

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO - CHILE



“DESARROLLO DE UN LENGUAJE DE PROGRAMACIÓN NO MANEJADO, FÁCIL DE APRENDER Y DE USAR”

ALONSO MANU ZAMORANO PONCE

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Xavier Bonnaire
Profesor Correferente: Horst von Brand

Diciembre - 2024

Agradecimientos

Quiero agradecer a la Kari y a mi familia por acompañarme y apoyarme durante todo el proceso de esta memoria. Y por motivarme y ayudarme a escribir.

Resumen

Resumen— En este trabajo se diseña y desarrolla un lenguaje de programación con el objetivo de ser fácil de aprender, de usar, y no tener un tope en cuanto a rendimiento. Para esto, se guía el diseño con Cognitive Dimensions Framework y se crea una implementación utilizando LLVM como backend. Se logra una implementación de una parte significativa del lenguaje diseñado. Junto con una biblioteca estándar mínima, una extensión para Visual Studio Code y un manual para el lenguaje.

Palabras Clave— Lenguaje de programación, Single Ownership, Value Semantics, LLVM

Abstract

Abstract—In the following work, a programming language is designed and implemented with the goal of being easy to use, easy to learn, and not having a performance ceiling. For this, the design is guided by the use of the Cognitive Dimensions Framework and an implementation is made using LLVM as backend. A significant part of the language designed is implemented. Along a minimal standard library, an extension for Visual Studio Code and a manual for the language.

Keywords— Programming languages, Single Ownership, Value Semantics, LLVM

Glosario

ABI: Application Binary Interface. Describe una interfaz que se usa para comunicarse entre procesos.

AST: Abstract Syntax Tree. Corresponde a una estructura de datos con forma de árbol usada para representar la sintaxis de los programas.

Backend: Corresponde a un componente en un compilador que su función es producir la salida final del compilador.

CPU: Central Processing Unit. Describe el procesador de un ordenador.

Frotend: Corresponde a un componente en un compilador que su función es analizar un programa sintáctica y semánticamente.

GCC: GNU Compiler Collection. Consiste en una colección de compiladores del proyecto GNU.

GDB: GNU Debugger. Consiste en un depurador portable del proyecto GNU.

LLD: Es un *linker* del proyecto LLVM que busca servir como reemplazo a los linkers que vienen con los sistemas.

LLVM: Corresponde a una colección de librerías utilizadas para el desarrollo de compiladores.

LLVM IR: Corresponde a LLVM Intermediate Representation. Es usada como etapa intermedia al compilar, en la cuál son aplicadas optimizaciones.

Runtime: Corresponde al sistema en tiempo de ejecución de un lenguaje de programación que corre cuando un programa es ejecutado. Esto incluye manejar la memoria, chequeos de seguridad, entre otros.

SIMD: Single Instruction, Multiple Data. Corresponde a aplicar una misma operación de forma paralela a múltiples escalares.

Target: Plataforma a la que puede ser compilado un programa. Esta depende del sistema operativo, de la arquitectura, entre otros.

vsce: Visual Studio Code Extensions. Corresponde a una aplicación de línea de comandos que sirve para publicar y manejar extensiones para Visual Studio Code.

Índice de contenidos

Resumen	II
Abstract	II
Glosario	III
Índice de figuras	VII
Índice de tablas	VII
Introducción	1
Capítulo 1: Definición del problema	3
1.1 Contexto	3
1.2 Situación actual	4
1.3 Objetivos	7
1.3.1 Objetivo general	7
1.3.2 Objetivos específicos	7
1.4 Alcance	7
Capítulo 2: Marco conceptual	8
2.1 Conceptos básicos	8
2.1.1 Arquitectura	8
2.1.2 Lenguaje de máquina	8
2.1.3 Assembly	8
2.1.4 Lenguaje de programación de alto nivel	9
2.1.5 Compilador	9
2.1.6 Cadena de compilación	10
2.1.7 Intérprete	11
2.1.8 Compilación Just-in-time (JIT)	11
2.1.9 Sistemas de tipos	11
2.1.10 Manejo de la memoria	13
2.2 Marco de trabajo	15
2.2.1 Cognitive Dimensions Framework	15
2.2.2 ¿Qué hace un lenguaje fácil de aprender?	17
2.2.3 ¿Qué hace un lenguaje fácil de usar?	21
2.2.4 Otras consideraciones	21
2.3 Estado del arte	22
2.3.1 C	22
2.3.2 C++	23
2.3.3 Rust	24
2.3.4 Zig	24
2.3.5 D	25

2.3.6	Julia	25
2.3.7	Swift	26
2.3.8	Java	26
2.3.9	Go	26
2.3.10	Propuesta	28
Capítulo 3: Propuesta de solución		30
3.1	Diseño	30
3.1.1	Sintaxis	31
3.1.2	Variables	32
3.1.3	Asignación	33
3.1.4	Estructuras de control	34
3.1.5	Tipos	34
3.1.6	Funciones	41
3.1.7	Paso de parámetros	42
3.1.8	Interfaces	45
3.1.9	Módulos	46
3.1.10	Single Ownership	47
3.2	Implementación	49
3.2.1	Tecnologías usadas	49
3.2.2	Estructura del proyecto	50
3.2.3	Arquitectura de la cadena de compilación	51
3.2.4	Lexer	52
3.2.5	Parser	53
3.2.6	Análisis semántico	54
3.2.7	Inferencia de tipos	57
3.2.8	Generación de código	58
3.2.9	Linking	60
3.2.10	ABI	60
3.2.11	Interfaz	62
3.2.12	Tests	65
3.3	Biblioteca estándar	66
3.4	Extensión	69
3.5	Manual	71
Capítulo 4: Validación de la solución		73
4.1	Objetivo 1: Manejo de memoria predecible en tiempo de compilación	73
4.2	Objetivo 2: Facilidad de aprender	75
4.2.1	Abstraction Gradient	75
4.2.2	Closeness of mapping	76
4.2.3	Consistency	76
4.2.4	Error-proneness	76
4.2.5	Hidden dependencies	77
4.2.6	Premature commitment	77

4.2.7	Progressive evaluation	77
4.2.8	Role-expressiveness	77
4.2.9	Secondary notation	77
4.2.10	Viscosity	78
4.3	Objetivo 3: Facilidad de uso	78
4.3.1	Closeness of mapping	78
4.3.2	Consistency	78
4.3.3	Error-proneness	78
4.3.4	Hidden dependencies	79
4.3.5	Premature commitment	79
4.3.6	Progressive evaluation	79
4.3.7	Secondary notation	79
4.3.8	Role-expressiveness	79
4.3.9	Viscosity	80
4.4	Resumen análisis de dimensiones	81
4.5	Ejemplos en el lenguaje	82
4.5.1	Juego de Pong	82
4.5.2	Implementación de un arreglo dinámico	84
Capítulo 5: Conclusiones		87
5.1	Conclusiones generales	87
5.2	Cumplimiento de objetivos	87
5.3	Trabajo futuro	88
Anexos		90
5.1	Gramática de Diamond	90
5.2	Gramática para la extensión de Visual Studio Code	93
Referencias bibliográficas		98

Índice de figuras

1	Software sectors and their communities	4
2	Size of programming language communities in Q3 2022	5
3	Diagrama cadena de compilación	10
4	Error producido por el compilador de Elm	20
5	Arquitectura de la cadena de compilación	52
6	Ventana abierta usando raylib y nuestro lenguaje	61
7	Interfaz de línea de comandos del compilador	63
8	Error generado al reasignar variable inmutable	64
9	Error generado al haber indentación inesperada	64
10	Salida producida al correr los tests del compilador	66
11	<i>Syntax highlighting</i> en Visual Studio Code de nuestro lenguaje	70
12	Vista de nuestra extensión en el Marketplace de Visual Studio Code	71
13	Manual del lenguaje	72
14	Prueba de Single Ownership	74
15	Dependencia entre conceptos para el aprendizaje de nuestro lenguaje	75

Índice de tablas

1	Comparación entre lenguajes de programación y propuesta	28
2	Resumen de análisis de dimensiones	81

Introducción

Existen diferentes categorías de rendimiento que un lenguaje de programación puede alcanzar. Por mucho tiempo, C y C++ han sido los reyes indiscutidos en cuanto rendimiento. En la actualidad, han surgido alternativas como Rust y Zig. Mejoran la experiencia de usuario significativamente en comparación a los anteriores. Pero distan de ser lenguajes de programación fáciles de aprender y de usar.

Por otro lado, han surgido lenguajes como Swift, Julia, Go, que son más ergonómicos y, en general, rápidos. Pero son lenguajes manejados, esto quiere decir que cuentan con manejo de la memoria automático en tiempo de ejecución. Debido a esto, es más difícil seguir y controlar cómo se usa la memoria en un programa. Por esto, otorgan menos control que C y C++ para aprovechar los recursos del ordenador. E implica que a veces no se contaría con el control necesario para resolver el problema que se está trabajando.

Este es el problema que esta memoria busca abordar, la falta de un lenguaje de programación que sea fácil de aprender y de usar. Y que mantenga el rendimiento de un lenguaje como C. Es decir, que tenga su nivel de control y que cuente con *runtime overhead*¹ mínimo. De esta manera, que no cuente con un tope en cuanto a rendimiento.

Para esto, se diseña un nuevo lenguaje de programación llamado Diamond, desarrollamos una cadena de compilación² para este, luego una biblioteca estándar mínima, posteriormente una extensión para una plataforma de desarrollo popular y por último, un manual para este.

Si bien que hace un lenguaje fácil de usar varía de persona a persona, para esta memoria, lo consideraremos como limitar la fricción innecesaria al programar y dificultar cometer errores. Se utiliza el marco de trabajo Cognitive Dimensions Framework [Green y Petre, 1996] a lo largo de la memoria para guiar y validar el diseño.

Por otro lado, para la implementación se utilizó C++ y LLVM como *backend*. Componente encargado de la optimización y generación de código, este es utilizado por los compiladores de Clang, Swift, Rust y Julia.

¹*runtime overhead* significa cualquier lógica adicional necesaria para la ejecución en un lenguaje de programación en particular

²Una cadena de compilación implica un compilador, generación de código y un *linker* para poder producir ejecutables

La estructura de este documento es la siguiente:

- *Capítulo 1, Definición del problema:* Se contextualiza el problema y se plantean los objetivos a resolver.
- *Capítulo 2, Marco conceptual:* Se definen conceptos usados en la memoria, referentes y estado del arte.
- *Capítulo 3, Propuesta de solución:* Se muestran las soluciones a los distintos objetivos.
- *Capítulo 4, Validación de la solución:* Se valida el diseño del lenguaje.
- *Capítulo 5, Conclusiones:* Aquí se expone lo aprendido, se chequea si los objetivos propuestos se cumplieron y se plantea trabajo futuro.

Capítulo 1

Definición del problema

1.1. Contexto

El término lenguaje manejado fue introducido por Microsoft [Microsoft, 2023] para diferenciar entre código que es manejado por una *runtime* de código que corre directamente sobre el procesador sin soporte de un sistema en tiempo de ejecución.

Esto quiere decir que los lenguajes *no manejados* no son interpretados ni corren en una máquina virtual. Y el programador se encarga de la gestión de memoria o lo hace el compilador en tiempo de compilación. Ejemplos de lenguajes no manejados son C, C++ y Rust.

Por otro lado, los lenguajes manejados requieren alguna forma de administrar la memoria en tiempo de ejecución. Y pueden ser interpretados, correr en una máquina virtual o ser compilados a código de máquina. Entre ellos, hay lenguajes dinámicos como Python, Ruby y Javascript y lenguajes con tipificado estático como Go, Java y Swift. Un lenguaje en esta categoría puede ser rápido, como es el caso de Go, pero el hecho de contar con un sistema en tiempo de ejecución implica que el uso de la memoria es poco predecible. Y que no se tiene control total de cómo se comportará el sistema de tiempo de ejecución. Lo que puede hacerlo difícil de usar para sistemas donde se requiere control estricto.

Es por estas razones que se prefieren lenguajes no manejados para sistemas donde el rendimiento es crítico. Otra razón para usar lenguajes no manejados es que tienen la ventaja de que usan menos memoria y energía [Pereira *et al.*, 2017]. Y además, los lenguajes no manejados son más fáciles de usar en combinación con otros lenguajes de programación, esto debido a que no hay que preocuparse por incompatibilidades en el manejo de memoria. Por lo tanto, si se quiere programar una biblioteca para ser usada en múltiples lenguajes, lo más fácil sería utilizar un lenguaje que sea no manejado.

Por otro lado, los lenguajes manejados implican una menor carga cognitiva por parte del programador. Pues no es necesario pensar en el manejo de la memoria. Por lo mismo, generalmente tienen una menor curva de aprendizaje. Y también es común que ofrezcan más garantías de seguridad. Además, en general, ofrecen menor fricción al momento de programar. Por lo que son más adecuados para prototipar. Ofrecen menor tiempo de iteración y es más fácil obtener una evaluación parcial del programa. Esto implica que generalmente son más fáciles de aprender [Gugerty y Olson, 1986]. Por último, normalmente incluyen más conveniencias de manera predeterminada, lo que los hace más fáciles de *debuggear*.

Es importante tener en cuenta que en muchos de los lenguajes manejados ofrecen opciones manejadas para el manejo de memoria. Por ejemplo, en C++ existen *shared pointers* y en Rust existe el tipo *Arc*. Estos permiten manejar la memoria utilizando *reference counting* en

C++ y Rust respectivamente.

Al mismo tiempo, hay lenguajes manejados que permiten escapar del contexto manejado. Un ejemplo de esto es el lenguaje de programación Julia. Si bien es dinámico, al especificar los tipos que tienen los argumentos en una función, Julia puede emitir código nativo sin información de tipos en tiempo de ejecución. Otro ejemplo, en el lenguaje de programación Nim, se puede elegir entre diversas opciones para el manejo de memoria. Siendo el manejo manual de la memoria una de las opciones.

1.2. Situación actual

Según una encuesta realizada por SlashData en 2019 [SlashData, 2019], los dominios de programación más populares son desarrollo web con 16,9 millones de programadores. Seguidos por servicios de *backend* con 13,6 millones. Luego, desarrollo de apps móviles con 13,1, aplicaciones de escritorio con 12,3, ML, AI y ciencia de datos con 12,2, *Internet of things* con 9,3, desarrollo de videojuegos con 8,8 y desarrollo de AR/VR con 5,8 millones de desarrolladores (Ver figura 1).

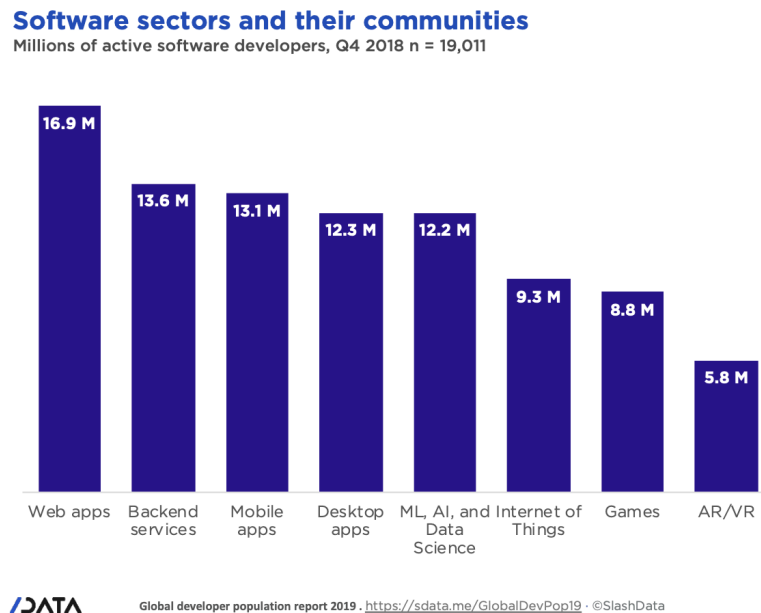


Figura 1: Software sectors and their communities
Fuente: SlashData

Complementando lo anterior, con una encuesta realizada por la misma empresa en 2022 [SlashData, 2022]. Los 5 lenguajes de programación con más usuarios activos son JavaScrip-

t/TypeScript, seguidos por Python, luego Java, después C/C++, y por último, C#. Esto podemos verlo a la figura 2.

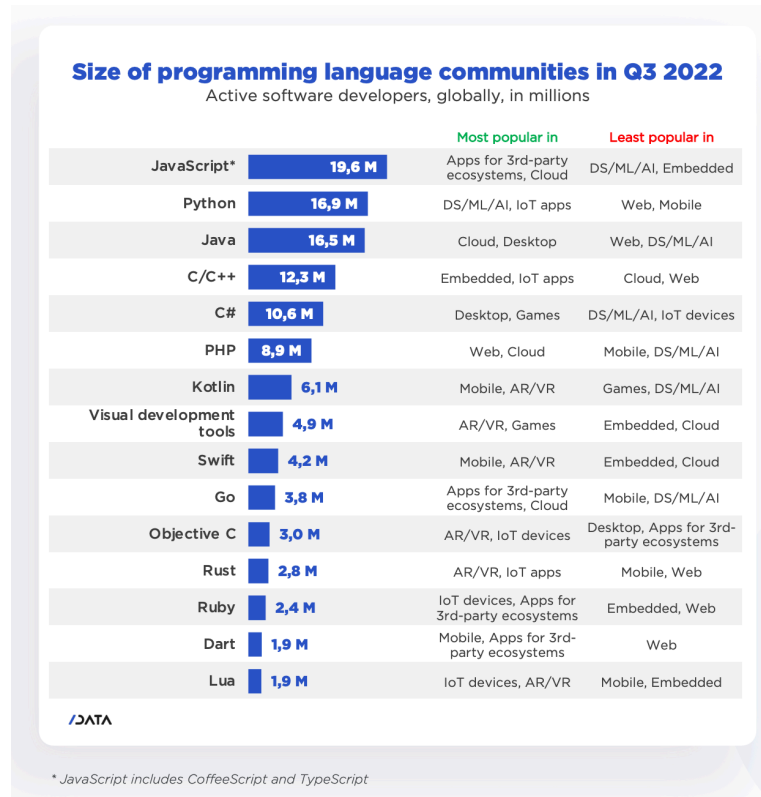


Figura 2: Size of programming language communities in Q3 2022

Fuente: SlashData

Los dominios de programación más populares son el desarrollo de aplicaciones web y servicios de *backend*. Seguido por aplicaciones móviles y de escritorio. En estos dominios generalmente el rendimiento no es crítico, o no es necesario un control estricto sobre el uso de memoria, por lo que se tiende a usar lenguajes manejados por su conveniencia. Generalmente, el costo de pagar más horas de programador es mayor que el de añadir o mejorar el hardware. [Atwood, 2008]

Por otro lado, hay dominios como ML, AI, videojuegos y AR/VR en los que es intrínseco tener el control sobre cómo se usa la memoria. Por lo que es común utilizar lenguajes no manejados.

Los lenguajes manejados más populares son JavaScript, Python, Java, C#, PHP, Kotlin, Swift, Go, Objective C, Ruby, Dart y Lua.

Los lenguajes no manejados más populares son C, C++ y Rust.

En tiempos recientes, han surgido nuevos lenguajes de programación no manejados, tales

como Rust, Zig, Odin, Austral y Hylo. Estos ofrecen mejoras sustanciales en cuanto a lenguajes como C. Rust ofrece manejo de la memoria automático en tiempo de compilación, junto con *developer tooling* de primera calidad y *memory safety* en tiempo de compilación. Zig repiensa C desde cero, produciendo un lenguaje que busca optimizar la legibilidad y el control. Además de las bondades propias del lenguaje, este es completamente interoperable con C y puede *cross compile*³ proyectos escritos en C, C++ y Zig. Odin también busca ser alternativa a C, diseñado para ser simple y agradable de usar. Austral busca responder a qué pasaría si hiciera un lenguaje fácil de aprender con tipos lineales⁴ como foco principal. Por último, Hylo busca explorar el paradigma *mutable value semantics*. [Teodorescu, 2023]

Sin embargo, estos lenguajes aún presentan desafíos significativos en cuanto a su curva de aprendizaje. Exponen al usuario a una complejidad inicial significativa. Y su sintaxis está en mayor o menor medida inspirada en C, la cual se ha mostrado que puede resultar difícil para programadores novatos. [Stefik y Siebert, 2013]

Al mismo tiempo, han surgido muchos lenguajes de programación manejados. Estos muy cercanos en rendimiento a lenguajes no manejados. Entre ellos, Swift, Go, Julia, Nim y Crystal. Pero todos usan alguna forma de recolección por omisión. Pueden ser muy eficientes con los recursos, pero la libertad que dan al programador en su usabilidad implica que la utilización de los recursos es más difícil de predecir y manejar.

Es por esto que, a la hora de empezar un nuevo proyecto, se debe elegir entre el rendimiento y predictibilidad de los lenguajes no manejados. Y la conveniencia de los lenguajes manejados.

Debido a esto, en muchos proyectos se usan múltiples lenguajes, al menos uno para poder aprovechar los recursos del ordenador. Y otro por usabilidad y productividad. Este patrón se usa extensivamente en el desarrollo de videojuegos. También se usa extensivamente en el ecosistema de Python para la programación de bibliotecas como Numpy, TensorFlow, PyTorch, entre otras. Esto es conocido en la ciencia de datos como "*The Two-Language problem*" [Storopoli et al., 2021] y tanto Julia como Mojo buscan solucionarlo. Igualmente, es habitual que, por requerimientos de rendimiento, alguna parte de una aplicación esté escrita en un lenguaje no manejado. Por ejemplo, en Figma, una aplicación web colaborativa para el diseño de interfaces, está escrita en C++ y TypeScript. Se usa TypeScript para la interfaz y C++ para las partes intensivas de la aplicación [Liaw, 2022].

Sin embargo, usar múltiples lenguajes de programación en un sistema tiene sus costos. Estos sistemas son más difíciles de entender, más difíciles de programar y más difíciles de mantener. El ciclo de vida de estos sistemas es más complejo y costoso. [Neitsch et al., 2012]

³Esta es la habilidad de un compilador para crear ejecutables para una plataforma distinta a en la cual se está desarrollando.

⁴Tipos lineales[Wadler, 1990] permiten tener manejo de la memoria automática en tiempo de compilación. Al mismo tiempo, modelar recursos como un *file handler* de tal manera que no pueda ser mal usado. Por ejemplo, si a uno se le olvidara cerrar el *file handler* el compilador podría avisarte en tiempo de compilación.[Borretti, 2023]

[Mayer *et al.*, 2017] [Abidi *et al.*, 2021] [Li *et al.*, 2022]

Por esto se propone la creación de un lenguaje de programación no manejado, que sea fácil de aprender y de usar. Se propone la creación de un lenguaje que cuente con la mayor cantidad posible de los beneficios de los lenguajes manejados. Pero que cuente con los beneficios de los lenguajes no manejados en rendimiento y gestión de recursos.

1.3. Objetivos

1.3.1. Objetivo general

Diseñar y desarrollar un lenguaje fácil de aprender y de usar, de uso general, que no tenga un tope en cuanto a rendimiento.

