textos es la concatenación o unión de los textos izquierdo y derecho en ese mismo orden, dado que el primer operador de la suma puede ser un número o un string y el segundo también, esta posee 4 variantes que son válidas, entre números, entre textos y un texto y un número o viceversa (Interpreters, 14/10/2023).

La siguiente categoría de expresiones que debemos poder resolver son las expresiones lógicas y relacionales, en el caso de este tipo de expresiones los resultados siempre serán falsos o verdaderos dependiendo de la evaluación de cada una de las proposiciones, lo en este caso es recorrer la expresión asegurándose que todos sus elementos también resuelvan a valores booleanos u otras expresiones lógicas ya que de no ser así la expresión no se puede determinar.

En el caso de la expresión ternaria al igual que el caso de las lógicas se debe validar que su primer argumento sea un valor o expresión lógica y luego que tanto su rama izquierda como derecha retornen un mismo tipo de valor, ya que de no ser así no se puede determinar el tipo de retorno.

Finalmente, en el caso de las expresiones de asignación la única validación que existe es asegurar que el tipo del miembro derecho sea el mismo que el tipo declarado de la variable a la que se asigna, si esta condición no se cumple entonces la expresión no es asignable y se ha encontrado un error en el programa.

5.5 Implementando generación de código

Debido a que el AST cumple con todos los criterios del análisis semántico para ser considerado un programa válido ya es posible utilizarlo como la base para poder traducir las instrucciones y sentencias del lenguaje de origen (Haibt) al lenguaje de destino (TypeScript). Gran parte de la generación de código es TypeScript, por ejemplo, UI en React se puede representar al completo con TypeScript ya que este también soporta JSX de manera nativa pero lamentablemente para el caso de Vue esto no es posible debido a que Vue tiene su propio lenguaje de script que, aunque basado ampliamente en la sintaxis de JavaScript posee estructuras incompatibles con TypeScript. Además, otros frameworks como Angular o Svete tienen sus propios compiladores sintaxis y patrones únicos que no se pueden replicar con TypeScript de ninguna manera.

Por esta razón el compilador de Haibt no puede tener una línea de producción estática ya que existen muchos frameworks y cada uno evoluciona diferente agregando características o cambiando la forma en que sus procesos internos funcionan. Por estas razones no resulta práctico tener un único generador de código. En su lugar el acercamiento que toma el compilador de Haibt es utilizar una estructura de plugins, esto significa que el código encargado de transformar insumos como el AST, tablas de símbolos e información de tipos en código específico para cada framework no se compila junto al código del resto del compilador, sino que es definido en algún proyecto a parte y luego se carga en la línea de ensamblado de manera dinámica cargando como un plugin.

La estructura de plugins posee varias ventajas para un proyecto como el compilador de Haibt, el primero es escalabilidad, ya que el código del plugin no forma parte del compilador, aunque el número de frameworks soportados incremente, el código dedicado a cada framework no hará que el proyecto incrementa su cantidad de código, lo que lo vuelve más fácil de organizar, mantener y depurar. La otra ventaja es que estos plugins se pueden cargar bajo demanda por lo que, si un usuario solamente necesita generar código de un framework,

el compilador únicamente generará lo que se le ha solicitado, sin necesidad de cargar los demás módulos que no sean necesarios.

Otra de las ventajas de separar los códigos es la habilidad de poder enfocarse en únicamente la funcionalidad común lo que naturalmente resulta en API más pulidas y con menos fallos. Esto se debe a que no se intenta adecuar el compilador a múltiples plugins sino al contrario, son los plugins lo que deben de utilizar las API comunes de manera efectiva. Dado que los plugins existen como proyectos por separado esto significa que sus desarrolladores pueden enfocarse únicamente en resolver los problemas que les competen al intentar traducir el código Haibt a un framework en particular. Esto también significa que cada plugin puede tener su ciclo de actualización parches que se adecúen a las necesidades del framework objetivos, lo que se traduce en menores tiempos de respuesta entre que un framework libere una nueva característica o versión y que esta sea implementada en el plugin. Esto también reduce los tiempos en que se solventan los errores del código.

Como el compilador de Haibt no es responsable de absolutamente todo el proceso, es posible abstraer ciertas funcionalidades de la generación del código que luego los plugins podrán aprovechar. Por ejemplo, imprimir el código generado a un archivo será una tarea que prácticamente todos los plugins implementan, para evitar esta duplicación de código, este tipo de funcionalidades comunes existirán dentro del compilador. La generación de código para tipos de TypeScript es también una de las funciones en común, la mayoría de frameworks poseen soporte directo para TypeScript por lo que implementar la generación de código común en TypeScript también es una tarea compartida.

El uso de plantillas JSX/HTML es algo bastante común dentro de los frameworks para UI, en cuanto a la sintaxis de las plantillas estas son muy similares y son solo algunas cosas como el binding o directivas las que realmente varían en sintaxis entre un framework y otro. Por esta razón la traducción de plantillas se implementa como una función compartida, pero sobre la cual se pueden sobrescribir la manera en que cada una de sus características se ejecuta, así si Vue utiliza una forma distinta a la de React, cada plugin deberá especificar a través de una función la manera adecuada de escribir elementos con binding o directivas.

CSS es otra parte que todos los framework tienen en común, ya que todos deben de poder interactuar con CSS para poder tener interfaces estéticas, implementaremos el compilador de Haibt de tal manera que pueda generar código de CSS por sí mismo. Si un plugin necesita acceso al código CSS estos podrán obtener el código fuente y enviarlo a un archivo externo para poder importarse en una etapa posterior, en otros casos los estilos son parte de la propia definición del componente. Sea cual sea la implementación en el plugin

siempre deben poder generar CSS en forma de código y esta funcionalidad la exportemos de la interfaz del Compilador de Haibt de tal forma que se pueda importar en los plugins y generar el código CSS en el lugar adecuado.

A partir de este punto es donde el proceso de generación de código se bifurca y para la primera versión del compilado de Haibt se especifica el proceso de generación de código en dos frameworks distintos, en React y en Vue.

5.6 Generación de código React

React js es una librería de JavaScript desarrollada por Facebook liberada en el año 2013, en su momento introdujo conceptos innovadores como el de los componentes como se convencen actualmente, binding de variables en las plantillas de forma unidireccional y JSX el cuál era una sintaxis declarativa para describir qué interfaz debe mostrarse, pero sin especificar todas y cada una de las acciones necesarias para que ocurriera. Estas y otras características dotaron a React de una gran popularidad e influencia en el desarrollo moderno de aplicaciones web. (Rippon, 2023)

En sus inicios las aplicaciones que utilizaban componente de React se escribían utilizando una clase que hereda de 'Component' de React. En esa clase base se encontraban funcionalidad como poder mantener un estado local que ayudará al componente a renderizar o cambiar de comportamiento según se necesite, las propiedades que el componente necesitaba se pasaban como parámetros del constructor y se puede sobrescribir métodos del ciclo de vida que se ejecutaban en momentos específicos como 'componentDidMount' cuando el componente se montaba en el DOM, es decir que pasaba a ser parte del HTML de la página que veía el usuario y otros como cuando el componente era dispuesto y no se necesitaba más. Esta forma de crear componentes se llamó 'class components'. Esta forma de definir y utilizar componentes en React rápidamente mostró signos de agotamiento dado que era complicada de usar y además tenía mucha veracidad, es decir código que realmente no hace nada, simplemente la sintaxis lo requiere escrito de cierta manera.

Esta falta de practicidad se evidenció más cuando se comparaban los class components con sintaxis alternativas para componentes en React, los function components. Los function components son la misma idea de los class components excepto que como el nombre indica estos utilizaban una función para declarar el cuerpo del componente. Los function components tenían la ventaja de ser más rápidos de escribir y que tienen las mismas características que los class componentes, esto agregado que el uso de hooks que era más

sencillo e intuitivo comparado con los métodos de ciclo de vida causó que estos se volvieran la sintaxis predilecta para la creación de componentes y aplicaciones en React. Por esta razón, aunque exista una forma de traducir un componente de Haibt a una clase de TypeScript en su lugar utilizaremos componentes funcionales.

5.6.1 Generación de componente en React

Dado que en React se implementarán los componentes como una función de propiedades y estado primero se debe poder generar la forma en que se pasarán las propiedades al componente. En los componentes funcionales las propiedades se transfieren como argumentos de la función que renderiza el componente, por lo que para lograr tipar las propiedades basta con declarar un tipo en TypeScript que sea del tipo objeto y será el primer argumento de la función.

Una vez que tengamos el tipo que defina a las propiedades creado, ya es posible declarar la firma de la función, para esto indicamos la palabra clave **function** de JavaScript seguida del nombre de la función, es decir del componente y entre los argumentos de la función un argumento que puede tener cualquier nombre, pero la convención sugiere que se llame prop. Se cierra la lista de argumentos de la función seguida del tipo de retorno, normalmente se anota como 'ReactElement' o 'JSX.Element'. Para nuestros propósitos ambos tipos indican prácticamente lo mismo. La declaración terminada se ve parecida a la figura 38.

```
Function MyComponent (props: MyComponentProps): JSX.Element {}
```

Figura 38. Ejemplo de componente funcional React, Fuente: Creación propia

Ahora que ya tenemos forma de traducir la declaración de un componente en Haibt a uno de React el siguiente paso es poder generar el código del resto del componente, particularmente dos cosas. La plantilla y el scripting necesario para renderizar la interfaz incluyendo sus complejidades como directivas y vinculación o integración. En el caso de React lo primero que se implementará será la desestructuración del objeto props en sus miembros, con esto ya podremos acceder a los valores de las props de manera sencilla sin tener que pasar siempre por el objeto props. Esto ayudará sobre todo en los casos donde tengamos propiedades interpoladas o necesitemos acceder a su valor para poder operar sobre este en alguna expresión.

En lo que a la plantilla respecta en React existe una convención y es que las etiquetas nativas de HTML como el párrafo, la división o el span se escriben con letras minúsculas, mientras que los componentes de React deben escribirse con letras mayúsculas al inicio. Utilizaremos esa convención en las plantillas

generadas para mantener coherencia con un código de React estándar, que es lo que probablemente un desarrollador hubiese hecho si escribiera estos componentes a mano.

Con respecto a las directivas, en el caso de las estructurales como if o else resulta que React no tiene ningún tipo de soporte, en su lugar si un elemento debe renderizar de manera condicional este debe de colocarse como una expresión de tipo AND o un operador ternario.

```
Function MyComponent(props: MyComponentProps): JSX.Element {
   const isVisible = true;
   return isVisible && <p>Hello</p>;
}
```

Figura 39. Ejemplo de función My Component, Fuente: Creación propia

En el ejemplo de Figura 39 se puede ver como la función retorna la etiqueta párrafo con el texto 'Hello' solo si el valor de la variable visible resulta ser cierto, en caso contrario la expresión resultante tiene el valor de false y la representación de false según react es no mostrar nada.

Otro tipo de directivas como el for tampoco se soportan en react, en su lugar se deben crear los elementos de manera dinámica y asignar una key o identificador a cada elemento que se cree de manera dinámica. Por lo general esto se realiza dentro de una función map y el arreglo resultante es lo que se pasa a React para que lo muestre en el DOM.

Para el caso de binding en las propiedades React utiliza la notación estándar de JSX al igual que Haibt por lo que las propiedades en concreto no necesitan ser traducidas o transformadas, estas se pueden reescribir prácticamente de la misma manera que se encontraban escritas en el programa original.

Las expresiones dentro de las plantillas en el caso de React utilizan la notación de JSX es decir se escriben entre una '{'y '}', dentro de las llaves se puede escribir cualquier tipo de expresión y en el caso particular de react el tipo de esta expresión realmente no importa, React tiene definido como mostrar cualquier tipo en el DOM ya sean datos primitivos o componentes. Como ejemplo se tiene el siguiente componente en Haibt.

```
component Text {
  prop text: string;

  render (
    <div id="someId" value= {5} styles={someStyle}>
      <span>{text}</span>
    </div>
  );
}
```

Figura 40. Ejemplo de componente en Haibt, Fuente: Creación propia

La definición del componente de Figura 40 es un componente llamado ‘text’ con una propiedad llamada text de tipo string que se usa para mostrar cómo está el texto en la página web. Adicionalmente el componente hace binding de tres atributos, id de tipo string, value de tipo número y styles el cual hace referencia a un identificador. El resultado en React TypeScript es lo que muestra la figura 41.

```
Export function Text (props: TextProps) {
   const { text } = props;
   return (
      <div id=”someId” value={5} styles={someStyle}>
        <span>{text}</span>
      </div>
   );
}
```

Figura 41. Resultado de TypeScript en react, Fuente: Creación propia

5.7 Generación de código Vue

El caso de Vue es distinto, habiendo sido diseñado después de React y Angular este pudo tomar inspiración de las cosas que hacen bien pero también tuvo la oportunidad de cambiar ciertas características o funcionalidades que no habían sido bien recibidas o usadas de manera popular. Una de las principales diferencias entre React y Vue es que Vue es un framework, no una librería (Camden, Di Francesco, Gurney, Kirkbride, & Shavin, 2020).

En sus primeras versiones Vue definió componentes en un archivo con extensión '. Vue' este era un archivo con una sintaxis personalizada para Vue. En él se podía escribir los estilos CSS, la plantilla HTML y el código TypeScript o JavaScript que le daría la interactividad al componente. Todo eso se encontraba en un solo archivo, posteriormente en versiones más avanzadas como la incorporación de JSX a Vue o la API de composición la cual permite definir componentes de manera programática. No obstante, la manera más popular de definir componente de Vue sigue siendo escribirlo todo en un único archivo empaquetado con la extensión '. Vue', este formato se conoce popularmente como single file components o SFC. Por esa razón el plugin de Vue para Haibt generará código de Vue en formato SFC (Vue 23/09/2023).

Dentro de un SFC se encuentran tres bloques, pueden estar dispuestos en cualquier orden, pero por lo general se encuentran en el bloque de script, template y finalmente el de estilos. Cada bloque se puede definir una única vez y se encuentra encerrado en etiquetas parecidas a HTML con el nombre de la sección. Sin embargo, la sintaxis que se utiliza dentro de cada bloque es completamente distinta. En el bloque de script se puede utilizar ya sea TypeScript o JavaScript, dentro de este bloque se definen las propiedades del componente, los valores por defecto, las variables que puede utilizar para interpolar en la plantilla, eventos con el nombre de emisores entre otros. Normalmente esta sección es donde se realiza toda la inicialización del componente y se puede definir acciones adicionales llamadas callbacks que normalmente se ejecutan como respuesta a un evento generado por acciones del usuario como dar clic en un elemento o presionar el teclado.

El bloque de la plantilla es donde se definen los elementos HTML o componentes que se deben agregar a la página. Se define su estructura y jerarquía, además de los valores que cada atributo posee, estos pueden ser vinculados utilizando valores constantes o la referencia al valor de alguna variable. A cada elemento en la plantilla también se le pueden añadir directivas. En el caso de Vue las directivas son un constructo nativo por lo que no es necesario plantear ningún tipo de mecanismos como sucedía en React, basta con asociar las directivas if de Haibt con su equivalente en Vue, este también es el caso para las demás directivas como else o for. La expresión que va como argumento de cada directiva también debe ser ajustada ya que Vue cuando se utiliza la sintaxis de SFC no se pueden encerrar entre llaves, sino que deben ir entre comillas dobles para elementos de tipo string o prefijadas con ':' dos puntos para elementos de cualquier otro tipo como números o referencias con identificadores. Finalmente, dentro de cada elemento de las plantillas también se puede hacer una interpolación, esto significa que el valor que retorna la expresión entre llaves dobles, es decir la expresión interpolada, se muestra en la página web. Esto es útil ya que Vue actualizará la interfaz HTML de manera automática cada vez que el valor de esta cambie.

