

Bachelor

December 27, 2024

---

# Dash

Design and implementation of a data-oriented programming language solving complexities and runtime faults found in other languages

**Henrik Nordgren**

of Küsnacht, Switzerland (20-934-592)

**supervised by**

Prof. Dr. Harald C. Gall  
Assistant



University of  
Zurich <sup>UZH</sup>





Bachelor

---

# Dash

Design and implementation of a data-oriented programming language solving complexities and runtime faults found in other languages

**Henrik Nordgren**



University of  
Zurich<sup>UZH</sup>



**Bachelor**

**Author:** Henrik Nordgren, [carlhenrikklaes.nordgren@uzh.ch](mailto:carlhenrikklaes.nordgren@uzh.ch)

**URL:** <https://github.com/chk-n/dash>

**Project period:** 28.06.2024 - 28.12.2024

Software Evolution & Architecture Lab  
Department of Informatics, University of Zurich

---

# Acknowledgements

I would like to thank my supervisor, Dr. sc. Carol Alexandru, for providing me with his support and letting me explore this new and interesting area of programming language design and compiler implementation. I would also like to thank Marius Urban and the team at Vărdex Suisse AG for allowing me the flexibility to focus on my bachelor thesis, especially in the last couple of months. This project would have been a lot more stressful without their understanding. Finally, I want to thank Victoria for her unwavering support and putting up with me working on this thesis at all sorts of odd hours for the past 6 months.



---

# Abstract

This thesis explores the design and implementation of Dash, a novel programming language based on data-oriented programming (DOP) principles, in order to prevent common software faults while preserving simplicity, reliability and performance. The DOP paradigm enforces immutability and generality for the sake of simplicity and reusability, while attempting to mitigate performance impacts. Moreover, DOP is considered a language agnostic paradigm with many implementations in other programming languages, creating an opportunity to explore how it could be implemented at the language level.

The thesis examines software vulnerabilities listed in the Common Weakness Enumeration (CWE) database in order to determine common software faults. By combining DOP principles with well-known safety mechanisms Dash implements language features like generic structural types, type predicates, explicit error handling and immutable data structures. The feasibility of DOP at the language level is demonstrated through a functioning compiler in Go using LLVM as the backend that implements the DOP principles as core language constructs. Moreover, the language's effectiveness is evaluated against Golang and Rust by implementing an arithmetic expression evaluator and comparing key aspects, such as state management, type system capabilities, safety guarantees, and code readability.

The thesis finds that certain modifications of the original principles are required to successfully implement DOP at the language level. Most notably, the language implements generic structs with compile-time type checking to preserve flexibility and type safety, rather than using hash maps with string-based field references as prescribed by DOP. Additionally, because concrete types are required for compilation, the separation between data schema and data representation becomes more closely coupled at the language level. However, this is lessened by features like type predicates, optional types and union types. Furthermore, in order to handle program state transitions while preserving immutability, the language uses atomic compare-and-swap operations instead of the conventional stateful class approach outlined in DOP.

This thesis reiterates how important careful language design is in preventing common software vulnerabilities. Dash's type system eliminates common issues like null pointer dereferences, type confusion and unhandled errors through compile-time checks and runtime guards. Integer overflow vulnerabilities are prevented through arbitrary precision number types, while string manipulation errors are mitigated by maintaining explicit length information rather than relying on null termination. Memory safety is enhanced through immutable data structures, reference counting and runtime checks, preventing faults such as use-after-free, expired pointer dereference and buffer overflow vulnerabilities.

The evaluation shows that Dash's immutable state management and explicit error handling provide stronger safety guarantees than Go while being simpler than Rust's ownership system. While Dash's approach results in slightly more verbose code compared to Go and Rust, its type system combines Go's simplicity with additional safety features like type predicates and null

safety, making it suitable for implementing simple and robust software systems.

Further research is needed to thoroughly validate the language's design decisions and safety guarantees in the real world that go beyond the arithmetic evaluator. Dash's performance characteristics, especially in relation to memory management overhead and the efficiency of its error handling system in preventing runtime crashes, need to be empirically compared with other similar languages. Moreover, unimplemented parts of the language specification and areas such as memory management, concurrency, performance optimisations and developer tooling need to be implemented to make Dash production ready.

---

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Background and Theory</b>	<b>3</b>
2.1	Data-Oriented Programming	3
2.1.1	Core Principles	3
2.1.2	State Management	6
2.1.3	Polymorphism	7
2.1.4	Testing	7
2.1.5	Data Validation	8
2.1.6	Concurrency	8
2.2	Common Software Faults	8
2.2.1	Memory Errors	9
2.2.2	Type Errors	10
2.2.3	Pointer Issues	11
2.2.4	Error Handling	13
2.2.5	Resource Management Errors	14
2.2.6	Number Errors	15
2.2.7	Developer Oversights	15
<b>3</b>	<b>Dash Language Design</b>	<b>17</b>
3.1	Philosophy	17
3.2	Grammar	17
3.3	Core Features	17
3.3.1	Assignments	17
3.3.2	For Loops	19
3.3.3	Functions	20
3.3.4	Pipe Operator	20
3.3.5	Modifying Structs	21
3.3.6	Generic Structs	21
3.3.7	Deferring Code	22
3.3.8	Pattern Matching	22
3.3.9	Annotations	24
3.3.10	Extern C	25
3.4	Type System	25
3.4.1	Primitive Types	25
3.4.2	Type Inference	27

3.4.3	Type Casting . . . . .	27
3.4.4	Type Definition and Aliases . . . . .	27
3.4.5	Type Predicates and Dirty Types . . . . .	28
3.4.6	Optional Types . . . . .	28
3.4.7	Tagged Unions . . . . .	29
3.4.8	Soundness . . . . .	30
3.5	Memory Model . . . . .	30
3.5.1	Memory Allocation . . . . .	30
3.5.2	Memory Deallocation . . . . .	31
3.5.3	Safety . . . . .	34
3.5.4	Pointers . . . . .	35
3.6	Polymorphism . . . . .	35
3.6.1	Parametric Polymorphism . . . . .	35
3.6.2	Ad-hoc Polymorphism . . . . .	36
3.7	Error Handling . . . . .	36
3.7.1	Defining Errors . . . . .	36
3.7.2	Raising Errors . . . . .	37
3.7.3	Catching Errors . . . . .	37
3.7.4	Error-prone Operations . . . . .	38
3.7.5	Global Error Handling . . . . .	39
3.8	Code Organization . . . . .	39
3.8.1	Libraries . . . . .	39
3.8.2	Importing . . . . .	39
<b>4</b>	<b>Implementation</b>	<b>41</b>
4.1	Lexer . . . . .	41
4.2	Parser . . . . .	41
4.2.1	Pratt Parser . . . . .	42
4.2.2	Recursive Descent Parser . . . . .	43
4.2.3	Error Recovery Strategy . . . . .	43
4.3	Semantic analyser . . . . .	43
4.3.1	Data Structures . . . . .	43
4.3.2	Type Checking . . . . .	44
4.3.3	Error Recovery Strategy . . . . .	44
4.4	IR Generator . . . . .	44
4.4.1	Generation Process . . . . .	44
4.4.2	Type Representation . . . . .	45
4.4.3	Copy-Update Expression . . . . .	48
4.4.4	Pattern Matching . . . . .	48
4.4.5	Compiler Hints . . . . .	48
4.5	Specification Validation . . . . .	48
4.6	Implementation Coverage . . . . .	49
<b>5</b>	<b>Evaluation</b>	<b>51</b>
5.1	Introduction . . . . .	51
5.2	Comparison Criteria . . . . .	51
5.3	Language Comparison . . . . .	53
5.3.1	State Management . . . . .	53
5.3.2	Type System . . . . .	55
5.3.3	Safety Guarantees . . . . .	57
5.3.4	Error Handling . . . . .	58

Contents	vii
5.3.5 Abstraction Mechanisms . . . . .	59
5.3.6 Code Clarity and Verbosity . . . . .	60
5.3.7 Interoperability . . . . .	62
<b>6 Conclusion</b>	<b>65</b>
<b>7 Appendix</b>	<b>67</b>
7.0.1 Dash Language Specification . . . . .	67
7.0.2 Dash Tree-Sitter Grammar . . . . .	67
7.0.3 Evaluation in Dash . . . . .	81
7.0.4 Evaluation in Go . . . . .	88
7.0.5 Evaluation in Rust . . . . .	94

## List of Figures

2.1	Structural sharing example . . . . .	6
-----	--------------------------------------	---

## List of Tables

3.1	Function and library visibility matrix . . . . .	40
4.1	Mapping between dash primitive types and LLVM IR types . . . . .	47

## List of Listings

2.1	An OOP method modifying internal state . . . . .	3
2.2	Initialising a byte buffer in a mutable way . . . . .	4
3.1	Types of assignments in Dash . . . . .	18
3.2	Example of incorrect reassignment of variable defined using let in Dash . . . . .	18
3.3	Variable reassignment defined using var in Dash . . . . .	18
3.4	Example of variable reassignment with function call in Dash . . . . .	18
3.5	Multiple variable assignments in Dash . . . . .	18
3.6	Multiple assignments with literal and function call in Dash . . . . .	19
3.7	C-style for loop in Dash . . . . .	19
3.8	For range loop in Dash . . . . .	19
3.9	For range loop in Dash with element . . . . .	19
3.10	Boolean for loop in Dash . . . . .	19
3.11	Infinite for loop in Dash . . . . .	20
3.12	Function definition in Dash . . . . .	20
3.13	Error-prone function definition in Dash . . . . .	20
3.14	Pipe operator in Dash . . . . .	20
3.15	Error handling with pipe operator . . . . .	21
3.16	Copy and update operator . . . . .	21
3.17	Generic struct definition . . . . .	21
3.18	Function accepting generic struct and modifying it . . . . .	21
3.19	Deferring functions and code . . . . .	22
3.20	Match by integer values . . . . .	22
3.21	Match by enum field . . . . .	23
3.22	Match by union type . . . . .	23
3.23	Match by union type with assignment . . . . .	23
3.24	Match by enum field as expression . . . . .	24
3.25	Annotation examples . . . . .	24
3.26	Defining unicode code points of varying length . . . . .	26
3.27	Defining a custom type in Dash . . . . .	27
3.28	Defining a type alias in Dash . . . . .	27
3.29	Type definition vs alias behaviour . . . . .	27
3.30	Example of a type predicate in Dash . . . . .	28
3.31	Struct field predicates in Dash . . . . .	28
3.32	Validating dirty type in Dash . . . . .	28
3.33	Function accepting optional type . . . . .	28
3.34	Null coalesce operator usage . . . . .	29

---

3.35	Conditional null check . . . . .	29
3.36	Force unwrap operation . . . . .	29
3.37	Unwrap and dereference precedence . . . . .	29
3.38	Union in Dash made up of integers and floats . . . . .	30
3.39	Allocating memory using make . . . . .	30
3.40	Using memory . . . . .	30
3.41	Counted type in Dash using special type signature . . . . .	32
3.42	Immutability prevents cyclical structures in Dash . . . . .	32
3.43	Lifetime of variables in recursive functions assuming 'a' is heap allocated . . . . .	33
3.44	Generic stack using generics . . . . .	35
3.45	Function overloading in Dash . . . . .	36
3.46	Error definitions in Dash . . . . .	36
3.47	Raising errors in Dash . . . . .	37
3.48	Error propagation in Dash . . . . .	37
3.49	Error catching in Dash . . . . .	37
3.50	Error catching in Dash through matching . . . . .	37
3.51	Error-prone operations . . . . .	38
3.52	Definition of a library . . . . .	39
3.53	Import statements in Dash . . . . .	40
4.1	Core Pratt parser implemented in Golang . . . . .	42
4.2	Missing '}' when parsing . . . . .	43
4.3	Literal type inference . . . . .	44
5.1	Rust state management using mut . . . . .	54
5.2	State is explicitly passed and returned . . . . .	54
5.3	Function accepting any struct which implements interface in Golang . . . . .	59
7.1	Dash tree-sitter grammar . . . . .	67
7.2	Evaluation source code in Dash . . . . .	81
7.3	Evaluation implemented in Golang . . . . .	88
7.4	Evaluation implemented in Rust . . . . .	94



# Introduction

This thesis is an exploration into how data-oriented programming (DOP) could be applied at the language level while eliminating complexities and common software faults [CWE24z] found in other programming languages. DOP is a relatively new paradigm that came into existence after the creation of the programming language called Clojure in 2007, with its roots in the late 1950s when Lisp was invented [Sha22]. The paradigm enforces immutability and generality for the sake of simplicity and reusability, while attempting to mitigate performance impacts [Sha22]. DOP is considered a language agnostic paradigm with many implementations in different programming languages [DC23, dgi24, ben19]. This creates an opportunity to explore if DOP could be implemented at the language level and how such an implementation would look like, serving both as an educational exercise in language design and as an investigation into how DOP principles translate to a different technical context.

The goal of this thesis is to implement a compiled programming language with a sound type system [Pie02a] that implements all the principles described by DOP while avoiding common software faults. By combining solutions found in the industry with DOP, the novel language aims to be simple yet effective to use, memory safe and performant. While there are data-oriented languages, their focus lies in different areas such as low-level systems programming [Bil24] or dynamically typed functional programming [Hic24].

To achieve this goal and evaluate the effectiveness of the implementation, the thesis will explore the following questions:

- What specific object-oriented programming complexities around state management does data-oriented programming address and how?
- How can language design prevent common runtime vulnerabilities identified in the common weakness enumeration list while maintaining data-oriented programming principles?
- How could data-oriented programming principles be applied in a compiled language?
- How does the data-oriented compiled language compare to Go and Rust in terms of state management, safety, abstraction mechanisms, error handling, verbosity and interoperability?

To address these research questions, this thesis takes a systematic approach by first explaining the DOP paradigm followed by an in-depth analysis of common software vulnerabilities documented in the CWE database to identify recurring patterns and faults. This is done in chapter 2. Rather than attempting to solve these issues through DOP alone, the language design, described in chapter 3, combines established safety mechanisms with DOP principles to create targeted language features.

The feasibility of this approach is demonstrated in chapter 4 through a working compiler implementation in Go using LLVM as the compiler backend [Tea24c]. Golang was chosen because of its simplicity and static typing system which helps prevent bugs in the compiler implementation, while its garbage collection allows for rapid prototyping without memory management overhead. On the other hand LLVM was chosen as it is a robust, feature-rich and widely used compiler backend [LA04].

To preliminary validate the design choices, an implementation of an arithmetic expression evaluator is compared against Golang and Rust in chapter 5 across a variety of categories to demonstrate benefits and drawbacks of the language. Golang and Rust were selected based on their differing approaches to language design. Golang values simplicity and minimalism, while Rust prioritises memory safety through its ownership system. These differences provide useful reference points when evaluating Dash. The arithmetic expression evaluator was chosen due to the limited scope of the compiler implementation, which currently lacks heap memory and standard library that would have made implementing a more complex project nearly impossible. Hence, the compiler acts as a proof-of-concept, putting in place sufficient functionality to demonstrate the main ideas of the language while reserving the implementation of the memory model, error handling and concurrency for later development. Nevertheless, the calculator demonstrates core DOP principles, which combined with a theoretical evaluation of language features, described in the language specification, provides concrete insights into the language's usability, safety guarantees and implementation trade-offs in a less abstract context.

The thesis is structured into four main parts:

- Chapter 2: Background on DOP principles and analysis of common software faults, establishing the theoretical foundation and identifying specific vulnerabilities that language design can address
- Chapter 3: Detailed design of the Dash programming language, showing how DOP principles can be adapted into concrete language features while maintaining safety and type guarantees
- Chapter 4: High level implementation details of the compiler, demonstrating the feasibility of implementing DOP at the language level through a working compiler
- Chapter 5: Evaluation and comparison with existing languages, analysing how Dash's approach compares to Go and Rust across key areas like state management, safety and code clarity

# Background and Theory

## 2.1 Data-Oriented Programming

There are multiple data-oriented approaches in computer science with unique distinctions. Data-oriented design, which is mostly used in game development and high-performance systems, focusses on carefully laying out data in memory to maximise CPU cache usage. On the other hand, data-driven programming uses descriptive data to generate domain-specific languages and makes up a branch of declarative programming [Sha22]. This thesis focuses on data-oriented programming (DOP) which is a programming paradigm. The following sections explain DOP in more detail.

### 2.1.1 Core Principles

Data-oriented programming stands on four core principles. These include 1) separating data from code, 2) ensuring all data is immutable, 3) being able to represent data in a generic way and 4) separating data schema from data representation [Sha22]. These principles are language-agnostic as they can be implemented in various programming languages [DC23,dgi24,ben19].

#### Separate data from code

The first DOP principle states that data should not be mangled with code. This means functions should not remember any state in consecutive executions. In the Object Oriented Paradigm (OOP) this would be called classes with methods modifying the internal state of the class as can be seen in example listing 2.1.

```
1 type User struct {
2     age int
3 }
4 func (u *User) setAge(a int) {
5     u.age = a
6 }
```

**Listing 2.1:** An OOP method modifying internal state

Separating data from the code that manipulates it reduces program complexity. Because of this division, it is no longer necessary to understand how methods internally manage their state, which simplifies reasoning about program behaviour [Sha22]. Additionally, it makes it possible

for developers to analyse algorithms and change data structures independently from one another, resulting in software systems that are flexible and maintainable.

Additionally, separating data from the code brings three major benefits. First, it enables code reuse across different contexts without requiring complex inheritance hierarchies or composition patterns. For example, functions operating on similar data structures can work across different data types without modification. Second, it simplifies isolated testing, as developers can create test data independently without needing to instantiate complex objects. Third, it reduces system complexity by creating cleaner separation of concerns; data and code can be understood and modified independently, resulting in systems that are easier to grasp and maintain.

However, there are several trade-offs with this separation. By separating data from code, the traditional encapsulation mechanisms found in OOP is not possible. While this might initially seem like a problem, the impact is mitigated due to data immutability, which prevents unwanted modifications and the simplicity of the data structures as they are only composed of maps, which reduces the need to hide implementation details. The second trade-off impacts code organization and discovery, as without the grouping that classes provide, developers lose the ability to easily locate related operations by examining class definitions. This is because functions operating on data structures are typically distributed across multiple modules and are generic. The final trade-off is that since functionality must now be explicitly provided for each data type rather than being inherited through class hierarchies, the separation of code and data may result in a greater number of unique types and functions in the system.

## Data is immutable

Data is not allowed to be mutated in DOP. Instead of modifying existing data, new versions are created when changes are needed. For example, instead of changing the value of a field within a structure, a new structure is created with the updated value while the original remains unchanged. This technique is called structural sharing and is expanded upon in section 2.1.2.

Since data cannot be changed while processing, data immutability ensures predictable behaviour in asynchronous operations. Firstly, it permits quick equality checks using reference comparison instead of deep value comparison. Secondly, it offers inherent thread safety in concurrent environments without the need for additional synchronisation mechanisms. Finally, it allows for confident data sharing across functions without unexpected modifications.

Even though immutability has many advantages, there are also some significant disadvantages that should be addressed when applied at the language level. For one there is a slight performance overhead compared to mutable operations, as data must be copied whenever a modification is made. To negate this impact, data structures like Hash-Array Mapped Tries (HAMT) and Relaxed Radix Balanced Trees (RRB-Trees) can be used to reduce performance issues by improving cache locality and reducing the memory footprint [SV15, Ped17].

Another significant drawback emerges when dealing with performance-critical code and the need for efficient data initialisation. For example, if an array wants to be initialised, in a mutable imperative programming language this could be done using an allocation and a simple for loop, as demonstrated in listing 2.2.

```
1 buf := make([]byte, 64)
2 for i := range buf {
3     buf[i] = byte(i)
4 }
```

**Listing 2.2:** Initialising a byte buffer in a mutable way

For performance-critical code, contiguous memory allocations can be highly efficient due to increased data locality [PVJB12, BCM04b]. However, when data is allocated recursively in a

bottom-up fashion (to avoid reallocating everything while building the data structure), memory fragmentation becomes a risk. This fragmentation occurs because multiple separate allocation calls are made instead of a single allocation, providing no guarantee of contiguous memory placement. This means there has to be a way to pre-allocate data without violating any of the DOP principles listed above. This is expanded upon in section 3.5.1.

In imperative programming languages, loops often require variable reassignment, such as incrementing an iterator variable. In DOP, the reassignment of variables in linear programs could be achieved without mutation by defining new variables, this approach fails in loops since variables defined within loop blocks have limited scope and do not persist between iterations. The traditional solution of maintaining a mutable variable outside the loop violates DOP principles as it requires mutating values. Therefore, a mechanism to emulate linear program flow in the presence of loops that allows reassignment without mutation is required. This implementation is detailed in section 3.3.1.

## Generic data representation

DOP advocates for representing data using generic data structures (structs), to allow for flexibility and reusability. While the original DOP book uses hash maps with string-based field references and no types, an alternative approach would be using generic structs that maintain type safety and field name validation while preserving flexibility. This is discussed in detail in section 3.3.6. The latter approach is also more convenient when implementing DOP at a language level as it leverages existing type system features and compile-time checks rather than requiring custom runtime type handling and validation. Furthermore, this approach aims to mitigate potential vulnerabilities (see section 2.2.2), complexity and performance overhead that can arise from dynamic typing and using hash maps.

The implementation of generic structs, offers several key advantages. It allows for the use of functions that can operate across different generic types as long it accepts the generic struct. It also provides access to rich language functions for data manipulation.

The limitation of this approach is that it does not enable as flexible of a data model, as described in the DOP book [Sha22], where hash maps can be modified dynamically, due to generic struct definitions being required. While this initially seems like a downside, it is actually beneficial as it provides clear data contracts, preventing runtime errors such as misspelled field names and type mismatches. Overall, this makes the codebase more maintainable by making data structures explicit and self-documenting. The compile-time safety guarantees and improved maintainability outweigh the minor inconvenience of defining new generic structs, while also mitigating the disadvantages of generic data representation described in the DOP book.

## Separating data schema from data representation

In traditional DOP, data schemas are separated from data representation through JSON schemas to validate hash maps at runtime [Sha22]. This allows developers to choose which data needs schema validation and which does not. This works well for dynamically typed languages or implementing DOP as a layer above a language i.e. as a framework.

However, it is impractical to use hash maps for all data representation, especially for a compiled language where type safety and performance are required. Hash maps compute the location of data by running the key through a hash function at runtime while structs calculate the offset of the data from the start pointer at compile time. Logically, the use case for hash maps is very different than structs, as they are commonly used when dynamic keys are used or the ability for the map to grow or shrink is required. Since functions typically operate on predefined data structures, structs are often the better choice, especially in compiled languages where concrete types

are needed. This means data schemas and data representation naturally become closely aligned when implementing DOP at the language level. This makes the principle difficult to implement. Nevertheless, flexibility can be maintained through language features like:

- Structural typing using field names and types for type equivalence
- Optional types letting developers choose when the absence of values is acceptable
- Union types that preserve type safety while maintaining flexibility
- Ad-hoc polymorphism that hides type dependent functionality behind a unified overloaded function.