1.3.2. Objetivos específicos

1. Definir elementos con los que deba contar el lenguaje de programación para que sea fácil de aprender y de usar de uso general, que no tenga un tope en cuanto rendimiento.
2. Desarrollar una cadena de compilación completa para el lenguaje con una interfaz inspirada en el compilador de Elm⁵.
3. Implementar una versión mínima de la biblioteca estándar.
4. Proporcionar una extensión para una plataforma de desarrollo popular.
5. Facilitar la divulgación del lenguaje mediante un manual para este.

1.4. Alcance

Desarrollar un lenguaje de programación en general toma años, ni hablar de la versión 1.0. Para esta memoria, el objetivo será diseñar los aspectos más fundamentales del lenguaje e implementar las características más desafiantes para probar que pueden ser implementadas.

⁵Elm es un lenguaje de programación creado para desarrollar interfaces. Actualmente es uno de los lenguajes de programación con mejores errores [Czaplicki, 2015] y experiencia de usuario.

Capítulo 2

Marco conceptual

2.1. Conceptos básicos

2.1.1. Arquitectura

Una arquitectura corresponde a un modelo abstracto que describe instrucciones que procesadores que siguen la arquitectura pueden ejecutar. Adicionalmente, existen subfamilias dentro de una misma arquitectura, con instrucciones más específicas. Esto haría que código compilado para un procesador en específico fuera más rápido que uno compilado para una arquitectura en general. Aunque tendría el costo de que el ejecutable final sería menos portable.

Actualmente, las arquitecturas más populares usadas en computadores personales corresponden a x86-64 y AArch64. Por mucho tiempo, x86-64 fue la más popular, usada en Windows y macOS, pero hace poco Apple hizo el cambio a AArch64. Una arquitectura que hasta el momento ha sido usada mayormente en móviles.

2.1.2. Lenguaje de máquina

Un programa en lenguaje de máquina corresponde a una secuencia de números en binario para un procesador dado. Es código que un procesador puede ejecutar directamente.

2.1.3. Assembly

Debido a la dificultad de programar directamente en lenguaje de máquina, se creó el lenguaje *assembly*. Existe una fuerte correspondencia entre el lenguaje de máquina y el *assembly*. Ambos son específicos a una arquitectura o procesador dado, pero al programar en *assembly* se programa en texto y es más legible ya que no es necesario programar usando solo números.

Sin embargo, como es específico a cada arquitectura o procesador, el código no sería portable.

2.1.4. Lenguaje de programación de alto nivel

Los lenguajes de programación de alto nivel son en lo que generalmente se piensa cuando se habla de lenguajes de programación.

Este nombre debido a que son independientes de la arquitectura o procesador en el que serán ejecutados. Debido a esto, programas escritos en ellos son portables, y pueden correr en *hardware* muy variado.

La mayoría de los lenguajes de programación son en base a texto, aunque también existen lenguajes de programación visuales como Scratch o basados en nodos como Blueprints de Unreal.

Estos lenguajes luego son traducidos, de alguna manera, a lenguaje de máquina para ser ejecutados.

2.1.5. Compilador

Un compilador es un programa que toma un programa en un lenguaje de alto nivel y lo traduce a un lenguaje de más bajo nivel.

Por lo común, hay dos opciones, o se traduce a lenguaje de *assembly* para un procesador físico, o se traduce para un lenguaje de máquina para un procesador virtual.

En general, las tareas que debe cumplir un compilador son:

1. *Lexing/Tokenizing/Scanning*: Consiste en transformar una serie de caracteres, en una serie de *tokens*. Se agrupan los caracteres que son parte de una misma unidad de significado y se eliminan los caracteres que no tienen carga semántica. De esta manera, en la siguiente etapa se trabaja con identificadores, *keywords*, puntuación, *strings*, etc. En vez de caracteres sueltos.
2. *Parsing*: Esto depende de la implementación del compilador, pero lo más común consiste en transformar una serie de *tokens* en un *Abstract Syntax Tree*, la cual es una representación del programa de entrada que hace más fácil los análisis posteriores. En esta etapa se verifica que la sintaxis del programa de entrada sea válida.
3. *Análisis semántico*: En esta etapa se chequea que las variables y funciones referenciadas existan. Se hace también el chequeo de tipos, es decir, asegurarse que las operaciones que se realizan son válidas. Dependiendo del lenguaje, puede haber otros análisis también, como por ejemplo la inferencia de tipos o chequear la vida de las variables.

4. *Generación de representaciones intermedias y optimización*: Esta corresponde a una etapa o etapas opcionales que corresponden a la generación de representaciones intermedias con el objetivo de realizar optimizaciones sobre estas.
5. *Generación de código*: En esta última etapa se toma la representación que se tiene y se traduce a *assembly* o *bytecode*. También pueden ser a otros lenguajes como C o JavaScript. En esta etapa se genera el *output* final del compilador.

2.1.6. Cadena de compilación

Una cadena de compilación corresponde a todo lo que hace un compilador, más generar código objeto y hacer *linking*. Es decir, las etapas necesarias para poder generar un ejecutable. Podemos ver un diagrama de la cadena de compilación en la figura 3.

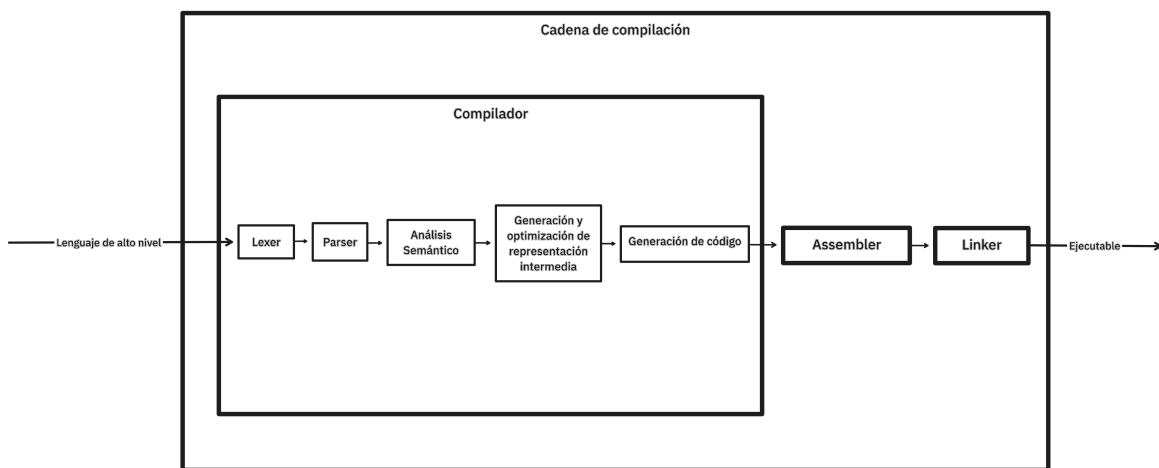


Figura 3: Diagrama cadena de compilación
Fuente: Elaboración propia

Podemos ver que incluye las etapas que ya vimos en el compilador, además de un *assembler* para convertir el código *assembly* a código objeto, y luego un *linker* para convertir el código objeto en el ejecutable final.

Por último, en el contexto de compiladores, se llama *frontend* a la parte del compilador que es independiente de la plataforma donde se ejecutará el *output*. Es decir, *lexing*, *parsing* y *análisis semántico*. Por otro lado, se conoce como *backend* la parte del compilador que es específica a una plataforma en particular. En general, se busca que un *frontend* pueda tener varios *backend*, es decir, que un lenguaje pueda tener como *target* varias plataformas u opciones para generar código. Y al mismo tiempo que un *backend* pueda ser reutilizado por diversos *frontend*, esto sería, por ejemplo, cuando más de un lenguaje comparten *backend*, como es el caso de Rust, Swift y Clang.

2.1.7. Intérprete

Un compilador traduce un programa en su completitud a un lenguaje de más bajo nivel. Por el contrario, un intérprete es un programa que va traduciendo y ejecutando instrucción por instrucción un programa. Crea una representación del programa y comienza a ejecutarla de inmediato. En ningún caso se genera lenguaje de máquina o *assembly*.

Gracias a esto, es necesario esperar menos para empezar a ejecutar un programa, pero implica que la ejecución de cada instrucción tendrá un *overhead* adicional. Una buena forma de conceptualizar la diferencia entre compilar e interpretar es como la diferencia entre traducir un texto completo y luego leerlo, a ir leyéndolo y traduciéndolo al mismo tiempo.

2.1.8. Compilación Just-in-time (JIT)

Es una técnica usada que consiste en compilar secciones de un programa a código de máquina en tiempo de ejecución para mejorar el rendimiento. Esto se hace cuando el sistema en tiempo de ejecución⁶ de un lenguaje establece que una sección de código ha sido ejecutada muchas veces y que probablemente será ejecutada muchas veces más, por lo que valdría pagar el costo de traducirla a código de máquina.

2.1.9. Sistemas de tipos

Un sistema de tipos corresponde a un sistema que asigna o chequea el tipo de cada expresión dentro de un programa. Este sirve para evitar la presencia de ciertos errores al programar. Qué tipo de errores evita y qué cosas permite expresar depende del sistema de tipos en particular. Al mismo tiempo, si tiene o no y cómo funciona la inferencia de tipos depende de este.

Si el sistema actúa en tiempo de compilación, se dice que es *estático*. Por el contrario, si actúa en tiempo de ejecución, se dice que el sistema de tipos es *dinámico*.

Un sistema de tipos se dice que soporta tipos *algebraicos* si es que soporta *product types* y *sum types*. En general, la mayoría de los lenguajes soportan *product types*, pero no todos soportan *sum types*. En la mayoría de los lenguajes, *product types* corresponde a estructuras o tuplas. Permiten decir que este tipo está compuesto por *esto*, *esto...* y *esto*. Por otro lado, *sum types* corresponde al concepto de unión. Permiten decir este tipo corresponde a *esto*, *esto...* o *esto*.

⁶El sistema en tiempo de ejecución de un lenguaje corresponde a la máquina virtual del lenguaje o al sistema encargado de su ejecución

2.1.9.1. Sistema de tipos Hindley-Milner

Es un sistema de tipos clásico. Fue descrito primero por J. Roger Hindley [1969] y luego redescubierto por Robin Milner [Milner, 1978].

Es conocido por permitir inferencia de tipos completa y que infiere el tipo más general posible para cada expresión. Swift y Rust usan inferencia de tipos inspirados en Hindley-Milner. Aunque aún así requieren anotaciones de tipos para las funciones. Haskell y lenguajes de la familia ML como OCaml y F# son conocidos por contar con este sistema de tipos.

La ventaja de este sistema de tipos es que hace no que sea necesario escribir las anotaciones de tipos, y pueden inferir los tipos de las expresiones de acuerdo a como son usadas. Por ejemplo, en el siguiente código:

```
a = 20
b = 3.5
print(a + b)
```

Inferiría correctamente que la variable `a` debe ser tipo flotante. Al mismo tiempo, si se tuviera el código:

```
a = []
a.append(1)
a.append(2)
a.append(3)
```

Se inferiría correctamente que la variable `a` debe ser una lista de enteros.

Por otro lado, las funciones pueden ser polimórficas. Es decir, funcionar con múltiples tipos, pero manteniendo la seguridad de tipos.

Cabe notar que esto es distinto a *templates* como es el caso de un lenguaje como C++. En C++:

```
template <class T>
T add(T a, T b) {
    return a + b;
}
```

El código anterior compilaría sin problemas y al momento de instanciar la función, se vería si `T` soporta el operador `+`. Por el contrario, en un lenguaje con un sistema Hindley-Milner sería necesario especificar que el `T` soporta el operador `+`. De otra forma, lo único que podría hacer sería retornar `T`, pues no sabría qué operaciones puedo realizar con él. La ventaja de esto es que no sería necesario llamar a una función, es decir utilizarla, para saber si es correcta a nivel de tipos. Y no sería necesario volver a analizar una función cada vez que es usada con tipos distintos.

Entre las desventajas que se tiene es que puede hacer más difícil razonar localmente (si es que no se tienen anotaciones de tipos), y en general es más costoso que sistemas de tipos más simples. Otra desventaja es que las funciones que no tienen anotaciones de tipos no pueden ser analizadas en paralelo.

Swift y Rust usan variaciones de este sistema de tipos. Siendo una de las principales diferencias que requieren anotaciones de tipos para las funciones en el ámbito global.

2.1.10. Manejo de la memoria

Como un lenguaje maneja la memoria, determina en gran medida sus características de rendimiento. Determina qué tan rápido puede ser el lenguaje y cuánto control tendrá el programador. Existen muchas maneras de manejar la memoria, a continuación se presentan las técnicas más comunes.

2.1.10.1. Manejo manual de la memoria

Queda bajo responsabilidad del programador la gestión de la memoria. Permite un gran control, pero puede hacer fácil cometer errores. Los errores pueden ser de *fugas de memoria* o de problemas de *memory safety*.

Una *fuga de memoria* sucede cuando no se libera memoria que ya no se está usando. Esto generalmente no es un problema para programas que corren por poco tiempo, pues al terminar el sistema operativo se encargará de liberar la memoria. Pero sí puede ser un problema para software que se mantiene corriendo indeterminadamente, terminando por agotar los recursos del sistema en el peor de los casos.

Problemas de *memory safety* ocurren cuando se usa mal memoria. Pueden ocurrir cuando se libera una sección de la memoria múltiples veces o se utiliza un puntero a una sección de la memoria que ya fue liberada, entre otros. Esto puede generar problemas difíciles de *debuggear* en el mejor de los casos, y vulnerabilidades en el peor.

Una de las ventajas de la memoria manual es que no tiene ningún *overhead* adicional en tiempo de ejecución. La otra gran ventaja es el control que permite al programar. Esta permite usar estrategias hechas a medida para el problema a resolver. Como solo pedir memoria dinámica al iniciar el programa o usar *allocators* especializados para gestionar la memoria [Zig, 2024b] [Odin, 2024] [Trebino, 2016].

2.1.10.2. Recolección de basura

La *recolección de basura* es una técnica para gestionar la memoria, creada por John McCarthy para simplificar el manejo de memoria en Lisp [McCarthy, 1960]. Permite al programador no tener que pensar sobre el manejo de la memoria y asegura que no haya problemas de *memory safety* al manejar la memoria. Es importante notar que fugas de memoria aún pueden existir.

Usar recolección de basura hace más fácil la vida del programador, pero implica que un sistema va a requerir más recursos para correr, y puede ser más lento o sus características de rendimiento ser menos predecibles. Lo cual la puede hacer imposible de usar en sistemas *empotrados* o en sistemas con requerimientos de tiempo real.

2.1.10.3. Reference counting

Reference counting es otra técnica para gestionar la memoria de manera automática. Para cada recurso de memoria asignado se tiene un registro de la cantidad de referencias que se tienen a él. Cada vez que una referencia deja de existir, por ejemplo, cuando una variable llega al final de un ámbito, se disminuye este contador. Cuando se llega a cero, el recurso es eliminado de la memoria.

Tiene la ventaja de ser más predecible que otras formas de *recolección de basura* y no tener largas pausas impredecibles. Pero pueden no recolectar referencias circulares si es que no cuentan con alguna forma de ayuda. Por ejemplo, en Swift se permite marcar ciertas referencias como *weak*. Por otro lado, en Python se tiene un *cycle collector* para recolectar las referencias circulares. Referencias circulares corresponden cuando objetos se referencian entre sí de forma recursiva.

Al mismo tiempo, tiene un costo significativo en el rendimiento de un sistema.

2.1.10.4. RAII/Single-Ownership

RAII (*resource acquisition is initialization*) o *Single-Ownership* es una forma de manejar la memoria usada por C++ y Rust. Consiste en asociar la asignación y liberación de la memoria de un objeto a su vida. E implica que en cada momento solo hay un responsable de un objeto. Si el ámbito de una variable termina, entonces los recursos asociados a la variable son liberados.

La mayor ventaja de *Single-Ownership* en comparación con *Recolección de basura* o *Reference counting* es que no incurre en un coste adicional en tiempo de ejecución. Otra ventaja es que hace claro cuando la memoria es asignada y liberada.

La mayor ventaja frente al manejo de memoria manual es que es más fácil de enseñar y hace más difícil cometer errores, o incluso imposible dependiendo del lenguaje.

Single Ownership es un compromiso entre el manejo de memoria manual y los enfoques automáticos para manejar la memoria en tiempo de ejecución.

2.2. Marco de trabajo

2.2.1. Cognitive Dimensions Framework

Cognitive Dimensions Framework es un marco de trabajo propuesto por Green y Petre [Green y Petre, 1996] para evaluar y comparar dispositivos interactivos y notaciones no interactivas. Como, por ejemplo, son los lenguajes de programación.

Las dimensiones que propone son las siguientes:

1. *Abstraction gradient*

Qué tanto un lenguaje favorece la abstracción. Hay lenguajes que no permiten abstracción, lenguajes que toleran la abstracción, y lenguajes que incentivan la abstracción. Un lenguaje de uso general por lo general tolera o incentiva la abstracción. Un ejemplo de lenguaje que tolera la abstracción es C, ya que este no la incentiva, pero la permite. Por otro lado, un ejemplo de lenguaje que incentiva la abstracción es Java.

La abstracción está estrechamente relacionada con la modularidad. Un lenguaje que no permite modularización es un lenguaje que no permite abstraer.

2. *Consistency*

Qué tanto puedo inferir de un lenguaje en base a lo que ya conozco de este.

3. *Closeness of mapping*

Qué tan bien un lenguaje se mapea al dominio del problema que estoy resolviendo. Cuantas más cosas hay que considerar al momento de programar no relacionadas con el dominio del problema que estoy intentando resolver, entonces menos bien se mapea la notación al problema.

4. *Diffuseness*

Qué tan terso o verboso es un lenguaje.

5. *Error-proneness*

Qué tanto un lenguaje facilita o ayuda a prevenir errores. Los errores pueden ser de carácter sintáctico o lógico. Por ejemplo, un lenguaje como Scratch buscaría disminuir los errores sintácticos al usar una interfaz para manipular bloques de código en vez

de usar texto. Por otro lado, el tipificado estático buscaría evitar errores en tiempo de ejecución al detectarlos en tiempo de compilación.

6. *Hard mental operations*

Esta dimensión está relacionada con que tanto el lenguaje requiere gran trabajo mental para entender y utilizar aspectos de la notación. Por ejemplo, si una notación es muy tersa, probablemente requerirá una mayor carga cognitiva utilizarla.

7. *Hidden dependencies*

Qué tanto un lenguaje permite que hayan dependencias escondidas entre componentes. Un lenguaje con menos *hidden dependencies* más facilita el razonamiento local. Y, por consiguiente, menos contexto es necesario tener en la mente al momento de programar.

8. *Premature commitment*

Qué tanto un lenguaje fuerza a tomar decisiones antes de contar con la información necesaria para hacerlo. O que tanto fuerza a pensar en el futuro.

9. *Progressive evaluation*

Qué tanto un lenguaje facilita la evaluación parcial de un programa. Está relacionado con qué tan rápido un lenguaje permite obtener *feedback*.

10. *Role-expressiveness*

Qué tan legible es un lenguaje. Qué tan fácil o difícil es reconocer las distintas partes o elementos de un programa o sistema. Esta dimensión está relacionada con que tan bien puedo responder a la pregunta: ¿Y para qué es que sirve esta notación?

11. *Secondary notation*

Esta dimensión corresponde a aspectos del lenguaje que permiten comunicar información adicional, que no son parte de la definición formal del lenguaje.

12. *Viscosity*

Qué tan fácil es hacer cambios en un lenguaje. Un lenguaje dinámico, por ejemplo, tendría menos viscosidad, pues hacer cambios requiere menos esfuerzo.

13. *Visibility*

Qué tan fácil es obtener la información necesaria para editar un segmento del programa. También puede ser entendido como cuántos pasos son requeridos para encontrar la información necesaria para hacer los cambios. Esta sección depende sobre todo del editor y del *tooling* usado al programar. Aunque también depende de cómo está programado un proyecto dado.

Ha sido usado en el pasado para evaluar lenguajes de programación y se ha demostrado su utilidad. [Clarke, 2001]

Por mucho que nos gustaría que estas dimensiones fueran ortogonales, inevitablemente interactúan. Por ejemplo, dependiendo de la tersidad del lenguaje, este va a ser más o menos legible.

Una limitación de estas dimensiones es que permiten evaluar que también un lenguaje permite modelar y expresar ideas de un dominio en particular. Pero no toma en cuenta, por ejemplo, el rendimiento de un lenguaje.

2.2.2. ¿Qué hace un lenguaje fácil de aprender?

A continuación revisaremos las dimensiones relevantes a la hora de aprender un lenguaje de programación y dónde este debiera situarse en esta dimensión para ser fácil de aprender.

2.2.2.1. Abstraction gradient

Un lenguaje es más fácil de aprender si cuenta con un menor número de abstracciones iniciales. Por ejemplo, para escribir un hola mundo en Python, el primer programa que generalmente escribe un lenguaje, se requiere entender menos conceptos que en Java. En Java, para entender realmente lo que está pasando, es necesario además entender clases, métodos, *static*, entre otros. [Pellet *et al.*, 2019]

En general, un lenguaje de uso general requiere un gran número de abstracciones. Pero un lenguaje es más fácil de aprender si no es necesario aprender todas las abstracciones desde un principio. Cuanto más incremental se pueden aprender los distintos elementos de un lenguaje mejor.[Scaife y Taylor, 1990]

2.2.2.2. Closeness of mapping

Cuanto mejor se mapean las abstracciones que ofrece un lenguaje a un problema dado, más fácil de aprender es el lenguaje. Esto implica que hay menos esfuerzo en traducir las ideas del problema al lenguaje.

Por ejemplo, si estoy trabajando en un problema relacionado con listas, es mucho más fácil mapear el problema a Python que a C, dado que Python incluye el concepto de listas de manera predeterminada.

Esta dimensión depende totalmente del problema y de cómo el usuario quiera resolver el problema. Pero hay formas de mejorar en esta dimensión para el caso general.

Una forma fácil de mejorar en esta dimensión es contar con un gran número de bibliotecas a mano, como es en el caso de Python o Go. De esta manera, lo más probable es que el usuario tenga las herramientas para resolver un problema dado en vez de tener que programarlas él.

Por otro lado, si se está trabajando en un problema que requiere de muchas operaciones matemáticas, en general contar con *operator overloading* ayudaría a mejorar el *closeness of mapping*.

2.2.2.3. Consistency

Un lenguaje es más fácil de aprender si es consistente. Cuantas más cosas puedo inferir a partir de las reglas que ya conozco, mejor.

Un ejemplo de falta de consistencia sería, por ejemplo, en JavaScript, en que es difícil inferir qué va a hacer el operador + dependiendo de los parámetros:

```
"5" + 3 // Se evalúa a "53"  
"5" - 3 // Se evalúa a 2  
"5" + "3" // Se evalúa a "53"  
"5" - "3" // Se evalúa a 2
```

Un ejemplo de consistencia sería en Python en que todas las variables son objetos. Esto simplifica el modelo mental y hace fácil inferir a priori que todas las cosas poseen atributos y métodos.

Es importante tener en cuenta que un lenguaje se puede beneficiar de reducir la consistencia en favor de aumentar la legibilidad en partes claves del lenguaje. Usar representaciones distintas para conceptos distintos ayuda a hacer más fácil recordar y reconocer estos conceptos.

