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 Examples of runtime type checking in other languages 8

2.2.1 Case study: Java 8

2.2.2 Case study: Rust 10

2.3 Tackling the issue at its core 13

3 Technology **14**

3.1 Preface & Background 14

3.2 Development Language 14

3.2.1 Selection criteria and weighing the options 15

3.2.2 Option - C++ 15

3.2.3 Option - Java 15

3.2.4 Option - TypeScript 15

3.2.5 Result 15

3.3 Parser & Lexer Generator 15

3.3.1 Selection criteria and weighing the options 15

3.3.2 Option - Antlr4 15

3.3.3 Option - Coco 15

3.3.4 Result 15

4 Implementation **16**

4.1 Compiler 16

4.1.1 Stages of compilation 16

4.2 Lexing & Parsing . . . 18
4.2.1 Syntax definition . . . 19
4.2.2 Lexical analysis . . . 21
4.2.3 Syntactic analysis . . . 24
4.2.4 AST (Abstract Syntax Tree) . . . 25
4.3 Semantic Analysis . . . 27
4.4 Type Analysis . . . 27
4.5 Error recovery . . . 27
4.5.1 Error recovery algorithm . . . 28
4.5.2 Special case: Syntax errors . . . 28
4.6 Type System . . . 28
4.6.1 Intended Purpose & Concept . . . 28
4.6.2 Existing Foundation & Environment . . . 28
4.6.3 Translating the foundation to Kipper . . . 32
4.6.4 Drawing comparisons to TypeScript . . . 32
4.6.5 Kipper Primitives . . . 32
4.6.6 Kipper Generics . . . 32
4.6.7 Kipper Interfaces & Duck-Typing . . . 32
4.6.8 Kipper Classes & Prototyping . . . 32
4.7 Output Generation . . . 32
4.7.1 Introduction . . . 32
4.7.2 Role of the AST in the output generation . . . 33
4.7.3 Algorithm . . . 33
4.7.4 Algorithms used for Output Generation . . . 35
4.7.5 Requirements . . . 37
4.7.6 Differences between the Target Languages . . . 41
4.7.7 Stylistic Choices . . . 42
4.8 Integrated Runtime . . . 44
4.8.1 Runtime Type implementations in other languages . . . 44
4.8.2 Runtime Type Concept in Kipper . . . 47
4.8.3 Base Type for the Kipper Runtime . . . 48
4.8.4 Built-in Types for the Kipper Runtime . . . 49
4.8.5 Runtime Errors . . . 50
4.8.6 Runtime Generation for Interfaces . . . 51
4.8.7 Matches Operator for Interfaces . . . 53

4.8.8 Typeof Operator 55

5 Compiler Reference **57**

5.1 Compiler API 57

5.2 Target API 57

5.3 Shell CLI 57

6 Demo & Showcase **58**

6.1 Working example in the web 58

6.2 Working example using Node.js 58

7 Conclusion & Future **59**

7.1 Potential Features 59

7.1.1 WebAssembly Support 59

7.1.2 IDE Support 60

7.2 Integration with other languages 60

7.3 Project Result 60

Acronyms **VI**

Glossary **VII**

Bibliography **IX**

List of Figures **X**

List of Tables **XI**

List of Source Code Snippets **XII**

Appendix **XIII**

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 [1]. 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.

```
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
```

Listing 1: Unchecked variable assignments due to missing type definitions

- 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.

```
let discountRate = cart.appliedDiscount; // Assuming it is actually "applied_discount" not "appliedDiscount" so it returns "undefined"

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

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

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.

```
1  // Assuming this is some kind of API response that
       incorrectly returned some data
2  let user = {
3    name: "Alice"
4  };
5
6  ...
7
8  const userAddress =  user.address; // Undefined
       property ("address" does not exist), returns
       simply "undefined"
9
10 ...
11
12 let shippingCost = getShippingCost(address.city); //
       -> TypeError: Cannot read property "city" of
       undefined
```

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

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

```
1  function greet(name) {
2    console.log("Hello, " + name + "!");
3  }
4
5  ...
6
7  let user = { firstName: "Alice" };
8
9  greet(user); // Unintended behavior: "Hello, [object
       Object]!"
```

Listing 4: Misaligned function arguments going unnoticed

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 can be called "trust-based casts". This means, that TypeScript trusts the developer to check for type compatibility. The language therefore allows any type to be cast to any other type.

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 is not 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.

```
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!
     "c1.superField" does not actually exist
23
24 const c2 = <LowerClass>new MiddleClass(); // Unchecked
     cast
25 console.log(c2.classField.superField.name); // Runtime
     Error! "c2.classField.superField" does not actually
     exist
```

Listing 5: Unchecked compile-time casts in TypeScript

In listing 5, 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 are 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 not access property "name", c1.superField is undefined). Furthermore, given that JavaScript only reports on such errors when a property of 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

A 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.

An example of such a situation is given in listing 6:

```
1 interface Data {
2   x: number;
3   y: string;
```

```
4     z: {
5       z1: boolean;
6     }
7   }
8
9   function receiveUserReq(): object {
10    // ...
11    return {
12      x: "1",
13      y: "2",
14      z: true
15    }
16  }
17
18  var data = <Data>receiveUserReq(); // Unsafe casting
        with unknown data
19  console.log(data.z.z1); // No Runtime Error! But returns
        "undefined"
```

Listing 6: Ambiguous dynamic data in TypeScript

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, 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" [2] and "joi" [3]. 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, 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 Examples of runtime type checking in other languages

2.2.1 Case study: Java

Java is a statically typed object-oriented programming language that runs on the Java Virtual Machine (JVM). It is widely used in enterprise applications, Android

development, and large-scale systems due to its robust type system and memory management capabilities. Unlike JavaScript and TypeScript, which Kipper transpiles to, Java enforces type safety at both compile-time and runtime. This enforcement ensures that type errors are caught early and that unexpected behavior due to type inconsistencies is minimized.

One of Java's key runtime type features is its use of reflection, which allows a program to inspect and modify its own structure during execution. Reflection enables Java to perform dynamic type checking, instantiate objects of arbitrary classes, and call methods dynamically based on type information. This mechanism is crucial for frameworks such as Spring and Hibernate, which rely on runtime type inspection for dependency injection and object-relational mapping.

Runtime Type Checking

Java enforces strict runtime type checking to ensure the correctness of operations involving polymorphism and casting. The JVM performs type checks before executing certain operations, preventing unsafe conversions. An example of this is listing 7. In this example, the `instanceof` operator ensures that the object is of type `Dog` before performing a downcast. This prevents a `ClassCastException`, which would otherwise occur if the object were of an incompatible type.

```java
class Animal {
  void makeSound() {
    System.out.println("Some sound");
  }
}

class Dog extends Animal {
  void makeSound() {
    System.out.println("Bark");
  }
}

public class TypeCheckExample {
  public static void main(String[] args) {
    Animal myAnimal = new Dog(); // Upcasting
    if (myAnimal instanceof Dog) {
      Dog myDog = (Dog) myAnimal; // Safe downcasting
      myDog.makeSound();
    }
  }
}
```

Listing 7: Runtime typechecks in Java

Reflection

Java provides a rich set of reflection APIs that allow the program to examine and manipulate class structures at runtime. The `java.lang.reflect` package enables dynamic method invocation, field modification, and even dynamic class loading. In listing 8, the Java reflection API allows to retrieve method names and invoke them dynamically. This is particularly useful in frameworks and tools that require runtime type information, such as dependency injection frameworks and dynamic proxies.