While complete data separation is not practical for a compiled language, the previously mentioned features preserve much of the flexibility benefits that this DOP principle aims to achieve. This means a balance between compile-time type safety and runtime flexibility needs to be found, so that the core benefits of this principle are maintained.

## 2.1.2 State Management

Understanding DOP's definition of pure functions is essential before exploring state management. DOP maintains a similar definition of pure functions to paradigms like functional programming, where the output solely depends on the function arguments [SAB98]. However, it takes a more pragmatic approach to I/O operations. Unlike functional programming, where the separation between pure functions and side effects is a core principle, DOP does not prescribe specific patterns for handling I/O. This has the negative side effect that it is possible for external state to be coupled to a function by reading an environment variable, reading from disk or communicating over the network.

On the other hand, this flexible approach to pure functions complements DOP's unique strategy for state management. While not prohibiting stateful components, DOP emphasises minimizing state by centralizing it in a single module. In the DOP a mutation is split into two phases; calculation phase and commit phase. The calculation phase computes the next version of the data. This usually occurs as data moves through functions that modify it in an immutable way. To keep this efficient a technique called structural sharing is used. Structural sharing allows for the efficient modification of immutable data structures by reusing unchanged portions of the original data structure while creating new nodes only for the modified parts, thereby minimizing memory usage and computational overhead [Sha22]. This example is portrayed in fig. 2.1 in a binary tree where only the parent nodes (dark green) of the new node (in light green) need to be updates while keeping the left hand side unchanged.

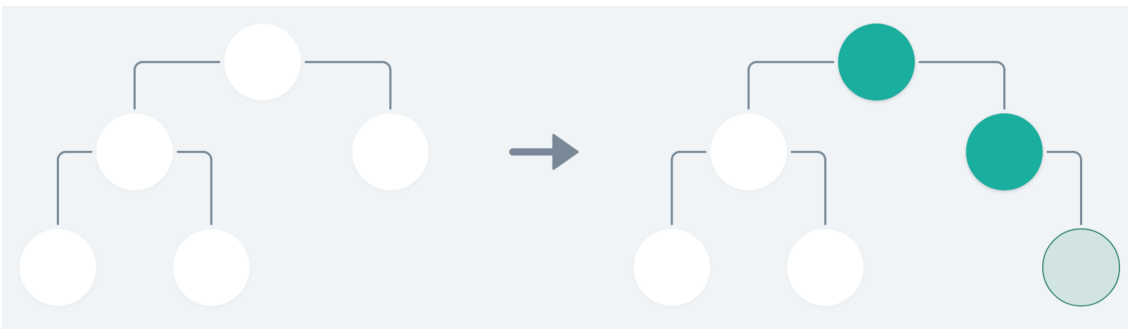


Figure 2.1: Structural sharing example

The second phase is the commit phase, which is the only stateful part of a system. During a commit the system state is moved forward so that it refers to the updated data from the calculation phase. This is described to occur in a stateful class that modifies the reference to the state and atomically swaps the old with the new data [Sha22]. This approach has limited applicability since it only works in languages that support OOP. For languages exclusively based on DOP, implementing OOP only for this part would be impractical. It would also undermine the language by allowing developers to circumvent its paradigm in favour of OOP. One possible way to resolve this would be to use atomic compare-and-swap operations provided by the CPU [Int24, Lim24] to manage program state using a variable, potentially complementing the type system with an atom type to make such operations easier as done in languages like Clojure [Com24a].

### 2.1.3 Polymorphism

Polymorphic type systems enable code reuse across multiple types. Two primary variants exist: parametric and ad-hoc polymorphism. Parametric polymorphism allows for generic type definitions that can be instantiated with specific types as needed, while maintaining consistent behaviour across all instances [Pie02b]. This usually comes in the form of generics which already exist in many programming languages today including Golang and Java [Gol24b, Ora21]. In contrast, ad-hoc polymorphism permits different behaviours for different types, primarily implemented through function overloading where the compiler selects appropriate implementations based on argument types [Pie02b]. While DOP describes a basic form of ad-hoc polymorphism through switch statements [Sha22], this approach is too simplistic and naive for a language-level implementation. A more sophisticated system would add ad-hoc polymorphic behaviour at a syntax level and determine the correct function to be called at compile-time by looking at the arguments passed, making it a powerful tool for implementing DOP at the language level.

### 2.1.4 Testing

Testing in DOP is simple due to its emphasis on functions only accepting and returning data without mutable operations or internal state [Sha22]. This simple data manipulation is consistent with common unit testing practices, where input data is generated, passed to the function and the output is compared against expected values [Ani22]. While this does not necessarily make unit testing easier compared to other paradigms, it simplifies testing by removing the need to think about state mutations due to complex interactions, as the input data encapsulates all necessary state. However, if there are interactions with databases or other external state sources this is not possible, meaning there has to be a way to inject mock dependencies when testing functions interacting with other systems.

Compared to the DOP book where validation of output with expected data occurs using a generic deep comparison of the two hash maps [Sha22], at a language level this would look differently. Implementing equality comparisons between structs requires specific functions for each data type. This can be easily resolved through compile-time code generation for the comparison functions, similarly to how some language generate code for JSON marshalling and unmarshalling [Com24e]. It is also possible to do this without generating additional code by relying on reflection at runtime [Pik11].

Overall testing in DOP follows similar patterns to other paradigms. Although, its explicit focus on data transformations and reduced state complexity provides meaningful benefits for writing and maintaining tests. In the end, this method makes tests easier to understand and more predictable.

### 2.1.5 Data Validation

As data is the integral part of DOP it should be simple to validate data. The method for data validation outlined in the DOP book relies on using JSON schemas [Sch24] to ensure the hash map conforms. As mentioned before, at a language-level using hash maps as the underlying data structure is not ideal so there has to be a way to validate structs and arrays easily and ensure the program is in a valid state. This goes beyond simply validating types, which, at a language-level, should be taken care of by the type system, as it also includes ensuring the data itself is valid. For example, that a number is within a given range or that a string matches a specific pattern. Moreover, it should be easy to ensure the validity of data as it flows through the system and catch any inconsistencies as early as possible. Data validation at a language level is further discussed in section 3.4.5.

### 2.1.6 Concurrency

DOP allows for efficient concurrency via optimistic concurrency control, which eliminates the need for locks and the accompanying deadlock risk [Sha22]. This method works because data is immutable, meaning once it is created, it cannot be changed. The system keeps numerous versions of the data, due to structural sharing, which also makes them very efficient.

The concurrency model described in the book is similar to Git's version control mechanism. When no concurrent modifications are present, the system conducts a fast-forward update, which means it safe to directly update to the new state without reconciliation. When concurrent changes occur but do not conflict, it does a three-way merge to combine the modifications. This reconciliation method is more complicated and explained in detail in the book [Sha22]. In rare circumstances where changes contradict, the operation is aborted and can be restarted. This optimistic method results in great throughput for both reads and writes because operations do not block while waiting for locks.

Reads are intrinsically thread-safe even in the absence of locks since immutable data cannot be updated during the read operation. When a function mutates data, the system generates a new version of the data structure while retaining references to the prior one.

## 2.2 Common Software Faults

MITRE is a non-profit organization established to enhance national security and serve the public interest as an independent adviser [MIT24a]. Importantly, MITRE manages the Common Weakness Enumeration (CWE) list, which is a community-developed catalogue of software and hardware weaknesses that describes security weaknesses in systems [CWE24y]. While there are around 400 software related CWEs documented by MITRE [CWE24z], this section only focuses on the most common ones. The weaknesses were selected following the CWE Top 25 list [MIT24b] and then branching out to related weaknesses in the software category. It should be noted that concurrency related weaknesses are not analysed as those are beyond the scope of this thesis.

The following sections use the words "weakness" and "vulnerability" interchangeably, referring to a "weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source" [CSR15].

## 2.2.1 Memory Errors

### **CWE-120: Buffer Copy without Checking Size of Input**

This error is also known as the "Classic Buffer Overflow" [CWE24b]. A buffer overflow vulnerability occurs when the program fails to validate whether the destination buffer has sufficient space before copying data from the source buffer. This usually occurs when data is copied without checking the size before copying.

The security implications of buffer overflows are severe. They can enable attackers to execute arbitrary code, potentially compromising other security controls. Additionally, buffer overflows often result in system crashes and can be used to cause denial of service by terminating programs or exhausting resources through endless loops.

### **CWE-124: Buffer Underwrite**

On the other hand there are buffer underwrites, also known as buffer underflow [CWE24c]. Buffer underflows happen when a program writes data to a memory location that comes before the start of a buffer. This situation occurs when a pointer gets moved backwards before the buffer's beginning or when using negative numbers to access memory positions.

A buffer underflow can cause several serious problems. When a program writes data outside its allowed memory space, it can crash the system or make it unstable. Moreover, attackers might use this weakness to run their own code or change how the program works.

### **CWE-125: Out-of-bounds Read**

An out-of-bounds read occurs when a program attempts to read memory outside the allocated buffer's boundaries, either before its start or after its end [CWE24d]. This is similar to accessing a buffer with an incorrect length value [CWE24r].

Out-of-bounds reads can expose sensitive data like encryption keys and memory addresses that attackers can leverage for further exploitation. Beyond data exposure, they may cause system crashes or produce incorrect results when accessing invalid memory. In severe cases, if an attacker can precisely control the out-of-bounds access, they may achieve arbitrary code execution similar to a traditional buffer overflow vulnerability.

### **CWE-129: Improper Validation of Array Index**

Improper Validation of Array Index is a form of out-of-bounds read that occurs when a program does not check array indexes from untrusted input properly [CWE24f].

The impact of this vulnerability can be severe. When accessing memory outside array bounds it can lead to; memory corruption, system crashes, sensitive data being read and the execution of malicious code if the attacker controls memory precisely.

### **CWE-131: Incorrect Calculation of Buffer Size**

Incorrect calculation of buffer size occurs when a miscalculation happens when determining how much memory to allocate [CWE24g]. This could cause buffers to be created that are either too small or too large for the intended data.

The vulnerability can lead to buffer overflow issues, potentially causing program crashes or enabling unauthorised code execution. When buffer size is smaller than needed, attackers could exploit this to read or modify memory outside intended bounds, potentially exposing sensitive data or compromising system security.

### **CWE-401: Missing Release of Memory after Effective Lifetime**

This CWE describes a memory management vulnerability in which software fails to appropriately deallocate memory after its lifetime has ended [PLO24e]. This is prevalent when applications handle malformed data incorrectly or have interrupted sessions, especially in programming languages where memory allocation and deallocation are manually managed.

The main consequences are decreased system performance and denial of service. An attacker could intentionally cause memory leaks to deplete system resources, resulting in program crashes, instability, or unusual behaviour under low memory situations. This vulnerability impacts both CPU and memory resources.

### **CWE-787: Out-of-bounds Write**

An out-of-bounds write error occurs when a program writes data beyond the allocated memory buffer boundaries, either before the start or after the end [CWE24q]. This vulnerability commonly happens during buffer operations where bounds checking is inadequate or missing, allowing data to be written to unintended memory locations outside the buffer's valid range.

Memory corruption from out-of-bounds writes can lead to system crashes through invalid memory access or allow attackers to execute arbitrary code by overwriting data like return addresses. Additionally, subsequent operations on corrupted memory can lead to undefined behaviour, potentially causing the program to enter a state that compromise its reliability and security.

## **2.2.2 Type Errors**

### **Integer Errors**

Integer errors are made up of several vulnerabilities, including coercion errors [CLA24a], unexpected sign extensions [CLA24b], signed-to-unsigned conversions [CLA24c], unsigned-to-signed conversions [CLA24d] and numeric truncation errors [PLO24d]. These weaknesses typically happen when type casting between primitives of various sizes or signs, which can result in data loss, unexpected transformations, or wrong interpretations. The underlying issue lies in the assumptions made during these conversions.

The impact of integer errors manifest in multiple ways, particularly when dealing with negative values, buffer sizes, or memory indices, that compromise system security and stability. These vulnerabilities can lead to undefined behaviour causing system crashes, infinite loops and resource exhaustion. Moreover, they can facilitate buffer overflow or underflow conditions, potentially enabling unauthorised code execution and memory corruption.

### **String Errors**

String vulnerabilities represent a class of software weaknesses that are caused by improper handling of character encodings and string boundaries. These vulnerabilities are; incorrect calculation of multi-byte string lengths [CWE24i] and improper null termination [PLO24a]. Multi-byte string length calculation errors occur when programs make incorrect assumptions about string lengths, particularly in contexts involving UTF-8 or UTF-16 where individual characters can span multiple bytes of memory instead of just one [Yer03, HY00]. On the other hand, null termination errors, cause software systems to fail to properly manage string boundaries through null character delimiters.

The security implications of these string processing vulnerabilities are severe. Buffer overflows can occur potentially enabling attackers to execute unauthorised code. System stability

can be compromised through memory corruption, which can lead to crashes and denial of service. Furthermore, these vulnerabilities may result in out-of-bounds memory access, potentially exposing sensitive data that attackers could leverage for more sophisticated exploits. Programs may also enter infinite loops, causing uncontrolled memory consumption, degrading system security and performance.

## Function Call With Incompatible Signature

There are two kinds of function call vulnerabilities related to incompatible signatures; incorrect argument quantity [CWE24j] and type mismatches during function invocation [CWE24k]. The argument quantity vulnerability occurs when functions are called with either excessive or insufficient parameters, while type mismatches emerges when argument types deviate from the function's expected signature, particularly prevalent in dynamically typed languages or scenarios where implicit casting occurs during compilation.

The technical impact of these incompatible function signatures largely results in quality degradation of the software system. This degradation is notable in languages that lack strict type enforcement at compile-time, where such mismatches between function declarations and their invocations can lead to undefined behaviour and subsequent weaknesses in the system's operational integrity.

### CWE-843: Type confusion

Type confusion, also known as Access of Resource Using Incompatible Type [CWE24x], occurs when a resource is allocated or initialised using one type but later accessed using an incompatible type. This mismatch happens when interpreting the same variable or memory location in multiple ways, which can trigger logical errors since the resource lacks expected properties.

The impact can lead to reading or writing memory outside buffer bounds if the allocated buffer is smaller than the attempted access type. This could result in system crashes, unauthorised code execution, or denial of service through program termination.

### CWE-1287: Improper Validation of Specified Type of Input

This weakness describes an error where a program receives input expected to be of a specific type but fails to properly validate if the input matches that type [CWE24e]. This commonly occurs in languages with an unsafe type system or when type casting is supported, potentially allowing input to evade intended type restrictions.

The impact varies by context but can lead to serious security issues. When type validation fails, attackers could trigger unexpected errors, cause incorrect program behaviour, or exploit vulnerabilities like SQL injections that would otherwise be prevented if proper type checks were implemented.

## 2.2.3 Pointer Issues

### CWE-468: Incorrect Pointer Scaling

The Incorrect Pointer Scaling vulnerability occurs in programming languages that allow pointer arithmetic without considering that pointer arithmetic automatically multiplies by the size of the data type [CLA24g]. For example, when adding 1 to an int32 pointer, it actually adds 4 bytes. This implicit multiplication can cause developers to accidentally access wrong memory locations when they forget this behaviour.

Memory can be read or modified unexpectedly when pointer scaling is done wrong. This mostly leads to buffer overflow issues where programs read or write outside intended memory bounds but can also cause attackers to potentially gain access to data they should not be able to see.

### **CWE-469: Use of Pointer Subtraction to Determine Size**

Pointer subtraction, occurs when code tries to figure out memory size by subtracting two pointers [CLA24h]. This becomes problematic when pointers reference different memory locations, leading to wrong size calculations that could corrupt memory operations.

The impact of this vulnerability lets attackers potentially execute arbitrary code with same privileges as the vulnerable program. Since wrong pointer arithmetic affects memory operations, attackers could modify or read memory they should not access, possibly leading to privilege escalation or unauthorised code execution.

### **CWE-476: NULL Pointer Dereference**

The null pointer dereference vulnerability happens when code tries to use a pointer that's null but expects it to be valid [Kin24c]. This is mainly seen in code that does not validate if a pointer is null before using it.

The main impact is program crashes and process failures unless proper error handling exists. In some edge cases, this can let attackers read or write memory directly, possibly leading to unauthorised code execution.

### **CWE-588: Attempt to Access Child of a Non-structure Pointer**

This vulnerability happens when code tries to access structure fields from a pointer that is not actually pointing to a structure [Com21]. This situation arises through type casting where memory gets interpreted as having fields but it does not match the expected layout.

When memory gets wrongly accessed this way, it can corrupt data stored next to the pointer in memory due to field assignments after casting. This corrupted memory state could lead program to crash or restart since memory access becomes undefined.

## **Manual Memory Deallocation Vulnerabilities**

Memory deallocation vulnerabilities occur when programs mishandle memory management functions. These issues happens in languages with manual memory management in two main ways; using `free()` [cpp24] on heap memory which is not the start of the buffer [CWE24l] and mixing different memory allocation/deallocation function implementations [CWE24m]. The problems arise from basic misuse of memory management functions that negatively affect how programs handle memory resources.

The impact of these vulnerabilities can lead to memory corruption, program crashes and possible unauthorised code execution by attackers. When `free()` operates on wrong memory locations, malicious users could potentially write data anywhere in memory and execute unauthorised commands. Additionally, mismatched memory management functions might corrupt critical program variables or cause denial of service through program termination. The severity depends on implementation but generally creates unstable program behaviour.

## Pointer Arithmetic Errors

Out-of-range pointer offset vulnerability happens when pointer arithmetic on a valid pointer occurs but the offset used can point outside the intended memory space [CWE24t]. The vulnerability manifests when programs use offsets to access structured data, where these offsets might come from untrusted sources [CWE24s] and bad calculations [CLA24g, CLA24h].

An attacker exploiting this can read sensitive memory if the bad pointer gets used in read operations. When the program accesses malformed data structures or memory it should not have access to, it can cause a crash. Moreover, if the pointer is used in function calls or write operations, arbitrary code execution could be possible.

### CWE-824: Access of Uninitialized Pointer

The Access of Uninitialised Pointer vulnerability occurs when a program reads from or writes to a pointer that lacks proper initialisation, potentially causing the pointer value to reference invalid memory locations [CWE24u].

The vulnerability allows for arbitrary code execution, denial of service through software crashes when accessing erroneous memory locations and unauthorised memory reads that could expose sensitive data. These effects are especially severe when attackers are able to change the target of pointer operations by altering the contents of the uninitialised memory.

### CWE-825: Expired Pointer Dereference

This vulnerability describes a situation where a program dereferences a pointer that points to a memory location that was previously valid but now no longer is [CWE24v]. This can occur when a program returns pointers to stack allocated values [Ven14], pointers to a memory region outside of the expected range [Kin14], or when releasing memory while maintaining a pointer to that memory [KBW24]. If the original pointer is accessed it could lead to reading or modifying data used by a different function.

The impact of dereferencing an expired pointer is severe. It could cause the program to crash, allow an attacker to access sensitive data stored in memory, or execute unauthorised code or instructions.

## 2.2.4 Error Handling

### Incorrect Error Checking

The absence of exception handling is a critical vulnerability preventing proper response to failures or unexpected events [PLO24f]. This relates closely to another weakness, where exceptions are not caught causing them to go unnoticed [Kin24b]. The latter is particularly problematic when developers assume function calls cannot fail. Similarly, forgetting to check the function return value [KS24] or incorrectly checking the return value [CLA24e] hinders error detection capabilities. This is reflected in practice, where a lot of critical runtime errors occur due to faulty error handling [YLZ<sup>+</sup>23].

The impact of these vulnerabilities primarily lead to unexpected states, denial of service due to crashes, program termination and exposure of sensitive data. The lack of proper error handling creates situations where systems may operate in undefined states, compromising their stability and security.

### **CWE-390: Detection of Error Condition Without Action**

This vulnerability occurs when a system identifies a specific error but fails to implement any error handling mechanisms [CLA24f]. The program essentially ignores the error leaving the system in an undefined state, which is a fundamental flaw in error management.

The impact of the vulnerability varies but usually enables attackers to exploit unhandled error conditions to force the system into unexpected states. Such abuse can trigger unintended logic execution, likely leading to broader system anomalies.

### **CWE-396: Declaration of Catch for Generic Exception**

Generic exception handling is a vulnerability, which leads to condensed but problematic error management code [KB24]. This practice obscures specific exceptions that require distinct handling approaches, effectively removing the advantages of exceptions, particularly as applications expand and introduce new exception types.

The impact of this vulnerability is not severe on its own but could mask more severe vulnerabilities, due to the lack of detailed error information.

## **2.2.5 Resource Management Errors**

### **CWE-771: Missing Reference to Active Allocated Resource**

This vulnerability occurs when a system fails to properly keep references to allocated resources, preventing their reclamation [CWE24n].

The impact of the vulnerability manifests through potential denial of service attacks, where malicious actors can manipulate resource allocation processes. By exploiting improper resource management, attackers could exhaust the available resource pool, effectively blocking other processes from accessing similar resources.

### **CWE-772: Missing Release of Resource after Effective Lifetime**

This CWE describes a vulnerability where software fails to release resources at the end of their lifetime [CWE24o]. This oversight occurs when resources like memory, CPU cycles or disk space remain allocated despite no longer being required by the program.

The primary impact of this vulnerability is denial of service, where attackers can create multiple instances of the resource depleting available resource pools. This strategy prevents other processes from accessing the same resources, which compromises system functionality.

### **CWE-826: Premature Release of Resource During Expected Lifetime**

This vulnerability occurs when a product releases a resource that should remain active for continued use by itself or another actor [CWE24w]. This weakness specifically involves premature release of resource such as memory, or file descriptors [CWE24p] that should remain valid.

The impact is multifold, as it can lead to exposure of sensitive data through read operations on reallocated resources, denial of service due to modifications leading to crashes and possible execution of unauthorised code when released memory resources are involved in function calls or write operations.

## 2.2.6 Number Errors

### Integer Wrap-around Errors

Integer wrap-around errors consists of a class of arithmetic vulnerabilities that occur when numeric operations exceed the maximum or minimum bounds of their data type. The following errors exist; integer overflow [PBS24] and integer underflow [PLO24b]. These vulnerabilities occur when the arithmetic operations result in values outside the representable range, which causes the value to "wrap around" to an unexpected very small or very large number.

