FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Math expression evaluator for literal types in TypeScript
Student:	Bc. Tat Dat Duong
Supervisor:	Ing. Jaroslav Šmolík
Study program:	Informatics
Branch / specialization:	Web Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2023/2024

Instructions

Template literal types [1], introduced in TypeScript 4.1, expand on string literal types for narrowing down a type to a particular string constant, with the ability to expand into many string literal types.

1. Analyze and describe relevant constructs of the TypeScript type system (concatenation, recursive types, conditional types etc.)
2. Implement a typesafe math expression evaluator with a set of basic operations, using a string literal type both as the input and output of the evaluator.
3. Pick appropriate tools for testing type annotations and ensure the validity of the evaluator with functional tests.
4. Discuss the practical uses of implemented meta types and theoretical and practical shortcomings of the TypeScript type system.
5. Publish the implementation as an open-source TypeScript library, which can be used for meta-programming, including source code and corresponding documentation.

[1] https://www.typescriptlang.org/docs/handbook/2/template-literal-types.html

Electronically approved by Ing. Jaroslav Kuchař, Ph.D. on 6 January 2023 in Prague.

Master's thesis

MATH EXPRESSION EVALUATOR FOR LITERAL TYPES IN TYPESCRIPT

Bc. Tat Dat Duong

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Jaroslav Šmolík
May 1, 2023

Citation of this thesis: Duong Tat Dat. *Math expression evaluator for literal types in TypeScript.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

List of Figures

List of Tables

List of Listings

I would like to thank Ing. Jaroslav Šmolík, my supervisor, for all the help, guidance and advice during the work on this thesis. I also want to thank my family and friends for their support and patience. Lastly, I would like to thank Markéta, my girlfriend, for her boundless emotional support and affection.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the “Work”), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on May 1, 2023 ....

Abstract

This thesis presents an implementation of a library for evaluating mathematical expressions in TypeScript. The main goal of this thesis is the implementation of the core mathematical operations and the accompanying evaluator. This work also covers other aspects of the implementation: limitations of the TypeScript type system, unit testing for correctness and measuring the impact of the library on the developer experience.

Keywords TypeScript, static typing, type system, type-level metaprogramming, mathematical expressions, LL(1) parser, template literal types

Abstrakt

Tato práce se věnuje tvorbě knihovny pro vyhodnocování matematických výrazů v jazyce TypeScript. Hlavním cílem teto práce je implementace samotných matematických operací a příslušného vyhodnocovače. Tato práce se také zabývá dalšími aspekty implementace: limitace dané typovým systémem, testování korektnosti a měření dopadu knihovny na vývojářské prostředí.

Klíčová slova TypeScript, statické typování, typový systém, metaprogramování na typové úrovni, matematické výrazy, LL(1) parser, šablonové literálové typy

Chapter 1
Introduction

1.1 Motivation

TypeScript, a typed superset of JavaScript, is quickly gaining popularity in the JavaScript development ecosystem, and type-safety, the concept of validating data types, is “eating the world”[1]. As of 2023, over 66% of developers are using TypeScript most of the time, either avoiding JavaScript entirely or spending the majority of time working with TypeScript codebases [2]. Over the years, TypeScript has transformed from a simple type annotation tool to a full-fledged programming language within the type system itself. Multiple libraries with advanced TypeScript types have emerged to improve the developer experience. Libraries such as Prisma for database type-safety [3], Zod for combining schema validation and static type inference [4], or tRPC for API end-to-end type-safety across boundaries [5]. With intelligent suggestions in the editor of choice, TypeScript ensures high code quality while avoiding any runtime costs due to the type system being evaluated during compilation. With editors and IDEs using a language server powered by Language Server Protocol (LSP) to provide the developer with valuable suggestions, there is an incentive to utilise the type system instead of running a daemon alongside or adding a build step.

However, TypeScript is only as powerful as the types declared and received. A significant burden is laid on the maintainers of libraries to provide descriptive and valuable types. This thesis aims to lay out and highlight the capabilities and techniques of the TypeScript type system when applied to a non-trivial problem domain. The type-only implementation of the math expression evaluator serves as a practical case study, demonstrating the power of the TypeScript type system and the benefits of type safety.

1.2 What is a static type system

For years, type systems in programming languages have been a well-known and heavily discussed topic. The main goal of a type system is to provide a formal specification of the types of data that a program can manipulate.

