HTL Leonding next level

JOHANNES KEPLER UNIVERSITÄT LINZ

HIGHER TECHNICAL FEDERAL COLLEGE LEONDING

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

Supervisor:

Peter Bauer

Project Partner:

Dr. Hanspeter Mössenböck, Johannes Kepler University

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

Abstract

This thesis examines the challenges present in JavaScript environments, particularly the lack of runtime types and comprehensive type safety, and introduces Kipper as a solution. Kipper is a high-level programming language designed to enforce strict type safety at runtime, addressing limitations found in JavaScript and TypeScript, where type correctness is either unchecked or dependent on static analysis tools.

To establish the foundation for Kipper's design, this thesis analyses the JavaScript ecosystem, identifying key issues related to type handling and security. Additionally, potential technologies for implementing Kipper are evaluated, considering their suitability for achieving the project's objectives. Based on this groundwork, the Kipper type system is introduced, incorporating runtime type determination, pattern matching, and interface-based duck typing to enforce type correctness dynamically.

A key focus of this thesis is the development of the Kipper compiler, which translates Kipper code into JavaScript or TypeScript while preserving its type safety mechanisms. The implementation details, including architectural choices and optimizations, are discussed in depth. By addressing the shortcomings of JavaScript and TypeScript, Kipper provides a structured and predictable approach to safer web development. Future work will focus on extending the language's feature set and refining its integration with existing technologies.

Zusammenfassung

Diese Arbeit untersucht die Herausforderungen in JavaScript-Umgebungen, insbesondere das Fehlen von Laufzeittypen und umfassender Typsicherheit, und stellt Kipper als Lösung vor. Kipper ist eine High-Level-Programmiersprache, die entwickelt wurde, um strenge Typsicherheit zur Laufzeit zu erzwingen. Sie behandelt die Einschränkungen, die in JavaScript und TypeScript zu finden sind, wo die Typkorrektheit entweder ungeprüft oder abhängig von statischen Analysetools ist.

Um die Grundlage für das Design von Kipper zu schaffen, wird in dieser Arbeit das JavaScript-Ökosystem analysiert und die wichtigsten Probleme im Zusammenhang mit der Typbehandlung und Sicherheit identifiziert. Darüber hinaus werden potentielle Technologien für die Implementierung von Kipper auf ihre Eignung zur Erreichung der Projektziele hin untersucht. Auf dieser Grundlage wird das Kipper-Typsystem vorgestellt, das Laufzeit-Typbestimmung, Pattern-Matching und Interface-basiertes Ducktyping zur dynamischen Durchsetzung von Typkorrektheit beinhaltet.

Ein Schwerpunkt dieser Arbeit ist die Entwicklung des Kipper-Compilers, der Kipper-Code in JavaScript oder TypeScript unter Beibehaltung der Typsicherheitsmechanismen übersetzt. Die Implementierungsdetails, einschließlich Architekturentscheidungen und Optimierungen, werden abgewogen. Durch die Behebung der strukturellen Probleme von JavaScript und TypeScript bietet Kipper einen konsistenten und konsequenten Ansatz für eine sicherere Webentwicklung. In der Zukunft wird sich das Projekt auf die Erweiterung des Funktionsumfangs der Sprache und die Verfeinerung der Integration mit bestehenden Technologien konzentrieren.

Contents

1 Introduction

Kipper, formally referred to as the Kipper programming language, is a high-level programming language designed to address various safety issues prevalent in the modern web environment. These issues often arise due to the lack of strict typing and type-checking functionality in JavaScript environments. While some of these issues can be mitigated through the use of linter tools or a compile-time type system, such as the one implemented by TypeScript, they typically rely on the user to correctly identify issues and account for all potential edge cases of an operation. This, however, proves dangerous, as allowing users to manage security within an environment creates a hazard for potential edge cases and complex issues that may not be easily resolved through debugging or detailed program analysis.

Kipper seeks to enhance these existing web languages, such as JavaScript and TypeScript, by offering a runtime environment that includes runtime type checks, type determination, and pattern matching—features previously absent in these ecosystems. This approach allows developers to properly identify types and compare them with defined structures during compile- and runtime, ensuring all aspects of an application are correctly validated. However, Kipper does not aim to replace these languages; rather, it should extend them by providing a comprehensive type system and language that accounts for all edge cases and eliminates untyped or potentially unsafe operations.

To achieve this goal, a compiler was developed as part of this thesis to provide these type safety features while generating output code compatible with both JavaScript and TypeScript environments. It functions similarly to other well-known transpilers, such as the TypeScript transpiler, and operates based on the Kipper programming language. Kipper shares similarities with TypeScript but incorporates syntax elements from other languages, such as Python, to simplify various development processes. Its most significant feature is the Kipper type system, which is inherently strict and definitive, preventing the assignment of incorrect or malformed data. This type system incorporates interface-based duck typing and Object Oriented Programming (OOP) paradigms to enable flexible yet safe development.

While the Kipper language is generally not unique in its methods nor its capabilities, its most important features are achieved through the strict reliance on a closed type system. This generally differentiates it from other projects aiming to solve similar problems as previously outlined. A language is generally considered a fairly large undertaking, which requires a lot of justification for why it provides a benefit over existing solutions. In the case of Kipper, it can be argued that runtime libraries could achieve a similar result with less overhead, but in general this can be refuted with the issue of continued reliance on the developer and their safety precautions. As a library generally adds another layer of complexity, there is yet another responsibility for the developer to properly maintain and safely utilize the provided functionality. In the case of Kipper, this would mean maintaining another set of runtime types that can be then used for type comparisons and checks. Furthermore, there is no guarantee that the system will work as intended, as it is left up to the developer to decide when and how to use it, which only worsens the problem of inconsistent and incoherent type safety.

It is important to note that Kipper was not originally conceived as a thesis project or a high-level language in its current form but began as a minor personal project around September 2021. The language evolved over time due to an interest in building upon the JavaScript ecosystem while addressing its commonly perceived limitations. Initially, Kipper incorporated fundamental features such as arithmetic operations, low-level data types as well as user-defined and built-in functions. With the introduction of major features—including error recovery, complex object types, runtime types, and classes—the language has significantly increased in complexity. In its core functionality, Kipper can now be compared to early versions of languages such as JavaScript or Python. This thesis represents a substantial advancement for what was originally a small-scale project. With continued development, Kipper is expected to achieve a full core feature set and direct integration with existing environments in the coming years. However, as of the time of writing, while Kipper is feature-complete within its planned scope, it does not yet include all features commonly found or required in modern programming languages. Essential functionalities such as imports, modules, and asynchronous operations are currently absent from the language.

Furthermore, this thesis should be regarded as a snapshot of the project's development rather than a comprehensive account of its entire history. It primarily focuses on the key features implemented during the 2024/25 period, when Kipper took its current form. Additionally, this thesis does not assert that the language will strictly adhere to all specifications outlined in this paper, nor does it guarantee backward compatibility

with the described features. As Kipper remains in active development, it will continue evolving to achieve a standard feature set suitable for a major v1.0 release.

With this in mind, this paper will discuss the background and challenges present in the web environment, examine various aspects of the Kipper compiler—including its design and the decisions made during its development—and inspect the role of Kipper and how it can help achieve a safer development process.

In Chapter Background, we will discuss the issues present in modern web environments and the underlying motivation for Kipper and its design. Following that, Chapter Technology will talk about the concrete implementation and the technologies used throughout the project. Furthermore, Chapter Type System presents the chosen design for the type system and its extent within Kipper. This is followed by Chapter Implementation, where the design choices, implementation details and the scope of the Kipper Compiler are thoroughly discussed. To showcase the various methods available to use Kipper, Chapter Compiler Reference presents the Compiler API and CLI. This is then followed up by Chapter Demo & Showcase, which presents various Kipper programs as examples for the capabilities of the language. The last Chapter Conclusion & Future outlines the projected future for the project and its potential direction following the completion of this thesis.

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 behaviour. 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 demonstrated 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 presented 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 testing and error correction, 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.

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

...

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

...

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

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.

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

...

let user = { firstName: "Alice" };

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 super-set 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 code bases, 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  class MiddleClass extends SuperClass {
5    superField: SuperClass = new SuperClass();
6
7    constructor() {
8      super();
9    }
10 }
11 class LowerClass extends MiddleClass {
12   classField: MiddleClass = new MiddleClass();
13
14   constructor() {
15     super();
16   }
17 }
18
19 const c1 = <MiddleClass>new SuperClass(); // Unchecked cast
20 console.log(c1.superField.name); // Runtime Error!
       "c1.superField" does not actually exist
21
22 const c2 = <LowerClass>new MiddleClass(); // Unchecked cast
23 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.

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 due

to their inherent limitations of adding another layer of abstraction that needs to be managed.

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

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

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

Listing 6: Runtime issues caused due to ambiguous dynamic data in TypeScript

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 behaviour 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.

```java
import java.lang.reflect.Method;

class ExampleClass {
  public void sayHello() {
    System.out.println("Hello, world!");
  }
}

public class ReflectionExample {
  public static void main(String[] args) throws Exception {
    Class<?> c = ExampleClass.class;
    Object obj = c.getDeclaredConstructor().newInstance();

    for (Method method : c.getDeclaredMethods()) {
      System.out.println("Method: " + method.getName());
      method.invoke(obj);
    }
  }
}
```

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.

```
1  use std::any::Any;
2
3  fn check_type(value: &dyn Any) {
4    if value.is::<i32>() {
5      println!("Value is an i32!");
6    } else if value.is::<String>() {
7      println!("Value is a String!");
8    } else {
9      println!("Unknown type");
10   }
11 }
12
13 fn main() {
14   let num = 42;
15   let text = "Hello".to_string();
16
17   check_type(&num);
18   check_type(&text);
19 }
```

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

This chapter will discuss the technologies selected and utilized in the Kipper project, detailing their application and comparing them to alternative tools and technologies. It will explore the reasoning behind these choices and evaluate their effectiveness in achieving the project's goals.

3.1 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. As such, the following sections will discuss technologies—TypeScript and Antlr4—that were chosen before the start of this thesis project.

Consequently, when the project was re-envisioned and expanded into its current form as part of this diploma thesis, the development technologies had already been determined and were simply continued to build upon the existing foundation.

Nevertheless, to highlight the various alternatives and the reasons behind the original decisions for the parser, lexer, and core compiler, the following sections will still examine each option, assess their viability within the context of this project, and elaborate on the choices made when the initial project was conceived.

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 too complex for its own language, 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

To accurately represent the various benefits and downsides of each individual option, we are going to rank each language based on a set of criteria with individual weights which will amount to a certain total score as presented in Table 1. Based on this score we can accurately represent the viability of each technology option presented.

Weights for the individual criteria		
Criterion	Weight between 0.0-1.0	Effective maximum possible score on a scale of 0-100
Speed	0.4	40
Maintainability	1.0	100
Platform Support	0.5	50
Extensibility	0.4	40
Similarity to Kipper	0.8	80

Table 1: Table showcasing the individual criteria for the programming language and their weights

3.2.2 Option - C++

C++ is widely recognized as one of the most prominent languages for low-level development and system programming, including compiler development. In addition to offering fast execution speeds, it provides flexible mechanisms like pointers for structuring essential constructs such as trees. The language utilizes OOP (Object Oriented Programming) to manage complex structures efficiently and supports templating, often also called generic type parameters, enabling versatile structures with unified behaviour and compatibility.

Speed

In terms of speed, C++ is mostly unrivalled, given the output to assembly code that can be directly run on the CPU without any overhead or interpreter. This would allow a compiler to be similarly fast and efficient, allowing even large programs to be compiled and translated quickly.

Maintainability

In compiler development, C++ is a common choice due to the wide-spread emphasis on minimizing compilation time and efficient processing of large programs. However, the complexity of memory management and the challenges associated with developing large structures can make C++ difficult to work with, potentially leading to unexpected development issues.

Platform Support

Furthermore, targeting multiple environments and systems is challenging, as C++ and by extension C commonly compile directly to machine and os-dependent assembly code, which means compiling the program for each operating system and architecture individually is necessary. Otherwise without virtualisation or subsystems the program can not be executed.

Extensibility

Due to the design of C++, the compiled code is largely locked down and any extensions to the compiler would have to be loaded using custom loaders or the user would have to fully re-compile the source code with the wanted changes.

Similarity to Kipper

While C++ aims to be more modern compared to its predecessor C, it still is largely a low-level language where programs needs to be structured with a lot of potential edge cases and faults in mind. Kipper on the other hand is a pure high-level transpilation-based language with interpreted dynamic targets, so concepts such as memory management, integer sizes and segmentation faults are largely alien. This means if a compiler were to be written in C++, it would require a lot of additional code taking these C++ concepts

into account that would have to be removed or heavily altered in a potential succeeding Kipper-based compiler.

3.2.3 Option - Java

Java is another commonly used OOP language for building compilers and compiler components. As a matter of fact, Antlr4, the lexer and parser generation technology, is developed solely in Java and for extended support simply provides output targets for its lexers and parsers used within other languages.

Speed

Speed is not a major concern when working with Java. The language and its VM are largely optimised and the JIT (Just-in-Time compilation) compiler performs additional optimisations at runtime by analysing the code and the memory of the program. It may in some cases be slower, especially due to the large amount of boilerplate present in a lot of systems, but it can overall be considered fast.

Maintainability

Java can be considered a pure OOP programming language, as all executable code is written in class methods, which can be called up and run from other parts of the program. Due to this all functionality adheres to some structure and is usually encapsulated in various classes and methods. While these features alone do not guarantee code that is easy to work with, they simplify that process and provide structures that can be easily changed or altered using inheritance or other functionality. However it often also leads to boilerplate-heavy code that is hard to read especially when programs become complex.

Platform Support

Java utilizes a VM in combination with a JIT compiler for executing its intermediate code. This intermediate code is largely system-independent and can be run on any system that has a working Java runtime installed. As such, the responsibility of handling the individual characteristics of each operating system and architecture is left up the Java runtime, which makes compiling fairly simple and allows for a broad support of various systems.

Extensibility

Besides providing support for multiple platforms, the architecture of Java allows it to be an easy language to set up and extend, as its modularity means loading additional code simply means compiling it to the same standardised intermediate format. Using a class loader, this new runnable code can be then picked up at runtime and immediately executed. Additionally, the source code classes can be referenced and used within subsequent implementations that extend the original compiler functionality.

Similarity to Kipper

Java is fairly similar to Kipper in its OOP design paradigm, but is still generally very different. Kipper runs as JavaScript or TypeScript in an interpreted environment, where code is ran line per line. While this is similar to the JIT compiler utilised by Java, Kipper can run high-level code and execute on-demand changes without compiling its code or having to generate intermediate code. Given though that structurally the two languages are very similar, migrating from a compiler written in Java to one written in Kipper would be mostly straightforward.