The impact of integer wrap-around errors is severe, as they can lead to undefined behaviour causing system crashes, resource exhaustion through infinite loops and memory corruption through buffer overflows. Exploiting these errors could lead to evasion of protection mechanisms, unauthorised code execution, or impacting security decisions around resource allocation and access controls. If the affected values are used in data operations it could lead to data corruption, potentially causing more severe security implications depending on the application context.

### CWE-369: Divide By Zero

Divide by zero, is an arithmetic vulnerability where a program performs division operations with a zero denominator [Com24b]. The vulnerability can occur in situations where improperly validated variables or user supplied values are used as denominators.

The main impacts of this vulnerability are runtime exceptions that cause system crashes or program termination. It has the potential to cause major downtime and reliability problems in systems by triggering denial of service.

### CWE-1339: Insufficient Precision or Accuracy of a Real Number

This CWE identifies a vulnerability where software systems fail to maintain adequate precision in real number processing, particularly in calculations requiring high accuracy [CWE24h]. This weakness occurs when the implementation cannot preserve enough precision when representing the fraction, due how the real number system is implemented.

The vulnerability can crash systems by producing undefined results, generate wrong calculations that put critical operations at risk and trigger buffer overflows that could let attackers run malicious code.

## 2.2.7 Developer Oversights

This section contains runtime faults that have to do with bad coding practices, logical errors and other oversights a developer might make while programming.

### CWE-193: Off-by-one Error

Off-by-one errors occur when software calculations or iterations miss the intended bounds by 1, typically due to mishandled array indices, loop conditions or buffer boundaries [PLO24c].

The vulnerability can result in system crashes, memory corruption and resource exhaustion. Moreover, these errors may enable unauthorised code execution or denial of service conditions, due to their difficulty in detection during testing.

## Expression Issues

Expression issues commonly arise from typos, where a different operator is used than the intended one, due to syntactic similarities. For example, using assignment = instead of comparison == operators and vice versa [CLA24i, CLA24j].

The impact of operator misuse can be serious. For example, using an assignment instead of comparison can change execution logic potentially bypassing security checks. On the other hand, comparing instead of assigning prevents the variable assignment, which could lead to unexpected program states and other failures.

### **CWE-563: Assignment to Variable without Use**

Assignment to Variable without Use, represents a weakness where a variable is assigned a value but never utilised within the program [N/A24]. This usually occurs when a variable is instantly reassigned a different value or leaves its scope before being used.

While this causes the code to be less efficient, it is also an indication of logical problems in the program flow or redundant code. Moreover, these unused variables can obscure code readability and complicate code maintenance causing decreased software quality and higher risk of future defects.

### **CWE-1108: Excessive Reliance on Global Variables**

Excessive Reliance on Global Variables, represents a common weakness in software design where programs overuse global variables for state management [CWE24a]. This usually occurs due to the lack of proper encapsulation and maintenance of data within local scopes.

This vulnerability creates a subtle problem, as excessive use of global variables makes code harder to read and maintain. Furthermore, the attack surface is increased as there are multiple points where the integrity of a system could be compromised. This complicates vulnerability discovery efforts and security audits, potentially extending the window of exposure for identified vulnerabilities.

# Dash Language Design

## 3.1 Philosophy

In addition to the DOP principles described in section 2.1.1, Dash builds upon the following core goals:

- **Simplicity:** The language provides core functionality for software development without unnecessary features. There should typically be one clear way to write code, making the language easier to learn and use effectively.
- **Safety:** The language should implement comprehensive compile-time checks and runtime guards to prevent software faults.
- **Explicitness:** The language should force developers to clearly express their intentions in code, making program behaviour immediately apparent for improved code readability and maintainability.
- **Performance:** The language should enable writing efficient code by default. Costly operations should require conscious consent rather than being hidden abstractions.
- **Robustness:** The language should prioritise creating reliable, fault-tolerant code. Language features actively guide developers toward robust implementations while making error-prone patterns difficult to express or at least explicit.

## 3.2 Grammar

The grammar for the Dash programming language has been formalised using Tree-Sitter [BC24] and can be found in the appendix under listing 7.1.

## 3.3 Core Features

This section introduces core features of the Dash programming language including syntax.

### 3.3.1 Assignments

All assignments in Dash start with either `let` or `var`, as can be seen below:

```
1 let x = 1
2 var y = 2
```

**Listing 3.1:** Types of assignments in Dash

Dash will not compile if there are unused variables to mitigate the weakness described in section 2.2.7. This rule does not apply to public global variables.

## Immutable Variables

A variable defined using `let` is immutable and can also not be reassigned. This means the following is not allowed:

```
1 let x = 1
2 x = 2 // ILLEGAL
```

**Listing 3.2:** Example of incorrect reassignment of variable defined using `let` in Dash

Immutable variables can be defined outside of functions within a library (see section 3.8.1). This allows programs to define constants that are accessible within the entire library or also to other programs if the library and variable are public, more on that in section 3.8.2.

## Reassignable Variables

In Dash, reassignable variables provide a special form of mutability that only affects variable references in the current scope, not the underlying data. While all data in Dash remains immutable, the reference that a reassignable variable holds can be updated, as shown below:

```
1 var x = 1
2 x = 2
```

**Listing 3.3:** Variable reassignment defined using `var` in Dash

This design raises important considerations when passing reassignable variables to functions. Consider the example in listing 3.4:

```
1 var x = 1
2 some_func(&x)
3 x = 2
```

**Listing 3.4:** Example of variable reassignment with function call in Dash

Even if `some_func()` retains a reference to `x` that outlives the function call, the subsequent reassignment of `x = 2` cannot affect the value that `some_func()` received. This safety is guaranteed because the function only has access to the immutable value it was originally passed, regardless of any later reassignments to the variable even if the address of that value was taken.

It should also be noted that reassignable variables may only be defined within a function block and not in global scope.

## Multiple Declarations

Dash allows the declaration of multiple variables within a single assignment statement. As can be seen in listing 3.5 an immutable variable `x` and a reassignable variable `y` is defined.

```
1 let, x var y = 1, 2
```

**Listing 3.5:** Multiple variable assignments in Dash

It is also possible to do this with function calls (see listing 3.6), where the return value of `some_func()` is assigned to `y` and `z`.

```
1 let, x let y, let z = 1, some_func()
```

**Listing 3.6:** Multiple assignments with literal and function call in Dash

## 3.3.2 For Loops

Dash includes different types of for loops to address a variety of iteration patterns.

### Classic Loop

The conventional C-style for loop allows for explicit initialisation, condition and increment statements:

```
1 for i = 0; i < 10; i++ {  
2     // ..  
3 }
```

**Listing 3.7:** C-style for loop in Dash

This pattern is useful when precise control is required.

### Range Loop

For range loops offer a simple syntax for iterating over arrays. There are two variations, one where only the index is accessible:

```
1 for i in arr {  
2     // ..  
3 }
```

**Listing 3.8:** For range loop in Dash

The other variation where the element is also available:

```
1 for i, e in arr {  
2     // ..  
3 }
```

**Listing 3.9:** For range loop in Dash with element

### Boolean Loop

A conditional loop that continues as long as a boolean expression returns true:

```
1 for queue.has_more(q) {  
2     // ..  
3 }
```

**Listing 3.10:** Boolean for loop in Dash

This is useful in cases where iteration is based on a dynamic or complex condition.

## Infinite Loop

A simplified syntax for generating infinite loops:

```
1 for {
2     // ..
3 }
```

**Listing 3.11:** Infinite for loop in Dash

## 3.3.3 Functions

Functions in Dash are defined using the `fn` keyword followed by the function name, parameter list and return types. The function statement may also be prepended with the `pub` keyword marking the function as public. To learn how this affects the ability for other libraries to import the function see section 3.8.1. The function's signature provides explicit information about its inputs and outputs, as shown in listing 3.12.

```
1 pub fn some_func() i64, bool {
2     return 0, false
3 }
```

**Listing 3.12:** Function definition in Dash

A function marked with `!`, as shown in listing 3.13, indicates that it may cause an error during execution. This approach promotes more transparent and maintainable code by making errors evident inside the type system itself, as developers can immediately see which function calls require error handling without having to check the implementation details or documentation. Refer to section 3.7 for further information on handling errors.

```
1 fn some_func()! i64 {
2     return 0
3 }
```

**Listing 3.13:** Error-prone function definition in Dash

## 3.3.4 Pipe Operator

Dash's pipe operator `|>` provides a simple syntax for function composition, allowing data to go through a sequence of modifications while keeping the code readable. This operator is very useful in DOP, where data frequently undergoes several sequential transformations. For example:

```
1 let arr = str
2     |> strip_whitespace
3     |> to_array
```

**Listing 3.14:** Pipe operator in Dash

This syntax eliminates deeply nested function calls and clearly shows the transformation sequence.

Moreover, the pipe operator seamlessly integrates with Dash's error handling (see section 3.7). When used with error-prone functions, only a single `try` or `catch` is required at the chain's beginning or end respectively:

```
1 let res = try buf
2     |> to_string
3     |> evaluate
```

**Listing 3.15:** Error handling with pipe operator

If any function in the pipeline raises an error, it propagates to the caller without requiring additional error handling at each step.

### 3.3.5 Modifying Structs

There are two primary methods for modifying structs in Dash. The first is the traditional approach of initialising a new struct instance with modified field values. While simple, this approach can become cumbersome when dealing with big structures or when only a few fields need to be modified.

To address these limitations, Dash provides a more concise copy-and-update operator that builds upon the standard shallow copy operator `^`. This operator allows for selective field updates while copying the values of unmodified fields:

```
1 let new = old^{
2     new.some_field = 2
3 }
```

**Listing 3.16:** Copy and update operator

The copy-and-update operator is particularly useful when working with generic structs. In such cases, the traditional approach of creating a new instance becomes problematic because the concrete type information is not available at the point of modification. This makes it impossible to properly initialise a new instance with the desired modifications. An example of this limitation and how the copy-and-update operator resolves it can be found in section 3.3.6, specifically in example listing 3.18.

### 3.3.6 Generic Structs

Dash implements generic structs as a cornerstone of its DOP paradigm, providing a flexible mechanism for type abstraction. This feature enables the definition of structural types that can match multiple concrete implementations based on field names and their corresponding types.

A generic struct declaration specifies a structural contract that the types must satisfy, as shown below:

```
1 gen struct sword {
2     strength i64
3 }
```

**Listing 3.17:** Generic struct definition

Functions accepting generic structs as parameters can operate on any struct type that satisfies the requirements defined by the generic struct, regardless of field ordering:

```
1 fn weaken_sword(s sword) sword {
2     let s' = s^{
3         s'.strength = s.strength - 10
4     }
5     return s'
```

```
6 }
```

**Listing 3.18:** Function accepting generic struct and modifying it

This generic struct system enables two powerful programming patterns:

- **Safety and Flexibility:** The system provides compile-time structural compatibility guarantees while retaining type flexibility.
- **Data Evolution:** Programs can be built to handle future type changes without modifying existing generic functions, providing extensibility.

While this solution provides flexibility, it differs from traditional DOP in numerous ways, as described in section 2.1.1. The requirement for explicit generic struct definitions increases verbosity when compared to employing hash maps and dynamic typing. This design decision sacrifices some convenience for type safety and performance.

### 3.3.7 Deferring Code

Dash’s defer mechanism implements lexically-scoped delayed execution of functions or code blocks (see listing 3.19). This construct assures that clean-up tasks are carried out in reverse order when the function returns, regardless of the execution path. The close proximity of allocation and deallocation operations in the code structure helps prevent resource leaks by ensuring proper resource clean-up, reducing the likelihood of forgotten deallocation operations as discussed in section 2.2.5.

```
1 // defer single function
2 defer os.close(fd)
3
4 // defer block of code
5 defer {
6     // ...
7 }
```

**Listing 3.19:** Deferring functions and code

### 3.3.8 Pattern Matching

Pattern matching is a key feature that allows for simple and type-safe handling of various data types. The language has three types of pattern matching; matching by value, matching by enum field and matching by type. Each form supports a specific use case while keeping a consistent syntax.

#### Match by Value

The most straightforward form of pattern matching is matching by value, which works with any literal type including scalar types and structs. This form allows direct comparison against specific values.

```
1 let x = 1
2 match x {
3 case 0:
4     //...
```

```
5 case _:  
6     //...  
7 }
```

**Listing 3.20:** Match by integer values

The default case, defined using `case _`, serves as a catch-all pattern that executes when none of the other cases match. This ensures exhaustive matching and prevents runtime errors from unhandled cases.

### Match by Enum Field

Dash allows pattern matching against enum variants, providing a clear syntax for handling different enum cases:

```
1 enum status {  
2     offline  
3     online  
4 }  
5  
6 match e {  
7 case status.offline:  
8     // ...  
9 case status.online:  
10    //...  
11 }
```

**Listing 3.21:** Match by enum field

### Match by Type

Union and error types can be matched against to handle different possible variants. This is useful when flexibility is required in the type system where the concrete type is only determined in specific parts of the code base:

```
1 union number {  
2     u64  
3     i64  
4 }  
5  
6 match n {  
7 case u64:  
8     // ...  
9 }
```

**Listing 3.22:** Match by union type

Moreover, it is also possible to assign a variable which will be the type of the matching case at runtime, dynamically. In the example below `n` is of type `number` while `n'` is the type of the case, which is determined at runtime:

```
1 union number {  
2     u64
```

```
3     i64
4 }
5
6 match n' = n {
7 case u64:
8     // ...
9 case i64:
10    //...
11 }
```

**Listing 3.23:** Match by union type with assignment

## Match Expressions

Match statements in Dash can also be used as expressions, allowing them to directly produce values. This allows for dynamic assignments which is useful in an immutable language:

```
1 let v = match s {
2     case status.offline: "system is offline"
3     case status.online: "system is online"
4 }
```

**Listing 3.24:** Match by enum field as expression

## 3.3.9 Annotations

Dash uses an annotation system to attach metadata to program elements using a concise @ prefix syntax. These annotations serve two functions: influencing semantic analysis during compilation and altering compilation behaviour. The language currently only allows built-in annotations, such as @packed for memory efficient struct layouts or @extern(c) for foreign function interface declarations (see section 3.3.10). Currently annotations can only be placed above structs or functions:

```
1 @packed
2 struct udp_header {
3     src_port u16
4     // ...
5 }
6
7 @extern(c)
8 pub fn write(c i64)
9
10 @deprecated("use new_func instead")
11 fn old_func() {
12     // ...
13 }
```

**Listing 3.25:** Annotation examples

Allowing developers to mark functions as deprecated alerts other developers at compile-time that they are using an old function, helping to reduce the likelihood of the weakness described in [Kin24a].

### 3.3.10 Extern C

Dash provides foreign function interface (FFI) support through two methods; direct linking of external C functions and compilation of Dash functions using the C calling convention via the `@extern(c)` annotation [Nor24]. This interoperability is important because C is the de facto standard protocol for cross-language FFI, with widespread support across programming languages [Tea24b,GJS+24,Com24d]. While the ability to link external C functions serves as a pragmatic bootstrapping mechanism, it exposes Dash to C's unsafe memory model. Dash supports automatic type conversion between native types and C counterparts, with certain limitations. The following types are specifically excluded from C interoperability by the type system; optional types, unions, enums, `int` and `float` (see section 3.4 for more information on Dash's type system). The language specification provides a detailed description of the type mapping between Dash and C under the section "Extern C" [Nor24].

It should be noted that the border between Dash and C is currently very fluid, with few defensive mechanisms in place to protect Dash's runtime integrity. While defensive strategies such as data copying could enhance isolation, they introduce performance overhead and may conflict with legitimate use cases, particularly when C functions require direct buffer access. A more robust solution would be introducing fine-grained annotations that enable developers to specify how data is used, allowing the compiler to implement appropriate safety measures while preserving optimal performance. Either way, it is an area that needs improvement in regards to safety.

## 3.4 Type System

All values in Dash require a type to be defined at compile-time, whether this type is explicitly defined or implicitly derived depends on the context. Dash has strict type equivalence and assignability rules, as described in the language specification under the section "Types" [Nor24]. This means CWEs described in section 2.2.2 are not possible. The following sections describe Dash's type system in more detail.

### 3.4.1 Primitive Types

Dash has the following types:

- Unsigned integer (8 to 64 bit)
- Signed integer (8 to 64 bit)
- Int
- Floating-point number (32 and 64 bit)
- Float
- Boolean
- Byte
- Char
- String
- Array
- Struct

Most of these types are self-explanatory but `int`, `float`, `char` and `string` type will be discussed in more detail in the following sections.

## Int Type

The `int` type refers to an arbitrary precision integer type that never overflows or underflows. When such behaviour is required due to some underlying mechanism or when claiming performance is of highest importance then it is allowed to use the set bit-width integers. In other programming languages `int` is also known as `BigInteger` in Java [Ja23] or `BigInt` in Golang [Gol24a]. This is the recommended type when working with integers as it prevents all CWEs described in section 2.2.2.

## Float Type

The `float` type in Dash implements arbitrary-precision floating-point arithmetic. This means that, unlike fixed-precision floating-point types such as IEEE 754's `double` [Ass20], Dash's float maintain extremely high precision and can only overflow in rare cases with a very large exponent e.g. below  $1 - 2^{30}$  or above  $2^{30} - 1$  [Lef23].

Fixed bit-width floating-point numbers should only be used when a lack of precision can be tolerated and performance is required, otherwise it is recommended to use `float`. This is because it mitigates overflow and underflow vulnerabilities including the lack of precision and accuracy vulnerability discussed in section 2.2.6.

## Char Type

The `char` type refers to a Unicode code point [Con22], which under the hood is represented using a `u32` type. Characters are defined using single quotes, such as `'a'`. Some letters prepended with `\` hold special semantic meaning such as the newline escape sequence: `\n`. These are escaped using the backslash. For a full list of special char values see the language specification under section "Char literal" [Nor24].

```
1 let ch = 'a'
2 let nl = '\n'
```

It is also possible to define a Unicode code point using 2, 4 or 8 hexadecimal digits as can be seen below:

```
1 let c1 = '\uE4'
2 let c2 = '\u00E4'
3 let c3 = '\u000000E4'
```

**Listing 3.26:** Defining unicode code points of varying length

This allows for accurate character representation while keeping syntactic clarity. The language can handle both simple character literals and complicated Unicode encodings.

## String Type

Strings in Dash are made up of a length and a pointer to a byte array. This approach avoids null termination, removing buffer overflow problems caused by missing terminators, as described in section 2.2.2. By keeping length as an explicit field, the language allows for constant-time length operations, memory-safe manipulations and efficient substring operations. This means miscalculating the string length can not occur. Note that, a developer can still assume the string length is equivalent to the Unicode character count. This oversight is difficult to mitigate as its a logic error but having a solid standard library providing well-tested and documented Unicode manipulation functions could help reduce the risk.

## 3.4.2 Type Inference

Type inference only occurs during compilation under some circumstances, leading to all types being explicitly defined afterwards. Types only need to be inferred when assigning a value to a variable and when a literal is used. Literal types are inferred using simple coalescing rules to match the expected type e.g. when passing a literal to a function or setting it in a struct. To learn more on how Dash handles type inference in those circumstances see section "Literal inference" in the language specification [Nor24].

## 3.4.3 Type Casting

Dash uses a restricted, explicit conversion mechanism that prioritises type safety and predictable behaviour. The language forbids implicit type conversions, requiring developers to explicitly state their intent when converting between types. In many cases conversions can occur without any error handling. Forcefully casting between two unrelated types is never allowed. These cast rules prevent multiple CWEs namely, type confusion described in section 2.2.2 and type cast issues described in section 2.2.2.

Dash distinguishes between three categories when it comes to type conversions; legal, error-prone and illegal. To see the precise casting rules visit the language reference under section "Types" [Nor24]. To better understand how error-prone operations are handled see section 3.7.

## 3.4.4 Type Definition and Aliases

A type definition allows a developer to define a custom type in Dash using the `type` keyword:

```
1 type age i64
```

**Listing 3.27:** Defining a custom type in Dash

Type definitions enable developers to create distinct types to prevent logical errors where two semantically different values might share the same underlying type. This type safety is complemented by increased code readability through meaningful identifiers, while also enabling type checking to catch errors at compile-time. Furthermore, type definitions support better documentation by making the code's intent clearer to other developers.

In contrast, a type alias is defined using the `alias` keyword:

```
1 alias age i64
```

**Listing 3.28:** Defining a type alias in Dash

Type aliases serve a different yet complementary role in the type system. They provide a mechanism for creating shorter or more descriptive names for other types without enforcing strict type safety.

The key distinction between type definitions and aliases is demonstrated in the following example:

```
1 type age i64
2 alias time_span i64
3
4 fn example() {
5     var age = age(25)
6     var span = time_span(25)
7 }
```

```

8     span = age // valid
9     age = span // ILLEGAL
10  }

```

**Listing 3.29:** Type definition vs alias behaviour

### 3.4.5 Type Predicates and Dirty Types

In many cases it is necessary to specify additional constraints which a type needs to satisfy to truly be of that type. Dash allows this through type predicates that apply to the value of the type and guard the type. As can be seen in listing 3.30, if a custom type `age` needs to be defined to only include ages from 18 to 65 it would be done through type predicates. This implements a core component of DOP described in section 2.1.5.

```

1 type age i64 | 18 <= age && age <= 65

```

**Listing 3.30:** Example of a type predicate in Dash

Moreover, these type predicates also apply to struct fields allowing for thorough validation of aggregate types:

```

1 struct user {
2     name string | len(name) != 0
3     age i64      | 18 <= age && age <= 65
4 }

```

**Listing 3.31:** Struct field predicates in Dash

These predicates are more powerful than just being able to validate a type, as unvalidated types are marked at a type system level through `dirty` types. For example, initialising a variable with type `age` requires it to be validated before it is truly of type `age`;

```

1 let u = age(18)
2 // u: dirty<age>
3 let u' = try validate(young_user)
4 // u': age

```

**Listing 3.32:** Validating dirty type in Dash

This forces a developer to validate unknown input otherwise the program would not compile preventing CWEs such as the improper validation of input [CWE24e] at a type system level. Validating a type using its predicates also requires error handling. For more on error handling see section 3.7.

