# Kipper - Programming Language for Improved Runtime Type-Safety

Diploma thesis

written as part of the

Matriculation and diploma examination

at the

Higher Department of Informatics

Handed in by:
Luna Klatzer
Lorenz Holzbauer
Fabian Baitura

Supervisor:
Peter Bauer

Project Partner:
Dr. Hanspeter Mössenböck, Johannes Kepler University

Leonding, April 4, 2025

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die Arbeit wurde bisher in gleicher oder ähnlicher Weise keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Die vorliegende Diplomarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Leonding, April 2025 L. Klatzer, L. Holzbauer & F. Baitura

# Abstract

Brief summary of our amazing work. In English. This is the only time we have to include a picture within the text. The picture should somehow represent your thesis. This is untypical for scientific work but required by the powers that are.


Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

# Zusammenfassung

Kurze Zusammenfassung unserer großartigen Arbeit. Auf Deutsch. Dies ist das einzige Mal, dass wir ein Bild in den Text einfügen müssen. Das Bild sollte in irgendeiner Weise Ihre Diplomarbeit darstellen. Dies ist untypisch für wissenschaftliche Arbeiten, aber von den zuständigen Stellen vorgeschrieben.


Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

# Contents


**1 Introduction** **1**

**2 Background** **2**

2.1 Dissecting the current issues . . . . . 2

2.1.1 The JavaScript problem . . . . . 2

2.1.2 TypeScript - One of many solutions . . . . . 5

2.2 How could it have been better . . . . . 8

2.2.1 Case study: Java . . . . . 8

2.2.2 Case study: Rust . . . . . 9

2.2.3 Drawing comparisons to JavaScript . . . . . 10

2.3 Tackling the issue at its core . . . . . 10

**3 Technology** **12**

3.1 Preface & Background . . . . . 12

3.2 Development Language . . . . . 12

3.2.1 Selection criteria and weighing the options . . . . . 13

3.2.2 Option - C++ . . . . . 13

3.2.3 Option - Java . . . . . 13

3.2.4 Option - TypeScript . . . . . 13

3.2.5 Result . . . . . 13

3.3 Parser & Lexer Generator . . . . . 13

3.3.1 Selection criteria and weighing the options . . . . . 13

3.3.2 Option - Antlr4 . . . . . 13

3.3.3 Option - Coco . . . . . 13

3.3.4 Result . . . . . 13

**4 Implementation** **14**

4.1 Compiler . . . . . 14

4.2 Semantic Analysis . . . . . 14

4.3 Type Analysis . . . 14
4.4 Type System . . . 14
4.4.1 Intended Purpose & Concept . . . 14
4.4.2 Existing Foundation & Environment . . . 14
4.4.3 Translating the foundation to Kipper . . . 18
4.4.4 Drawing comparisons to TypeScript . . . 18
4.4.5 Kipper Primitives . . . 18
4.4.6 Kipper Generics . . . 18
4.4.7 Kipper Interfaces & Duck-Typing . . . 18
4.4.8 Kipper Classes & Prototyping . . . 18
4.5 Output Generation . . . 18
4.5.1 Introduction . . . 18
4.5.2 Algorithm . . . 19
4.5.3 Algorithms used for Output Generation . . . 21
4.5.4 Requirements . . . 23
4.5.5 Differences between the Target Languages . . . 26
4.5.6 Stylistic Choices . . . 28
4.6 Integrated Runtime . . . 28
4.6.1 Runtime Type Concept . . . 28
4.6.2 Runtime Type Implementations in other Languages . . . 29
4.6.3 Runtime Base Type . . . 31
4.6.4 Runtime Built-in Types . . . 32
4.6.5 Runtime Errors . . . 34
4.6.6 Runtime Generation for Interfaces . . . 34
4.6.7 Matches Operator for Interfaces . . . 36
4.6.8 Typeof Operator . . . 39

**5 Compiler Reference** **41**
5.1 Compiler API . . . 41
5.2 Target API . . . 41
5.3 Shell CLI . . . 41

**6 Demo & Showcase** **42**
6.1 Working example in the web . . . 42
6.2 Working example using Node.js . . . 42

**7 Conclusion & Future** **43**

**Acronyms** **VI**

**Glossary** **VII**

**List of Figures** **IX**

**List of Tables** **X**

**List of Source Code Snippets** **XI**

**Appendix** **XII**

# 1 Introduction

# 2 Background

## 2.1 Dissecting the current issues

### 2.1.1 The JavaScript problem

JavaScript, originally developed by Netscape in 1995 to enable interactive web pages, has become the foundational programming language for modern web browsers with 60% of developers using the language in their profession as of 2023 [**?**]. While its initial scope was limited to enhancing the functionality of websites, JavaScript has since evolved into a versatile language that serves as the foundation for diverse applications, including those outside traditional browser environments. This success has been largely made possible by its versatile and modifiable nature allowing the language to take many shapes with a relatively consistent and easy-to-use syntax.

However, this rapid expansion was not accompanied by fundamental changes to the language's initial design, leading to inherent limitations and challenges when addressing complex, large-scale systems. In modern web development, JavaScript's role extends far beyond front-end scripting. Its omnipresence is reflected in its adoption across back-end platforms (e.g. Node.js), desktop applications (e.g. Electron), and mobile development frameworks (e.g. React Native). Accompanying this growth is a vast ecosystem of libraries, frameworks, and tools that offer developers flexibility in solving specific challenges. Despite these advantages, the language presents significant difficulties for developers.

#### Type Ambiguity and Uncontrolled Flexibility

JavaScript's dynamic typing provides developers with flexibility and expressiveness in their code. However, this flexibility can also lead to ambiguity and unintended behavior. Unlike statically typed languages, JavaScript does not enforce type constraints, meaning variable types are determined at runtime. It permits almost any value to be assigned to any variable without restrictions, relying solely on the developer to manage type consistency. This lack of enforcement makes the language prone to errors caused by implicit type coercion or unexpected values. Additionally, JavaScript does not inherently

handle type errors and assumes that most operations are valid, even providing special cases for operations that would typically be considered invalid.

For instance:

- A variable intended to store a number can unexpectedly hold a string due to developer oversight or API misuse, as can be seen in listing 1.

Listing 1: Unchecked variable assignments due to missing type definitions

```
let id; // Variable allowing any value

...

let resp = resp.json(); // Object: { id: "1234", ... }
id = resp.id; // Assigns a string even though that was not intended
```

- Implicit conversions, such as treating "null" or "undefined" as valid inputs in arithmetic operations, often yield confusing results, as can be seen in listing 2.

Listing 2: Broad ability to perform "invalid" operations despite clear error case

```
let discountRate = cart.appliedDiscount; // Let's assume it's actually "applied_discount" not "appliedDiscount" so it returns "undefined"

return price * (1 - discountRate); // -> yields NaN (Not A Number)
```

Such flexibility complicates debugging, as issues may only surface during runtime. This increases the risk of critical bugs making it into production and requires developers to rely heavily on additional tooling or rigorous testing to mitigate the risks inherent in dynamic typing.

#### Runtime-Bound Error Handling & Catching

Error handling in JavaScript is largely runtime-bound, except for syntax errors, and relies on tools like try-catch blocks and asynchronous error handling with Promise chains. This is because JavaScript is an interpreted language, meaning the program does not perform checks before executing each line. As a result, developers lack pre-execution validation.

While try-catch blocks can handle most errors, typically those defined by the developer, JavaScript allows certain operations that would be illegal in other languages, like

accessing non-existent properties by simply returning undefined instead of an error. This can make the underlying cause of an issue difficult to identify, with the code potentially failing at a different location, sometimes long after the original problem has occurred.

Examples include:

- Accessing a missing property, which returns "undefined", and later receiving an error due to "undefined" not being an object, as demonstrated in listing 3.

Listing 3: Accessing a missing property returns undefined which later causes an error

```
// Let's assume this is some kind of API response that incorrectly returned some data
let user = {
  name: "Alice"
};

...

const userAddress =  user.address; // Undefined property ("address" does not exist), returns simply "undefined"

...

let shippingCost = getShippingCost(address.city); // -> TypeError: Cannot read property "city" of undefined
```

- Misaligned function arguments, such as passing an object where a string was expected, going unnoticed until execution, as illustrated in listing 4.