```
1  import java.lang.reflect.Method;
2
3  class ExampleClass {
4    public void sayHello() {
5      System.out.println("Hello, world!");
6    }
7  }
8
9  public class ReflectionExample {
10   public static void main(String[] args) throws
         Exception {
11     Class<?> c = ExampleClass.class;
12     Object obj =
           c.getDeclaredConstructor().newInstance();
13
14     for (Method method : c.getDeclaredMethods()) {
15       System.out.println("Method: " + method.getName());
16       method.invoke(obj);
17     }
18   }
19 }
```

Listing 8: Reflection in Java

Java's runtime type system offers strong guarantees by enforcing type safety through reflection, dynamic type checking, and explicit exception handling. In contrast, JavaScript's dynamic nature makes it more flexible but also prone to type-related errors.

2.2.2 Case study: Rust

Rust is a statically typed systems programming language designed for performance and safety. Unlike Java, which relies on a garbage collector and runtime type checks, Rust enforces strict compile-time guarantees to ensure memory safety and type correctness. This results in minimal runtime overhead, making Rust an appealing choice for high-

performance applications. However, Rust does provide mechanisms for runtime type inspection when necessary, such as trait objects and dynamic dispatch.

Rust's type system is centered around ownership, borrowing, and lifetimes, ensuring memory safety at compile time without the need for a garbage collector. Despite its strong static typing, Rust allows for limited runtime type checking through the Any trait and dynamic trait objects. Unlike JavaScript, Rust minimizes dynamic type handling in favor of compile-time guarantees.

Runtime Type Checking

Rust does not depend on runtime type checking as Java or JavaScript do. Instead, it enforces type safety at compile time. Nevertheless, Rust provides runtime tools for handling dynamically typed values, notably the `Any` trait from the `std::any` module. This trait enables type inspection and downcasting at runtime, functioning similarly to Java's `instanceof` operator.

In listing 9, the `Any` trait is employed to examine and downcast a trait object to a specific concrete type. The `is::()` method checks whether a value belongs to a certain type, while `downcast_ref::()` provides a safe way to obtain a reference to the underlying type.

```rust
use std::any::Any;

fn check_type(value: &dyn Any) {
  if value.is::<i32>() {
    println!("Value is an i32!");
  } else if value.is::<String>() {
    println!("Value is a String!");
  } else {
    println!("Unknown type");
  }
}

fn main() {
  let num = 42;
  let text = "Hello".to_string();

  check_type(&num);
  check_type(&text);
}
```

Listing 9: Runtime type checking in Rust

While the `Any` trait enables some level of runtime type checking, its use is generally discouraged in idiomatic Rust. The language prefers static polymorphism through generics and trait bounds, ensuring type correctness at compile time.

Trait Objects and Dynamic Dispatch

Rust's trait system allows for polymorphism without runtime type checking. However, when dynamic dispatch is required, Rust provides trait objects, which enable runtime method resolution similar to Java's virtual method table (vtable) approach. Unlike Java's reflection-based system, Rust's trait objects still enforce strict type constraints at compile time while allowing method calls to be dynamically resolved at runtime.

In listing 10, a trait `Animal` is defined, and trait objects (`&dyn Animal`) are used to allow polymorphism. This enables different implementations to be stored in the same collection and accessed dynamically.

```
1  trait Animal {
2    fn make_sound(&self);
3  }
4
5  struct Dog;
6  impl Animal for Dog {
7    fn make_sound(&self) {
8      println!("Bark");
9    }
10 }
11
12 struct Cat;
13 impl Animal for Cat {
14   fn make_sound(&self) {
15     println!("Meow");
16   }
17 }
18
19 fn main() {
20   let animals: Vec<&dyn Animal> = vec![&Dog, &Cat];
21   for animal in animals {
22     animal.make_sound(); // Dynamically dispatched
23   }
24 }
```

Listing 10: Trait objects and dynamic dispatch in Rust

This mechanism provides dynamic polymorphism without exposing the entire type system to runtime modification as in Java's reflection API. Unlike Java's reflection,

Rust does not allow modifying class structures at runtime, maintaining stricter safety guarantees.

Rust's runtime type system differs significantly from Java's and JavaScript's approaches. Java relies on runtime reflection and dynamic type checking to handle polymorphism and introspection, whereas Rust enforces stricter compile-time type safety and only allows limited runtime type inspection through `Any` and dynamic trait objects. JavaScript, on the other hand, is dynamically typed, meaning type errors may only surface at runtime, whereas Rust eliminates most type-related errors before execution.

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 "Do not 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.

The above approach is exactly the aim of Kipper—a language that implements a custom runtime type system, which is later transpiled into JavaScript or TypeScript. By incorporating an additional runtime and non-standard syntax, Kipper enables runtime type checks. 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 connecting 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 the language is predefined and implemented using the lexer and parser generated by the Antlr4 parser generator.

An interesting aspect of the Kipper compiler is its largely modular design, which allows various components to be replaced or extended as needed. This modularity primarily serves to enable the compiler's structure to adapt to future changes in the language and its target output environment. Given the rapid evolution of web technologies and the frequent addition of new functionality, it is essential to ensure that the compiler remains current. Additionally, this design allows users to create plugins or extensions for the compiler to support custom functionality that may not be natively available.

Furthermore, as discussed in greater detail in section 4.7, the compiler is capable of targeting more than one output format. Specifically, it supports both standard JavaScript, adhering to the ES6/ES2015 specification, and standard TypeScript as defined by Microsoft. Similar to other compiler systems, users can configure the compiler according to their preferences and specify the desired target language.

4.1.1 Stages of compilation

The Kipper compiler processes the program through multiple phases, each building upon the previous one to progressively enrich the semantic and logical representation of the program. The phases and their corresponding responsible components are as follows:

Lexical Analysis

- Detailed explanation in section 4.2
- Performed by: Antlr4 Kipper Lexer
- This phase tokenizes the source code into a stream of lexemes, identifying the basic units of syntax.

Syntax Analysis - Parsing

- Detailed explanation in section 4.2
- Performed by: Antlr4 Kipper Parser
- In this phase, the lexed tokens are organized into a parse tree based on the grammar rules of the Kipper language.

Parse Tree Translation to AST

- Detailed explanation in section 4.2.4
- Performed by: Kipper Core Compiler
- Converts the parse tree into an Abstract Syntax Tree (AST), a more abstract and language-independent representation of the code.

Semantic Analysis

- Detailed explanation in section 4.3
- Performed by: Kipper Core Compiler
- This phase is split into three separate steps:
    1. Primary Semantic Analysis: Validates language semantics such as variable declarations and scope resolution.
    2. Preliminary Type Analysis: This phase performs initial type validations and checks. It includes tasks such as loading type definitions that might be referenced elsewhere in the program. These types need to be pre-loaded to ensure they are available and correctly resolved during subsequent stages of compilation.
    3. Primary Type Analysis: Conducts in-depth type validation and resolves type-related issues for the given statements and expressions.

Target-Specific Requirement Checking

- Detailed explanation in section 4.2.4
- Performed by: Target Semantic Analyser
- Ensures compliance with the specific requirements of the target platform (JavaScript or TypeScript).

Optimization

- Detailed explanation in section 4.2.4
- Performed by: Kipper Core Compiler
- Performs code optimizations, currently focused on treeshaking to eliminate unused code.

Target-Specific Translation

- Detailed explanation in section 4.7
- Performed by: Target Translator
- Translates the optimized AST into the desired target language (JavaScript or TypeScript).

Each phase of the compiler is executed sequentially, with each step requiring the successful completion of the previous one. This ensures that each module of the compiler can safely rely on the correctness and completeness of the information provided by earlier steps. However, this approach limits the compiler's ability to recover from errors or detect all faults in a single execution. As a trade-off, this method simplifies the implementation and reduces overall complexity. Unlike TypeScript, for example, this means that the compiler cannot ignore certain errors and work around them.

4.2 Lexing & Parsing

The first step in the compilation process is the lexing and parsing of the input program. This involves the tokenization of the program source code, where individual strings are classified into predefined categories, followed by syntactical analysis that organizes these tokens into statements, expressions, and declarations.

In the Kipper compiler, these two steps are carried out by the Kipper Lexer and Parser generated by Antlr4, rather than being directly implemented within the compiler itself. These Antlr4-generated components are constructed based on the predefined token and syntax rules specific to the Kipper language.

4.2.1 Syntax definition

The primary utility provided by Antlr4 lies in its ability to generate lexers and parsers automatically from an input file written in the Antlr4-specific ".g4" context-free grammar format. For Kipper, the lexer and parser each have distinct definitions, specifying the individual tokens and the rules for constructing the syntax tree that organizes and groups these tokens.

These definitions are created in a manner similar to other syntactical specification methods, such as BNF (Backus-Naur Form), commonly used for context-free formal grammars. However, unlike BNF, Antlr4 grammars have the added capability to include programmatic conditions and invocations, enabling more dynamic and adaptable grammar definitions.

Lexer Grammar Definition

In the case of the Kipper Lexer, it uses its own token definition file, which defines how characters are grouped into tokens. This file specifies the rules for identifying individual tokens while ignoring special characters that are not required for syntax analysis. Additionally, the lexer separates the matched tokens into various channels, allowing for more efficient handling and categorization during subsequent parsing steps. (See 4.2.2 for more detail)

For example, in Kipper, the lexer grammar includes constructs for identifying both comments and language-specific keywords. These definitions can be seen showcased below in listing 11.