### 3.4.6 Optional Types

As pointers can not be `null` in Dash, another way to represent nothing is required. This is done through optional types. Note that optional types may only be used within function signatures and not within type definitions or aliases to prevent obscuring the type. To define an optional type a `?` needs to be prepended in front of the type as can be seen in listing 3.33.

```

1 fn check_null(u ?user) user { }

```

**Listing 3.33:** Function accepting optional type

## Operating with Optional Types

Dash provides three ways to operate on optional type. The first is null coalescing, which allows defining a default value:

```
1 fn check_null(u ?user) user {
2     return u ?? user{name: "henrik", age: 23}
3 }
```

**Listing 3.34:** Null coalesce operator usage

The second mechanism is simply checking if the optional is equal to `null`. Dash should detect that a null check for the optional variable occurred within or above the current scope allowing for safe use afterwards, as can be seen below:

```
1 fn check_null(u ?user) user {
2     if u == null {
3         // ...
4     }
5     return u
6 }
```

**Listing 3.35:** Conditional null check

The third method is called force unwrapping, denoted by the `?` operator. This operation requires an explicit error annotation `!` on the function signature, as it may fail at runtime. For more information on error handling, see section 3.7.

```
1 fn check_null(u ?user)! user {
2     return ?u
3 }
```

**Listing 3.36:** Force unwrap operation

When combining force unwrap with pointer dereferencing, operator precedence can become ambiguous. To improve code readability by avoiding using parentheses, the placement of `?` and `*` determines the operation order: `?*u` first dereferences then unwraps, while `*?u` first unwraps then dereferences:

```
1 let u1 = ?*u
2 let u2 = *?u
```

**Listing 3.37:** Unwrap and dereference precedence

### 3.4.7 Tagged Unions

Tagged unions allow for variables that can hold values of different types, with runtime checking to ensure type safety. Unlike untagged unions, which can lead to type confusion and undefined behaviour (see section 2.2.2), tagged unions keep track of which variant is currently stored, making them a safer alternative.

In Dash, tagged unions provide compile-time type checking while maintaining runtime type information. Each value of a union type carries both its data and a type descriptor indicating which variant it contains. This enables safe pattern matching and eliminates the possibility of accessing the wrong variant. Dash enforces exhaustive matching on union types, ensuring all possible cases are handled. See below an example union called `number`:

```
1 union number {
2     u8, u16, u32, u64
3     i8, i16, i32, i64
4     f32, f64
5 }
```

**Listing 3.38:** Union in Dash made up of integers and floats

### 3.4.8 Soundness

Dash's type system has not been proven to be sound yet. This is future work and is beyond the scope of this thesis.

For Dash's type system to be considered sound it needs to satisfy the following two theorems [Pie02a]:

- "Progress: A well-typed term is not stuck (either it is a value or it can take a step according to the evaluation rules)."
- "Preservation: If a well-typed term takes a step of evaluation, then the resulting term is also well"

## 3.5 Memory Model

### 3.5.1 Memory Allocation

Memory is allocated in two ways; either explicitly using the built-in function `make` and also implicitly when the compiler determines through escape analysis that data escapes. This means unlike C, returning stack allocated data would not cause an error when the caller attempts to access it as the compiler would automatically heap allocate it. How this precisely is done is beyond the scope of this thesis.

#### Using Memory

When explicitly allocating memory using `make`, the returned type has a special structure. It consists of a `memory` type that contains the inner requested type. This is different compared to more conventional programming languages, but is required because of Dash's immutability and the limitations described in section 2.1.1.

```
1 let buf = make([]byte, 64)
2 // buf: memory<[]byte>
```

**Listing 3.39:** Allocating memory using `make`

The memory type gets converted to the underlying type when used, allowing for brief linear mutability within the `use` block necessary to overcome the limitations described in section 2.1.1.

```
1 let buf = make([]byte, 64)
2 let buf' = use buf {
3     for i = 0; i < len(buf); i++ {
4         buf[i] = byte(i)
5     }
```

```
6 }  
7 // buf': []byte
```

**Listing 3.40:** Using memory

The trade-off for limited linear mutability within the use block is that memory can not be passed to functions within the block and if elements are passed, they remain immutable. This contains the mutability to a lexically limited scope.

Moreover, it creates a clear separation within the type system between usable memory and already allocated memory. After using the memory, `buf` becomes unusable and attempts to use it result in a compile-time error, as memory can only be used once.

## 3.5.2 Memory Deallocation

Dash's memory model should alleviate the burden of manual memory allocation and eliminate unsafe memory operations. There are multiple ways this could be achieved; through reference counting, garbage collection or region-based memory management [SBF12, BCM04a, TT97].

Dash uses a combination of an optimised version of reference counting and static analysis of lifetimes to determine when heap allocated data should be freed. While reference counting is one of the oldest ways to manage memory [Col60], it works nicely with the Dash programming language due to certain attributes of the language such as immutability. Moreover, it enables structural sharing by allowing multiple pointers to reference the same data (see section 2.1.1).

One of the drawbacks of reference counting is that research suggests it can incur a significant performance hit during `free` operations when it has to recursively deallocate large data structures causing pauses [Boe04]. Research demonstrates that a performant reference counting implementation for Lean, an immutable and eager functional language, successfully mitigated the main drawbacks and even outperformed state-of-the-art implementations in certain cases [UdM21]. This shows great potential for Dash's use case, where data is also immutable and executions occur imperatively.

### When is a variable counted?

A heap allocated variable, which consists of a struct or array, is only counted if:

1. a reference of the variable was assigned to a struct field
2. a reference of the variable was assigned to an array element
3. a reference of the variable was used within body of a function closure
4. a reference of the variable was used in an error
5. a slice of the variable was taken (only applies to arrays)

In the above described five cases it can be said that the variable was "consumed" by the operation.

Types that have to be counted require explicit `+` annotation in front of the type. See the example listing 3.41 below. This is enforced through semantic analysis like for any other type. The `+` annotation automatically implies the data is a pointer, which means `++` or `*+` is not allowed as its a tautology. This makes it clear to the developer that variables they pass will be counted or if a counted variable is returned. It also allows for the compiler to efficiently initialise counting when the variable is created. Moreover, it speeds up statical analysis by not having to parse function bodies and nested functions to determine when counting is required.

```

1 struct abc {
2     f +xyz
3 }
4 var x = +xyz{} // x needs to be marked as counted
5 var y = abc{f: x}

```

**Listing 3.41:** Counted type in Dash using special type signature

If the developer wishes to avoid counting, a shallow copy of the heap allocated variable can be made using the `^` operator. Copying pointers is not allowed, as the consumption rules described above apply to the underlying data.

## What about cycles?

The major drawback of reference counting is that cycles in data structures lead to the data never being reclaimed [McB63]. This is not possible in Dash due to its immutability. Consider the following examples:

```

1 let x = node{}
2 let y = node{child: &x}
3 let x' = x^{
4     x'.child = &y
5 }

```

**Listing 3.42:** Immutability prevents cyclical structures in Dash

In the above example no cycle is possible as `x` can not be mutated to reference `y`. An immutable update using the copy-update expression is possible but `y` keeps referencing `x` and not `x'`.

The following example demonstrates a case where it seems like a cycle is created in Dash:

```

1 let m = make([]*node, 2)
2 let m' = use m {
3     m'[0] = &node{}
4     m'[1] = &node{child: m'[0]}
5     m'[0] = &node{child: m'[1]}
6 }

```

This is not an issue, because none of the structs are counted as they are literals. Moreover, when using `make` the memory is owned by the variable assigned to it, which in this case is `m` and not the individual underlying elements. Using outside variables within `m'` can not lead to a cycle due to immutability as was demonstrated in listing 3.42 above.

## When is a variable statically freed?

All variables that are consumed are marked as counted which allows the remaining variables lifetime to be statically determined using a simple algorithm. The algorithm requires variables in the abstract syntax tree to be marked as heap allocated if they escape or are too large to fit in a register or stack. For each function block, starting with the root function, the instructions are looked at in reverse. For each non-counted variable that is heap allocated a `free` instruction is placed after its last use and in every possible control flow path that the variable can take. How "last use" is defined depends on how the variable is used:

- If the variable is read and later not used any more then a `free` instruction is placed after the read.

- If a variable is used within a loop the free call should be placed outside the loop so that its freed when the loop exits. Additionally, if there is a `return` or `raise` instruction within the loop then a free call should be placed before those instructions.
- If a variable is copied then a free instruction is placed after the copy instruction.
- If the variable is passed to a function, analysis continue within the function to determine the location of the `free` call.
- If the variable is passed to a recursive function the same rules apply as for a normal function as there are only three possibilities for the variable within the recursive function; it is either passed recursively, its copied before being passed recursively or only part of the variable is passed recursively. This is demonstrated in listing 3.43
- If the variable is used in a defer statement a free instruction is placed after the defer statement ran.

Note that multiple rules may apply at the same time leading to numerous free calls being placed so that the variable is freed no matter what path is taken at runtime.

```
1 fn test1(a abc) {
2     if a.x == 0 {
3         // a goes out of scope here
4         return
5     }
6     test1(a)
7 }
8
9 fn test2(a abc) {
10    if a.x == 0 {
11        // a goes out of scope here
12        return
13    }
14    // a goes out of scope here (after copy is done)
15    test2(a^)
16 }
17
18 fn test3(a abc) {
19    if a.x == 0 {
20        // a goes out of scope here
21        return
22    }
23    // a goes out of scope here (after x is read)
24    test3(abc{x: a.x})
25 }
```

**Listing 3.43:** Lifetime of variables in recursive functions assuming 'a' is heap allocated

## Implementation

Reference counting in Dash would have an overhead of 8 bytes per reference. Moreover, if it can be shown that the data is not shared between threads, the increment and decrement operation can occur very quickly as its a simple add or sub operation. No atomic compare-and-swap (CAS)

operations are required as no synchronisation is needed. This means reference counting can be implemented as an efficient fat pointer in many cases. If data is shared between threads a CAS operation is required to perform the addition in a thread-safe way [UdM21] which generally is a lot more expensive [Boe04]. There are optimisations that can be performed to increase performance in concurrent environments such as through Biased Reference Counting [CST18].

Moreover while garbage collectors have to track all heap allocated objects, this implementation only needs to track heap allocated data that was consumed. The other heap allocated variables can be statically analysed at compile time to determine their lifetime as mentioned before. In addition, optimistic free calls can be invoked leading to eager freeing of data. The last part will need to be measured carefully as there is research suggesting long pause times can occur in worst-case scenarios [Boe04].

Reaching the maximum representable number when using a 64 bit integer will likely not occur any time soon. In the best case, where each reference points to nothing, it would require  $(2^{64} - 1) * 64$  bytes, 147.57 Exabytes, of memory to remember all the counts.

## Optimisations

There are various optimisations that can be performed to further increase the performance of reference counting:

- Destructive updates; if a variable is shown to not have any references to it the memory location can be reused when updating [JL96].
- Avoiding CAS operations; if data is not shared between threads no CAS operation is required
- Optimising CAS usage; if a CAS operation is required for synchronisation, Biased Reference Counting can be used [CST18].
- Borrowed references; if a reference is taken without owning the variable and it can be shown that the variable continues to exist at least as long as the reference then the counter does not need to be updated [UdM21].
- Reduce bits of count; only using limited bits for reference counting, since most data has very low maximum reference counts [SBF12].
- Eliding reference counting; if data is short-lived avoid counting it [SBF12].
- Coalescing reference mutations; if there is a chain of reference mutations the system can avoid intermediate operations [SBF12].

### 3.5.3 Safety

The automatic deallocation strategy described before mitigates the following vulnerabilities described in sections 2.2.1 and 2.2.3:

- CWE-401: Missing release of memory after effective lifetime
- CWE-761: Free of pointer not at start of buffer
- CWE-762: Mismatched memory management routine
- CWE-825: Dereferencing expired pointer

Moreover, Dash enforces bound checking when indexing and slicing arrays. This prevents nearly all vulnerabilities described in section 2.2.1:

- CWE-120: Buffer overflow from unchecked buffer copies

- CWE-124: Buffer underwrite/underflow issues
- CWE-125: Out-of-bounds read operations
- CWE-129: Improper validation of array indices
- CWE-787: Out-of-bounds write operations
- CWE-805: Buffer access with incorrect length values

While many errors are prevented or caught by Dash can not stop a developer from making logic errors. For example, Dash can not prevent off-by-one errors described in section 2.2.7. Although, an error would be raised if the access is out of bounds.

These safety checks prevent memory corruption, unauthorised data access and potential system crashes that could otherwise be exploited for arbitrary code execution. Moreover, index and slice operations are error-prone operations which is discussed further in section 3.7.

### 3.5.4 Pointers

Dash implements several restrictions on pointers to prevent common vulnerabilities described in section 2.2.3:

- No pointer arithmetic: Prevents incorrect pointer scaling (CWE-468), incorrect pointer subtraction (CWE-469) and out-of-range pointer offset (CWE-823) vulnerabilities
- Non-nullable pointers: Eliminates NULL pointer dereference (CWE-476) issues
- Strict typing: Prevents accessing child of non-structure pointer (CWE-588)
- All variables need to be initialised: Prevents accessing uninitialised pointers (CWE-824)

## 3.6 Polymorphism

Polymorphism in Dash allows for the description of abstract algorithms and data structures via numerous techniques, each fulfilling a specific use case in the type system. The language allows for both parametric and ad-hoc polymorphism.

### 3.6.1 Parametric Polymorphism

Dash offers parametric polymorphism via a powerful generics system that allows for type-safe generic programming. Dash's generic types are universally quantified, allowing functions and data structures to perform consistently across multiple concrete types while maintaining type safety at compile-time.

The type system allows for constraint declaration, with `T any` indicating an unconstrained type parameter:

```

1 struct node<T any> {
2     v T
3     child ?*node<T>
4 }
5
6 fn pop_stack<T any>(n *node<T>) T { }
```

**Listing 3.44:** Generic stack using generics

## 3.6.2 Ad-hoc Polymorphism

Unlike parametric polymorphism, ad-hoc polymorphism allows for specialised implementations of various types. Ad-hoc polymorphism is implemented through function overloading, which allows function implementations with different parameter types to share the same name (see listing 3.45). This approach allows for type-specific behaviour while exposing a unified interface to callers. Moreover, this offers optimal runtime performance as overloaded functions are resolved at compile-time.

```
1
2 pub fn to_string {
3     i64_to_string
4     bool_to_string
5 }
6
7 fn i64_to_string(i i64) string {
8     // ..
9 }
10
11 fn bool_to_string(b bool) string {
12     // ..
13 }
```

**Listing 3.45:** Function overloading in Dash

## 3.7 Error Handling

Dash follows the error as value paradigm popularised by Golang [Pik15], which provides a clear and explicit approach to error handling. While the language enforces strict error handling, it offers convenient abstractions to define, handle and propagate errors. The following sections elaborate on each stage of the error handling life-cycle.

### 3.7.1 Defining Errors

Defining errors in Dash is straightforward and follows an algebraic type pattern similar to how enumerations work in Rust [Com24c]. The language distinguishes between two types of errors, static and dynamic (see listing 3.46). Static errors can be declared with an identifier without parameters or using constant literals as parameters, making them immutable at runtime. Dynamic errors, on the other hand, take type parameters that can be populated with different values at runtime, allowing for context-dependent error information. This enables developers to select the most appropriate error representation for their error handling needs.

```
1 // Static errors
2 error division_by_zero
3 error negative_number(rsn: "number was negative")
4
5 // Dynamic errors
6 error out_of_bounds(index i64, size i64)
```

**Listing 3.46:** Error definitions in Dash

### 3.7.2 Raising Errors

Errors are raised using the `raise` keyword, similar to how values are returned from functions (see listing 3.47). Functions that can raise errors must be marked with an exclamation mark `!` to signify that they are error-prone:

```
1 fn divide(a, b i64)! i64 {
2     if b == 0 {
3         raise division_by_zero
4     }
5     return a / b
6 }
7
8 fn get_element(arr []i64, index i64)! i64 {
9     if index >= len(arr) {
10        raise out_of_bounds(index, len(arr))
11    }
12    return arr[index]
13 }
```

**Listing 3.47:** Raising errors in Dash

### 3.7.3 Catching Errors

When working with error-prone functions, developers have two options. They can use `try` to propagate errors up the call stack without any intermediate error handling:

```
1 fn process_data(arr []i64)! i64 {
2     let value = try get_element(arr, 0)
3     return value * 2
4 }
```

**Listing 3.48:** Error propagation in Dash

On the other hand, they can use `catch` to handle errors explicitly when additional error handling logic is needed:

```
1 fn safe_process(arr []i64) i64 {
2     let value = get_element(arr, 0) catch e {
3         // ...
4     }
5     return value * 2
6 }
```

**Listing 3.49:** Error catching in Dash

It is also possible to directly match against errors for more fine grained control within the catch block. The only drawback is that the semantics within the catch block work like a match statement, meaning no other expressions are allowed apart from the cases:

```
1 fn safe_process(arr []i64) i64 {
2     let value = get_element(arr, 0) catch e {
3         case out_of_bounds(index, size):
```

```

4         // ...
5     case _:
6         // ...
7     }
8     return value * 2
9 }

```

**Listing 3.50:** Error catching in Dash through matching

Dash does not allow errors to go unhandled; either a default value is set or the error is re-raised. This prevents all weaknesses described in section 2.2.4 including the weakness described in section 2.2.4. Although, Dash allows defining a default case in the `catch` block meaning the generic error handling weakness defined in section 2.2.4 is not mitigated.

### 3.7.4 Error-prone Operations

Certain operations in Dash are inherently error-prone and require functions containing them to be marked with `!` (see listing 3.51 for examples). The compiler automatically detects these cases and ensures proper marking. Common error-prone operations include:

- Slicing and indexing outside of range-checked loops
- Dividing two numbers (as denominator could be zero)
- Force unwrapping
- Dynamic memory allocation using `make`
- All recursive function calls

These operations do not require explicit `try` statements in front of them. This is true for all listed operations except when a calling a recursive function, then the standard calling convention for error-prone functions apply. The compiler enforces this requirement to maintain clear error handling across the codebase.

```

1 fn example()! {
2     let arr = [1, 2, 3]
3     let value = arr[5] // fn requires '!' due to unsafe indexing
4     let result = 10 / 2 // fn requires '!' due to division
5     let mem = try make([i64, 1000) // fn requires '!' due to allocation
6     try recursive_func() // "try" required
7 }

```

**Listing 3.51:** Error-prone operations

The reason why Dash defines all recursive functions as error-prone is that proving a recursive function terminates is generally an undecidable problem as it is related to the well-known Halting Problem [Hro04]. There are cases where the termination can be proven, such methods have several limitations such as when the function terminates based on a "domain specific semantic argument" e.g. reaching a winning state in a game with complex game rules [BKN07]. In NASA's famous "The Power of Ten" rules they disallow using recursion due to the risks it can pose to mission-critical software [Hol06]. Recursive functions are also often the cause of the critical stack overflow vulnerability [CLA06] with similar consequences described in section 2.2.1. Although, they do allow for very elegant algorithmic solutions.

### 3.7.5 Global Error Handling

Dash cannot handle certain runtime errors at the function level, especially ones that affect the entire program execution environment. As explained in section 3.5.1, Dash uses an automated memory management technique that hides certain heap allocation details from developers. Because of this design choice, developers cannot always determine at development time whether specific data structures will be heap allocated. As a result, certain out-of-memory errors are treated as global failures in the system because their occurrence cannot be handled at specific allocation sites by the developer. The specific errors that are considered global errors is implementation specific.

While Dash's error handling model provides strong guarantees through its explicitness, it does present certain drawbacks:

1. **Error Path Obscurity:** As global error handlers populate the call stack, tracing the exact path taken by a global error becomes more difficult. This can complicate debugging and error analysis in large systems.
2. **Global Error Complexity:** The difference between function-level and global errors complicates error handling solutions since developers must consider both local and global error cases.

Hence using global error handlers should be reserved for critical clean up tasks such as gracefully shutting down a http service.

## 3.8 Code Organization

### 3.8.1 Libraries

A library is a collection of Dash files within a directory that may contain nested libraries in sub-directories. This means libraries mirror the filesystem structure. Each file within a directory must declare the same library name at the top of the file as can be seen in listing 3.52. Subdirectories are allowed to declare different library names, as long as every file within uses the same name.

```
1 lib main
```

**Listing 3.52:** Definition of a library

Currently library names have to be unique within the directory they are in. This means its illegal to have two directories in the same parent directory defined using the same library name, the same as having two directories named the same does not work in a filesystem.

Dash does not allow state management, such as variables defined using `var`, to live in the global library scope to help reduce complexity and to prevent the weakness described in section 2.2.7.

### 3.8.2 Importing

Libraries would not be very useful on their own without being able to be used by other libraries. The import system in Dash supports nested directory structures through path-based imports. As shown in listing 3.53, libraries are imported using their full path relative to the project root. The path components are separated by forward slashes `/`, similar to filesystem paths. This import

structure allows for better organization of code and helps prevent naming conflicts between different library modules. There is one exception to this rule and that is imports defined using `std` are part of the standard library and not part of the project.

```
1 lib main
2
3 import std/json
4 import path/to/library
```

**Listing 3.53:** Import statements in Dash

Another rule when it comes to importing libraries is that Dash does not support import cycles. This constraint keeps dependencies acyclic and avoids initialisation challenges.

The scope of accessibility for functions and libraries is controlled using the `pub` keyword. Table 3.1 depicts the effect of function-level and library-level visibility modifiers.

	Library private	Library public
Function private	Only local	Only local
Function public	Internal	External

**Table 3.1:** Function and library visibility matrix

Local visibility restricts function access to the current library, whereas public functions in a private library are only accessible by other libraries within the project. When both the function and library are public, the function is accessible externally by third parties. These visibility rules do not only apply to functions but also to global variables, struct definitions, type definitions, type aliases, unions and enums.

# Implementation

The current implementation of the Dash compiler, spanning over 20,000 lines of code with approximately half dedicated to testing, is written in the Go programming language leveraging the LLVM compiler backend as the optimiser and code generator [Tea24c]. The compiler frontend which consists of the lexical analyser (lexer), parser, semantic analyser and LLVM intermediate representation (IR) generator will be further explained.

## 4.1 Lexer

The lexer performs the first phase of compilation by transforming source code into a sequence of tokens, including source location and token type. This process is driven by the parser, where the next token is requested on the fly. By generating tokens incrementally, the system reduces its memory footprint, eliminating the need to save the entire token sequence. Moreover, the demand-driven approach guarantees that only necessary tokens are created, when parsing fails early or partial parsing is required.

Each generated token contains three properties:

- **Literal:** The actual text sequence from the source code that the token represents
- **Type:** The syntactic category (e.g. keyword, operator)
- **Source Position:** Line and column number in original source code

The implementation can be found in the directory "dash/src/lexer" which also contains the test cases [Nor24].

## 4.2 Parser

The parser consists of two complementing parsing strategies; a Pratt parser for expressions and a simple recursive descent parser for statements. This separation enables the proper handling of various language constructs while preserving parsing performance. The parser is deterministic, as in each step there is always only one valid action that can be taken with each token only parsed once. This means the parser runs in linear time  $O(n)$  without any backtracking required.

The implementation can be found in the directory "dash/src/parser" which also contains the test cases [Nor24].