Listing 4: Misaligned function arguments going unnoticed

```
function greet(name) {
  console.log("Hello, " + name + "!");
}

...

let user = { firstName: "Alice" };

greet(user); // Unintended behavior: "Hello, [object Object]!"
```

#### Modern approaches

These limitations force a defensive programming approach, requiring the developer to anticipate and safeguard against potential failures. While third-party tools such as linters or test frameworks can help identify issues, they are generally not equivalent to built-in, language-level type and error guarantees.

Given though that JavaScript is so prominent and can hardly be replaced in the modern web and server-side space, alternative solutions have been developed in response to these and other challenges to improve JavaScript's reliability and ease of use. One of the most prominent and widely adopted of these is TypeScript, which like the topic of this thesis, implements a transpilation-based language with independent libraries and functionality building on top of the existing JavaScript environment.

### 2.1.2 TypeScript - One of many solutions

TypeScript has emerged as the most widely adopted enhancement to JavaScript, functioning as a statically typed superset of the language. It introduces features such as object-oriented programming constructs and compile-time type checking, aligning its capabilities with those of traditionally typed languages like Java or C#. By providing type annotations and a robust compilation process, TypeScript enables developers to build safer applications compared to their JavaScript counterparts. Errors related to type mismatches, for instance, can be identified during development, reducing the likelihood of runtime failures and improving overall code reliability.

Despite its advantages, TypeScript is constrained by its core design philosophy of maintaining full compatibility with JavaScript. This approach allows developers to easily integrate TypeScript with existing JavaScript codebases, promoting incremental adoption. However, it also imposes limitations on the language's capabilities. For example, because JavaScript was not originally designed with type safety in mind, the TypeScript compiler operates as a static analysis tool, enforcing type rules only at compile time. This design choice ensures compatibility but leaves runtime type enforcement unaddressed. Consequently, developers must rely on a "trust-based" system, wherein the correctness of types is assumed during runtime based on the accuracy of their compile-time annotations.

These constraints highlight the challenges inherent in adapting a dynamically typed language to support static typing. While TypeScript significantly mitigates many of

JavaScript's shortcomings, its reliance on compile-time type checking alone limits its ability to provide comprehensive runtime guarantees, requiring developers to remain vigilant when integrating with dynamically typed JavaScript components.

### Unchecked compile-time casts

As already mentioned TypeScript works on a compile-time-only basis, which does not allow for any runtime type checks. That also naturally means any standard functionality like casts can also not be checked for, since such type functionality requires the language to be able to reflect on its type structure during runtime. Given the fact though that casts, which allow the developer to narrow the type of a value down, are a necessity in everyday programs, TypeScript is forced to provide what you can call "trust-based casts". The developer can, like in any other language, specify what a specific value is expected to be, but unlike usual casts are primarily unchecked, meaning you can, if you want, cast anything to anything with no determined constraint.

While in principle this maintains the status quo and provides the developer with more freedom, it also opens up another challenge that must be looked out for when writing code. If one of those casts goes wrong and isn't valid, the developer will only know that at runtime and will have no assistance to fix it. To overcome this developers can themselves implement runtime type checks, which prevent type mismatches in ambiguous contexts. While it is a common approach, it is fairly impractical and adds a heavy burden on the developer as it requires constant maintenance and recurring rewrites to ensure the type checks are up-to-date and valid.

Let's look at an example 5.

Listing 5: Unchecked compile-time casts in TypeScript

```
1  class SuperClass {
2    name: string = "Super class";
3  }
4
5  class MiddleClass extends SuperClass {
6    superField: SuperClass = new SuperClass();
7
8    constructor() {
9      super();
10   }
11 }
12
13 class LowerClass extends MiddleClass {
14   classField: MiddleClass = new MiddleClass();
15
```

```
16    constructor() {
17      super();
18    }
19  }
20
21  const c1 = <MiddleClass>new SuperClass(); // Unchecked
      cast
22  console.log(c1.superField.name); // Runtime Error!
      Doesn't actually exist
23
24  const c2 = <LowerClass>new MiddleClass(); // Unchecked
      cast
25  console.log(c2.classField.superField.name); // Runtime
      Error! Doesn't actually exist
```

Here we have a simple example of an inheritance structure, where we access the properties of a child that is itself also another object. Due to the nature of TypeScript operations such as casts are mostly unchecked and usually work on the base of trusting the developer to know what they're doing. That means that in the example given above, the compiler does not realise that the operation the developer is performing is invalid and will result in a failure at runtime (can't access property "name", c1.superField is undefined). Furthermore, given that JavaScript only reports on such errors when a property on an undefined value is accessed, the undefined variable may go unused for a while before it is the cause of any problem. This leads to volatile code that can in many cases not be guaranteed to work unless the developer actively pays attention to such errors and makes sure that their code does not unintentionally force unchecked casts or other similar untyped operations.

#### Ambiguous dynamic data

Another similar issue occurs when dealing with dynamic or untyped data, which does not report on its structure and as such is handled as if it were a JavaScript value, where all type checks and security measures are disabled. This for one makes sense given the goal of ensuring compatibility with the underlying language, but it also creates another major problem where errors regarding any-typed values can completely go undetected. Consequently, if we were to receive data from a client or server we can not ensure that the data we received is fully valid or corresponds to the expected pattern. This is a problem that does not have a workaround or a solution in TypeScript.

For example 6:

Listing 6: Ambiguous dynamic data in TypeScript

```
interface Data {
  x: number;
  y: string;
  z: {
    z1: boolean;
  }
}

function receiveUserReq(): object {
  // ...
  return {
    x: "1",
    y: "2",
    z: true
  }
}

var data = <Data>receiveUserReq(); // Unsafe casting with unknown data
console.log(data.z.z1); // No Runtime Error! But returns "undefined"
```

For the most part, developers are expected to simply watch out for such cases and implement their own security measures. There are potential libraries which can be utilised to add runtime checks which check the data received, but such solutions require an entirely new layer of abstraction which must be managed manually by a developer. This additional boilerplate code also increases the complexity of a program and has to be actively maintained to keep working.

Good examples of technologies that provide runtime object schema matching are "Zod" [?] and "joi" [?]. Both are fairly popular and actively used by API developers who need to develop secure endpoints and ensure accurate request data. While they are a good approach to fixing the problem after the fact, they still create their own difficulties. We will examine these later in the implementation section, where we will more thoroughly compare Kipper's approach to other tools.

## 2.2 How could it have been better

### 2.2.1 Case study: Java

TypeScript has emerged as the most widely adopted enhancement to JavaScript, functioning as a statically typed superset of the language. It introduces features such as object-oriented programming constructs and compile-time type checking, aligning its

capabilities with those of traditionally typed languages like Java or C#. By providing type annotations and a robust compilation process, TypeScript enables developers to build type-safe applications. Errors related to type mismatches, for instance, can be identified during development, reducing the likelihood of runtime failures and improving overall code reliability. Despite its advantages, TypeScript is constrained by its core design philosophy of maintaining full compatibility with JavaScript. This approach allows developers to seamlessly integrate TypeScript with existing JavaScript codebases, promoting incremental adoption. However, it also imposes limitations on the language's capabilities. For example, because JavaScript was not originally designed with type safety in mind, the TypeScript compiler operates as a static analysis tool, enforcing type rules only at compile time. This design choice ensures compatibility but leaves runtime type enforcement unaddressed. Consequently, developers must rely on a "trust-based" system, wherein the correctness of types is assumed during runtime based on the accuracy of their compile-time annotations. These constraints highlight the challenges inherent in adapting a dynamically typed language to support static typing. While TypeScript significantly mitigates many of JavaScript's shortcomings, its reliance on compile-time type checking alone limits its ability to provide comprehensive runtime guarantees, requiring developers to remain vigilant when integrating with dynamically typed JavaScript components.

### 2.2.2 Case study: Rust

Rust is a systems programming language designed to offer memory safety without a garbage collector. One of the standout features of Rust is its ownership system, which enforces strict rules for memory allocation and deallocation, preventing common bugs like null pointer dereferencing or data races in concurrent programming. This guarantees memory safety at compile-time without needing a runtime environment to manage memory, unlike languages such as Java and JavaScript, which use garbage collection to manage memory dynamically.