Por último, está el bloque de estilos, en este bloque se pueden definir los estilos CSS o con sintaxis de SCSS que acompañarán al componente. Una de las ventajas que tiene Vue al poseer un bloque dedicado de CSS es que este se puede marcar como **scoped** lo que significa que únicamente afectan al componente donde los estilos están definidos, esto ayuda a evitar una situación conocida como style leaking lo que ocurre cuando estilos de un componente o pares de este sobrescriben de forma no esperada los estilos de otro. Como la generación de código CSS es una tarea común el compilador de Haibt ya tiene forma de generar código CSS, por lo tanto, lo único que este plugin debe de hacer es indicar que dicho código CSS sea generado dentro de este bloque.

Con estrategias para poder generar código en cada uno de estos tres bloques ya es posible generar código funcional dentro de la sintaxis SFC de Vue, sin embargo, por la naturaleza de los SFC es imposible definir más de un componente por archivo, por esa razón es necesario idear una forma en la que a partir de un archivo de Haibt con múltiples componentes se puedan generar un SFC para cada uno de ellos.

Lo primero y más sencillo es mover el código de tipos en un archivo aparte el cual todos diferenciaron de manera individual, esto simplifica la generación de código debido a que no tenemos que resolver la interdependencia de módulos, sino que en su lugar podemos simplemente resolver y generar código de manera individual. En figura 42 un ejemplo de componente Haibt.

```
component Text {
  prop text: string;

  render (
    <div id="someId" value= {5} styles={someStyle}>
      <span>{text}</span>
    </div>
  );
}
```

Figura 42. Componente de Haibt, Fuente: Creación propia

Posee su equivalente en vue como indica la Figura 43.

```
<script setup lang="ts">
  import { TextProps } from './types';
  defineProps<TextProps>();
</script>

<template>
  <div id="someId" :value="5">
    <span>
      {{ text }}
    </span>
  </div>
</template>
```

Figura 43. Componente de Vue, Fuente: Creación propia

5.8 Generación de código CSS

La forma en que Haibt describe sus estilos es con una sintaxis parecida a la de SCSS, pero omitiendo varias de sus características avanzadas como importación de módulos, creación de variables entre otras. La sintaxis de estilos en Haibt permite hacer la mayoría de las cosas que se pueden hacer en CSS normal, como por ejemplo declarar el nombre de una clase anteponiendo el carácter '.' Seguido de un nombre, y luego definiendo una lista de reglas de estilos que apliquen a esa clase. También es posible al igual que en CSS definir una lista de especificadores para una secuencia de reglas, en Haibt se puede hacer de dos formas, escribiendo toda la lista de especificadores al igual que se haría en CSS o anidar la lista de especificadores lo cual tiene el mismo efecto.

Adicionalmente es posible definir modificaciones en la lista de especificadores como la unión de dos clases para indicar que la lista de reglas solo aplica a elementos con ambas clases o utilizar el operador '>' que indica que la lista de reglas solo aplica a los elementos que sean descendientes directos de una clase.

Para poder escribir el código CSS a partir del AST de Haibt lo que el compilador hace es llevar una lista de las clases que se van procesando y el orden en que estas son usadas, estas se introducen en un stack que serán un prefijo para todas dentro de la clase que se encuentra actualmente en el stack.

Con esta lista de prefijos en el stack ya podemos escribir la lista de identificadores adecuada para la secuencia de especificadores en CSS. Dentro de cada bloque después de una secuencia de especificadores se escriben las distintas reglas de CSS que existen.

Debido a que las reglas de CSS en Haibt no pueden poseer identificadores con guiones entre los nombres es necesario tomar las reglas que se escribieron en camelCase y reescribirse usando los nombres adecuados para que el CSS funcione de la manera esperada. Tomando ambas cosas en consideración ya es posible empezar con la generación de código.

El algoritmo es realmente simple, consiste en tomar la lista de modificadores en el stack, escribirlos en el archivo y dentro de su bloque escribí todas las reglas CSS que posea, si posee subclases es necesario cerrar el bloque anterior y agregar los nombres de la subclase al stack. Acto seguido es escribir las reglas de las subclases y al terminar retornar, eliminando el nombre de la subclase del stack, posteriormente realizar el mismo procedimiento hasta que se termine la lista de clases en el módulo y que el stack termine vacío.

Es importante también que antes de retornar de la generación del código para una subclase de CSS se hayan escrito todas las reglas, de no ser así es posible que se dupliquen los especificadores de una clase. Para evitar esta situación de ocurrir lo que se hace durante la etapa de creación del AST es usar una estrategia llamada hoisting.

La estrategia de hoisting consiste en reordenar el código en una forma distinta a la forma en la que se ha escrito. Así, por ejemplo, si existe una regla en la primera línea luego una subclase y después de estas mas reglas de CSS, lo que se hace es tomar todas las reglas en el mismo nivel, agruparlas y procesarlas todas primeros, antes que cualquier subclase.

Reescribir el orden de las reglas de CSS asegura no que se duplicaran la secuencia de especificadores y que así se generad un código CSS más adecuado. Otro efecto es que escribirlo de esta manera también provoca que las listas con mayor numero de identificadores naturalmente vayan siendo procesados hacia el final, lo que significa que en el archivo los estilos más generales estarán al inicio del código y los más específicos hacia el fondo.

CAPÍTULO 6. PRUEBAS Y RESULTADOS

6.1 Introducción a las pruebas

Durante la construcción del compilador para Haibt se ejecutaban pruebas sencillas que solo pretendían probar partes aisladas del proceso de compilación. Esto debía ser así ya que durante la construcción se necesitaba tener seguridad que las características que se iban implementando e integrando con el resto del código funcionaran como se esperaba, lo que es más si fallaban era más fácil apuntar al posible lugar en el código que causara el problema. Este acercamiento a problemas complejos es bastante común en el área de la computación ya que de otra manera se tendría que haber depurado la completitud del programa, el cual según un reporte del programa CLOC (Count Lines Of Code) ya asciende a más 10 mil de líneas de código.

Sin embargo, al hacer uso del compilador para el propósito que fue creado, estas pruebas de casos ideales no son representativas del código que un usuario promedio podría llegar a generar. Para asegurarnos que el compilador pueda lidiar con una gama más amplia de programas posibles se preparara una suite de pruebas que ayudaran a comprobar puntos importantes del proceso como el rendimiento, la compatibilidad de los códigos generados, así como la precisión de la conversión entre el código fuente y el código generado.

Estas pruebas además intentaran encontrar áreas de mejora en el compilador, ya que no todas las características se han implementado al completo algunos programas que las utilicen podrían resultar en código invalido o código que no sea preciso con respecto al código fuente. También en el caso de existir bugs es posible que las pruebas puedan exponer las causas de la ocurrencia de estos.

6.1.1 Entorno de pruebas

Es importante recalcar que al hacer pruebas de rendimiento sobre el software el Hardware en el que se ejecutan juega un papel muy crucial, por esta razón los resultados de rendimiento de las pruebas son únicamente válido para el hardware y software que se describe a continuación.

Hardware:

- CPU: Intel Core i7-9700K con reloj a 5.0GHz
- RAM: 32 Gigas DDR-4 a 3200MHz timings 16-18-18-36 configuración 4X8
- SSD: Samsung 980 NVME PCIe 3.0

Software:

- OS: Windows 10 Pro 22H2 19045.3930
- Entorno de ejecución de JavaScript: NodeJS LTS 20.11.0
- Compilador de TypeScript: TSC 5.2.2
- Bundler: Rollup 4.3.0

6.2 Metodología de las pruebas

Las pruebas se separan en dos categorías, las pruebas de funcionalidad cuyo objetivo es validar primero que el código tenga un comportamiento equivalente una vez se ejecute en el framework al que haya sido compilado, pero también se revisara la exactitud del código generado para asegurarse de que la semántica del programa también sea equivalente. El otro tipo de pruebas son las de rendimiento, el objetivo de estas será mostrar el uso de recursos que el compilado hace de memoria y tiempo de ejecución con archivos progresivamente más extensos hasta encontrar algún limite donde se rompa o uno donde ya no sea practico debido al tiempo o memoria que necesite.

Para las pruebas de funcionalidad los paso a ejecutar son:

1. Crear un programa en Haibt que incluya la funcionalidad que se desea probar
2. Ejecutar el compilador con logs en modo de solo errores, con cada uno de los frameworks
3. Revisar la salida del programa de manera manual y comparar contra el código fuente para validar que el resultado es el esperado.
4. Finalmente se tomar el código generado e incorporarlo en una aplicación que utilice el framework de salida y validar que al ejecutarse efectivamente hace lo que se espera
5. Se anexará una figura con los códigos generado a manera de evidencia.

Para las pruebas de rendimiento necesitaremos programas auxiliares que ayudaran en ciertas tareas, el primero será un script que genere un archivo de Haibt en el sistema de archivo. Este programa tendrá como entrada 5 parámetros, estos son:

1. La cantidad de componentes
2. La cantidad de propiedades de cada componente
3. La cantidad de nodos en la plantilla indicado con dos parámetros
4. El número de niveles que cada plantilla genera

5. La cantidad de atributos para cada nodo en la plantilla

El otro programa auxilia es uno que ejecute la compilación sobre el mismo archivo múltiples veces y genere un registro de las métricas en un archivo. Es importante ejecutar cada archivo múltiples veces para obtener promedios estadísticos aparte de poder definir mínimos y máximos para poder tener una visión clara de la variabilidad que pueda llegar a darse.

El registro constara de los siguientes elementos en un archivo JSON.

1. Nombre del archivo ejecutado
2. Numero de la ejecución
3. Para cada ejecución tiempo de inicio y tiempo de finalización registrado en nanosegundos
4. Una lista de la cantidad de memoria utilizada al final de 9 etapas distintas del proceso
    1. Inicio del programa
    2. Parseo
    3. Simplificación
    4. Registro de símbolos
    5. Reemplazo de clases CSS
    6. Type checking
    7. Análisis de directivas en la plantilla
    8. Cargar el plugin
    9. Compilar y escribir archivos en disco

Al final de las pruebas para cada archivo se hará un análisis para determinar el promedio de memoria y tiempo de ejecución. Con esta información se podrán generar gráficos para mostrar la información de manera visual y poder elaborar conclusiones. Estos gráficos se incluyen como figuras y los archivos JSON estarán disponibles en GitHub, ver anexo B.

6.3 Pruebas de funcionalidad

Las pruebas de funcionalidad se ejecutaran en un orden de complejidad incremental, es decir se empezaran las pruebas con las estructuras fundamentales y progresivamente se irán agregando nodos al programa para incrementar la complejidad de código a analizar y validar no solo que el compilador puede manejar los casos por separados si no en combinación entre ellos, esto es importante debido a que un programa regular sin necesidad de ser muy extenso ya puede hacer uso de varias de las características implementadas a la vez.

Primera prueba. Archivo vacío. Con esta prueba se pretende comprobar que el compilador no llega a ningún valor nulo o indefinido que genere un fallo en el mismo. Lo esperado es que simplemente ignore el archivo vacío y no genere ninguna salida. El código fuente es como se indica en la Figura 44.

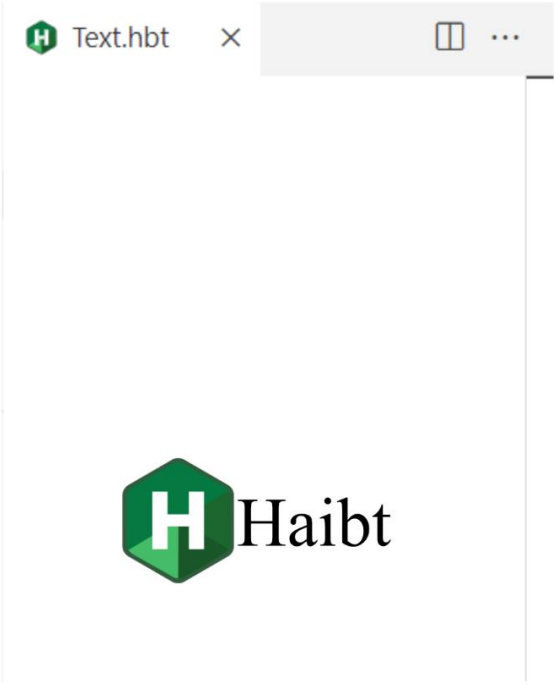

Figura 44 Archivo vacío de Haibt, Fuente: creación propia

Esta prueba no genera ninguna salida en términos de archivo, así que para asegurar que el compilador logra ejecutarse de inicio hasta el fin se ejecutó el siguiente comando que activa la salida de logs en nivel informativo. Así se puede evitar asumir que se ejecutó correctamente o que fallo silenciosamente. El comando para activar los logs es “crossbind -p react -l info .\Text.hbt”, esto produce la salida que se indica en Figura 45

```
PS D:\Documentos\GitHub\output\mylib> crossbind -p react -l info .\Text.hbt
Found 1 files to compile
Files:
.\Text.hbt
React plugin loaded
Vue plugin loaded
Compilation complete
```

Figura 45. Logs informativos de la primera prueba, Fuente: creación propia

Los resultados de esta prueba demuestran que el compilador se comporta como se espera, por tanto, se considera exitosa.

Segunda prueba. Estructuras primigenias. Esta prueba pretende determinar si las estructuras más sencillas de cada tipo dentro del compilador son válidas. La expectativa es que todas las estructuras vacías sean validas, ya que es de estas que todas las demás estructuras posibles en el lenguaje descienden. Si se determina que el compilador puede manejar estos casos, implica que los sistemas internos del compilador funcionan adecuadamente para cada tipo de estructura.

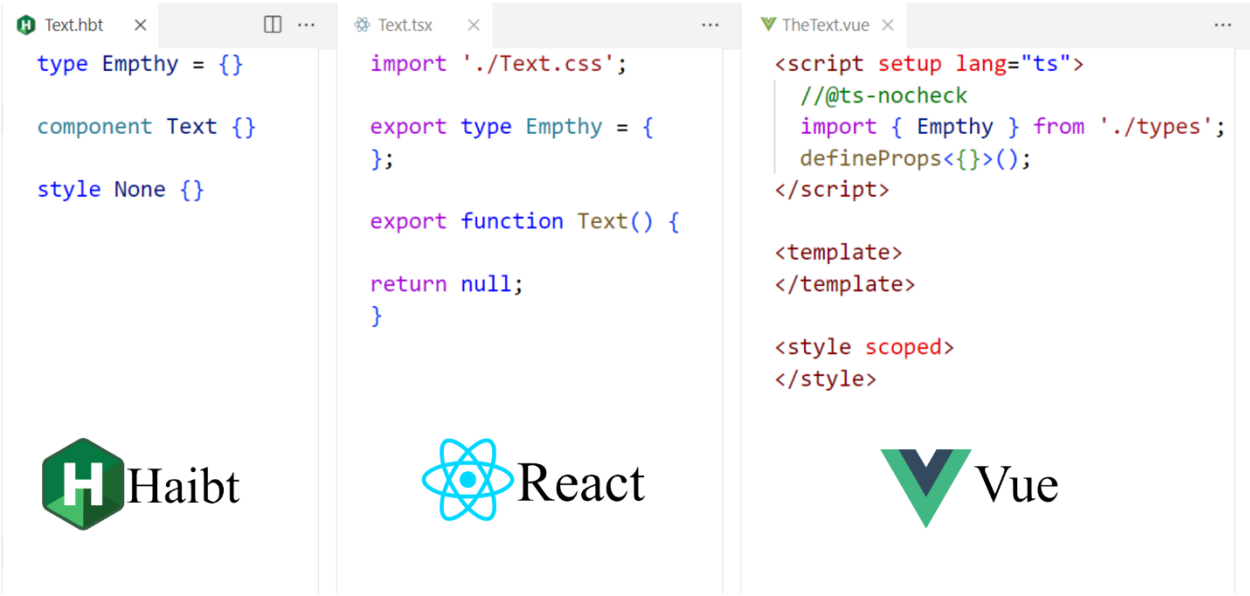

Figura 46. Estructuras primigenias en Haibt, Fuente: creación propia

Tras observar el código generado en la Figura 46 se concluye que el compilador ha generado código de manera correcta y que este mantiene la semántica del código fuente. Este es el resultado esperado por lo que esta prueba se considera exitosa.