2.2.2.4. Error-proneness

Un lenguaje cuanto más difícil hace cometer errores más fácil es de aprender. Idealmente, el lenguaje debería ayudar a prevenir la mayor cantidad de errores posible. Aunque hay que tener en cuenta que a menudo, esfuerzos para reducir errores implican retrocesos en otras dimensiones. Por ejemplo, un sistema de tipos estático ayuda enormemente a la hora de reducir errores, pero al mismo tiempo, puede hacer los programas más difíciles de cambiar y también forzar a tomar decisiones prematuramente.

Lenguajes que hacen bien en esta dimensión, en general, son lenguajes con sistemas de tipos estáticos y fuertes, que chequean muchas cosas. Esto se ve en lenguajes como Rust, Swift,

Haskell, entre otros.

2.2.2.5. Hidden dependencies

Esta dimensión hace más sentido en cuanto a facilidad de uso. Pues hacerlo mejor en esta dimensión ayuda a hacer el código más mantenible. Pero ser explícito en las dependencias también puede hacer un lenguaje más fácil de aprender, sobre todo en cuanto al uso de sus bibliotecas. Por ejemplo, puede ser más fácil saber cómo usar una función si es qué se tiene claro que tipos recibe la función.

2.2.2.6. Premature commitment

Un lenguaje es más fácil de aprender si no fuerza al usuario a tomar decisiones para las cuales todavía no se cuenta con la información necesaria. Por lo que debiera proveer un bajo nivel de *premature commitment*.

Si bien muchas veces las decisiones que un lenguaje puede forzar a tomar al usuario no son innecesarias como tal, tienen su razón de ser, aumentan considerablemente la fricción al momento de aprender. Un ejemplo de esto es, por ejemplo, la gran cantidad de tipos para *strings* con la que cuenta Rust.

2.2.2.7. Progressive evaluation

Un lenguaje es más fácil de aprender cuando más rápidamente ofrece retroalimentación al usuario. Esto implica que, en general, los lenguajes compilados son más difíciles de aprender debido a que el ciclo de iteración es más mayor en general. Eso sí, en la actualidad esto es aliviado por la retroalimentación que puede ofrecer un editor al momento de programar gracias a un servidor de lenguaje u otra forma de *tooling* de editor.

Esta dimensión también está ligada a la explorabilidad de un lenguaje. La explorabilidad consiste en qué tan fácil es aprender un sistema sin tener que recurrir a documentación externa. Mayor explorabilidad puede reducir la fricción al momento de aprender o reaprender un lenguaje. Esta se puede mostrar en la forma de una *REPL* (*Read-Eval-Print-Loop*) o un buen servidor de lenguaje.

2.2.2.8. Role-expressiveness

Un lenguaje es más fácil de aprender si es legible. Es difícil definir qué hace un lenguaje más legible que otro. Dado que esto es subjetivo y depende de muchos factores, entre ellos la fa-

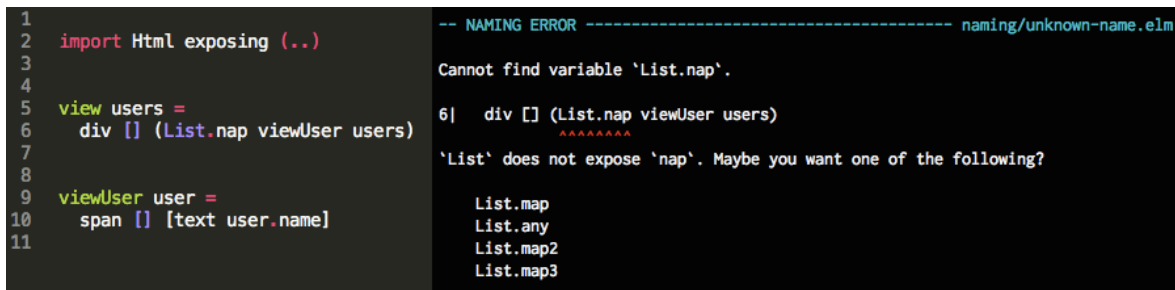
miliaridad. Esfuerzos se han hecho por llegar a resultados concretos [Stefik y Siebert, 2013], pero es difícil de comparar objetivamente.

2.2.2.9. Secondary notation

Existen elementos que permiten comunicar información adicional al momento de leer código, que no son parte de la definición formal del lenguaje. Como, por ejemplo, las convenciones que son usadas al momento de programar.

Otro canal de comunicación entre el lenguaje y el programador consiste en la interfaz en la línea de comandos del lenguaje. Y los errores que este produce. Buenos mensajes de error hacen una gran diferencia al momento de aprender un lenguaje de programación. Entre los lenguajes más aclamados por los errores que producen se encuentran Elm y, más recientemente, Rust, inspirado por el anterior.

En la figura 4 podemos ver un ejemplo de un error producido por el compilador de Elm. Lo que sería un buen ejemplo de usar *secondary notation* de una manera eficaz.



```
1 import Html exposing (..)
2
3
4
5 view users =
6   div [] (List.nap viewUser users)
7
8
9 viewUser user =
10   span [] [text user.name]
11
```

```
-- NAMING ERROR ----- naming/unknown-name.elm
Cannot find variable `List.nap`.

6|   div [] (List.nap viewUser users)
   ~~~~~
`List` does not expose `nap`. Maybe you want one of the following?

List.map
List.any
List.map2
List.map3
```

Figura 4: Error producido por el compilador de Elm
Fuente: Elm

2.2.2.10. Viscosity

Esta dimensión refiere a que tan fácil es hacer cambios. Un lenguaje es más fácil de aprender si tiene un bajo nivel de viscosidad. Esto es vital en una etapa de aprendizaje en que es requerido probar y evaluar conceptos.

Un lenguaje con baja viscosidad sería Python por ejemplo. Por otro lado, un lenguaje con alta viscosidad podría ser Java, debido a lo verbosa que es su sintaxis.

2.2.3. ¿Qué hace un lenguaje fácil de usar?

La mayoría de los aspectos que hacen un lenguaje fácil de aprender también lo hacen más fácil de usar. Aunque hay algunos aspectos que no constituyen parte de las características que hacen de un lenguaje fácil de aprender.

Un ejemplo de esto es la mantenibilidad. La mantenibilidad puede ser entendida como qué tan fácil es hacer cambios en un sistema y que este siga funcionando correctamente. Esta dimensión es producto de qué tan fácil el lenguaje permite razonar localmente (*Hidden dependencies*) y qué tanto ayuda a evitar errores (*Error-proneness*). El mayor problema es que, en general, las características que hacen un lenguaje mantenible impactan negativamente en la facilidad para hacer cambios (*Viscosity*) y pueden forzar a tomar decisiones que aún no se quieren tomar/considerar (*Premature commitment*).

Debido a esto, se debe hacer un compromiso entre qué tan fácil es prototipar en un lenguaje y qué tan fácil es mantener un programa escrito en este lenguaje, una vez ya se tiene un proyecto de tamaño considerable.

2.2.4. Otras consideraciones

2.2.4.1. Comunidad

La comunidad puede ser un factor importante a la hora de elegir un lenguaje. Esta puede ser amigable, hostil, grande, pequeña, entre otros. De esto dependerá el ecosistema y para qué será usado el lenguaje.

2.2.4.2. Documentación

Qué tan buena es la documentación que ofrece el lenguaje. Esto podría incluir qué tan buenos son los ejemplos, qué tan bien está escrita, qué tan directos son los pasos para la instalación y comenzar a usar el lenguaje, entre otros.

2.2.4.3. Experiencia de instalación

Este apartado hace alusión a elementos externos al lenguaje formal. Cuanto más fácil de instalar un lenguaje, más probabilidades hay de que un usuario logre aprender el lenguaje.

2.2.4.4. Familiaridad

Un lenguaje es más fácil de aprender cuanto más familiar es. Es decir, cuanto más se parece a sistemas que el usuario ha usado previamente. Cuanto más familiar sea un lenguaje a uno que el usuario ya conozca, menos conceptos va a tener que aprender y reaprender. Por lo cual más fácil será adoptar el nuevo lenguaje.

Por esto mismo, se dice que en general los lenguajes nuevos cuentan con un *novelty budget* [Nystrom, 2021] o *strangeness budget* [Klabnik, 2015]. Lo que significa que hay un número limitado de cosas que se pueden hacer distintas a lenguajes anteriores si es que se quiere que su lenguaje sea popular.

2.2.4.5. Reproducibilidad

Qué tan fácil es correr un programa que otra persona escribió. A menudo, esto no es trivial y requiere de trabajo. Cuanto más fácil un lenguaje permita instalar las dependencias y correr programas escritos por otros, más fácil de aprender y de usar será el lenguaje.

2.2.4.6. Tooling

Qué tan buenas son las herramientas que ayudan a programar en el lenguaje. Incluyen cosas como editores, servidores de lenguaje, herramientas para *testing*, depuradores, *profilers*, administradores de paquetes, entre otros. Hoy en día, son fundamentales si es que un lenguaje quiere ofrecer una buena experiencia de usuario.

2.3. Estado del arte

A continuación se presentan lenguajes de programación existentes en el espacio de solución y cercanos a él.

2.3.1. C

Creado en los 70 por Dennis Ritchie [Ritchie, 1993]. Y difundido gracias al libro *The C Programming Language* [Kernighan y Ritchie, 2002]. Hasta el día de hoy sigue siendo ampliamente usado y es considerado la *lingua franca* de los lenguajes de programación. Conocido así porque su ABI (*Application Binary Interface*) es usada como interfaz para la comunicación entre los distintos procesos de los sistemas operativos.

Fue diseñado originalmente para programar sistemas operativos y para ser fácil de portar. Numerosas piezas fundamentales del software actual están escritas en este lenguaje. Se buscaba que fuera de alto nivel, pero cercano al *hardware*. Por lo mismo, cuenta con operaciones calcadas del *assembly*, como por ejemplo la operación *shift*.

Entre las ventajas que tiene es que pocos lenguajes pueden competir en cuanto al rendimiento y control que ofrece. Al mismo tiempo, es un lenguaje pequeño y fácil de portar a nuevos sistemas.

Entre algunas desventajas que tiene es que puede ser fácil cometer errores, tantos errores benignos como errores que pueden involucrar vulnerabilidades de seguridad. Uno de los errores más comunes que se pueden cometer, por ejemplo, es realizar un *buffer overflow*⁷. Este lenguaje ofrece muy pocas garantías al momento de programar. Y además, ofrece una biblioteca estándar mínima, por lo que hay que programar mucho desde cero.

Hola mundo:

```
#include <stdio.h>

int main() {
    printf("Hello, World!");
}
```

2.3.2. C++

C++ fue diseñado por Bjarne Stroustrup para combinar las fortalezas de Simula con las facilidades para la programación de sistemas de C [Stroustrup, 1999]. Al hacer esto, se buscó crear un lenguaje para uso general con un sesgo hacia la programación de sistemas. Para lograrlo, Stroustrup añade clases a C y también facilidades para programar de manera genérica.

Además, añade RAII al lenguaje, lo que hace significativamente más fácil trabajar con algunas partes del lenguaje.

Entre las principales ventajas están las facilidades para programar genéricamente, permitir programación orientada a objetos más fácilmente y una biblioteca estándar mucho más extensa que la de C. Hace que sea mucho más fácil empezar un nuevo proyecto con respecto a C.

Entre sus principales desventajas están la complejidad del lenguaje y sus tiempos de compilación. Por último, no tiene una ABI estable, por lo que es más difícil usar código escrito en C++ desde otros lenguajes.

⁷Un *buffer overflow* es cuando se escribe memoria fuera de la memoria que se tiene asignada

Hola mundo:

```
#include <iostream>

int main() {
    std::cout << "Hello World!" << std::endl;
}
```

2.3.3. Rust

El lenguaje fue creado en un comienzo por Graydon Hoare como un proyecto personal en 2006 mientras trabajaba en Mozilla. A continuación se hizo público en 2010 y alcanzó la versión 1.0 en 2015.

El principal diferenciador de Rust respecto a sus sucesores es que asegura *memory safety* en tiempo de compilación sin imponer un *overhead* adicional en tiempo de ejecución.

Esto lo logra usando *Single Ownership* para el manejo de la memoria y utilizando un *Borrow Checking* [Klabnik y Nichols, 2022] para evitar errores con el manejo de la memoria. El *Borrow Checking* fuerza en tiempo de compilación que una variable solo puede tener una referencia mutable o múltiples referencias inmutables.

Rust también fue diseñado con la intención de que la gente pudiera escribir programas con concurrencia y/o paralelismo sin miedo. Esto debido a la dificultad para programar este tipo de programas de una manera libre de errores. Curiosamente, se descubrió que *Borrow Checking* también era útil para evitar esta clase de errores.

Las principales desventajas de Rust consisten en su curva de aprendizaje [Ferdowsi, 2023] y sus altos tiempos de compilación.

Hola mundo:

```
fn main() {
    println!("Hello, world!");
}
```

2.3.4. Zig

Es un lenguaje creado por Andrew Kelley. Su primera versión fue publicada en 2017 [Zig, 2017]. Sus principales puntos fuertes son su legibilidad, su reusabilidad y el control sobre los recursos del ordenador que brinda.

Legibilidad en el sentido de que el lenguaje cuenta con pocas abstracciones, por lo que el

código escrito en él en general es muy directo. Esto gracias a que Zig no permite *hidden control-flow* ni *hidden allocations* [Zig, 2024a], por lo que al código no le queda opción que hacer lo que dice que hace a primera vista. Esto hace el código fácil de auditar.

Por otra parte, reusabilidad en el sentido de que en Zig el manejo de la memoria es manual y es convención que, si es que una función pide memoria dinámica, reciba como parámetro un *allocator* a usar. Esto implicaría que cualquier biblioteca escrita en Zig podría ser usada en sistemas empujados donde no se puede pedir memoria de manera dinámica.

Hola mundo:

```
const std = @import("std");

pub fn main() void {
    std.debug.print("Hello, World!\n");
}
```

2.3.5. D

D es un lenguaje de programación creado por Walter Bright. Su desarrollo comenzó en 1999 y la primera alfa apareció en 2001 [Bright *et al.*, 2020]. Fue creado como un sucesor de C++, aunque toma inspiración de otros lenguajes como Java, Python, Ruby, C# y Eiffel.

En cuanto al manejo de memoria, D es interesante en el sentido de que la mayor parte del lenguaje usa *recolección de basura*, pero si se quiere, uno puede especificar que en ciertas secciones no se use.

Hola mundo:

```
import std.stdio;

void main()
{
    writeln("Hello, World!");
}
```

2.3.6. Julia

Julia surgió como una solución ambiciosa al problema de los dos lenguajes [Storopoli *et al.*, 2021]. Sus desarrolladores querían un lenguaje que fuera tan dinámico como Ruby y tan rápido como C [Jeff Bezanson, 2012]. En muchos sentidos lo lograron. Para ciertos casos puede ser muy rápido, sin embargo, es un lenguaje manejado y, además dinámico. Se puede trabajar alrededor de estas restricciones cuando se requiere, pero aún así Julia no funciona tan bien como un lenguaje de sistemas.

Hola mundo:

```
println("Hello world!")
```

2.3.7. Swift

Swift es un lenguaje de programación creado por Apple y publicado en 2014. Este fue diseñado para ser el sucesor de Objective-C, el lenguaje usado para crear aplicaciones para sus plataformas. Toma inspiración de C++, C#, F#, Go, Haskell, Java, JavaScript, Python, Ruby y Scala [García *et al.*, 2015].

Para el manejo de la memoria utiliza *reference counting*. Swift es un caso interesante de una migración exitosa gracias a que la compatibilidad con Objective-C siempre fue una consideración.

Hola mundo:

```
print("Hello, World!")
```

2.3.8. Java

Fue diseñado por James Gosling y apareció por primera vez en 1995. Fue creado para que el código escrito en él sea más portable que el código escrito en C++. Esto lo logra emitiendo *bytecode* y corriendo sobre la JVM (*Java Virtual Machine*). Además, fue diseñado para ofrecer *threading* de manera predeterminada, cosa que C++ hasta entonces no ofrecía.

Java es uno de los lenguajes más populares, usado principalmente para servidores y software empresarial.

Hola mundo:

```
class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello, World!");  
    }  
}
```

2.3.9. Go

Es un lenguaje diseñado en Google por Robert Griesemer, Rob Pike y Ken Thompson [Go, 2021]. Fue diseñado para resolver problemas habituales con el uso de C++ y Java den-

tro de Google. Es un lenguaje que toma mucho de C, pero que fue diseñado de cero para lograr los objetivos propuestos. Primero que todo, fue diseñado para facilitar la programación concurrente y paralela. También fue diseñado para ser simple, es decir, contar con pocas abstracciones, haciéndolo fácil de aprender y, al mismo tiempo, fácil de leer código escrito por otras personas. En este sentido, Zig toma mucha inspiración de Go. Por otro lado, fue diseñado para ofrecer tiempos de compilación rápidos. Además, fue Go quien popularizó el uso de formateadores de código como estándar y contar con una guía de estilos unificada para todo el código escrito en el lenguaje.

Hola mundo:

```
package main

import "fmt"

func main() {
    fmt.Println("hello world")
}
```

2.3.10. Propuesta

Podemos ver el problema que esta memoria busca solucionar viendo la siguiente tabla 1.

Tabla 1: Comparación entre lenguajes de programación y propuesta
Fuente: Elaboración propia.

	Manejo de la memoria predecible en tiempo de compilación	Curva de aprendizaje	Fricción	Facilidad para cometer errores
C	Si	Media	Alta	Alta
C++	Si	Alta	Alta	Alta
Rust	Si	Alta	Alta	Baja
Zig	Si	Media	Media	Media
D	Parcial	Media	Media	Media
Julia	No	Baja	Baja	Media
Swift	No	Baja	Baja	Baja
Java	No	Media	Media	Baja
Go	No	Baja	Baja	Media
Python	No	Baja	Baja	Alta
Diamond	Si	Baja	Baja	Baja

En esta memoria, se busca crear un lenguaje de programación que tenga manejo de la memoria predecible en tiempo de compilación, tenga una baja curva de aprendizaje, exista poca fricción al momento de programar y que no permita cometer errores fácilmente.

Con fricción nos referimos a qué tan fácil se puede llegar y resolver un problema dado. Es importante tener en cuenta que esto tiene estrecha relación con la familiaridad con una herramienta dada. Pero, si no conozco una herramienta, pero es fácil de aprender, entonces esta tendrá menos fricción que una herramienta con una curva de aprendizaje mayor. Al mismo tiempo, si un lenguaje cuenta con una biblioteca estándar extensa, será mucho más fácil comenzar a resolver problemas con este lenguaje. El *tooling* del lenguaje también es muy importante a la hora de disminuir la fricción. Si un lenguaje es fácil de instalar y de correr, entonces también tendrá menos fricción que un lenguaje en que esto es engorroso. También, para esta dimensión importa mucho reducir el *premature commitment*, es decir, no forzar al usuario a tomar decisiones sobre cosas que aún no le importan. Por último, que haya pocas formas de hacer las cosas, es decir que en todo momento haya un claro camino para resolver un problema ayuda.

Por otro lado, la fricción también existe al momento de mantener un proyecto. Un lenguaje con tipificado estático hace visible a priori los tipos que una función recibe, por lo que hace más fácil entender cómo usar una función dada. Esta es una forma de *hidden dependencies* en los lenguajes dinámicos.

El manejo de la memoria predecible en tiempo de compilación es importante porque permitiría que el código pudiera ser usado sin problemas desde otros lenguajes de programación. Y al mismo tiempo, permitiría alcanzar el mayor rendimiento posible cuando fuera requerido. De esta manera no sería necesario tener que usar múltiples lenguajes para un mismo proyecto.

Capítulo 3

Propuesta de solución

Para resolver el problema planteado en las secciones anteriores, diseñamos y desarrollamos Diamond.

3.1. Diseño

El diseño fue un proceso iterativo-incremental. Definiendo elementos del lenguaje y volviéndolos a revisar una vez que nuevos elementos han sido implementados. Esto, debido a que la mayoría de las decisiones que se toman en un aspecto del lenguaje afectan al resto.

Se ha avanzado en la implementación al mismo tiempo que se ha avanzado en su diseño. Se optó por desarrollar tajos verticales de funcionalidad para que en cada momento se tenga algo que se puede ejecutar.

De acuerdo a lo considerado en los capítulos anteriores, se decidió por las siguientes características de manera general para el lenguaje:

- *Manejo de la memoria en tiempo de compilación basado en Single Ownership*

Debido a que libera al programador de una gran carga cognitiva al mismo tiempo que es totalmente predecible. De la misma forma, no cuenta con un *overhead adicional* en tiempo de ejecución, lo cual no es compatible con los objetivos del lenguaje.

- *Tipificado estático*

Gracias a que facilita la legibilidad del código y hace más difícil cometer errores. Al mismo tiempo, usar tipos dinámicos tiene de por sí un *overhead* adicional en tiempo de ejecución.

- *Anotaciones de tipos opcionales con inferencia de tipos completa*

Pues facilita prototipar y hace más fácil hacer cambios. Al mismo tiempo, permite retrasar el enseñar a anotar tipos, haciendo la curva de aprendizaje más incremental.

- *Tipos de datos algebraicos*

Porque permiten modelar y componer tipos de forma fácil y segura, en tiempo de compilación. Con este enfoque, se utilizan estructuras y uniones para modelar la mayoría de las cosas.

- *Programación genérica mediante interfaces*

Interfaces del tipo *type classes* o *traits* de Rust. Estas permiten escribir código genérico, pero de una manera que no es necesario analizar una función o pila de funciones para cada posible instancia, ver sección 2.1.9.1 para más detalle.

- *Paso de parámetros por valor de manera predeterminada con formas alternativas de pasar parámetros*

Pues es un modelo con una carga cognitiva baja, permitiendo trabajar solo con valores, haciendo la curva de aprendizaje más incremental.

Cosas explícitamente no incluidas:

- *Orientación a objetos*

Se buscó utilizar un modelo más simple (tipos de datos algebraicos) que orientación a objetos para el sistema de tipos. La mayoría de las cosas pueden ser modeladas con estructuras y uniones. Y la encapsulación, uno de los conceptos más importantes de la orientación a objetos, puede ser conseguida mediante módulos.

- *Macros*

Estos tienen un gran costo para la legibilidad, por eso se ha buscado evitar agregarlos al lenguaje. Aunque se está abierto a la posibilidad de agregarlos en el futuro, quizás no necesariamente macros como tal, pero algo anexo como ejecución en tiempo de compilación.

- *Excepciones*

Se prefiere utilizar valores como errores. Si una función puede retornar un error, entonces es parte del tipo que puede retornar la función. Para esto se usaría uniones.

3.1.1. Sintaxis

Se optó por una sintaxis inspirada en Lua y Python. Se podría haber elegido una sintaxis inspirada en C, pero ya existen suficientes lenguajes de este tipo. Además, Lua y Python están asociados a ser fáciles de usar y de aprender, por lo que sería ideal para un lenguaje que busca ser fácil de usar y aprender.

Al igual que en Python, se usa indentación para delimitar bloques de código. Con las ventajas y desventajas que esto significa.

Un programa se constituye de definiciones e instrucciones. Las definiciones son elementos en tiempo de compilación como funciones, tipos, interfaces y constantes. No importa el orden en que son declaradas. Por otro lado, las instrucciones corresponden al programa como tal y son ejecutadas en el orden que son declaradas.

Un módulo correspondería a un archivo con definiciones.

La sintaxis en detalle puede ser encontrada en el anexo, en la sección 5.1.

3.1.1.1. Hola Mundo

Para hacer el lenguaje lo más fácil de enseñar posible, se determinó por no requerir una función `main`.