```
1   BlockComment : '/' .? '*/' -> channel(COMMENT) ;
2
3   LineComment : '//' CommentContent -> channel(COMMENT) ;
4
5   Pragma : '#pragma' CommentContent -> channel(PRAGMA) ;
6
7   InstanceOf : 'instanceof';
8
9   Const : 'const';
```

```
10
11    Var : 'var';
```

Listing 11: Sample snippet from Kipper Lexer grammar

In the grammar above, comments are directed to a separate channel using the `-> channel` annotation. This helps isolate them from the main parsing flow while still retaining them for potential processing or documentation purposes. Pragmas, which are typically compiler directives, are handled similarly but redirected to a `PRAGMA` channel.

Antlr4 grammars also include support for defining keywords, operators, and contextual language constructs. For instance, `Const` and `Var` are tokenized as reserved keywords, ensuring they are recognized unambiguously during lexical analysis. This explicit tokenization is critical for constructing a clear and predictable syntax tree, which serves as the foundation for the subsequent phases of interpretation and compilation.

Syntactical Grammar Definition

Like the Kipper Lexer, the Kipper Parser is defined by its own syntax definition file. In this file, various rules are grouped into syntax rules that collectively form a hierarchical structure. The parser traverses this structure to determine which syntactical construct is represented by the individual tokens. These syntax rules are organized in a way that allows the parser to construct a parse tree by following specific paths, with each path representing a distinct syntactical structure in the Kipper language.

For example, a grammar rule defining a function declaration in Kipper could be expressed as seen in listing 12:

```
1    functionDeclaration : 'def' declarator '('
        parameterList? ')' '->' typeSpecifierExpression
        compoundStatement? ;
2
3    parameterList : parameterDeclaration (','
        parameterDeclaration)* ;
4
5    parameterDeclaration : declarator ':'
        typeSpecifierExpression ;
```

Listing 12: Function Declaration Grammar

In this grammar, the rule for `functionDeclaration` begins with the keyword `'def'`, followed by a `declarator` (which represents the function name or identifier), and then an

optional `parameterList`. The rule includes a `typeSpecifierExpression`, indicating the return type of the function, and optionally a `compoundStatement`, which represents the function's body.

The `parameterList` rule handles the optional inclusion of one or more parameters, each defined by `parameterDeclaration`. Each `parameterDeclaration` consists of a `declarator` (the parameter's name) followed by a type specification. This structure clearly defines the expected syntax for a function declaration in Kipper, ensuring that the parser can accurately identify and process this construct.

Using these syntax rules, the Kipper parser can build a detailed parse tree, with nodes representing various language constructs, such as function definitions, parameters, and expressions. This hierarchical structure allows for efficient interpretation or compilation of Kipper code.

4.2.2 Lexical analysis

Primary tokenisation

The primary task of the lexical analysis is the tokenization of the individual characters into grouped tokens which represent syntactical elements, such as identifiers, keywords, constant values, etc. Tokens serve as the building blocks for higher-level constructs in the parsing process, providing a simplified and structured representation of the raw source code.

The step of tokenization is fairly straightforward as it simply follows the definitions provided in the grammar file (See 4.2.1) and throws errors in case no associated token definition is found. Each token is assigned a specific type based on the grammar rules, ensuring that the source code adheres to the language's syntactical structure. If a character sequence does not match any defined rule, an error is raised, indicating the presence of invalid syntax.

The lexer operates as a state machine, scanning through the input character stream and categorizing sequences based on patterns defined in the grammar. These patterns may include regular expressions to match identifiers, numerical constants, or specific language keywords. Once tokens are identified they are put into the associated channels as explained in 4.2.2.

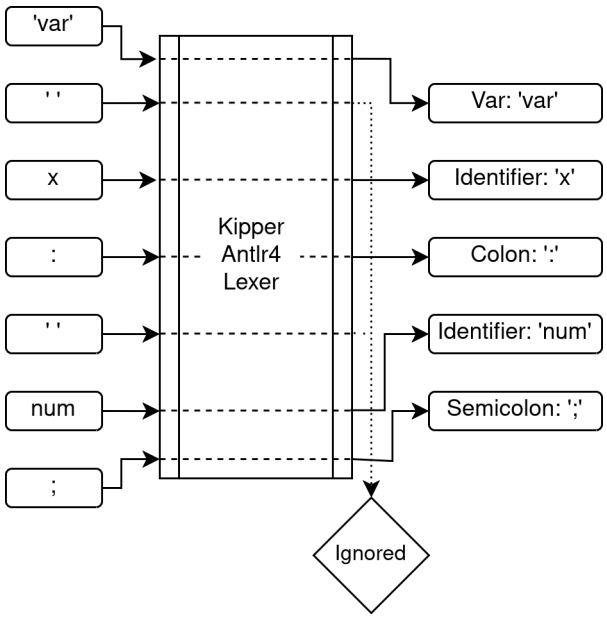


Figure 1: The lexing process which categories the various tokens

Token Channels

Next to the definition of the various tokens the grammar file also specifies what channel each token should be put into. These channels act as a stream of tokens where each stream represents different semantic parts of the program.

The channels which are implemented in the case of Kipper are:

- **Default channel**
- **Comments channel**
- **Pragma channel**
- **Ignored channel**

The **default channel** serves as the primary stream, storing nearly all tokens in the program. It is the main channel used during the parsing step to construct a parse tree of the program.

The remaining channels are special-purpose streams that are excluded during parsing and cater to specific functionalities:

- The **comments channel** is dedicated to storing all comments, which are logically irrelevant to the program and do not require parsing or further processing.
- The **pragma channel** contains compiler pragmas—special instructions to the compiler that are processed independently from the standard syntax rules.

- The **ignored channel** is reserved for special characters that are significant only for token differentiation but have no logical relevance to the program, such as spaces. While spaces are critical for separating tokens, like with `var x` and `varx` they do not contribute to the program's logic and are therefore excluded from parsing.

Nested Sub-Lexing

Besides standard sequential processing of the input, the Kipper Lexer also employs a technique called sub-lexing. Sub-lexing involves branching off the main lexing process and invoking a sub-lexer that operates under its own set of rules and guidelines. This specialized lexer handles specific subsets of tokens that require unique processing rules.

Sub-lexing is crucial for Kipper due to features such as templating, where code fragments are embedded within strings. In such cases, the lexer must correctly differentiate between string elements and code atoms (the inserted snippets inside the string). By using a simple push-and-pop mechanism, the lexers can function similarly to a stack, layering processing contexts on top of one another. Each context processes its corresponding string subset, enabling correct parsing of both regular syntax and embedded code within strings. This modular approach ensures cleaner handling of complex tokenization scenarios and increases the lexer's flexibility.

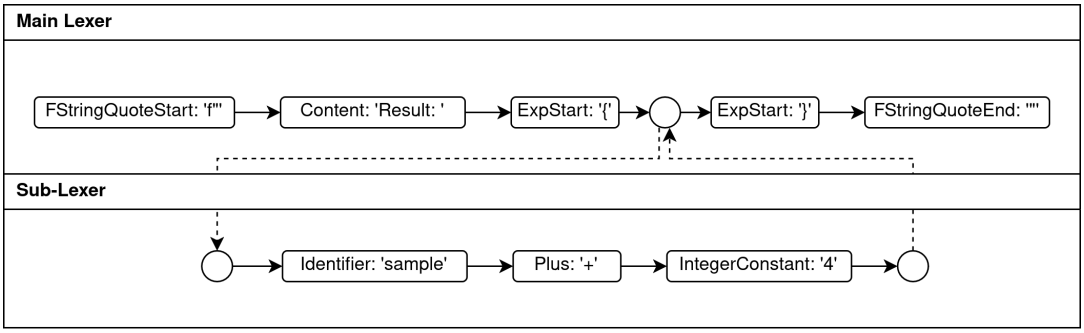


Figure 2: The process of invocating a sub-lexer with the sample input `f"Result: {sample + 4}"` (An example of a template string, or also format string, in the Kipper language), where all content between `{` and `}` is passed onto the sub-lexer.

4.2.3 Syntactic analysis

Primary syntactic analysis of the token stream

With the lexer having already identified all tokens in a given program and ensured that only valid elements are present, the parser proceeds to analyze the program's structure and logic. This step is inherently more complex and often demands a significant amount of processing time. The complexity arises partly from the computational effort required to transform a token stream into a viable syntax tree and partly from the design of Antlr4, which generates detailed context objects for each node in a program and as such requires a lot of memory and processing to call up each context.

The parser's primary task is to verify that the sequence of tokens conforms to the grammatical rules specified by the Kipper grammar. This involves checking whether constructs such as statements, expressions, and control structures are correctly formed.

Building the parse tree

Similar to the hierarchical structure defined by Kipper's grammar rules, the parse tree generated by the Kipper Parser also exhibits a hierarchical organization. Each parse node may have multiple child nodes and is connected to a single parent node, positioned syntactically one level higher within the tree. These nodes can either represent a rule node, such as `expression`, or correspond directly to a simple lexer token.