Tercera prueba. Componente con plantilla. Una de las características más populares de los frameworks es el poder definir interfaz que parece HTML dentro del componente, por su importancia se dedicaran varias pruebas a validar la utilidad del sistema de plantilla de Haibt. Se creará un componente que haga uso de varias de las etiquetas más utilizadas dentro de los proyectos de desarrollo web para variar el contenido de la plantilla.

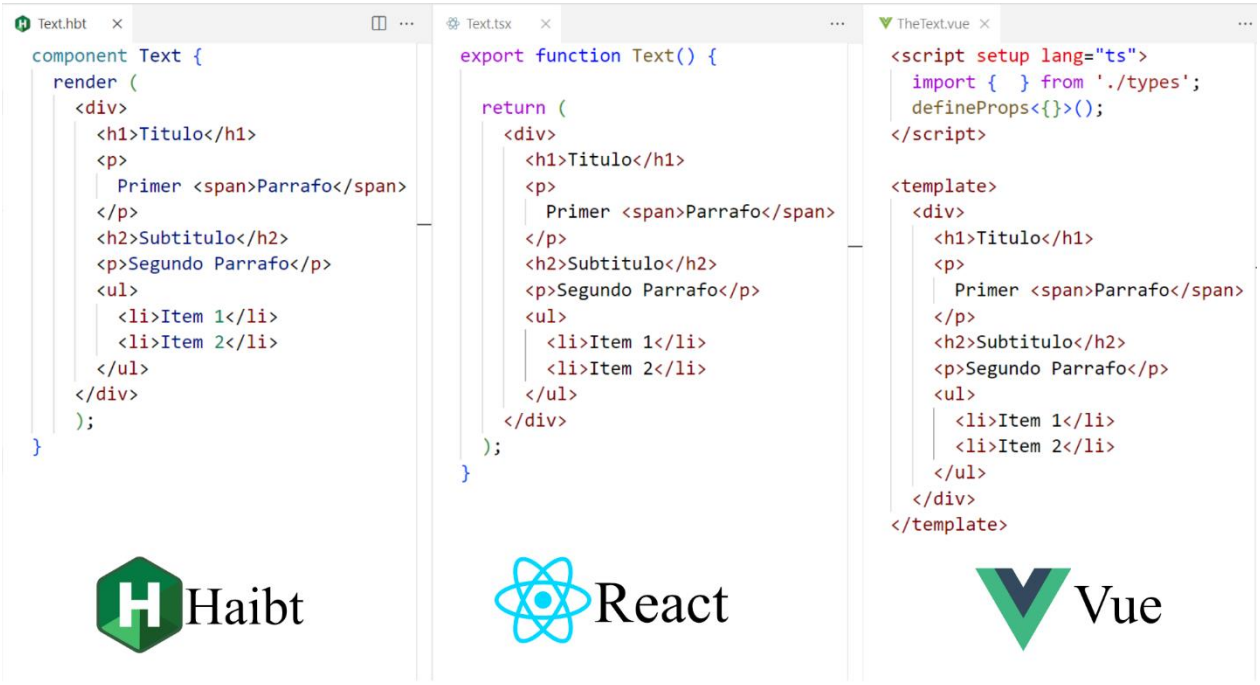

Figura 47 Plantillas HTML de Haibt, Fuente: creación propia

Los resultados mostrados en Figura 47 indican que la plantilla se generó adecuadamente, esto se puede aprecias ya que ni el código de React (centro) o Vue (derecha) tienen el orden alterado en su salida, la estructura entre la entrada y las salidas se preserva. Además del orden en que las etiquetas HTML aparecen, también se puede ver que los textos que acompañan la plantilla también se preservan y que la jerarquía de las etiquetas permanece sin alteraciones. Todo esto indica que por lo menos en cuanto a estructuras se refiere el compilador está logrando generar código de manera adecuada. El único inconveniente detectado es en Vue (derecha) donde se está importando nada de un archivo de tipos vacío, esto no afecta en nada a la ejecución del programa, pero es un posible punto de mejora. El compilador genera plantilla de la manera adecuada por lo que esta prueba se considera exitosa, aunque con margen de mejora.

Cuarta prueba. Múltiples plantillas de nivel raíz. Esta prueba se centra principalmente sobre un inconveniente que React posee. En React no es posible tener dos elementos en la raíz de la plantilla, esto se debe a un comportamiento interno de React, sin embargo, es posible evitar esta limitación colocando de manera manual un fragmento (una etiqueta que no se muestra en la página web), de tal forma que este actúe como un único contenedor para dos o más plantillas en la raíz. En Haibt esto no es necesario, ya que Haibt soporta de manera nativa múltiples plantillas a nivel raíz, lo mismo es cierto para Frameworks como Vue y

Angular. Esta prueba consistirá en colocar dos elementos en la raíz y el resultado deberá ser en React un fragmento insertado automáticamente, mientras que en Vue debería de aparecer una plantilla equivalente.

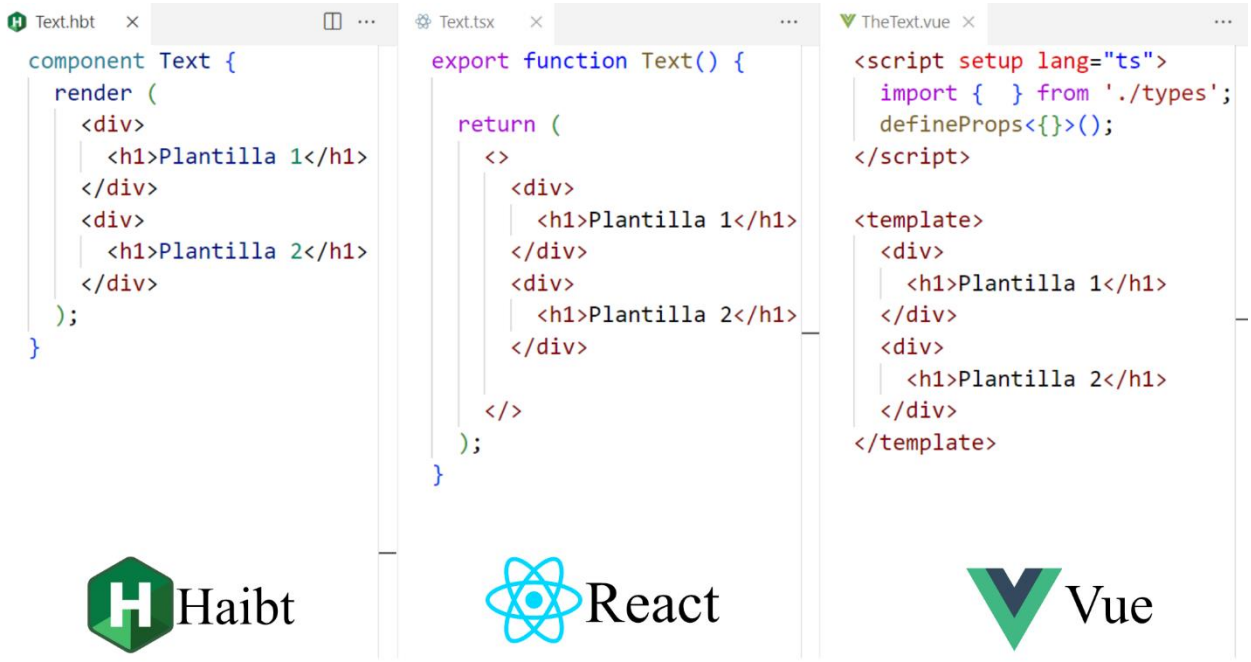

Figura 48 Plantillas HTML con múltiples raíces en Haibt, Fuente: creación propia

Los resultados que muestra la Figura 48 son adecuados a las expectativas. En el caso del código fuente Haibt (izquierda) se puede ver que pudo procesar dos elementos en el nivel raíz de la plantilla, mientras que en React (centro) se ha incrustado un fragmento de manera automática (denotado por <> apertura y </> cierre). Finalmente, en Vue (derecha) Haibt se aprovecha que Vue puede manejar estos elementos sin necesidad de insertar código adicional por lo que la plantilla es básicamente una inyectiva. Tras observar los resultados se concluye que esta prueba se ha superado exitosamente.

Quinta prueba. Atributos. Una de las características más fascinantes de los componentes es que son piezas reutilizables, parte de esta reusabilidad proviene del hecho que pueden adaptarse al contexto donde serán utilizados gracias a la parametrización a través de atributos. Los atributos permiten que un componente pueda recibir data desde otro componente y ajustar su comportamiento o interfaz en consecuencia. Para verificar que estos mecanismos en su forma más simple, pasando valores estáticos funcione correctamente se ha creado una plantilla con 3 elementos los cuales reciben valores en sus atributos, la expectativa es que la plantilla sea generada en React y Vue manteniendo los mismos valores y semántica que el original.

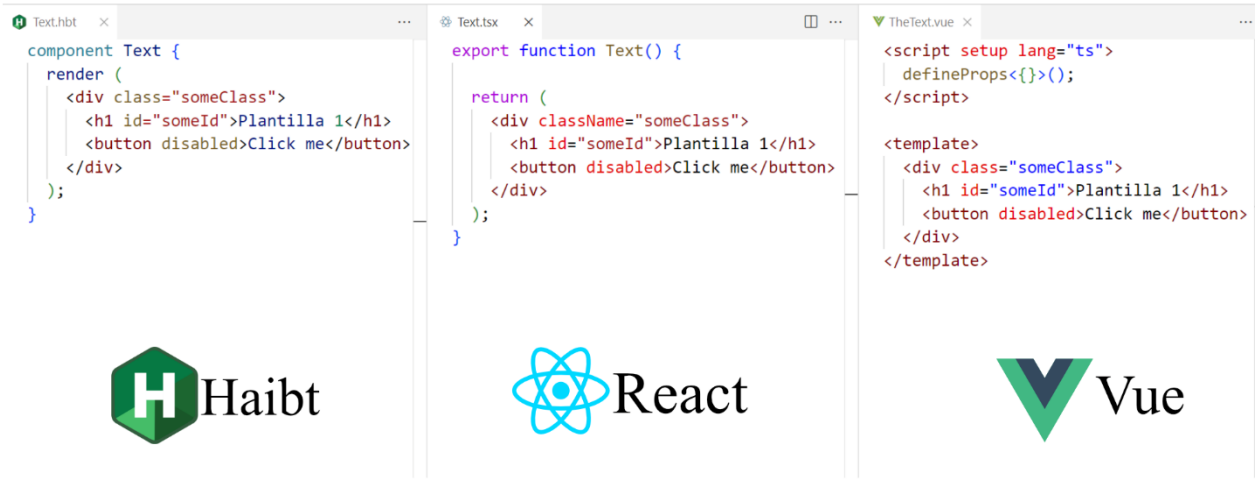

Figura 49 Plantillas HTML con atributos de Haibt, Fuente: creación propia

En la Figura 49 se presentan tres atributos, dos de tipo text, estos siendo class e id, estos utilizados en las primeras etiquetas de la plantilla. Ambos atributos se generaron correctamente en ambas salidas del compilador. El tercer atributo, disabled de tipo booleano es un atajo para indicar que el botón esta deshabilitado, en el código generado este mismo atributo acortado se preserva sin problemas. Dado que los atributos preservaron su valor y ubicación de manera íntegra se concluye que la quitan prueba también termina de manera exitosa.

Sexta prueba. Estilos CSS. Las plantillas por si solas no pueden proporcionar una interfaz de usuario muy sofisticada, para lograr eso la incorporación de CSS es indispensable. La formar en que Haibt incorpora los estilos es a través de una estructura dedicada llamada styles, dentro de ella solo se puede escribir CSS, de esta menara la sintaxis se puede mantener aislada y simplificar la escritura. Esta prueba consistirá en crear un componente relativamente sencillo que utilice dos clases CSS, dichas clases estarán definidos en una estructura de estilos aledaña. La expectativa será que las clases CSS utilizadas sean reemplazados por los textos con un valor igual al nombre de dichas clases CSS y que la estructura como tal desaparezca transformados en CSS que si se pueda utilizar en el navegador.

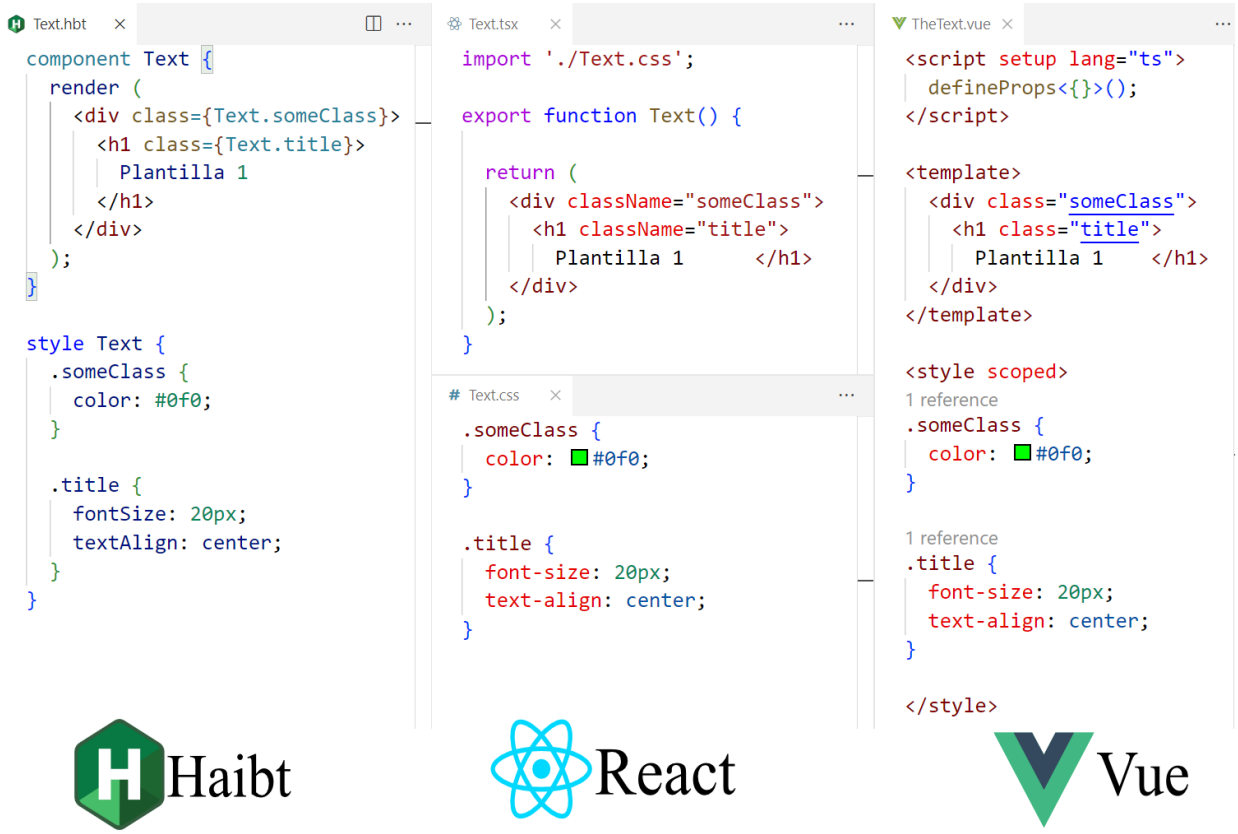

Figura 50 Plantillas HTML y CSS en Haibt, Fuente: creación propia

El resultado de la compilación del código fuente mostrado en Figura 50 muestra varios efectos interesantes. Uno de estos es que la expresión Text.someClass fue reemplazada por un string equivalente, como esto ocurre antes de que la plantilla se procese los nombres del componente y de estilo no colisionan, es decir se puede un solo nombre para dos cosas distintas. Esto fue un efecto más bien inesperado.