3.2.4 Option - TypeScript

TypeScript is a super-set of JavaScript, which works on the same dynamically-typed and interpreted system and is during runtime de-facto the same language. As such, it is generally not preferred when it comes to implementing a compiler, since its dynamic structure and functionality do not serve a lot of use in a compiler with very specific functionality and limit performance. Still, it is often used for experiments and online compilers, which may not perform as well, but can still achieve good performance.

Speed

In terms of speed, TypeScript can be very fast, almost as fast as WASM (WebAssembly) in some cases. Although it still generally has the same limitations as any other web language. Given that the language needs interpretation and does not utilise JIT compilation or a compiled format, it often can not compete with languages such as Java, C++ or C#. Still, with its heavily optimised interpreter regular algorithms and programs can be fast, but as complexity grows it is not as efficient as comparable languages.

Maintainability

In terms of maintainability TypeScript is generally regarded as maintainable, due to its simplified interpreter structure and detailed compiler errors, but also as problematic due to its common problems that are caused by its own compile-time only types and limitations in performing proper runtime checks or accounting runtime values into a given program. Great care must be put how specific operations are performed and how dynamic data is handled, as errors can easily occur due to oversight or negligence in appropriate type checks.

Platform Support

TypeScript, or its generated JavaScript counterpart, can run on any system that has a browser or a local browser-like counterpart like Node.js, Deno or Bun installed. However, every version of the JavaScript interpreprogramming-language-criteria-weightster comes with its own limitations and even in browsers running code can cause problems across various systems. This is more noticeable with locally installed systems such as Node.js, Deno or Bun, which implement their own libraries and design systems that need to be accounted for. Generally it can be said though that if the correct environment is set up, the code can run on any OS or architecture without issue. In the case of a standard browser, this often becomes even more simple, as browsers try to largely implement the same standards and concepts so that websites and their JavaScript code can run with identical results.

Extensibility

TypeScript is a very dynamic language with a lot of flexibility in terms of extending and modifying structures with ease. Even if a compiler were not able to support modifications, hooking into the object and modifying it if needed is relatively easy. Additionally by simply providing interfaces and specific arguments that need to be implemented a compiler could allow custom classes and data to be executed. Furthermore, referencing the original classes and objects is fairly simple and like Java can be used for subsequent implementations and custom versions of the compiler.

Similarity to Kipper

TypeScript serves as the primary inspiration for Kipper and shares many similarities with it, despite its limitations and challenges related to type safety. This similarity arises

from Kipper's design goal of maintaining compatibility with the JavaScript ecosystem and extending it in a type-safe manner. Consequently, Kipper avoids incorporating functionality or design structures that could break compatibility with existing JavaScript tools and frameworks. With this in mind, theoretically any TypeScript code could be inserted into Kipper with only certain syntax changes and type safety changes necessary.

3.2.5 Result

Based on the analysis performed on each of the options listed in sections 3.2.2, 3.2.3 and 3.2.4, the following results were determined as presented in Table 2.

Result values for the criteria selected in 3.2.1						
Language	Speed	Maintain-ability	Platform Support	Extensi-bility	Similarity to Kipper	Result
C++	40	50	20	10	30	150
Java	35	80	40	30	55	240
TypeScript	25	70	50	40	70	255

Table 2: Table showcasing the individual criteria and the scores each language scored

As presented in the Table 2, C++ is the least adequate with 150 points scored, Java is largely satisfactory in second place with its 240 points scored and TypeScript is the most suitable with 255 points in total. While TypeScript did not score as well as the other two in the "Speed" criterion, it was mostly first place or close behind Java in all other criteria.

This result supports the pre-existing decision presented in Section 3.1.

3.3 Parser & Lexer Technology

The choice of parser and lexer technology, alongside the development language, is a critical factor in the efficiency, extensibility, and maintainability of a programming language. Parsing is inherently complex and can contribute significantly to compilation time. While many language compilers implement custom parsers from scratch to optimise efficiency and minimise execution time, employing parser and lexer generation technology can offer notable advantages.

Using a parser and lexer generator provides greater flexibility, allowing modifications to the language grammar with minimal effort. Additionally, these tools often support multiple languages and compilation targets, enhancing adaptability if the language evolves or expands to additional platforms in the future.

3.3.1 Selection criteria and weighing the options

To accurately represent the various benefits and downsides of each individual option, we are going to rank each technology based on a set of criteria with individual weights which will amount to a certain total score as presented in Table 3. Based on this score we can accurately represent the viability of each parser & lexer generator option presented.

Weights for the individual criteria		
Criterion	Weight between 0.0-1.0	Effective maximum possible score on a scale of 0-100
Speed	0.6	60
Integration	1	100
Maintainability	0.8	80

Table 3: Table showcasing the individual criteria for the parser & lexer technology and their weights

3.3.2 Option - Antlr4

Antlr4 is one of the most widely used lexer and parser generators employing a top-down LL(*) parsing process, offering a comprehensive set of tools and multiple output targets, including a target for native TypeScript. It processes Antlr-specific grammar files and, based on the defined tokens and syntax rules, generates both a lexer and a parser, which are individually executable. Using either a listener or a visitor approach, the resulting parse tree can be traversed and transformed into an AST (Abstract Syntax Tree) for subsequent compilation [4].

Although Antlr4 provides extensive tooling and allows for the extraction of detailed metadata from source code, its generated lexer and parser are generally not considered highly performant compared to the alternatives. Parsing can take a significant amount of time, particularly for large programs. While Antlr4 employs various optimisation techniques—such as improving performance over time as the parser "warms up" and refines

its internal connections—the generic design and modularity introduce overhead that hinders performance optimisations commonly found in other parsing technologies.

Additionally, Antlr4 generates detailed objects and multiple context objects for each token and tree node, leading to high memory consumption. However, in most modern use cases, this is not a major concern.

3.3.3 Option - Coco/R

Coco/R is a compiler generator that, based on a grammar file, produces a scanner (lexer) and parser employing top-down LL(*) parsing capable of validating the syntactic integrity of a file while also performing semantic and type analysis. Unlike a pure lexer and parser generator, Coco/R integrates aspects of semantic analysis into its processing, thereby optimising performance by preemptively conducting checks that do not require complex context-based analysis [5] [6].

By instantiating the scanner and parser within a compiler class, Coco/R-generated classes can be used in a manner similar to Antlr4. However, for generating an AST, Coco/R grammars can be directly configured to construct a tree and build individual custom nodes, similar to the process of generating a parse tree. Using the generated AST, subsequent processing and translation can be performed in a manner nearly identical to Antlr4 [7].

Unlike Antlr4, however, Coco/R does not provide an explicit target for TypeScript. Supporting TypeScript would require either implementing a custom target or translating the generated C# or Java code into WASM, which could then be executed within a TypeScript context. Both approaches would require significant development effort but could likely surpass Antlr4 in performance.

3.3.4 Option - GNU Bison

GNU Bison is a parser generator that employs a bottom-up LR parsing algorithm to validate the syntactic structure of an input token stream [8]. Unlike Antlr4 or Coco/R, Bison primarily generates parsers for C and C++, with an experimental Java target also available. This focus on lower-level languages provides performance advantages, as Bison is highly efficient [9].

Bison processes a grammar definition file and generates a parser that integrates with a lexer, typically implemented using Flex [10]. While it offers strong performance, it lacks

a native TypeScript target. Integrating Bison with a TypeScript-based project would require running the generated C/C++ code via WASM like in the case of Coco/R, adding complexity and overhead due to the additional process layer. Important to also mention is that its tooling and extensibility are less modern compared to Antlr4 and the tool is heavily based on traditional compilers.

Despite these challenges, Bison is a good choice for performance-critical applications where an LR parser is preferred and cross-language execution is acceptable.

3.3.5 Option - Custom Implementation

A custom implementation provides full control over the lexer, parser, and subsequent processing, making it a suitable option for projects with specific requirements or performance constraints. By manually designing a lexer and parser in TypeScript, both speed and memory usage can be optimised while avoiding the overhead associated with general-purpose parser generators.

The primary advantage of a custom parser is its flexibility. Unlike for example Antlr4 or Bison, it can be precisely tailored to the needs of the project, reducing unnecessary complexity. However, this approach demands significant development effort, including manual error handling and debugging, which can be complex and time-consuming.

Additionally, developing a lexer and parser requires a deep understanding of parsing algorithms and careful handling to prevent ambiguity errors and incorrect results. In many cases, a custom implementation may ultimately be slower and less flexible than using a parser generator, as the latter benefits from years of optimisation and widespread use in compiler design, which is hard to compete with without extensive knowledge.

3.3.6 Result

Based on the analysis performed on each of the options listed in sections 3.3.2, 3.3.3, 3.3.4 and 3.3.5, the following results were determined as presented in Table 4.

Result values for the criteria selected in 3.3.1				
Technology	Speed	Integration	Maintainability	Result
Antlr4	35	90	70	195
Coco/R	50	40	50	140
GNU Bison	50	40	40	130
Custom	25	100	20	145

Table 4: Table showcasing the individual criteria and the scores each technology scored

As presented in Table 4, Bison is the least suitable option, scoring 130 points, followed by Coco/R with 140 points. A custom implementation ranks second-best with 145 points, while Antlr4 emerges as the most suitable choice with 195 points.

Despite their performance and capabilities, both Coco/R and Bison present significant drawbacks, primarily due to their poor integration within the target environment and the additional development effort required for full utilisation.

Furthermore, while a custom parser implementation offers advantages in flexibility and control, it would likely under-perform compared to other options and demand substantial effort to maintain and develop further.

This result supports the pre-existing decision presented in Section 3.1.

4 Type System

The type system is a fundamental aspect of the Kipper language. Together with its runtime counterpart (see Section 5.6 for more details), it provides the various type safety features envisioned for this thesis. The type system also represents the most significant difference between the underlying JavaScript and TypeScript environments, which Kipper compiles to. Unlike these languages, Kipper does not adhere to their inherent limitations but instead extends and enhances them, addressing certain edge cases and providing a more complete and robust type system.

4.1 Primitives

The foundation of the type system consists of primitive values, which are predefined by the language and encapsulate fundamental type concepts such as numbers, strings, and objects. These primitives represent the most abstract and basic form of values, meaning they are inherently incompatible with one another since they do not share a common parent type that would enable implicit conversion or compatibility. The only exception to this rule is the `any` type which diverges from this behaviour and represents any potential value (see Section 4.5 for more details).

This distinction is crucial because Kipper allows types to extend other types, thereby narrowing their definition. For example, interfaces refine the broad object type by imposing specific constraints on objects. By enforcing strict type boundaries at the primitive level, Kipper ensures that more complex structures built upon these primitives maintain clear and predictable type behaviour.

4.1.1 Null Type

In Kipper, the `null` type is a distinct type used to explicitly indicate the absence of a value. Unlike many other languages where `null` is treated as a special value assignable to an `obj` type, Kipper enforces a stricter separation. This design choice prevents implicit nullability in objects, ensuring that an object type does not inherently allow `null` as a valid assignment unless explicitly stated.

This approach aligns with Kipper's design philosophy of strict type safety, where developers must explicitly define whether a variable can hold `null`. If nullability is required, it must be explicitly included in the type definition using union types. By enforcing this distinction, Kipper eliminates potential runtime errors caused by unintended `null` values, reducing the risk of null reference exceptions.

4.1.2 Undefined Type

The `undefined` type in Kipper, like `null`, is a distinct type that can only be assigned to its direct value, `undefined`. It serves as a way to mark the absence or non-existence of a value. This behaviour is inherited from JavaScript, where `undefined` often indicates an uninitialized variable or a missing function parameter.

Kipper does not enforce a strict distinction between when to use `null` versus `undefined` for indicating missing values. Instead, it leaves this choice up to the developer, allowing for flexibility depending on the specific use case. However, since both `null` and `undefined` are treated as separate types, developers must explicitly account for them when handling potential missing values, reinforcing strict type safety.

4.1.3 String Type

The `str` type in Kipper is the standard string type, capable of containing any UTF-16 character, as dictated by the underlying JavaScript specification [11]. Each character in a string is represented by a UTF-16 code unit, which means it can store up to 65,536 different characters directly. Additionally, Kipper allows the use of surrogate pairs-two combined code units-to represent characters beyond the Basic Multilingual Plane (BMP), such as various emoji and less common Unicode symbols.

Because Kipper builds on JavaScript's string handling, string operations such as indexing, slicing, and concatenation follow familiar rules. However, developers must be mindful of multi-code-unit characters, as certain operations (like accessing a string by index) may return only part of a surrogate pair rather than the full intended character.

4.1.4 Number Type

The `num` type in Kipper is the standard numeric type and operates in the same manner as the regular JavaScript `number` type. It can store any value, but its accuracy may

decrease beyond the range of $-2^{53}-1$ and $2^{53}-1$. Unlike JavaScript, however, Kipper does not support the `bigint` type for numbers outside this range.

To specify a number, hexadecimal, octal, binary, and standard base-10 representations may be used.

4.1.5 Boolean Type

The `bool` type is a standard boolean type with only two possible values: `true` and `false`. Unlike in other languages, it is not directly compatible with numbers and cannot be used to represent `0` or `1`.

4.1.6 Object Type

The `obj` type is a primitive type in Kipper. Unlike many other languages, this type simply expects any object and does not require specific properties or methods to be present. It is also the parent type of any interface (see Section 4.3 for more details on interfaces) and class (see Section 4.4 for more details on classes). However, since the `obj` type does not define any child members, it is practically unusable unless cast to a specific interface or class type, as the compiler will not permit unsafe member access.

4.1.7 Void Type

The `void` type is a specific feature unique to Kipper. Unlike JavaScript or TypeScript, this type acts as an alias for `undefined` and carries the special type information that the function marked with `void` does not return a value. This distinction is important, as `undefined` is technically a returnable value. To enforce that a function does not return a value, it must be explicitly marked as returning `void`.

4.1.8 Meta-Type Type

The meta-type `type` is unique to Kipper and represents the runtime type, which can be obtained using `typeof` or referenced by a type name in an expression. This type allows types to be treated like values and compared with one another, enabling two `typeof` results to be compared (see Section 5.6 for more info on the semantic of a runtime type).

4.2 Generics

Generics are a special type in Kipper, as they are parameterised and can take on various forms that do not necessarily have to be compatible with one another. Due to their complexity, Kipper, as of the time of writing, only supports built-in types as generics. Neither classes nor interfaces can have generic parameters, unlike in TypeScript or C#.