The construction of a parse tree begins with a designated root node representing an entire file. From this root, branching occurs through intermediate rule nodes that capture various grammatical constructs, such as statements, definitions and expressions. These intermediate nodes eventually lead to leaf nodes, which are simple lexer tokens forming the smallest syntactic components of the program, such as identifiers, keywords, operators, and literals.

This hierarchical structure enables easy traversal and analysis of the program during later stages of compilation or interpretation. For instance, an arithmetic expression in a program, such as `a + b * c`, would form a subtree where the root node corresponds to an expression rule, with child nodes representing the individual terms and operations in the correct precedence order.

A simplified representation of this tree structure is shown in figure 3.

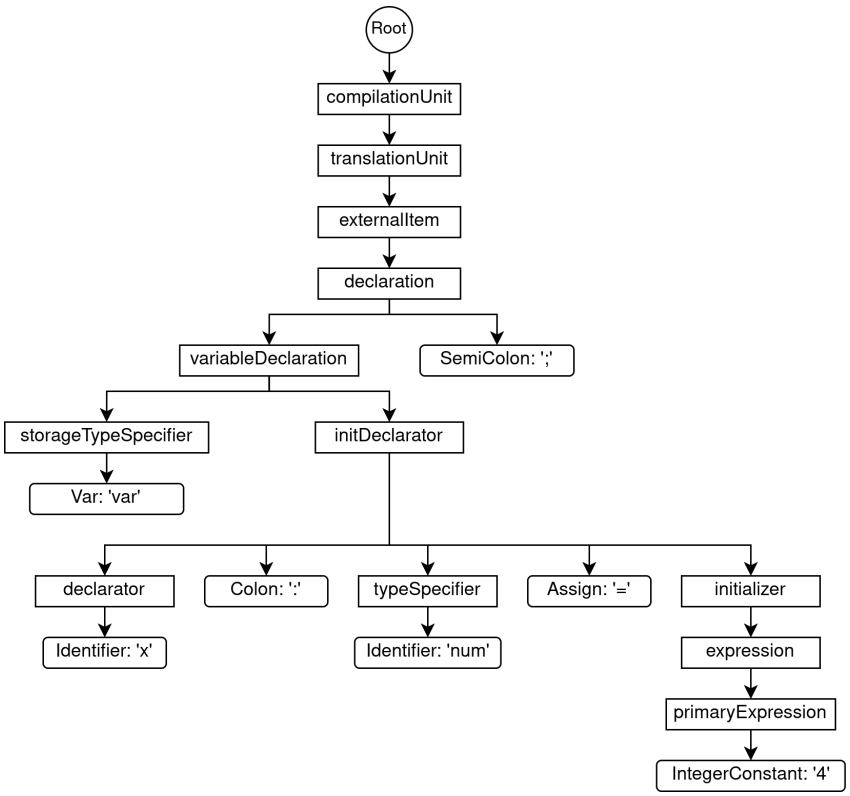


Figure 3: A simplified parse tree representation of the statement `var x: num = 4;`.

Programmatic conditions & context-sensitive rules

In addition to standard context-free rules, there are grammar rules that incorporate programmatic conditions, requiring specific requirements to be met beyond the standard syntactic structure. These rules are inherently context-sensitive, as they cannot be identified without considering the program's position and overall logic.

An example of a context-sensitive rule is the compound statement `{ }`, which groups multiple statements and is typically used as the body of functions and methods. By definition, such statements are generally permitted only as top-level program nodes or as children of other statements, excluding cases where expressions such as lambda expressions specify one as a child. To accommodate lambdas having a structured body, two types of compound statements are defined. While they serve the same practical purpose, they are logically different due to their different contexts of usage: one as a child of another statement and another as a child of a lambda expression.

4.2.4 AST (Abstract Syntax Tree)

The AST represents a tree which groups together the most logically essential elements of a specific items and disregards all the other items not necessary in further processing. Lexer tokens, such as `:`, `=` or `;` may be important syntactically as indicators for specific

operations and structures, but once a specific parser rule kind has been determined and the meaning can be derived from that alone these tokens are not necessary anymore and can be discarded.

Parse Tree Walking

To transform the parse tree generated by the Kipper Parser, the compiler uses a tree-walking algorithm that systematically traverses the tree by entering and exiting grammar rules. During this process, the algorithm invokes specific handlers when they are defined for particular parse nodes. These handlers are designed to process only the most significant parse nodes, which typically correspond to essential syntactic constructs of the source code.

When a handler is called, it constructs a new AST node and attaches it to the growing AST. This selective processing approach ensures that irrelevant parse tree details, such as redundant intermediate nodes or syntactic artifacts, are automatically excluded from the AST. The resulting AST is a simplified and more abstract representation of the program's structure, capturing only the semantically relevant elements needed for subsequent stages of the compilation process.

The AST generation result of such a tree walk process can be seen in figure 4.

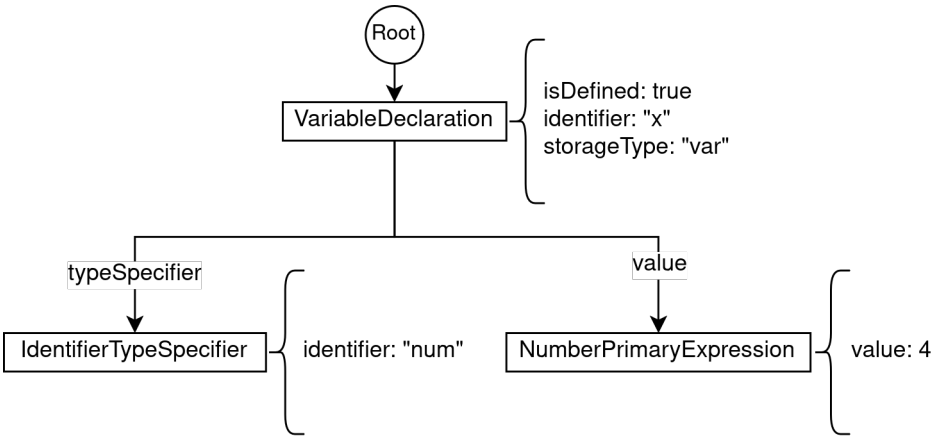


Figure 4: An AST produced by the statement `var x: num = 4;`. The data in the brackets is in reality only defined and error checked during semantic analysis (See 4.3), but for the sake of clarity it is already provided here as to not cause confusion due to the missing metadata.

Utility provided by the AST

In addition to providing a simplified abstraction of the original parse tree, individual AST nodes also handle processing for several subsequent stages. Encapsulating these steps

within a single class allows semantic analysis and type analysis to be efficiently layered and properly structured. This design is a core aspect of the compiler, as it centralizes data storage and ensures that the results of one processing stage directly influence subsequent stages.

For instance, if a node fails to successfully complete semantic analysis, the subsequent stages are automatically skipped. The node and any related parent structures are marked as faulty, allowing the compiler to handle errors gracefully without immediately crashing (See 4.5 for a detailed explanation). Furthermore, the standardized and detailed structure of the AST enables easy integration with other processing steps, as it stores or references all necessary information for specific parts of a program. This is particularly important during output generation (See 4.7), where all existing information is required to accurately generate output code which logically adheres to the original program.

4.3 Semantic Analysis

4.4 Type Analysis

4.5 Error recovery

The functionality of error recovery is integral to most modern compilers, as it allows the compiler to report multiple errors in a single compilation pass. Basic compilers often operate on an immediate fail-safe principle: upon encountering an error, the compilation process halts immediately to prevent the compiler from making incorrect assumptions about the program. While this approach ensures the integrity of the compilation process, it introduces a significant drawback. For large and complex programs, developers may need to repeatedly correct errors and recompile to uncover additional issues, leading to unnecessary delays and inefficiencies during the development process.

Given these issues, Kipper has implements its own error recovery algorithm into the semantic analysis which is able to recover from errors in given contexts and continue operation in following expressions or statements, which aren't associated with the original error.

4.5.1 Error recovery algorithm

4.5.2 Special case: Syntax errors

4.6 Type System

The type system is the primary element of a modern object-oriented programming languages and in the case of Kipper it is the primary area of attention.

4.6.1 Intended Purpose & Concept

The purpose of a type system pre-defines the conditions and expectations of how the type system should act in various cases of ambiguity, type matching and derivative type definitions.

4.6.2 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.

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

Listing 13: Holding Values of different Types during Runtime

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.

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

Listing 14: Statically Typed Language TypeScript

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.