El otro efecto observado es que el componente Text si creo objetos en React (centro) una función y en Vue (derecha) un SFC (Single File Component) y con los estilos esto no pasa, en el caso de React, se genera un archivo CSS (centro, inferior) que luego ce referencia en el componente de React (centro) pero sin crear ningún objeto adicional. En el caso de Vue (derecha), este posee una estructura dedicada para estilos CSS de manera nativa por lo que Haibt nuevamente toma ventaja de eso y únicamente adapta el contenido a la estructura de Vue. Un tercer efecto un tanto inadvertido es que en Haibt las reglas CSS se escriben utilizando camel case, pero estas reglas en el CSS estándar se escriben con u guion, Haibt se encarga automáticamente de reemplazar estos casos según se necesario para poder producir CSS que sea funcional.

Los resultados de la prueba mayormente cubren las expectativas, sin embargo, el hecho que un nombre pueda referirse a dos entidades distintas en el programa sin indicar un contexto claro puede abrir la posibilidad de que algunos programas referencien de manera errónea algunos datos, llevando a compilaciones de programas con comportamientos inesperados, por esta razón se concluye que esta prueba es una falla hasta que se corrija ese problema.

Séptima prueba. Propiedades del componente. Hasta esta prueba nuestro componente únicamente ha pasado data hacia sus componentes hijos en la plantilla. Esto raramente ocurre en aplicaciones reales, por tanto, es necesario probar que los mecanismos que Haibt tiene para recibir data de manera externa también funcionen. Para esta prueba se definirán tres atributos con los tipos nativos de Haibt, la expectativa siendo en este caso que los programas generados también permitan la obtención de estas tres propiedades conservando los tipos y nombres definidos pero adaptados a los mecanismos de cada Framework.

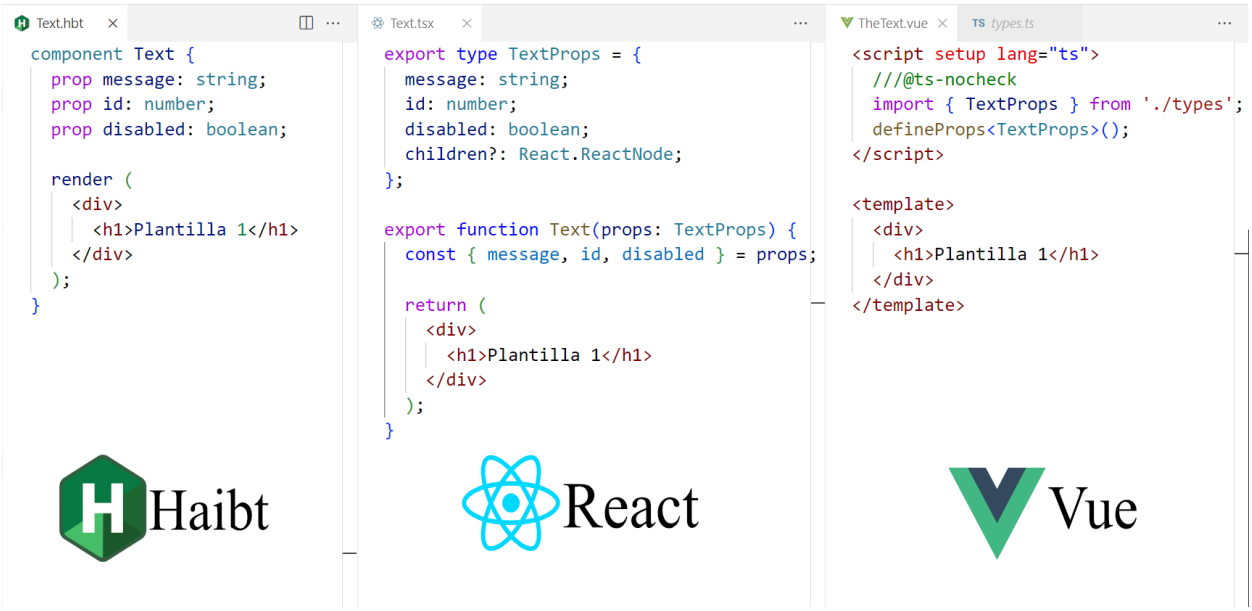

Figura 51 Componente Haibt con tres propiedades, Fuente: creación propia

Con el código generado por el compilador de Haibt a partir del código Fuente (izquierda), se puede validar que la creación de las propiedades tanto en React (centro) como en Vue (derecha) se ha completado de acuerdo con las expectativas, por tanto, se concluye que esta prueba es un éxito.

Octava prueba. Interpolación de variables y expresiones. Ya que se validó que declara y recibir datos desde el exterior funciona adecuadamente es momento de utilizarlos en las plantillas, este es un caso de uso común ya que muchos de los parámetros que el componente recibe a menudo se utilizan como reemplazos de texto en las plantillas o para otros cálculos que afectan el comportamiento del componente. La expectativa de esta prueba es que dado un texto y un numero estos se puedan agregar a la plantilla para que al momento de ejecución estos se reemplacen por los valores que representan.

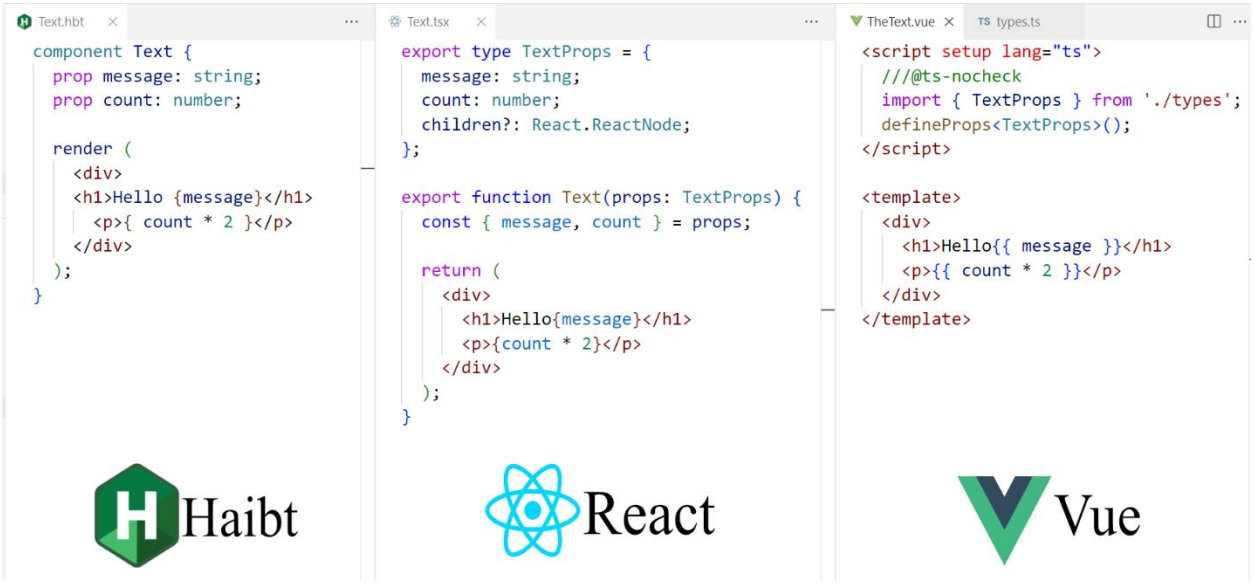

Figura 52 Componente Haibt con platilla interpolada, Fuente: creación propia

Tras revisar el código generado por el compilador se ha determinado que las interpolaciones como tal, están bien formadas. Es decir, cumplen con la expectativa, sin embargo, un problema que se detectó con la plantilla en este momento de la revisión es que esta no preserva los espacios de manera adecuada. Se hicieron pruebas con otros códigos similares y fue posible concluir que los espacios y otras formas de espacios en blanco del texto son ignorados, como los saltos de línea en ciertos casos específicos. Esto no implica que el programa funcione o llegue a funcionar de manera incorrecta pero dada la posibilidad de que se lleguen a mostrar texto unidos, cuando esta no era la intención, causa que esta prueba se clasifique como una falla.

Novena prueba. Enlace de atributos. En muchos de los casos los atributos que los componentes pasan a sus hijos no son valores que se mantiene estativos a lo largo de la ejecución del programa, sino que se mantienen cambiando mayormente debido a la interacción del usuario o como respuesta a una acción que este haya desatado, por ejemplo, cargar información desde el servidor. La expectativa de esta prueba es asignar las

propiedades del componente padre a las propiedades de los distintos hijos y confirmar que se pueden interpolar expresiones y variables para enlazar sus valores a los atributos, así cada vez que estos se actualicen, los componentes hijos también tendrán un valor actualizado.

```
component Text {
  prop message: string;
  prop count: number;
  prop disabled: boolean;

  render (
    <div>
    <h1>{message}</h1>
      <p>{ count * 2 }</p>
      <button disabled={disabled}>
        Click me
      </button>
    </div>
  );
}
```

Figura 53 Código haibt de la novena prueba, Fuente: creación propia

Desafortunadamente, el compilador no cumplió con las expectativas de la prueba. En su lugar se generó un error de compilación sugiriendo que una propiedad de tipo booleano no se puede asignar a un atributo de tipo booleano. Esto desde luego es erróneo y habría que depurar esta parte del sistema de tipos para entender por qué el compilador llega a ese resultado. La salida de consola se puede ver en Figura 54.

```
Compilation complete
PS D:\Documentos\GitHub\output\mylib> crossbind -p react -l info .\Text.hbt
Found 1 files to compile
Files:
.\Text.hbt
.\Text.hbt (10,15): type disabled is not assignable to property disabled type boolean
Compilation failure, found 1 errors
PS D:\Documentos\GitHub\output\mylib>
```

Figura 54. Salida del compilador de Haibt con el código de la octava prueba, Fuente: creación propia

Décima prueba. Inicialización de propiedades con valores por defecto. Dado que las propiedades son uno de los pocos mecanismos que los componentes tienen para intercambiar data, es bastante común encontrarse con componente que posee varias propiedades, a veces incluso más de una docena diferente. No obstante, es poco conveniente tener que definir todas las propiedades cada vez que se desea usar un componente, por esta razón se necesita una forma de poder asignar valores por defecto a las propiedades, de este modo ya no será necesario especificar todas y cada una, para los casos comunes se puede confiar en los valores por defecto. La expectativa de esta prueba es poder inicializar las propiedades con algún valor por defecto y que estas se vuelvan opcionales para el consumidor del componente.

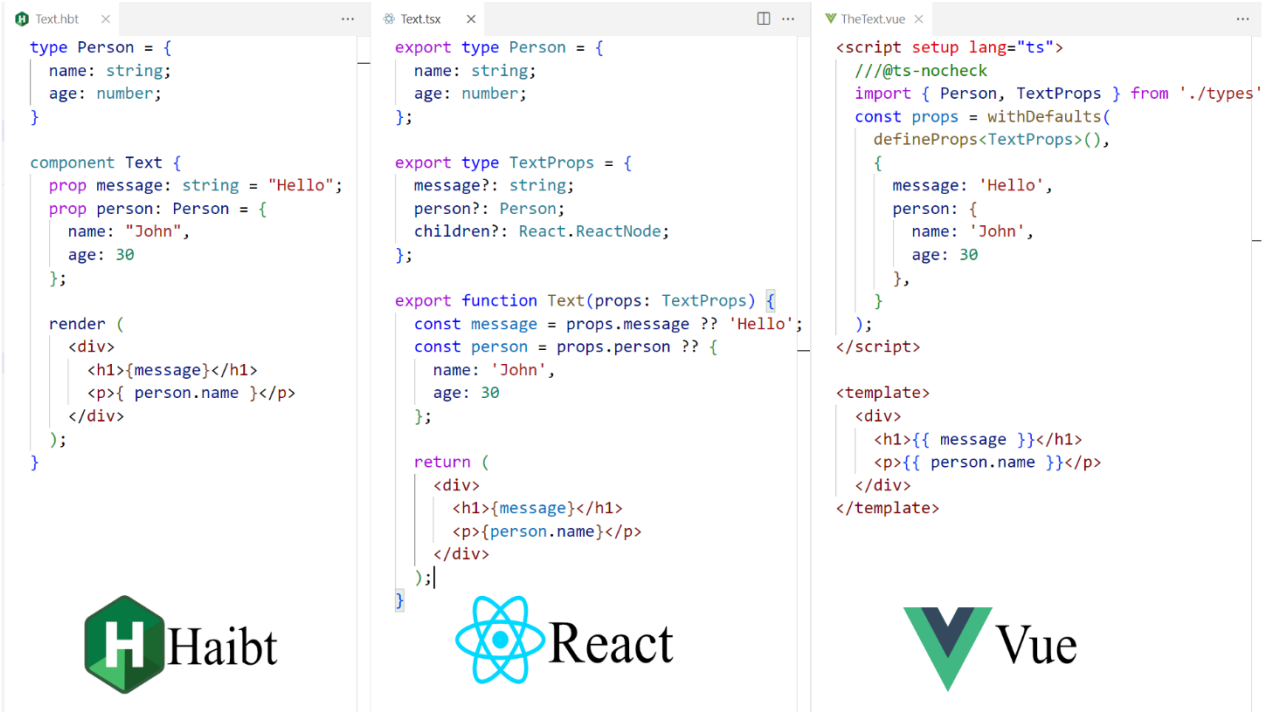

Figura 55. Código haibt con propiedades con valores por defecto, Fuente: creación propia

El código Haibt (izquierda) consiste en un tipo definido por el usuario llamado Person y un componente llamado Text que recibe dos propiedades. El tipo Person posee dos propiedades, las propiedades son name, de tipo texto y age, de tipo numérico, mientras que el componente Texto tiene una propiedad llamada message de tipo texto y un valor por defecto “Hello”, su otra propiedad es un objeto literal con los campos name inicializado en Jhon y el campo age en un valor de 10.

En la plantilla del código Haibt (izquierda) se hace uso de estas dos propiedades para interpolarlas en las plantillas, así el mensaje cambiara dependiendo de lo indicado en la propiedad message y si no se provee

ningún valor, entonces usara el valor “Hello”. La otra propiedad de un tipo personalizado utiliza el valor del campo name en Person para mostrar ese dato en la plantilla.

Con las semánticas del programa Haibt (izquierda) explicado se puede confirmar que el programa en React (centro) también tiene las mismas semánticas. El compilador ha generado de manera correcta el código React (centro) del componente y ha interpolado de manera correcta las expresiones que se utilizan en la plantilla. Además, los tipos declarado por el usuario también se ha conservado al igual que la inicialización de la propiedad de tipo Person. En el caso de código Vue (derecha) este también ha sido generado de manera correcta. Por esas razones se concluye que esta prueba ha sido un éxito.

Nota: el formato del código de Vue (derecha) se modificó manualmente para poder tomar una captura de pantalla más adecuada. A parte de eso es la misma salida que el compilado generó.

Décimo primera prueba. Renderizado condicional. Otra de las situaciones típicas es mostrar y ocultar elementos de manera dinámica basados en alguna condición. Para lograr esto utilizaremos la directiva if de Haibt. La expectativa de esta prueba es que los mecanismos que habilitan el renderizado condicional en los Frameworks objetivo se mantengan coherente con las semánticas de Haibt.