The two supported generic types are `Array<T>` (see Section 4.2.1) and `Func<T..., R>` (see Section 4.2.2). Both are represented as generics during runtime and retain the specified generic arguments (see Section 5.6.4 for more information on generics at runtime).

4.2.1 Array Type

The `Array<T>` type is the generic type representation of an array, or more accurately, a dynamic list. This list can have any length and may contain any member values, provided they satisfy the given `T` argument. Arrays have unique type-checking requirements, as both the type and the generic type must match and be compatible. The length, however, does not matter and is ignored, unlike in some other languages.

4.2.2 Function Type

The `Func<T..., R>` type is the generic type representation of a function or lambda, which can have `n` numbers of `T` arguments and must have a return type `R`. Similar to TypeScript, the parameter declaration has no limit on the number of parameters and can realistically support any number of parameters.

As with the `Array<T>` type, functions also have similar type-checking requirements, with the additional condition that the number of parameters must match, alongside the parameter types and return type being compatible.

4.3 Interfaces

Interfaces are one of the most important features in Kipper, as they are essential for runtime type checks, type matching (see Section 5.6.7 for more information), and typing of dynamic data. This is particularly important for input or dynamic data, which is produced outside a controlled environment and must be type-checked.

As described in Section 5.6.6, interfaces also have a runtime counterpart, unlike TypeScript, which stores important type information and allows type checks during both compile time and runtime.

Defining an interface only requires a name and the required properties and methods that the envisioned object should contain. An example of this is demonstrated in Listing 11.

```
interface Intf {
  foo(bar: num): num;
  x: num;
  y: str;
}
```

Listing 11: The definition of a simple interface with a method and properties

Besides explicit interface definitions, any constant object automatically receives an anonymous implicit interface type, which allows for further type checking and assignments to interface types. Using duck typing-a concept where the so-called "Duck Test" is performed, i.e. "if it quacks like a duck and walks like a duck, then it must be a duck"-the interfaces can then be compared and matched with one another to check whether they are compatible.

In this context, compatibility simply means that if the expected type `A` is a subset of or identical to incoming type `B`, then `B` is assignable to `A`. Direction is key here, as it does not work the other way around. A value being assigned to a specific type must implement all members of that type to be considered assignable. In our example, this would mean that `B` is being assigned to `A`, and therefore `B` must at least have all the requirements of type `A`, but anything beyond that does not matter and is ignored.

```
interface A {
  foo(bar: num): num;
  x: num;
  y: str;
}

var a: A = {
  foo: (bar: num): num -> bar * 2,
  x: 1,
  y: "2",
  z: true
};
```

Listing 12: Demonstration of duck typing within Kipper

The example shown in Listing 12 demonstrates this behaviour, as the example object defines all the required members of interface `A`, but additionally defines properties unknown to `A`. Thus despite the type being different, the example object passes the "Duck Test" and is compatible with type `A`.

4.4 Classes

Classes in Kipper are encapsulated structures used to define objects, containing both properties and methods. Unlike interfaces, which act as blueprints, classes can have implemented functionality and can be instantiated using the `new` keyword. A class may have a constructor that takes parameters, which must be provided for instantiation, and it can execute specific logic during object creation.

Class properties in Kipper are always public and can be modified both internally by methods and externally by other code. They hold no value upon instantiation and must be defined in the constructor to properly store a value. Similarly, class methods are also always public and can be called by other internal methods or by external code invoking the specific method on the instance object.

Defining a class only requires a name and can otherwise remain empty. Any properties, methods, and the constructor are optional, and by default, a class can simply be instantiated by calling `new ...()` with `...` being the name of the class.

```
1  class Cls {
2    x: num;
3    y: str;
4
5    constructor(x: num, y: str) {
6      this.x = x;
7      this.y = y;
8    }
9
10   foo(): void {
11     print(f"Hello from {this.y}!");
12   }
13
14   bar(): num {
15     return this.x * 2;
16   }
17 }
```

Listing 13: The definition of a simple class with a constructor methods and properties

As illustrated in Listing 13, a typical class contains a number of properties and methods, as well as constructor initialization parameters that define the properties of the class.

Other statements can also be included in the constructor, such as writing output to the console or calling functions.

Furthermore, initializing the class demonstrated in Listing 13, can be done using `new Cls(..., ...)`. Accessing properties or methods follows the standard dot-notation used in most programming languages, allowing direct referencing of the desired property or method as shown in Listing 14. In this way, a class instance behaves like any other object in Kipper.

```
1 var cls: Cls = new Cls(1, "my class");
2 cls.foo(); // -> "Hello from my class"
3 print(cls.bar()); // -> 2
```

Listing 14: Accessing the members of a Kipper class

4.5 Any Type

The `any` type is a special type in Kipper, primarily used to define dynamic data whose type cannot be determined at compile time but only at runtime. It is mainly intended for handling dynamic data received from web requests or APIs, such as the browser DOM or the Node.js library API, which often return this type due to the absence of type information or restrictions.

Since the `any` type is compatible with all other types, it is only assignable to itself and cannot be directly used with specific types without narrowing through type casts, such as `cast as`, `force as`, or `try as` (see Section 4.8 for more information). However, once the `any` type is narrowed, it can be used like any other value without inherent limitations.

4.6 Null Type Union

The null-type union operator `T?`, commonly referred to as the null-able operator, applies to the provided type `T` by creating a union with the `null` type, indicating that a value may be either `T` or `null`. This operator allows types to be easily marked as potentially missing, which is particularly important for cast operations such as `try as T` (see Section 4.8).

4.7 Undefined Type Union

Similar to the `T?` operator, Kipper implements the `T??` operator, which functions similarly but unifies the provided type `T` with the `undefined` type, thereby marking it as potentially missing. While this operator is less commonly used-since `try as T` only returns `T?`-it remains useful in various scenarios, particularly in modern APIs and libraries where `undefined` is often preferred over `null`. This design choice aligns with TypeScript's approach, where the null-able operator in objects applies only to `undefined`, requiring explicit marking for `null`.

4.8 Narrowing Type Casts

Casts are an essential component of the Kipper type system and form a core aspect of the language. They can be performed at both compile time and runtime, with runtime-bound type checks ensuring all-round type safety. By using these casts, both values known at compile time and those resolved only at runtime can be appropriately handled within a program. This approach ensures that type checking remains in-tact while preventing potential edge cases or the circumvention of type safety mechanisms.

4.8.1 Compile-time Cast

The compile-time cast operator `cast as` is the standard cast operation in Kipper, functioning similarly to TypeScript and other statically typed languages. It allows one type to be cast to another under the supervision of the compiler, which will produce an error if the cast is invalid or impossible. While some languages, such as C#, perform runtime checks to ensure cast safety, Kipper does not, as it enforces stricter compile-time type safety and does not allow potentially unsafe casts using `cast as`. For cases where compatibility cannot be determined at compile time, the `force as` and `try as` operators can be used instead.

As demonstrated in Listing 15, the `cast as` operator can perform type narrowing as long as the compiler is aware of the fact that the provided types are compatible.

```
1 interface Input {
2   x: num;
3   y: num;
4 }
5
6 def fun(input: Input) -> void {
7   ...
```

```
8  }
9
10 var input: Input = {
11   x: 1,
12   y: 2,
13   z: 2,
14 } cast as Input; // Allowed
15 fun(input);
```

Listing 15: Correctly casting an object to an interface using the `cast as` operator

In case the cast is unsafe or ambiguous, the compiler will throw an error as is the case as presented in Listing 16.

```
1  interface Input {
2    x: num;
3    y: num;
4  }
5
6  def fun(input: Input) -> void {
7    ...
8  }
9
10 var input: Input = {
11   x: 1,
12 } cast as Input; // Throws compile time error - Property 'y'
       is missing
13 fun(input);
```

Listing 16: Incorrectly casting an object to an interface using the `cast as` operator

4.8.2 Runtime Force Cast

The runtime cast operator `force as` is one of the two available runtime casting mechanisms in Kipper, utilizing runtime type information and the matching functionality embedded in the runtime system. Unlike the `matches` operator (see Section 5.6.7 for more details), `force as` performs a comprehensive type check across all supported types to determine compatibility. This includes not only interfaces but also primitive types and classes, ensuring that all relevant type constraints are enforced at runtime.

If the cast is invalid at runtime, an error is thrown, causing the program to fail at that location. However, this guarantees that the compiler can assert the return type of the cast operation will always be of type `T`, as an invalid cast would terminate execution, preventing the operation from being performed with an incorrect type.

As demonstrated in Listing 17, the `forc as` operator can perform type narrowing as long as the compiler is aware of the fact that the provided types are compatible.

```
interface Input {
  x: num;implements
  y: num;
}

def fun(input: Input) -> void {
  ...
}

var input: Input = {
  x: 1,
  y: 2,
  z: 2,
} force as Input; // Allowed
fun(input);
```

Listing 17: Correctly casting an object to an interface using the `force as` operator

In case the cast is unsafe or ambiguous, the compiler will throw an error as is the case as presented in Listing 18.

```
interface Input {
  x: num;
  y: num;
}

def fun(input: Input) -> void {
  ...
}

var input: Input = {
  x: 1,
} force as Input; // Throws KipTypeError: Invalid force cast
    from 'obj' to 'Input'.
fun(input);
```

Listing 18: Incorrectly casting an object to an interface using the `force as` operator resulting in a runtime error

4.8.3 Runtime Try Cast

The runtime cast operator `try as` is the second of the two runtime cast operators in Kipper, enabling conditional casting by returning `null` if the cast is invalid. This differs from `force as`, as it requires the developer to handle the potentially null return type `T?`. By employing a conditional statement, the return value can be verified and appropriately processed, allowing the developer to account for both possible outcomes of the cast operation.

As shown in Listing 19, the return value of `try as` must be assigned to a variable that supports both `T` and `null` using the previously mentioned `T?` null type union operator (see Section 4.6). By utilising an if-statement, both outcomes can be handled: in one case, the program simply ends, while in the other, the program proceeds to prints out the cast value.

```
1 var data: any = 2;
2
3 var userId: num? = data try as num;
4 if (userId == null) {
5   print("Invalid input data");
6 } else {
7   print(f"The user id: {data}");
8 }
```

Listing 19: Casting an `any` type value using the `try as` cast operator and handling both possible outcomes

5 Implementation

This chapter will discuss the implementation of the Kipper compiler, detailing its various components, language features, and the thought process behind their development. It will also include comparisons with other languages and provide background information on the design choices made during implementation.

5.1 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 5.5, 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.

5.1.1 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 5.2
- 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 5.2
- 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 5.2.4
- 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 5.3
- Performed by: **Kipper Core Compiler**
- This phase is split into three separate steps:
    1. Primary Semantic Analysis (see Section 5.3.3): Validates language semantics such as variable declarations and scope resolution.
    2. Preliminary Type Analysis (see Section 5.3.4): 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 (see Section 5.3.5): Conducts in-depth type validation and resolves type-related issues for the given statements and expressions.

Target-Specific Requirement Checking

- Addressed shortly in Section 5.5.4
- Performed by: **Target Semantic Analyser**
- Ensures compliance with the specific requirements of the target platform (JavaScript or TypeScript). This step is relatively minor and primarily ensures that no code is translated in a way that would cause incompatibilities with the target.

Code Optimization

- Addressed shortly in in Section 5.5.5
- Performed by: **Kipper Core Compiler**
- Performs code optimizations, currently focused on tree-shaking to eliminate unused code.

Target-Specific Translation

- Detailed explanation in Section 5.5
- 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.

5.2 Lexing & Parsing

The first step in the compilation process is the lexing and parsing of the input program. This involves the tokenisation 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.

5.2.1 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 Section 5.2.2 for more info).

For example, the lexer grammar includes constructs for identifying both comments and language-specific keywords.