Rust's type system is strongly and statically typed, like Java, but it emphasizes immutability and borrowing concepts to manage data lifetimes and concurrency safely. Unlike Java, Rust does not have reflection, but it provides powerful meta-programming features via macros. Rust also promotes zero-cost abstractions, ensuring that high-level abstractions have no runtime overhead, making it a popular choice for applications requiring both performance and safety. Despite working on the basis of a completely

different programming paradigm it still manages to be type-safe or more accurately memory-safe. The compiler makes sure that there are no ambiguities left that could potentially lead to runtime errors and provides absolute safety in a way that still allows a certain freedom to the developer.

### 2.2.3 Drawing comparisons to JavaScript

Unlike the two languages we've just described, JavaScript is rather unique in its design and structure. As already mentioned, there is no proper reflection system, enforced type checks or type safety when running code, only really throwing errors when there is no other way around it. Moreover, you can say that JavaScript has no design or structure at all, and was more conceptualised as a fully dynamic type-less language with no OOP support in mind. This has caused quite a few problems in the years following the original version of JavaScript, as it has more and more developed into an OOP language while not providing any proper type functionality commonly present in such systems. Even languages like Python, which is also a dynamic interpreted language, provide static type hints and checks to ensure proper type safety when writing code.

Nonetheless, as JavaScript is currently one of the most important languages out there, the system can under no circumstances be changed as it would break backwards compatibility with previous systems and destroy the web as we know it today. This has caused quite a dilemma, which persists until today. Many tools like TypeScript have been developed since then and are seen as the de-facto solution for these problems, but it's a rather bad solution given all the current restraints, unavoidable edge cases and vulnerabilities that can be easily introduced.

## 2.3 Tackling the issue at its core

As we have already mentioned, JavaScript is a language that can under no circumstances be changed or it would mean that most websites would break in newer browser versions. This phenomenon is also often described as "Don't break the web", the idea that any new functionality must incorporate all the previous standards and systems to ensure that older websites work and look the same. Naturally this also then extends to TypeScript, which has at its core a standard JavaScript system that can also not be changed or altered to go against the ECMAScript standard. Consequently, the most effective approach to ensuring a safe development environment for developers is to build upon

JavaScript by extending its standard functionalities through a custom, unofficial system that incorporates the necessary structures and safety measures.

This is where Kipper comes into play—a language that implements a custom system, which is later transpiled into JavaScript or TypeScript. By incorporating an additional runtime and non-standard syntax, Kipper enables runtime type checks, addressing gaps left by TypeScript's compile-time-only system. This approach is particularly advantageous, as it allows the system to extend beyond JavaScript standards, introducing structures tailored to the requirements of modern programs. Accordingly, developers can rely on Kipper to enhance code security and ensure that no dynamic structures remain unchecked or bypass the type system due to edge cases.

# 3 Technology

## 3.1 Preface & Background

Before introducing the technologies and tools used in the Kipper project, it is important to note that the project existed on a smaller scale prior to the initiation of this diploma thesis and the following sections will talk about technologies that were chosen before the start of this thesis project. As such, when the project was reimagined and expanded into its current form as a diploma thesis, the development technologies were already determined and simply continued to allow building on top of the existing foundation.

However, to highlight the various differences and the reasons behind the original decisions for the parser, lexer and core compiler, the sections will still consider each option, talk about their viability in the context of this project and elaborate on the decisions made when the initial project was envisioned.

## 3.2 Development Language

The choice of development language, or compiler programming language, is a crucial initial decision when starting a project of this nature. This decision significantly impacts factors such as distribution, accessibility, and cross-platform integration. The selected language sets the foundational conditions for the entire project and influences the ability to effectively utilize the language being created.

Many compilers are initially developed in a different language, often one closely related to the language being designed. Once the language takes on proper shape, the compilers are then often migrated into the newly created language, effectively becoming one of the first test programs to utilize the language directly. This approach allows the compiler to validate its own functionality—serving as a self-serving cycle where each iteration of updates also benefits the compiler itself.

This is not the case here, however. As the Kipper compiler remains highly complex, it is currently written in another language. Nonetheless, the compiler could be migrated to the Kipper language in the future if the required changes are implemented.

### 3.2.1 Selection criteria and weighing the options

### 3.2.2 Option - C++

### 3.2.3 Option - Java

### 3.2.4 Option - TypeScript

### 3.2.5 Result

## 3.3 Parser & Lexer Generator

### 3.3.1 Selection criteria and weighing the options

### 3.3.2 Option - Antlr4

### 3.3.3 Option - Coco

### 3.3.4 Result

# 4 Implementation

## 4.1 Compiler

The Kipper Compiler is the core component of the Kipper project, serving as the central piece that connects the Kipper language to its target environment. It functions similarly to other transpilation-based compilers, such as the TypeScript compiler, by producing high-level output code from high-level input code—specifically, code written in the Kipper language. The syntax of this language is predefined and implemented using the lexer and parser generated by the Antlr4 parser generator.

## 4.2 Semantic Analysis

## 4.3 Type Analysis

## 4.4 Type System

### 4.4.1 Intended Purpose & Concept

### 4.4.2 Existing Foundation & Environment

As Kipper is a language that transpiles to either JavaScript or TypeScript (a superset of JavaScript), it is essential to consider the existing foundation and environment when designing and developing its type system. While Kipper is not as constrained as TypeScript, ensuring seamless interoperability and ease of translation with the underlying environment remains critical. This is particularly important since Kipper is designed to run both on the web and locally using Node.js or other JavaScript runtimes.

To achieve this, it is crucial to understand the foundation upon which JavaScript itself was built. JavaScript, created in 1995 by Brendan Eich, was initially envisioned as a lightweight scripting language to add interactivity to web pages. Its dynamic and loosely-typed nature provided flexibility and adaptability but also introduced

susceptibility to errors in larger applications. Over time, JavaScript has been refined to improve its reliability and scalability for handling complex applications. However, its core type system has largely remained the same and hasn't been altered as a safe guard to ensure that backwards compatibility is preserved.

### Weak Dynamic Type System

JavaScript is a dynamically typed language, meaning variables can hold values of different types during runtime. Unlike statically typed languages such as TypeScript or Java, JavaScript does not require explicit declaration of a variable's data type.

Listing 7: Holding Values of different Types during Runtime

```
let foo = 10; // foo is a number
foo = "Hello"; // foo is now a string
foo = [1, 2, 3, 4, 5]; // foo is now an array
```

In contrast, TypeScript enforces static typing, ensuring that a variable's type remains consistent throughout its lifecycle. This prevents unintentional type changes and improves code reliability.

Listing 8: Statically Typed Language TypeScript

```
let foo: number = 1; // x is explicitly declared as a
    number
foo = "Hello"; // Error: Type 'string' is not assignable
    to type 'number"
```

JavaScript is also a weakly typed language, meaning that it allows operations between different data types without the need for explicit type conversion. This flexibility can sometimes lead to unexpected results, as JavaScript automatically coerces values to the appropriate type when performing operations.

Listing 9: Automatic type conversion in JavaScript

```
let quantity = 7; // quantity is an integer
let value = "20"; // value is a string

let total = quantity + value; // JavaScript
    automatically converts quantity to string
console.log(total); // Output: "720"
```

In the example above, the number 7 is stored in the variable quantity, and the string "20" is stored in the variable value. Typically, when attempting to add a number and a string, one might expect an error due to the mismatch in data types. However,

JavaScript performs implicit type coercion, automatically converting the number into a string before performing the operation.

In this case, JavaScript converts the number 7 to the string "7" and concatenates it with the string "20", resulting in the string "720". This type conversion occurs implicitly, without the need for explicit instructions to JavaScript.

However, implicit type coercion can sometimes lead to unintended results if not handled carefully. It is important to understand how JavaScript performs these conversions to prevent unexpected behavior in your code. Awareness of these implicit conversions helps ensure that operations between different data types do not produce erroneous or undesirable outcomes.