```
print("Hola, mundo!\n")
```

3.1.2. Variables

En base a los objetivos del lenguaje, se consideró importante que existieran tres maneras de declarar variables:

1. Variables en tiempo de ejecución que *pueden cambiar su valor*.
2. Variables en tiempo de ejecución que *no pueden cambiar su valor*.
3. Variables en tiempo de compilación.

La primera es la más general y se encuentra en cualquier lenguaje imperativo. Permite modelar valores que cambian en el tiempo y permite programar de imperativamente.

Sintaxis:

```
a = 10    -- Declara una variable que puede cambiar su valor.  
a = 20    -- No habría problema con esto!
```

La segunda es un poco más difícil de justificar. Agrega complejidad al lenguaje, pero hace más fácil razonar localmente sobre un programa.

Sintaxis:

```
a be 10    -- Declara una variable que no puede cambiar  
            -- de valor. Una vez creada, no puede ser  
            -- mutada ni declarada nuevamente.  
  
a be 20    -- Esto generaría un error de compilación!
```

La tercera consiste en variables en tiempo de compilación, es decir constantes.

Sintaxis:

```
a is 10    -- Declara una constante
a is 40    -- Esto generaría un error de compilación!
```

Serían equivalentes a funciones que no toman parámetros. La gracia de esto es que el tipo de esta sería distinto dependiendo de su uso, no tendría un tipo definido como tal. Por ejemplo:

```
ten is 10    -- Declara una constante

-- Luego
a = 30 + ten  -- Aquí tendría tipo entero
b = 3.5 + ten -- Aquí tendría tipo flotante
```

Esto sería distinto a una variable en tiempo de ejecución que no cambia de valor, pues esta tendría un tipo concreto.

```
ten be 10    -- Declara una variable
              -- Tipo entero es inferido

-- Luego
a = 30 + ten  -- Aca no hay problema
b = 3.5 + ten -- Esto daría error
```

3.1.3. Asignación

Se diseñó el lenguaje con la intención de permitir razonar localmente. Por esto se decidió cambiar cómo funcionan las variables mutables con respecto a Python. En este, una asignación puede estar modificando una variable o declarando una nueva variable. Y es necesario conocer el contexto para determinar la diferencia. Por ejemplo, supongamos que tenemos el siguiente código en Python:

```
a = 10

if some_condition:
    a = 20

print(a)
```

```
>> 20
```

En Python `a = 20` puede estar declarando una nueva variable, o modificando una de un ámbito superior. Por lo tanto, es necesario tener más contexto en mente para entender qué está haciendo el código, lo que hace más fácil cometer errores.

En cambio, en Diamond, `a = 20` siempre declara una nueva variable:

```
a = 10

if someCondition
  a = 20

print(a)
```

```
>> 10
```

Y si es que se quisiera modificar una variable de un ámbito superior, sería posible con `:=`:

```
a = 10

if someCondition
  a := 20

print(a)
```

```
>> 20
```

El operador `:=` corresponde al concepto de asignación encontrado en otros lenguajes de programación.

Estos cambios permitirían al programador expresar su intención de forma clara y harían el código más fácil de entender y cambiar para alguien que no lo escribió. Aumentaría la curva de aprendizaje, pero aumentaría la legibilidad del código y haría más difícil cometer errores.

3.1.4. Estructuras de control

Nuestra propuesta contaría con las estructuras de control más comunes encontradas en otros lenguajes. Es decir, *if else*, *while* y *for*.

Se consideró agregar otras como las encontradas en un lenguaje como Ruby, pero por el momento no se consideró necesario.

3.1.5. Tipos

Los valores y variables en un lenguaje de programación pueden ser clasificados en categorías. Estos corresponden a tipos. Estos sirven para hacer más difícil la presencia de errores y también mejoran la legibilidad de los programas.

Todos los tipos en Diamond usan la convención UpperCase.

3.1.5.1. Primitivos

Los tipos primitivos son tipos que son implementados por el compilador. El lenguaje contaría con los siguientes primitivos:

Tipos entero

Int8 Int16 Int32 Int64
Uint8 Uint16 Uint32 Uint64

Tipos flotante

Float16 Float32 Float64

Otros primitivos

- *Bool*: Verdadero o falso.
- *StaticString*: Cadena de texto que no se puede cambiar, va embebida en el ejecutable. Corresponde a `const char*` en C.
- *None*: Usado para especificar que una función no retorna nada.
- *Index*: Depende de la arquitectura. El máximo valor que puede tener corresponde a la máxima cantidad de elementos que puede tener un arreglo para una arquitectura dada. Corresponde a `size_t` en C.

3.1.5.2. Punteros

Un puntero corresponde a la ubicación en la memoria de algo. Son similares a los encontrados en otros lenguajes.

```
a = 10
p = &a
*p = 20
print(*p) -- 20
print(a) -- 20
```

3.1.5.3. Arreglos

Los arreglos corresponden a una secuencia de elementos de tamaño fijo.

```
array = [10, 20, 30] : Array3[Int64]
```

Se podría acceder a los elementos del arreglo con su posición en el arreglo:

```
print(array[1]) -- 10
print(array[2]) -- 20
print(array[3]) -- 30
```

3.1.5.4. Estructuras

Las estructuras sirven para agrupar variables relacionadas. En nuestro lenguaje se definirían e inicializarían de la siguiente manera:

```
type Point
  x: Int64
  y: Int64

point = Point{x: 2, y: 3}
print(point)
```

```
>> Point{x: 2, y: 3}
```

Se eligió {} para inicializar en vez de () para hacer explícita la diferencia entre crear una estructura y llamar a un constructor.

3.1.5.5. init

Para utilizar constructores se usaría la siguiente sintaxis:

```
type Point
  x: Int64
  y: Int64

init Point(x: Int64, y: Int64)
  this.x = x
  this.y = y

point = Point(x: 2, y: 3)
print(point)
```

Usar sintaxis distintas para crear una estructura e inicializar mediante un constructor ayudaría en la legibilidad del código y permitiría entender más fácilmente qué está haciendo en el código.

3.1.5.6. copy

Por omisión, una función para copiar cada tipo es generada por el compilador. Pero a veces, la lógica para hacer esto no es directa. En ese caso, una función `copy` debiera ser provista para el tipo.

```
type Point
  x: Int64
  y: Int64

copy Point(ref point: Point)
  this.x = point.x
  this.y = point.y
```

3.1.5.7. deinit

De la misma forma que `copy`, la mayoría de los tipos pueden ser desinicializados automáticamente, pero en caso de ser necesaria lógica específica, un destructor puede ser provisto:

```
type Point
  x: Int64
  y: Int64

deinit Point()
  print("Bye {this.x}, {this.y}!")
```

Esto podría ser para liberar recursos que no se pueden liberar de manera automática. Por ejemplo, si se tuviera un *file handle* se podría hacer que automáticamente se cerrara cuando la variable que lo referencia dejara su ámbito.

3.1.5.8. Estructuras con casos

Adicionalmente, en nuestro lenguaje, las estructuras podrían tener casos. Esto busca unificar el concepto estructuras con el concepto enumeraciones encontrado en otros lenguajes.

La ventaja de esto es que permitiría modelar conceptos de forma más directa que si fueran conceptos separados. Por ejemplo, imaginemos que estamos creando un tipo para trabajar con figuras. Este tipo tendría la ubicación de su centro, y otros atributos dependiendo de qué figura es. En nuestro lenguaje podríamos modelar esto de la siguiente manera:

```
type Shape
  centerX: Float64
  centerY: Float64
```

```
case Square
  side: Float64

case Rectangle
  width: Float64
  height: Float64

case Circle
  radius: Float64
```

Esto se entendería como: Shape es un tipo que siempre tiene los atributos centerX y centerY. Dependiendo de qué figura sea, puede que tenga los atributos side, width y height, o radius.

Por otro lado, en un lenguaje en que estructuras y enumeraciones son conceptos separados, habría que crear dos tipos distintos y componerlos. Por ejemplo, así quedaría en Rust:

```
struct Shape {
  center_x: f64,
  center_y: f64,
  kind: ShapeKind,
}

enum ShapeKind {
  Square{side: f64},
  Rectangle{width: f64, height: f64},
  Circle {radius: f64},
}
```

3.1.5.9. Enumeraciones

Con los casos también se podría modelar enumeraciones.

```
type Weekday
  case Monday
  case Tuesday
  case Wednesday
  case Thursday
  case Friday
  case Saturday
  case Sunday
```

Básicamente, sería un tipo sin atributos y con solo casos.

3.1.5.10. Uniones

A veces necesitamos que una variable pueda ser de más de un tipo a lo largo de su vida. Por ejemplo:

```
a = 10 : Int64|String

if random() <= 0.5
  a := "10"

if a is Int64
  print("a is still Int64")

else a is String
  print("a is now String")
```

Sería una forma de componer tipos. Permite decir quiero que esta variable pueda almacenar este tipo o este tipo o este tipo...

3.1.5.11. @nocopy

Para todos los tipos, el compilador por omisión genera la función copy. De esta manera todos los tipos pueden ser usados como si fueran valores. Pero, a veces, hay tipos que no pueden o no deberían ser copiados. Esto se puede comunicar con la anotación @nocopy.

Al agregarla a un tipo, especificaría al compilador que no genere la función copy para el tipo y además que sería un error que el usuario intente definirla.

Esto serviría para modelar cosas que no pueden ser copiadas, cosas que no deberían ser copiadas y cosas que en general deberían evitarse copiar. Cualquier cosa que conceptualmente no tenga sentido tratar como un valor.

Por ejemplo, si tuviéramos el estado de un juego, y sería muy costoso copiarlo, entonces podríamos agregarle @nocopy para asegurarnos de que no sea copiado innecesariamente.

```
@nocopy
type Game
...
```

Otro ejemplo podría ser un *file handle*, que si bien puede ser copiado, si es que se hace sin cuidado y no se deja claro quién es el encargado de manejarlo, podría ser cerrado cuando aún está siendo usado, o cerrado múltiples veces. Esto se podría evitar utilizando @nocopy junto con *Single Ownership*. Con esto, siempre habría solo un encargado, y no podría ser mal utilizado.

Si es que se quisiera duplicar, se podría definir una función `clone`. Esta se usaría explícitamente, a diferencia de `copy` que es implícita. Luego, la función `clone`, al menos en Linux, podría usar la llamada de sistema `dup` para duplicar efectivamente el *file handle*.

Así podría verse su API:

```
@nocopy
type FileHandle
deinit FileHandle()

function open(file: Path): FileHandle
function read(file: FileHandle): String
function write(file: FileHandle, content: String): None
function clone(ref file: FileHandle): FileHandle
```

`clone` existiría para decir: sí, puedes copiar esto, aunque puede ser costoso.

Estos tipos solo podrían ser movidos o pasados por referencia. Esto se podría ver en más detalle en la sección 3.1.7.

3.1.5.12. Opacos

Un tipo puede marcarse como opaco con la palabra clave `opaque`. Esto le dice al compilador que los atributos del tipo no deberían ser accesibles desde fuera del módulo en que está definido el tipo. Esto permitiría abstraer la implementación de un tipo.

```
-- person.dmd

opaque type Person
  name: String

init Person(name: String)
  this.name = name

function getName(person: Person): String
  return person.name

function setName(mut person: Person, newName: string): None
  person.name = newName
```

```
--- main.dmd
use person

personA = person::Person(name: "Alice")
person::setName(mut personA, "Bob")
print(person::getName(personA))
```

mut será explicado en la sección de 3.1.7. Pero en términos simples significa que per son es pasado por referencia y puede ser mutado dentro de la función.

3.1.6. Funciones

Las funciones serían una forma de reutilizar código. Declarar una función en nuestro lenguaje se podría hacer con la palabra clave `function`:

```
function add(a, b)
  return a + b

print(add(20, 30))
print(add(3.5, 2.3))
```

```
>> 50
>> 5.8
```

Esto declararía una función genérica. El lenguaje inferiría los tipos más generales posibles que la función puede recibir. Lo anterior sería equivalente a:

```
function add[t: +](a: t, b: t): t
  return a + b

print(add(20, 30))
print(add(3.5 2.8))
```

```
>> 50
>> 5.8
```

La anterior declaración se lee como: `add` es una función que recibe como parámetros un tipo desconocido `t` que implementa `+`. En este caso, `+` correspondería a una interfaz, estas se verán más adelante en más detalle.

Como es esperado, también si es que se quisiera ser explícito con los tipos, también sería posible:

```
function add(a: Int32, b: Int32): Int32
  return a + b

print(add(20, 30))
```

```
>> 50
```

Que las anotaciones de tipos sean opcionales permitiría iterar sin mucha fricción en la etapa de exploración. Cuando ya se tenga claro que se está construyendo, uno podría explicitar los

tipos y de esta manera hacer el código más fácil de leer y más mantenible.

3.1.7. Paso de parámetros

Habría distintas formas en que los parámetros podrían ser pasados a las funciones. Estas serían:

1. Paso de parámetros por valor.
2. Paso de parámetros por referencia.
3. Paso de parámetros por referencia mutable.
4. Paso de parámetros por movimiento.

3.1.7.1. Por valor

Es la forma predeterminada de pasar parámetros. Cuando una variable es pasada por valor significa que una copia de la variable es creada y es pasada a la función.

```
type Person
  name: String

function sayName(person: Person): void
  print(person.name)

person = Person{name: "Alice"}
sayName(person)
```

```
>> "Alice"
```

Dentro de la función `sayName` se podría mutar `person` y esto no produciría cambios visibles desde fuera de la función.

3.1.7.2. Por referencia inmutable

Por otro lado, para pasar un parámetro por referencia, se puede agregar la palabra clave `ref` adelante de este:

```
type Person
  name: String
```

```
function sayName(ref person: Person): void
    print(person.name)

person = Person{name: "Alice"}
sayName(person)
```

```
>> "Alice"
```

Pasar parámetros por referencia inmutable permitiría evitar la copia al pasar parámetros.

De la misma manera que al pasar por valor, pasar por referencia inmutable aseguraría que no hubiera cambios visibles desde fuera de la función `sayName`.

3.1.7.3. Por referencia mutable

A veces se quiere crear funciones que modifican sus parámetros. Para especificar que un parámetro se pasa por referencia mutable, se debe agregar la palabra clave `mut` delante del parámetro.

```
type Person
    name: String

function changeName(mut person: Person, newName: String): void
    person.name = newName

person = Person{name: "Alice"}
changeName(mut person, "Bob")
print(person.name)
```

```
>> "Bob"
```

De esta manera, se especifica que la función `changeName` cambiaría el parámetro `person`. Y también esto sería visible en la llamada, no sería necesario ir a leer una función para saber que está mutando algo.

3.1.7.4. Por movimiento

Por último, hay veces que es necesario transferir al dueño de una variable. No es necesario cuando se trabaja con variables que almacenan valores, pero sí con variables que no pueden ser copiadas.

Cuando un parámetro no puede ser copiado, este es pasado por movimiento. Es decir, el *owner* del valor es transferido:

```
@nocopy
type Person
  name: String

function sayNameAndConsume(person: Person): None
  print(person.name)

person = Person{name: "Alice"}
sayNameAndConsume(person)
```

```
>> Alice
```

Después de la llamada a `sayNameAndConsume` la variable `person` ya no estaría disponible en el ámbito. Ahora el recurso estaría ligado al ámbito de la función al que fue pasado.

Al mismo tiempo, cuando una variable con un tipo que no puede ser copiado es retornada, es retornada por movimiento. De esta manera se podría retornar un tipo que no se puede copiar.

Esto permitiría que las funciones genéricas funcionen tanto con tipos que pueden ser copiados como que no pueden ser copiados. Por ejemplo:

```
@nocopy
type Person
  name: String

function identity[t](x: t): t
  return x

x = 20
x = identity(x)

p = Person{name: "Alice"}
p = identity(p)
```

Ahora, si es que se copiara la variable dentro de una función genérica, se tendría que especificar que el tipo implementa la interfaz `copy`.

Una variable solo sería consumida si es que no puede ser copiada.

```
function identity[t: copy](x: t): t
  x2 = x    -- Aquí se crea una copia de x
  return x
```

Si es que se intentara usar una variable que fue consumida, entonces se comunicaría un error en tiempo de compilación.

3.1.8. Interfaces

Una interfaz es una función o funciones que un tipo puede implementar. Corresponderían al concepto de *Traits* en Rust o *Type Classes* en Haskell.

Para lograr esto, existirían dos formas de declarar interfaces. La primera sería una interfaz que corresponde a una función:

```
interface toString[t](object: t): String
```

Por otro lado, a veces hay veces que queremos que una interfaz esté compuesta de múltiples funciones. Por ejemplo, supongamos que estamos modelando el concepto de figura, podríamos hacerlo de la siguiente manera:

```
interface area[t](shape: t): Float64
interface perimeter[t](shape: t): Float64
interface numberOfSides[t](shape: t): Int64

shape
  is area
  and perimeter
  and numberOfSides
```

Esto significaría que para que un tipo pueda ser considerado una figura, tendríamos que definir `area`, `perimeter` y `numberOfSides`. Por ejemplo:

```
-- Figura 1
type Rectangle
  width: Float64
  height: Float64

function area(rectangle: Rectangle): Float64
  return rectangle.width * rectangle.height

function perimeter(rectangle: Rectangle): Float64
  return 2 * rectangle.width + 2 * rectangle.height

function numberOfSides(rectangle: Rectangle): Int64
  return 4

-- Figura 2
type Circle
  radius: Float64

function area(circle: Circle): Float64
  return math::pi * circle.radius ** 2

function perimeter(circle: Circle): Float64
  return 2 * math::pi * circle.radius
```

```
function numberOfSides(circle: Circle): Int64
    return 0
```

Luego, se podría usar la interfaz para programar funciones genéricas:

```
function printShape(shape: shape): None
    print("My area is {area(shape)}")
    print("My perimeter is {perimeter(shape)}")
    print("And I have {numberOfSides(shape)} number of sides")

printShape(Rectangle{width: 10, height: 30})
printShape(Circle{radius: 5})
```

Si bien en este ejemplo se implementó la interfaz `shape` solo para dos tipos, la gracia de las interfaces es que no hay un tope en cuanto a quienes podrían implementar la interfaz. Una biblioteca podría proveer una interfaz y luego el usuario de la biblioteca podría implementar las interfaces para sus tipos para que el código genérico provisto por la biblioteca funcione con los tipos definidos por el usuario.

3.1.9. Módulos

A medida que crece un proyecto, es deseable dividir un programa en unidades más pequeñas para hacer el proyecto más manejable. Para eso existen los módulos. Existen múltiples enfoques:

- *Archivos como módulos*: El enfoque más común usado por Python, JavaScript y múltiples lenguajes. Cada archivo es tratado como un módulo. Adicionalmente, puede haber formas de especificar que se quiere que una carpeta sea tratada como un módulo. Para esto, generalmente un archivo dentro o fuera de la carpeta representa al módulo. Y en este archivo se decide qué exponer.
- *Carpetas como módulos*: Enfoque usado por Go y Odin. Tiene la ventaja de que hace fácil dividir el código en múltiples archivos. Puede ser conceptualmente más simple, y hacer fácil saber qué archivos comparten ámbito.

Para el caso de Diamond, usamos la opción más común de que cada archivo sea su propio módulo. Esto por las siguientes razones:

- Permitiría limitar la extensión de las funciones sin anotaciones de tipos. Cuanto más contexto se tenga sobre una función sin anotaciones, más fácil será saber cómo usarla. Estas serían más útiles en un contexto local. Con este enfoque se podría especificar que

las funciones sin anotaciones de tipos no puedan ser usadas fuera del archivo en que son declaradas.

- Si bien no hace tan claro saber qué archivos se usan entre sí sin tener que abrir los archivos, dentro de un archivo permite saber de dónde viene cada símbolo.

Dado lo anterior, la propuesta sería la siguiente:

- Cada archivo es un módulo.
- Un módulo puede *usar* o *incluir* otros módulos en la misma carpeta.
- Se puede *usar* un módulo para poder utilizar sus definiciones.
- Se puede *incluir* un módulo para reexportar sus definiciones.
- Si se quiere tratar una carpeta como un módulo, se debe crear un archivo con el mismo nombre de la carpeta dentro de esta. Todo lo que contenga e incluya este archivo será tratado como el módulo.
- Si una carpeta es un módulo, solo se podrá acceder desde fuera del módulo al archivo `mod.dmd` que lo representa.

No estaría contemplado que se pudiera referenciar código desde internet, al menos de momento.

3.1.10. Single Ownership

La memoria de cada variable y atributo es manejada mediante *Single Ownership*. Esto quiere decir que en cada momento hay un único responsable por recurso. Si un tipo implementa copy entonces en general este es copiado. Si no lo hace, la variable es movida. Al final de un ámbito, todos los recursos del ámbito son liberados.

3.1.10.1. Boxed y new

La mayoría de las cosas que pueden ser asignadas en el *stack* son asignadas en este, pero a veces no sabemos cuánto espacio algo va a ocupar. Como, por ejemplo, una cadena de texto o una lista. Como no sabemos cuánto espacio va a ocupar en tiempo de compilación, no podemos asignarla en el *stack*. Por lo que es necesario pedir memoria dinámica. Para pedir memoria de manera dinámica se podría utilizar la palabra clave `new` seguida de un literal o un tipo.