In statically typed languages, the type of a variable is known at compile time. The compiler uses the additional information about data types to verify the source code during compilation. The data type itself can be deduced from the usage in the code (type inference), or a programmer explicitly specifies the data type of a variable before usage. Examples of such languages using static typing are, for instance, Java, C# or C++.

Whereas in dynamically typed languages, the type of a variable is determined at runtime based on the value being assigned, it does not need to be explicitly declared by the developer or known at compile time via type inference. Some of the popular dynamically typed languages include Python, Ruby, PHP, and, most notably, JavaScript, which is widely used to create interactive and dynamic user interfaces on the web platform. Dynamically typed languages tend to be more flexible and allow developers, notably beginner developers, to write code faster and iterate quicker.

Static typing offers numerous compelling benefits that can enhance the development process. First, a large class of errors is caught earlier in the development process. This reduces the likelihood of bugs and runtime issues that can be difficult to diagnose and debug. With static typing, developers can rely on a compiler system to ensure the code conforms to the expected data types. Developers can also refactor existing typed code more confidently, as the system gives developers direct feedback when refactoring.

Furthermore, by writing type annotations, developers are actively self-documenting the code, making the code more readable and easier to understand, especially when dealing with unfamiliar code. Finally, even though an initial commitment is necessary by writing type annotations at first, a more powerful type system can determine the developer's intent without writing additional code as the development progresses.

1.3 Structure of the work

This thesis will provide a comprehensive analysis of relevant advanced constructs found in the TypeScript type system and how they can be used to allow robust meta-programming within the types themselves. An implementation of a generic math expression evaluator library that operates strictly on the type level is provided to demonstrate the capabilities of the type system, followed by a discussion on testing and performance of the library and the impact on type checking and development experience in the editor.

Chapter 2

Analysis

2.1 Static Typing in JavaScript

JavaScript is a dynamically typed programming language where users do not need to assign types to a variable or a function, and the type is inferred automatically by the JavaScript engine at runtime. This feature lowers the barrier of entry to writing JavaScript code allowing developers to prototype and write code quickly. It can plausibly be one of the possible growth drivers of JavaScript in the last decade, making it the most commonly used programming language according to the 2022 Stack Overflow Developer Survey [6].

However, dynamic typing has its drawbacks. It is harder to detect trivial errors in the code without running it beforehand, and it is more difficult to refactor the code without breaking it, which often leads to poor software quality [7]. Proponents of static typing insist that static types allow developers to spot potential bugs and mistakes earlier during development and that it allows for better tooling, such as more rich code completion and refactoring tools.

There is an upcoming TC39 proposal for adding type annotations, broadly inspired by TypeScript syntax [8]. These annotations are only useful for build-time tooling as they are ignored in runtime. The proposal suggests that these annotations should be erased by an additional compilation step. Even though users can already provide static types using JSDoc right now, the syntax is not as clean as the proposed TypeScript-like syntax.

Regardless, many projects aim to introduce static typing to JavaScript, such as Flow or TypeScript, or alternative languages which compile back to JavaScript, such as Elm or ReScript.

2.1.1 Elm

Elm is a functional programming language designed specifically for building web applications [9]. The language compiles to JavaScript and has a strong static Hindley-Milner-based type system, which allows inferring types more often and reliably. Elm does not provide any escape hatches such as `any` in TypeScript. Thus it is harder to write unsafe code, as the types must be valid for the code to be compiled.

Elm also includes a lot of quality-of-life improvements and benefits, for instance: enforced purity of functions, out-of-the-box immutability, `case` pattern matching, JSON decoders and encoders for strict parsing, `Maybe` and `Result` monads for avoiding `null` and `undefined` references or its own virtual DOM implementation for efficient rendering of interactive user interfaces. Notably, the Elm Architecture, where the application code is organised into three parts: model, update and view [10], has greatly inspired other libraries and frameworks like Redux [11].

2.1.2 ReScript

ReScript is a programming language built on top of the OCaml toolchain. Unlike Flow or TypeScript, ReScript is not a superset of JavaScript. Instead, the language compiles to JavaScript. ReScript was created as a spin-off from the Reason programming language and accompanying BuckleScript compiler, aiming to vertically integrate and streamline the adoption barrier caused by the need to be familiar with multiple unrelated tools and toolchains [12].

The language aims to be more sound with more powerful type inference than TypeScript, borrowing the Hindler-Milner type system from OCaml implementation [13, 14]. Thus, most of the time, the types can be inferred automatically without the need to annotate them explicitly, whereas TypeScript utilises bidirectional type checking [15].