### Primitives & Core Types

In JavaScript, a primitive (or primitive value, primitive data type) is a data type that is not an object and does not have methods or properties. There are seven primitive data types:

- string
- number
- bigint
- boolean
- undefined
- symbol
- null

Unique among the primitive types in JavaScript are undefined and null.

In JavaScript, null is considered a primitive value due to its seemingly simple nature. However, when using the typeof operator, it unexpectedly returns "object". This is a known quirk in JavaScript and is considered a historical bug in the language that has been maintained for compatibility reasons.

Listing 10: typeof null return "object" in JavaScript

```
1   console.log(typeof null); // "object"
```

undefined is a primitive value automatically assigned to variables that have been declared but not yet assigned a value, or to formal function parameters for which no actual arguments are provided.

Listing 11: typeof null return "object" in JavaScript

```
1  let item; // declare a variable without assigning a
       value
2
3  console.log(‘The value of item is ${item}‘); // logs
       "The value of item is undefined"
```

All primitives in JavaScript are immutable, meaning they cannot be altered directly. It is important to distinguish between a primitive value and a variable that holds a primitive value. While a variable can be reassigned to a new value, the primitive value itself cannot be modified in the same way that objects, arrays, and functions can be altered. JavaScript does not provide utilities to mutate primitive values.

Primitives, such as numbers and strings, do not inherently have methods. However, they appear to behave as though they do, due to JavaScript's automatic wrapping, or "auto-boxing," of primitive values into their corresponding wrapper objects. For example, when a method like toString() is called on a primitive number, JavaScript internally creates a temporary Number object. The method is then executed on this object, not directly on the primitive value.

Consider the primitive value value = 10;. When a method such as value.toString() is invoked, JavaScript automatically wraps the primitive 10 in a Number object and calls Number.prototype.toString() on it. This behavior occurs invisibly to the programmer and serves as a helpful mental model for understanding various behaviors in JavaScript. For example, when attempting to "mutate" a primitive, such as assigning a property to a string (str.foo = 1), the original string is not modified. Instead, the value is assigned to a temporary wrapper object.

#### Custom Dynamic Structures

#### Prototype Inheritance System

### 4.4.3 Translating the foundation to Kipper

### 4.4.4 Drawing comparisons to TypeScript

### 4.4.5 Kipper Primitives

### 4.4.6 Kipper Generics

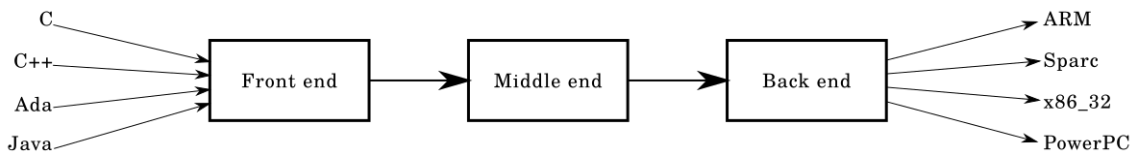
### 4.4.7 Kipper Interfaces & Duck-Typing

### 4.4.8 Kipper Classes & Prototyping

## 4.5 Output Generation

### 4.5.1 Introduction

The Kipper compiler adopts a modular architecture that enables the definition of custom targets, providing flexibility in adapting to different use cases. This modularity extends beyond configuration, allowing developers to specify target-specific features or behaviors. This is implemented by seperating the compiler in two parts, a frontend and a backend. In case of the GCC compiler, the modularity is achieved by seperating the compiler in three parts 1. The frontend verifies the syntax and semantics, scans the input and performs type checking. It then generates an intermediate representation of the code. The middleend then optimizes this intermediate representationm which is independent of the cpu target. Examples for middle-end code optimizations include dead code removal and and detecting unreachable code. The backend takes the optimized intermediate code and generates the target-dependant code. In the case of GCC, it generates assembly [?].




Source: https://commons.wikimedia.org/wiki/File:Compiler_design.svg

Figure 1: The design of the GCC compiler

As of now Kipper produces either TypeScript or JavaScript code. In the Kipper CLI utility, the flag ”–target=js|ts” sets the target language. When not specified, the default target is JavaScript.

Currently Kipper does not have a middleend, due to the additional complexity of a full middleend and the only marginal improvements in code performance. Instead, the frontend passes the AST directly to the backend. We do perform post-analysis optimization, although this is currently limited to treeshaking.

### 4.5.2 Algorithm

Kipper uses an AST as the main way of translating source code into the target language. It serves as a hierarchical representation of the source code. The root node of the AST correpsonds to the entire program, with each child node representing a specific statement, scope or other construct. The nodes contain the semantic and type-semantic data of a statement, as well as a string representation of the original parser node. Each node additionaly has a ”kind”, which is a unique number that is hard-coded in the compiler. This helps to get the type of the statement, which can be a declaration, an expression or an assignment. Every node has a list of children, which represent nested or dependent components of the construct. This data is then wrapped and sent to the code generator.

Kipper is designed to process the AST nodes in a bottom-to-top manner. This means, that the translation of the children is completed, before the parents get processed. Therefore the parent can extract information from its children. Complex structures depend on this, as they use their childrens generated code and simply embed it into their own. The code gets passed to the parent as a string. This is completely safe, as the AST has already been checked before regarding syntactical and semantical correctness. Each node is respnsible for it’s own output code. This localized decision-making ensures modularity and flexibility.

Kippers output generation works by setting up the target, and then generating the requirements. These are described in detail in section 4.5.4. After the setup the compiler iterates over the children and calls the ”translateCtxAndChildren” function, which recursively walks down the children’s children and generates the code of the the nodes. Each node returns a string, which gets processed by it’s parent node, which returns a string as well until the process is back at the root node. In the root node, the strings

get merged and sent to a function that writes the code to a file. The implementation of the algorithm can be seen in example 12.

Listing 12: The translation algorithm

```
public async translate():
    Promise<Array<TranslatedCodeLine>> {
  // SetUp and WrapUp functions
  const targetSetUp: TargetSetUpCodeGenerator =
    this.codeGenerator.setUp;
  const targetWrapUp: TargetWrapUpCodeGenerator =
    this.codeGenerator.wrapUp;

  // Add set up code, and then append all children
  const { global, local } = await
    this.programCtx.generateRequirements();
  let genCode: Array<TranslatedCodeLine> = [...(await
    targetSetUp(this.programCtx, global)), ...local];
  for (let child of this.children) {
    genCode.push(...(await
      child.translateCtxAndChildren()));
  }

  // Add wrap up code
  genCode.push(...(await targetWrapUp(this.programCtx)));

  // Finished code for this Kipper file
  return genCode;
}
```

The code generation function of a node takes the node as an argument and gets it's semantic and typesemantic data. This is then used to translate the children, which are properties of the semantic data, into source code. The result is then concatenated into a single string array and returned. This process can be seen in example 13. In this example, the instanceOf expression gets built.

Clearly visible is the same "translateCtxAndChildren" function as in example 12. This is the treewalker function. It visits every node in the AST and starts the code generation by the bottom-most node. It generates the code for a nodes children and then the node itself. This allows a node to control the order of compilation of their children.

Listing 13: The code generation function of a instanceOf expression

```
instanceOfExpression = async (node:
    InstanceOfExpression): Promise<TranslatedExpression>
    => {
  const semanticData = node.getSemanticData();
  const typeData = node.getTypeSemanticData();
```

```
4     const operand = await
          semanticData.operand.translateCtxAndChildren();
5     const classType =
          TargetJS.getRuntimeType(typeData.classType);
6
7       return [...operand, " ", "instanceof", " ",
            classType];
8     };
```

The code generator functions of Kipper are implemented in a class called ”JavaScript-TargetCodeGenerator”. Due to the similarity between TypeScript and JavaScript, the TypeScript code generator extends the JavaScript code generator and overrides the functions that differ. This eliminates duplicate code fragments.

### 4.5.3 Algorithms used for Output Generation

There is a plethora of algoritms available to generate code in the target language. They can be classified by their input data structure. Some algoritms need a tree-like IR, others need a linear IR structure.