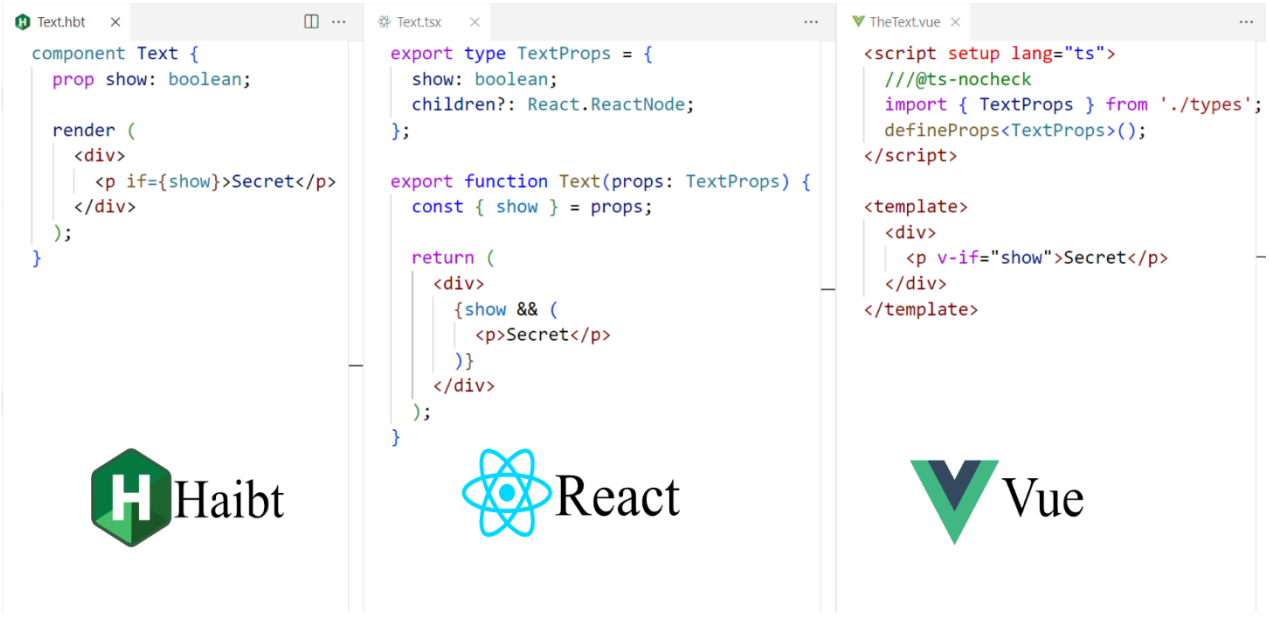

Figura 56. Código haibt utilizando renderizado condicional, Fuente: creación propia

Nuevamente el código generado por el compilador mantiene la semántica de mostrar y ocultar elementos de manera condicional, pero adaptando el código a los mecanismos especifico que utiliza cada uno de los frameworks. En el caso de React (centro) el renderizado condicional se implementa como una expresión que retorna JSX en caso de mostrar el contenido o retornar cualquier de los tipos que react ignora al momento de mostrarlo en pantalla como null o false.

Con respecto a Vue (derecha) este nuevamente vuelve a implementar un mecanismo idéntico a Haibt por lo que la compilación en este aspecto se reduce a cambiar el nombre de la directiva.

Décimo segunda prueba. Renderizado condicional alternativo. Por razones similares a las de la prueba anterior ocurren situaciones donde las aplicaciones necesitan mostrar y ocultar contenido, pero al mismo tiempo necesitan mostrar un contenido alternativo que lo sustituya. Para estos casos podrían usarse dos directivas if con condiciones inversas, pero hay una solución más elegante. La directiva else.

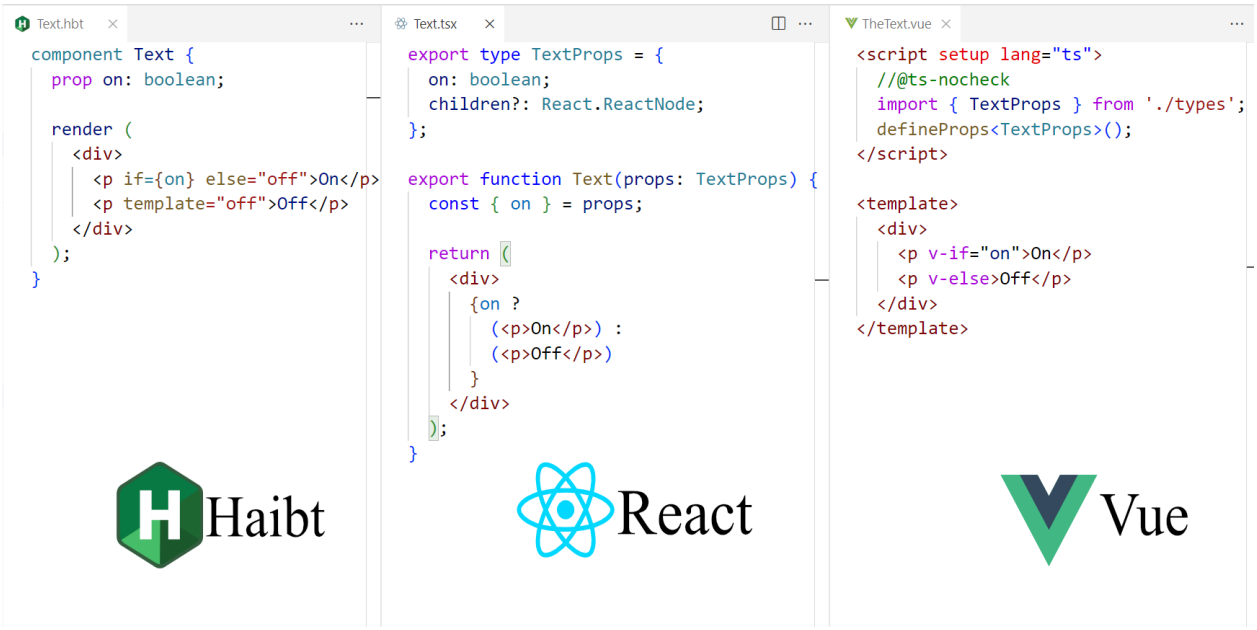

Figura 57. Código haibt utilizando directivas If, Else y Template, Fuente: creación propia

En el código Haibt anterior de la Figura 57 (izquierda) se puede ver le uso de la directiva if en conjunto con la directiva else. La directiva else toma como su único argumento un texto que debe apuntar al nombre de la plantilla que debe mostrar en caso de que la condición de la directiva if sea un valor falso. Para asignar nombre a las plantillas es necesario el uso de una directiva auxiliar, esta es la directive template.

La directiva template no tiene mayor función excepto como una manera de referenciar las plantillas por la directiva else, por esta razón se encuentra bajo el estatus de auxiliar. En el caso de Vue (derecha) cuando las plantillas primarias y alternativa están una después de la otra es posible utilizar la directiva else y esta automáticamente se enlazada con la directiva if anterior. Finalmente, en el caso de React (centro) como este no posee directivas de manera nativa corresponde retornar el JSX de una expresión, si están juntas conviene usar un operador ternario para simplificar el código generado.

Existirán caso donde las plantillas no se encuentran una al lado de la otra, sino que tiene elementos intermedios. Es en estos casos donde es necesario separar la estructura if-else en dos if con condiciones inversas. Afortunadamente, el compilador de Haibt toma esto en cuenta y aplica esta distinción de casos según sea conveniente. Un ejemplo de este caso se puede observar en la figura 58.

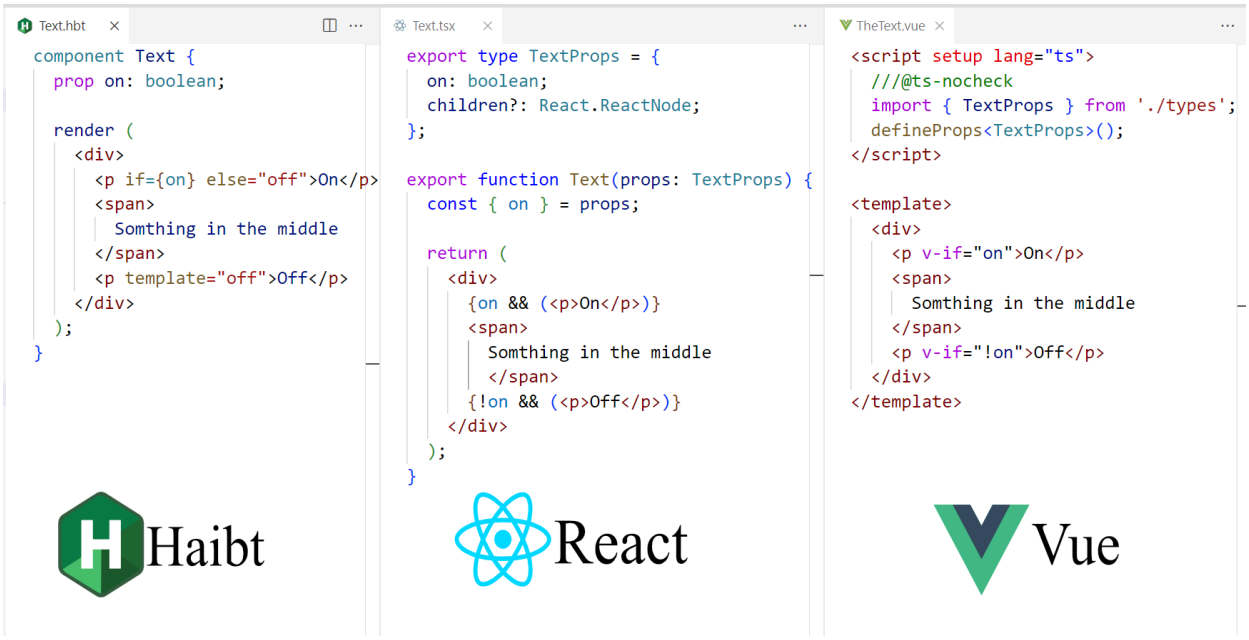

Figura 58. Código Haibt con directiva else separada, Fuente: creación propia

En el ejemplo de la figura 58 se puede ver que tanto React (centro) cómo Vue (derecha) el código generado posee condiciones invertidas para lograr el efecto de plantillas mutuamente excluyentes, mientras que en Haibt (izquierda) esta distinción entre estructuras If-Else juntas o separadas no es necesaria.

6.4 Pruebas de rendimiento

Las pruebas de rendimiento pretenden estresar el compilador de Haibt para encontrar y definir los limites bajo los que o no resulta factible la compilación debido a tiempo de ejecución o cantidad de memoria ocupada, o en el peor de los casos debido a una excepción como resultado de un error. Estas pruebas se centrarán únicamente estresando la parte de la plantilla ya que en programas reales la plantilla suele ser la parte más extensa y compleja.

6.4.1 Pruebas con elementos lineales

Esta suite de pruebas se subdividirá en dos categorías. La primera siendo pruebas de elementos lineares, esto significa que todos los elementos en la plantilla están al mismo nivel, además en cada prueba la cantidad de estos incrementara también de manera lineal. La estructura del código de las pruebas lineares se ve como en la figura 59.

```
component component0 {
  render (
    <div></div>
    <div></div>
    <div></div>
    <div></div>
    <div></div>
    <div></div>
  );
}
```

Figura 59 código ejemplo de una prueba linear. Fuente: creación propia

El ejemplo de la Figura 59 es uno corto pero el mismo patrón se usará y repetirá múltiples veces. El compilador funcionó correctamente con las configuraciones por defecto de Node JS hasta 865 elementos, a los 866 elementos se generaba un fallo en Node JS, RangeError: Maximum call stack size exceded. El error reportado indica que se ha alcanzado el límite máximo de funciones que pueden llamarse en secuencia. Este error ocurre en una librería de parseo probablemente debido a que el diseño de la gramática lo fuerza a utilizar recursión de manera intensiva.

A continuación, se presentan los datos para 18 sets de prueba. Este set consiste en 51 elementos y progresivamente se añaden 50 elementos adicionales hasta 850, luego de eso se agrega uno adicional con 865 elementos. Estos sets lineales nos ayudaran a comprobar como el compilador se comporta con una cantidad creciente de elementos secuenciales.

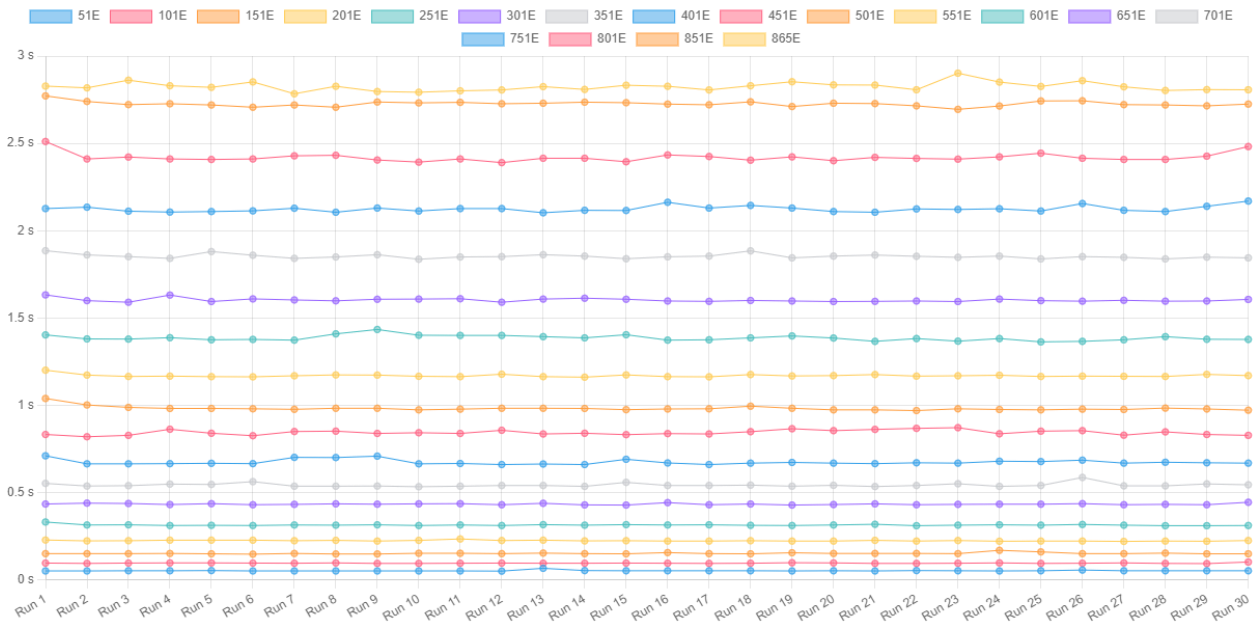

Figura 60 set de pruebas lineares. Fuente: creación propia

En la Figura 60 tenemos una gráfica que muestra para cada uno de los 18 sets, el tiempo que le tomo ejecutarse de principio a fin. Cada set se ejecutó 30 veces, terminando y volviendo a iniciar el proceso de Node JS para evitar que algún tipo de cache o de optimizaciones del interprete afecte los resultados. Bajo condiciones ideales estas deberían ser líneas perfectamente horizontales indicando que para una cantidad de elementos determinada el tiempo de ejecución siempre es el mismo. Debido al planificador del sistema operativo, procesos en segundo plano entre otras variantes estos tiempos son desde imperceptibles hasta significativos. Con el set de datos actuales se puede concluir que el margen de error es aceptable y que no existe ninguna ejecución del programa que este muy alejado del promedio.

Ya que el ser de datos se considera aceptable, para poder visualizar el escalado del compilador con la cantidad de nodos en la entrada se procederá a incluir los promedios de cada set en una gráfica de líneas, esto mostrará como el costo en tiempo de ejecución escala con la cantidad de nodos lineales.