```
a = new 20 : Boxed[Int64]
print(*a)
```

```
>> 20
```

El operador `new` crea un valor con tipo `Boxed[Int64]`. Este funciona igual que un puntero, con la diferencia que es considerado el `owner` de la memoria a la que apunta. Si `a` sale del ámbito, entonces la memoria a la que apunta es liberada.

Se puede hacer lo mismo, pero en vez de especificar un literal, se especifica un tipo:

```
a = new Int64 : Boxed[Int64]
*a = 30
print(*a)
```

```
>> 30
```

Otro ejemplo interesante sería el pedir memoria para un arreglo de tamaño no conocido en tiempo de compilación:

```
size = valueComputedAtRuntime()
a = new Array{size}[Int64] : Boxed[Array[Int64]]
```

3.1.10.2. Memory Safety

En la actualidad es difícil diseñar un nuevo lenguaje de sistemas sin considerar el aspecto de *memory safety*. Esta es considerada una de las principales características de Rust. Y hay mucha gente que considera que nuevos proyectos no deberían ser empezados en lenguajes sin *memory safety*. Incluso la Casa Blanca ha emitido un comunicado al respecto de esto [The White House, 2024].

No entraremos mucho en la controversia del asunto, pero es importante recalcar que no todo es blanco y negro y que siempre hay *tradeoffs*. Un lenguaje interesante en este sentido es Zig, pues, si bien no cuenta con el mismo nivel de *memory safety* de Rust, hace mucho por reducir las posibles fuentes de *undefined behaviour* e incluso en ciertos casos incluso ofrece más control para asegurar la correctitud del software [Kladov, 2024].

En cuanto a nuestra propuesta, se quiere que esta ofrezca *memory safety* por omisión. Esto, para ofrecer una mejor experiencia de usuario y evitar vulnerabilidades.

Aunque aún no está decidido por cuál camino lograr esto. Hay dos opciones. Agregar un elemento similar a `unsafe` de Rust y no permitir usar características no seguras fuera de este ámbito. Dos, proveer *memory safety* mediante verificaciones en tiempo de ejecución.

Y de esta manera tener un solo lenguaje.

Las dos opciones son interesantes y tienen ventajas y desventajas. Por un lado, la opción de Rust asegura que es imposible la existencia de *undefined behaviour* en *tiempo de compilación*. Aunque al mismo tiempo no asegura la inexistencia de *undefined behaviour* en código *unsafe*. Pero si es que existe, la gracia es que estaría limitado a una sección limitada del programa. Por otro lado, usar verificaciones en tiempo de ejecución permitiría tener un modelo mental unificado del lenguaje, y aseguraría que es imposible causar *undefined behaviour* mientras se mantengan las verificaciones habilitadas. Pero implicaría un costo adicional en tiempo de ejecución debido a las verificaciones que se tendrían que hacer. Sería interesante ver qué tan eficiente se puede hacer esto.

3.2. Implementación

La implementación de esta memoria puede ser encontrada en github.com/diamond-lang/diamond. Aquí se encuentra el código del compilador, junto con instrucciones para su compilación.

Actualmente, el lenguaje puede ser compilado y correr exitosamente todos los *tests* en las plataformas *Windows*, *macOS* y *Linux*.

Los *tests* corresponden a pequeños programas en nuestro lenguaje con la salida esperada que prueban la correcta implementación de las distintas funcionalidades del lenguaje.

No todas las funcionalidades han sido implementadas debido al tiempo que toma desarrollar un lenguaje de programación, pero si una funcionalidad está implementada, habrá uno o varios *tests* para esta.

3.2.1. Tecnologías usadas

Dado que los objetivos de la memoria son crear un lenguaje que permita aprovechar los recursos del ordenador, era necesario que el compilador produjera código de máquina.

Para hacer esto había tres alternativas. Uno, crear un compilador que produzca C y luego llamar a otro compilador. Dos, crear un compilador que emita código de máquina. Tres, usar un *backend* ya establecido como el de GCC o LLVM.

Emitir C implicaba invocar un compilador de C y tener que reparsear y recompilar el código. Y haría más difícil dejar de depender de C. Por otro lado, el código de máquina es dependiente de la arquitectura y de la plataforma que se tiene como *target*. Implementar la generación de código sería imposible por el tiempo, pues habría que hacerla para cada *target* que se quisiera soportar. Y además, no contaría con los años de trabajo ya realizado en optimización

en los *backends* ya existentes. Por todo esto, se decidió por utilizar un *backend* ya existente.

Debido a que una gran cantidad de lenguajes usan LLVM (Clang, Rust, Swift, Julia, etc...) y además que este cuenta con mejor documentación, se decidió por utilizar LLVM como *backend*. La ventaja de utilizar un *backend* ya existente es que, además de proveer optimizaciones del código a nivel general, también pueden optimizar en la generación del *assembly*, emitiendo código específico para el procesador final para el que estoy compilando.

Se podría haber utilizado cualquier lenguaje para programar el compilador, pero la elección final fue C++. Esto, principalmente porque LLVM está escrito en C++ y la mayoría de la documentación es para este lenguaje.

3.2.2. Estructura del proyecto

La estructura del proyecto es la siguiente:

```
compilador
├── .github
├── deps
│   ├── musl
│   └── llvm
├── src
│   ├── codegen
│   ├── semantic
│   ├── ast.hpp
│   ├── ast.cpp
│   ├── codegen.hpp
│   ├── data_structures.cpp
│   ├── data_structures.hpp
│   ├── errors.cpp
│   ├── errors.hpp
│   ├── lexer.cpp
│   ├── lexer.hpp
│   ├── main.cpp
│   ├── parser.cpp
│   ├── parser.hpp
│   ├── semantic.hpp
│   ├── shared.hpp
│   ├── tokens.cpp
│   ├── tokens.hpp
│   ├── utilities.cpp
│   └── utilities.hpp
└── std
```

```
|  
├─ test  
├─ build.py  
├─ get_llvm.py  
├─ LICENSE  
├─ README.md  
└─ test.py
```

Carpetas de importancia:

- *src*: Contiene código fuente del compilador. Siendo *main.cpp* el punto de entrada.
- *test*: Contiene *tests* para el compilador. Consisten en pequeños programas que buscan probar que el compilador y el lenguaje funcionan como esperado.
- *std*: Contiene los primeros componentes de la biblioteca estándar de nuestro lenguaje.
- *.github*: Carpeta específica a *Github*, aquí se detalla el flujo para asegurarse que el compilador siga funcionando correctamente al hacer cambios.
- *semantic*: Contiene los archivos del compilador dedicados al análisis semántico del compilador. Corresponde a la parte más extensiva y con mayor complejidad del compilador.
- *codegen*: Contiene los archivos para la generación de código. Esta parte del compilador corresponde al *backend*.

Archivos de importancia:

- *get_llvm.py*: *Script* para obtener una versión ya compilada de LLVM. Esto debido a que LLVM es una base de código de tamaño significativo y al menos toma media hora de compilar en un buen computador.
- *build.py*: *Script* que compila el proyecto. Genera el ejecutable del compilador.
- *test.py*: *Script* que corre todos los *tests* del proyecto.

3.2.3. Arquitectura de la cadena de compilación

La estructura de la cadena de compilación puede verse en la figura 5.

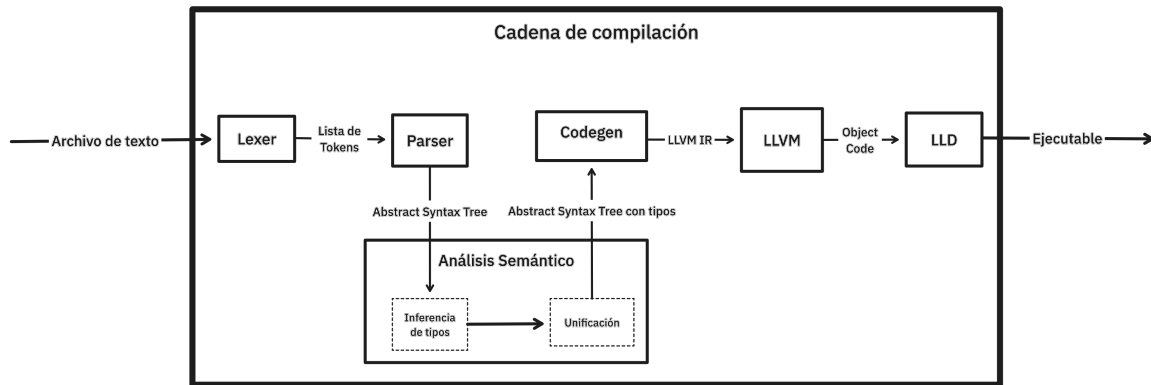


Figura 5: Arquitectura de la cadena de compilación

Fuente: Elaboración propia

En el diagrama se pueden apreciar los distintos componentes y etapas del compilador. Cada etapa tiene una entrada y una salida. Los componentes de *Lexer*, *Parser* y *Análisis semántico* corresponden al *frontend*. Los componentes componente de *Codegen* y *LLVM* corresponden al *backend*.

La salida de cada etapa es inspeccionable. Al pasarle un programa al compilador, se le puede pedir que imprima el resultado de una etapa en particular. A continuación veremos cómo hacer esto al ver cada etapa en particular.

3.2.4. Lexer

El lenguaje hasta el momento cuenta con los siguientes *tokens*:

LeftParen, *RightParen*, *LeftBracket*, *RightBracket*, *LeftCurly*, *RightCurly*, *Comma*, *Plus*, *Slash*, *Modulo*, *Star*, *Minus*, *Colon*, *Ampersand*, *Dot*, *Not*, *NotEqual*, *Greater*, *GreaterEqual*, *Less*, *LessEqual*, *ColonEqual*, *Equal*, *EqualEqual*, *Be*, *Integer*, *Float*, *Identifier*, *String*, *StringLeft*, *StringMiddle*, *StringRight*, *If*, *Else*, *While*, *Function*, *Interface*, *Builtin*, *Type*, *True*, *False*, *Or*, *And*, *Use*, *Break*, *Continue*, *Return*, *Mut*, *New*, *Include*, *Extern*, *LinkWith*, *NewLine*, *EndOfFile*

Generalmente, en el *lexer* no es necesario hacer mucho trabajo, son menos de 400 líneas de código. En esta etapa se agrupan caracteres en grupos que hacen sentido para el lenguaje. Se reconocen los elementos mínimos de la notación del lenguaje, y cosas como palabras claves, identificadores, *strings*, enteros y flotantes. No se utilizó una herramienta existente para esto.

Cabe notar que, si bien uno podría no hacer esta etapa, con las ventajas que esto conlleva, esto resultó ser significativamente más lento, probablemente porque fuerza al *parser* a hacer más *backtracking*. Es decir, volver atrás y rehacer trabajo. Por ejemplo, podríamos estar

parseando una sección de texto como una sentencia. Primero probamos *parsearla* como una asignación, *parseamos* un identificador correctamente y luego notamos que el siguiente carácter no es un signo igual. Entonces, ahora vamos a probar *parsear* la sentencia como una llamada. Hay que nuevamente *parsear* el identificador.

La salida de etapa sería una serie de *tokens*, supongamos tenemos el archivo `hello.dmd` con el siguiente contenido:

```
a = 3.56
print("Hello")
```

Podríamos obtener los *tokens* con la opción `emit --tokens`:

```
diamond emit --tokens hello.dmd
```

La salida sería la siguiente:

```
token::Identifier(a)
token::Equal
token::Float(3.56)
token::NewLine
token::Identifier(print)
token::LeftParen
token::String("Hello")
token::RightParen
token::EndOfFile
```

La implementación puede ser encontrada en github.com/diamond-lang/diamond/blob/main/src/lexer.cpp.

3.2.5. Parser

Existen múltiples formas en que se puede escribir un *parser*, para este proyecto se probaron dos enfoques. En primer lugar, se probó un enfoque inspirado en *Compiling to Assembly from Scratch* [Keleshev, 2020] utilizando *parser combinators*. Era interesante porque permitía ahorrarse la etapa de *lexing* y, en general, implicaba menos código, con las ventajas que eso conlleva. Pero al final se terminó por usar otro enfoque con el que ya se había trabajado antes. Siendo este escribir un *recursive descent parser* y añadiendo una etapa de *lexing* inspirada en *Crafting Interpreters* [Nystrom, 2021]. Sobre todo por el rendimiento. Con el primer enfoque, el compilador pasaba la mayoría del tiempo en el *parser*, cuando en general, el cuello de botella debería encontrarse en el análisis semántico y la generación de código.

La entrada de esta etapa corresponde a una lista de *tokens* y su salida es una representación del programa como un árbol. Este es conocido como AST (Abstract Syntax Tree). Si es que en esta etapa se encuentra algún error, entonces el compilador muestra los errores y termina su ejecución.

Si es que queremos, podemos pedirle al compilador que emita el árbol generado a partir de un archivo de texto. Por ejemplo, con el mismo archivo anterior `hello.dmd`:

```
a = 3.56
print("Hello")
```

Y se usa el comando:

```
diamond emit --ast hello.dmd
```

La salida sería la siguiente:

```
program
├── =
│   ├── a
│   └── 3.56
└── print
    └── "Hello"
```

Acá podemos notar que el compilador correctamente reconoce las dos sentencias del programa y crea el AST correctamente.

3.2.6. Análisis semántico

En esta etapa se analiza, se realiza inferencia de tipos y se chequea que el programa tenga sentido. Algunas de las cosas que se verifican:

- Se determina y chequea el tipo de las variables
- Se verifica la existencia de variables, tipos y funciones
- Se verifica que cuando acceda a un atributo, este exista
- Se verifica que todos los atributos de una estructura sean inicializados
- Se verifica que no se re-declare o mute variables inmutables

- Se verifica que las llamadas retornen None si es que se usan como sentencia
- Se determina qué funciones son llamadas

En resumen, todas las reglas del lenguaje que son chequeadas en esta etapa. Si es que se encuentra algún error, entonces el compilador imprime los errores y termina su ejecución.

Si es que no existen errores, significa que el programa es correcto y ya conoce los tipos de todas las expresiones del programa.

Al igual que en las etapas anteriores, se puede pedir al compilador que emita la salida de esta etapa. Supongamos que tenemos un archivo `addFour.dmd`:

```
function addFour(a)
    return a + 4

number = 30
number2 = 30.5
print(addFour(number))
print(addFour(number2))
```

Inspeccionar la salida de esta etapa se puede hacer con el siguiente comando:

```
diamond emit --ast-with-types addFour.dmd
```

La salida sería la siguiente:

```
program
├──=
│   ├──number
│   └──30: Int64
├──=
│   ├──number2
│   └──30.5: Float64
├──print: None
│   └──addFour: Int64
│       └──number: Int64
├──print: None
│   └──addFour: Float64
│       └──number2: Float64
└──function addFour[t: + and number](a: t): t
    ├──return
    │   ├──+; t
    │   │   ├──a: t
    │   │   └──4: t
```

Podemos notar que el programa infiere correctamente que la `addFour` es una función genérica. Recibe un parámetro de tipo `t` e identifica correctamente que este tipo debiera implementar las interfaces `+` y `number`.

La función genérica es instanciada dos veces. Una con el tipo `Int64` y otra con el tipo `Float64`. Podemos pedirle al compilador que emita el código instanciado con el siguiente comando:

```
diamond emit --ast-with-concrete-types hello2.dmd
```

La salida sería la siguiente:

```
program
├──=
│   ├──number
│   └──30: Int64
├──=
│   ├──number2
│   └──30.5: Float64
├──print: None
│   └──addFour: Int64
│       └──number: Int64
├──print: None
│   └──addFour: Float64
│       └──number2: Float64
├──function addFour(a: Int64): Int64
│   └──return
│       └──add: Int64
│           ├──a: Int64
│           └──4: Int64
└──function addFour(a: Float64): Float64
    └──return
        └──add: Float64
            ├──a: Float64
            └──4: Float64
```

Cabe notar que en esta etapa no se crea un nuevo árbol, sino que, se reusa el ya creado y se añade el tipo a cada nodo que sea pertinente.

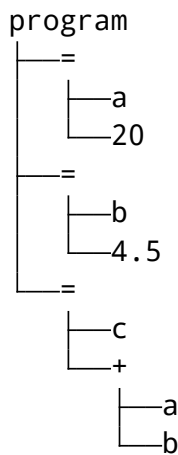
3.2.7. Inferencia de tipos

La inferencia de tipos está estructurada como dos iteraciones sobre el AST en la etapa de análisis semántico.

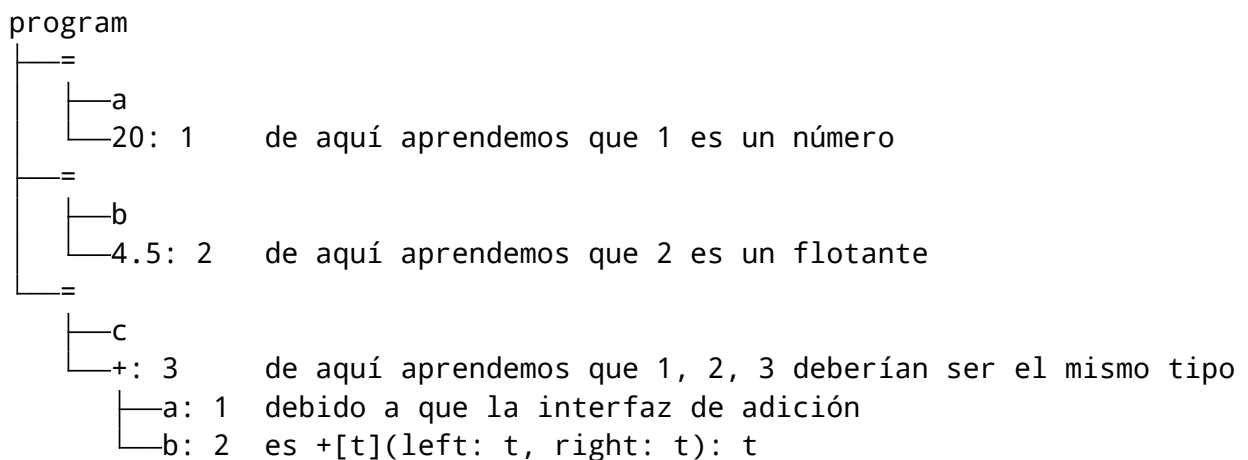
En la primera iteración se recorre el AST y se crean variables de tipos para cada expresión cuyo tipo aún no está claro. Al mismo tiempo, se crean restricciones para las variables en base a las interfaces usadas. Por ejemplo, imaginemos que tenemos el siguiente programa:

```
a = 20
b = 4.5
c = a + b
```

Esto crearía el árbol:



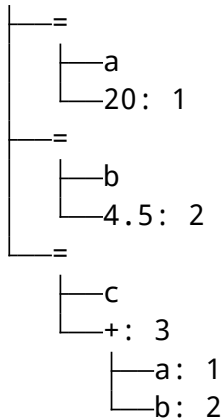
Las variables de tipos son representadas como números y las restricciones como conjuntos. Teniendo esto claro, lo primero que haríamos sería asignar variables de tipo y obtener restricciones:



Una vez se tienen las restricciones, se unen los conjuntos que comparten variables. Luego,

se busca un representante en cada conjunto. Este representante debe ser un tipo en concreto, como `Int64`. Si es que no se encuentra un representante al conjunto, se le asigna una variable de tipo *final*. Estas últimas son representadas como letras y serían como las que se utilizan en las funciones genéricas.

Este sería el resultado de esta fase en este ejemplo en particular:



Labelled type constraints

`t: {1, 2, 3}`

Interface constraints

`t: float and +`

Con esto ya tendríamos todo lo necesario para pasar a la siguiente etapa. La cual correspondería a unificación. En esta etapa se asigna el tipo final que tendría cada expresión. Para hacer esto, se busca en qué conjunto se encuentra la variable de una expresión. El representante sería el tipo final de la expresión.

En este caso, todas las variables se encuentran en el mismo conjunto. Por lo que `t` les correspondería. Pero como `t` también es una variable y necesitamos un tipo concreto, elegimos el tipo concreto predeterminado que podría tomar en base a las interfaces a las que adhiere un tipo. En este caso, el tipo predeterminado sería `Float64` pues es el tipo por omisión para la interfaz `float`. El tipo predeterminado no dependería de la plataforma.

Una vez terminada esta etapa, si es que no hubo errores, se sabe que el programa es válido y se puede proceder a generar el código.

3.2.8. Generación de código

Esta última etapa corresponde a la generación de código. Como ya comentamos, usamos LLVM para esto. Para lograrlo, se recorre el AST y se genera una representación intermedia que LLVM puede entender y optimizar. Esta es conocida como LLVM IR (LLVM Intermediate Representation). Es un lenguaje con una estructura similar al *assembly*, pero de más alto nivel. LLVM usa esta representación intermedia para realizar optimizaciones. Luego de que

las optimizaciones han sido aplicadas, se traduce el LLVM IR a código de máquina. LLVM soporta una variedad de *targets* y más pueden ser creados.

Para generar el código se utiliza un objeto que provee LLVM llamado IRBuilder. Esta es una clase que facilita la generación de LLVM IR.

Podemos ver el código generado utilizando la opción `emit --llvm-ir`. Por ejemplo, si quiéramos ver el código generado para el archivo `add.dmd`:

```
function add(a: Float64, b: Float64): Float64
    return a + b

result = add(4, 5)
print(result)
```

Podríamos usar el comando:

```
diamond emit --llvm-ir add.dmd
```

La salida sería la siguiente:

```
; ModuleID = 'add.dmd'
source_filename = "add.dmd"

@0 = private unnamed_addr constant [2 x i8] c"\0A\00", align 1
@1 = private unnamed_addr constant [3 x i8] c"%g\00", align 1

declare ptr @malloc(i64)

declare void @free(ptr)

declare i32 @printf(ptr, ...)

define void @"print[Float64]"(double %value) {
entry:
    %calltmp = call void @"printWithoutLineEnding[Float64]"(double %value)
    %putchar = call i32 @putchar(i32 10)
    ret void
}

define void @"printWithoutLineEnding[Float64]"(double %value) {
entry:
    %calltmp = call i32 (ptr, ...) @printf(ptr noundef nonnull @1, double %value)
    ret void
}

define double @add(double %a, double %b) {
entry:
    %addtmp = fadd double %a, %b
```

```
ret double %addtmp
}

; Function Attrs: nofree nounwind
declare noundef i32 @putchar(i32 noundef) #0

define i32 @main() {
entry:
  %result = alloca double, align 8
  %calltmp = call double @add(double 4.000000e+00, double 5.000000e+00)
  store double %calltmp, ptr %result, align 8
  %result1 = load double, ptr %result, align 8
  %calltmp2 = call void @"print[Float64]"(double %result1)
  ret i32 0
}

attributes #0 = { nofree nounwind }
```

3.2.9. Linking

La última parte de la cadena de compilación es *linking*. Este consiste en tomar el código objeto generado y enlazarlo con las bibliotecas que va a usar el programa para convertirlo en un ejecutable. Actualmente, los programas generados por el compilador tienen como dependencia la biblioteca estándar de C, aunque en un futuro se espera que esto deje de ser el caso.

La forma más fácil de crear un ejecutable a partir de código objeto es pasarlo a un compilador de C. La otra opción es utilizar un *linker* desde una biblioteca e invocarlo con todos los argumentos necesarios para enlazar el código objeto con la biblioteca estándar de C. Requiere un poco más trabajo, pero es considerablemente más rápido. Además, hace más fácil en el futuro dejar de depender de C. En nuestro caso utilizamos LLD, ya que es parte del proyecto de LLVM y se puede compilar junto con él.

3.2.10. ABI

Lamentablemente, LLVM no se hace cargo de la ABI, es decir, de la interfaz binaria. Esta determina cómo se pasan y retornan objetos entre programas a nivel de lenguaje de máquina. Y cada plataforma tiene su propia convención. Estas especifican cómo se pasan estructuras, arreglos, cómo se retornan, qué registros se usan para qué, entre otras cosas.

Se implementó parcialmente la ABI para Linux x86-64. Con el objetivo de lograr utilizar la biblioteca raylib, una biblioteca para el desarrollo de videojuegos. Gracias a las consideraciones que se tomaron, se logró correctamente abrir una ventana y mostrar una etiqueta de texto. Esto lo podemos evidenciar en la figura 6.

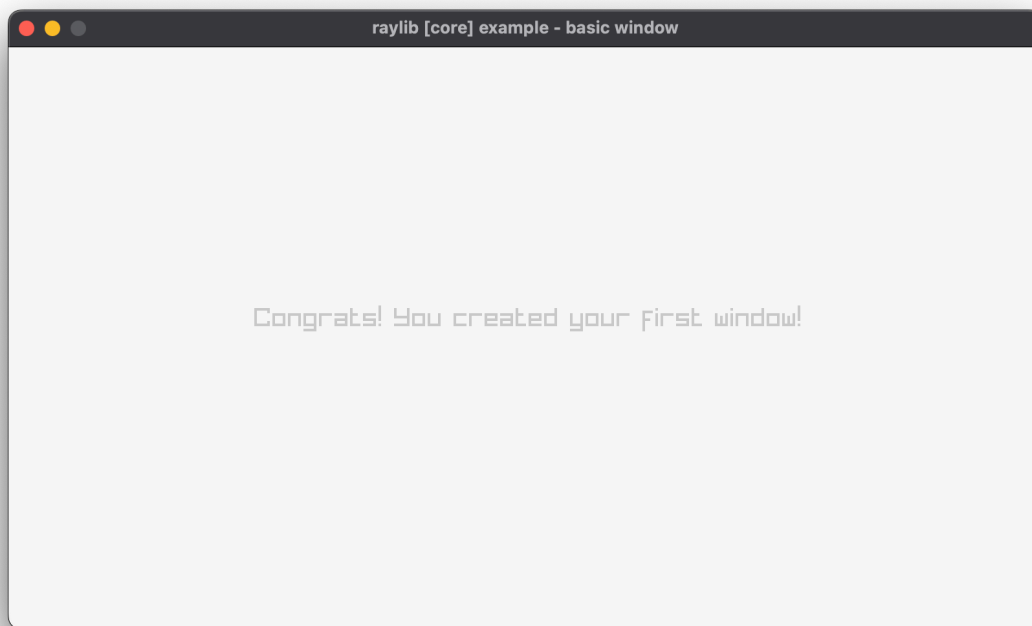


Figura 6: Ventana abierta usando raylib y nuestro lenguaje
Fuente: Elaboración propia

Esto se logra con el siguiente código en nuestro lenguaje:

```
use "raylib"

InitWindow(800, 450, "raylib [core] example - basic window")

SetTargetFPS(60)

while not WindowShouldClose()
    BeginDrawing()
    ClearBackground(raywhite())
    DrawText("Congrats! You created your first window!", 190, 200, 20, lightgray())
    EndDrawing()

CloseWindow()
```

Con la ayuda de módulo llamado raylib teniendo las siguientes definiciones:

```
link_with "$(pkg-config --libs --cflags raylib)"

extern InitWindow(width: Int32, height: Int32, title: String): None
extern WindowShouldClose(): Bool
extern CloseWindow(): None
```

```
extern SetTargetFPS(fps: Int32): None
extern BeginDrawing(): None
extern ClearBackground(color: Color): None
extern DrawText(text: String, pos_x: Int32, pos_y: Int32, font_size: Int32, color:
    Color): None
extern EndDrawing(): None

type Color
  r: Int8
  g: Int8
  b: Int8
  a: Int8

function raywhite(): Color
  return Color{r: 245, g: 245, b: 245, a: 255}

function lightgray(): Color
  return Color{r: 200, g: 200, b: 200, a: 255}
```

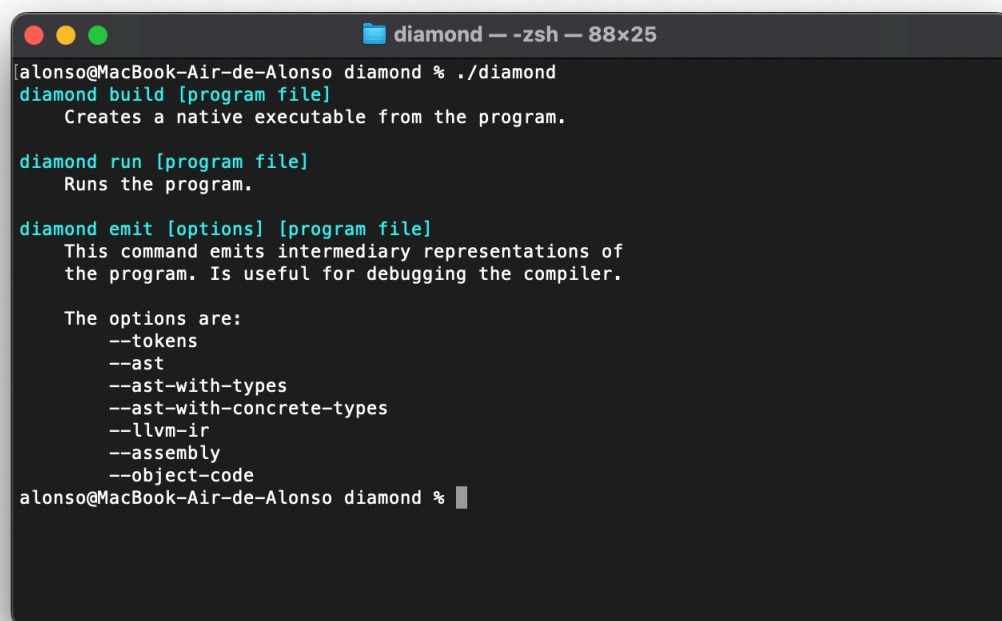
3.2.11. Interfaz

Parte de la implementación de un lenguaje de programación es su interfaz y cómo comunica al usuario. Como se comentó en los objetivos, se quería implementar una interfaz inspirada por el lenguaje de programación Elm.

Para ello se agregó color al *output* del compilador, para esto se usó *ANSI escape codes*. Esto en general funciona de manera predeterminada en sistemas Unix, como Linux y macOS, aunque requirió más trabajo en Windows.

Se siguieron las heurísticas presentadas en *Compiler Errors for Humans* [Czaplicki, 2015], y también se tomó en cuenta qué características no incluir para poder producir errores que sean claros. Hay características que puede tener un lenguaje de programación que pueden hacer muy difícil producir buenos mensajes de error.

Actualmente, la interfaz del compilador se puede ver la figura 7.

A screenshot of a terminal window titled "diamond — zsh — 88x25". The prompt is "alonso@MacBook-Air-de-Alonso diamond %". The user has entered the command "diamond", which has triggered a help message. The help message lists three main commands: "diamond build [program file]" (Creates a native executable from the program.), "diamond run [program file]" (Runs the program.), and "diamond emit [options] [program file]" (This command emits intermediary representations of the program. Is useful for debugging the compiler.). Below these, it lists options: --tokens, --ast, --ast-with-types, --ast-with-concrete-types, --llvm-ir, --assembly, and --object-code. The prompt is now "alonso@MacBook-Air-de-Alonso diamond %".

```
alonso@MacBook-Air-de-Alonso diamond % ./diamond
diamond build [program file]
    Creates a native executable from the program.

diamond run [program file]
    Runs the program.

diamond emit [options] [program file]
    This command emits intermediary representations of
    the program. Is useful for debugging the compiler.

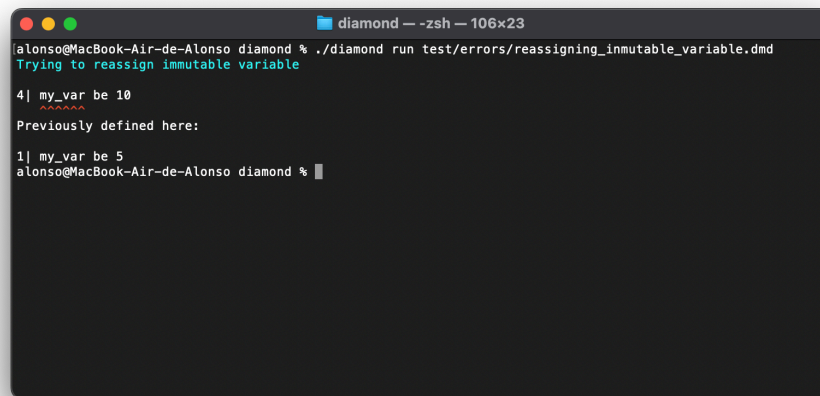
    The options are:
    --tokens
    --ast
    --ast-with-types
    --ast-with-concrete-types
    --llvm-ir
    --assembly
    --object-code
alonso@MacBook-Air-de-Alonso diamond %
```

Figura 7: Interfaz de línea de comandos del compilador
Fuente: Elaboración propia

Podemos ver algunos ejemplo de errores. Supongamos tenemos el siguiente código:

```
my_var be 5
my_var be 10
```

En la figura 8 podemos ver la salida del compilador.



```
diamond --zsh -- 106x23
alonso@MacBook-Air-de-Alonso diamond % ./diamond run test/errors/reassigning_immutable_variable.dmd
Trying to reassign immutable variable
4| my_var be 10
~~~~~
Previously defined here:
1| my_var be 5
alonso@MacBook-Air-de-Alonso diamond %
```

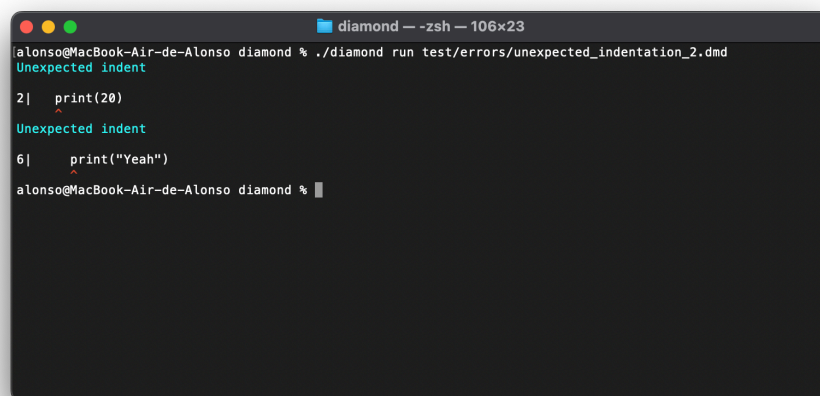
Figura 8: Error generado al reasignar variable inmutable
Fuente: Elaboración propia

Por otro lado, si tuviéramos el siguiente código:

```
print(10)
  print(20)
print(false)

  print("Yeah")
```

La salida sería de la de la figura 9.



```
diamond --zsh -- 106x23
alonso@MacBook-Air-de-Alonso diamond % ./diamond run test/errors/unexpected_indentation_2.dmd
Unexpected indent
2|  print(20)
   ^
Unexpected indent
6|  print("Yeah")
   ^
alonso@MacBook-Air-de-Alonso diamond %
```

Figura 9: Error generado al haber indentación inesperada
Fuente: Elaboración propia

3.2.12. Tests

Los *tests* corresponden a pequeños programas con un *output* esperado. Corresponden a *tests* de integración. Por ejemplo, uno de los primeros *tests* escritos:

```
print(4.0 + 6.0 * 3.0 - 6.0)
print(-7.0 + 4.0)
print(-2.5 - 5.0)
print(-(-6.0))
print((4.0 + 6.0) * 3.0)
print(5.0 + (-(-(-7.0))))
```

--- Output

```
16
-3
-7.5
6
30
-2
---
```

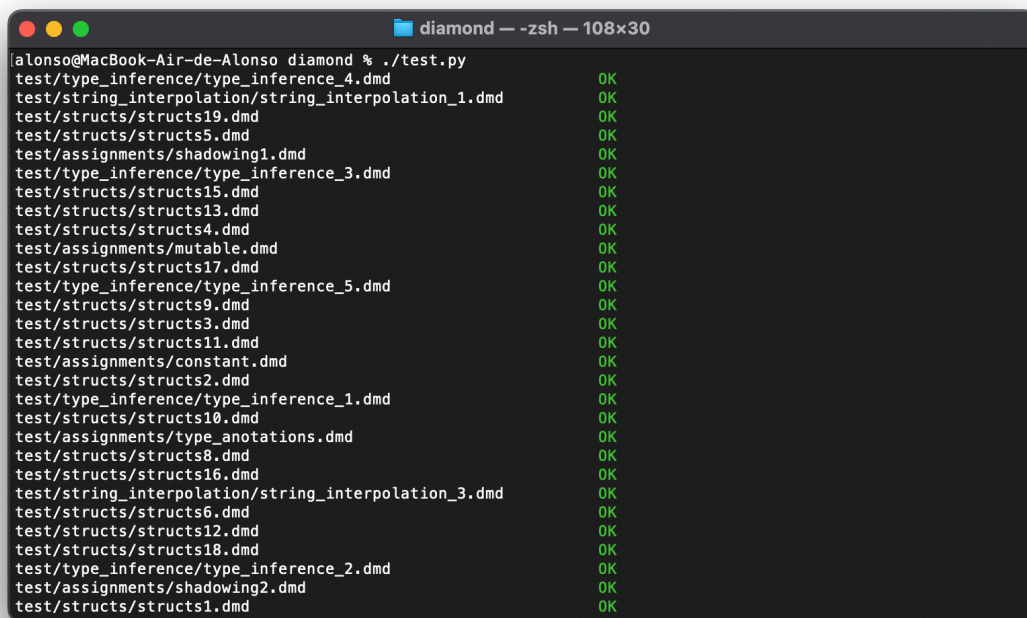
Este *test* busca probar el correcto funcionamiento de expresiones aritméticas. Hasta el momento, existen 116 *tests*. Cada vez que se añade una nueva funcionalidad, o se encuentra un *bug*, nuevos *tests* son agregados.

Actualmente, los *tests* están divididos en las siguientes categorías:

arrays, assignments, control_flow, errors, expressions, functions, modules, mut, new, pointers, string_interpolation, structs, type_inference.

El *script* llamado *test.py* es encargado de correr los *tests*. Por omisión, corre todos los *tests* en la carpeta *test*, aunque puede pasársele como argumento una carpeta o un archivo específico para restringir los *tests* ejecutados. Este *script* para cada *test* invoca al compilador pasándole el *test* como input. El compilador corre el programa y luego el *script* compara el *output* con el esperado.

En la figura 10 se puede ver la salida de correr los *tests script*.



```
alonso@MacBook-Air-de-Alonso diamond % ./test.py
test/type_inference/type_inference_4.dmd          OK
test/string_interpolation/string_interpolation_1.dmd OK
test/structs/structs19.dmd                        OK
test/structs/structs5.dmd                         OK
test/assignments/shadowing1.dmd                   OK
test/type_inference/type_inference_3.dmd          OK
test/structs/structs15.dmd                        OK
test/structs/structs13.dmd                        OK
test/structs/structs4.dmd                         OK
test/assignments/mutable.dmd                     OK
test/structs/structs17.dmd                        OK
test/type_inference/type_inference_5.dmd          OK
test/structs/structs9.dmd                         OK
test/structs/structs3.dmd                         OK
test/structs/structs11.dmd                        OK
test/assignments/constant.dmd                    OK
test/structs/structs2.dmd                         OK
test/type_inference/type_inference_1.dmd          OK
test/structs/structs10.dmd                       OK
test/assignments/type_annotiations.dmd           OK
test/structs/structs8.dmd                         OK
test/structs/structs16.dmd                        OK
test/string_interpolation/string_interpolation_3.dmd OK
test/structs/structs6.dmd                         OK
test/structs/structs12.dmd                        OK
test/structs/structs18.dmd                        OK
test/type_inference/type_inference_2.dmd          OK
test/assignments/shadowing2.dmd                   OK
test/structs/structs1.dmd                         OK
```

Figura 10: Salida producida al correr los tests del compilador

Fuente: Elaboración propia

Estos permiten realizar una cobertura completa de las funcionalidades del lenguaje.

3.3. Biblioteca estándar

La biblioteca estándar aún está en una etapa muy temprana, pero no se podía dejar fuera, porque sin una, un lenguaje no puede producir efectos visibles. Por esto mismo, de las primeras cosas implementadas fue la habilidad para poder imprimir.

Si bien, en un comienzo, esta habilidad la proveía el compilador directamente, ahora se encuentra definida en la biblioteca estándar. Para esto se hace uso de la habilidad para declarar funciones a ser implementadas por bibliotecas externas. En este caso, `printf` se declara como una función externa.

```
interface printWithoutLineEnding[t](object: t): None

extern printf(format: String, ...): Int32

function print[t](value: t): None
  printWithoutLineEnding(value)
```

```
_ = printf("\n")

function printWithoutLineEnding(value: Float64): None
  _ = printf("%g", value)

function printWithoutLineEnding(value: Int64): None
  _ = printf("%d", value)

function printWithoutLineEnding(value: Int32): None
  _ = printf("%d", value)

function printWithoutLineEnding(value: Int8): None
  _ = printf("%d", value)

function printWithoutLineEnding(value: Bool): None
  if value _ = printf("true")
  else _ = printf("false")

function printWithoutLineEnding(value: String): None
  _ = printf(value)

function printWithoutLineEnding[t](value: Array[t]): None
  _ = printf("[")
  i = 1
  while i <= size(value)
    printWithoutLineEnding(value[i])
    if not i == size(value)
      _ = printf(", ")
    i := i + 1
  _ = printf("]")

function printWithoutLineEnding[t: type](struct: t): None
  printStruct(struct)

builtin printStruct[t: type](struct: t): None
```

Se declara el *core* del lenguaje como parte de la biblioteca estándar. Los operadores están definidos como interfaces y son implementados para diferentes tipos. De esta manera se usa la misma maquinaria que para el resto de las funciones.

```
interface ==[t](left: t, right: t): Bool
interface !=[t](left: t, right: t): Bool

interface <[t](left: t, right: t): Bool
interface >[t](left: t, right: t): Bool
interface <=[t](left: t, right: t): Bool
interface >=[t](left: t, right: t): Bool

interface +[t](left: t, right: t): t
interface -[t](left: t, right: t): t
interface -[t](number: t): t
```

```
interface *[t](left: t, right: t): t
interface /[t](left: t, right: t): t
interface %[t](left: t, right: t): t

builtin ==(left: Float64, right: Float64): Bool
builtin ==(left: Int64, right: Int64): Bool
builtin ==(left: Int32, right: Int32): Bool
builtin ==(left: Int8, right: Int8): Bool
builtin ==(left: Bool, right: Bool): Bool

builtin !=(left: Float64, right: Float64): Bool
builtin !=(left: Int64, right: Int64): Bool
builtin !=(left: Int32, right: Int32): Bool
builtin !=(left: Int8, right: Int8): Bool

builtin <(left: Float64, right: Float64): Bool
builtin <(left: Int64, right: Int64): Bool
builtin <(left: Int32, right: Int32): Bool
builtin <(left: Int8, right: Int8): Bool

builtin <=(left: Float64, right: Float64): Bool
builtin <=(left: Int64, right: Int64): Bool
builtin <=(left: Int32, right: Int32): Bool
builtin <=(left: Int8, right: Int8): Bool

builtin >(left: Float64, right: Float64): Bool
builtin >(left: Int64, right: Int64): Bool
builtin >(left: Int32, right: Int32): Bool
builtin >(left: Int8, right: Int8): Bool

builtin >=(left: Float64, right: Float64): Bool
builtin >=(left: Int64, right: Int64): Bool
builtin >=(left: Int32, right: Int32): Bool
builtin >=(left: Int8, right: Int8): Bool

builtin +(left: Float64, right: Float64): Float64
builtin +(left: Int64, right: Int64): Int64
builtin +(left: Int32, right: Int32): Int32
builtin +(left: Int8, right: Int8): Int8

builtin -(left: Float64, right: Float64): Float64
builtin -(left: Int64, right: Int64): Int64
builtin -(left: Int32, right: Int32): Int32
builtin -(left: Int8, right: Int8): Int8

builtin -(number: Float64): Float64
builtin -(number: Int64): Int64
builtin -(number: Int32): Int32
builtin -(number: Int8): Int8

builtin *(left: Float64, right: Float64): Float64
builtin *(left: Int64, right: Int64): Int64
builtin *(left: Int32, right: Int32): Int32
```

```
builtin *(left: Int8, right: Int8): Int8

builtin /(left: Float64, right: Float64): Float64
builtin /(left: Int64, right: Int64): Int64
builtin /(left: Int32, right: Int32): Int32
builtin /(left: Int8, right: Int8): Int8

builtin %(left: Float64, right: Float64): Float64
builtin %(left: Int64, right: Int64): Int64
builtin %(left: Int32, right: Int32): Int32
builtin %(left: Int8, right: Int8): Int8

builtin not(boolean: Bool): Bool
builtin and(left: Bool, right: Bool): Bool
builtin or(left: Bool, right: Bool): Bool

builtin [][][t](array: Array[t], index: Int64): t
builtin [][][t](mut array: Array[t], index: Int64): mut t
```

Esto es la funcionalidad que está implementada en la biblioteca estándar hasta el momento, aunque si se quisiera, se podrían usar otras funciones de la biblioteca estándar de C de la misma forma que se usa `printf`, con la palabra clave `extern` seguido del prototipo de la función.

3.4. Extensión

Se implementó una extensión mínima para el lenguaje que provee *syntax highlighting* para Visual Studio Code. Se eligió este editor porque es uno de los más populares en la actualidad y que tiene menor curva de aprendizaje. Podemos ver la extensión en uso en la figura 11.

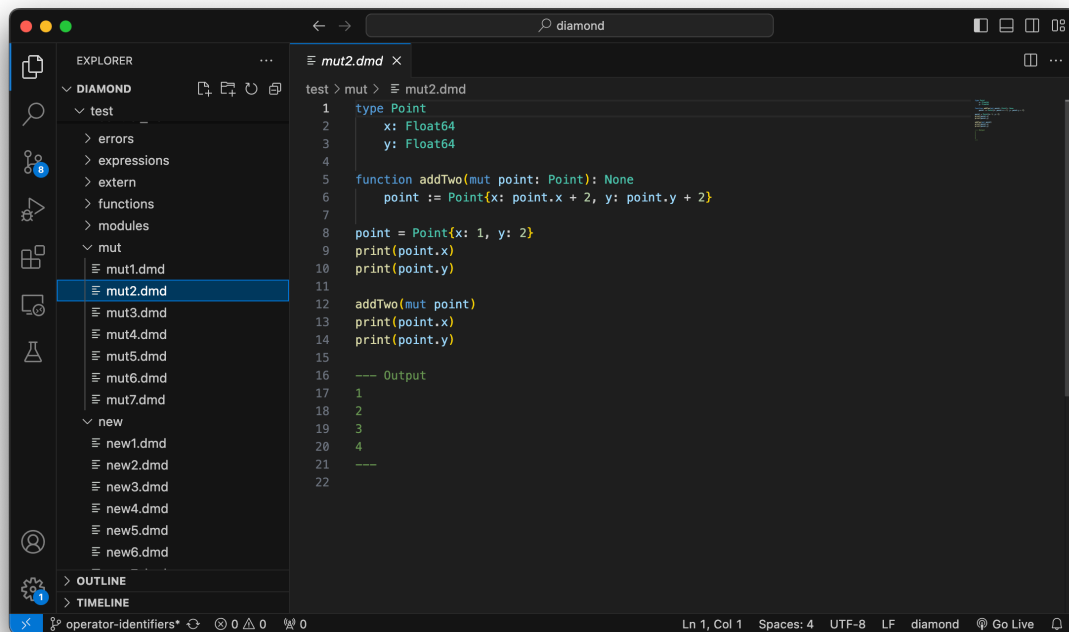


Figura 11: *Syntax highlighting* en Visual Studio Code de nuestro lenguaje

Fuente: Elaboración propia

Para crear la extensión se siguió la documentación code.visualstudio.com/api/language-extensions/syntax-highlight-guide. Esto significó crear una gramática de *TexMate*. Esta puede ser encontrada en el anexo.