#### Linear Algorithms

Linear algoritms are most often used when compiling a high-level language to machine code or bytecode. They treat the input as a flat, ordered sequence and sequentially process it. Intermediate representations often are either three-address-code or static-single-assignment code. Linear algorithms process the code one instruction at a time in a sequence. When an instruction is complete, it gets pushed to a list, which in the end is concatenated and written to the output file. When using linear algorithms, there are two major ways of representing the IR.

Three-address code consists of three operands and is typically an assignment and a binary operator [?]. An instruction can have up to three operands, although it may have less. In the example 14, the problem gets split up into multiple instructions. This allows the compiler to easily transform the instructions into assembly language or bytecode, which are similar in their structure. Furthermore, the compiler can identify unused code by checking if a variable gets used later on in the code.

Listing 14: Three-address code

```
1 // Problem
2 x = (-b + sqrt(b^2 - 4*a*c)) / (2*a)
3
```

```
// Solution
t1 := b * b
t2 := 4 * a
t3 := t2 * c
t4 := t1 - t3
t5 := sqrt(t4)
t6 := 0 - b
t7 := t5 + t6
t8 := 2 * a
t9 := t7 / t8
x := t9
```

The Static single-assignment form is a different intermediate representation, where each variable gets assigned exactly once [**?**]. It is used in the most widely in compilers like GCC. The main benefit is that it simplifies the code and improves the results of compiler optimizations. In example 15, the problem shows a variable ”y” that gets assigned twice. The first assignment is therefore unnecessary. With the static single-assignment form, the compiler can determine that the assignment of ”y1” was unnecessary, because it is never used in the code that comes after. Examples of optimizations that were enhanced by this form include dead-code elimination, constant propagation and register allocation.

Listing 15: Static single-assignment form

```
// Problem
y := 1
y := 2
x := y

// Solution
y1 := 1
y2 := 2
x1 := y2
```

### Tree-based Algorithms

Tree-based algorithms are often used in transpilers, as the code needs to stay human readable most of the time, and the general structure of the code should be preserved. This means, that the strucure of scopes and statements should roughly stay the same. Therefore the components are stored in a node in a tree-like structure. Kipper uses a bottom-up code generation algorithm. This means, that a treewalker recurses through the tree and generates the output starting from the most deeply nested node. We chose to use this method, because it was easier to visualize and implement and we wanted the source to stay human readable and extendable without aggressive optimizations and

changes on the source. This is not possible with linear algorithms, as they transform the code either into the three-address code form or the static single-assignment form. While this happend, important information about the context and structure of the code gets lost.

Tree-based code generation can be used to generate bytecode. This bytecode gets optimized and further compiled by a linear algorithm. Tree-based algorithms have the disadvantage of allowing only local optimizations in the respective nodes. In addition, they can be computationally expensive, in case the input gets too complex, as the treewalker is a recursive function. Therefore, in professional compilers the tree-based design is not often used. GCC uses a tree-based design in two of it's language independent IRs, GIMPLE and GENERIC [**?**].

### 4.5.4 Requirements

Kipper is designed to have a runtime that is as small as possible while still having all the needed functionality bundled in it. This means, that the compiler should only include functions and objects into the runtime, that are needed by the user. This alligns with our goal of keeping the compiler modular and minimal. Due to the removal of unused components in a process called "treeshaking", the output code is kept small and efficient.

Kipper includes a range of built-in functions and features to support its runtime environment. These built-ins are organized and managed using a scoped approach. The global scope contains core runtime features that are always required, such as basic type handling and error reporting. Beyond the global scope, additional features are selectively included based on the specific requirements of the program being compiled.

#### Conditional features

Conditional features are runtime components that are included only when explicitly required by the program being compiled. These features can range from commonly used operations like match and slice to more specialized or program-specific utilities. Unlike essential runtime components housed in the global scope, conditional features are added selectively based on an analysis of the program's structure and functionality. The decision to include conditional features occurs during the requirements generation phase. As the compiler traverses the Abstract Syntax Tree (AST), it examines each node to determine whether a specific built-in function or runtime operation is invoked.

If a feature like slice is used in the source code, it is flagged as necessary and included in the consolidated requirements list. This ensures that only the relevant components are integrated into the final runtime environment.

An example of a conditional feature would be the ”slice” function 16. This function takes an array as input and extracts a section starting from the index the first argument provides and ending at the index the second argument provides.

Listing 16: The Slice Operation

```
var valid: str = "321";
print(valid[1:2]); // 2
```

When the compiler encounters the slice operator, it registers the function as needed and therefore includes it into the runtime. This process can be seen in example 17. The compiler calls slice function in the ”BuiltInGenerator”. The ”BuiltInGenerator” is a collection of functions that generate the code for the built in functions in the target languages. This means that each target language has to have a full set of ”BuiltInGenerators”. The generator in the example generates the required JavaScript code for the function by utilizing the built in ”slice” function of JavaScript.

Listing 17: Slice in the JavaScript BuiltInGenerator

```
async slice(funcSpec: InternalFunction):
    Promise<Array<TranslatedCodeLine>> {
  const signature = getJSFunctionSignature(funcSpec);
  const objLikeIdentifier = signature.params[0];
  const startIdentifier = signature.params[1];
  const endIdentifier = signature.params[2];

  return genJSFunction(
    signature,
    `{ return ${objLikeIdentifier} ?
      ${objLikeIdentifier}.slice(${startIdentifier},
      ${endIdentifier}) : ${objLikeIdentifier}; }`,
  );
  }
```

The generated "slice" function can be found in example 18.

Listing 18: Slice in the target language

```
slice: function slice<T>(objLike: T, start: number |
    undefined, end: number | undefined): T {
  return objLike ? objLike.slice(start, end) : objLike;
  },
```

### The global scope

The global scope in programming represents the top-level execution context where variables, functions, and objects are accessible throughout the entire runtime environment unless explicitly restricted. In JavaScript, the global scope is particularly significant as it varies across runtime environments like browsers, Node.js, and Web Workers. This variability necessitates robust mechanisms for identifying and managing the global context to ensure compatibility across environments.

For example, JavaScript defines several global objects, such as window in browsers, global in Node.js, and self in Web Workers. Modern JavaScript unifies these under globalThis, a standardized global object that provides a consistent way to access the global scope regardless of the environment. However, not all environments support globalThis, which is why fallback mechanisms are often used.

The Kipper global scope contains all the runtime features that are required. It is important, that the global scope exists only once, therefore the programm needs to check at runtime, if the scope already exists. This can be seen in example 19. It first checks if "___globalScope" is already defined and uses it if available. If not, it attempts to use globalThis, the modern standard JavaScript global object. If "globalThis" is not defined, it checks for window in browser environments, global in Node.js, or self in Web Workers. If none of these are defined, it falls back to an empty object. This ensures that the "___globalScope" variable is always initialized, regardless of the environment, allowing consistent and safe access to the global scope.

Listing 19: Global Scope Logic

```
1 var __globalScope = typeof __globalScope !== "undefined"
    ? __globalScope :
2     typeof globalThis !== "undefined" ? globalThis :
3     typeof window !== "undefined" ? window :
4     typeof global !== "undefined" ? global :
5     typeof self !== "undefined" ? self : {};
```

### Internal functions

Internal functions in Kipper serve as an essential part of the runtime, yet they are designed to remain hidden from the user-facing API. These functions provide support for various runtime operations and compiler processes but are not directly accessible or callable in the user's program. This ensures that the runtime environment remains clean and minimal while still delivering the required functionality.

An example of an internal function would be the ”assignTypeMeta” function, which adds metadata to a runtime type. This is useful for runtime type comparison and described in detail in chapter 4.6.4. This function never gets exposed to the user but is called internally when an interface gets declared.

A key characteristic of internal functions is their dynamic inclusion in the runtime environment. During the requirements generation phase, the compiler identifies whether a program's functionality depends on any internal mechanisms. If so, the corresponding internal functions are included in the runtime.

#### Requirements Generation

The requirements generation process in Kipper begins during the compilation phase. As the compiler traverses the Abstract Syntax Tree (AST), it analyzes the nodes to determine the features needed by the program. This analysis produces a set of requirements, which are then used to configure the runtime environment. This works by using feature registration. If an AST node needs a certain feature, the reference is added to the program context by using the ”this.programCtx.addInternalReference” function. When a feature is added more once, all further additions get ignored, as the function is already available. After all the features are registeres, the target generates the source code in the required language and inserts it into the output code.