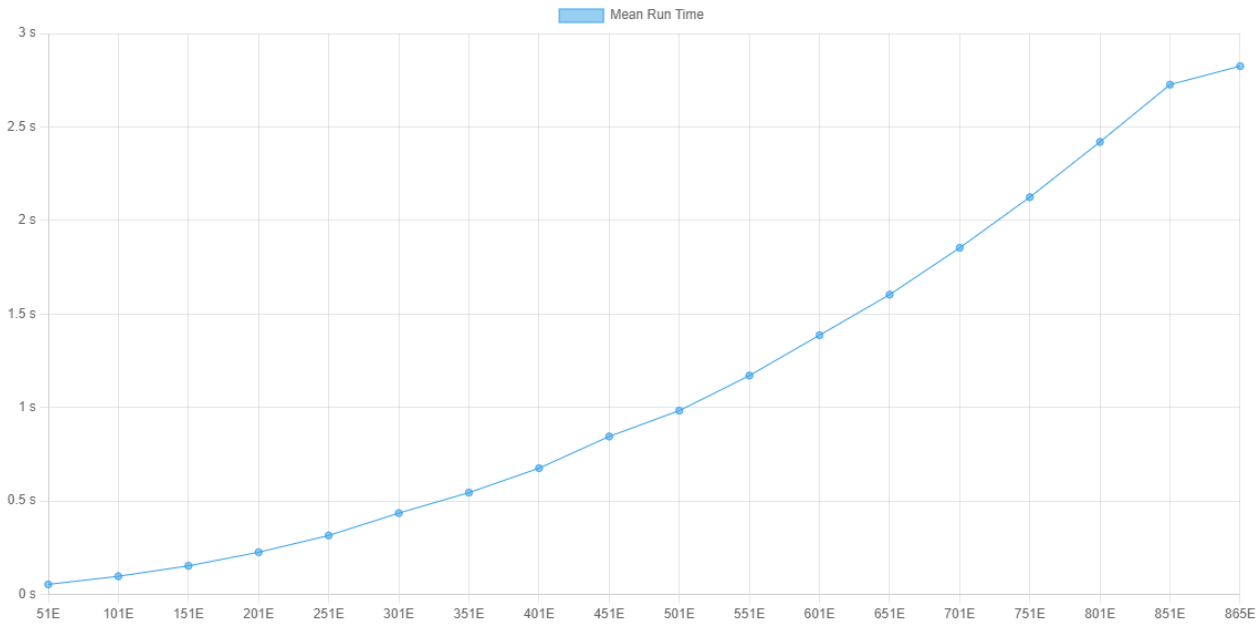

Figura 61 set de pruebas lineares. Fuente: creación propia

La figura 61 muestra el gráfico de los tiempos de ejecución promedio para cada una de las pruebas. Los valores se sitúan tener los 0 y 3 segundos. Con esta información ya podemos intuir que la relación entre los nodos y el tiempo de ejecución no es lineal, en primera instancia debido a la forma curvada hacia arriba de la gráfica y en segundas instancias debido las aproximaciones lineales siempre se encuentran muy lejos de los valores reales. Utilizando un script en Python y el método de mínimos cuadrados se encontró que un polinomio que aproxima muy bien esta grafica es el polinomio de grado tres en la ecuación 1.

$$f(x) = 1.144 \times 10^{-9}x^3 + 1.975 \times 10^{-6}x^2 + 0.0006908x + 0.006838$$

ecuación 1 aproximación del tiempo de ejecución

La ecuación 1 puede servir para interpolar valores en las cantidades de nodos intermedia sin necesidad de ejecutar los 865 sets deferentes. En figura 61 se encuentra la gráfica original y la generada a partir de la ecuación 1.

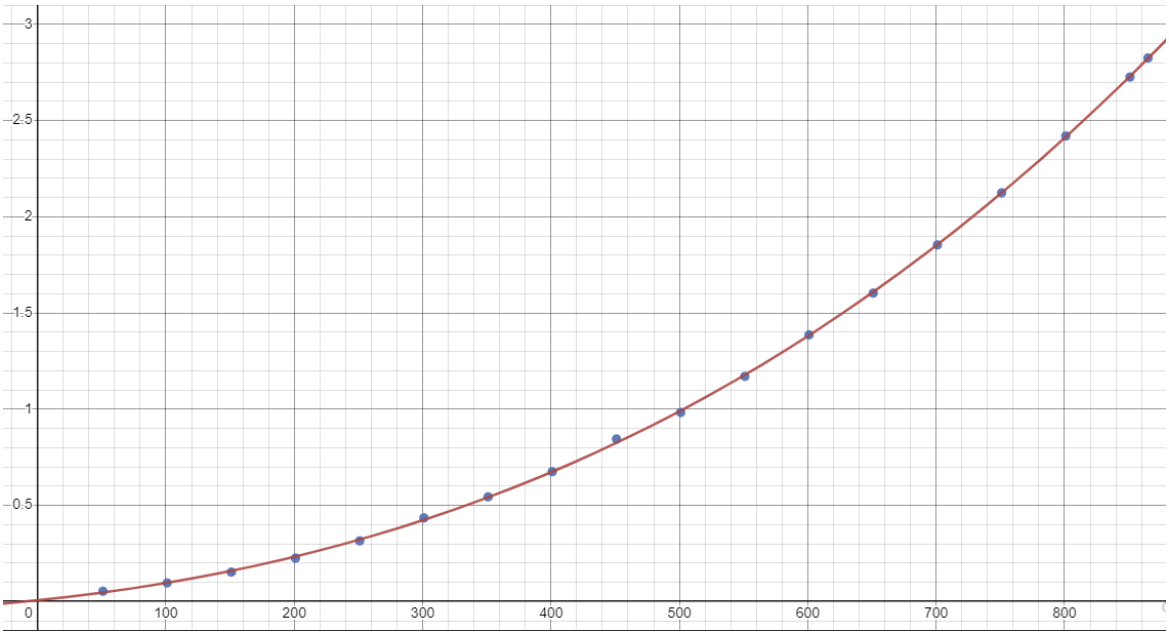

Figura 62 Grafica de la interpolación usando ecuación 1. Fuente: creación propia

Hasta el momento solo se ha tomado en cuenta el tiempo de ejecución total, sin embargo, no todas las etapas del compilador funcionan de la misma forma, por lo que en la siguiente parte del análisis veremos de manera más granular las contribuciones individuales de cada etapa, esto con la intención de identifica aquellas etapas donde una optimización del código genere mayor impacto en la eficiencia del programa, en otras palabras se busca identificar los cuellos de botella que limitan el rendimiento del compilador. El desglosé de las contribuciones individuales se puede ver en figura 63.

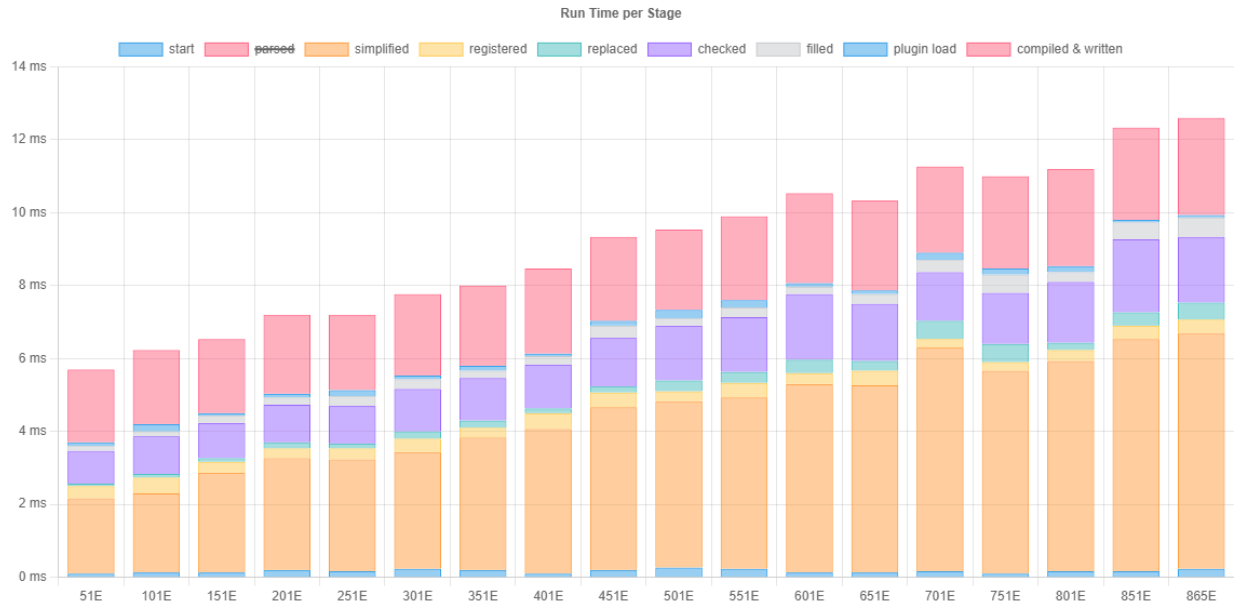

Figura 63. grafica de contribuciones por etapa sin parseo. Fuente: creación propia

En la figura 64 se puede identificar las distintas etapas por colores, y la altura del bloque indica cuando tiempo ha sumado esa etapa al total en milisegundos. Los tiempos no coinciden con la gráfica anterior de promedios debido a que la etapa de parseo de ha dejado fuera de esta grafica debido a que esta es la etapa que más contribuye a alargar el tiempo. Ignorando la etapa de parseo se puede ver que el escalado con la cantidad de elementos con una tendencia que puede ser aproximada con una línea más fácilmente. El que la gráfica se asemeja más a una línea cuando la etapa de parseo no se incorpora a la gráfica sugiere que es precisamente esta donde está el problema. En figura 63 se muestra la gráfica incorporando la etapa de parseo.

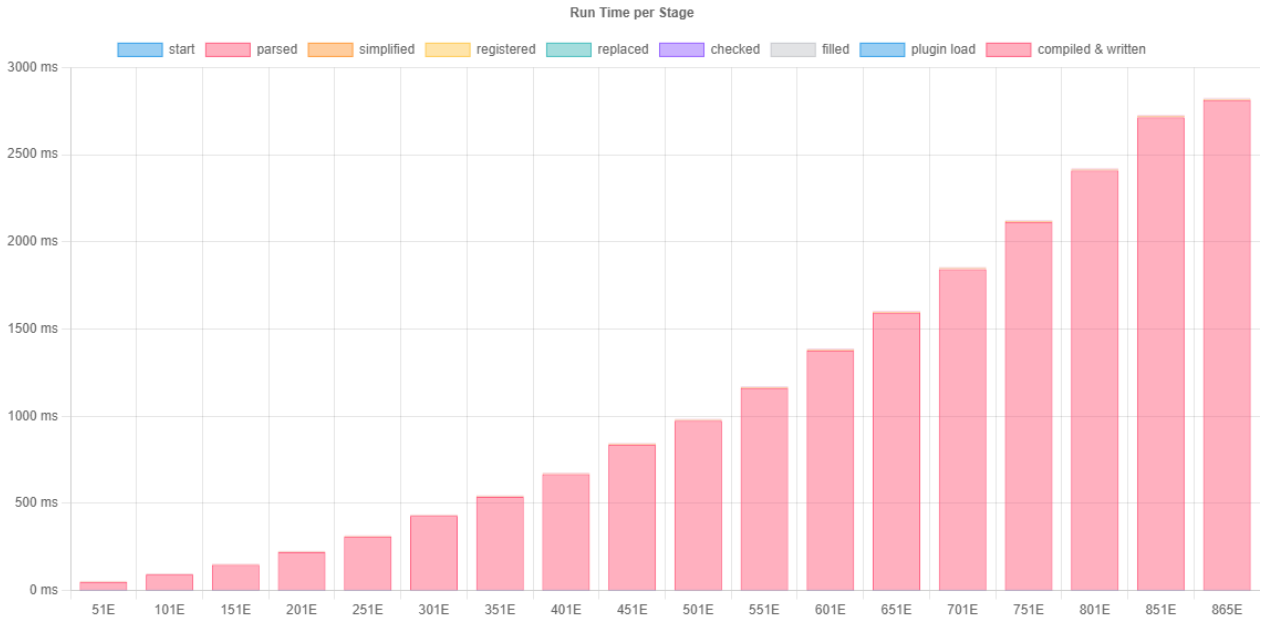

Figura 64. grafica de contribuciones por etapa. Fuente: creación propia

Al incorporar las contribuciones del parser se muestra cuan insignificante es la contribución de las demás etapas juntas, por esta razón se concluye que optimizar el parseo es la etapa que más impactaría a la compilación de elementos lineales.

El tiempo de ejecución de un programa es solo un aspecto para tomar en cuenta cuando se habla de rendimiento, otro aspecto importante de las aplicaciones computacionales es la memoria que utilizan. En la figura 64 se incluye una gráfica de la memoria que la aplicación está utilizando en el heap, en color azul y en rojo la memoria total alocada por Node JS (Memoria reservada para el proceso).

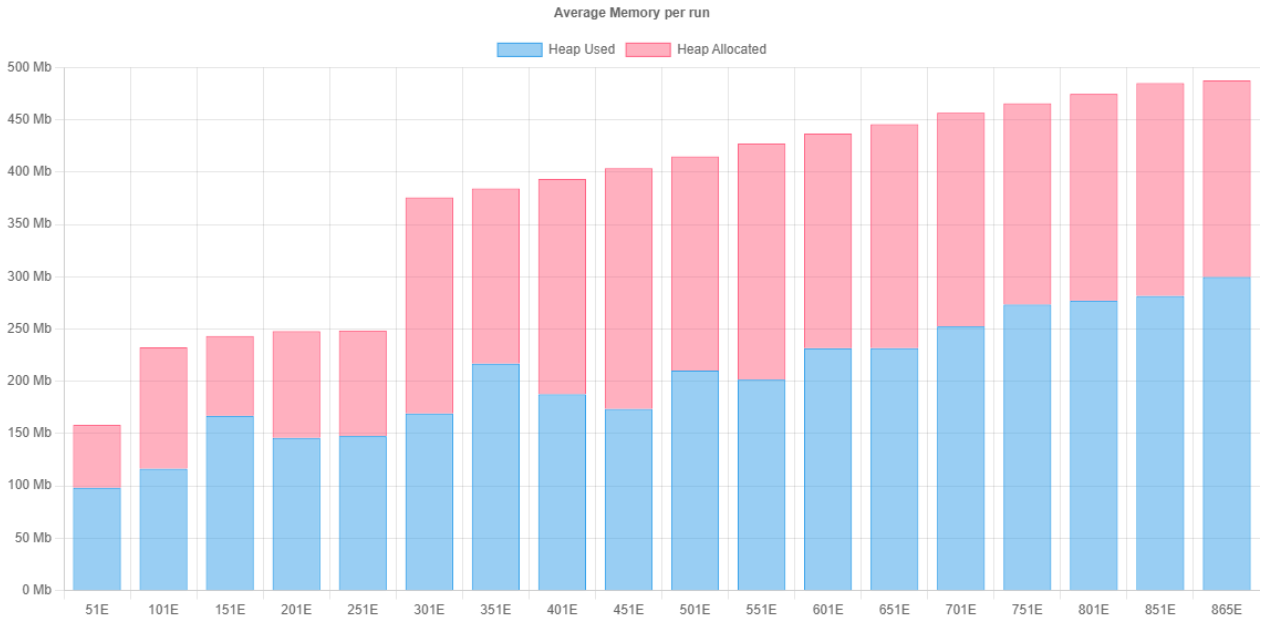

Figura 65. grafica de asignaciones de memoria. Fuente: creación propia

De la gráfica de la figura 65 se concluye que, aunque la cantidad de memoria no es exagerada para los estándares del año 2024 el hecho que un compilador consuma tanto con tan solo 865 elementos sí es descomunal. Este aspecto técnico es, por ende bastante mediocre. Similar al análisis del tiempo de ejecución también se procederá al desglose de las aplicaciones individuales de cada etapa. Esta gráfica se encuentra en figura 66.