In the grammar as demonstrated in Listing 20, 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.

```
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 20: Sample snippet from the Kipper Lexer grammar

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 tokenisation 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 21:

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

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

Listing 21: Function Declaration Grammar

`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.

5.2.2 Lexical analysis

Primary tokenisation

The primary task of the lexical analysis is the tokenisation 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 tokenisation is fairly straightforward as it simply follows the definitions provided in the grammar file (see Section 5.2.1) 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 5.2.2.

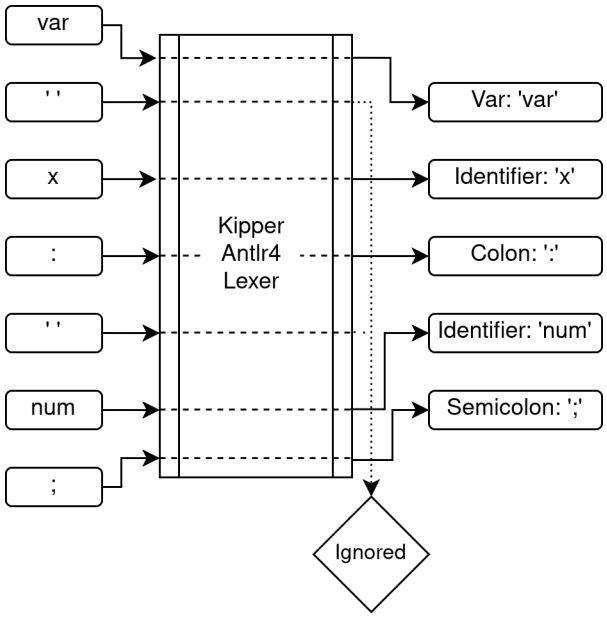


Figure 1: The lexing process which categories the various tokens

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 tokenisation scenarios and increases the lexer's flexibility.

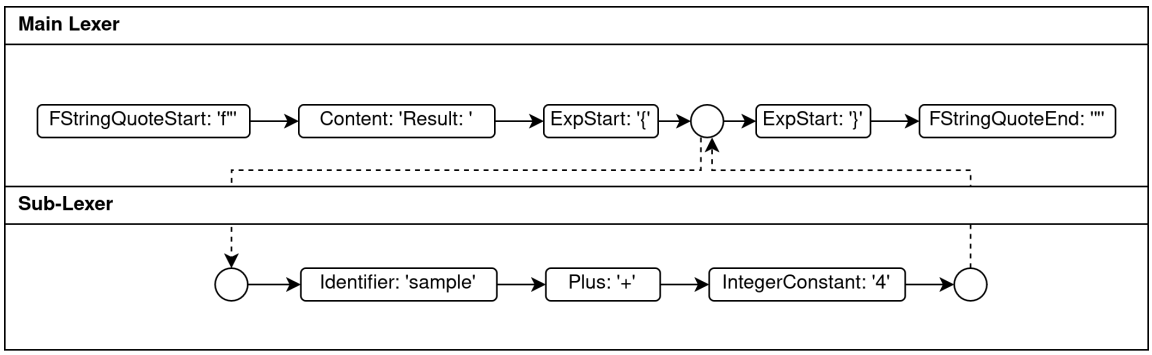


Figure 2: The process of invoking a sub-lexer with the sample input `f"Result: {sample + 4}"` (An example of a template string, or also format string, in the Kipper language), where all content between `{` and `}` is passed onto the sub-lexer.

5.2.3 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 analyse 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 sub-tree 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.

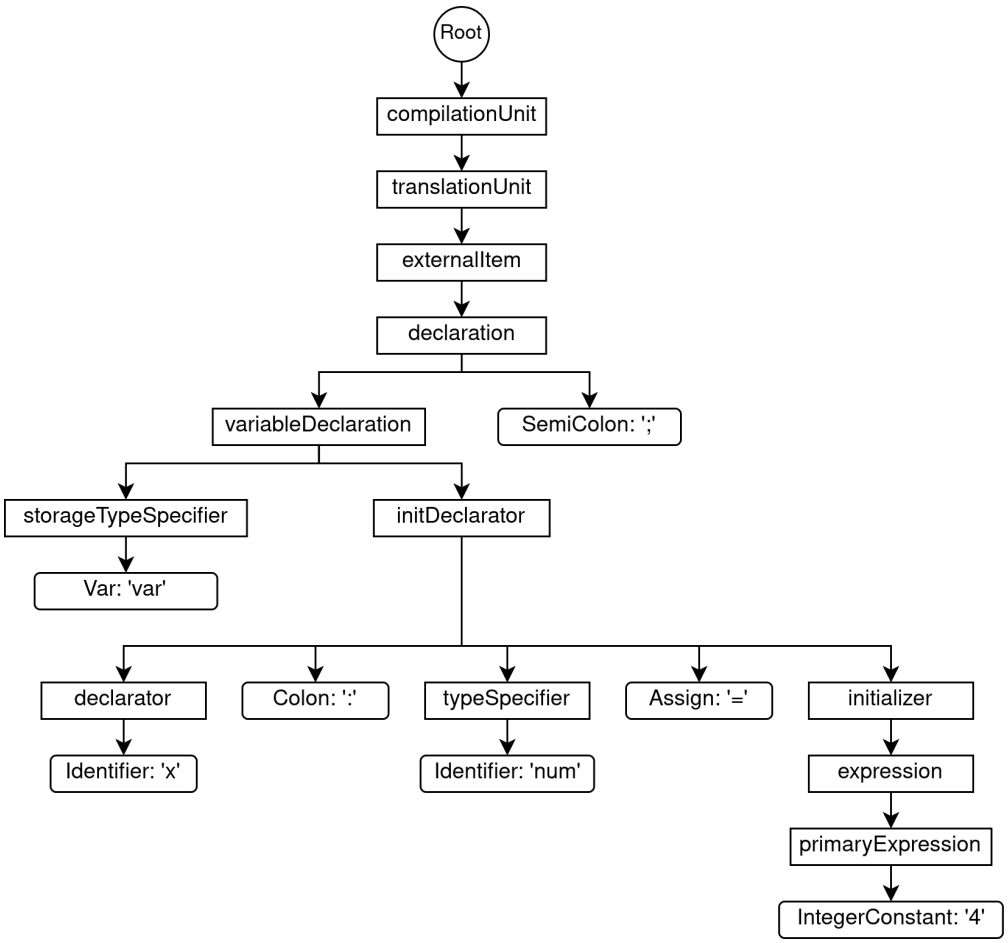


Figure 3: A simplified parse tree representation of the statement `var x: num = 4;`.

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.

5.2.4 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 artefacts, 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.

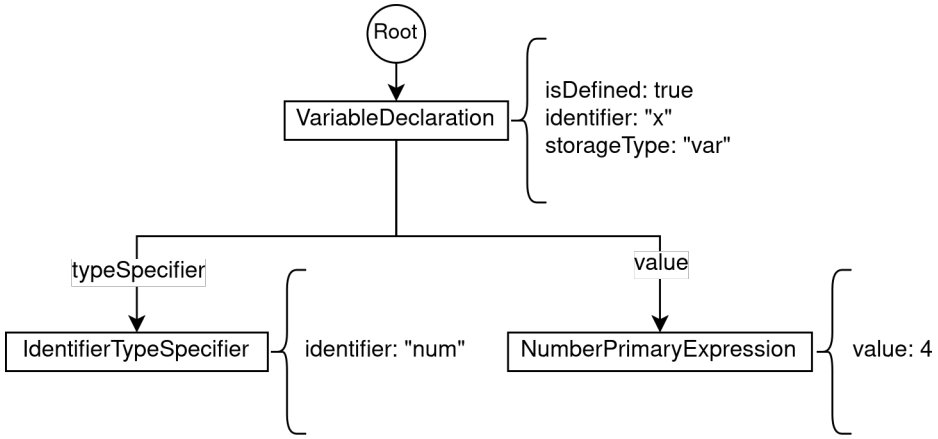


Figure 4: An AST produced by the statement `var x: num = 4;`. The data in the brackets is in reality only defined and error checked during semantic analysis (see Section 5.3), but for the sake of clarity it is already provided here as to not cause confusion due to the missing metadata.

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 Section 5.4 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 Section 5.5), where all existing information is required to accurately generate output code which logically adheres to the original program.

5.3 Semantic Analysis

Semantic analysis is the most complex and critical stage of the compiler, as it extracts essential logical information from the input program and populates the AST built in the previous step with the essential information related to each individual node, ensuring that subsequent processing steps have the necessary data to operate correctly.

Semantic analysis can be broadly divided into two key tasks:

- **Evaluating Semantic Data**
- **Validating Integrity and Raising Errors**

These tasks are inherently interdependent, as the compiler typically operates on an assertion-based principle—expecting specific conditions to be met and raising an error when the required input is absent or invalid.

Processing can differ largely in size depending on the node in question, as certain nodes can take advantage of the fact that certain truths are implicitly implied by the parser and therefore don't require verification. Other more ambiguous expressions or statements may require more checks to verify that the input code is valid.

5.3.1 Separation of Concerns

The Kipper compiler utilises the AST as the foundation for semantic analysis, traversing the tree and analysing individual nodes. This approach follows the principle that each node is responsible for processing itself and its children, allowing nodes to be evaluated independently of their broader context while focusing on their essential aspects. This methodology is particularly significant, as Kipper aims to preserve the abstract program structure and ensure that each node possesses the necessary information for translation. This differs from other compilers, which may employ a flat translation approach, generating symbol tables and instructions as needed without maintaining abstraction.

This design also establishes a trust-based system in which parent nodes can rely on their children to provide the required information for later translation. Each AST node implementation consists of the class itself, an interface defining the semantic data to be populated, and an interface defining the type-related semantic data to be populated. This separation of concerns is a distinctive feature of the Kipper compiler, as semantic and type analysis are often performed concurrently or combined in other compilers. However, in Kipper, the entire AST is traversed separately for each semantic analysis stage.

This staged approach is essential for maintaining program integrity, ensuring logical processing order and guaranteeing that all required data is available when needed. For example, references can bypass the regular processing sequence and may require specific data to be available from another node. To enforce this logic and prevent any logical processing issue or circular reference issue, the Kipper compiler follows a strict rule: each stage must be internally complete and may only reference data generated by earlier nodes in the program or by any node in a preceding stage.

5.3.2 Algorithm

The Kipper compiler performs a total of five distinct steps during semantic analysis, though two of these play a relatively minor role, as they currently apply only to specific use cases. One of these is the warnings check, which is only supported for certain nodes and simply invokes the warning check function if a node has implemented it. The other is more closely related to target generation, as it executes additional semantic checks defined by the specified target (see Section 5.5.4 for more info)

The remaining three stages are the most significant, as they each contribute to a specific aspect of semantic analysis, node integrity verification and data definition. These stages are as follows:

- **Primary Semantic Analysis**
- **Preliminary Type Analysis**
- **Primary Type Analysis**

Each stage follows the same traversal algorithm, where the compiler begins at the root AST node and recursively invokes checks on its child nodes as demonstrated in Figure 5. These child nodes, in turn, call their own children, continuing the process throughout the tree. This traversal can be described as a bottom-to-top, left-to-right approach in terms of the AST or as a top-down, left-to-right order from the smallest child to the greatest parent when considering a regular file with individual lines.

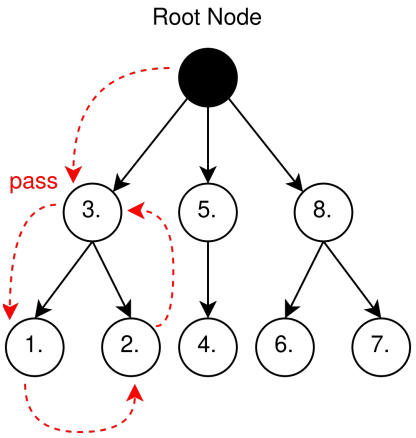


Figure 5: A simplified representation of the semantic analysis tree-walking algorithm. The red dotted line illustrates the path the compiler follows to locate the outermost left leaves, which are processed first, before backtracking to process their parent node. Each node has a ordinal number assigned indicating the order in which it is processed.

Once a node has completed processing, it will have defined its own semantic data object, storing the necessary information for that specific node. If no data is present, this indicates that the node has failed processing, and an error has already been generated either within that node or in a related child node.

5.3.3 Primary Semantic Analysis

Primary semantic analysis is the initial stage of semantic analysis and is responsible for handling fundamental aspects of a program, such as verifying the availability of names for declarations and definitions, evaluating references, and processing other essential

elements. This stage does not involve type checking but instead focuses solely on extracting and validating general semantic data from the core elements of the language. Every node in the AST must undergo this step, as without it, the node would lack all functionality.

This step also is largely responsible for verifying references by building up the symbol tables for the various scopes created by functions, compound statements, classes or other related elements. (see Section 5.3.6 for further info).

5.3.4 Preliminary Type Analysis

Preliminary type analysis serves as a precursor to standard type analysis, focusing on processing type definitions before other elements reference or modify them. This step is essential, as types do not always follow a strict hierarchical structure. For instance, a class definition contains elements that reference the class type itself, despite being its children.

This scenario conflicts with the principle outlined in Section 5.3.1, which states that a node may only reference elements from previous stages of semantic analysis or nodes already processed before it. Since child nodes must be analysed before their parent nodes, reversing this order would be problematic. To address this, preliminary type analysis ensures that types are loaded first, allowing them to be referenced later without the risk of missing or undefined types. This approach maintains the integrity of the algorithm and preserves the intended order of each semantic analysis stage.

As in the previous stages, this step populates the essential symbol tables with the types defined and processed during this step, and verifies their validity.

5.3.5 Primary Type Analysis

Primary type analysis is the final major stage of semantic analysis, responsible for performing type checks, handling type operations, and resolving type references. Given its role in managing the complexity of objects, interfaces, classes, and unions, it serves as a fundamental component of the compiler. Similar to primary semantic analysis, it ensures the correctness of references and verifies that all type references are valid.

During this phase, all nodes have their type data defined, specifying what type information they store and how they function within a given context. In the case of expressions, this means determining the type to which they evaluate, enabling parent nodes to

compare child nodes and validate operations, such as arithmetic operations between numerical values. This is crucial for expressions, as they are composed hierarchically, requiring validation of child nodes within their specific context.

In contrast, statements and declarations do not evaluate to a type. While they may perform operations or define types, they cannot be used within another node as part of an expression, except as standalone instructions within a program.

5.3.6 Reference Handling and Scopes

Reference handling in Kipper operates similarly to most other languages, relying on individual scopes with their own symbol tables that store variables, functions, types, and other relevant entries. These scopes are structured in a hierarchical manner, allowing each scope to access information from the ones below it.

This mechanism is crucial throughout semantic analysis, as each stage is responsible for managing symbol table entries, ensuring that references are valid, and preventing improper overwriting or modification of existing entries.

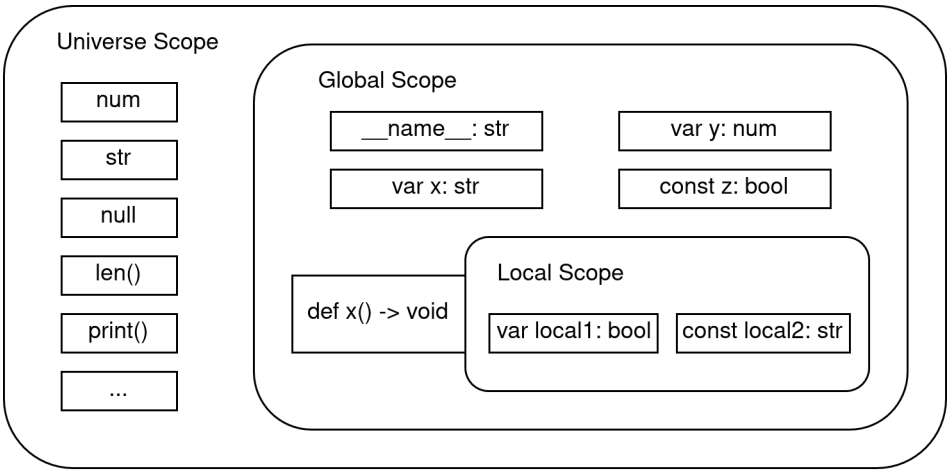


Figure 6: A simplified representation of the various scopes within Kipper that each act as a layer on top of a stack.

Kipper currently implements three types of scopes as also visualised in Listing 6:

- **Universe Scope**: This scope defines all built-in types and functions that are always available in Kipper.
- **Global Scope**: This scope contains all global definitions for a specific Kipper program, including file-specific built-ins such as `name`, which returns the filename.

- **Local Scope**: Any scope nested within the Global Scope is considered a Local Scope. Entries defined within a Local Scope are accessible only within that scope and its child scopes.

Within a Kipper program, only the **Local Scope** can be defined by the user, while all other scopes are either compiler-bound or file-bound.

5.4 Error recovery

The concept of error recovery is fundamental to most modern compilers, as it enables the detection and reporting of multiple errors within a single compilation run. Basic compilers often employ an immediate fail-safe approach, terminating the compilation process upon encountering an error to prevent incorrect assumptions about the program. While this approach preserves the integrity of the compilation process, it presents a significant drawback: in large and complex programs, developers must repeatedly correct errors and recompile to identify additional issues, resulting in inefficient error handling and unnecessary delays during development.

To address these challenges, Kipper implements its own error recovery algorithm, which allows the compiler to recover from errors and continue processing by advancing to subsequent expressions or statements that are not directly associated with the original error node.

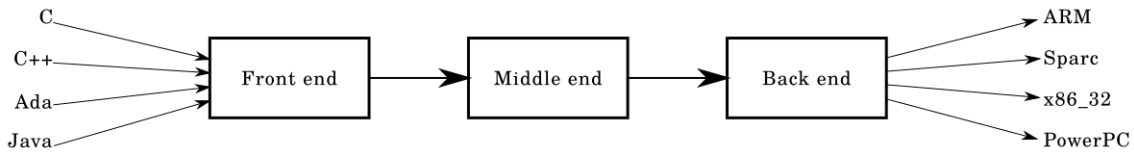
However, Kipper's error recovery is limited to the semantic and type analysis phases. The parser, as described in Section 5.2, does not support error recovery and terminates processing upon encountering a syntax error. This behaviour is atypical for a lexer and parser, as they usually allow parsing to continue despite syntactically faulty code blocks. However, due to specific limitations in the version of Antlr4 used, this functionality is not currently supported.

5.5 Output Generation

The Kipper compiler is built on a modular architecture, allowing for the definition of custom output targets and providing flexibility to accommodate various use cases. This modularity extends beyond basic configuration, enabling developers to specify target-specific features and behaviours. The modular design is structured around two primary components: the frontend and the backend.

For comparison, the GCC (GNU Compiler Collection) compiler achieves modularity by dividing its architecture into three components, as illustrated in Figure 7. 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.



Source: https://commons.wikimedia.org

Figure 7: The design of the GCC compiler

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.

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 tree-shaking.

5.5.1 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.

5.5.2 Algorithms used for Output Generation

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

Linear Algorithms

Linear algorithms are commonly employed when compiling high-level languages to machine code or bytecode. These algorithms treat the input as a flat, ordered sequence and process it sequentially. Intermediate representations (IR) are typically in the form of three-address code or static-single-assignment (SSA) form. Linear algorithms process the code one instruction at a time in sequence. Upon completion of an instruction, it is added to a list, which is eventually concatenated and written to the output file.

Three-address code consists of three operands and typically represents an assignment with a binary operator [12]. An instruction may have up to three operands, although fewer can also be used. In Listing 22, the problem is divided into multiple instructions. This structure allows the compiler to easily translate the instructions into assembly language or bytecode, which share a similar format. Additionally, the compiler can identify unused code by determining whether a variable is referenced later in the code.

The Static Single-Assignment (SSA) form is an alternative intermediate representation in which each variable is assigned exactly once [13]. It is widely used in compilers such as GCC. The primary advantage of SSA is that it simplifies the code and enhances the effectiveness of compiler optimizations.

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

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

Listing 22: Three-address code

In Listing 23, a variable `y` is assigned twice. The first assignment is redundant. In SSA form, the compiler can identify that the assignment to `y1` is unnecessary, as it is not used in the subsequent code. Optimizations improved by the use of SSA include dead-code elimination, constant propagation, and register allocation.

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

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

Listing 23: Static single-assignment form

Tree-based Algorithms

Tree-based algorithms are widely used in transpilers because they preserve the readability and structural integrity of the source code. By maintaining the original scopes and statements, they ensure that the transformed code remains as close as possible to its initial form. To achieve this, code components are stored as nodes in a tree-like structure.

Kipper employs a bottom-up code generation algorithm, in which a tree-walker recursively traverses the syntax tree, starting from the deepest nested nodes to generate the output. This approach was selected for its simplicity in visualization and implementation, as well as its ability to maintain human-readable and easily extendable source

code without aggressive transformations. Linear algorithms, by contrast, convert code into intermediate representations such as three-address code or static single-assignment (SSA) form, often stripping away valuable structural and contextual information.

Although tree-based code generation can produce bytecode, which is subsequently optimized and compiled using linear algorithms, it has inherent limitations. Specifically, optimizations remain localized to individual nodes, and the recursive traversal can lead to high computational costs for complex inputs. Due to these constraints, tree-based designs are rarely used in professional bytecode-generating compilers. However, GCC incorporates a tree-based design in two of its language-independent intermediate representations (IRs): GIMPLE and GENERIC [14].

5.5.3 Generation Algorithm

The output generation process in Kipper starts by initializing the target environment and setting up any necessary dependencies, as outlined in Section 5.5.5. Once this setup is complete, the compiler iterates through the previously generated AST nodes, invoking the `translateCtxAndChildren` function for each node. This function performs a recursive traversal of the syntax tree, generating code for each child node. Each node returns a string containing its corresponding output code, which is then processed by its parent node. The process continues until the root node collects and merges all generated strings, passing the final output to a function responsible for writing the translated code to a file.

This bottom-up processing approach ensures that child nodes are translated before their parent nodes, allowing parent nodes to retrieve and incorporate necessary information from their children. This methodology is particularly crucial for handling complex structures, as these often rely on the embedded code produced by their child nodes. The generated code is passed up the tree as an array of tokens representing the textual form of the output.

The output generation process ensures reliability by leveraging the fact that the AST has already been validated for syntactic and semantic correctness during earlier compilation phases.

The implementation of this translation algorithm is illustrated in Figure 8.

The code generation function of a node accepts the node as an argument and extracts its semantic and type-semantic data. This data is then utilized to translate the node's

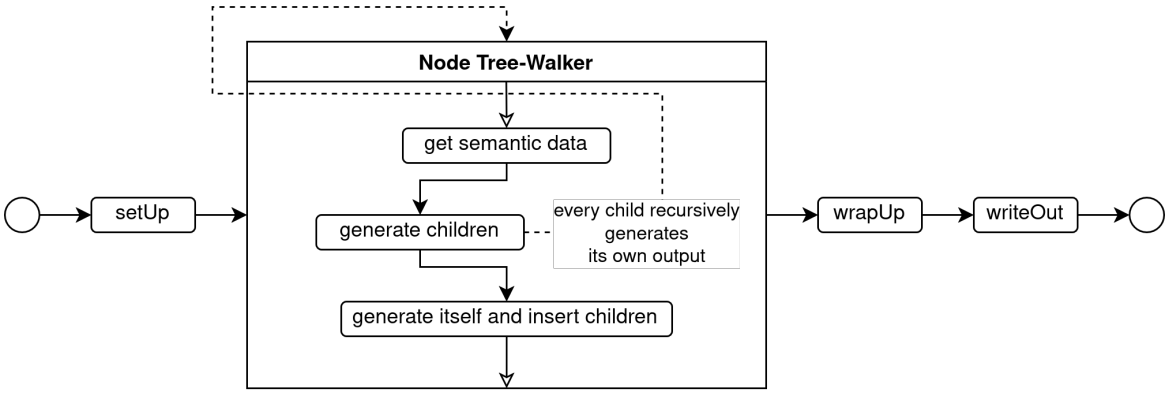


Figure 8: A simplified version of the tree-walker generation algorithm.

children—properties defined within the semantic data—into source code. The resulting code fragments are collected into a string array, concatenated, and returned as the final output for that node.

This approach ensures that each node is processed in a structured manner, maintaining consistency in code generation. Listing 24 demonstrates this process with an example of translating an `instanceOf` expression, illustrating how the compiler systematically generates code based on semantic analysis.