## 4.2.1 Pratt Parser

The Pratt parser is not a novel parser but is very simple and powerful allowing efficient parsing of expressions using its core innovation; top-down operator precedence [Pra73]. This combination of recursive descent parsing with lexical semantics offers an elegant solution to expression parsing that avoids the complexity of traditional precedence climbing or grammar transformation techniques [CT22a].

The parsing mechanism is based on three basic components for each token. The first is the null denotation function, also known as `nud`, which interprets tokens that initiate expressions. This method handles cases where a token does not have a left context, such as prefix operators or atomic values. The second component, the left denotation function `led`, handles tokens with left context, which are often infix or postfix operators. Finally, the parser correctly generates hierarchical expressions since each token has a binding power that represents precedence relationships.

The parser uses a context variable, conventionally named `left`, to track the most recently parsed sub-expression's value. This approach handles infix operations efficiently, while expressing precedence and associativity rules without complex lookahead or backtracking.

The simplicity of the Pratt parser is demonstrated by the number of lines of code the core algorithm requires in Golang:

```
1 func (p *Parser) parseExpression(precedence int) ast.Expression {
2     prefix := p.prefixParseFns[p.curToken.Type]
3     if prefix == nil {
4         p.nextToken()
5         return nil
6     }
7     leftExp := prefix()
8
9     postfix := p.postfixParseFns[p.curToken.Type]
10    if postfix != nil {
11        leftExp = postfix(leftExp)
12    }
13
14    for precedence < p.curPrecedence() {
15        infix := p.infixParseFns[p.curToken.Type]
16        if infix == nil {
17            return leftExp
18        }
19        leftExp = infix(leftExp)
20    }
21
22    return leftExp
23 }
```

**Listing 4.1:** Core Pratt parser implemented in Golang

Pratt parsing is powerful because it approaches lexical and syntactic analysis in a cohesive manner. The combination of token-centric design and recursive descent concepts, provide simple precedence management and extensibility.

## 4.2.2 Recursive Descent Parser

Dash uses a simple one token look-ahead recursive descent parser to parse statements. This is possible because part of the grammar can be considered a LL(1) grammar. A LL(1) grammar is a context-free grammar that can be deterministically parsed from left to right creating a left most derivation by only looking one token ahead without any ambiguity [CT22b].

## 4.2.3 Error Recovery Strategy

The parser's error recovery strategy is quite simplistic. Once an invalid token is reached the parser raises a syntax error and stops parsing that specific statement or expression any further. This means the parser continues parsing the next statement, expression or sub-expression with the goal of returning all syntax errors found and not just the first:

```
1 fn test2() {  
2 //<- error: invalid token "fn" expected "  
3 fn test2() { }
```

**Listing 4.2:** Missing `}` when parsing

The parser deviates slightly from the language specification when it comes to the set of source code it accepts. Consider the expression `let add = 1 2`, where the developer left out the `+` operator. The parser incorrectly parses this input as two separate components; an assignment expression `let add = 1` followed by an isolated literal `2`. This goes against Dash's constraints, which restrict unassigned literals and expressions. While this anomaly exists, the semantic analyser discovers and reports these errors.

## 4.3 Semantic analyser

The semantic analyser walks the tree using a combination of a breadth first and depth first approach. The headers in a source file are analysed first which includes function signatures, structs, enums, unions, type aliases and type definitions. Headers are analysed twice, once to populate the symbol tables with required data and a second time to infer and validate everything is correct. These two passes of the headers are required as functions, types and variables can be defined out of order within the global scope. The second pass validates the global statements (excluding functions) such as ensuring structs do not have any illegal cycles and that the types defined exist. Functions are processed last and involve a depth first approach where each statement and expression is fully analysed before proceeding to the next statement in a function block.

The implementation can be found in the directory `"dash/src/semsis"` which also contains the rigorous test cases for the semantic analysis [Nor24].

### 4.3.1 Data Structures

The semantic analyser uses different types of data structures to validate code. The most basic one is a simple cache, which maps keys to values for example for global types and functions. The analyser also uses a stacked symbol table that stores variables and their type which gets scoped in as the analyser goes into a new block e.g. when entering a new if-else condition or function block. When the analysis of the block is completed the symbol table is popped off the stack. Moreover, the semantic analyser also uses a simple stack to remember what the current scope is. This helps ensure that certain statements and expressions are only used in permitted contexts such as the `break` keyword only being allowed within for loops.

### 4.3.2 Type Checking

In the Dash programming language all values need to have a valid type at compile-time, which is enforced through semantic analysis. This step ensures every expression in the abstract syntax tree (AST) has a type and that every literals type is coalesced or inferred depending on how it is used. The inference process works in a bottom-up fashion by visiting leaf nodes of the AST and working its way back up. If a literal is assigned to a variable the type is inferred, while if it is set in a struct, assigned to an array, passed to a function or returned from a function then the literal is coalesced as long as it is possible (see listing 4.3). Coalescing literals occurs to avoid any friction in the language by reducing the need for excessive type casting. For exact coalescing rules see the language specification under section "Literal inference" [Nor24].

```
1 let a = 1 // literal infered as default type "i64"
2 // a: i64
3
4 test(1) // literal coalesced to byte
5 test(256) // ILLEGAL, '256' overflows byte (u8)
6
7 fn test(b byte) { }
```

**Listing 4.3:** Literal type inference

During semantic analysis, the compiler uses a special `unknown` placeholder type that exists only within this compilation stage. The placeholder type is assigned to named types that have not yet been resolved to their concrete types, such as struct fields using custom types. The unknown named type is particularly important during the first pass of header analysis, with resolution to concrete types occurring during the second pass. These unknown types later serve as a verification mechanism to ensure no errors exist in the analyser.

### 4.3.3 Error Recovery Strategy

Once a semantic rule is broken the analyser stores which node in the AST it occurred at and what the error was. To prevent overwhelming the developer with cascading errors, the analyser suppresses related errors, particularly during type inference where faulty expressions may have undetermined types. If type inference fails, Dash skips further processing of statements and expressions that depend on knowing the type. While this process can still be improved upon and tested more rigorously it works reasonably well at finding semantic errors without having to rerun compilation to show new errors.

## 4.4 IR Generator

The goal of the IR generator is to walk the AST and generate the corresponding IR. The structure of the generator is very similar to the semantic analyser. Currently no additional lowering occurs, everything is directly generated to LLVM IR. The following sections explain the generation methodology and the low-level representation of high-level language constructs.

### 4.4.1 Generation Process

The IR generator uses a combination of breadth-first and depth-first traversal of the AST, with the objective of producing LLVM IR instructions. The generation process follows a hierarchical

order. The first step is to generate IR for the headers such as type definitions, global variables and function signatures. Then each statement and expression in the function block is generated in a depth first manner. Each statement generally has a corresponding function that generates the IR for it.

## 4.4.2 Type Representation

The following describes how high-level types are converted to LLVM IR. Note that the identifiers used for the struct names mainly serve a symbolic purpose and do not always reflect the actual identifier in the IR.

### Array

Arrays are represented as a struct containing the length and a pointer to the underlying data which lies in contiguous memory (type `T` simply stands for any Dash type):

```
1 struct array {
2     len i64
3     ptr *[]T
4 }
```

### String

String types follow a similar pattern to arrays but specifically handle byte sequences, where `len` is number of bytes:

```
1 struct string {
2     len i64
3     ptr *[]byte
4 }
```

### Union

Tagged unions combine a type descriptor with padding to accommodate the largest variant:

```
1 struct unions {
2     descriptor i64
3     padding []byte
4 }
```

The type descriptor is currently being generated by hashing the concatenated library and the type identifier. While this approach works adequately for single file compilation, ensuring uniqueness across a project would require incorporating the library's relative path into the hash computation. The padding ensures correct space allocation when unions are returned at runtime, enabling consistent memory allocation between caller and callee functions. The padding length is determined by computing the size of the union's largest type, which sets a fixed size for all possible union variants. For example, consider the following struct and union definitions and the corresponding generated IR:

```
1 struct abc {
2     x i64
```

```

3 }
4
5 union xyz {
6     abc
7 }

```

The generated IR would look like this:

```

1 // specific union type
2 struct union_abc {
3     desc u32
4     x i64
5 }
6
7 // generic union type
8 struct union_xyz {
9     desc u32
10    padding [8]byte
11 }

```

The reason why there also exists a specific union type named `union_abc`, is for the compiler to be able to generate the correct code to access the underlying data in cases where the union is matched against or casted to its concrete type.

## Enum

Enum types are represented as named integer, where the `id` field refers the index of the field within the enum statement:

```

1 struct enum {
2     id i64
3 }

```

Dash generates a separate LLVM struct definition for each enum rather than a single `enum` struct type. This is done to benefit from LLVM's type system through proper type checking.

## Optional

Optional types have distinct representations based on the underlying type. For scalar types, optionals are represented using the value directly within the struct as such:

```

1 struct optional {
2     is_valid bool
3     val T
4 }

```

For aggregate types, the optional field `val` merely points to the aggregate value:

```

1 struct optional {
2     is_valid bool
3     val *T
4 }

```

## Generic Structs

In Dash, generic structs enable fields to be matched and operated on based on their structure rather than precise type definitions. When the compiler encounters a generic struct being used as a function argument, struct field or array element, it generates an internal representation that facilitates runtime operations. This internal representation contains the base pointer to the start of the struct and an array of offsets corresponding to the generic struct's field definitions:

```
1 struct generic_struct {
2     base_ptr u64
3     offsets []u64
4 }
```

Tracking the offsets this way enables generic structs to match with concrete structs independent of field order:

```
1 gen struct sword {
2     strength i64
3     durability i64
4 }
5
6 struct strong_sword {
7     name string
8     durability i64
9     strength i64
10 }
```

In this scenario, the generic struct `sword` works with `strong_sword` because the `strength` field exists with the same type `i64`, even though they are defined out of order. The compiler creates the necessary offsets in the internal representation to appropriately identify and access the corresponding fields at runtime.

## Scalar Types

The remaining scalar types are directly mapped to LLVM types, as can be seen in table 4.1. To see detailed examples visit the unit tests that are found in "dash/src/generator/generator\_test.go" [Nor24].

Dash Type	LLVM Type
bool	i1
byte	i8 (unsigned)
char	i32 (unsigned)
u8..u128	i8..i128 (unsigned)
i8..i128	i8..i128 (signed)
f32	float
f64	double

**Table 4.1:** Mapping between dash primitive types and LLVM IR types

### 4.4.3 Copy-Update Expression

The Dash copy-update expression enables immutable updates of structs and arrays by first copying the source data, followed by modifying the desired parts according to the code within the block (see listing 3.16). The copying is done using the LLVM intrinsic called `llvm.memcpy` [Tea24d]. This operation can incur significant memory overhead as it requires allocating and copying the entire contiguous memory block, regardless of the size of the actual modification.

### 4.4.4 Pattern Matching

Pattern matching is implemented in the generator using LLVM's switch instruction [Tea24d]. The switch statement works by matching an integer-based scrutinee with a set of integers to determine which block of code to run, allowing direct comparison of unions, enums and primitive types. For union types, the type descriptor is used as the scrutinee, while enum matching uses the field index stored within the enum structure. Floating-point comparisons are handled by bitcast operations to unsigned 32-bit or 64-bit integers, depending on the value's width. The remaining types which include `bool`, `byte`, `char`, `u8..u64` and `i8..i64`, are all represented using integers under the hood making pattern matching straightforward. Matching against struct, array and string literals is currently not implemented.

### 4.4.5 Compiler Hints

LLVM offers a variety of compiler hints called parameter attributes [Tea24d], which can be applied to functions and arguments to provide additional semantic information to the compiler's optimisation stages. These attributes allow for fine-grained control over function and parameter behaviours, including memory access patterns (e.g. `readonly`), pointer characteristics (e.g. `nonnull`) and value constraints (e.g. `noundef`). While Dash does not presently use these optimisation hints, incorporating them could improve compiler optimisations by specifying invariants precisely.

## 4.5 Specification Validation

The compiler implementation of the language specification is validated using different testing techniques. Unit tests cover the core components; lexer, parser, semantic analyser and generator. As tests move through successive compilation stages, they slowly become integration tests, since each stage depends on the output of previous stages. To test the code generator, for example, valid input from a working parser is required. Moreover, all generated IR is also executed using LLVM's JIT compiler through its execution engine [Tea24e] to verify the code runs properly. These tests primarily run to detect fatal memory issues e.g. to ensure no segmentation faults occur.

Additionally, the compiler includes tests written in Dash itself to verify semantic correctness, proper code generation and execution. These tests verify error-free operation, where the expected values are compared with output of the tested functions. They can be found in the directory "dash/tests/" [Nor24]. There are planned improvements to the testing framework, such as implementing fuzz testing for the parser and semantic analyser.

## 4.6 Implementation Coverage

The current implementation represents a Minimum Viable Product that implements the core functionality defined in the specification. The following categories highlight some of the missing features:

### Runtime Features

- Error handling
- Heap memory allocation and management

### Library System

- Libraries spanning multiple files
- Import functionality for external libraries

### Pattern Matching

- String pattern matching
- Struct pattern matching

### Other Language Features

- Nested union types
- For-range loop constructs
- Struct field predicates
- Extended `char` support:
  - Multiple character literals
  - Escape sequence support
- Generic struct manipulation via copy-update operator



# Evaluation

## 5.1 Introduction

The qualitative evaluation focusses only on syntax, semantics and core language features, excluding tooling, ecosystem components, standard library coverage, compilation performance, concurrency models, language community factors, third-party library availability and runtime performance metrics. While the excluded considerations are extremely important to have a well rounded and productive programming language Dash is simply too novel to have these.

The original project goal was to implement a subset of JQ [Con24a] for JSON parsing and querying. However, due to the limitations of the current Dash compiler this was unfeasible. An arithmetic expression evaluator was chosen as an alternate project, to demonstrate Dash's core features while sticking to implementation constraints, specifically only being able to use stack memory. The entire source code for the arithmetic calculator for Dash, Golang and Rust can be found in the appendix under listing 7.2, listing 7.3 and listing 7.4 respectively.

For the evaluation, Golang and Rust were chosen as reference languages because they take unique but complementary approaches to language design. Golang embodies minimalist design principles and simple semantics, while Rust's ownership architecture ensures strong safety guarantees. These features offer useful context for examining Dash.

## 5.2 Comparison Criteria

Dash will be evaluated across seven categories to help understand how the language works and how it improves upon or constrains certain actions. These include; state management, type system, safety guarantees, error handling, abstraction mechanisms, code clarity and interoperability with other languages. For each category a predefined assessment criteria is used targeting specific features.

### State Managements

The state management evaluation focuses on how Dash handles state updates and whether these modifications add additional complexity. Moreover, the trade-offs between the immutable and mutable paradigms is analysed, considering how it impacts program complexity and performance.

The evaluation is carried out using the following assessment criteria:

- How is state updated?

- How is state encapsulated?
- How is state shared between different components?
- What are the trade-offs between immutable and mutable state?

## Type System

The type system evaluation looks at core mechanisms Dash uses to implement a sound type system with a focus on type inference capabilities, generic programming support, type definitions and type casting.

The following assessment criteria is used in this evaluation:

- What type inference capabilities exist and what are their limits?
- How are type conversions validated and secured?
- How are user-defined types supported?
- How can types be converted safely?

## Safety Guarantees

This section evaluates the fundamental safety guarantees provided by Dash in comparison to Golang and Rust. The analysis considers how null references are prevented or managed and what memory safety guarantees are implemented. Moreover, the evaluation explores mechanisms for preventing type safety violations and examines protections against numeric overflow and underflow conditions. Furthermore, it investigates how each language approaches the prevention of resource leaks.

This analysis uses the following assessment criteria:

- How are null references prevented or handled?
- What memory safety guarantees are provided?
- How are type safety violations prevented?
- What overflow/underflow protections exist?
- What attempts are made to prevent resource leaks?

## Error Handling

While error handling is not yet implemented in the Dash language the syntax and semantics will be analysed. This includes an analysis of error definition, error raising and error catching mechanisms.

The following assessment criteria will be used:

- How are errors defined?
- How are errors raised?
- How are errors caught?

## Abstraction Mechanisms

This section evaluates the abstraction mechanisms provided by Dash in comparison to the other two. The analysis examines the types of code abstraction available and mechanisms that facilitate code reuse. The evaluation also evaluates which methods the language offers to organise and modularise code.

The evaluation makes use of the following assessment criteria:

- What mechanisms exist for code abstraction and reuse?
- How is code organised and modularised?

## Code Clarity and Verbosity

Code clarity and verbosity are important aspects in a language that influence developer productivity and code maintainability. This examination looks at three core aspects of language expressiveness; implementation conciseness, syntactic abstractions and boilerplate reduction methods.

- How many lines of code are typically needed for common programming tasks?
- What syntactic sugar exists to reduce common patterns?
- What language features help reduce boilerplate code?

## Interoperability

Interoperability is an important aspect of programming languages, meaning how well a language can interact with code written in other languages usually the C programming language. This analysis looks at three important components of language interoperability; foreign function interface capabilities and the ability to expose functions to external languages.

The evaluation makes use of the following assessment criteria:

- Does the language allow code to call foreign functions?
- Can functions be exposed to other programming languages?

# 5.3 Language Comparison

## 5.3.1 State Management

Dash uses explicit copy-and-modify operations using the caret `^` operator to enforce immutable state patterns:

```
1 let l = old^{
2   l.ch = l.input[old.next_pos]
3   l.pos = old.next_pos
4   l.next_pos = old.next_pos + 1
5 }
```

This stands in contrast with Go's object-oriented approach, which uses direct mutation:

```
1 func (l *Lexer) readChar() {
2   if l.nextPos >= len(l.input) {
3       l.ch = 0
```

```

4     } else {
5         l.ch = l.input[l.nextPos]
6     }
7     l.pos = l.nextPos
8     l.nextPos++
9 }

```

While Golang’s strategy is simple, it provides very little protection against unintended modifications as state is shared through pointer passing. Moreover, it requires special care when working in concurrent environments to prevent race conditions. This means defensive programming strategies such as deep copying the data and mutual exclusion are required to be sure it is not modified by another piece of code.

Rust, on the other hand, implements a sophisticated state management system centred around ownership and borrowing rules [Con24b]:

```

1 fn skip_whitespace(&mut self) {
2     while self.ch == b' ' {
3         self.read_char();
4     }
5 }

```

**Listing 5.1:** Rust state management using mut

The `&mut self` option explicitly shows state modification capabilities, with the ownership system guaranteeing exclusive access to mutable state. This maintains memory safety while limiting concurrent modifications. The borrowing mechanism offers a sophisticated framework for controlled state sharing, with the compiler enforcing strict access rules via lifetime annotations and borrow checks. This method preserves the safety features of immutability while enabling fine-grained control over state modifications. It successfully enforces compile-time guarantees by preventing incorrect memory access and data races while staying flexible and performant.

Regarding state encapsulation, Dash takes a unique approach by making state completely public and relying on explicit function passing rather than encapsulation. This means any function receiving the data can read it and potentially return a modified immutable copy of it. In Golang and Rust state can be encapsulated by keeping struct fields private and thus only accessible by methods tied to that struct. The lack of encapsulation in Dash is considered a drawback when thinking in terms of OOP principles but it is actually the opposite in DOP, where one of the core principles states that data should be represented in a generic way section 2.1.1. If data could be tied to a specific function or library by making it private then it would not be generic any more, breaking the principle.

Dash passes state explicitly between functions, which makes data flow very clear:

```

1 fn evaluate_expression(e evaluator, p precedence) evaluator, i64 {
2     var e, var res = evaluate_prefix(e)
3     // ...
4 }

```

**Listing 5.2:** State is explicitly passed and returned

Although this makes the function signature and assignments more verbose, it is nothing out of the ordinary when it comes to DOP as functions have to be stateless. Dash attempts to mitigate this in certain cases using the pipe operator `|>` to abstract away chains of functions operating on data.

On the other hand, explicitly passing state introduces a significant drawback where developers must ensure they do not accidentally ignore a returned value or use an old version. This is because it can introduce subtle state bugs that are hard to detect at compile time. For example:

```
1 var e = new_evaluator(1)
2 _, let result = evaluate_expression(e, precedence.LOWEST)
3 // Bug: Using 'e' after this point would reference stale state
```

A possible solution to this problem could entail disallowing return values being ignored. This is not ideal as ignoring a return value might signal a conscious decision by the developer. Detecting a stale state bug at compile time is not feasible as it would imply that the compiler can find a connection between an ignored return value and a variable of the same type in the current scope. The challenge for the compiler is demonstrated below:

```
1 var a = new_evaluator(1)
2 var e = new_evaluator(1)
3 _, let result = evaluate_expression(e, precedence.LOWEST)
4 // Did the developer mean to reassign 'a' or 'e'?
```

Go and Rust do not have this issue when invoking methods as the current state is tied to the function call.

Dash's immutability offers better reasoning about state changes as functions can not modify state but comes with a performance overhead due to copying. Dash attempts to mitigate this using a copy-on-write approach and bundling modifications within the update block. In addition to that, the syntax makes state transitions explicit which aligns with DOP patterns, at the cost of verbosity. Another potential drawback is that Dash does not allow allocating more space than required, where the allocated buffer is only partially used. Although, the overhead of modifying data can be mitigated using performant data structures as mentioned in section 2.1.1. Either way, further evaluation is required to determine the performance overhead caused by this.

Go's mutable approach is quicker since it does not involve copying, but it provides less safety guarantees for state changes. This is because it uses OOP principles, which makes it harder to reason about state changes and can be error-prone in complex settings [Sha22], especially when it comes to testing [KZA10].

Rust's ownership and borrowing mechanism facilitates confined mutability while preventing unsafe state changes. This allows for performance to be maintained by minimising unnecessary copying while guaranteeing exclusive mutable access. This also avoids data race conditions. On the other hand, the mutability annotations and ownership rules can make code verbose and make designing algorithms and data structures more complex [LAC<sup>+</sup>15, AMP<sup>+</sup>20].

In conclusion, Dash's state management approach prioritises safety, explicitness and makes code easier to reason about. Its immutable design with copy-on-write semantics and structural sharing minimises the performance impact, even though modifications are verbose compared to Go's and Rust's direct mutation.

### 5.3.2 Type System

Dash uses a conservative approach to type inference that closely resembles Go's philosophy. Dash enables type inference for local variables using the `let` and `var` keywords:

```
1 let x = 1 // i64
2 var arr = [1, 2, 3] // []i64
```

Compared to Rust's more sophisticated type inference system which can work across function boundaries, Dash's approach is more limited but provides clarity and predictability. Go similarly requires explicit type annotations for function parameters and returns. This is demonstrated below in a hypothetical example:

```
1 let elem = 5u8;
2 let mut vec = Vec::new();
3 vec.push(elem);
4 // compiler now knows vec is of type Vec<u8>
```

While in Dash an explicit generic type annotation is required when instantiating the growable list of type `u8`:

```
1 let e = u8(5);
2 var l = list.new<u8>(); // generic type annotation required
3 l = list.push(l, e); // no annotation required here as its inferred from 'l'
```

In this regard Dash's and Golang's philosophies align where both prefer being explicit rather than have implicit behaviour.