### 4.5.5 Differences between the Target Languages

The implementation of a compiler or transpiler targeting multiple programming languages often requires handling the specific quirks and requirements of each target. As Kipper is a webdevelopment language, we target both TypeScript and JavaScript. Both languages share a common foundation but diverge significantly in their syntax rules, semantics, and type systems.

One of the primary challenges in supporting both JavaScript and TypeScript as target languages is their differing treatment of identifiers, reserved keywords, and type declarations. While JavaScript is dynamically typed and relatively permissive in terms of variable naming and usage, TypeScript enforces a stricter set of rules due to its static type-checking capabilities.

### Reserved Keywords

Both JavaScript and TypeScript have a set of reserved keywords that cannot be used as identifiers. However, TypeScript introduces additional constraints by reserving type-related keywords, which are not present in JavaScript. For instance, class is a reserved keyword in both languages and cannot be used as a variable name. In contrast, TypeScript also reserves names like let, number, and other type names, making them invalid as variable or function names.

Kipper handles these reserved keywords by checking for them at compile time. The compiler compares against a hardcoded list of keywords and in case it finds one, it throws an "ReservedIdentifierOverwriteError". This list of keywords contains both the reserved words of JavaScript and TypeScript, as this minimizes redundancy and complexity. In addition to that, it also forces the developer to use sensible variable names, as JavaScript is quite leaneant with it's reserved keywords. Example 20 illustrates this.

Listing 20: Reserved Keywords in TS and JS

```
1 // Invalid in TypeScript
2 let let = 5;  // Error: Cannot use 'let' as an identifier
3 let number = 10;  // Error: Cannot use 'number' as an
     identifier
4
5 // Valid in JavaScript
6 var let = 5;  // No error
7 var number = 10;  // No error
```

### Type Annotations

TypeScript introduces type annotations as part of its static type system. This means that while generating the TypeScript output, Kipper has to append type information in variable assignments, functions and lambdas. This works by overriding the JavaScript implementation of the code generator function and converting the AST-internal type of the node to a TypeScript type. Example 21 shows the difference between the JavaScript code generator function and the TypeScript code generator function for variable assignments. In the codeblock that generates TypeScript, there is an additional bit of code after the storage and the identifier, that inserts the type of the object. The function declaration and lamdas work similarly.

Listing 21: Difference in code generator functions

```
1 // JavaScript
2 return [
```

```
3     [
4       storage, " ",
5       semanticData.identifier, ...(assign.length > 0 ? ["
          ", "=", " ", ...assign] : []), ";"
6     ]
7   ];
8   // Result: let x = 5;
9
10  // TypeScript
11  return [
12    [
13      storage, " ", semanticData.identifier, ":", " ",
          tsType, // This inserts the type
14      ...(assign.length > 0 ? [" ", "=", " ", ...assign] :
          []),
15      ";",
16    ],
17  ];
18  // Result: let x: number = 5;
```

### 4.5.6 Stylistic Choices

## 4.6 Integrated Runtime

### 4.6.1 Runtime Type Concept

The primary goal of the Kipper runtime type system is to allow untyped values to be compared with defined types, such as primitives, arrays, functions, classes, and interfaces, removing any ambiguities that could cause errors. During code generation, all user-defined interfaces are converted into runtime types that store the information needed to perform type checks. These are then utilised alongside the built-in runtime types, such as "num", "str" or "obj", to enable the compiler to add necessary checks and runtime references for any cast, match or typeof operation.

With the exception of interfaces, classes, and generics, types are primarily distinguished by their names. In these cases, type equality checks are performed using nominal comparisons, where the name acts as a unique identifier within the given scope e.g. type "num" is only assignable to "num". For more complex structures, additional information—such as members or generic parameters—is also considered.

In the case of interfaces, the names and types of fields and methods are used as discriminators. These fields and methods represent the minimum blueprint that an object

must implement to be considered compatible with the interface and thus "assignable." In this regard, Kipper adopts the same duck-typing approach found in TypeScript.

For generics, which include "Array<T>" and "Func<T..., R>", the identifier is used alongside the provided generic parameters to determine assignability. This ensures that when one generic is assigned to another, all parameters must match. For instance, "Array<num>" cannot be assigned to "Array<str>" and vice versa, even if their overall structure is identical.

For user-defined classes, the compiler relies on the prototype to serve as the discriminator. In practice, this behaviour is similar to that of primitives, as different classes cannot be assigned to each other.

To ensure future compatibility with inheritance, Kipper also includes a "baseType" property, which allows types to be linked in an inheritance chain. However, this feature is currently unused.

### 4.6.2 Runtime Type Implementations in other Languages

#### Nominal Type Systems

Nominal type systems are used in most modern object-orientated programming languages like Java and C#. In these systems, types are identified by their unique names and can only be assigned to themselves. Additionally, two types are considered compatible, if one type is a subtype of the other one, as can bee seen in listing 22. Here a "Programmer" is an "Employee", but not the other way around. This means that "Programmer" instances have all the properties and methods an "Employee" has while also having additional ones specific to "Programmer". The relationships are as such inherited, so a "SeniorDeveloper" is still an "Employee" and a "Programmer" at the same time. Even though the Senior Developer adds no new functionality to the "Programmer", it is not treated the same. Nominal typing improves code readability and maintainability, due to the explicit inheritance declaration. On the other hand, this increases code redundancy for similar or even identical but not related structures.

Listing 22: Example of nominal typing in Java

```
1 class Employee {
2   public float salary;
3 }
4
5 class Programmer extends Employee {
6   public float bonus;
```

```
7 }
8
9 class SeniorDeveloper extends Programmer { }
```

### Structural Type Systems

Structural type systems compare types by their structure. This means, if two differently named types have the same properties and methods, then they are the same type. An example of this would be OCaml, with its object subsystem being typed this way. Classes in OCaml only serve as functions for creating objects. In example 23 there is a function that requires a function "speak" returning the type "string". Both the "dog" object as well as the "cat" object fulfill this condition, therefore both are treated equal. Most importantly, these compatibility checks happen at compile time, as OCaml is a static language. Structural typing allows for a lot of flexibility as it promotes code reuse. Furthermore it avoids explicit inheritance hierarchies.

Listing 23: Example of structural typing in Ocaml

```
1   let make_speak (obj : < speak : string >) =
2   obj#speak
3
4   let dog = object
5   method speak = "Woof!"
6   end
7
8   let cat = object
9   method speak = "Meow!"
10  end
11
12  let () =
13  print_endline (make_speak dog);
14  print_endline (make_speak cat);
```

### Duck Typed Systems - Duck Typing

Duck Typing is the usage of a structural type system in dynamic languages. It is the practical application of the "Duck Test", therefore if it quacks like a duck, and walks like a duck, then it must be a duck. In programming languages this means that if an object has all methods and properties required by a type, then it is of that type. The most prominent language utilizing Duck Typing is TypeScript. As can be seen in listing 24, the duck and the person have the same methods and properties, henceforth they are of the same type. The dog object on the other hand does not implement the

"quack" function, which equates to not being a duck. Duck typing simplifies the code by removing type constraints, while still encouraging polymorphism without complex inheritance.

Listing 24: Example of duck typing in TypeScript

```
interface Duck {
  quack(): void;
}

const duck: Duck = {
  quack: function () {
    console.log("Quack!");
  }
};

const person: Duck = {
  quack: function () {
    console.log("I'm a person but I can quack!");
  }
};

const dog: Duck = {
  bark: function () {
    console.log("Woof!");
  }
}; // <- causes an error in the static type checker
```

Given that duck typing allows dynamic data to be easily checked and assigned to any interface, Kipper adopts a similar system to that of TypeScript but introduces notable differences in how interfaces behave and how dynamic data is handled. For instance, casting an "any" object to an interface in Kipper will result in a runtime error if the object does not possess all the required members. In contrast, TypeScript permits such an operation without performing any type checks at runtime.

### 4.6.3 Runtime Base Type

In practice, all user-defined and built-in types inherit from a basic "KipperType" class in the runtime environment. This class is a simple blueprint of what a type could do and what forms a type may take on. A simple version of such a class can be seen in listing 25.