Una vez logrado el funcionamiento deseado, el siguiente paso consistía en publicar la extensión. Para ello fue necesario crear una organización en Azure DevOps y luego obtener un *token* de acceso para poder publicar la extensión mediante *vsce*. Completados todos estos pasos, la extensión fue publicada en el *Marketplace* de *Visual Studio*. Se puede ver la extensión publicada en la figura 12.

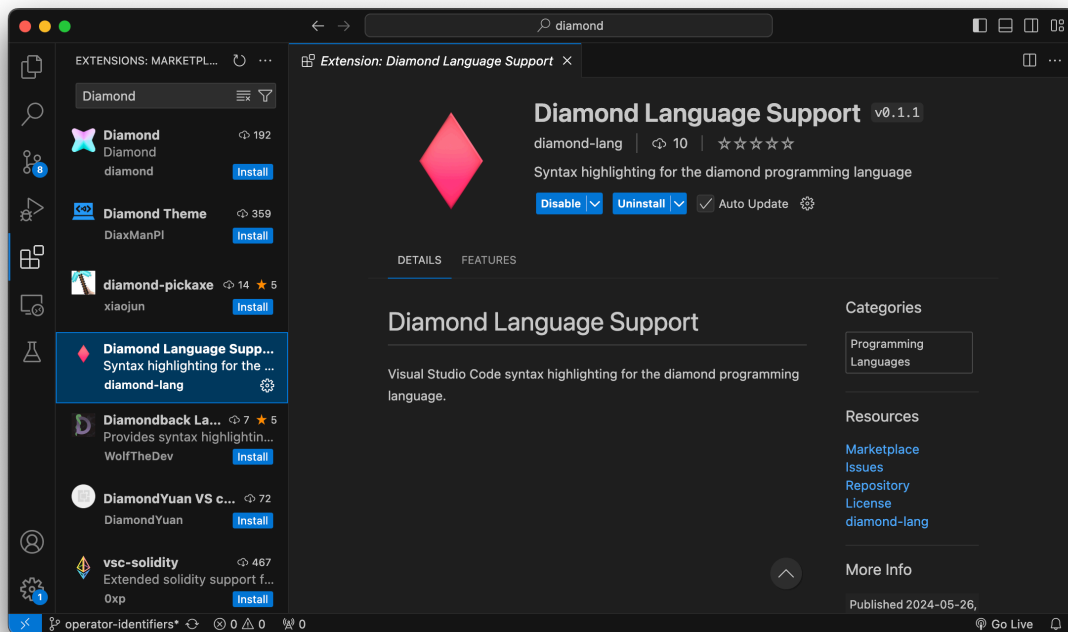


Figura 12: Vista de nuestra extensión en el Marketplace de Visual Studio Code

Fuente: Elaboración propia

3.5. Manual

Se escribió un manual para poder facilitar dar a conocer el lenguaje. Se utilizó GitBook para esto, pues es gratuito y hace fácil escribir y publicar documentación. El manual puede ser encontrado en amzamora.gitbook.io/diamond. El manual, a diferencia de la memoria, está escrito en inglés. Puede verse en la figura 13.

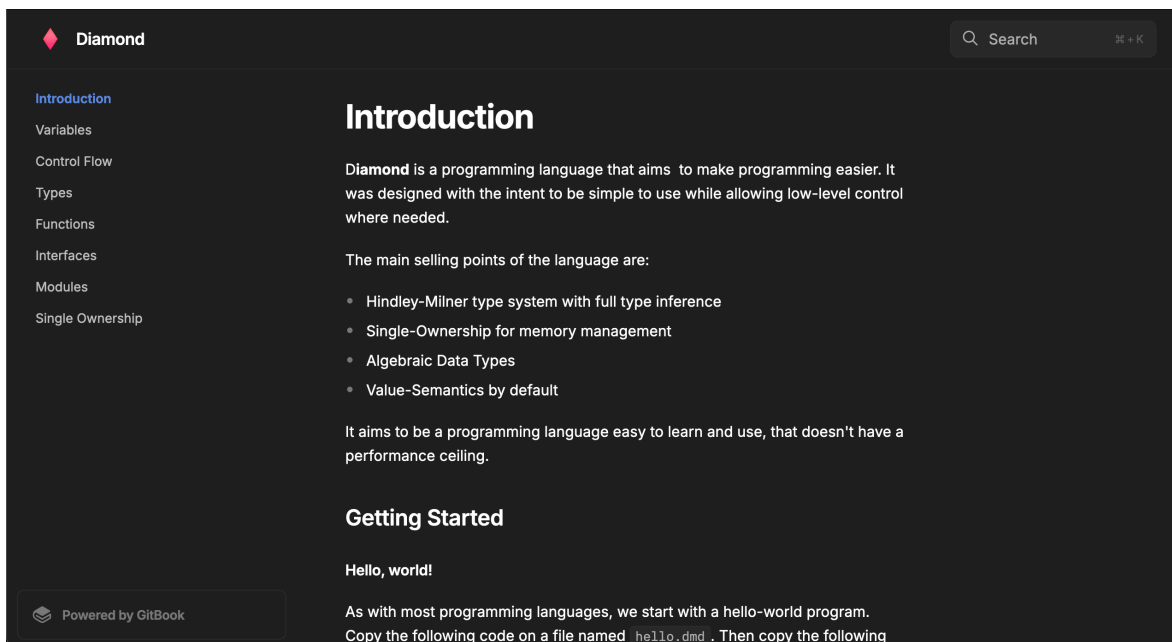


Figura 13: Manual del lenguaje
Fuente: Elaboración propia

Capítulo 4

Validación de la solución

Para validar la solución, mostraremos cómo el diseño del lenguaje cumple los objetivos propuestos. Además, proveeremos programas un poco más extensos que hasta el momento para mostrar cómo la estructura de un programa más complejo se vería y cómo las características del lenguaje interactuarían entre sí.

Los objetivos consistían en:

- *Objetivo 1:* Diseñar un lenguaje de programación con manejo de memoria predecible en tiempo de compilación.
- *Objetivo 2:* Diseñar un lenguaje de programación fácil de aprender.
- *Objetivo 3:* Diseñar un lenguaje fácil de usar. Lo cual puede ser desglosado como:
 - Limitar la fricción al programar.
 - Dificultar cometer errores.

Primero validaremos el objetivo 1. Luego, los objetivos 2 y 3, basándonos en el marco de trabajo Cognitive Dimensions Framework [Green y Petre, 1996].

4.1. Objetivo 1: Manejo de memoria predecible en tiempo de compilación

Este objetivo se cumple por diseño. El lenguaje no contaría con manejo de la memoria automático en tiempo de ejecución. En vez de esto, se utilizaría *Single Ownership*. La funcionalidad mínima fue implementada como prueba de concepto.

Por ejemplo, si compilamos el siguiente archivo llamado `new11.dmd` (un test del lenguaje):

```
type Point
  x: Boxed[Int64]
  y: Boxed[Int64]
  z: Boxed[Int64]

point = Point{x: new 100, y: new 200, z: new 300}
point = Point{x: new 1000, y: new 2000, z: new 3000}
print(*point.x)
print(*point.y)
print(*point.z)
```

Con el comando:

```
./diamond build new11.dmd
```

Y luego lo corremos con la herramienta Leaks, una herramienta disponible para macOS similar a Valgrind. Podemos notar que el programa pide memoria al sistema operativo y luego la libera correctamente.

```
leaks --atExit -- ./new11
```

Podemos ver el resultado en la figura 14.

```
alonso@MacBook-Air-de-Alonso diamond % leaks --atExit -- ./new11
new11(61633) MallocStackLogging: could not tag MSL-related memory as no_footprint, so those pages will be included in process footprint
new11(61633) MallocStackLogging: recording malloc and VM allocation stacks using lite mode
1000
2000
3000
Process 61633 is not debuggable. Due to security restrictions, leaks can only show or save contents of readonly memory of restricted processes.

Process:          new11 [61633]
Path:             /Users/USER/*/new11
Load Address:     0x104c3c000
Identifier:       new11
Version:          ???
Code Type:        ARM64
Platform:         macOS
Parent Process:   leaks [61632]

Date/Time:        2024-10-28 16:48:23.502 -0300
Launch Time:      2024-10-28 16:48:23.173 -0300
OS Version:       macOS 14.6.1 (23G93)
Report Version:   7
Analysis Tool:    /Applications/Xcode.app/Contents/Developer/usr/bin/leaks
Analysis Tool Version: Xcode 16.0 (16A242d)

Physical footprint: 3697K
Physical footprint (peak): 3697K
Idle exit:         untracked
-----

leaks Report Version: 4.0, multi-line stacks
Process 61633: 184 nodes malloced for 14 KB
Process 61633: 0 leaks for 0 total leaked bytes.

alonso@MacBook-Air-de-Alonso diamond %
```

Figura 14: Prueba de Single Ownership

Fuente: Elaboración propia

Lo interesante es que *Single Ownership* junto con *Value Semantics* harían el lenguaje fácil de aprender comparado con otros enfoques, ya que no sería necesario preocuparse del manejo de la memoria desde el principio.

4.2. Objetivo 2: Facilidad de aprender

4.2.1. Abstraction Gradient

Para limitar la cantidad de conceptos necesarios aprender al inicio, nuestro lenguaje no utiliza una función `main` como punto de entrada. Esto lo comparte con Python y con Swift. Conceptualmente, un programa es una lista de instrucciones. Es posible reducir aún más la complejidad inicial. Un ejemplo de esto es el lenguaje de programación Hedy [Hermans, 2020], pero este está diseñado específicamente para el aprendizaje. Hacer esto en un lenguaje de uso general significaría un costo grande en las otras dimensiones para los objetivos del lenguaje.

En cuanto al aprendizaje, el lenguaje puede ser aprendido en dos niveles. En primer lugar, como un lenguaje que solo trata con valores. En segundo, los elementos más avanzados necesarios para el desarrollo de estructuras de datos y relacionados con el manejo de la memoria y la vida de las variables. Podemos ver la dependencia entre los conceptos del lenguaje en la figura 15.

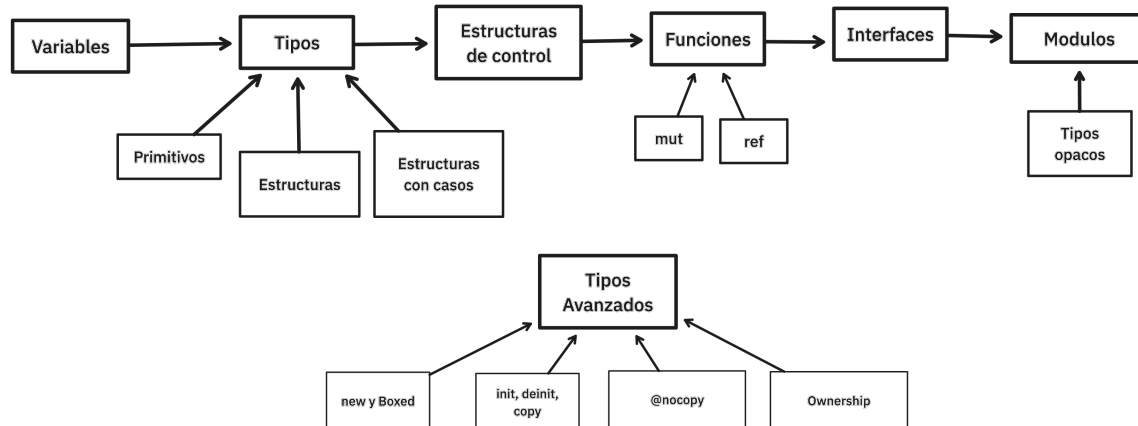


Figura 15: Dependencia entre conceptos para el aprendizaje de nuestro lenguaje

Fuente: Elaboración propia

El hecho de no contar con uso extendido de macros ayudaría también a hacer el lenguaje más fácil de aprender. Pues estos en general no siguen las reglas de las funciones y hacen difícil entender qué está haciendo el código.

4.2.2. Closeness of mapping

Esta es una dimensión difícil de evaluar, pues al ser un lenguaje de uso general no se busca priorizar *closeness of mapping* a un dominio específico. Aunque se espera que los elementos existentes sean suficientes para modelar sin mucho esfuerzo problemas de dominios diversos. Se espera que con tipos de datos algebraicos, interfaces y constructos de bajo nivel, una gran cantidad de problemas puedan ser modelados y resueltos en el lenguaje sin mucha carga cognitiva adicional a la del problema.

4.2.3. Consistency

Se buscó que la sintaxis del lenguaje tuviera un nivel de consistencia adecuado. Para esto, las palabras claves del lenguaje en general no son abreviaciones. Y cuando lo son es porque si no, la facilidad para hacer cambios, legibilidad y tersidad serían impactadas negativamente. Por otro lado, todos los tipos del lenguaje usan la convención Upper Case. Y las interfaces y variables de tipos usan lowercase. Al mismo tiempo, todos los tipos tienen la misma estructura: El nombre del tipo y si es que tiene, seguido de sus parámetros. Estas cosas harían fácil identificar que es un tipo, que es una interfaz, entre otros.

Si bien siempre se puede hacer un lenguaje más consistente y ortogonal en sus características, muchas veces esto no vale la pena. Generalmente, la consistencia siempre está en conflicto con *viscosity* y *role-expressiveness*.

Otra cosa, es interesante tomar en cuenta que usar *value semantics* por omisión tendría el beneficio que implicaría que la mayoría de los tipos se comportarían al igual que los primitivos. Este es un aspecto en que la consistencia haría más fácil de aprender el lenguaje.

Algo que ayuda mucho en esta dimensión es la existencia de una guía de estilos unificada para el lenguaje. Esta puede ser enforzada por un *formateador* de código. El lenguaje Go fue el primero en popularizar su uso. En la actualidad, la mayoría de los lenguajes de programación nuevos ofrecen esto.

4.2.4. Error-proneness

La mayoría de las características del lenguaje que ayudan a evitar errores ayudan a hacer el lenguaje más fácil de usar que de aprender. Hedy o Scratch lo hacen mejor en esta dimensión en cuanto al aprendizaje, pero estos son lenguajes pensados para el aprendizaje y para el uso general.

4.2.5. Hidden dependencies

Que no existan macros reduce las *hidden dependencies* desde la perspectiva de un principiante. Haciendo el lenguaje más fácil de aprender.

4.2.6. Premature commitment

El lenguaje hace bien en esta dimensión, pues se reducen al máximo las decisiones que el usuario debe tomar en un principio.

4.2.7. Progressive evaluation

Aún hay espacio que mejorar en esta dimensión. Hay varios caminos para esto. Pero esta es una característica sobre todo de la implementación y del *tooling* más que del lenguaje como tal. En un principio, se espera crear un servidor del lenguaje. Aunque también sería interesante la opción de desarrollar un intérprete para el lenguaje para poder ofrecer retroalimentación más rápidamente. Un punto medio es algo que se está probando en un lenguaje en desarrollo llamado Roc, en el cual, a pesar de ser compilado, permite ejecutar programas con errores de tipos hasta llegar a algún error de tipos, si es que lo hay.

4.2.8. Role-expressiveness

Se buscó crear una sintaxis intuitiva. Para esto, se prefieren palabras claves en vez de operadores. Esto hace más fácil reconocer los elementos si es que uno viene de otro lenguaje, o buscarlos si es que no es el caso.

4.2.9. Secondary notation

Esta dimensión corresponde a los canales de comunicación entre el lenguaje y el usuario que no son parte de la descripción formal del lenguaje. Se ha hecho mucho trabajo en producir errores de mensaje que sean comunicativos con el usuario, aunque aún hay mucho que mejorar.

4.2.10. Viscosity

Esta dimensión está más relacionada con el uso que con el aprendizaje. Pero igual, poder hacer cambios rápidamente ayuda a poder recibir *feedback* más rápidamente, lo cual es bueno cuando se está aprendiendo. El lenguaje busca reducir al máximo la viscosidad innecesaria gracias a la inferencia de tipos completa y la sintaxis para declarar y asignar variables.

4.3. Objetivo 3: Facilidad de uso

4.3.1. Closeness of mapping

Al ser un lenguaje sin una gran carga cognitiva, podría ser utilizado para una gran cantidad de problemas, sin salir del dominio del problema. Al mismo tiempo, se espera proveer *tooling* fácil de usar y una biblioteca estándar extensa, permitiendo llegar y resolver problemas en un gran ámbito de dominios.

4.3.2. Consistency

Todas las cosas que ya se dijeron respecto a la consistencia en la sección anterior aplicarían en esta. Pues muchas veces usar un lenguaje implica reaprender aspectos de este si es que ha pasado un tiempo desde que se trabajó con él.

4.3.3. Error-proneness

Hay varios elementos del lenguaje que ayudarían a evitar cometer errores. Estas características no importan mucho en programas pequeños, pero facilitan mucho la vida cuando se está trabajando en proyectos no triviales. Añaden seguridad y confianza al momento de hacer cambios en que el software siga funcionando correctamente.

Por un lado, tenemos un sistema de tipos estático inspirado en Hindley-Milner. Permite detectar muchos tipos de errores de manera automática en tiempo de compilación.

Una ventaja de un sistema Hindley-Milner, además, es que las funciones genéricas son chequeadas de manera general, no al momento de instanciarla. Esto permite producir mejores errores al usuario y al mismo tiempo saber que todos los caminos posibles en el código son chequeados.

Otras características que ayudan a no cometer errores son las que favorecen el razonamiento local. Como es el caso de tener sintaxis distinta para declarar y asignar, contar con *value*

semantics y hacer explícito cuando una función muta un argumento, y no contar con variables globales.

Asimismo, contar con *memory safety* por omisión también ayudaría a prevenir errores.

4.3.4. Hidden dependencies

En esta dimensión el lenguaje se desempeña bien, pues no cuenta ni con macros ni con variables globales. En general, las dependencias son bastante explícitas.

4.3.5. Premature commitment

El lenguaje hace un buen trabajo al momento de evitar decisiones prematuras gracias al uso de *value semantics* por omisión.

Otra característica que ayuda a esto es que solo existen funciones. De esta manera, no hay que decidir para cada procedimiento si es que este debiera ser una función o un método.

4.3.6. Progressive evaluation

Las mismas características que ya se hablaron en la sección anterior ayudarían de igual manera a hacer el lenguaje más fácil de usar. Poder iterar rápidamente no solo sirve para aprender más rápido, sino que también para avanzar más rápido.

4.3.7. Secondary notation

Contar con mensajes de error comunicativos y que muestren claramente dónde está el problema también ayudaría a ser más productivo.

4.3.8. Role-expressiveness

Esta dimensión importa menos en cuanto a la facilidad de uso, puesto que cuando uno ya es familiar con una notación, leerla no debería ser un problema. De todas maneras, siempre es útil que la notación sea intuitiva, pues a veces pasa tiempo sin usar un lenguaje.

4.3.9. Viscosity

El usar un sistema de tipos con inferencia de tipos completa significa que no es necesario definir todos los tipos de antemano. Esto permite iterar rápidamente.

Por otro lado, la sintaxis de declaración y asignación ayudaría a poder mover código de una sección a otra sin tener que hacer muchos cambios. También, en cuanto a sintaxis, no tener que utilizar comas en llamadas y estructuras que ocupan múltiples líneas para separar los elementos ayudaría a poder hacer cambios más fácilmente.

4.4. Resumen análisis de dimensiones

En la tabla 2 podemos ver el resumen del análisis de dimensiones realizado en las secciones anteriores.

Tabla 2: Resumen de análisis de dimensiones

Fuente: Elaboración propia.

Abstraction gradient	Baja curva de aprendizaje para un lenguaje de uso general. El lenguaje de por sí no favorece la abstracción. Aunque un poco más que un lenguaje como Elm o C.	Premature commitment	No se fuerza la toma de decisiones de forma prematura gracias a value-semantics y el sistema de tipos.
Closeness of mapping	Buen nivel de closeness of mapping para un lenguaje de uso general. Se espera mejorar esta dimensión con la creación de bibliotecas.	Progressive evaluation	Aún falta por mejorar en esta dimensión. Se espera crear un intérprete o un servidor de lenguaje para esto.
Consistency	Buen balance entre consistencia y el resto de las dimensiones. Se mantiene la consistencia cuando se puede y se rompe si es que significa una mejora significativa.	Role-expressiveness	Buena legibilidad de sintaxis gracias al uso de palabras completas. Sus características son fáciles de buscar gracias a esto.
Error-proneness	Se dificulta cometer errores gracias a la presencia de un sistema de tipos estático y reglas de ownership.	Secondary notation	Aún falta por mejorar en esta dimensión. Se espera mejorar errores y proveer alguna manera de generar documentación fácilmente.
Hidden dependencies	Fácil razonar localmente gracias al sistema de tipos estático y el resto de características que facilitan esto, como las distintas formas de pasar parámetros.	Viscosity	Fácil hacer cambios gracias a inferencia de tipos completa y sintaxis que hace fácil mover código de un lado para otro.

4.5. Ejemplos en el lenguaje

A continuación, veremos ejemplos de cómo se verían programas más complejos en nuestro lenguaje. Y cómo distintos conceptos podrían ser modelados.

4.5.1. Juego de Pong

Utilizando la biblioteca de raylib para la creación de videojuegos, así es como se vería una implementación del videojuego de Pong en nuestro lenguaje:

```
use raylib

-- Types
type Ball
  x: Int64
  y: Int64
  dx: Int64
  dy: Int64
  size: Int64

type Paddle
  x: Int64
  y: Int64
  width: Int64
  height: Int64

-- Constants
windowWidth is 800
windowHeight is 600
paddleSpeed is 10

-- Initialize the game window
raylib::initWindow(windowWidth, windowHeight, "classic game: pong")
raylib::setTargetFPS(60)

-- Set up game objects
ball = Ball{x: 400, y: 300, dx: 5, dy: 5, size: 10}
paddle1 = Paddle{x: 50, y: 250, width: 20, height: 130}
paddle2 = Paddle{x: 740, y: 250, width: 20, height: 130}
score1 = 0
score2 = 0

-- Main loop
while not raylib::windowShouldClose()
  raylib::beginDrawing()
  raylib::clearBackground(raylib::rayWhite)

  raylib::drawCircle(ball.x, ball.y, ball.size, raylib::red)
  raylib::drawRectangle(
```

```
    paddle1.x
    paddle1.y
    paddle1.width
    paddle1.height
    raylib::black
)
raylib::drawRectangle(
    paddle2.x
    paddle2.y
    paddle2.width
    paddle2.height
    raylib::black
)

-- Handle user input
if raylib::isKeyDown("W")
    paddle1.y -= paddleSpeed

if raylib::isKeyDown("S")
    paddle1.y += paddleSpeed

if raylib::isKeyDown(raylib::KeyUp)
    paddle2.y -= paddleSpeed

if raylib::isKeyDown(raylib::KeyDown)
    paddle2.y += paddleSpeed

-- Move the ball and check for collisions with walls and paddles
ball.x += ball.dx
ball.y += ball.dy

if ball.x < 0
    score2 += 1
    reset(mut ball)

else ball.x > windowHeight
    score1 += 1
    reset(mut ball)

if ball.y - ball.size < 0 or ball.y + ball.size > windowHeight
    ball.dy = -ball.dy

if intersects(ball, paddle1) or intersects(ball, paddle2)
    ball.dx = -ball.dx

-- Check for game over condition
if score1 >= 10 or score2 >= 10
    displayGameOver()
    break

-- Display current scores
displayScores(score1, score2)
```

```
-- Draw objects
raylib::endDrawing()

raylib::closeWindow()

-- Helper functions
function reset(mut ball: Ball): None
    ball.x = 400
    ball.y = 300
    ball.dx = -ball.dx
    ball.dy = raylib::getRandomValue(0, 10) - 5

function intersects(ball: Ball, paddle: Paddle): Bool
    return
        not ball.x + ball.size < paddle.x
        or  ball.x > paddle.x + paddle.width
        or  ball.y + ball.size < paddle.y
        or  ball.y > paddle.y + paddle.height

function displayScores(score1: Int64, score2: Int64): None
    player1 be "Player 1: {score1}"
    player2 be "Player 2: {score2}"
    raylib::drawText(player1, 10, 20, 20, raylib::gray)
    raylib::drawText(
        player2
        windowHeight - raylib::measureText(player2, 20) - 10
        20
        20
        raylib::gray
    )

function displayGameOver(): None
    raylib::drawText("Game Over", 350, 300, 20, raylib::gray)
```

En este ejemplo podríamos observar cómo podría ser estructurado un programa, y cómo funcionaría pasar parámetros por referencia mutable.