```
1 instanceOfExpression = async (node: InstanceOfExpression):
     ... => {
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 24: The code generation function of a `instanceOf` expression

The code generator functions for the individual nodes are implemented in the code generator class `JavaScriptTargetCodeGenerator`. Due to the similarity between TypeScript and JavaScript, the TypeScript code generator extends the JavaScript code generator and overrides the functions that are unique to TypeScript. This eliminates duplicate code fragments.

5.5.4 Differences between the Target Languages

The implementation of a compiler or transpiler targeting multiple programming languages often requires addressing the specific quirks and requirements of each target.

```
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 25: Reserved Keywords in TS and JS

As Kipper is a web development language, it targets both TypeScript and JavaScript. Although both languages share a common foundation, they diverge significantly in their syntax rules, semantics, and type systems.

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

Target Requirements & 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 example, `class` is a reserved keyword in both languages and cannot be used as a variable name. In contrast, TypeScript also reserves names such as `let`, `number`, and other type names, making them invalid as variable or function names.

Kipper addresses these reserved keywords by checking for them at compile time. The compiler compares the identifiers against a hard-coded list of keywords, and if a match is found, it throws a `ReservedIdentifierOverwriteError`. This list includes both the reserved words of JavaScript and TypeScript, which helps minimise redundancy and complexity. Furthermore, this approach encourages the use of sensible variable names, as JavaScript is relatively lenient regarding reserved keywords, as can be seen illustrated in Listing 25.

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.

Figures 9 and 10 shows the difference between the JavaScript code generator function and the TypeScript code generator function for variable assignments with the source example `var x: num = 5;` being translated. In the code block that generates TypeScript, there are additional code tokens after the storage and the identifier that insert the type of the object into the output code.

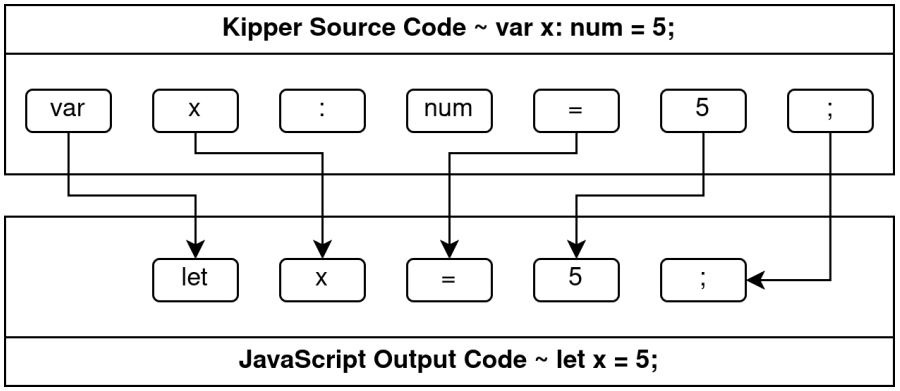


Figure 9: The simplified translation process of the example `var x: num = 5;` into JavaScript.

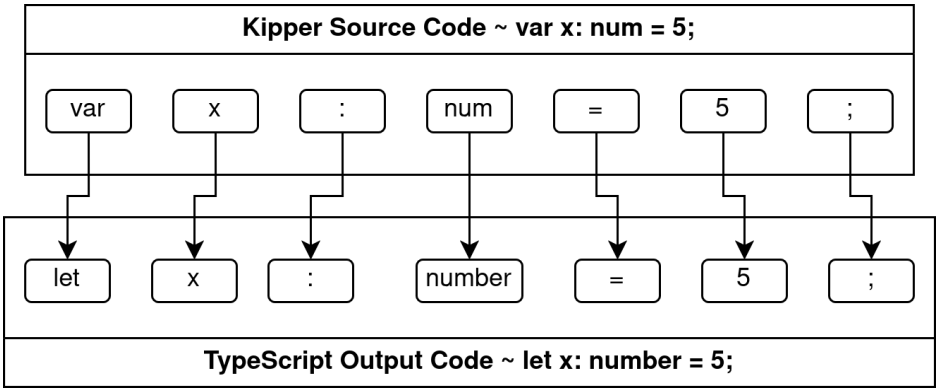


Figure 10: The simplified translation process of the example `var x: num = 5;` into TypeScript.

Other code generators such as the ones for function declarations and lambdas behave similarly when taking type annotations into account.

5.5.5 Target Requirements Generation & Optimizations

Kipper is designed to maintain a minimal runtime while ensuring all necessary functionality is included. This requires the compiler to incorporate only the functions and objects essential to the user's program. This approach aligns with the objective of keeping the compiler both modular and lightweight. By eliminating unused components through a process known as "tree-shaking," the resulting output code remains compact and efficient.

Kipper provides a range of built-in functions and features to support its runtime environment. These built-ins are structured using a scoped approach. The global scope includes core runtime features that are always required, such as fundamental type handling and error reporting. Beyond the global scope, additional features—such as helper functions specific to certain syntax, including `slice` and `match`—are selectively incorporated based on the specific requirements of the compiled program.

Conditional features

Conditional features are runtime components that are included only when explicitly required by the program being compiled. These features range from commonly used operations, such as `match` and `slice`, to more specialised or program-specific utilities.

Unlike essential runtime components contained within the global scope, conditional features are selectively incorporated based on an analysis of the program's structure and functionality. The inclusion of these features is determined during the requirements generation phase.

As the compiler traverses the AST, it analyses each node to determine whether a specific built-in function or runtime operation is invoked. If a feature such as `slice` is detected in the source code, it is marked as necessary and added to the consolidated requirements list. This approach ensures that only the relevant components are integrated into the final runtime environment.

An example of a conditional feature is the `slice` function, as demonstrated in Listing 26.

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

Listing 26: The Slice operator being used on a string

This function takes an array as input and extracts a section starting from the index specified by the first argument and ending at the index specified by the second argument.

When the compiler encounters the slice operator, it registers the function as necessary within the context of the program being compiled. The code generator called after the semantic analysis then includes it in the output code. This code generator for the example `slice` is demonstrated in Listing 27.

The compiler calls the slice function in the `BuiltInGenerator`. The abstract `BuiltInGenerator` class is a collection of functions responsible for generating the code for the built-in functions in a specific target language. This means that each target language must have an individual implementation for each built-in function of Kipper within its output generator, which subsequently adds the corresponding code into the resulting program.

The generator as demonstrated in Listing 27 generates the required JavaScript code for the function by utilizing the built-in `slice` function provided by standard 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 27: Slice in the JavaScript BuiltInGenerator

The `slice` function generated by the code generator as shown in Listing 27 is represented in Listing 28. This function will be executed at runtime and perform the required operation, in this case slicing the given argument.

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

Listing 28: Slice in the target language TypeScript

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 different runtime environments, such as 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 `__globalScope`, a standardized global object that provides a consistent way to access the global scope regardless of the environment. However, not all environments support `__globalScope`, which is why fallback mechanisms are often employed.

The Kipper global scope encompasses all the required runtime features. It is crucial that the global scope exists only once, which is why the program checks at runtime whether the scope has already been defined. This logic is demonstrated in Listing 29. The program first checks if `__globalScope` is 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 approach ensures that the `__globalScope` variable is always initialised, regardless of the environment, thereby enabling consistent and safe access to the global scope.

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

Listing 29: Global Scope Logic

Internal functions

Internal functions in Kipper serve as a crucial part of the runtime, but 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

within the user's program. This approach ensures that the runtime environment remains clean and minimal while still delivering the necessary functionality.

An example of an internal function is the `assignTypeMeta` function, which adds metadata to a runtime type. This function is essential for runtime type comparison, as discussed in detail in Section 5.6.4. Although this function is never exposed to the user, it is invoked internally when for example an interface or array is declared.

A key characteristic of internal functions is their dynamic inclusion in the runtime environment, as also previously discussed in Section 5.5.5. During the requirements generation phase, the compiler determines whether any internal mechanisms are required based on the program's functionality. If so, the corresponding internal functions are incorporated into the runtime.

5.5.6 Stylistic Choices

The syntax of Kipper is specifically designed to ease the transition of existing TypeScript and JavaScript developers to Kipper. Consequently, it was important to maintain output that closely resembles these languages as much as possible. This objective was achieved 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 TypeScript or JavaScript 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, although 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, which can hinder readability. While compression is useful in production environments to reduce payload size, it is opposed to the goals of Kipper, as it negatively 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 declaration keywords such as `let`, `const`, and `function` to discriminate between 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 only readable but also editable. Developers can treat the generated TypeScript/JavaScript 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, one of the most widely used JavaScript-to-JavaScript transpilers, produces highly optimized output for compatibility across different environments. However, this

optimization can result in dense and less human-readable code, making manual modifications more challenging. In contrast, Kipper prioritises maintainability over aggressive optimization, ensuring that the output remains clear, editable, and approachable for developers.

5.6 Integrated Runtime

5.6.1 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 30. 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.