Type conversion safety is where Dash improves upon Go. As mentioned in section 3.4.3, Dash implements a strict type conversion system that categorises conversions into three categories: legal, error-prone and illegal. Error-prone conversions are required to be handled by the developer. This differs significantly from Go's approach, which does not enforce checking if type cast succeeded and unchecked type casts can panic:

```
1 val, ok := old.(*SomeType) // ok can be ignored
2 val := old.(*SomeType) // can panic if types don't match
```

Moreover, if the `ok` is not validated properly it could cause the program to panic when `val` is used as it would be `nil`.

In addition to type safety, Dash's type system offers clearer syntax through specialized keywords. Dash provides distinct syntactical keywords for each construct similar to Rust. While Go employs the `type` keyword generically, Dash distinguishes between type definitions, type aliases, unions, structs and generic structs. This is shown in the calculator implementation, where each type is defined with its own keyword:

```
1 struct evaluator {
2     // ...
3 }
4
5 type infix_fn fn(evaluator, i64) evaluator, i64
6
7 enum precedence {
8     // ...
9 }
```

This creates a clear distinction between the semantics of each statement.

Overall, Dash's type system combines the simplicity of Go with some of Rust's safety and inference features, while also bringing new aspects such as generic structs to facilitate implementing DOP principles. This restrictive and explicit approach allows Dash to enforce safety while keeping the syntax simple.

### 5.3.3 Safety Guarantees

#### Null Safety

While Golang permits null pointers, Dash and Rust both have type systems that strictly ensure null safety. To express absent values, Dash utilises optional types indicated with `?` followed by the type, whereas Rust uses `Option<T>`. In the current implementation Dash's compiler is not intelligent enough to detect null checks and automatically infer the underlying type when used. For example in `evaluate_expression` the `force unwrap` is still required:

```
1 // evaluate_expression
2 let infix = get_infix_fn(e.cur_tkn)
3 if infix == null {
4     return e, res
5 }
6 let func = ?infix
7 e, res = func(e, res)
```

In Dash, null checks for optionals are explicitly required, while in Golang if the null check is omitted a runtime panic would occur.

#### Memory Safety

Golang and Rust provide memory safety guarantees, but through different mechanisms. Rust uses its ownership system with borrowing rules to manage memory automatically, while Golang uses garbage collection. Both languages provide bound checking and avoid risky operations such as pointer arithmetic. It should be noted that both languages allow for ways to write unsafe code usually evading the safety mechanisms imposed by the language.

Dash should offer similar guarantees but there is not much that can be said as it is currently not implement. Moreover, this would also require an empirical evaluation to compare Dash's safety and performance to that of similar programming languages, especially with regard to memory usage patterns and de-/allocation overhead.

#### Type Safety

All three languages are statically typed and will not compile if types do not match, but Dash offers more robust type safety guarantees thanks to the compile-time validation through built-in type predicates. While type predicates are also not found in the implementation, their application should be highlighted:

```
1 type age i64 | 18 <= age && age <= 65
```

The benefit of it being part of the language is that Dash can enforce type validation rules at compile-time and ensure only validated data is propagated through the system. This is something that would have to be implemented through a third party package/crate in the other two languages and carefully used. This makes Dash safe beyond simply the type but also ensures the underlying data is validated and propagated safely throughout the system.

#### Numeric Safety

Dash's arbitrary precision `int` and `float` types offer higher numeric safety, whereas Rust and Go require explicit management of overflow scenarios:

```

1 // Rust - handle overflow explicitly
2 let x = i64::MAX.checked_add(1); // Returns None
3
4 // Go - silent overflow
5 x := math.MaxInt64
6 x += 1

```

Dash's approach avoids common numerical errors that can occur in Golang integer types while remaining less verbose than Rust. It should be noted Dash has fixed bit-width integers that would express the same safety drawbacks as Golang. This is because there are legitimate cases where such types are required, for example when interacting with network protocols. This could potentially be avoided by introducing special overflow and underflow aware arithmetic operators that would raise a runtime error at the cost of some performance.

## Resource Management

Dash and Golang employ defer statements for resource management, whereas Rust's resource acquisition is initialization paradigm [Mai24] offers compile-time assurances for structs defining the `Drop` trait. Dash does not enforce clean-up at the moment. For instance, when a file is opened:

```

1 fn read_file() {
2     let fd = os.file_open("file.txt")
3     defer os.file_close(fd) // no warning if omitted
4 }

```

This is an area where Dash can draw inspiration from Rust to prevent resource management faults that arise from missing clean-up, perhaps using its annotation system for enforcement.

### 5.3.4 Error Handling

Defining errors in Dash is done using an algebraic type, using the `error` keyword:

```

1 error division_by_zero
2 error out_of_bounds(index i64, size i64)

```

Go takes a simpler approach, primarily using string-based errors through the error interface:

```
1 errors.New("division by zero")
```

Rust uses algebraic types through enums to define error types:

```

1 enum MathError {
2     DivisionByZero,
3     OutOfBounds { index: i64, size: i64 }
4 }

```

Golang's approach is by far the simplest but can lead to additional imports and code if more error information is required:

```
1 fmt.Errorf("index %d out of bounds for length %d", index, size)
```

Dash's approach also makes it possible to have very performant error matching as they function similarly to unions using a type identifier. In Golang this is less performant in many cases as string matching is required.

Regarding raising errors, Dash provides a `raise` keyword and error-prone functions must be annotated with an exclamation mark:

```
1 fn divide(a, b i64)! i64 {
2     if b == 0 {
3         raise division_by_zero
4     }
5     return a / b
6 }
```

This keeps the code clear and removes the need to manually return each value as is required in Golang's more verbose approach:

```
1 func divide(a, b int64) (int64, error) {
2     if b == 0 {
3         return 0, errors.New("division by zero")
4     }
5     return a / b, nil
6 }
```

Dash has two error handling mechanisms; `try` for direct propagation and `catch` for explicit handling before propagating, retrying or returning a default value. This is very similar to how Rust handles errors using pattern matching or `?` operator to propagate the error. On the other hand, while Golang requires explicit error checking using an if statement it does not guarantee the error will be dealt with, potentially leaving the system in a undefined state:

```
1 res, err := divide(1, 0)
2 if err != nil {
3     // developer forgot to return error
4 }
5 // undefined state
```

In conclusion, the language maintains type safety similar to Rust's `Result` type while achieving explicit error handling without the verbosity and limitations of Go's if/else checks and string error type. Although, there will be more overall error handling due to Dash's error-prone operations crowding the code base with functions marked as error-prone and `try` / `catch` keywords.

## 5.3.5 Abstraction Mechanisms

### Code Abstraction

When it comes to polymorphism Dash has support for two variants; parametric and ad-hoc polymorphism. Golang and Rust both only support the latter which is often referred to as generics. Dash also has support for a more specific version of generics through its generic structs, as explained in section 3.3.6.

On the other hand, Golang supports static structural typing [Pie02b] through its interfaces allowing for simple dependency inversion allowing for great modularity and testability:

```
1 type Reader interface {
2     Read(p []byte) (n int, err error)
3 }
4 func processInput(r Reader) error {
5     // ..
6 }
```

**Listing 5.3:** Function accepting any struct which implements interface in Golang

In listing 5.3 above any struct that implements the method called `Read` is accepted by `processInput`. While Dash only allows data to be passed and returned from functions, a different way needs to be found to allow for testability. This could potentially be solved through a testing framework that can replace functions that need to be mocked before the test starts.

## Code Organisation

The implementations reveal different methods for code organisation. Golang and Rust's struct-based encapsulation serves as an example of object-oriented principles:

```

1 struct Lexer {
2     // ...
3 }
4 impl Lexer {
5     fn next_token(&mut self) -> Token {
6         // ...
7     }
8 }

```

Dash, on the other hand, uses a flatter structure in which functions work with structs but are not contained within them:

```

1 struct lexer {
2     // ...
3 }
4
5 fn next_token(l lexer) lexer, token {
6     // ...
7 }

```

The developer experience and code organisation are impacted by this architectural difference. Because Dash keeps all functions at the same scope level, it might affect developer productivity. For instance, autocompletion tools in IDEs would show the developer all functions or types within a library, while in Golang or Rust it would be scoped within the current object.

In the context of project wide code organisation, all three languages feature a modular approach, with certain key differences. In Go's package system, each directory represents a single package, which places an emphasis on clarity and explicit dependencies. In Rust, cargo supports declarative dependencies and the crate system offers fine-grained control over module hierarchies and visibility. Dash's library system is most similar to Golang, providing layered libraries and employing filesystem-based library organisation. This method allows for logical code grouping through directory structures while maintaining simplicity.

## 5.3.6 Code Clarity and Verbosity

### Syntactic Abstractions

Go's support for grouped case statements allows for more concise switch expressions:

```

1 func (e *Evaluator) getInfixFn(tkn Token) infixFn {
2     switch tkn.Type {
3     case PLUS, MINUS, ASTERISK, SLASH:
4         return e.evaluateInfix

```

```
5     }
6     return nil
7 }
```

Dash currently requires separate cases for each token type, leading to more verbose match statements. This is a syntactic abstraction that Dash can add to its language to remove duplicate cases.

## Mutability vs Immutability

Another example of Dash's verbosity is found when reading characters in the lexer:

```
1 fn read_char(old lexer) lexer {
2     if old.pos > len(old.input) {
3         let l = old^{
4             l.ch = 0
5         }
6         return l
7     }
8
9     let l = old^{
10        l.ch = l.input[old.next_pos]
11        l.pos = old.next_pos
12        l.next_pos = old.next_pos + 1
13    }
14    return l
15 }
```

As all data is immutable Dash has to perform a copy when updating the lexer state. This makes `read_char` a lot more verbose when compared to Golang's and Rust's approach:

```
1 func (l *Lexer) readChar() {
2     if l.nextPos >= len(l.input) {
3         l.ch = 0
4     } else {
5         l.ch = l.input[l.nextPos]
6     }
7     l.pos = l.nextPos
8     l.nextPos++
9 }
```

This is a fundamental difference between Dash's immutability and the other languages mutability which can not be improved upon; Dash will always be more verbose in such cases.

Moreover, Dash also has a lot of examples where a function argument is redefined in the beginning of the block using `var` to make it reassignable. This adds code which would not be required in the other languages.

## Lines of Code

In terms of lines of code:

- Rust: 270 lines
- Go: 300 lines

- Dash: 350 lines

However, Dash's implementation includes boilerplate code that would be provided by the language in the future, such as bound checking not being required in `str_to_i64`, error handling mechanisms and IO library. When accounting for these differences, Dash's core implementation size is comparable to Go's, while Rust remains concise.

It should be noted that the implementations in Rust and Go were created to mirror Dash's general structure (e.g. type, struct and function names, class structure), potentially limiting certain abstractions that would be more natural in those languages. Furthermore, the implementation in Dash could have been made more concise, but it was deliberately expanded to showcase features.

### 5.3.7 Interoperability

Interoperability with C is supported by all three languages, with notable differences between their implementation while keeping syntax similar. Dash uses a minimalist approach using the `@extern(c)` decorator for both importing and exporting functions. This is demonstrated in its straightforward foreign function interface (FFI) syntax:

```
1 @extern(c)
2 pub fn read(fd i64, buf *memory<[]byte>, count i64) i64
```

Rust uses the `extern "C"` block syntax with special C types for importing external functions:

```
1 extern "C" {
2     fn compute_sum(numbers: *const c_int, length: c_size_t) -> c_int;
3 }
```

Go uses a different strategy through `cgo` [Tea24a], requiring special comment directives:

```
1 /*
2  #include <stdio.h>
3  #include <stdlib.h>
4  */
5
6  import "C"
7
8  // call C functions
```

Dash also allows functions to be exported using the same syntax but with a function body:

```
1 @extern(c)
2 pub fn add(a, b i64) i64 {
3     // ...
4 }
```

While Golang uses the special `//export` comment to signify a function as exportable:

```
1 import "C"
2
3 //export read
4 func read(fd C.int, buf *C.char, count C.size_t) C ssize_t {
5     // ...
6 }
```

---

When Dash adds additional protection mechanisms when passing aggregate data structures such as copying to maintain immutability, it could add a significant performance hit if the structure is large or the function is called often enough. Measuring performance overhead between the three FFI implementations is beyond the scope of this thesis and is an area requiring further analysis.

Overall, all three languages offer similar functionality when working with external C libraries with the main difference being that Dash uses built-in types in the function signature and automatically match them to the relevant C types.



# Conclusion

This thesis successfully explored common software faults and how a language implementing the DOP paradigm and addressing the software faults would look like, through the design and implementation of Dash. The findings show that while many software faults can be prevented, logic errors and developer oversights remain difficult to protect against. Moreover, the results show that the fundamental principles of DOP can be successfully used as language constructs, although some modifications from the original paradigm are required at the language-level.

Dash's language design prevents several CWEs, such as buffer overflows, null pointer dereferences, integer overflow problems and type confusion vulnerabilities. Dash prevents error handling oversights by making potentially problematic operations more likely to be caught by defining error-prone operations and explicit error handling. Compile-time analysis and runtime guards work together to provide a solid defence against type vulnerabilities and memory corruption.

Dash enhances and implements DOP concepts with robust type safety. Moreover, Dash uses generic structs with compile-time checks for generic data representation rather than hash maps, as prescribed in DOP. This improves efficiency and safety guarantees, but also adds more verbosity through explicit struct definitions. Although more research is needed to determine the effect on performance, the language's immutable-by-default strategy combined with explicit copy-and-update procedures guarantees predictable data transformations. The type system demonstrates how DOP principles can be enhanced with static typing by fusing simple syntax with more advanced features like type predicates and polymorphism.

The current implementation shows off fundamental language features like pattern matching, generic structs and a strong type system as a proof-of-concept. Further study and development are needed in a number of important areas such as memory management, error handling and concurrency. A memory management model must be implemented and validated to automatically and safely manage allocation and deallocation. Moreover, it is necessary to look into concurrency models that integrate with the DOP principles and work with the memory management strategy. Furthermore, an understanding of the overhead of Dash's immutability is required through performance benchmarking with other languages.

In addition to the core language, a set of developer tools needs to be developed including support for integrated development environments, a build system, a testing framework and a debugger. It is also necessary to have a thorough standard library with documented, tested functions for common operations, as these are crucial for language adoption.

Due to the limitations of the Dash compiler, the evaluation was limited to a simple arithmetic expression evaluator. In order to evaluate Dash's abilities in more detail in realistic situations more real-world examples are needed. Moreover, Dash's performance characteristics are not known due to the lack of benchmarking tools for the language.

A great language is not just made up of its design, the ecosystem around it matters just as much. Thus, Dash still has a long way to go.

# Appendix

## 7.0.1 Dash Language Specification

Due to the size of the language specification it can be found on Zenodo [Nor24].

## 7.0.2 Dash Tree-Sitter Grammar

```
1  /// <reference types="tree-sitter-cli/dsl" />
2  // @ts-check
3  const newline = /\n/;
4  const terminator = choice(newline, '\0');
5
6  const PREC = {
7    primary: 9,
8    paren: 8,
9    postfix: 7,
10   prefix: 6,
11   null_coalesce: 5,
12   multiplicative: 4,
13   additive: 3,
14   comparative: 2,
15   and: 1,
16   or: 0,
17   composite_literal: -1
18 };
19 const decimal_digit = /[0-9]/;
20 const decimal_digits = seq(decimal_digit, repeat(seq(optional('_'), decimal_digit)));
21
22 const int_literal = choice('0', seq(/[1-9]/, optional(seq(optional('_'), decimal_digits)));
23 const float_literal = choice(
24   seq(decimal_digits, '.', optional(decimal_digits)),
25 );
26
27 module.exports = grammar({
28   name: 'dash',
```

```
29
30 extras: $ => [
31     /\s/,
32     $.comment
33 ],
34
35 word: $ => $.identifier,
36
37 rules: {
38     source_file: $ => seq(
39         // NOTE: consider allowing partial statements and expressions
40         // to allow parsing of code snippets in documentation
41         $.library_statement,
42         repeat($.top_level_statement)
43     ),
44
45     library_statement: $ => seq(
46         'lib',
47         $.identifier
48     ),
49
50     _top_level_statement: $ => choice(
51         $.function_statement,
52         $.struct_statement,
53         $.enum_statement,
54         $.type_statement,
55         $.type_alias_statement,
56         $.union_statement,
57         $.assignment_statement
58     ),
59
60     // ----- //
61     // Types //
62     // ----- //
63
64     _type: $ => choice(
65         $.primitive_type,
66         $.array_type,
67         $.pointer_type,
68         $.optional_type,
69         $.function_type,
70         $.memory_type,
71         $_.type_identifier
72     ),
73
74     primitive_type: $ => choice(
75         'i8', 'i16', 'i32', 'i64', 'i128',
76         'u8', 'u16', 'u32', 'u64', 'u128',
```

```
77     'f32', 'f64',
78     'bool', 'string',
79     'byte', 'char'
80 ),
81
82 array_type: $ => choice(
83     seq('[', ']', $_.type),
84     seq(
85         '[',
86         field("length", $_.expression),
87         ']',
88         $_.type
89     )
90 ),
91
92 pointer_type: $ => prec(PREC.prefix,
93     seq('*', $_.type)
94 ),
95
96 optional_type: $ => prec(PREC.prefix,
97     seq('?', $_.type)
98 ),
99
100 function_type: $ => prec.right(1, seq(
101     'fn',
102     field("parameters", $.type_parameter_list),
103     field("result", optional($_.function_return_type))
104 )),
105
106 memory_type: $ => seq(
107     'memory',
108     '<',
109     field("type", $_.type),
110     '>'
111 ),
112
113 _function_return_type: $ => choice(
114     $_.type,
115     $.multi_return_type
116 ),
117
118 multi_return_type: $ =>
119     prec.right(1, seq(
120         $_.type,
121         repeat1(seq(',', $_.type))
122     )),
123
124 // ----- //
```

```
125 // Expressions //
126 // ----- //
127
128 _expression: $ => choice(
129     $.identifier,
130     $.int_literal,
131     $.float_literal,
132     $.string_literal,
133     $.char_literal,
134     $.true,
135     $.false,
136     $.null,
137     $.call_expression,
138     $.group_expression,
139     $.prefix_expression,
140     $.infix_expression,
141     $.postfix_expression,
142     $.index_expression,
143     $.slice_expression,
144     $.dot_expression,
145     $.use_expression,
146     $.copy_expression,
147     $.array_literal,
148     $.struct_literal,
149     $.function_literal,
150 ),
151
152 copy_expression: $ => prec.right(1, choice(
153     seq(
154         field("name", $.identifier),
155         '^',
156     ),
157     seq(
158         field("name", $.identifier),
159         '^',
160         field("body", $.block_statement),
161     ),
162 ));
163
164 group_expression: $ => prec(PREC.paren, seq(
165     '(',
166     $_expression,
167     ')',
168 ));
169
170 prefix_expression: $ => prec(PREC.prefix, choice(
171     seq('-', $_expression),
172     seq('!', $_expression),
```

```
173     seq('&', $_expression),
174     seq('* ', $_expression),
175   )),
176
177   postfix_expression: $ => prec(PREC.postfix, choice(
178     seq($_expression, '++'),
179     seq($_expression, '--'),
180   )),
181
182   infix_expression: $ => choice(
183     ...[
184       ['*', PREC.multiplicative],
185       ['/', PREC.multiplicative],
186       ['%', PREC.multiplicative],
187       ['+', PREC.additive],
188       ['-', PREC.additive],
189       ['<', PREC.comparative],
190       ['<=', PREC.comparative],
191       ['>', PREC.comparative],
192       ['>=', PREC.comparative],
193       ['==', PREC.comparative],
194       ['!=', PREC.comparative],
195       ['&&', PREC.and],
196       ['||', PREC.or],
197       ['??', PREC.null_coalesce],
198     ].map(([operator, precedence]) =>
199       prec.left(precedence, seq(
200         field("left", $_expression),
201         field("operator", operator),
202         field("right", $_expression),
203       ))
204   )
205 ),
206
207   if_expression: $ => prec.right(seq(
208     'if',
209     field("condition", $_expression),
210     field("body", $_if_expression_block),
211     optional(seq(
212       'else',
213       field("alternative", choice($_if_expression, $_if_expression_block))
214     ))
215   )),
216   _if_expression_block: $ => prec.right(seq(
217     '{',
218     optional(repeat($_statement)),
219     $_expression,
220     '}',
```

```
221    )),
222
223
224 match_expression: $ => prec.right(seq(
225     'match',
226     field("scrutinee", $_expression),
227     '{',
228     repeat($_match_expression_case),
229     optional($_match_expression_default),
230     '}')
231 ),
232 match_expression_case: $ => seq(
233     'case',
234     field("condition", $_expression),
235     ':',
236     field("body", $_match_expression_block),
237 ),
238 match_expression_default: $ => seq(
239     'case',
240     '_',
241     ':',
242     field("body", $_match_expression_block),
243 ),
244 _match_expression_block: $ => seq(
245     optional(repeat($_statement)),
246     $_expression,
247 ),
248
249 assignable_expression_list: $ => prec.right(seq(
250     $_assignable_expression,
251     optional(seq(',', $_assignable_expression))
252 )),
253
254 _assignable_expression: $ => prec.right(choice(
255     $_expression,
256     $_if_expression,
257     $_match_expression,
258 )),
259
260 call_expression: $ => prec(PREC.primary, choice(
261     seq(
262         field("name", alias('make', $_identifier)),
263         field("arguments", $_special_argument_list)
264     ),
265     seq(
266         field("name", $_identifier),
267         '(',
268         optional(
```

```
269     field("arguments", seq(
270       sepBy(',', $_expression),
271       optional(',', ' ')
272     )),
273   ')')
274 )),
275 special_argument_list: $ => seq(
276   '(',
277   optional(seq(
278     $_type,
279     repeat(seq(',', ' '), $_expression)),
280   optional(',', ' ')
281 )),
282 ')')
283 ),
284
285 index_expression: $ => prec(PREC.primary, seq(
286   field('operand', $_expression),
287   '[' ,
288   field('index', $_expression),
289   ']')
290 ),
291
292 slice_expression: $ => prec(PREC.primary, seq(
293   field('operand', $_expression),
294   '[' ,
295   choice(
296     seq(
297       optional(field('start', $_expression)),
298       ':',
299       field('end', $_expression),
300     ),
301     seq(
302       field('start', $_expression),
303       ':',
304       optional(field('end', $_expression)),
305     )
306   ),
307   ']')
308 ),
309
310 dot_expression: $ => prec(PREC.primary, seq(
311   field('operand', $_expression),
312   '.',
313   field('field', choice($_.identifier, /\d+/))
314 ),
315
316 use_expression: $ => seq(
```

```

317     "use",
318     field("name", $.identifier),
319     field("body", $.block_statement)
320 ),
321
322 // ----- //
323 // Literals //
324 // ----- //
325 int_literal: $ => token(int_literal),
326 float_literal: $ => token(float_literal),
327 string_literal: $ => /"[^"]*"/,
328 char_literal: $ => token(seq(
329     "'",
330     choice(
331         /[^'\\]/,
332         /\\[\\]/
333     ),
334     "'")
335 ),
336
337 function_literal: $ => prec.right(1, seq(
338     'fn',
339     field('parameters', $.parameter_list),
340     field('result', optional($_.function_return_type)),
341     field('body', optional($.block_statement)),
342 ),
343
344 struct_literal: $ => prec.right(PREC.composite_literal, choice(
345     $_.struct_literal,
346     $_.struct_anonymous_literal,
347 ),
348
349 _struct_literal: $ => seq(
350     field("name", $_.type_identifier),
351     token.immediate('{'),
352     choice(
353         seq(sepBy(',', $.struct_named_field), optional(',')),
354         seq(sepBy(',', $.struct_unnamed_field), optional(','))
355     ),
356     '}'
357 ),
358 // NOTE: the precedence here is definitely too high
359 // as tests pass i will leave as is. This will for
360 // sure be a source of bugs in future.
361 _struct_anonymous_literal: $ => prec(PREC.primary, seq(
362     '{',
363     choice(
364         seq(sepBy(',', $.struct_named_field), optional(',')),
365         seq(sepBy(',', $.struct_unnamed_field), optional(','))