4.5.2. Implementación de un arreglo dinámico

En el siguiente ejemplo podríamos ver uso de características más avanzadas de lenguaje. Necesarias para implementar estructuras de datos, como lo es un arreglo dinámico.

En el código a continuación se implementa un arreglo dinámico, con dos constructores y las operaciones [], append, pop y find.

```
-- List.dmd
use memory

initialCapacity is 16
```

```
type List[t]
  size: Index
  elements: Boxed[Array[t]]|None

init List[t]()
  list.elements = None

init List[t](elements: Array[t])
  this.size = size(elements)
  this.elements = new Array{size(elements)}[t]

  for i in 1...size(elements)
    (*this.elements)[i] = elements[i]

copy List[t: copy](ref list: List[t]): List[t]
  newList = List()
  for item in list
    append(mut newList, item)
  return newList

function [][t](mut list: List[t], index: Int64): mut t
  assert(this.elements isnt None)
  assert(1 <= index and index <= list.size)
  return (*this.elements)[index]

function append[t](mut list: List[t], value: t): None
  if list.elements is None
    list.elements = new Array{initialCapacity}[t]

  else this.size + 1 > size(*list.elements)
    newBuffer = new Array{size(*list.elements) * 2}[t]
    memory::realloc(mut list.elements, newSize: size(*list.elements) * 2)

    (*list.elements)[list.size + 1] = value
    list.size += 1

function append[t](list: List[t], value: t): List[t]
  append(mut list, value)
  return list

function pop[t](mut list: List[t]): None
  if list.size > 0
    list.size -= 1

function find[t](ref list: List[t], value: t): Int64|None
  for i in 1...(list.size)
    if value == (*list.elements)[i]
      return i

  return None
```

Luego, el tipo podría ser usado en otro archivo la siguiente manera:

```
-- main.dmd
use list (List)

a = List([1, 2, 3])
  .append(4)
  .append(5)

a = List([1, 2, 3])
a.append(4)
a.append(5)
```

Notamos que, gracias al uso de opaque la implementación sería inaccesible desde archivos que no fueran `List.dmd`.

También que sería posible sobrecargar una función cuando se pasa por referencia mutable. Y cuál se usaría dependería de cómo es llamada.

Por último, sería interesante que la estructura de datos podría funcionar con tipos que implementan copy y que no implementan copy. Solo sería necesario que sus elementos implementen copy si es que se quiere copiar la lista.

Capítulo 5

Conclusiones

5.1. Conclusiones generales

En esta memoria se llevó a cabo el diseño e implementación de un nuevo lenguaje de programación, con el objetivo de producir un lenguaje que fuera fácil de aprender y de usar y que no tuviera un tope en cuanto rendimiento. De esto resultó Diamond.

Se logró producir un diseño coherente utilizando *value semantics* y *single ownership* en el núcleo del lenguaje. También se logró exitosamente desarrollar un compilador con manejo de la memoria en tiempo de compilación y inferencia de tipos completa. Y utilizando LLVM como *backend*. Gracias a esto, es posible generar código con optimizaciones para todas las plataformas que éste soporta.

Se crearon *tests* para el compilador y se utilizó integración continua exitosamente durante el desarrollo para evitar regresiones. Gracias a esto, se aseguró con esto el correcto funcionamiento del compilador en las plataformas Windows, Linux y macOS.

Se produjo de manera satisfactoria una biblioteca estándar mínima. Se desarrolló y publicó una extensión para Visual Studio Code. Y se escribió un manual para el lenguaje utilizando GitBook.

Este proceso fue un periodo de constante aprendizaje. Hay muchos lenguajes de programación nuevos en la actualidad, y aún no está claro cuáles perduran en el tiempo. Si bien es verdad que casi todo se puede hacer en cualquier lenguaje, estos influyen de manera significativa cómo pensamos y resolvemos problemas. También influyen en las garantías con las que contamos. Es una de las razones por las que Rust se está haciendo cada vez más popular. El espacio de búsqueda es inmenso y vale la pena explorarlo.

5.2. Cumplimiento de objetivos

Para esta memoria, se propusieron 5 objetivos. Estos fueron desarrollados y cumplidos a lo largo de la memoria.

1. *Definir elementos con los que deba contar el lenguaje de programación para que sea fácil de aprender y de usar, de uso general, que no tenga un tope en cuanto rendimiento.*

Se definieron los elementos del lenguaje. Las principales características resultaron ser:

- Manejo de la memoria en tiempo de compilación basado en *Single Ownership*
- Tipificado estático
- Anotaciones de tipos opcionales con inferencia de tipos completa
- Tipos de datos algebraicos
- Programación genérica mediante interfaces
- Paso de parámetros por valor de manera predeterminada con formas alternativas de pasar parámetros

Este diseño luego fue validado usando *Cognitive Dimensions Framework*.

2. *Desarrollar un compilador para el lenguaje con una interfaz inspirada en el compilador de Elm.*

Se desarrolló un compilador para el lenguaje usando C++ para este y LLVM como backend. La interfaz del compilador fue inspirada en la del compilador de Elm.

3. *Implementar una versión mínima de la biblioteca estándar.*

Se implementó una versión mínima de la biblioteca estándar en nuestro lenguaje utilizando interfaces para modelar los operadores y la salida de los programas.

4. *Proporcionar una extensión para una plataforma de desarrollo de popular.*

Se desarrolló una extensión para Visual Studio Code que provee *syntax highlighting* para el lenguaje.

5. *Facilitar la divulgación del lenguaje mediante un manual para este.*

Se escribió un manual para el lenguaje usando GitBook en inglés para poder comunicar de mejor manera las características del lenguaje.

Gracias a esto podemos decir que el objetivo general de *Diseñar y desarrollar un lenguaje fácil de aprender y de usar de uso general que no tenga un tope en cuanto rendimiento* se cumplió.

5.3. Trabajo futuro

Si bien se logró producir un compilador para una parte significativa del lenguaje, aún quedan características por implementar. Siendo la más notoria estructuras con casos. Se espera implementar las características faltantes y terminar de implementar la ABI correctamente. Una vez hecho esto, el lenguaje podría ser utilizado sin problemas para desarrollar videojuegos.

A continuación, se espera comenzar a implementar una biblioteca estándar más completa, y dejar de depender de C. Luego, lo siguiente sería pulir la experiencia de usuario. Es decir,

mejorar los errores, arreglar el compilador para que produzca código compatible con GDB y desarrollar un servidor para el lenguaje.

En cuanto al diseño, aún quedan desafíos interesantes, como cuál sería un buen modelo de concurrencia y paralelismo para el lenguaje. Y también, cuál sería un buen modelo para SIMD.

Un desafío de diseño e implementación sería definir el modelo apropiado para *memory safety* de los descritos en la sección 3.1.10.2.

Anexos

5.1. Gramática de Diamond

```
program = block;

block = statement*;

statement = function
    | interface
    | builtin
    | extern
    | link_with
    | type_definition
    | return
    | if_else
    | while
    | break
    | continue
    | use
    | dereference_assignment
    | declaration
    | assignment
    | call
    | field_assignment
    | index_assignment;

block_or_statement = statement
    | "\n"+ block;

block_or_statement_or_expression = "\n"+ block
    | "\n"+ expression
    | expression
    | statement;

type_parameters = "[" identifier (":" interface_type)? ("," identifier (":"
    interface_type)?)? "]";

function_argument = "mut"? identifier;

function = "function" identifier type_parameters? "(" (function_argument (":" type)
    ? ",")* ")" (":" type)? block_statement_or_expression;

interface = "interface" identifier type_parameters "(" (function_argument ":" type)
    ",")* ")" ":" type;

builtin = "builtin" identifier type_parameters? "(" (function_argument ":" type)
    ",")* ")" ":" type;
```

```
extern = "extern" identifier "(" (identifier ":" type "..."? ("," IDENTIFIER ":" type "...")*)? ")" ":" type;

link_with = "link_with" string;

type_definition = "type" identifier ("\n"+ identifier ":" type)*;

type = identifier ("[" type ("," type)* "]"*)*;

declaration = identifier ("be"|"=") expression (":" type)?;

assignment = identifier "!=" expression (":" type)?;

field_assignment = field_assignment "=" expression (":" type)?;

dereference_assignment = dereference "=" expression (":" type)?;

index_assignment = index_access "=" expression (":" type);

return = "return" expression?;

break = "break";

continue = "continue";

if_else = "if" expression block_or_statement ("\n"* "else" block_or_statement);

while = "while" expression block;

use = ("use"|"include") string;

call_argument = "mut"? identifier ":" expression
                | "mut"? expression;

call = identifier "(" (call_argument (("","|("\n"+)) call_argument)*)* ")";

struct = identifier "{" (identifier ":" expression (","|("\n"+)))* "}"

expression = if_else_expression
            | new
            | not
            | binary;

if_else_expression = "if" expression "\n"* expression ("else" "\n"* expression)?;

new = "new" expression;

not = "not" expression;

binary = equality;
equality = comparison ("=="|"!=") comparison)*;
comparison = term (">"|">="|"<"|"<=") term)*;
term = factor ("+"|"-" ) factor)*;
```

```
factor = primary (("*"|"/"|"%" ) primary)*;

primary = negation
        | dereference
        | address_of
        | grouping
        | array
        | float
        | integer
        | boolean
        | string
        | interpolated_string
        | call
        | struct
        | grouping_or_assignable;

grouping_or_assignable = grouping
                        | identifier
                        | index_access
                        | field_access
                        | call;

negation = "-" primary;

address_of = "&" (field_access|identifier|index_access);

dereference = "*" (dereference|assignable);

grouping = "(" expression ")";

float = FLOAT;

integer = INTEGER;

boolean = TRUE|FALSE;

identifier = IDENTIFIER;

string = STRING;

interpolated_string = STRING_LEFT expression (STRING_MIDDLE expression)*
                     STRING_RIGHT;

array = "[" (expression ("," expression)*)* "]";

field_access = assignable "." identifier ( "." identifier)*;

index_access = assignable "[" expression "]";
```

5.2. Gramática para la extensión de Visual Studio Code

```
{
  "$schema": "https://raw.githubusercontent.com/martinring/tmlanguage/master/
    tmlanguage.json",
  "name": "diamond",
  "patterns": [
    {
      "include": "#comments"
    },
    {
      "include": "#strings"
    },
    {
      "include": "#constants"
    },
    {
      "include": "#functions"
    },
    {
      "include": "#keywords"
    },
    {
      "include": "#types"
    },
    {
      "include": "#typeField"
    },
    {
      "include": "#operators"
    },
    {
      "include": "#variables"
    }
  ],
  "repository": {
    "comments": {
      "patterns": [
        {
          "name": "comment.block.diamond",
          "begin": "---",
          "end": "---"
        },
        {
          "name": "comment.line.diamond",
          "match": "--.*"
        }
      ]
    },
    "keywords": {
      "name": "keyword.diamond",
      "patterns": [
```

```
{
  "name": "keyword.control.diamond",
  "match": "\\b(if|else|return|while)\\b"
},
{
  "name": "keyword.operator.diamond",
  "match": "\\b(not|or|and)\\b"
},
{
  "name": "keyword.other.diamond",
  "match": "\\b(mut|new)\\b"
}
]
},
"strings": {
  "name": "string.quoted.double.diamond",
  "begin": "\"",
  "end": "\"",
  "patterns": [
    {
      "name": "constant.character.escape.diamond",
      "match": "\\\\"
    }
  ]
},
"constants": {
  "name": "constant.diamond",
  "patterns": [
    {
      "name": "constant.numeric.diamond",
      "match": "\\b([0-9]*[.]?[0-9]+)\\b"
    },
    {
      "name": "constant.language.diamond",
      "match": "\\b(true|false)\\b"
    }
  ]
},
"functions": {
  "patterns": [
    {
      "begin": "(function|interface|builtin|extern)\\s+([a-zA-Z_][a-zA-Z0-9_]*|\\+|-|\\*|=|!|=|<|=|>|=|>|\\*|\\/|%|\\[\\|\\|\\|)\\s*(\\[.*\\])?\\s*\\(",
      "end": "\\)(\\s*(mut)?\\s*(.*)?)?",
      "beginCaptures": {
        "1": {
          "name": "storage.type.function.diamond"
        },
        "2": {
          "name": "entity.name.function.diamond"
        },
        "3": {
          "patterns": [

```

```
        {
          "name": "entity.name.type.diamond",
          "match": "[a-zA-Z_][a-zA-Z0-9_]*"
        }
      ]
    },
    "endCaptures": {
      "2": {
        "name": "keyword.other.diamond"
      },
      "3": {
        "name": "entity.name.type.diamond"
      }
    },
    "patterns": [
      {
        "name": "keyword.other.diamond",
        "match": "\\b(mut)\\b"
      },
      {
        "match": "([a-zA-Z_][a-zA-Z0-9_]*)\\s*(\\:)?\\s*([a-zA-Z_][a-zA-Z0-9_]*)?",
        "captures": {
          "1": {
            "name": "variable.parameter.diamond"
          },
          "2": {
            "name": "keyword.operator.key-value.diamond"
          },
          "3": {
            "name": "entity.name.type.diamond"
          }
        }
      }
    ]
  },
  {
    "match": "\\b(function|interface|builtin|extern)\\s+([a-zA-Z_][a-zA-Z0-9_]*)?",
    "captures": {
      "1": {
        "name": "storage.type.function.diamond"
      },
      "2": {
        "name": "entity.name.function.diamond"
      }
    }
  }
],
"types": {
  "match": "(type)\\s+([A-Z][a-zA-Z0-9_]*)?",
```

```
"captures": {
  "1": {
    "name": "storage.type.diamond"
  },
  "2": {
    "name": "entity.name.type.diamond"
  }
},
"typeField": {
  "match": "^(?:|\\t)+([a-zA-Z_][a-zA-Z0-9_]*\\s*:\\s*)([a-zA-Z_][a-zA-Z0-9_]*)",
  "captures": {
    "1": {
      "name": "variable.parameter.diamond"
    },
    "2": {
      "name": "entity.name.type.diamond"
    }
  }
},
"operators": {
  "name": "keyword.operator.diamond",
  "patterns": [
    {
      "name": "keyword.operator.comparison.diamond",
      "match": "(==|!=|<=<|>=>)"
    },
    {
      "name": "keyword.operator.arithmetic.diamond",
      "match": "(\\+|\\*|\\-|/)"
    },
    {
      "name": "keyword.operator.assignment.diamond",
      "match": "(=|:=|be)"
    },
    {
      "name": "keyword.operator.key-value.diamond",
      "match": ":"
    }
  ]
},
"variables": {
  "patterns": [
    {
      "name": "entity.name.type.diamond",
      "match": "[A-Z][a-zA-Z0-9_]*"
    },
    {
      "name": "entity.name.function",
      "match": "[a-zA-Z_][a-zA-Z0-9_]*\\("
    }
  ]
}
```

```
        "name": "variable.other.diamond",  
        "match": "[a-zA-Z_][a-zA-Z0-9_]*"  
    }  
  ]  
}  
,  
"scopeName": "source.diamond"  
}
```

Referencias bibliográficas

- [Abidi et al., 2021] Abidi, M., Rahman, M. S., Openja, M., y Khomh, F. (2021). Are multi-language design smells fault-prone? an empirical study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(3):1–56.
- [Atwood, 2008] Atwood, J. (2008). Hardware is cheap, programmers are expensive.
- [Borretti, 2023] Borretti, F. (2023). Introducing austral: A systems language with linear types and capabilities.
- [Bright et al., 2020] Bright, W., Alexandrescu, A., y Parker, M. (2020). Origins of the d programming language. *Proceedings of the ACM on Programming Languages*, 4(HOPL):1–38.
- [Clarke, 2001] Clarke, S. (2001). Evaluating a new programming language. En *PPIG*, volumen 13, pp. 275–289.
- [Czaplicki, 2015] Czaplicki, E. (2015). Compiler errors for humans.
- [Ferdowsi, 2023] Ferdowsi, K. (2023). The usability of advanced type systems: Rust as a case study. *arXiv preprint arXiv:2301.02308*.
- [García et al., 2015] García, C. G., Espada, J. P., Bustelo, B. C. P. G., y Lovelle, J. M. C. (2015). Swift vs. objective-c: A new programming language. *IJIMAI*, 3(3):74–81.
- [Go, 2021] Go (2021). What is the history of the project? <https://go.dev/doc/faq#history> [Accessed: 28/05/2024].
- [Green y Petre, 1996] Green, T. R. G. y Petre, M. (1996). Usability analysis of visual programming environments: a ‘cognitive dimensions’ framework. *Journal of Visual Languages & Computing*, 7(2):131–174.
- [Gugerty y Olson, 1986] Gugerty, L. y Olson, G. M. (1986). Comprehension differences in debugging by skilled and novice programmers. En *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pp. 13–27.
- [Hermans, 2020] Hermans, F. (2020). Hedy: a gradual language for programming education. En *Proceedings of the 2020 ACM conference on international computing education research*, pp. 259–270.
- [1969] Hindley, Roger and others (1969). The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60.
- [Jeff Bezanson, 2012] Jeff Bezanson, Stefan Karpinski, V. B. S. A. E. (2012). <https://julialang.org/blog/2012/02/why-we-created-julia/>. <https://github.com/mtrebi/memory-allocators> [Accessed: 24/05/2024].

- [Keleshev, 2020] Keleshev, V. (2020). *Compiling to Assembly from Scratch*.
- [Kernighan y Ritchie, 2002] Kernighan, B. W. y Ritchie, D. M. (2002). The c programming language.
- [Klabnik, 2015] Klabnik, S. (2015). The language strangeness budget. <https://steveklabnik.com/writing/the-language-strangeness-budget/> [Accessed 14/11/2024].
- [Klabnik y Nichols, 2022] Klabnik, S. y Nichols, C. (2022). *The Rust Programming Language*. No Starch Press.
- [Kladov, 2024] Kladov, A. (2024). Modern Systems Programming: Rust and Zig.
- [Li et al., 2022] Li, W., Li, L., y Cai, H. (2022). On the vulnerability proneness of multilingual code. En *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 847–859.
- [Liaw, 2022] Liaw, J. (2022). A Tour of the C++ Engine which Powers That Design MMO Called Figma.
- [Mayer et al., 2017] Mayer, P., Kirsch, M., y Le, M. A. (2017). On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. *Journal of Software Engineering Research and Development*, 5:1–33.
- [McCarthy, 1960] McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195.
- [Microsoft, 2023] Microsoft (2023). What is "managed code"? <https://learn.microsoft.com/en-us/dotnet/standard/managed-code> [Accessed 14/11/2024].
- [Milner, 1978] Milner, R. (1978). A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375.
- [Neitsch et al., 2012] Neitsch, A., Wong, K., y Godfrey, M. W. (2012). Build system issues in multilanguage software. En *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pp. 140–149. IEEE.
- [Nystrom, 2021] Nystrom, R. (2021). *Crafting interpreters*. Genever Benning.
- [Odin, 2024] Odin (2024). Odin Allocators. <https://odin-lang.org/docs/overview/#allocators> [Accessed: 24/05/2024].
- [Pellet et al., 2019] Pellet, J.-P., Dame, A., y Parriaux, G. (2019). How beginner-friendly is a programming language? a short analysis based on java and python examples.
- [Pereira et al., 2017] Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., y Saraiva, J. (2017). Energy efficiency across programming languages: how do energy, time, and memory relate? En *Proceedings of the 10th ACM SIGPLAN international conference on software language engineering*, pp. 256–267.

- [Ritchie, 1993] Ritchie, D. M. (1993). The development of the c language. *ACM Sigplan Notices*, 28(3):201–208.
- [Scaife y Taylor, 1990] Scaife, M. y Taylor, J. (1990). Graduated learning environments for developing computational concepts in 7-11 year old children. *Journal of Interactive Learning Research*, 2(2):31.
- [SlashData, 2019] SlashData (2019). The global developer population 2019.
- [SlashData, 2022] SlashData (2022). State of the developer nation 23rd edition.
- [Stefik y Siebert, 2013] Stefik, A. y Siebert, S. (2013). An empirical investigation into programming language syntax. *ACM Trans. Comput. Educ.*, 13(4).
- [Storopoli et al., 2021] Storopoli, J., Huijzer, R., y Alonso, L. (2021). *Julia Data Science*.
- [Stroustrup, 1999] Stroustrup, B. (1999). An overview of the c++ programming language. *Handbook of object technology*, p. 72.
- [Teodorescu, 2023] Teodorescu, L. R. (2023). Value-Oriented Programming.
- [The White House, 2024] The White House (2024). Press Release: Future Software Should Be Memory Safe. <https://www.whitehouse.gov/oncd/briefing-room/2024/02/26/press-release-technical-report/> [Accessed 23/10/2024].
- [Trebino, 2016] Trebino, M. (2016). Custom memory allocators in C++ to improve the performance of dynamic memory allocation. <https://github.com/mtrebi/memory-allocators> [Accessed: 24/05/2024].
- [Wadler, 1990] Wadler, P. (1990). Linear types can change the world! En *Programming concepts and methods*, volumen 3, p. 5. Citeseer.
- [Zig, 2017] Zig (2017). Zig 0.1.1 release. <https://github.com/ziglang/zig/releases/tag/0.1.1> [Accessed: 20/05/2024].
- [Zig, 2024a] Zig (2024a). Why Zig when there is already C++, D, and Rust? https://ziglang.org/learn/why_zig_rust_d_cpp/ [Accessed: 21/05/2024].
- [Zig, 2024b] Zig (2024b). Zig Allocators. <https://zig.guide/standard-library/allocators/> [Accessed: 24/05/2024].