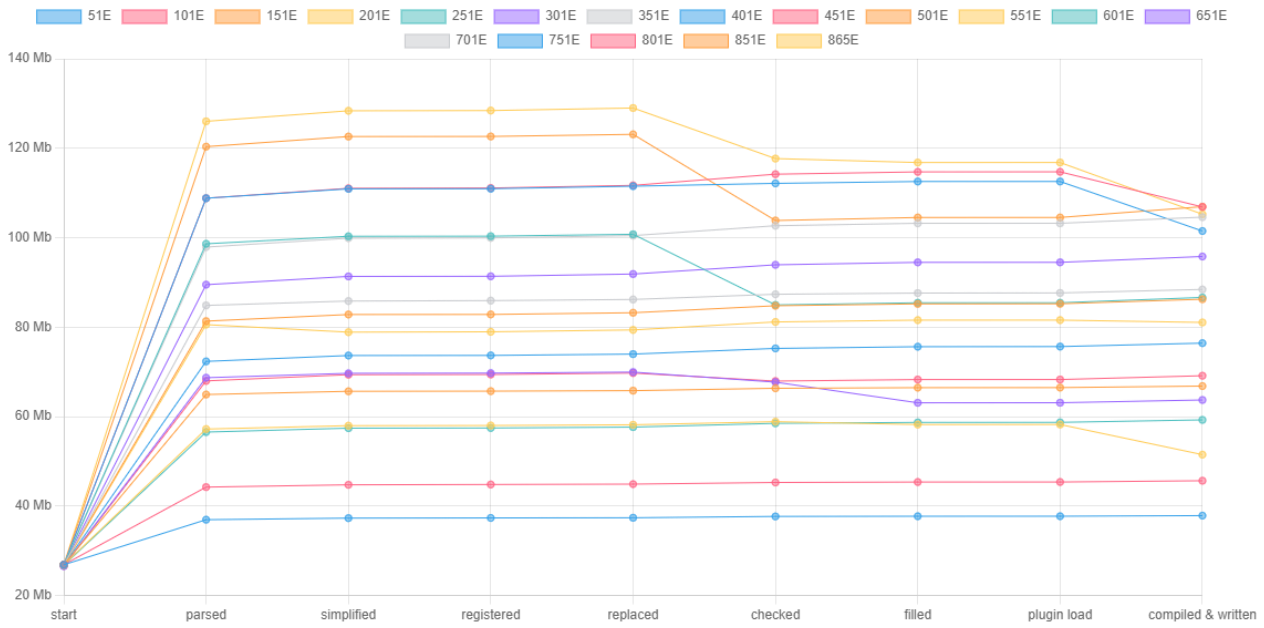

Figura 66. grafica de asignaciones por etapa sin parseo. Fuente: creación propia

De la gráfica en la figura 66 se puede concluir que nuevamente es el parser el que dispara asignación una asignación de memoria desproporcionada si comparamos el inicio, alrededor de 26 megabytes de memoria. Esto sugiere nuevamente que al optimizar el parser causa el mayor impacto en el consumo de memoria del compilador. El resto de las etapas generan líneas que en perspectiva parecen horizontales, siguen disparando asignaciones de memoria al ejecutar sus tareas sin embargo ninguna es tan desproporcionada como las que el parser necesita. En los casos donde la línea va hacia abajo es cuando el recolector de basura tiene tiempo suficiente para liberar la memoria que ya no se necesita.

Eso concluye con el análisis del tiempo de ejecución para elementos y memoria para elementos lineales.

6.4.2 Pruebas con elementos anidados

Las pruebas con elementos anidados consisten en tener estructuras de plantilla HTML con geométricamente más elementos. Los elementos varían en su estructura y van desde la configuración N1-L1 hasta N5-L4, a partir de ese punto el tiempo de ejecución era muy lenta como para esperar a que terminen las 30 rondas. La nomenclatura indica que se generan 5 nodos de etiqueta inicialmente, cada nodo con 5 hijos y cada uno de sus hijos con 5 elementos haciendo un total de 125 nietos y cada nieto con 5 hijos así sucesivamente hasta llegar a nivel cero.

Como el rango de datos es demasiado amplio para mostrarlo en un solo set de gráficos, este se subdivide en 3 categorías. Las subdivisiones son nivel bajo, para cualquier configuración con menos de 157 elementos, nivel medio para cualquier configuración con menos de 1366 elementos y alto par configuraciones con hasta 5461 elementos. Únicamente se utilizará el nivel medio para el análisis, el resto de las gráficas estarán disponibles en GitHub, ver anexo B.

Como se puede deducir por alguna razón el parser se comporta mejor cuando la estructura de los elementos es anidad. Esto es un comportamiento fuera de lo esperado y además un poco impredecibles, por lo que se variaron las configuraciones en los que los elementos se encuentran distribuidos en las plantillas.

En la figura 67 se puede ver un ejemplo de una prueba con una configuración N2-L1, eso significa que tiene dos nodos por cada nivel, y dentro de cada nivel existen nuevamente dos nodos con el nivel reducido en una unidad, es decir cero.

Dado el código del ejemplo la cantidad de etiquetas div es 6 (cada elemento se compone por la etiqueta de apertura y la de cierre). La cuenta llega a 6 debido a que en el nivel uno existe dos div, pero luego en el

siguiente también existirán dos div. En resumen, los dos div del nivel inicial se sumarán con la cuenta de sus hijos (2 div por cada uno) estos siendo 4 y eso hace un total de 6 nodos. El mismo patrón se repite dada cualquier cantidad de nodos N y cualquier cantidad de niveles L.

```
component component0 {
  render (
    <div>
      <div></div>
      <div></div>
    </div>
    <div>
      <div></div>
      <div></div>
    </div>
  );
}
```

Figura 67. Código de prueba en configuración N2 L1. Fuente: creación propia

Al igual que en la sección anterior se empieza visualizando el tiempo total de cada ejecución individual de cada set en una gráfica. Cada set se ha ejecutado 30 veces.

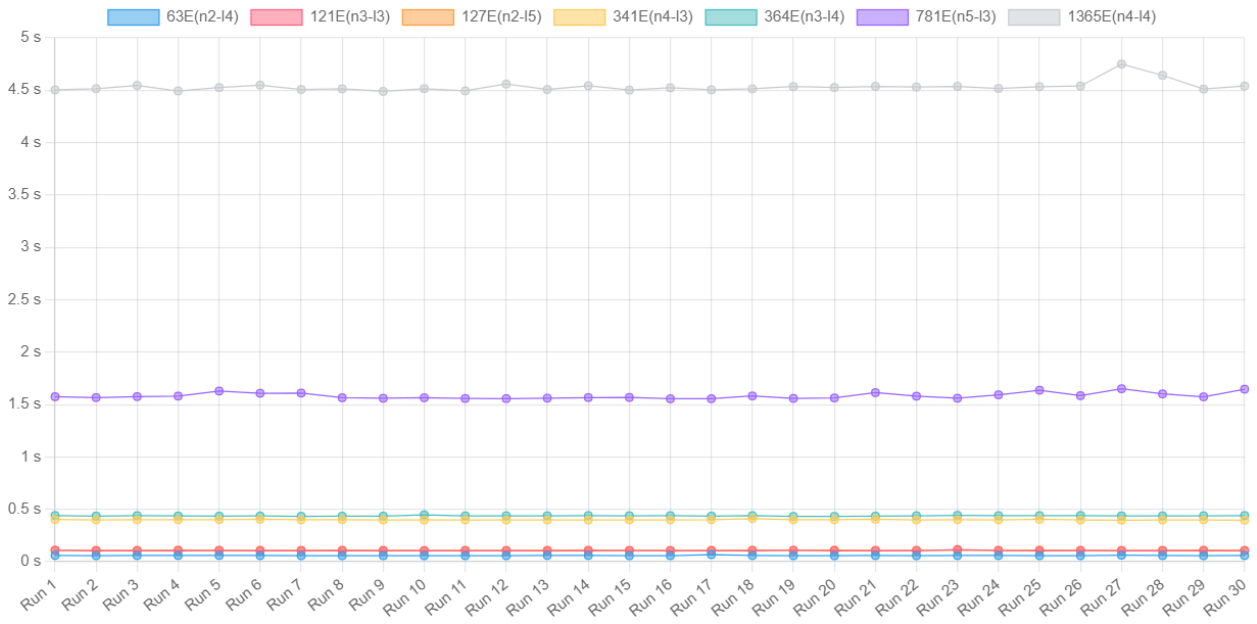

Figura 68. Código de prueba en configuración N2 L1. Fuente: creación propia

En la gráfica de la figura 68 se muestran los tiempos de cada set, los sets con menor cantidad de elementos son los más consistente mientras que los que poseen más elementos se vieron más afectados por variables externas al program debido a su mayor tiempo de ejecución.

Aunque este set de datos no es tan consistentes como el anterior después de una inspección más detallada, se asume la inconsistencia de los resultados en nombre de poder obtener información que revele las tendencias del comportamiento del programa en términos de memoria y tiempo de ejecución.

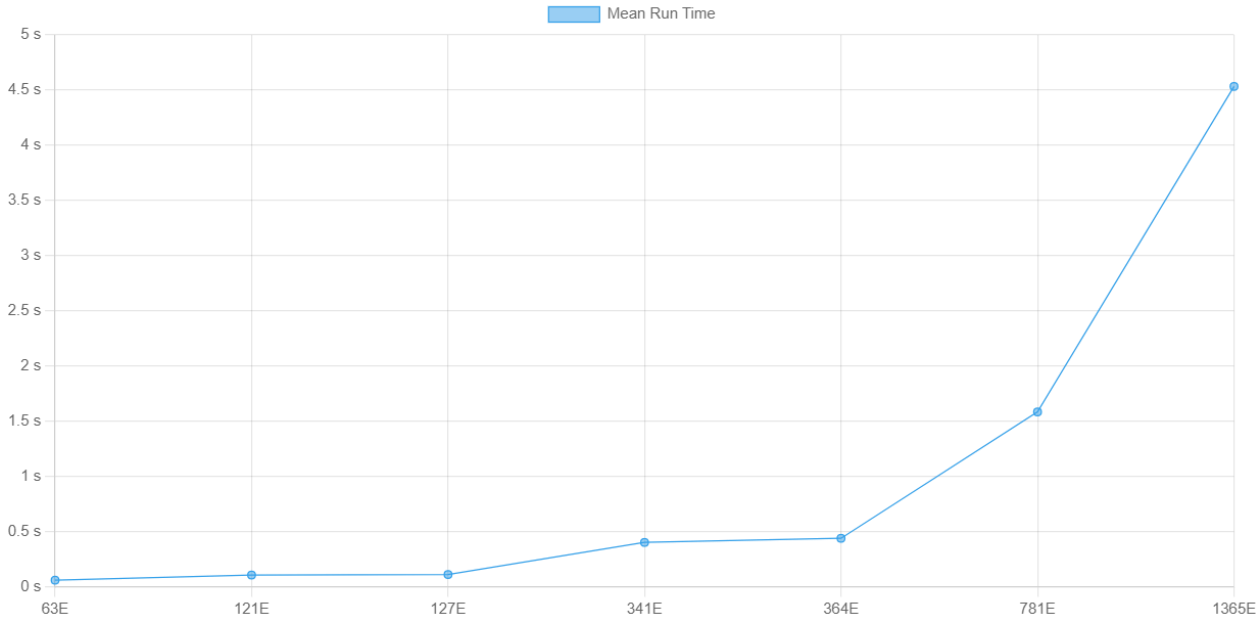

Figura 69. Tiempo de ejecución promedio para el set medio. Fuente: creación propia

Lo primero se evidencia es que ahora el parser logro terminar el proceso sin lanzar excepciones, es decir no fallo. Aunque aún sigue escalando de manera pésima con la cantidad de nodos en la entrada. Debido a que el parser es generado indagar en por que este fenómeno ocurre implica ir a revisar código fuente tanto generado como de la librería que se utiliza. El parser es generado por una librería llamada ANTLR4 que es un generador de parsers LL<*>, utilizándolo en para parsear otras gramáticas puede procesar muchísimos más nodos, por lo que la conclusión sobre este asunto es que la gramática está mal diseñada por abusar de las palabras vacías y eso es lo que ha generado un parser inefectivo.

En las pruebas se logró procesar con éxito hasta 5461, sin embargo, no fue porque se haya encontrado un límite como, sino más bien el tiempo de ejecución con más nodos resultaba en tiempos de ejecución demasiado largos (30 minutos aproximadamente), por lo que realizar 30 pruebas, incluso estás siendo automáticas no era muy factible debido a los largos tiempos de espera.

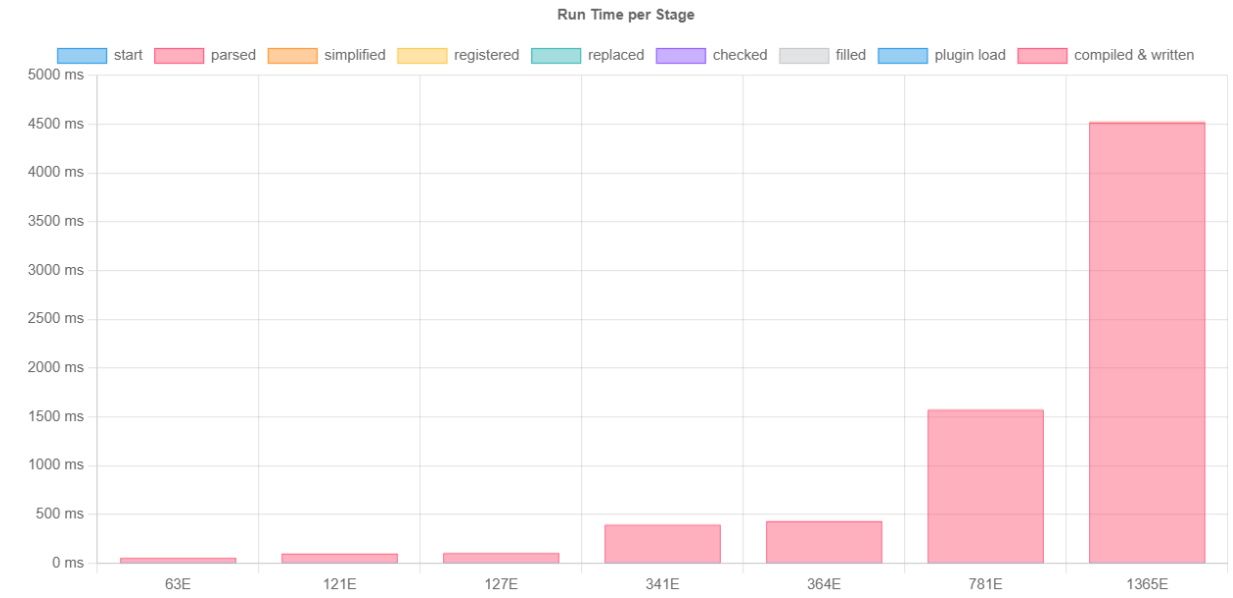

Figura 70. Tiempo de ejecución por etapas del set medio. Fuente: creación propia

Al igual que en la sección anterior utilizando elementos lineales el parser acapara la inmensa mayoría del tiempo de ejecución.