```

```
365     ),
366     '}'
367   ),),
368   struct_named_field: $ => seq(
369     seq(
370       field("key", $.identifier),
371       ":",
372       field("value", $_.expression),
373     ),
374   ),
375   struct_unnamed_field: $ => seq(
376     field("value", $_.expression),
377   ),
378
379   array_literal: $ => prec(PREC.composite_literal, seq(
380     '[' ,
381     optional(seq(
382       sepBy(' ', $_.expression),
383       optional(' ')
384     )),
385     ']'
386   )),
387
388   // ----- //
389   // Statements //
390   // ----- //
391
392   _statement: $ => choice(
393     $.expression_statement,
394     $.assignment_statement,
395     $.return_statement,
396     $.if_statement,
397     $.for_statement,
398     $.match_statement,
399     $.block_statement,
400     $.defer_statement,
401     $.next_keyword,
402     $.break_keyword,
403   ),
404
405   expression_statement: $ => prec.right(-1, $_.expression),
406
407   assignment_statement: $ => seq(
408     field("left", $.declaration_list),
409     '=',
410     field("right", $.assignable_expression_list),
411   ),
412
```

```
413 declaration_list: $ => prec.right(1, seq(
414     choice(
415         $.let_statement,
416         $.var_statement,
417         $.identifier,
418         $.dot_expression,
419         $.index_expression,
420         $.slice_expression,
421     ),
422     optional(seq(',', choice(
423         $.let_statement,
424         $.var_statement,
425         $.identifier,
426         $.dot_expression,
427         $.index_expression,
428         $.slice_expression,
429     ))),
430 ));
431
432 let_statement: $ => seq(
433     'let',
434     $.identifier
435 ),
436
437 var_statement: $ => seq(
438     'var',
439     $.identifier
440 ),
441
442 return_statement: $ => prec.right(1, seq(
443     'return',
444     optional(seq(
445         $_expression,
446         optional(seq(',', $_expression))
447     ))
448 ));
449
450 if_statement: $ => prec.right(seq(
451     'if',
452     field("condition", $_expression),
453     field("body", $.block_statement),
454     optional(seq(
455         'else',
456         field("alternative", choice($.if_statement, $.block_statement))
457     ))
458 ));
459
460 for_statement: $ => seq(
```

```
461     'for',
462     optional(choice(
463         $.for_classic,
464         $_.expression
465     )),
466     field("body", $.block_statement)
467 ),
468 for_classic: $ => seq(
469     field("initialiser", $.assignment_statement), ';',
470     field("condition", $_.expression), ';',
471     field("update", choice($_.expression, $.assignment_statement)),
472 ),
473
474 match_statement: $ => seq(
475     'match',
476     field("scrutinee", $_.expression),
477     '{',
478     repeat($.match_statement_case),
479     optional($.match_statement_default),
480     '}'
481 ),
482
483 match_statement_case: $ => seq(
484     'case',
485     field("condition", $_.expression),
486     ':',
487     optional(field("body", repeat($_.statement)))
488 ),
489
490 match_statement_default: $ => seq(
491     'case',
492     '_,',
493     ':',
494     optional(field("body", repeat($_.statement)))
495 ),
496
497 block_statement: $ => seq(
498     '{',
499     repeat($_.statement),
500     '}'
501 ),
502
503 defer_statement: $ => seq(
504     'defer',
505     choice($.call_expression, $.block_statement)
506 ),
507 attribute_statement: $ => choice(
508     $.function_attribute,
```

```
509     ),
510
511     function_statement: $ => prec.right(1, seq(
512         repeat($.attribute_statement),
513         optional('pub'),
514         'fn',
515         field('name', $.identifier),
516         field('parameters', $.parameter_list),
517         field('result', optional($_.function_return_type)),
518         field('body', optional($.block_statement)),
519     )),
520
521     function_attribute: $ => choice(
522         seq('@extern', '(', 'c', ')'),
523         seq('@inline', '(', choice('never', 'hint', 'always'), ')')
524     ),
525
526     struct_statement: $ => seq(
527         optional('pub'),
528         optional('gen'),
529         'struct',
530         field("name", $.identifier),
531         field("body", $.struct_body),
532     ),
533
534     struct_body: $ => seq(
535         '{',
536         optional(choice(
537             seq(
538                 $.field_statement,
539                 repeat(seq(',', $.field_statement)),
540             ),
541             seq(
542                 newline,
543                 repeat(seq(
544                     $.field_statement,
545                     newline,
546                 )),
547             )),
548         )),
549         '}'
550     ),
551
552     field_statement: $ => prec.right(1, seq(
553         field("name", optional($.identifier)),
554         field("type", $_.type),
555         optional(seq('|', $_.expression))
556     )),
```

```
557
558 enum_statement: $ => seq(
559     optional('pub'),
560     'enum',
561     field("name", $.identifier),
562     field("body", $.enum_body)
563 ),
564 enum_body: $ => seq(
565     '{',
566     newline,
567     repeat(seq(
568         $.field_statement,
569         newline
570     )),
571     '}'
572 ),
573
574 type_statement: $ => seq(
575     'type',
576     field("name", $.identifier),
577     field("type", $_.type),
578     optional(seq('|', $_.expression))
579 ),
580
581 type_alias_statement: $ => seq(
582     'alias',
583     field("name", $.identifier),
584     field("type", $_.type)
585 ),
586
587 union_statement: $ => seq(
588     optional('pub'),
589     'union',
590     field("name", $.identifier),
591     field("body", $.union_body)
592 ),
593 union_body: $ => seq(
594     '{',
595     newline,
596     repeat(seq(
597         $_.type,
598         newline
599     )),
600     '}'
601 ),
602
603 // ----- //
604 // Helpers //
```

```

605 // ----- //
606
607 type_parameter_list: $ => seq(
608   '(',
609   optional(seq(
610     sepBy(',', $_type),
611     optional(',')
612   )),
613   ')',
614 ),
615
616 parameter_list: $ => seq(
617   '(',
618   optional(seq(
619     sepBy(',', $.parameter),
620     optional(',')
621   )),
622   ')',
623 ),
624
625 parameter: $ => seq(
626   field("name", $.identifier),
627   optional(seq(',', $.identifier)),
628   field("type", $_type)
629 ),
630
631 // keywords
632 break_keyword: $ => 'break',
633 next_keyword: $ => 'next',
634 goto_keyword: $ => 'goto',
635
636 // Terminals
637 identifier: $ => /[a-zA-Z_][a-zA-Z0-9_]**/,
638 _type_identifier: $ => alias($.identifier, $.type_identifier),
639 true: $ => 'true',
640 false: $ => 'false',
641 null: $ => 'null',
642 comment: $ => token(choice(
643   seq('//', /.*/),
644   seq('/*/', /^[^*]*\*+([/*]*\*+)*\/, '/')
645 ))
646 }
647 });
648
649 /**
650  * Creates a rule to optionally match one or more instances of rule separated by sep
651  */
652 function sepBy(sep, rule) {

```

```
653     return optional(seq(rule, repeat(seq(sep, rule))));
654 }
```

**Listing 7.1:** Dash tree-sitter grammar

### 7.0.3 Evaluation in Dash

```
1  lib main
2
3  fn main() {
4      for {
5          print(">> ")
6          let buf = make([]byte, 128)
7          let n = read(std_in, &buf, len(buf))
8          if n <= 0 {
9              next
10         }
11
12         let src = string(buf[0:n])
13         let l = new_lexer(src)
14         var e = new_evaluator(l)
15
16         e, let result = evaluate_expression(e, precedence.LOWEST)
17         print_res(result)
18         print("\n")
19     }
20 }
21
22 // ----- //
23 // Token //
24 // ----- //
25
26 struct token {
27     typ token_type
28     literal string
29 }
30
31 enum token_type {
32     PLUS
33     MINUS
34     ASTERISK
35     SLASH
36     LPAREN
37     RPAREN
38     INT
39     EOF
40 }
41
```

```
42 // ----- //
43 // Lexer //
44 // ----- //
45
46 struct lexer {
47     pos i64
48     next_pos i64
49     ch byte
50     input string
51 }
52
53 fn new_lexer(input string) lexer {
54     return lexer{
55         pos: 0,
56         next_pos: 1,
57         ch: input[0],
58         input: input,
59     }
60 }
61
62 fn next_token(l lexer) lexer, token {
63     var tkn = token{typ: token_type.EOF, literal: ""}
64
65     if l.ch == 0 {
66         return l, tkn
67     }
68
69     var l = skip_whitespace(l)
70
71     match l.ch {
72     case '+':
73         tkn = token{typ: token_type.PLUS, literal: string(l.ch)}
74     case '-':
75         tkn = token{typ: token_type.MINUS, literal: string(l.ch)}
76     case '/':
77         tkn = token{typ: token_type.SLASH, literal: string(l.ch)}
78     case '*':
79         tkn = token{typ: token_type.ASTERISK, literal: string(l.ch)}
80     case '(':
81         tkn = token{typ: token_type.LPAREN, literal: string(l.ch)}
82     case ')':
83         tkn = token{typ: token_type.RPAREN, literal: string(l.ch)}
84     case _:
85         if is_digit(l.ch) {
86             let l, let tkn = read_int(l)
87             return l, tkn
88         }
89         return l, tkn // return EOF
```

```
90     }
91     l = read_char(l)
92     return l, tkn
93 }
94
95 fn read_char(old lexer) lexer {
96     if old.pos > len(old.input) {
97         let l = old^{
98             l.ch = 0
99         }
100        return l
101    }
102
103    let l = old^{
104        l.ch = l.input[old.next_pos]
105        l.pos = old.next_pos
106        l.next_pos = old.next_pos + 1
107    }
108    return l
109 }
110
111 fn is_digit(ch byte) bool {
112     return '0' <= ch && ch <= '9'
113 }
114
115 fn read_int(l lexer) lexer, token {
116     var l = l
117     let prev = l.pos
118     for is_digit(l.ch) {
119         l = read_char(l)
120     }
121     return l, token{typ: token_type.INT, literal: l.input[prev:l.pos]}
122 }
123
124 fn skip_whitespace(l lexer) lexer {
125     var l = l
126     for l.ch == ' ' {
127         l = read_char(l)
128     }
129     return l
130 }
131
132 // ----- //
133 // Evaluate //
134 // ----- //
135
136 type infix_fn fn(evaluator, i64) evaluator, i64
137
```

```
138 enum precedence { // lowest to highest
139     LOWEST
140     ADDSUB
141     MULDIV
142     GROUP
143 }
144
145 struct evaluator {
146     l lexer
147     cur_tkn token
148 }
149
150 fn new_evaluator(l lexer) evaluator {
151     let l, let tkn = next_token(l)
152     return evaluator{l: l, cur_tkn: tkn}
153 }
154
155 fn next_term(e evaluator) evaluator {
156     let l, let tkn = next_token(e.l)
157     return evaluator{l: l, cur_tkn: tkn}
158 }
159
160 fn evaluate_expression(e evaluator, p precedence) evaluator, i64 {
161     var e, var res = evaluate_prefix(e)
162     let tkn = e.cur_tkn
163     for p < get_precedence(e.cur_tkn) {
164         let infix = get_infix_fn(e.cur_tkn)
165         if infix == null {
166             return e, res
167         }
168         let func = ?infix
169         e, res = func(e, res)
170     }
171
172     return e, res
173 }
174
175 fn evaluate_prefix(e evaluator) evaluator, i64 {
176     let tkn = e.cur_tkn
177
178     match tkn.typ {
179     case token_type.INT:
180         let res = str_to_i64(tkn.literal)
181         let e = next_term(e)
182         return e, res
183     case token_type.LPAREN:
184         var e = next_term(e)
185         e, let res = evaluate_expression(e, precedence.LOWEST)
```

```
186     let tkn = e.cur_tkn
187     if tkn.typ != token_type.RPAREN {
188         panic("expected closing parenthesis")
189     }
190     e = next_term(e)
191     return e, res
192 case token_type.MINUS:
193     var e = next_term(e)
194     e, let res = evaluate_prefix(e)
195     return e, -res
196 }
197 panic("unexpected token")
198 return e, 0
199 }
200
201 fn evaluate_infix(e evaluator, left i64) evaluator, i64 {
202     var e = e
203
204     let tkn = e.cur_tkn
205     match tkn.typ {
206     case token_type.PLUS:
207         e = next_term(e)
208         e, let right = evaluate_expression(e, precedence.ADDSUB)
209         return e, left + right
210     case token_type.MINUS:
211         e = next_term(e)
212         e, let right = evaluate_expression(e, precedence.ADDSUB)
213         return e, left - right
214     case token_type.ASTERISK:
215         e = next_term(e)
216         e, let right = evaluate_expression(e, precedence.MULDIV)
217         return e, left * right
218     case token_type.SLASH:
219         e = next_term(e)
220         e, let right = evaluate_expression(e, precedence.MULDIV)
221         if right == 0 {
222             panic("division by zero")
223         }
224         return e, left / right
225     }
226     panic("invalid token")
227     return e, 0
228 }
229
230 fn get_precedence(tkn token) precedence {
231     match tkn.typ {
232     case token_type.PLUS:
233         return precedence.ADDSUB
```

```
234     case token_type.MINUS:
235         return precedence.ADDSUB
236     case token_type.ASTERISK:
237         return precedence.MULDIV
238     case token_type.SLASH:
239         return precedence.MULDIV
240     case token_type.LPAREN:
241         return precedence.GROUP
242     }
243     return precedence.LOWEST
244 }
245
246 fn get_infix_fn(tkn token) ?infix_fn {
247     match tkn.typ {
248     case token_type.PLUS:
249         return evaluate_infix
250     case token_type.MINUS:
251         return evaluate_infix
252     case token_type.ASTERISK:
253         return evaluate_infix
254     case token_type.SLASH:
255         return evaluate_infix
256     }
257     return null
258 }
259
260 // ----- //
261 // Helpers //
262 // ----- //
263
264 fn str_to_i64(s string) i64 {
265     if len(s) == 0 {
266         panic("length of string zero")
267     }
268
269     var start = 0
270     let is_neg = s[0] == '-'
271     if is_neg {
272         start = 1
273     }
274
275     var result = 0
276     for i = start; i < len(s); i++ {
277         result = result * 10 + i64(s[i] - '0')
278     }
279
280     if is_neg {
281         return -result
```

```
282     }
283     return result
284 }
285
286 fn print(msg string) {
287     let out = []byte(msg)
288     write(std_out, &out, len(out))
289 }
290
291 fn print_res(i i64) {
292     let m = make([]byte, 20)
293
294     var pos = len(m)-1
295     let m = use m {
296         var i = i
297         let is_neg = i < 0
298
299         if is_neg {
300             i = -i
301         }
302         // initialise with default
303         m[pos] = '0'
304         for i > 0 {
305             m[pos] = byte(i % 10) + '0'
306             i = i / 10
307             pos--
308         }
309         pos--
310
311         if is_neg {
312             m[pos] = '-'
313         }
314     }
315
316     write(std_out, &m, len(m))
317 }
318
319 fn panic(rsn string) {
320     let buf = []byte(rsn + "\n")
321     write(std_err, &buf, len(buf))
322     exit(1)
323 }
324
325 // -- //
326 // OS //
327 // -- //
328
329 let std_in = 0
```

```

330 let std_out = 1
331 let std_err = 2
332
333 // returns bytes read
334 @extern(c)
335 pub fn read(fd i64, buf *memory<[]byte>, count i64) i64
336
337 @extern(c)
338 pub fn write(fd i64, buf *[]byte, size i64)
339
340 @extern(c)
341 pub fn exit(c i64)

```

**Listing 7.2:** Evaluation source code in Dash

## 7.0.4 Evaluation in Go

```

1 package main
2
3 import "syscall"
4
5 func main() {
6     for {
7         writeString(stdout, ">> ")
8
9         buf := make([]byte, 128)
10        n, err := syscall.Read(stdin, buf)
11        if err != nil || n <= 0 {
12            return
13        }
14
15        input := string(buf[:n])
16        l := newLexer(input)
17        e := newEvaluator(l)
18
19        result := e.evaluateExpression(LOWEST)
20        writeInt64(stdout, result)
21        writeString(stdout, "\n")
22    }
23 }
24
25 // ----- //
26 // Token //
27 // ----- //
28
29 type TokenType int
30
31 const (

```

```
32     PLUS TokenType = iota
33     MINUS
34     ASTERISK
35     SLASH
36     LPAREN
37     RPAREN
38     INT
39     EOF
40 )
41
42 type Token struct {
43     Type TokenType
44     Literal string
45 }
46
47 // ----- //
48 // Lexer //
49 // ----- //
50
51 type Lexer struct {
52     pos int
53     nextPos int
54     ch byte
55     input string
56 }
57
58 func newLexer(input string) *Lexer {
59     l := &Lexer{
60         input: input,
61         pos: 0,
62         nextPos: 1,
63     }
64     if len(input) > 0 {
65         l.ch = input[0]
66     }
67     return l
68 }
69
70 func (l *Lexer) readChar() {
71     if l.nextPos >= len(l.input) {
72         l.ch = 0
73     } else {
74         l.ch = l.input[l.nextPos]
75     }
76     l.pos = l.nextPos
77     l.nextPos++
78 }
79
```

```
80 func (l *Lexer) skipWhitespace() {
81     for l.ch == ' ' {
82         l.readChar()
83     }
84 }
85
86 func isDigit(ch byte) bool {
87     return '0' <= ch && ch <= '9'
88 }
89
90 func (l *Lexer) readInt() Token {
91     start := l.pos
92     for isDigit(l.ch) {
93         l.readChar()
94     }
95     return Token{Type: INT, Literal: l.input[start:l.pos]}
96 }
97
98 func (l *Lexer) nextToken() Token {
99     l.skipWhitespace()
100
101     if l.ch == 0 {
102         return Token{Type: EOF, Literal: ""}
103     }
104
105     var tok Token
106     switch l.ch {
107     case '+':
108         tok = Token{Type: PLUS, Literal: string(l.ch)}
109     case '-':
110         tok = Token{Type: MINUS, Literal: string(l.ch)}
111     case '*':
112         tok = Token{Type: ASTERISK, Literal: string(l.ch)}
113     case '/':
114         tok = Token{Type: SLASH, Literal: string(l.ch)}
115     case '(':
116         tok = Token{Type: LPAREN, Literal: string(l.ch)}
117     case ')':
118         tok = Token{Type: RPAREN, Literal: string(l.ch)}
119     default:
120         if isDigit(l.ch) {
121             return l.readInt()
122         }
123         tok = Token{Type: EOF, Literal: ""}
124     }
125     l.readChar()
126     return tok
127 }
```

```
128
129 // ----- //
130 // Evaluator //
131 // ----- //
132
133 type infixFn func(int64) int64
134
135 type Precedence int
136
137 const (
138     LOWEST Precedence = iota
139     ADDSUB // + -
140     MULDIV // * /
141     GROUP // ()
142 )
143
144 type Evaluator struct {
145     l *Lexer
146     curTkn Token
147 }
148
149 func newEvaluator(l *Lexer) *Evaluator {
150     e := &Evaluator{l: l}
151     e.curTkn = l.nextToken()
152     return e
153 }
154
155 func (e *Evaluator) nextToken() {
156     e.curTkn = e.l.nextToken()
157 }
158
159 func getPrecedence(t Token) Precedence {
160     switch t.Type {
161     case PLUS, MINUS:
162         return ADDSUB
163     case ASTERISK, SLASH:
164         return MULDIV
165     case LPAREN:
166         return GROUP
167     default:
168         return LOWEST
169     }
170 }
171
172 func (e *Evaluator) evaluatePrefix() int64 {
173     switch e.curTkn.Type {
174     case INT:
175         val := strToInt64(e.curTkn.Literal)
```

```
176     e.nextToken()
177     return val
178 case LPAREN:
179     e.nextToken()
180     val := e.evaluateExpression(LOWEST)
181     if e.curTkn.Type != RPAREN {
182         panic("expected closing parenthesis")
183     }
184     e.nextToken()
185     return val
186 case MINUS:
187     e.nextToken()
188     return -e.evaluatePrefix()
189 default:
190     panic("unexpected token in prefix position")
191 }
192 }
193
194 func (e *Evaluator) evaluateExpression(precedence Precedence) int64 {
195     result := e.evaluatePrefix()
196
197     for precedence < getPrecedence(e.curTkn) {
198         infix := e.getInfixFn(e.curTkn)
199         if infix == nil {
200             return result
201         }
202         result = infix(result)
203     }
204     return result
205 }
206
207 func (e *Evaluator) evaluateInfix(left int64) int64 {
208     switch e.curTkn.Type {
209     case PLUS:
210         e.nextToken()
211         right := e.evaluateExpression(ADDSUB)
212         return left + right
213     case MINUS:
214         e.nextToken()
215         right := e.evaluateExpression(ADDSUB)
216         return left - right
217     case ASTERISK:
218         e.nextToken()
219         right := e.evaluateExpression(MULDIV)
220         return left * right
221     case SLASH:
222         e.nextToken()
223         right := e.evaluateExpression(MULDIV)
```

```
224     if right == 0 {
225         panic("division by zero")
226     }
227     return left / right
228 default:
229     panic("invalid token type")
230 }
231 }
232
233 func (e *Evaluator) getInfixFn(tkn Token) infixFn {
234     switch tkn.Type {
235     case PLUS, MINUS, ASTERISK, SLASH:
236         return e.evaluateInfix
237     }
238     return nil
239 }
240
241 // ----- //
242 // Helpers //
243 // ----- //
244
245 func strToInt64(s string) int64 {
246     var result int64
247     var isNeg bool
248     start := 0
249
250     if len(s) > 0 && s[0] == '-' {
251         isNeg = true
252         start = 1
253     }
254
255     for i := start; i < len(s); i++ {
256         result = result*10 + int64(s[i]-'0')
257     }
258
259     if isNeg {
260         return -result
261     }
262     return result
263 }
264
265 func writeString(fd int, s string) {
266     syscall.Write(fd, []byte(s))
267 }
268
269 func writeInt64(fd int, n int64) {
270     if n == 0 {
271         writeString(fd, "0")
```

```
272     return
273 }
274
275 var buf [20]byte
276 var i int = 19
277 isNeg := n < 0
278 if isNeg {
279     n = -n
280 }
281
282 for n > 0 {
283     i--
284     buf[i] = byte(n%10) + '0'
285     n /= 10
286 }
287
288 if isNeg {
289     i--
290     buf[i] = '-'
291 }
292
293 syscall.Write(fd, buf[i:])
294 }
295
296 // const file descriptors
297 const (
298     stdin = 0
299     stdout = 1
300     stderr = 2
301 )
```