```
1 let quantity = 7; // quantity is an integer
2 let value = "20"; // value is a string
3
4 let total = quantity + value; // JavaScript
    automatically converts quantity to string
5 console.log(total); // Output: "720"
```

Listing 15: Automatic type conversion in JavaScript

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.

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

Listing 16: typeof null return "object" in JavaScript

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.

```
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"
```

Listing 17: typeof null return "object" in JavaScript

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

4.6.3 Translating the foundation to Kipper

4.6.4 Drawing comparisons to TypeScript

4.6.5 Kipper Primitives

4.6.6 Kipper Generics

4.6.7 Kipper Interfaces & Duck-Typing

4.6.8 Kipper Classes & Prototyping

4.7 Output Generation

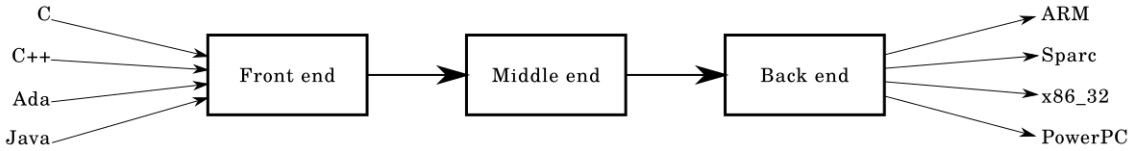
4.7.1 Introduction

The Kipper compiler utilizes a modular architecture, allowing for the definition of custom targets and providing flexibility to accommodate various use cases. This modularity extends beyond basic configuration, enabling developers to specify target-specific features or behaviors. The modular design is achieved by dividing the compiler into two primary components: a frontend and a backend.

For comparison, the GCC compiler achieves modularity by dividing its architecture into three components, as illustrated in Figure 5. The frontend is responsible for verifying syntax and semantics, scanning the input, and performing type checking. It subsequently generates an intermediate representation (IR) of the code.

The middle-end then optimizes this intermediate representation, which is designed to be independent of the CPU target. Examples of middle-end optimizations include dead code elimination and detection of unreachable code. Finally, the backend takes the optimized intermediate code and generates target-dependent code. In the case of GCC, this involves generating assembly code.

As of now, Kipper generates either TypeScript or JavaScript code as its output. The target language is specified in the Kipper CLI utility using the flag –target=js|ts. If this flag is not provided, the default target is JavaScript.



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

Figure 5: The design of the GCC compiler

Currently, Kipper does not include a middle-end component. This decision was made to avoid the additional complexity associated with implementing a full middle-end, as the resulting performance improvements in generated code would be only marginal. Instead, the frontend directly passes the AST (Abstract Syntax Tree) to the backend. While Kipper does perform some post-analysis optimizations, these are presently limited to treeshaking.

4.7.2 Role of the AST in the output generation

Kipper generally uses an AST as the primary representation of the code from the input program. The AST serves as a hierarchical structure that represents the program's source code. The root node of the AST corresponds to the entire program, while each child node represents a specific construct such as a statement, scope, or other language feature.

Each node in the AST contains the semantic data and type-related information of the corresponding statement, as well as a string representation of the original parser node. Additionally, every node includes a kind identification property—a unique, hard-coded number in the compiler—to uniquely identify the type of the node. This property aids in determining the type of construct, such as a declaration, an expression, or a statement.

Every node also maintains a list of child nodes, representing nested or dependent components of the construct. Once the AST is fully constructed, it is wrapped and passed to the code generator for further processing.

4.7.3 Algorithm

Kipper processes the previously generated AST and its nodes in a bottom-to-top manner. This approach ensures that the translation of child nodes is completed before their parent nodes are processed. Consequently, parent nodes can extract necessary

information from their children. This is especially important for complex structures, as they depend on the code generated by their children, embedding it into their own output. The generated code is passed to the parent as an array of tokens—simply output code in text form. This process is entirely safe since the AST has already been validated for syntactical and semantic correctness during earlier phases. Each node is independently responsible for generating its own output code, a design choice that ensures modularity and flexibility.

The output generation in Kipper begins by setting up the target environment and generating any necessary requirements, as detailed in section 4.7.5. Once the setup is complete, the compiler iterates over the child nodes by calling the "translateCtxAndChildren" function. This function recursively traverses the tree, generating code for each child node. Each node returns a string containing its output code, which is then processed by its parent node. The process continues until the root node aggregates all the strings and sends the final merged output to a function that writes the code to a file.

The implementation of this translation algorithm is provided in listing 18.

```
1  public async translate():
       Promise<Array<TranslatedCodeLine>> {
2    // SetUp and WrapUp functions
3    const targetSetUp: TargetSetUpCodeGenerator =
       this.codeGenerator.setUp;
4    const targetWrapUp: TargetWrapUpCodeGenerator =
       this.codeGenerator.wrapUp;
5
6    // Add set up code, and then append all children
7    const { global, local } = await
       this.programCtx.generateRequirements();
8    let genCode: Array<TranslatedCodeLine> = [...(await
       targetSetUp(this.programCtx, global)), ...local];
9    for (let child of this.children) {
10     genCode.push(...(await
         child.translateCtxAndChildren()));
11   }
12
13   // Add wrap up code
14   genCode.push(...(await targetWrapUp(this.programCtx)));
15
16   // Finished code for this Kipper file
17   return genCode;
18 }
```

Listing 18: The translation algorithm

The code generation function of a node takes the node as an argument and gets it's semantic and type-semantic 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 listing 19. In this example, the instanceOf expression gets built.

Clearly visible is the same "translateCtxAndChildren" function as in listing 18. 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.

```
1 instanceOfExpression = async (node:
     InstanceOfExpression): Promise<TranslatedExpression>
     => {
2    const semanticData = node.getSemanticData();
3    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    };
```

Listing 19: The code generation function of a instanceOf expression

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.

4.7.4 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 algorithms 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 [4]. An instruction can have up to three operands, although it may have less. In the listing 20, 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.

```
1  // Problem
2  x = (-b + sqrt(b^2 - 4*a*c)) / (2*a)
3
4  // Solution
5  t1 := b * b
6  t2 := 4 * a
7  t3 := t2 * c
8  t4 := t1 - t3
9  t5 := sqrt(t4)
10 t6 := 0 - b
11 t7 := t5 + t6
12 t8 := 2 * a
13 t9 := t7 / t8
14 x := t9
```

Listing 20: Three-address code

The Static single-assignment form is a different intermediate representation, where each variable gets assigned exactly once [5]. 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 listing 21, 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.

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

```
5
6 // Solution
7 y1 := 1
8 y2 := 2
9 x1 := y2
```

Listing 21: Static single-assignment form

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 [6].

4.7.5 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 22. 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.

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

Listing 22: The Slice Operation

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 listing 23. 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.

```
1 async slice(funcSpec: InternalFunction):
     Promise<Array<TranslatedCodeLine>> {
2   const signature = getJSFunctionSignature(funcSpec);
```

```
3    const objLikeIdentifier = signature.params[0];
4    const startIdentifier = signature.params[1];
5    const endIdentifier = signature.params[2];
6
7    return genJSFunction(
8      signature,
9      `{ return ${objLikeIdentifier} ?
         ${objLikeIdentifier}.slice(${startIdentifier},
         ${endIdentifier}) : ${objLikeIdentifier}; }`,
10   );
11   }
```

Listing 23: Slice in the JavaScript BuiltInGenerator

The generated "slice" function can be found in listing 24.

```
1  slice: function slice<T>(objLike: T, start: number |
     undefined, end: number | undefined): T {
2    return objLike ? objLike.slice(start, end) : objLike;
3    },
```

Listing 24: Slice in the target language

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 listing 25. 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.

```
var __globalScope = typeof __globalScope !== "undefined"
    ? __globalScope :
      typeof globalThis !== "undefined" ? globalThis :
      typeof window !== "undefined" ? window :
      typeof global !== "undefined" ? global :
      typeof self !== "undefined" ? self : {};
```

Listing 25: Global Scope Logic

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 4.8.4. 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.

4.7.6 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. listing 26 illustrates this.

```
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
```

Listing 26: Reserved Keywords in TS and JS

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. Listing 27 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 lambdas work similarly.