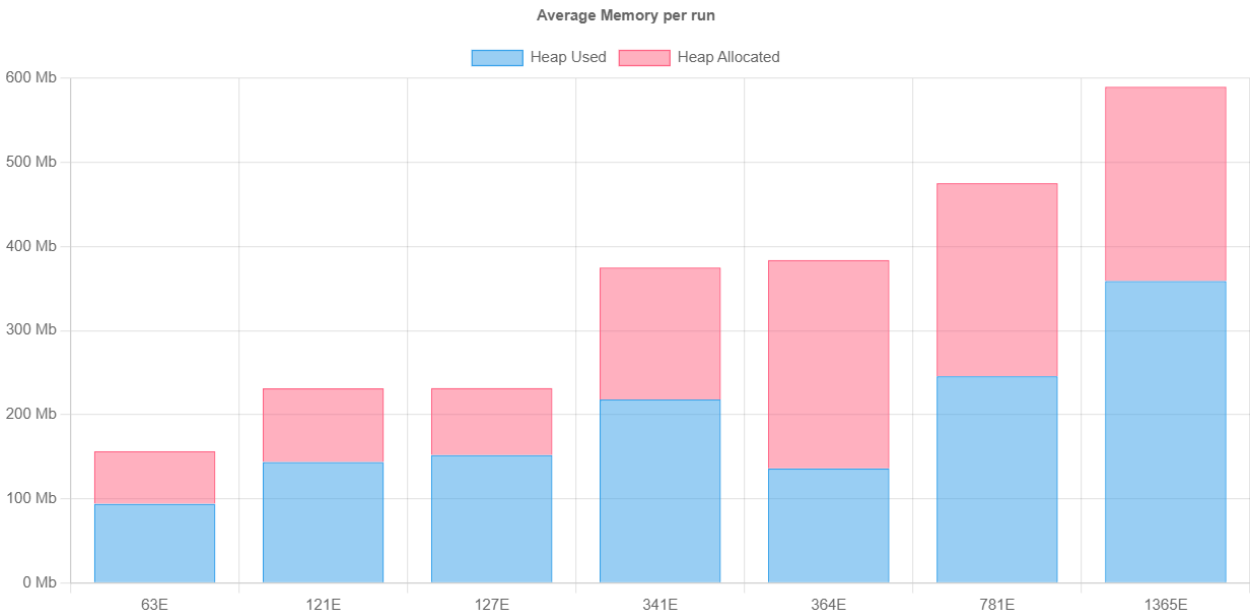

Figura 71. Tiempo de ejecución por etapas del set medio. Fuente: creación propia

Tras la revisión de la gráfica de la figura 71, ya es evidente que mientras el parser no se mejore en alguna medida no se recomienda el uso de este compilador para nada mas haya de unos cientos de líneas. Dado su pobre rendimiento en parseo, sin embargo, el hecho que solo esta parte represente problemas graves también facilita la solución, ya que, solo es necesario enfocarse en esta etapa.

CAPÍTULO 7. CONCLUSIONES Y RECOMENDACIONES

7.1 Conclusiones

- Al descartar las diferencias sintácticas entre los distintos frameworks, se abstraen en común las propiedades, la plantilla y el estado local, esto significa que estos tres elementos son lo único que se necesita para definir un componente, por esta razón es posible que un lenguaje externo compilado que comparta los mismos fundamentos también los pueda generar.

- Utilizar una abstracción sobre tecnologías web de estilos, scripting y plantillas en una sintaxis especial de un DSL, permite a los desarrolladores enfocarse en escribir sus librerías una única vez. Al compilarla dicho DSL, el código generado conserva las mismas características que el original, pero sin preocuparse por los detalles sintácticos de cada framework.

- Una gramática de tipo LL 1 satisface las necesidades del lenguaje, dotándolo de la expresividad para poder ser usado en formas variadas. La estructura de esta gramática presenta todos los elementos necesarios para que el compilador pueda analizar el código sin la necesidad de obtener esta información de otras fuentes y aun así generar código con las mismas características.

- La estructura de plugins implementada en el compilador funciona espléndidamente, dado que las complejidades individuales de cada uno, no se entrelazan con el código base del compilador. Esto hace posible el expandir el compilador con más plugins sin afectar a los que ya existen. Además, se pueden realizar optimizaciones y ajustes personalizados a cada framework.

- Los compiladores son herramientas muy eficaces para resolver el problema de múltiples implementaciones para un único tipo de programa, pero variando su plataforma. En este caso, el compilador Haibt permite llevar interfaces gráficas de componentes en el DSL hacia los distintos frameworks.

7.2 Recomendaciones

- Antes de iniciar con el proceso de generación de código directamente en un archivo, es más conveniente transformar el AST en otra forma de representación intermedia, ya que, la manipulación de estructuras en memoria se puede realizar de una forma mucho más simple y eficiente que modificar string en memoria. La principal desventaja de generar código directamente del AST es que una vez escrita una sección de este no se puede regresar a editarla, por esa razón modificar una estructura temporal en memoria lo vuelve más sencillo.
- Es necesario crear una API que está mejor definida entre el compilador y los plugins, ya que actualmente se hace uso del tipado dinámico de JavaScript para poder pasar datos entre plugins y compilador. Esta situación puede llevar a API inestables y que no sean confiables, por esa razón hay que crear una API que sea más estricta en los argumentos y los tipos de retorno.
- Robustecer el análisis de los programas, actualmente el compilador aún deja pasar ciertos errores que podrían llevar a un programa incorrecto o que no sea utilizable en el lenguaje de destino, por esta razón aún hace falta trabajar en la verificación de más tipos de errores para poder obtener programas más correctos.
- La sintaxis de CSS se define por separado, sin embargo, heredó la limitación de no poder tener identificadores separados por guiones, por esta razón hizo falta crear un paso adicional para renombrar estas reglas de CSS, si se refactoriza la gramática para poder soportar este tipo de identificadores en CSS esto causaría la eliminación de un paso en el proceso de compilación y habilita a que el código CSS de otras bases de código simplemente pudiera ser importado utilizando el copiar y pegar sin necesidad de ajustar nada.
- El sistema de tipos es bastante limitado en cuanto a lo que puede definir, y dada la naturaleza dinámica de JavaScript es importante dotarlo de más herramientas que les permitan a los desarrolladores poder tipar una gama más amplia de objetos. Esto se podría lograr incorporando adiciones de TypeScript como genéricos, tipos mapeados, tipos de alto orden y tipos de utilidad.
- Ya que la gramática ha sido desarrollada y revisada en las partes más prominentes, ya es seguro abandonar la generación de parsers con ANTLR e implementar un parser que esté más adecuado a las necesidades del proyecto y del lenguaje Haibt.

GLOSARIO

API: Estas son todas aquellas funcionalidades que se exponen entre el compilador principal de Haibt y sus distintos plugins, estos pueden ir en cualquier sentido. Esto les permite reutilizar estas funcionalidades sin necesidad de que cada uno las defina de manera separada cada vez que se utilizan, además define la forma en la que interactúan y comunican ambos programas.

Compilador: Este es el programa encargado de tomar el código fuente, interpretarlos, validarlo, optimizarlo y generar una versión equivalente en otro lenguaje de programación, por lo general de nivel más bajo.

Estado: En el contexto de componentes web el estado hace referencia al conjunto de información y datos que determinan como un componente se ve ante el usuario o como este se comporta. Ejemplo de estos son una barra de carga que puede estar cargando o no y un botón deshabilitado que se comporta diferente en ambos estado, activo o inactivo.

Módulo: Dentro de Haibt un módulo hace referencia a un componente, un estilo o una definición de tipo, básicamente un módulo se vuelven aquellos elementos que se pueden declarar al nivel más alto del lenguaje. Así por ejemplo dentro de un archivo puede haber múltiples módulos, pero un módulo sólo hace referencia a un único elemento del lenguaje.

Plugin: Son programas que se definen de manera externa y luego se cargan de manera dinámica en otro programa que actúa como anfitrión. Esta es una relación donde el programa anfitrión ofrece ciertas funcionalidades comunes y útiles y en retorno el plugin dota al programa base con nuevas funcionalidades que originalmente no poseía.

Renderizar: En el contexto de componentes web hace referencia a la acción de dibujar en la página web las etiquetas HTML que conforman la interfaz gráfica, rellenando los espacios con información dinámica como números o textos donde haga falta.

REFERENCIAS

Aho, A., Lam, M., Sethi, R., & Ullman, J. (2007). *Compilers Principles, Techniques, & Tools Second Edition.* Pearson Education, Inc.

Bampakos, A., & Deeleman, P. (2023). *Learning Angular - Fourth Edition.* Packt Publishing.

Buckler, C. (2022). *Node.js: Novice to Ninja.* SitePoint.

Camden, R., Di Francesco, H., Gurney, C., Kirkbride, P., & Shavin, M. (2020). *Front-End Development Projects with Vue.js.* Packt Publishing.

Casciaro, M., & Mammino, L. (2020). *Node.js Design Patterns - Third Edition.* Packt Publishing.

Crowder, P., & Crowder, D. (2008). *Creating Web Sites Bible, Third Edition.* Wiley Publishing, Inc.

Fowler, M. (2010). *Domain Specific Languages.* Addison-Wesley Professional.

Frisbie, M. (2023). *Professional JavaScript for Web Developers, 5th Edition.* Wrox.

Nicholus, R. (2016). *Beyond jQuery.* Apress.

Olsson, M. (2014). *CSS Quick Syntax Reference.* Apress.

Pillora, J. (2014). *Getting Started with Grunt: The JavaScript Task Runner.* Packt Publishing.

Rippon, C. (2023). *Learn React with TypeScript - Second Edition.* Packt Publishing.

FUENTES ELECTRONICAS:

GitHub. (2/09/2023). Recuperado de https://github.com/BuilderIO/mitosis

ANTLR (Consulta: 6/09/2023). Recuperado de https://www.antlr.org/

GitHub (20/09/2023). C Grammar. recuperado de https://github.com/antlr/grammars-v4/blob/master/c/C.g4

Web Components. (18/09/2023). Introducción Web Components recuperado de https://www.webcomponents.org/introduction

DSL Engineering (28/09/2023). Recuperado de https://voelter.de/dslbook/markusvoelter-dslengineering-1.0.pdf

Node (10/11/2023) recuperado de https://nodejs.org/docs/latest/api/path.html

Vue (23/09/2023). Recuperado de https://vuejs.org/guide/essentials/template-syntax.html

Interpreters (14/10/2023). Recuperado de https://craftinginterpreters.com/resolving-and-binding.html#a-resolver-class

React (26/10/2023). Recuperado de https://legacy.reactjs.org/docs/hello-world.html.

Princeton (09/09/2023). Recuperado de

https://www.cs.princeton.edu/courses/archive/spring20/cos320/LL1

ANEXO A

LISTA DE PRODUCCIONES

program: module EOF;

module: (uses | component | style | typeDef) module | ;

uses: Use usePath SemiColon module;

usePath: Identifier useSubModule;

useSubModule: Dot Identifier useSubModule | ;

typeDef: Type Identifier Assign OBrace typeDefBody CBrace;

typeDefBody: Identifier optional Colon varType SemiColon typeDefBody | ;

optional: Question | ;

style: Style Identifier OBrace styleClasses CBrace;

styleClasses: styleClass styleClasses | ;

styleClass: Dot Identifier styleCombine OBrace styleRules CBrace;

styleCombine: stlyeModifier Identifier styleCombine | ;

stlyeModifier: Dot | GreaterThan Dot;

styleRules: styleRule styleRules | ;

styleRule: (Identifier | Color) Colon styleRuleFollow SemiColon | styleClass;

styleRuleFollow: (HEX_COLOR | measure | Identifier) styleRuleFollow |;

measure: NumberValue measureUnit;

measureUnit: Px | Rem | Em | Mod |;

component: Component Identifier OBrace componentBody CBrace;

componentBody: (varDeclaration | propDeclaration | render | functionDeclaration) componentBody | ;

varDeclaration: varMutability Identifier Colon varType initValue SemiColon ;

varMutability: Var | Val;

varType: primitiveType array | Identifier array;

array: OBracnk CBracnk |;

primitiveType: Number | String | Boolean | Void | Color | Undefined;

propDeclaration: Prop Identifier Colon varType initValue SemiColon

initValue: Assign expression | ;

render: Render OParen renderFollow;

renderFollow: (Undefined | template) CParen SemiColon;

expression: assignmentExpression;

assignmentExpression: conditionalExpression assignmentFollow;

assignmentFollow: Assign assignmentExpression | ;

conditionalExpression: logicalOrExpression ternaryExpression;

ternaryExpression: Question expression Colon conditionalExpression | ;

logicalOrExpression: logicalAndExpression logicalOrFollow;

logicalOrFollow: Or logicalOrExpression | ;

logicalAndExpression: equalityExpression logicalAndFollow;

logicalAndFollow: And logicalAndExpression | ;

equalityExpression: relationalExpression equalityFollow;

equalityFollow: Equal equalityExpression | NotEqual equalityExpression | ;

relationalExpression: additiveExpression relationalFollow;

relationalFollow:

LessThan relationalExpression

| LessThanOrEqual relationalExpression

| GreaterThan relationalExpression

| GreaterThanOrEqual relationalExpression

| ;

additiveExpression: multiplicativeExpression additiveFollow;

additiveFollow: Plus additiveExpression | Minus additiveExpression | ;

multiplicativeExpression: castExpression multiplicativeFollow;

multiplicativeFollow: Star multiplicativeExpression | Slash multiplicativeExpression | Mod multiplicativeExpression | ;

castExpression: unaryExpression castFollow;

castFollow: As varType | ;

unaryExpression: postfixExpression | unaryOperator postfixExpression;

unaryOperator: Plus | Minus | Not;

postfixExpression: primaryExpression postfixFollow;

postfixFollow:

PlusPlus postfixFollow

| MinusMinus postfixFollow

| Dot Identifier postfixFollow

| OParen argExpList CParen postfixFollow

| OBracnk expression CBracnk postfixFollow

| ;

argExpList: expression argExpListFollow | ;

argExpListFollow: Comma argExpList | ;

primaryExpression: Identifier | constantExpression | OParen expression CParen | objectLiteral | arrayLiteral;

arrayLiteral: OBracnk argExpList CBracnk;

objectLiteral: OBrace objectLiteralBody CBrace;

objectLiteralBody: Identifier Colon expression objectLiteralBodyFollow |;

objectLiteralBodyFollow: Comma objectLiteralBody |;

constantExpression: NumberValue | StringLiteral | BoolValue | Undefined | HEX_COLOR;

template: LessThan Identifier attributes templateFollow | ;

templateFollow: (GreaterThan templateBody CloseTag Identifier GreaterThan template) | Slash GreaterThan template;

attributes: attribute attributes | ;

attribute: OBrace Identifier CBrace| directive| attributeBind;

directive: directiveName Assign attributeValue;

attributeBind: Identifier attributeBindFollow;

attributeBindFollow: Assign attributeValue | ;

directiveName: If | Else | Switch | Case | Template;

attributeValue: StringLiteral | OBrace expression CBrace;

templateBody: template | charData templateBody | OBrace expression CBrace templateBody

```
charData: ~('<'|'</'|'{')+;
functionDeclaration: Function Identifier OParen parameterList CParen returnType functionBody ;
returnType: Colon varType | ;
parameterList: parameter parameterListFollow | ;
parameter: Identifier Colon varType;
parameterListFollow: Comma parameterList | ;

functionBody: OBrace statementList CBrace;
statementList: statement statementList | ;
statement:
  returnStatement |
  assigmentStatement |
  varDeclaration |
  expressionStatement |
  ifStatement ;
returnStatement: Return result SemiColon;
result: expression | ;
assigmentStatement: Identifier Assign expression SemiColon ;
expressionStatement: expression SemiColon;
ifStatement: If OParen expression CParen ifBody elseStatement;
ifBody: OBrace statementList CBrace;
elseStatement: Else ifBody | ;
```

ANEXO B

SOFTWARE AUXILIAR

Extensión personalizada para Visual Studio Code:
https://github.com/ShulkMaster/crossbind-ext-vscode

Archivos JSON de pruebas
https://github.com/ShulkMaster/xbind-view/tree/master/public

Visualizador de gráficos:
https://github.com/ShulkMaster/xbind-view
https://shulkmaster.github.io/xbind-view

Generadores de pruebas
https://github.com/ShulkMaster/xbind2/tree/master/benchmark