**Listing 7.3:** Evaluation implemented in Golang

## 7.0.5 Evaluation in Rust

```
1 use std::io::{self, Write};
2
3 fn main() {
4     let stdin = io::stdin();
5     let mut stdout = io::stdout();
6     loop {
7         print!(">> ");
8         stdout.flush().unwrap();
9
10        // Read input
11        let mut input = String::new();
12        match stdin.read_line(&mut input) {
13            Ok(n) if n == 0 => break,
```

```
14         Ok(_) => (),
15         Err(_) => break,
16     }
17
18     let lexer = Lexer::new(input.trim().as_bytes());
19     let mut evaluator = Evaluator::new(lexer);
20
21     match evaluator.evaluate_expression(Precedence::Lowest) {
22         Ok(result) => println!("{}", result),
23         Err(msg) => println!("Error: {}", msg),
24     }
25 }
26 }
27
28 // ----- //
29 // Token //
30 // ----- //
31
32 #[derive(Clone, Copy, PartialEq)]
33 enum TokenType {
34     Plus,
35     Minus,
36     Asterisk,
37     Slash,
38     LParen,
39     RParen,
40     Int,
41     Eof,
42 }
43
44 struct Token {
45     typ: TokenType,
46     literal: [u8; 32],
47     len: usize,
48 }
49
50 impl Token {
51     fn new(typ: TokenType, literal: &[u8]) -> Token {
52         let mut t = Token {
53             typ,
54             literal: [0; 32],
55             len: literal.len(),
56         };
57         t.literal[..literal.len()].copy_from_slice(literal);
58         t
59     }
60 }
61
```

```
62 // ----- //
63 // Lexer //
64 // ----- //
65
66 struct Lexer {
67     pos: usize,
68     next_pos: usize,
69     ch: u8,
70     input: [u8; 128],
71     input_len: usize,
72 }
73
74 impl Lexer {
75     fn new(input: &[u8]) -> Lexer {
76         let mut l = Lexer {
77             input: [0; 128],
78             input_len: input.len(),
79             pos: 0,
80             next_pos: 1,
81             ch: 0,
82         };
83         l.input[..input.len()].copy_from_slice(input);
84         if input.len() > 0 {
85             l.ch = input[0];
86         }
87         l
88     }
89
90     fn read_char(&mut self) {
91         if self.next_pos >= self.input_len {
92             self.ch = 0;
93         } else {
94             self.ch = self.input[self.next_pos];
95         }
96         self.pos = self.next_pos;
97         self.next_pos += 1;
98     }
99
100     fn skip_whitespace(&mut self) {
101         while self.ch == b' ' {
102             self.read_char();
103         }
104     }
105
106     fn read_int(&mut self) -> Token {
107         let start = self.pos;
108         while is_digit(self.ch) {
109             self.read_char();
110         }
111     }
112 }
```

```
110     }
111     Token::new(TokenType::Int, &self.input[start..self.pos])
112 }
113
114 fn next_token(&mut self) -> Token {
115     self.skip_whitespace();
116
117     if self.ch == 0 {
118         return Token::new(TokenType::Eof, b"");
119     }
120
121     let tok = match self.ch {
122         b'+' => Token::new(TokenType::Plus, &[self.ch]),
123         b'-' => Token::new(TokenType::Minus, &[self.ch]),
124         b'*' => Token::new(TokenType::Asterisk, &[self.ch]),
125         b'/' => Token::new(TokenType::Slash, &[self.ch]),
126         b'(' => Token::new(TokenType::LParen, &[self.ch]),
127         b')' => Token::new(TokenType::RParen, &[self.ch]),
128         _ => {
129             if is_digit(self.ch) {
130                 return self.read_int();
131             }
132             Token::new(TokenType::Eof, b"")
133         }
134     };
135     self.read_char();
136     tok
137 }
138 }
139
140 // ----- //
141 // Evaluator //
142 // ----- //
143
144 #[derive(PartialEq, PartialOrd)]
145 enum Precedence {
146     Lowest,
147     AddSub,
148     MulDiv,
149     Group,
150 }
151
152 struct Evaluator {
153     lexer: Lexer,
154     cur_token: Token,
155 }
156
157 impl Evaluator {
```

```
158 fn new(lexer: Lexer) -> Evaluator {
159     let mut e = Evaluator {
160         lexer,
161         cur_token: Token::new(TokenType::Eof, b""),
162     };
163     e.next_token();
164     e
165 }
166
167 fn next_token(&mut self) {
168     self.cur_token = self.lexer.next_token();
169 }
170
171 fn get_precedence(t: &Token) -> Precedence {
172     match t.typ {
173         TokenType::Plus | TokenType::Minus => Precedence::AddSub,
174         TokenType::Asterisk | TokenType::Slash => Precedence::MulDiv,
175         TokenType::LParen => Precedence::Group,
176         _ => Precedence::Lowest,
177     }
178 }
179
180 fn evaluate_prefix(&mut self) -> Result<i64, &'static str> {
181     match self.cur_token.typ {
182         TokenType::Int => {
183             let val = str_to_i64(&self.cur_token.literal[..self.cur_token.len])
184                 ;
185             self.next_token();
186             Ok(val)
187         }
188         TokenType::LParen => {
189             self.next_token();
190             let val = self.evaluate_expression(Precedence::Lowest)?;
191             if self.cur_token.typ != TokenType::RParen {
192                 return Err("Expected closing parenthesis");
193             }
194             self.next_token();
195             Ok(val)
196         }
197         TokenType::Minus => {
198             self.next_token();
199             let val = self.evaluate_prefix()?;
200             Ok(-val)
201         }
202         _ => Err("Unexpected token in prefix position"),
203     }
204 }
```

```
205 fn evaluate_expression(&mut self, precedence: Precedence) -> Result<i64, &'
    static str> {
206     let mut left = self.evaluate_prefix()?;
207
208     while precedence < Self::get_precedence(&self.cur_token) {
209         left = self.evaluate_infix(left)?;
210     }
211     Ok(left)
212 }
213
214 fn evaluate_infix(&mut self, left: i64) -> Result<i64, &'static str> {
215     match self.cur_token.typ {
216         TokenType::Plus => {
217             self.next_token();
218             let right = self.evaluate_expression(Precedence::AddSub)?;
219             Ok(left + right)
220         }
221         TokenType::Minus => {
222             self.next_token();
223             let right = self.evaluate_expression(Precedence::AddSub)?;
224             Ok(left - right)
225         }
226         TokenType::Asterisk => {
227             self.next_token();
228             let right = self.evaluate_expression(Precedence::MulDiv)?;
229             Ok(left * right)
230         }
231         TokenType::Slash => {
232             self.next_token();
233             let right = self.evaluate_expression(Precedence::MulDiv)?;
234             if right == 0 {
235                 return Err("Division by zero");
236             }
237             Ok(left / right)
238         }
239         _ => Err("Invalid operator"),
240     }
241 }
242 }
243
244 fn is_digit(ch: u8) -> bool {
245     ch >= b'0' && ch <= b'9'
246 }
247
248 fn str_to_i64(s: &[u8]) -> i64 {
249     let mut result = 0i64;
250     let mut is_neg = false;
251     let mut start = 0;
```

```
252
253     if s.len() > 0 && s[0] == b'-' {
254         is_neg = true;
255         start = 1;
256     }
257
258     for &digit in &s[start..] {
259         result = result * 10 + (digit - b'0') as i64;
260     }
261
262     if is_neg {
263         -result
264     } else {
265         result
266     }
267 }
```

**Listing 7.4:** Evaluation implemented in Rust

---

# Bibliography

- [AMP<sup>+</sup>20] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. How do programmers use unsafe rust? *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.
- [Ani22] Maurizio Aniche. *Effective Software Testing*. Simon and Schuster, May 2022.
- [Ass20] IEEE Standards Association. Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, 2020.
- [BC24] Max Brunsfeld and Tree-Sitter Contributors. tree-sitter, 2024.
- [BCM04a] S.M. Blackburn, P. Cheng, and K.S. McKinley. Oil and water? high performance garbage collection in java with mmtk. In *Proceedings. 26th International Conference on Software Engineering*, pages 137–146, 2004.
- [BCM04b] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Myths and realities: the performance impact of garbage collection. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '04/Performance '04*, page 25–36, New York, NY, USA, 2004. Association for Computing Machinery.
- [ben19] benbakhar. Github - benbakhar/rudash: Rudash - lodash for ruby apps, 2019.
- [Bil24] G. Bill. Odin programming language, 2024.
- [BKN07] Lukas Bulwahn, Alexander Krauss, and Tobias Nipkow. Finding lexicographic orders for termination proofs in isabelle/hol. *Lecture notes in computer science*, page 38–53, Aug 2007.
- [Boe04] Hans-J. Boehm. The space cost of lazy reference counting. *SIGPLAN Not.*, 39(1):210–219, January 2004.
- [CLA06] CLASP. Cwe-121: Stack-based buffer overflow, Jul 2006.
- [CLA24a] CLASP. Cwe-192: Integer coercion error (4.12), Oct 2024.
- [CLA24b] CLASP. Cwe-194: Unexpected sign extension (4.16), Oct 2024.
- [CLA24c] CLASP. Cwe-195: Signed to unsigned conversion error (4.4), Oct 2024.
- [CLA24d] CLASP. Cwe-196: Unsigned to signed conversion error (4.16), Oct 2024.

- [CLA24e] CLASP. Cwe-253: Incorrect check of function return value, Oct 2024.
- [CLA24f] CLASP. Cwe-390: Detection of error condition without action (4.16), Oct 2024.
- [CLA24g] CLASP. Cwe-468: Incorrect pointer scaling, Oct 2024.
- [CLA24h] CLASP. Cwe-469: Use of pointer subtraction to determine size, Oct 2024.
- [CLA24i] CLASP. Cwe-481: Assigning instead of comparing, Oct 2024.
- [CLA24j] CLASP. Cwe-482: Comparing instead of assigning, Oct 2024.
- [Col60] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, Dec 1960.
- [Com21] CWE Community. Cwe-588: Attempt to access child of a non-structure pointer, Oct 2021.
- [Com24a] Cojure Community. Clojure - atoms, Nov 2024.
- [Com24b] CWE Community. Cwe-369: Divide by zero, Oct 2024.
- [Com24c] Rust Community. Enumerations - the rust reference, 2024.
- [Com24d] Rust Community. extern - rust, 2024.
- [Com24e] Serde Community. Serde json, Nov 2024.
- [Con22] Unicode Consortium. Glossary, 2022.
- [Con24a] JQ Contributors. jq manual (development version), 2024.
- [Con24b] Rust Contributors. The borrow checker - rust compiler development guide, 2024.
- [cpp24] cppreference. free - cppreference, Mar 2024.
- [CSR15] CSRC. vulnerability - glossary | csrc, 2015.
- [CST18] Jiho Choi, Thomas Shull, and Josep Torrellas. Biased reference counting: minimizing atomic operations in garbage collection. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, New York, NY, USA, 2018. Association for Computing Machinery.
- [CT22a] Keith D Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, Aug 2022.
- [CT22b] Keith D Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, Aug 2022.
- [CWE24a] MITRE Content Team CWE. Cwe-1108: Excessive reliance on global variables, Oct 2024.
- [CWE24b] MITRE Content Team CWE. Cwe-120: Buffer copy without checking size of input (“classic buffer overflow”), Oct 2024.
- [CWE24c] MITRE Content Team CWE. Cwe-124: Buffer underwrite (“buffer underflow”) (4.11), Oct 2024.
- [CWE24d] MITRE Content Team CWE. Cwe-125: Out-of-bounds read, Oct 2024.

- [CWE24e] MITRE Content Team CWE. Cwe-1287: Improper validation of specified type of input, Oct 2024.
- [CWE24f] MITRE Content Team CWE. Cwe-129: Improper validation of array index, Oct 2024.
- [CWE24g] MITRE Content Team CWE. Cwe-131: Incorrect calculation of buffer size, Oct 2024.
- [CWE24h] MITRE Content Team CWE. Cwe-1339: Insufficient precision or accuracy of a real number, Oct 2024.
- [CWE24i] MITRE Content Team CWE. Cwe-135: Incorrect calculation of multi-byte string length, Oct 2024.
- [CWE24j] MITRE Content Team CWE. Cwe-685: Function call with incorrect number of arguments, Oct 2024.
- [CWE24k] MITRE Content Team CWE. Cwe-686: Function call with incorrect argument type (4.5), Oct 2024.
- [CWE24l] MITRE Content Team CWE. Cwe-761: Free of pointer not at start of buffer, Oct 2024.
- [CWE24m] MITRE Content Team CWE. Cwe-762: Mismatched memory management routines, Oct 2024.
- [CWE24n] MITRE Content Team CWE. Cwe-771: Missing reference to active allocated resource, Oct 2024.
- [CWE24o] MITRE Content Team CWE. Cwe-772: Missing release of resource after effective lifetime, Oct 2024.
- [CWE24p] MITRE Content Team CWE. Cwe-775: Missing release of file descriptor or handle after effective lifetime, Oct 2024.
- [CWE24q] MITRE Content Team CWE. Cwe-787: Out-of-bounds write, Oct 2024.
- [CWE24r] MITRE Content Team CWE. Cwe-805: Buffer access with incorrect length value (4.13), Oct 2024.
- [CWE24s] MITRE Content Team CWE. Cwe-822: Untrusted pointer dereference, Oct 2024.
- [CWE24t] MITRE Content Team CWE. Cwe-823: Use of out-of-range pointer offset, Oct 2024.
- [CWE24u] MITRE Content Team CWE. Cwe-824: Access of uninitialized pointer, Oct 2024.
- [CWE24v] MITRE Content Team CWE. Cwe-825: Expired pointer dereference, Oct 2024.
- [CWE24w] MITRE Content Team CWE. Cwe-826: Premature release of resource during expected lifetime, Oct 2024.
- [CWE24x] MITRE Content Team CWE. Cwe-843: Access of resource using incompatible type, Oct 2024.
- [CWE24y] MITRE Content Team CWE. Cwe list version 4.6, 2024.
- [CWE24z] MITRE Content Team CWE. Cwe view: Software development, 2024.
- [DC23] John-David Dalton and Lodash Contributors. lodash, May 2023.

- [dgi24] dgilland. Github - dgilland/pydash: The kitchen sink of python utility libraries for doing “stuff” in a functional way. based on the lo-dash javascript library., 2024.
- [GJS<sup>+</sup>24] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. Chapter 8. *classes*, 2024.
- [Gol24a] Golang. big package - math/big - go packages, 2024.
- [Gol24b] Golang. The go programming language specification - go.dev, Jun 2024.
- [Hic24] Rich Hickey. Clojure, 2024.
- [Hol06] G.J Holzmann. The power of 10: Rules for developing safety-critical code. *IEEE Computer*, 39(6):95–97, Jun 2006.
- [Hro04] Juraj Hromkovic. *Theoretical computer science : introduction to Automata, computability, complexity, algorithmics, randomization, communication, and cryptography*. Springer, Berlin, 2004.
- [HY00] Paul E. Hoffman and François Yergeau. Utf-16, an encoding of iso 10646, Feb 2000.
- [Int24] Intel. *Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume*, volume 2A. Intel, Oct 2024.
- [Ja23] Oracle (Java). Biginteger (java se 21), 2023.
- [JL96] Richard Jones and Rafael Lins. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., USA, 1996.
- [KB24] 7 Pernicious Kingdoms and Drew Buttner. Cwe-396: Declaration of catch for generic exception, Oct 2024.
- [KBW24] 7 Pernicious Kingdoms, Abhi Balakrishnan, and CWE ICS/OT SIG 62443 Mapping Fall Workshop. Cwe-416: Use after free, 2024.
- [Kin14] 7 Pernicious Kingdoms. Cwe-466: Return of pointer value outside of expected range, Oct 2014.
- [Kin24a] 7 Pernicious Kingdoms. Cwe - cwe-477: Use of obsolete function, 2024.
- [Kin24b] 7 Pernicious Kingdoms. Cwe-248: Uncaught exception, Oct 2024.
- [Kin24c] 7 Pernicious Kingdoms. Cwe-476: Null pointer dereference, Jan 2024.
- [KS24] 7 Pernicious Kingdoms and Martin Sebor. Cwe-252: Unchecked return value, Oct 2024.
- [KZA10] Sadaf Khalid, Saima Zehra, and Fahim Arif. Analysis of object oriented complexity and testability using object oriented design metrics. In *Proceedings of the 2010 National Software Engineering Conference*. Association for Computing Machinery, 2010.
- [LA04] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004.

- [LAC<sup>+</sup>15] Amit Levy, Michael P. Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. Ownership is theft: experiences building an embedded os in rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*, PLOS '15, page 21–26. Association for Computing Machinery, 2015.
- [Lef23] Vincent Lefèvre. The generic multiple-precision floating-point addition with exact rounding (as in the mpfr library). *Hal.science*, page 135–145, 2023.
- [Lim24] Arm Limited. *Arm® Instruction Set Reference Guide*. Arm Limited, 2024.
- [Mai24] Rust Maintainers. *Raii - rust by example*, 2024.
- [McB63] J Harold McBeth. Letters to the editor. *Communications of the ACM*, 6(9):575–575, Sep 1963.
- [MIT24a] MITRE, 2024.
- [MIT24b] MITRE. *Cwe - cwe top 25 most dangerous software weaknesses*, 2024.
- [N/A24] N/A. *Cwe-563: Assignment to variable without use*, Oct 2024.
- [Nor24] Henrik Nordgren. *The dash project*, Dec 2024.
- [Ora21] Oracle. *The java® language specification*, 2021.
- [PBS24] PLOVER, Abhi Balakrishnan, and CWE-CAPEC ICS/OT SIG. *Cwe-190: Integer overflow or wraparound*, Oct 2024.
- [Ped17] Juan Pedro. Persistence for the masses: Rrb-vectors in a systems language. *ACM on Programming Languages*, 1(ICFP):1–28, Aug 2017.
- [Pie02a] Benjamin C Pierce. *Types and programming languages*. The Mit Press, Cambridge, Mass. ; London, 2002.
- [Pie02b] Benjamin C Pierce. *Types and programming languages*. The Mit Press, Cambridge, Mass. ; London, 2002.
- [Pik11] Rob Pike. *The laws of reflection - go.dev*, Sep 2011.
- [Pik15] Rob Pike. *Errors are values - the go programming language*, 2015.
- [PLO24a] PLOVER. *Cwe-170: Improper null termination*, Oct 2024.
- [PLO24b] PLOVER. *Cwe-191: Integer underflow (wrap or wraparound)*, Oct 2024.
- [PLO24c] PLOVER. *Cwe-193: Off-by-one error*, Oct 2024.
- [PLO24d] PLOVER. *Cwe-197: Numeric truncation error (4.0)*, Oct 2024.
- [PLO24e] PLOVER. *Cwe-401: Missing release of memory after effective lifetime*, Oct 2024.
- [PLO24f] PLOVER. *Cwe-431: Missing handler*, Oct 2024.
- [Pra73] Vaughan R Pratt. Top down operator precedence. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 41–51, 1973.

- [PVJB12] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. Colt: Coalesced large-reach tlbs. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 258–269, 2012.
- [SAB98] AMR SABRY. What is a purely functional language? *Journal of Functional Programming*, 8(1):1–22, Jan 1998.
- [SBF12] Rifat Shahriyar, Stephen M. Blackburn, and Daniel Frampton. Down for the count? getting reference counting back in the ring. *SIGPLAN Not.*, 47(11):73–84, June 2012.
- [Sch24] JSON Schema, 2024.
- [Sha22] Yehonathan Sharvit. *Data-Oriented Programming*. Simon and Schuster, Sep 2022.
- [SV15] Michael J Steindorfer and Jurgen J Vinju. Optimizing hash-array mapped tries for fast and lean immutable jvm collections. *HAL (Le Centre pour la Communication Scientifique Directe)*, page 783–800, Oct 2015.
- [Tea24a] Go Team. Go wiki: cgo - the go programming language, 2024.
- [Tea24b] Golang Team. Go wiki: cgo - the go programming language, 2024.
- [Tea24c] LLVM Admin Team. The llvm compiler infrastructure project, 2024.
- [Tea24d] LLVM Admin Team. Llm language reference manual — llvm 16.0.0git documentation, 2024.
- [Tea24e] LLVM Admin Team. Mcjit design and implementation — llvm 20.0.0git documentation, 2024.
- [TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, Feb 1997.
- [UdM21] Sebastian Ullrich and Leonardo de Moura. Counting immutable beans: reference counting optimized for purely functional programming. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages, IFL '19*, New York, NY, USA, 2021. Association for Computing Machinery.
- [Ven14] Anonymous Tool Vendor. Cwe-562: Return of stack variable address, Oct 2014.
- [Yer03] F. Yergeau. Utf-8, a transformation format of iso 10646. *www.rfc-editor.org*, Nov 2003.
- [YLZ+23] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *USENIX*, 2023.