```
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;
```

Listing 27: Difference in code generator functions

4.7.7 Stylistic Choices

The syntax of Kipper is specifically designed to ease the transition of existing TypeScript and JavaScript developers to Kipper. Therefore it was important, to keep the output

as similar to these languages as possible. We achieved this by adhering to the following principles.

Human readable output

The primary goal of Kipper's output generation is to produce code that mirrors human-written TS or JS as closely as possible. To achieve this, Kipper avoids unnecessary abstractions or layers that could obscure the intent of the code. For instance, variable names and function identifiers are preserved during transpilation without introducing machine-generated names or hashing schemes. This contrasts with languages like CoffeeScript, where the output, while functional, often requires familiarity with the transpiler's conventions to interpret effectively.

No Code Compression

Kipper explicitly avoids code compression techniques such as minification or inlining that can hinder readability. While compression is useful in production environments to reduce payload size, it is opposed to the goals of Kipper, as it negativly impacts code readability.

Standardized Style Format

Kipper enforces a standardized style format for its output to ensure consistency and predictability. It uses two spaces for block-level indentation to maintain clarity and avoid confusion with tab-based formatting. Braces are explicitly used for block delimiters, and semicolons terminate statements, adhering to common TypeScript/JavaScript conventions.

Scope Visibility

A critical aspect of Kipper's design is the clear representation of scopes in the generated output. Kipper employs explicit declarations such as let, const, and function to demarcate variable and function scopes. Indentation and brace placement further enhance the visual hierarchy, making it easy to identify nested scopes and understand their boundaries. This approach contrasts with languages like Python, where indentation alone determines scope, or languages like Lua, where scope visibility may rely on implicit conventions.

Editable Code

One of Kipper's unique selling points is that its transpiled output is not just readable but also editable. Developers can treat the generated TS/JS code as if it were written manually, enabling seamless integration with existing projects. By making the output editable, Kipper empowers developers to tailor the transpiled code to their specific needs without relying solely on the original Kipper source.

Comparison with other languages

CoffeeScript aimed to simplify JavaScript syntax but often produced output that was hard to debug due to its reliance on non-standard conventions. Kipper avoids these pitfalls by aligning its syntax and output with established TypeScript/JavaScript practices. While TypeScript generates clean and maintainable JavaScript, it requires a compilation step that may introduce additional complexity. Kipper simplifies this process by directly transpiling to both TypeScript and JavaScript, offering flexibility without sacrificing readability. Babel's output is highly optimized for compatibility but can be dense and difficult to modify. Kipper prioritizes maintainability over optimization, ensuring that the output remains approachable for developers.

4.8 Integrated Runtime

4.8.1 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 28. Here a `Programmer` is an `Employee`, but not the other way around. This means `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 `SeniorDeveloper` is still an `Employee` and a `Programmer` at the same time. Even though the `SeniorDeveloper` 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.

```
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 { }
```

Listing 28: Example of nominal typing in Java

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 listing 29 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.

```
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);
```

Listing 29: Example of structural typing in Ocaml

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 30, 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.

```
1  interface Duck {
2    quack(): void;
3  }
4
5  const duck: Duck = {
6    quack: function () {
7      console.log("Quack!");
8    }
9  };
10
11 const person: Duck = {
12   quack: function () {
13     console.log("I am a person but I can quack!");
14   }
15 };
16
17 const dog: Duck = {
18   bark: function () {
19     console.log("Woof!");
20   }
21 }; // <- causes an error in the static type checker
```

Listing 30: Example of duck typing in TypeScript

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.

4.8.2 Runtime Type Concept in Kipper

As previously explained (see section 4.6) the Kipper programming language utilises a similar type system to TypeScript with static typing and a duck-typing approach to complex data and OOP structures. However, unlike TypeScript, we want to ensure full type safety at a runtime level and force the developer to specify the required types and handle edge cases, such as casts and type inference.

Using this approach Kipper allows untyped values, as is the case with the `any` type that is often returned by requests or web elements, or dynamic values to be compared with types that have clearly defined boundaries, such as primitives, arrays, functions, classes, and interfaces, removing any ambiguities that could cause errors.

To allow this functionality, during code generation all user-defined interfaces are converted into runtime types that store the information needed to perform type checks, which form the basis of casts and strict type safety. 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 to any cast, match or typeof operation, guaranteeing that they are fully type-safe.

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.

4.8.3 Base Type for the Kipper Runtime

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 31.

```
1   class KipperType {
2     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  }
```

Listing 31: The structure of a runtime type

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 `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.

4.8.4 Built-in Types for the Kipper Runtime

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 listing 32.

```
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);
```

Listing 32: Examples for the built-in runtime types

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 33.

```
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});
```

Listing 33: Generic built-in types

As can be seen in listing 33, generic types are implemented using a special `KipperGenericType` class. This class, shown in listing 34, extends the `KipperType` and includes an additional field for generic arguments. Most importantly, it includes the method `changeGenericTypeArgument` 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 `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 34.

```
class KipperGenericType extends KipperType {
  constructor(name, fields, methods, genericArgs,
     baseType = null) {
    super(name, fields, methods, baseType);
    this.genericArgs = genericArgs;
  }
  isCompatibleWith(obj) {
    return this.name === obj.name;
  }
  changeGenericTypeArguments(genericArgs) {
    return new KipperGenericType(
    this.name,
    this.fields,
    this.methods,
    genericArgs,
    this.baseType
    );
  }
}
```

Listing 34: Generic Kipper Type

4.8.5 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 35. Additional error types inherit this base type and extend it with additional error information. For instance, the `KipperIndexError` is used whenever an index was out of bounds.

```
class KipperError extends Error {
  constructor(msg) {
    super(msg);
    this.name = "KipError";
  }
}

class KipperIndexError extends KipperError {
  constructor(msg) {
    super(msg);
    this.name = 'KipIndexError';
  }
}
```

Listing 35: Kipper error types

4.8.6 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 36.

```
1  interface Car {
2    brand: str;
3    honk(volume: num): void;
4    year: num;
5  }
6
7  interface Person {
8    name: str;
9    age: num;
10   car: Car;
11 }
```

Listing 36: Example interfaces in the Kipper language

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 36 it would look similar to that in listing 37.

```
1   const __intf_Car = new __kipper.Type(
2     "Car",
3     [
4       new __kipper.Property("brand",
            __kipper.builtIn.str),
5       new __kipper.Property("year",
            __kipper.builtIn.num),
6     ],
7     [
8       new __kipper.Method("honk", __kipper.builtIn.void,
9         [
10          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  );
```

Listing 37: The runtime representation of the previous interfaces

As shown in listing 37, 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 type-checking include the `matches` operator (See section 4.8.7) and the `typeof` operator (See section 4.8.8).

4.8.7 Matches Operator for Interfaces

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 given in listing 38.

```
1  interface Y {
2    v: bool;
3    t(gr: str): num;
4  }
5
6  interface X {
7    y: Y;
8    z: num;
9  }
10
11 var x: X = {
12   y: {
13     v: true,
14     t: (gr: str): num -> {
15       return 0;
16     }
17   },
18   z: 5
19 };
20
21 var res: bool = x matches X; // -> true
```

Listing 38: The Kipper matches operator

As can be seen in listing 38, 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 given in listing 39.