```
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 30: Example of nominal typing in Java

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.

Structural Type Systems

Structural type systems compare types based on their structure. This means that if two differently named types have the same properties and methods, they are considered

equivalent. An example of this is OCaml, where its object subsystem follows structural typing. In OCaml, classes function solely as constructors for creating objects rather than defining rigid type hierarchies.

Listing 31 demonstrates a function that requires a `speak` method returning a `string`. Both the `dog` and `cat` objects satisfy this requirement, and as a result, they are treated as the same type. Most importantly, these compatibility checks occur at compile time, as OCaml is a statically typed language.

```
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 31: Example of structural typing in Ocaml

Structural typing enhances flexibility by promoting code reuse and eliminating the need for explicit inheritance hierarchies. This approach allows for greater modularity and adaptability in software design, as types are determined based on their capabilities rather than their declared names.

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 32, 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.

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

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

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

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

Listing 32: 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.

5.6.2 Runtime Type Concept in Kipper

As previously discussed (see Section **??**), the Kipper programming language employs a type system similar to TypeScript, featuring static typing and a duck-typing approach for complex data structures and OOP constructs. However, unlike TypeScript, Kipper enforces full type safety at runtime, requiring developers to explicitly specify types and handle edge cases such as type casts and inference.

This approach allows Kipper to support untyped values, similar to TypeScript's `any` type, which is often encountered in web requests or dynamically generated content. However, Kipper ensures that such values can be compared against well-defined types—such as primitives, arrays, functions, classes, and interfaces—eliminating ambiguities that could lead to runtime errors.

To implement this functionality, all user-defined interfaces are converted into runtime types during code generation. These runtime types store the necessary metadata to perform type checks, forming the basis for strict type safety in operations such as type casts, `matches`, and `typeof` evaluations. These runtime-defined interfaces work alongside built-in types such as `num`, `str`, and `obj`, allowing the compiler to insert necessary checks and references to ensure complete type safety at runtime.

With the exception of interfaces, classes, and generics, type identity is primarily determined by name. In these cases, type equality checks are performed using nominal typing, where the name acts as a unique identifier within the given scope (e.g., `num` is only assignable to `num`). For more complex structures, additional attributes—such as member properties or generic parameters—are also considered in the type-checking process.

For interfaces, type compatibility is determined based on the names and types of fields and methods. These elements define the minimum structure an object must implement to be considered assignable to a given interface. In this regard, Kipper follows the same duck-typing approach found in TypeScript.

For generics, including structures such as `Array<T>` and `Func<T..., R>`, assignability is determined based on both the generic identifier and the provided type parameters. This ensures that when a generic type is assigned to another, all parameters must match exactly. For instance, `Array<num>` is not assignable to `Array<str>` and vice versa, even if their overall structure appears identical.

For user-defined classes, the compiler relies on the prototype as a discriminator. This behavior aligns with that of primitive types, ensuring that distinct classes are not assignable to one another, preserving strict type integrity.

5.6.3 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 33.

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

```
class KipperType {
  constructor(name, fields, methods, baseType = undefined,
     customComparer = undefined) {
    this.name = name;
    this.fields = fields;
    this.methods = methods;
    this.baseType = baseType;
    this.customComparer = customComparer;
  }

  accepts(obj) {
    if (this === obj) return true;
    return obj instanceof KipperType && this.customComparer
       ? this.customComparer(this, obj) : false;
  }
}
```

Listing 33: The structure of a runtime type

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 type-checking 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.

5.6.4 Built-in Types for the Kipper Runtime

Built-in runtime types form the core of Kipper's type system and serve as building blocks for more complex constructs like interfaces. These types are uniquely defined at the beginning of the output code and are accessible globally, ensuring consistent type comparisons through reference equality.

As demonstrated in Listing 34, the fundamental built-in types such as `any`, `undefined`, and `str` are implemented during runtime. These types are instantiated as instances of the `KipperType` class, which provides essential metadata and comparison logic.

In addition to the core primitive types—such as `bool`, `str`, `num`, Kipper also includes built-in implementations for generic types such as `Array<T>` and `Func<T..., R>`. These

```
const __type_any =
new KipperType("any", undefined, undefined);

const __type_undefined =
new KipperType("undefined", undefined, undefined, undefined,
    (a, b) => a.name === b.name);

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

Listing 34: Examples for the built-in runtime types

generic types define their parameters, which typically default to `any`, as demonstrated in Listing 35.

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

Listing 35: Generic built-in types

As demonstrated in Listing 35, generic types in Kipper are implemented using a specialized class called `KipperGenericType`. This class, detailed in Listing 36, extends `KipperType` and introduces an additional field for storing generic arguments.

A crucial feature of `KipperGenericType` is the `changeGenericTypeArguments` method, which allows for the dynamic modification of a type's generic arguments at runtime. This functionality is essential for handling lambda functions and arrays, where a built-in generic type serves as the base and is then adjusted to match the specific type parameters provided by the user.

For instance, when an array is initialized, it is first assigned the default runtime type `Array<any>`. The `changeGenericTypeArguments` method is subsequently invoked to refine this type, converting it into a more specific type like `Array<num>`. This ensures that arrays maintain strong type consistency while allowing flexibility in their definitions.

Similarly, functions utilize this mechanism to define both their return type and argument types dynamically. The `Func<T..., R>` type plays a key role in lambda expressions, where user-defined functions without explicit names must still adhere to a strict type structure. By leveraging runtime type modifications, Kipper guarantees that generic constructs maintain type safety without compromising flexibility.

```
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 36: Generic Kipper Type

5.6.5 Runtime Errors

Other built-ins include error classes, which are used in the error handling system to represent runtime errors caused by invalid user operations. The base `KipperError` type has a name property and extends the target language's error type as can be seen in Listing 37.

Additional error types inherit this base type and extend it with additional error information. For example, the `KipperIndexError` is used when an index is out of bounds, while the `KipperTypeError` is thrown when a `force as` cast fails (see Section 4.8.2 for more details on force casts).

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

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

Listing 37: Kipper error types

5.6.6 Runtime Generation for Interfaces

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

Now take for example the given interfaces presented in Listing 38 and 39.

```
1 interface Car {
2   brand: str;
3   honk(volume: num): void;
4   year: num;
5 }
```

Listing 38: Example interface `Car` in the Kipper language

```
1 interface Person {
2   name: str;
3   age: num;
4   car: Car;
5 }
```

Listing 39: Example interface `Person` in the Kipper language including a reference to a different interface

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.

Translating the interface `Car` shown in Listing 38 would result in an output runtime code identical to that in Listing 40 and translating `Person` would result in the code as presented in Listing 41.

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

Listing 40: The runtime representation of the example interface `Car`

```
const __intf_Person = new __kipper.Type(
  "Person",
  [
    new __kipper.Property("name", __kipper.builtIn.str),
    new __kipper.Property("age", __kipper.builtIn.num),
    new __kipper.Property("car", __intf_Car),
  ],
  []
);
```

Listing 41: The runtime representation of the example interface `Person`

As shown in Listing 40 and Listing 41, 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 5.6.7) and the `typeof` operator (see Section 5.6.8).

5.6.7 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 42.

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

Listing 42: The Kipper matches operator

As demonstrated in Listing 42, the matches operator enables interface comparison based on properties and methods. It accepts two arguments: an object and a target type to match against. Properties are compared recursively, while methods are compared by name, arguments, and return type.

The comparison process involves iterating over both methods and properties. When examining properties, the algorithm verifies the presence of each property name within the target type, disregarding the order of properties. Once a matching name is found, type equality is checked using the previously discussed runtime types and nominal type comparison. If a property is of a non-primitive type, the matches function is recursively applied to ensure structural compatibility.

This property matching algorithm is implemented as illustrated in Figure 11.

After verifying the properties, the matches expression iterates over the methods, as illustrated in Figure 12. The process begins by searching for the method name within the target type. If a match is found, the return type is compared. Subsequently, each argument is examined by name, ensuring that the method signatures are identical. Additionally, the number of parameters is compared to enforce exact correspondence.

If none of these conditions evaluate to false, the input object is deemed compatible with the input type.

5.6.8 Typeof Operator

In the Kipper programming language, the `typeof` operator is used to obtain the type of an object at runtime. This operator can be used to check whether a variable or expression is of a particular type, such as a string, number, boolean, etc. It is most commonly used to check for `null` and `undefined` objects, in order to avoid type errors when the type of an object is unknown. The returned type object can be compared by reference to check for type equality. As shown in Listing 43, the parentheses are optional. Both syntax styles are supported to align with Kipper's design goal of being similar to TypeScript and JavaScript, which implement it in the same way.

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

Listing 43: 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 `typeof null` returns `object` due to historical reasons, Kipper correctly identifies `null` as `__kipper.builtIn.null`.

At runtime, the provided object is checked for its type using the target language's type features. A part of this process can be seen in Figure 13. The primitive types return their respective `KipperRuntimeType`. Objects are a special case, as they can in JavaScript either be `null`, an array, a class, or an object, such as one that implements an interface.

Although syntactically similar, the `typeof` operator in the type declaration of a variable operates fundamentally differently, as demonstrated in Listing 44. This operator is referred to as `TypeOfTypeSpecifier` and evaluates the type of a variable at compile time.

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

Listing 44: Specifying the type based on a reference variable

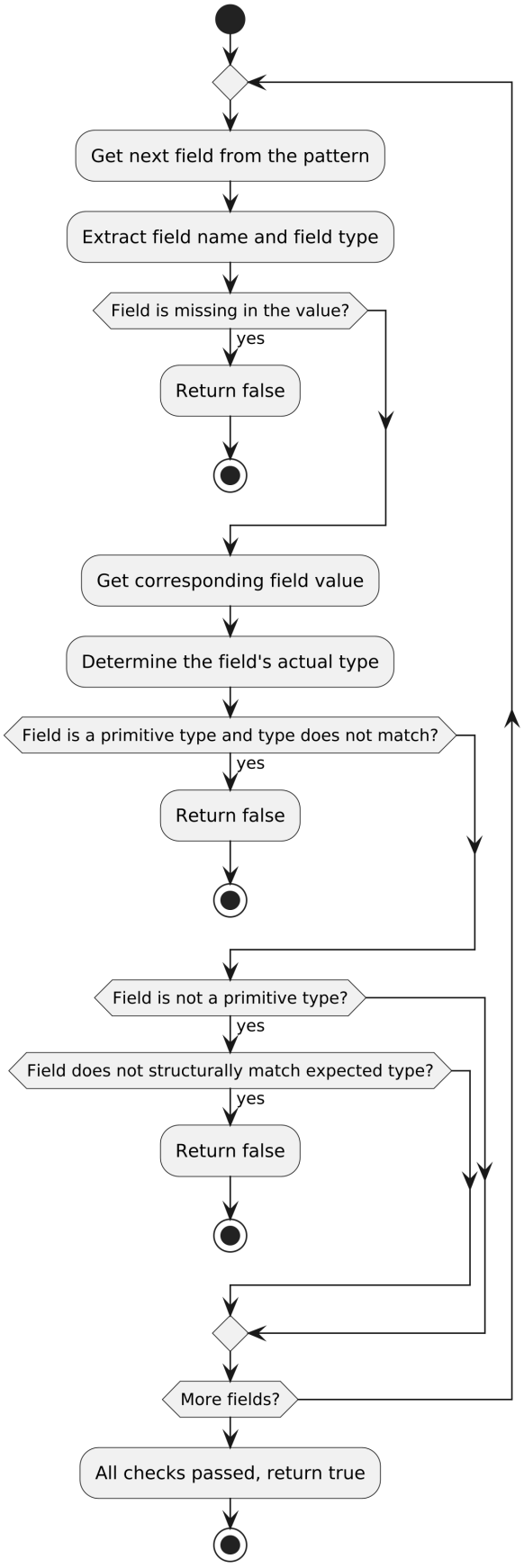


Figure 11: Matches operator property comparison.

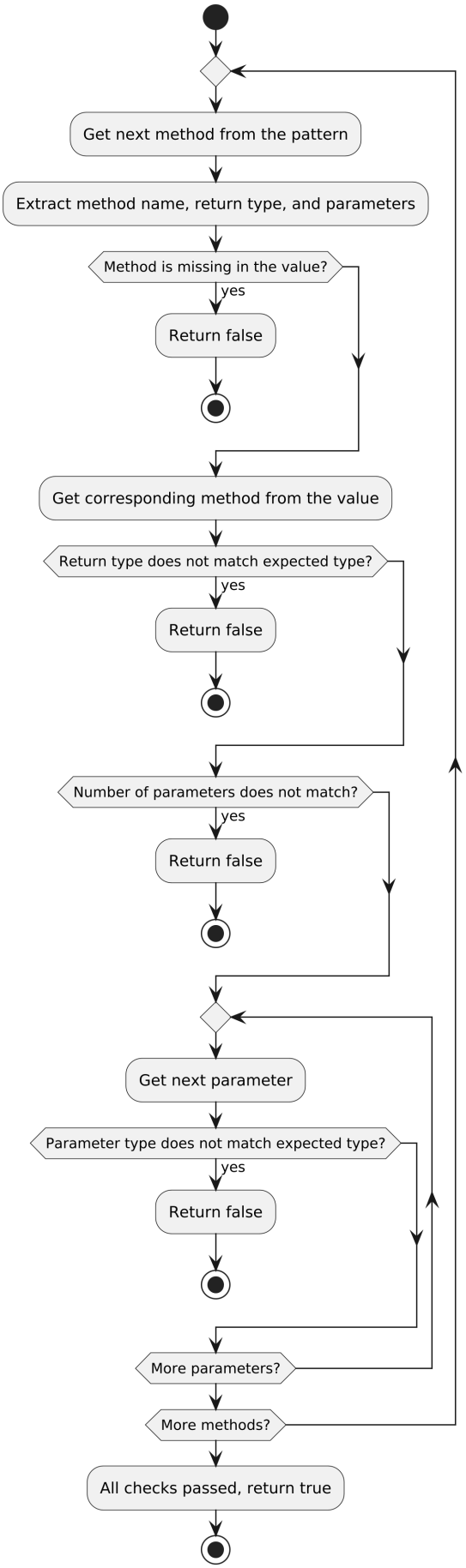


Figure 12: Matches operator method comparison.

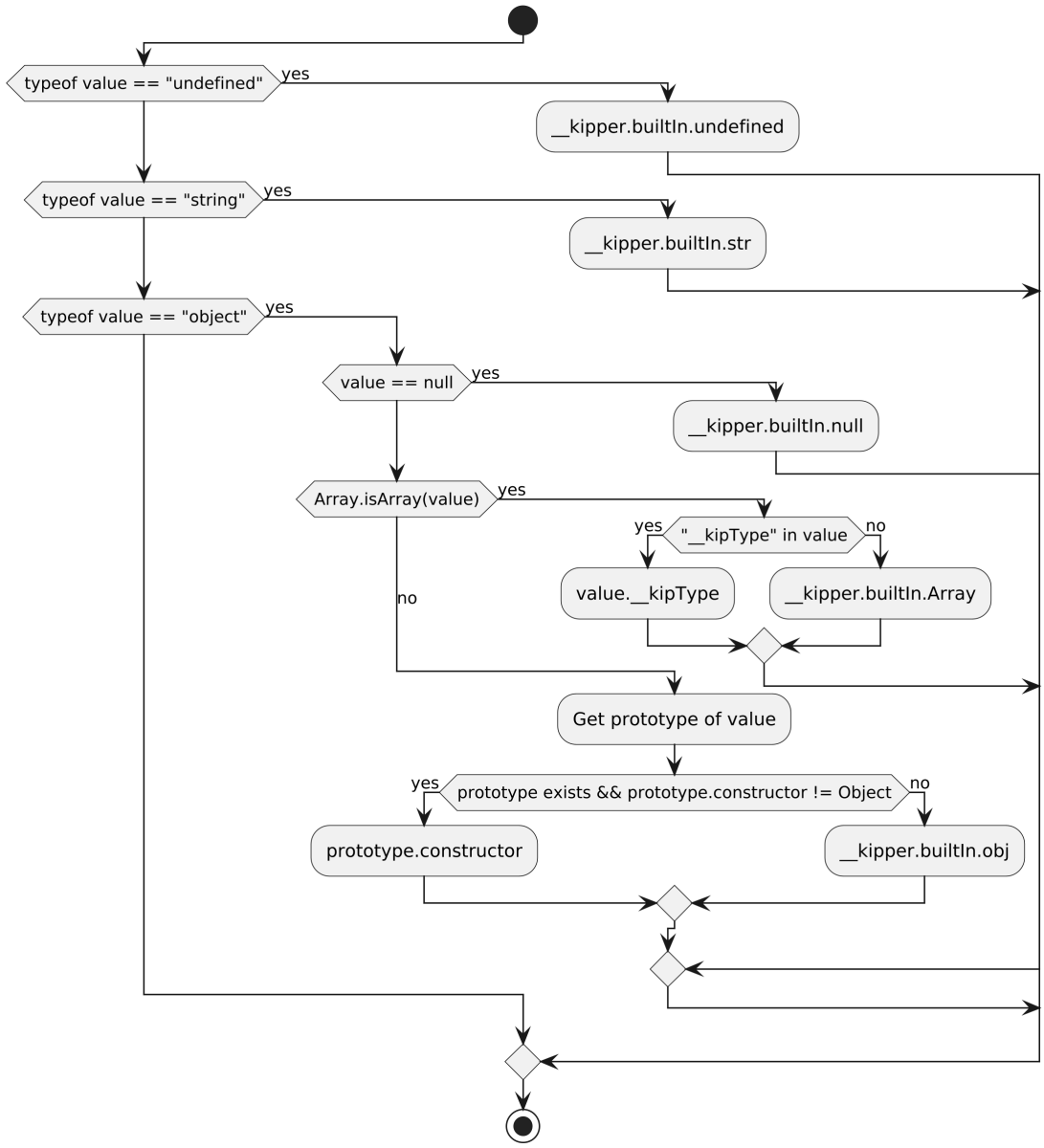


Figure 13: Logical implementation of the `typeof` operator in the target JavaScript/-TypeScript environment.

6 Compiler Reference

This chapter explores the various ways to interact with the Kipper compiler and its systems through the exposed Kipper API. Since the compiler serves as the core of the project, utilizing the language effectively requires setting up the compiler and integrating it properly for specific use cases.

The API provides all the essential functionality needed to compile and transpile Kipper code, enabling developers to integrate it into their workflows. Additionally, the API exposes core interfaces, classes, and types that allow users to extend and modify the compiler's behaviour. This means developers can inherit from existing components or introduce custom modifications to tailor the compilation process to their needs.

6.1 Compiler API

The Kipper Compiler API provides the functionality required to compile Kipper source code into JavaScript or TypeScript. The primary entry points into the API are the `KipperCompiler` class, which offers core compilation functionality and facilitates file processing, and the `KipperProgramContext` class, which stores contextual information about a performed compilation and can be referenced for additional program metadata.

6.1.1 Initializing the Compiler

To compile Kipper code, an instance of `KipperCompiler` must be created. The compiler requires a logger instance and an optional configuration object.