Listing 25: The structure of a runtime type

```
  class KipperType {
    constructor(name, fields, methods, baseType =
      undefined, customComparer = undefined) {
```

```
3         this.name = name;
4         this.fields = fields;
5         this.methods = methods;
6         this.baseType = baseType;
7         this.customComparer = customComparer;
8       }
9
10      accepts(obj) {
11        if (this === obj) return true;
12        return obj instanceof KipperType &&
              this.customComparer ? this.customComparer(this,
              obj) : false;
13      }
14    }
```

As already mentioned types primarily rely on identifier checks to differentiate themselves from other types. Given though that there are slight differences in how types operate, they generally define themselves with what they are compatible using a comparator function. This comparator is already predefined for all built-ins in the runtime library and any user structures build on top of the existing rules established in the library.

Type "any" is an exception and is the only type that accepts any value you provide. However, assigning "any" to anything other than "any" is forbidden and it is necessary to cast it to a different type in order to use the stored value. By design "any" is as useless as possible, in order to force the developer into typechecking it.

Furthermore, classes are also exempt from this comparator behaviour, as classes behave like a value during runtime and provide a prototype which can simply be used to check if an object is an instance of that class.

### 4.6.4 Runtime Built-in Types

Built-in runtime types serve as the foundation of the type system and make up the parts of more complex constructs like interfaces. Built-in runtime types are compared at runtime by comparing their references, as they are uniquely defined at the start of the output code and available in the global scope. The implementations of such structures can be seen in the listing 26 down below.

Listing 26: Examples for the built-in runtime types

```
1   const __type_any =
2   new KipperType("any", undefined, undefined);
3
4   const __type_undefined =
```

```
5    new KipperType("undefined", undefined, undefined,
        undefined, (a, b) => a.name === b.name);
6
7    const __type_str =
8    new KipperType("str", undefined, undefined, undefined,
        (a, b) => a.name === b.name);
```

In addition to the core primitive types—such as "bool", "str", "num", and others—there are built-in implementations for generic types, including "Array<T>" and "Func<T..., R>". These additionally define their generic parameters which generally default to a standard "any" type as can be seen in listing 27.

Listing 27: Generic built-in types

```
1 const __type_Array = new KipperGenericType("Array",
     undefined, undefined, {T: __type_any});
2 const __type_Func = new KipperGenericType("Func",
     undefined, undefined, {T: [], R: __type_any});
```

As can be seen in listing 27, generic types are implemented using a special "KipperGenericType" class. This class, shown in listing 28, extends the "KipperType" and includes an additional field for generic arguments. Most importantly, it includes the method "changeGenericTypeArguments", which allows for modifying a type's generic arguments at runtime. It is used in lambda and array definitions, where the built-in generic runtime type is used and then modified to represent the specified generic parameters. When for example an array is initialized, it first gets assigned the default "Array<any>" runtime type, which is then modified by the "changeGenericTypeArguments" method to create the required type, such as "Array<num>". Arrays for example use the specified type for their elements, whilst functions require a return type as well as an array of argument types. The "Func<T..., R>" type on the other hand is used by lambda definitions, which are user-defined functions with a specific return type and arguments without a name.

Listing 28: Generic Kipper Type

```
1 class KipperGenericType extends KipperType {
2   constructor(name, fields, methods, genericArgs,
       baseType = null) {
3     super(name, fields, methods, baseType);
4     this.genericArgs = genericArgs;
5   }
6   isCompatibleWith(obj) {
7     return this.name === obj.name;
8   }
9   changeGenericTypeArguments(genericArgs) {
```

```
10      return new KipperGenericType(
11      this.name,
12      this.fields,
13      this.methods,
14      genericArgs,
15      this.baseType
16      );
17    }
18  }
```

### 4.6.5 Runtime Errors

Other built-ins include error classes, which are used in the error handling system to represent runtime errors caused by invalid user operations. The base "KipperError" type has a name property and extends the target language's error type as can be seen in listing 29. Additional error types inherit this base type and extend it with additional error information. For instance, the "KipperNotImplementedError" is used whenever a feature that is not yet implemented is used by the developer.

Listing 29: Kipper error types

```
1  class KipperError extends Error {
2    constructor(msg) {
3      super(msg);
4      this.name = "KipError";
5    }
6  }
7  class KipperNotImplementedError extends KipperError {
8    constructor(msg) {
9      super(msg);
10     this.name = "KipNotImplementedError";
11   }
12 }
```

### 4.6.6 Runtime Generation for Interfaces

Unlike TypeScript, in Kipper all interfaces possess a runtime counterpart, which stores all the required information to verify type compatibility during runtime. This process is managed by the Kipper code generator, which adds custom type instances to the compiled code that represent the structures of the user-defined interfaces with all its methods and properties including their respective types.

Now take for example the given interfaces seen in listing 30.

Listing 30: Example interfaces in the Kipper language

```
interface Car {
  brand: str;
  honk(volume: num): void;
  year: num;
}

interface Person {
  name: str;
  age: num;
  car: Car;
}
```

At compile time, the generator function iterates over the interface's members and differentiates between properties and methods. The function keeps separate lists of already generated runtime representations for properties and methods.

If it detects a property, the type and semantic data of the given property is extracted. When the property's type is a built-in type, the respective runtime type already provided by the Kipper runtime library is used. If not, we can assume the property's type is a reference to another type structure, which will be simply referenced in our new type structure. This data is stored in an instance of "___kipper.Property", which is finally added to the list of properties in the interface.

In case a method is detected, the generator function fetches the return type and the method's name. If the method has any arguments, the name and type of each argument also gets evaluated and then included in the definition of the "___kipper.Method". After that, it gets added to the interface as well and is stored in its own separate method list.

If we translate the interfaces shown above in listing 30 it would look similar to that in listing 31.

Listing 31: The runtime representation of the previous interfaces

```
const __intf_Car = new __kipper.Type(
  "Car",
  [
    new __kipper.Property("brand",
       __kipper.builtIn.str),
    new __kipper.Property("year",
       __kipper.builtIn.num),
  ],
  [
    new __kipper.Method("honk", __kipper.builtIn.void,
      [
        new __kipper.Property("volume",
           __kipper.builtIn.num),
```

```
11            ]
12          ),
13        ]
14    );
15
16    const __intf_Person = new __kipper.Type(
17      "Person",
18      [
19        new __kipper.Property("name",
              __kipper.builtIn.str),
20        new __kipper.Property("age", __kipper.builtIn.num),
21        new __kipper.Property("car", __intf_Car),
22      ],
23      []
24    );
```

As shown in listing 31, the properties and methods of an interface are encapsulated within a "KipperType" instance, identified by the "___intf_" prefix. The code for this runtime interface is included directly in the output file, where it can be accessed by any functionality that requires it. To reference the generated interface, the compiler maintains a symbol table that tracks all defined interfaces. The code generator then inserts runtime references to these interfaces wherever necessary.

Notable usages for runtime typechecking include the "matches" operator 4.6.7 and the "typeof" operator 4.6.8.

### 4.6.7 Matches Operator for Interfaces

The primary feature of the Kipper programming language is its runtime type comparison. There are multiple approaches for comparing objects at runtime. One method is comparison by reference, which is implemented using the "instanceof" operator. This method determines that an object is an instance of a class if there is a reference to that class, leveraging JavaScript's prototype system.

Another approach is comparison by structure, where two objects are considered equal if they share the same structure, meaning they have the same properties and methods. Kipper supports both methods of comparison. Reference-based comparison is implemented via the "instanceof" operator and is exclusively used for class comparisons. Structural comparison, referred to as "matching," is applied to primitives and interfaces. Structural comparisons are implemented using the matches operator as can been seen in listing 31.

Listing 32: The Kipper matches operator

```
interface Y {
  v: bool;
  t(gr: str): num;
}

interface X {
  y: Y;
  z: num;
}

var x: X = {
  y: {
    v: true,
    t: (gr: str): num -> {
      return 0;
    }
  },
  z: 5
};

var res: bool = x matches X; // -> true
```

As can be seen in example 32, the matches operator can compare interfaces by properties and methods. It takes two arguments, an object and a type which it should match. Properties are compared recursively and methods are compared by name, arguments and return type.