```
1  for (const field of pattern.fields) {
2    const fieldName = field.name;
3    const fieldType = field.type;
4
5    if (!(fieldName in value)) {
6      return false;
7    }
8
9    const fieldValue = value[fieldName];
10   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 }
```

Listing 39: Matches operator property comparison

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.

```
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  }
```

Listing 40: Matches operator method comparison

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

4.8.8 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 listing 41, 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.

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

Listing 41: Typeof operator used to determine the type of an input expression

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 listing 42. 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.

```
typeOf: (value) => {
    const prim = typeof value;
    switch (prim) {
        case 'undefined':
            return __kipper.builtIn.undefined;
        case 'string':
        return __kipper.builtIn.str;
        ...
        case 'object': {
            if (value === null) return
                __kipper.builtIn.null;
            if (Array.isArray(value)) {
                return '__kipType' in value ?
                    value.__kipType :
                    __kipper.builtIn.Array;
            }
            const prot = Object.getPrototypeOf(value);
            if (prot && prot.constructor !== Object) {
                return prot.constructor;
            }
            return __kipper.builtIn.obj;
        }
    }
}
```

Listing 42: Logical implementation of the typeof operator in TypeScript

Although linguistically quite similar, the `typeof` operator in the type declaration of a variable works fundamentally different 43. It is called `TypeOfTypeSpecifier` and it evaluates the type of a variable at compile time.

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

Listing 43: Specifying the type based on a reference variable

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

7.1 Potential Features

Currently, Kipper primarily serves web developers by offering integration with the TypeScript (TS) and JavaScript (JS) ecosystems. However, Kipper remains in the early stages of development as a modern programming language, and several features common in other languages have yet to be implemented. The following sections will discuss these features and explore potential experimental functionality.

7.1.1 WebAssembly Support

WebAssembly is an open standard that enables high-performance, portable code execution across diverse environments, including browsers, servers, and embedded systems. Adding WebAssembly as a compilation target for Kipper provides a potential avenue for enhancing performance and allows Kipper to function as a standalone server application. Emerging technologies such as Wasmer demonstrate that WebAssembly could also support cloud-based container environments while occupying only a fraction of the size of traditional containers.

Implementing WebAssembly as a target for Kipper presents several challenges. Unlike TypeScript and JavaScript, which are dynamically typed and compatible with Kipper's runtime model, WebAssembly requires a more structured type system and memory management approach. This necessitates that the compiler handle low-level memory management, including garbage collection. Additionally, a compatibility layer with a runtime would need to be developed to manage Kipper's dynamic data requirements.

Given these challenges, it is unlikely that WebAssembly support will be included in Kipper in the near future.

7.1.2 IDE Support

IDE support is an important factor in fostering developer adoption and productivity. Providing tooling within popular IDEs can position Kipper as a viable option for developers working on TypeScript (TS) and JavaScript (JS) projects. Effective IDE integration can contribute to code quality and the overall development process.

A key requirement for Kipper IDE support is syntax highlighting. Highlighting keywords, data types, and structures specific to Kipper can improve code readability and reduce cognitive load for developers. This feature is available in the web editor of Kipper. IntelliSense and autocompletion are also significant components, enabling IDEs to provide context-aware suggestions for code completion, parameter hints, and documentation pop-ups, similar to the functionality common in TS and JS environments.

Static analysis tools can assist in maintaining code quality. Integrating a Kipper-specific linter within IDEs allows developers to receive feedback on potential errors, best practices, and stylistic consistency during development.

Initial development efforts may focus on Visual Studio Code due to its widespread use and extensibility. However, consideration for additional IDEs, such as JetBrains products including WebStorm and IntelliJ IDEA, can help accommodate different developer preferences and workflows. Broader IDE compatibility may increase accessibility and support diverse development environments.

7.2 Integration with other languages

7.3 Project Result

Acronyms

AST	Abstract syntax tree
BNF	Backus–Naur Form
GCC	GNU compiler collection
IR	Intermediate representation
OOP	Object Oriented Programming

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 [7] for more information.
Backus–Naur Form	Backus-Naur form is a
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 representation, 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.
Object Oriented Programming	

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.

Bibliography

[1] JetBrains s.r.o. (2024) The state of developer ecosystem in 2023 infographic. [Online]. Available: https://www.jetbrains.com/lp/devecosystem-2023/javascript/

[2] Colin McDonnell. (2024) zod. [Online]. Available: https://github.com/colinhacks/zod

[3] hapi.js. (2024) joi. [Online]. Available: https://github.com/hapijs/joi

[4] Wikipedia. (2024) Three-address code. [Online]. Available: https://en.wikipedia.org/wiki/Three-address_code

[5] ——. (2024) Static single-assignment form. [Online]. Available: https://en.wikipedia.org/wiki/Static_single-assignment_form

[6] Free Software Foundation. (2024) Analysis and optimization of gimple tuples. [Online]. Available: https://gcc.gnu.org/onlinedocs/gccint/Tree-SSA.html

[7] Terence Parr. (2024) Antlr. [Online]. Available: https://www.antlr.org

List of Figures

1 The lexing process which categories the various tokens 22
2 The process of invocating a sub-lexer with the sample input `f"Result: {sample + 4}"` (An example of a template string, or also format string, in the Kipper language), where all content between `{` and `}` is passed onto the sub-lexer. 23
3 A simplified parse tree representation of the statement `var x: num = 4;`. 25
4 An AST produced by the statement `var x: num = 4;`. The data in the brackets is in reality only defined and error checked during semantic analysis (See 4.3), but for the sake of clarity it is already provided here as to not cause confusion due to the missing metadata. 26
5 The design of the GCC compiler 33

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 Runtime typechecks in Java 9
8 Reflection in Java 10
9 Runtime type checking in Rust 11
10 Trait objects and dynamic dispatch in Rust 12
11 Sample snippet from Kipper Lexer grammar 19
12 Function Declaration Grammar 20
13 Holding Values of different Types during Runtime 29
14 Statically Typed Language TypeScript 29
15 Automatic type conversion in JavaScript 29
16 typeof null return "object" in JavaScript 30
17 typeof null return "object" in JavaScript 30
18 The translation algorithm 34
19 The code generation function of a instanceOf expression 35
20 Three-address code 36
21 Static single-assignment form 36
22 The Slice Operation 38
23 Slice in the JavaScript BuiltInGenerator 38
24 Slice in the target language 39
25 Global Scope Logic 40
26 Reserved Keywords in TS and JS 41
27 Difference in code generator functions 42
28 Example of nominal typing in Java 45
29 Example of structural typing in Ocaml 45
30 Example of duck typing in TypeScript 46
31 The structure of a runtime type 48
32 Examples for the built-in runtime types 49
33 Generic built-in types 49
34 Generic Kipper Type 50
35 Kipper error types 50
36 Example interfaces in the Kipper language 51
37 The runtime representation of the previous interfaces 52
38 The Kipper matches operator 53
39 Matches operator property comparison 54
40 Matches operator method comparison 54
41 Typeof operator used to determine the type of an input expression . . . 55
42 Logical implementation of the typeof operator in TypeScript 56
43 Specifying the type based on a reference variable 56

Appendix