```
1 import * as kipper from "@kipper/core";
2
3 const logger = new kipper.KipperLogger((level, msg) => {
4   console.log(`[${Kipper.getLogLevelString(level)}] ${msg}`);
5 });
6
7 const compiler = new kipper.KipperCompiler(logger, {});
```

Listing 45: Initializing the Kipper Compiler

The logger instance provides error reporting functionality, making it an essential part of the compilation process.

6.1.2 Compiling Kipper Code

Once the compiler is initialized, it can be used to transpile Kipper code into JavaScript or TypeScript. The `compile` method takes the source code as a string and a configuration object specifying the compilation target.

```
1  import * as kipperJS from "@kipper/target-js";
2
3  const result = await compiler.compile(
4    `print("Hello world!");`,
5    { target: new kipperJS.TargetJS() }
6  );
7  const jsCode = result.write();
8
9  // Execute the compiled JavaScript
10 eval(jsCode);
```

Listing 46: Compiling Kipper Code to JavaScript

The compilation result is an object containing the generated JavaScript or TypeScript code, which can be executed or written to a file.

6.1.3 Compilation Options

The `compile` method accepts a configuration object that allows customization of the compilation process. The most important options are:

- `target`: Specifies the output language. Available targets are `TargetJS` for JavaScript and `TargetTS` for TypeScript.
- `filename`: Specifies the filename of the generated code.
- `optimisationOptions`: Currently `optimiseInternals` and `optimiseBuiltIns` are available. They can be enabled by setting them to `true`.

6.1.4 Accessing compilation metadata and potential errors

The Kipper compiler returns upon the execution of `compile` a result object storing all compilation-related information. This instance of the class `KipperCompileResult` represents the result of the compilation and depending on the success of the compilation will contain various program metadata.

The following properties and methods are exposed by `KipperCompileResult`:

- `programCtx`: An instance of `KipperProgramContext` stores all essential metadata of the program, including the parse tree, AST, global symbol table, internal references, compiler arguments, and other relevant information. It contains everything generated by the compiler. However, it may be `undefined` if syntax errors occur during compilation, preventing the construction of an AST and any subsequent analysis.
- `result`: The result of the compilation represented as an array containing string arrays, where each string array represents the individual lines and their corresponding tokens.
- `success`: A boolean field indicating the success of the compilation.
- `warnings`: The list of warnings reported by the compiler.
- `errors`: The list of encountered syntax errors or semantic errors.
- `write(lineEnding: string = "\n")`: Writes the whole translated program to a string.

6.2 Target API

The Kipper compiler supports multiple compilation targets, allowing the transpilation of Kipper code into different languages. Developers can extend Kipper by implementing custom targets.

6.2.1 Using Predefined Targets

Kipper provides predefined targets for JavaScript and TypeScript. These can be used as follows:

```
1 import * as kipperJS from "@kipper/target-js";
2 import * as kipperTS from "@kipper/target-js";
3
4 const jsTarget = new kipperJS.TargetJS();
5 const tsTarget = new kipperTS.TargetTS();
```

Listing 47: Using Compilation Targets

6.2.2 Creating Custom Targets

To create a custom target, a new class must extend `KipperCompileTarget` and implement the three required classes for translation. As demonstrated in Listing 48, the three classes `CustomTargetSemanticAnalyser`, `CustomTargetCodeGenerator`, and `CustomTargetBuiltInGenerator` each define the various functionality required for translating a program. These classes are grouped within the target class `CustomTarget`, which can then be passed to a compilation.

```
1  import { KipperCompileTarget, KipperTargetBuiltInGenerator,
       TranslatedExpression } from "@kipper/core";
2
3  export class CustomTargetSemanticAnalyser extends
       KipperTargetSemanticAnalyser {
4    ...
5  }
6
7  export class CustomTargetCodeGenerator extends
       KipperTargetCodeGenerator {
8    ...
9  }
10
11 export class CustomTargetBuiltInGenerator extends
       KipperTargetBuiltInGenerator {
12   ...
13 }
14
15 class CustomTarget extends KipperCompileTarget {
16   constructor() {
17     super(
18       "MyLanguage",  // name
19       new CustomTargetSemanticAnalyser(),
20       new CustomTargetCodeGenerator(),
21       new CustomTargetBuiltInGenerator(),
22       "ext" // file extension
23     );
24   }
25 }
```

Listing 48: Creating a Custom Compilation Target

Creating a custom target requires the implementation of the translation for all Kipper AST nodes and the definition of the required built-in functions. Additionally, custom semantic checks can be implemented if specific requirements apply to a given target.

6.3 Command Line Interface (CLI)

The Kipper CLI provides a command-line interface for compiling Kipper code and managing a project within a local system directory. The CLI is available via the `@kipper/cli` package.

6.3.1 Installing the CLI

To install the Kipper CLI using the JavaScript package manager npm, use the following command:

```
1 npm install @kipper/cli
```

Listing 49: Installing Kipper CLI

This installs Kipper in the current npm project environment.

To install Kipper globally, use the `-g` flag. This may require superuser privileges, depending on the installation location.

6.3.2 Compiling a File

To compile a Kipper source file, use the `kipper compile` command:

```
1 kipper compile source.kip --target js
```

Listing 50: Compiling a Kipper file

This command transpiles `source.kip` to JavaScript. It is possible to specify the option `--target=ts` to switch to the TypeScript target. When no target option is specified, the default target is JavaScript.

6.3.3 Running a Kipper Program

To run a Kipper source file, use the `kipper run` command:

```
1 kipper run source.kip
```

Listing 51: Running a Kipper file

This command transpiles to the specified target and then executes the code.

6.3.4 Creating a New Project

To create a new Kipper project, use:

```
1 kipper new my_project
```

Listing 52: Creating a new Kipper project with a default file setup and config file

This generates a new project with a configuration file that specifies the compiler arguments, which are automatically passed to the compiler upon invocation.

7 Demo & Showcase

This chapter explores various sample use cases for Kipper to showcase its capabilities and demonstrate the environments in which it can be utilized. To keep things concise and focused, the examples provided will cover simple scenarios. More complex applications are beyond the scope of this paper, but can be explored in further documentation (see **??** for further details).

7.1 Example Program

Listing 53 demonstrates the computation of the prime factors of a given number using Kipper. The program is given a number, determines the prime factors and prints the result.

```
1  def primeFactors(n: num) -> void {
2    while (n % 2 == 0) {
3      print(2);
4      n /= 2;
5    }
6
7    for (var i: num = 3; i * i <= n; i += 2) {
8      while (n % i == 0) {
9        print(i);
10       n /= i;
11     }
12   }
13
14   if (n > 2) {
15     print(n);
16   }
17 }
18
19 primeFactors(18);
```

Listing 53: A basic programs that determines the prime factors of an integer

The given example in Listing 53 will be used as the sample code in the following chapters.

7.2 Working example using CLI

As previously explained in Section 6.3, Kipper provides a command line interface (CLI) that allows users to compile and execute programs directly from the terminal. To run the above example using the CLI, the following steps are required:

1. Install the Kipper CLI globally using the JavaScript package manager npm: `npm install -g @kipper/cli`
2. Save the Kipper source code in a file, e.g. `prime.kip`.
3. Compile and execute a Kipper file: `kipper run prime.kip`

After execution, the prime factors of the provided number should be printed to the console.

7.3 Working example in the web

Kipper can also be run in the browser using the `@kipper/web` package. Listing 54 demonstrates how to include Kipper in an HTML file and execute the same prime factorization program.

```
<!DOCTYPE html>
<head>
  <title>Kipper Demo</title>
  <script
     src="https://cdn.jsdelivr.net/npm/@kipper/web@latest
  /kipper-standalone.min.js"></script>
</head>
<body>
  <script type="module">
    const logger = new Kipper.KipperLogger((level, msg) => {
      console.log(`[${Kipper.getLogLevelString(level)}]
         ${msg}`);
    });
    const compiler = new Kipper.KipperCompiler(logger, {});
    const code = ''; // Replace with prime.kip

    compiler.compile(
      code,
      { target: new KipperJS.TargetJS() }
    ).then((result) => {
      const jsCode = result.write();
      eval(jsCode);
    });
  </script>
</body>
```

Listing 54: Running Kipper in the browser

When opened in a browser, this file will execute the compiled JavaScript version of the prime factorization program, displaying the results in the browser console.

7.4 Working example using Node.js

Kipper can also be utilized and run within a Node.js JavaScript environment.

The following steps outline how to run a Kipper program using Node.js:

1. Install the Kipper runtime package via a package manager like npm: `npm install @kipper/core`
2. Save the Kipper source code in a file, e.g. `prime.kip`.
3. Use a simple Node.js script to compile and execute the Kipper code.

Executing a Kipper program within Node.js works similarly to the browser, but requires imports from the various Kipper packages. As demonstrated in Listing 55, the compiler can be simply imported from `@kipper/core` and then run with the read file.