Comparison works by iterating over the methods and properties. When iterating over the properties, it checks for the property's name being present in the type it should check against. The order of properties does not matter. When the name is found, it checks for type equality. This checking is done using the aforementioned runtime types and nominal type comparison. In case a non-primitive is detected as the properties type, the matches function will be recursively executed on non-primitives.

This property match algorithm is implemented as can seen in listing 33.

Listing 33: Matches operator property comparison

```
for (const field of pattern.fields) {
  const fieldName = field.name;
  const fieldType = field.type;

  if (!(fieldName in value)) {
    return false;
  }

  const fieldValue = value[fieldName];
  const isSameType = __kipper.typeOf(fieldValue) ===
     field.type;
```

```
11
12    if (primTypes.includes(field.type.name) &&
          !isSameType) {
13      return false;
14    }
15
16    if (!primTypes.includes(fieldType.name)) {
17      if (!__kipper.matches(fieldValue, fieldType)) {
18        return false;
19      }
20    }
21  }
```

After checking the properties, the matches expression iterates over the methods. It first searches for the method name in the target type. If found, it compares the return type. Then each argument is compared by name. As the methods signatures need to be exactly the same, the amount of parameters is compared as well.

Listing 34: Matches operator method comparison

```
1  for (const field of pattern.methods) {
2    const fieldName = field.name;
3    const fieldReturnType = field.returnType;
4    const parameters = field.parameters;
5
6    if (!(fieldName in value)) {
7      return false;
8    }
9
10   const fieldValue = value[fieldName];
11   const isSameType = fieldReturnType ===
        fieldValue.__kipType.genericArgs.R;
12
13   if (!isSameType) {
14     return false;
15   }
16
17   const methodParameters =
        fieldValue.__kipType.genericArgs.T;
18
19   if (parameters.length !== methodParameters.length) {
20     return false;
21   }
22
23   let count = 0;
24   for (let param of parameters) {
25     if (param.type.name !==
          methodParameters[count].name) {
26       return false;
27     }
28     count++;
```

```
29     }
30 }
```

When none of these condition is false, the input object matches the input type and they can be seen as compatible.

### 4.6.8 Typeof Operator

In the Kipper Programming Language, the ”typeof” operator is used to get the type of an object at runtime. This operator can be used to check if a variable or expression is of a particular type, such as a string, number, boolean, etc. Most commonly, it is used to check for null and undefined objects, to avoid type errors when an object is of unknown type. The returned type object can be compared by reference to check for type equality. As can bee seen in example 35, the parantheses are optional. We decided to allow both syntax styles, due to our goal of being similar to TypeScript and JavaScript, which both implement it the same way.

Listing 35: Typeof operator

```
1 typeof 49; // "__kipper.builtIn.num"
2 typeof("Hello, World!"); // "__kipper.builtIn.str"
```

The ”typeof” operator in Kipper mirrors the functionality of TypeScript and JavaScript, but with enhancements tailored to Kipper’s type system. Unlike JavaScript, where the ”typeof null” returns ”object” due to historical reasons, Kipper correctly identifies ”null” as ”___kipper.builtIn.null”.

At runtime, the provided object is checked for it’s type using the target languages type features. A part of this process can be seen in example 36. The primitive types return their respective ”KipperRuntimeType”. Objects are a special case, as they can either be null, an array, a class or an object, for example one that implements an interface.

Listing 36: Typeof implementation

```
1 typeOf: (value) => {
2     const prim = typeof value;
3     switch (prim) {
4         case 'undefined':
5             return __kipper.builtIn.undefined;
6         case 'string':
7         return __kipper.builtIn.str;
8         ...
9         case 'object': {
```

```
10              if (value === null) return
                    __kipper.builtIn.null;
11              if (Array.isArray(value)) {
12                  return '__kipType' in value ?
                        value.__kipType :
                        __kipper.builtIn.Array;
13              }
14              const prot = Object.getPrototypeOf(value);
15              if (prot && prot.constructor !== Object) {
16                  return prot.constructor;
17              }
18              return __kipper.builtIn.obj;
19          }
20      }
21  }
```

Although linguistically quite similar, the ”typeof” operator in the type declaration of a variable works fundamentally different 37. It is called ”TypeOfTypeSpecifier” and it evaluates the type of a variable at compile time.

Listing 37: TypeOfTypeSpecifier

```
1  var t: num = 3;
2  var count: typeof(t) = 4;
```

# 5 Compiler Reference

## 5.1 Compiler API

## 5.2 Target API

## 5.3 Shell CLI

# 6 Demo & Showcase

## 6.1 Working example in the web

## 6.2 Working example using Node.js

# 7 Conclusion & Future

# Acronyms

| | |
|---|---|
| AST | Abstract syntax tree |
| GCC | Gnu compiler collection |
| IR | Intermediate representation |

# Glossary

| | |
|---|---|
| abstract syntax tree | A hierarchical, tree-like representation of the abstract syntactic structure of source code. Each node corresponds to a construct in the code, such as expressions or statements, providing a simplified view of the code's logical structure. Essential in compilers for tasks such as semantic analysis and output generation. |
| Antlr4 | The fourth version of the ANTLR project, which enables the generation of a lexer, parser and related tools through the use of a grammar file that defines the language's structure. See [**?**] for more information. |
| gnu compiler collection | GCC is an open-source compiler system that supports languages like C, C++, and Fortran. It converts source code into machine code or intermediate code, enabling program execution across different platforms. It is widely used in software development for its efficiency and portability. |
| intermediate representation | In compiler design, the Intermediate Representation (IR) is an abstract, machine-independent code form used between the source and target code. It simplifies compilation, supports optimizations, and enables portability across architectures. IR can be linear (e.g., SSA) or tree-like (e.g., AST) and is central to modern compiler pipelines. |

| | |
|---|---|
| transpilation | Act of compiling high-level language code to high-level code of another language. This term is mostly used in context of JavaScript and its subsidary languages building on top of the language. |

# List of Figures

1 The design of the GCC compiler . . . . . 18

# List of Tables

# List of Source Code Snippets


1 Unchecked variable assignments due to missing type definitions . . . . . 3
2 Broad ability to perform "invalid" operations despite clear error case . . 3
3 Accessing a missing property returns undefined which later causes an error 4
4 Misaligned function arguments going unnoticed . . . . . 4
5 Unchecked compile-time casts in TypeScript . . . . . 6
6 Ambiguous dynamic data in TypeScript . . . . . 7
7 Holding Values of different Types during Runtime . . . . . 15
8 Statically Typed Language TypeScript . . . . . 15
9 Automatic type conversion in JavaScript . . . . . 15
10 typeof null return "object" in JavaScript . . . . . 16
11 typeof null return "object" in JavaScript . . . . . 17
12 The translation algorithm . . . . . 20
13 The code generation function of a instanceOf expression . . . . . 20
14 Three-address code . . . . . 21
15 Static single-assignment form . . . . . 22
16 The Slice Operation . . . . . 24
17 Slice in the JavaScript BuiltInGenerator . . . . . 24
18 Slice in the target language . . . . . 24
19 Global Scope Logic . . . . . 25
20 Reserved Keywords in TS and JS . . . . . 27
21 Difference in code generator functions . . . . . 27
22 Example of nominal typing in Java . . . . . 29
23 Example of structural typing in Ocaml . . . . . 30
24 Example of duck typing in TypeScript . . . . . 31
25 The structure of a runtime type . . . . . 31
26 Examples for the built-in runtime types . . . . . 32
27 Generic built-in types . . . . . 33
28 Generic Kipper Type . . . . . 33
29 Kipper error types . . . . . 34
30 Example interfaces in the Kipper language . . . . . 34
31 The runtime representation of the previous interfaces . . . . . 35
32 The Kipper matches operator . . . . . 36
33 Matches operator property comparison . . . . . 37
34 Matches operator method comparison . . . . . 38
35 Typeof operator . . . . . 39
36 Typeof implementation . . . . . 39
37 TypeOfTypeSpecifier . . . . . 40

# Appendix