```
1  import { promises as fs } from "fs";
2  import * as kipper from "@kipper/core";
3  import * as kipperJS from "@kipper/target-js";
4
5  const path = "prime.kip";
6  fs.readFile(path, "utf8").then(async (fileContent) => {
7    const logger = new kipper.KipperLogger((level, msg) => {
8      console.log(`[${level}] ${msg}`);
9    });
10   const compiler = new kipper.KipperCompiler(logger);
11
12   let result = await compiler.compile(fileContent, {
13     target: new kipperJS.TargetJS(),
14   });
15   let jsCode = result.write();
16
17   eval(jsCode);
18 });
```

Listing 55: Executing Kipper code in Node.js

Executing this script will transpile the Kipper code to JavaScript and run it within Node.js, printing the prime factors to the console.

8 Conclusion & Future

8.1 Potential Future 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.

8.1.1 WebAssembly Support

WASM (WebAssembly) is a low-level, binary instruction format designed as a portable compilation target for high-performance applications [15]. It allows code to run efficiently across diverse execution environments, including web browsers, cloud services, and embedded systems. Given the increasing adoption of WebAssembly in cloud and server-side applications, extending Kipper to support WASM as a compilation target presents an opportunity to enhance performance, portability, and deployment flexibility.

Incorporating WASM as a target for Kipper introduces multiple advantages. The potential for improved execution speed is one of the most significant benefits, as WebAssembly executes more efficiently than JavaScript due to its statically typed nature and optimized memory handling. Unlike JavaScript and TypeScript, which depend on runtime interpretation, WebAssembly enables Kipper programs to be compiled ahead of time, resulting in lower runtime overhead and faster execution.

Additionally, WASM facilitates the development of standalone applications, making it possible for Kipper programs to run independently of JavaScript runtime environments such as NodeJS. This characteristic is particularly beneficial for server-side applications and microservices. Furthermore, due to its compact binary format, WebAssembly modules require less memory compared to traditional virtualized environments, thereby reducing startup time and improving efficiency. Technologies like Wasmer demonstrate that WebAssembly can function as a lightweight alternative to traditional containerized environments while maintaining strong isolation and security properties [16].

Despite these advantages, integrating WASM into Kipper presents several technical challenges. A major hurdle is memory management, as WASM does not include built-in garbage collection. While experimental support for garbage collection in WebAssembly is under development, Kipper's runtime model would require significant adjustments to manage memory allocation and deallocation efficiently. Furthermore, integrating WebAssembly as a compilation target requires a well-defined mechanism for interoperability with JavaScript, ensuring that Kipper-generated Wasm modules can interface seamlessly with JavaScript-based systems. The Kipper standard library, which currently relies on JavaScript APIs, would also need extensive modifications to accommodate WebAssembly's execution model.

Given the complexity of these challenges, full WebAssembly support for Kipper is not planned for the foreseeable future. The significant technical hurdles associated with type system modifications, memory management, and standard library adaptation make WebAssembly integration infeasible at this stage of Kipper's development. While an initial proof of concept could theoretically evaluate the feasibility of compiling simple Kipper functions to WebAssembly, the extensive modifications required for full compatibility outweigh the potential benefits at this time. As a result, Kipper will remain focused on its existing compilation targets, prioritizing improvements and optimizations within the JavaScript and TypeScript ecosystems.

8.1.2 IDE and Code Editor Language Plugins

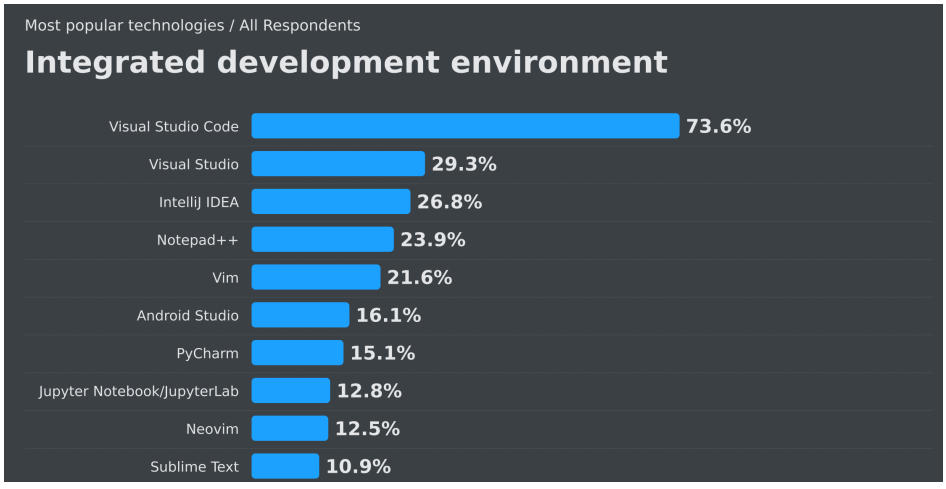
The integration of a programming language into code editors and IDEs plays a fundamental role in its adoption and usability. Effective language support facilitates code writing, debugging, and maintenance by providing features such as syntax highlighting, code completion, error detection, and debugging tools. Given that Kipper transpiles to TypeScript (TS) and JavaScript (JS), ensuring comprehensive IDE support is essential for practical usability.

A primary component of language support is syntax highlighting. By distinguishing keywords, data types, function names, and operators through distinct visual cues, syntax highlighting improves code readability and reduces cognitive effort for developers. While Kipper currently provides syntax highlighted editing in its web editor (See Appendix A for info on the Kipper documentation and website), expanding this feature to widely used IDEs such as Visual Studio Code (VS Code) and IntelliJ IDEA would increase accessibility and efficiency.

Additionally, a standardized code formatter is required to ensure consistency across Kipper projects. Existing formatting tools such as Prettier and ESLint are widely adopted in TS and JS environments. A Kipper-specific formatter, following established formatting conventions, would improve code maintainability.

Modern IDEs incorporate intelligent code assistance features, commonly referred to as IntelliSense in VS Code. These features include code completion, which suggests function names, variables, and methods dynamically based on context. Parameter hints display expected function parameters to assist in correct API usage. Hover documentation provides inline explanations for functions, types, and keywords, enhancing code comprehension. Finally, symbol navigation enables quick access to function definitions, imports, and variable declarations. To enable these features, a Language Server Protocol (LSP) implementation specific to Kipper is necessary, allowing real-time context-aware suggestions and streamlining the development workflow.

Static analysis tools contribute to code quality by identifying potential issues and enforcing best practices. A dedicated Kipper linter could detect syntax errors prior to transpilation, warn about potential runtime issues, provide recommendations for writing idiomatic Kipper code, and enforce coding conventions such as indentation styles and variable naming patterns. A rule-based linter, similar to ESLint for JavaScript, would facilitate maintaining high-quality code and ensuring adherence to standardized practices.



Source: https://survey.stackoverflow.co

Figure 14: The most popular integrated development environments (IDEs) among developers, according to the 2024 Stack Overflow Developer Survey [17]. Visual Studio Code is the highest ranking with 73.6% usage, followed by Visual Studio and IntelliJ IDEA. All entries with less than 10% popularity have been cut off, but can be viewed at the survey source.

A recent developer survey as can be seen in Figure 14 indicates that Visual Studio Code is the most widely used IDE, with approximately twice as many users as Visual Studio, which holds the second position [17]. IntelliJ IDEA ranks third in popularity. The preference for VS Code can be attributed to its lightweight nature, extensive extension marketplace, and strong integration with JavaScript and TypeScript ecosystems. This data supports the decision to prioritize VS Code support in the initial phases of Kipper's IDE integration strategy.

8.2 Integration with other languages

Kipper is not intended to function as a standalone language but is designed to integrate into modern web and server-side environments. By enabling developers to import and export generated modules in their preferred language—either JavaScript or TypeScript—the Kipper compiler allows for gradual adoption without necessitating a complete system overhaul or exclusive reliance on Kipper-generated code. This process also works the other way around, enabling libraries and existing code bases to be imported and utilised within Kipper projects. This would be supported through external module imports and a compatibility layer that adapts imports to Kipper's runtime features and type safety standards.

Due to its complexity, this functionality was not included in the scope of this paper but could be incrementally implemented once Kipper's core feature set is finalised and stabilised. By integrating import wrappers and code loaders into the Kipper compiler—capable of verifying the existence and integrity of external variables, functions, interfaces and classes—the compiler could enable seamless interoperability without compromising safety or disabling essential functionality to achieve direct one-to-one compatibility.

8.3 Project Result & Moving forward

As of the time of writing, Kipper has met the initially defined objectives of this thesis but has not yet achieved a stable feature set or reached a v1.0 release. However, with the functionality outlined in this paper now implemented, the project will advance by refining and expanding upon the existing features to develop a language suitable for various scenarios and capable of bidirectional integration with its target environments.

While this thesis does not serve merely as a proof of concept, Kipper remains a closed-off project at this stage, lacking proper support for additional environments and much of the functionality expected from a fully developed modern programming language. However, the language will continue to be developed and extended with the long-term goal of establishing itself as a reliable programming language.

Furthermore, as an open-source project focused on all-round type safety and leveraging both transpilation and the dynamic nature of an interpreter runtime, Kipper will continue to evolve and expand its feature set over the coming years, hoping to build support and grow as a project that can be utilised by a variety of developers.

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 [4].
Backus–Naur Form	A formal notation used to define the syntax of programming languages, data formats, and other structured information. It represents rules through production expressions, where symbols define how components are composed. BNF uses non-terminal symbols, terminal symbols, and recursive definitions to describe complex language structures concisely and precisely. Most programming languages have their own syntax grammar also publically available in BNF.
Basic Multilingual Plane	The Basic Multilingual Plane (BMP) is the first plane of the Unicode character set, also known as Plane 0. It consists of 65,536 (or 2^{16}) code points, ranging from U+0000 to U+FFFF. The BMP includes characters used in most modern languages, as well as special symbols, punctuation marks and diacritical signs.

Bun	Bun is a cross-platform JavaScript runtime that is based on the WebKit JavaScriptCore engine and allows the local execution of code on a desktop or server without the need of a browser. It is a drop-in replacement for Node.js and is implemented in the Zig programming language aiming to improve performance and efficiency compared to its Node.js counterpart [18].
Deno	Deno is a cross-platform JavaScript runtime and package manager that is based on the V8 JavaScript engine and allows the local execution of code on a desktop or server without the need of a browser. It is often perceived as an alternative to Node.js and has many similar features, but provides its own unique system APIs, libraries and package management system [19].
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.

Just-in-Time compilation	Just-in-Time compilation is a technique in which source code or bytecode is translated into machine code during program execution and immediately executed. By performing compilation at runtime, JIT compilers can apply optimisations based on information gathered during execution, such as identifying and removing redundant code.
Node.js	Node.js is a cross-platform JavaScript runtime that is based on the V8 JavaScript engine and allows the local execution of code on a desktop or server without the need of a browser. It provides various system APIs and libraries to interact with the system underneith unlike standard browser-based JavaScript engines which are locked-in in their browser environment [20].
Object Oriented Programming	A programming design philosophy based on objects, which encapsulate data in fields and behaviors in methods. OOP promotes concepts like inheritance, Polymorphism, encapsulation, and abstraction to model real-world entities, improve code modularity, and encourage reusability.
Polymorphism	A core concept in Object Oriented Programming that allows the same function, method, or operation to behave differently depending on context. It can occur through inheritance (where subclasses override parent methods), function overloading (methods with the same name but different parameters), or generic typing (writing flexible code that works with multiple data types).
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.
Transpile	To perform transpilation on code.

Transpiler	A type of compiler performing transpilation.
WebAssembly	WebAssembly is a binary instruction format designed for efficient code execution on web browsers. WebAssembly can be generated from C, C++, and Rust, running at near-native speed. It allows building applications with performance-intensive tasks while maintaining portability and security across platforms.

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

[5] Hanspeter Mössenböck, Markus Löberbauer and Albrecht Wöß. (2025) The compiler generator coco/r. [Online]. Available: https://ssw.jku.at/Research/Projects/Coco

[6] Hanspeter Mössenböck, *The Compiler Generator Coco/R*, 2010. [Online]. Available: https://ssw.jku.at/Research/Projects/Coco/Doc/UserManual.pdf

[7] ——, *How to Build Abstract Syntax Trees with Coco/R*, September 2011. [Online]. Available: https://ssw.jku.at/Research/Projects/Coco/Doc/AST.pdf

[8] Free Software Foundation, Inc. (2025) Bison. [Online]. Available: https://www.gnu.org/software/bison/manual/bison.html

[9] ——, *Bison*, September 2021. [Online]. Available: https://www.gnu.org/software/bison/manual/bison.html

[10] The Flex Project. (2025) flex. [Online]. Available: https://github.com/westes/flex

[11] ——, "String - javascript," 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

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

[13] ——. (2024) Static single-assignment form. [Online]. Available: https://en.wikipedia.org/wiki/Static_single-assignment_form

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

[15] World Wide Web Consortium (W3C). (2025) WebAssembly - WASM. [Online]. Available: https://webassembly.org

[16] Wasmer Project Team. (2025) Wasmer: Webassembly containers that run everywhere. [Online]. Available: https://wasmer.io

[17] Stack Exchange, Inc. (2024) 2024 developer survey. [Online]. Available: https://survey.stackoverflow.co/2024/technology#1-integrated-development-environment

[18] Oven. (2025) Bun — a fast all-in-one javascript runtime. [Online]. Available: https://bun.sh

[19] Deno Land, Inc. (2025) Deno, the next-generation javascript runtime. [Online]. Available: https://deno.com

[20] OpenJS Foundation. (2025) Node.js — run javascript everywhere. [Online]. Available: https://nodejs.org

List of Figures

List of Tables

List of Source Code Snippets

Appendix A

The Kipper programming language has an official project website that provides a detailed changelog and comprehensive documentation on its individual features available at https://kipper-lang.org. This documentation offers an in-depth explanation of the language's specifications and usage details. Unlike this paper, which presents only key points and summaries of the main features, the Kipper documentation explores the language at a fundamental level and should be referenced for practical usage details.

It is important to emphasise that this thesis was not intended as a user manual or setup guide for Kipper. Instead, it focuses on the theoretical and implementation aspects of the project, highlighting the key concepts essential to understanding Kipper's design and development. Any details not covered in this paper regarding specific language features should be consulted in the official documentation, as they fall beyond the scope of this work.

Additionally, for minor programs or test cases, the official Kipper project website features an online compiler and editor that allows users to execute single-file Kipper programs directly in the browser. It should be used for minor programs, general experiments with the language and showcasing the functionality of Kipper.