2.1.3 Flow

Flow is a static type checker for JavaScript [16, 17], which allows developers to annotate their code with static types. Flow is developed by Meta and is internally used in production by Facebook, Instagram and React Native. Type annotations in Flow are fully erasable, which means that the type annotations can be fully removed from the Flow code to emit valid JavaScript code. The checking of these types occurs at compile-time before removal in build-time. Flow is also a superset of JavaScript, which means any JavaScript code is a valid Flow code.

One of the primary goals of Flow is to provide type soundness, the ability to catch every error that might happen in runtime at compile-time, no matter how likely it is to happen. A valid Flow code can provide developers with some guarantees about the type a value has in runtime, at the expense of catching errors, which are unlikely to happen in runtime.

Both Flow and TypeScript are similar regarding features at the time of writing. Most of the type-safety differences between Flow and TypeScript have been addressed with the newer versions of TypeScript, even though a “provably correct” type system is a specific non-goal by the TypeScript team [18]. However, developers must opt-in to these features by setting `"strict"` to `"true"` in `tsconfig.json`, whereas these features are enabled by default in Flow.

2.1.4 TypeScript

TypeScript is a statically typed programming language developed and maintained by Microsoft [19]. It is a language that transpiles to JavaScript and adds static type checking to JavaScript [20]. Unlike Elm or ReScript, TypeScript is a syntactical superset of JavaScript, which means that any valid JavaScript code can be a valid TypeScript code.[1] Similar to Flow, type annotations provided by the developer are fully erasable either by the TypeScript compiler CLI or by other community build tools, such as `babel`[21], `esbuild`[22] or `swc`[23].

Type system in TypeScript is considered to be less sound and more forgiving, as soundness is stated as an explicit non-goal for the design team of TypeScript [18], with emphasis on striking a balance between productivity and correctness. By default, the TypeScript type checker is not

[1] With a lax type checker configuration

strict, and the language itself includes an escape hatch for developers to opt out of type checking by using the `any` type or using `@ts-ignore` comment annotations. Nevertheless, with proper type checker configuration, the type system of TypeScript can be as sound as in Flow.

Both Flow and TypeScript support advanced features such as generics and utility types, with the latter supporting template string literal types and better support for conditional types, unlocking the potential of writing more expressive types, which this master thesis will further explore in more detail.

With deep integration with Visual Studio Code [24], the rich build ecosystem and high compatibility with existing JavaScript libraries and tools, TypeScript has become one of the fastest growing languages in terms of usage according to the 2022 Octoverse report by Github [25].

2.2 Usage of TypeScript

The TypeScript project is made of two major parts available to developers:

- `tsc`: the TypeScript Compiler, which is responsible for both type checking and outputting valid JavaScript files,
- `tsserver`: the TypeScript Standalone Server, which encapsulates the TypeScript Compiler and language services for use in editors and IDEs [26].

While a type-checker is most likely executed manually more often and is the entry point for developers when using TypeScript, the language server is equally as useful, as it communicates with the editor via Language Server Protocol (LSP) to provide important language services. These include code completion, auto-importing, symbol renaming etc.

The term “compilation” in this thesis refers specifically to the process of type erasure itself. Although the source code may contain various type-related errors, the TypeScript Compiler (`tsc`) will generate valid JavaScript files by default as long as the input source file can be correctly parsed. This enables developers to gradually improve their code and quickly iterate on its functionality without fixing type errors immediately. In this sense, the TypeScript Compiler functions more like a code analyser rather than a traditional compiler seen in other programming languages. Regardless, in this thesis, the terms “compiling” and “type-checking” will be used interchangeably.

2.3 Typescript syntax

In TypeScript, types are typically annotated using `:[type annotation]` syntax, adding annotations to various constructs in JavaScript, such as variables, function parameters and function return values, to add constraints to values. Type annotations in TypeScript can be categorised into primitive types, literal types, data structure types, union types, intersection types and type parameters. In the following sections, we will explore each of these types in more detail. The following Listing 2.1 shows a basic example of TypeScript annotations:

At runtime, every variable has a single concrete value, but in TypeScript, the variable has only a type. A useful mental model for understanding types is to think of the type as a set of permitted values [27], effectively describing the domain of the type.

Developers can declare types directly in type annotations, but sometimes developers need to reuse the same type in multiple annotations. To avoid repeating the same declaration, we can

Listing 2.1 Basic TypeScript annotation example

```
const prefix: string = "Hello world"
const user: {
  name: string;
  age: number
}

function formatUserGreeting(
  user: {
    name: string;
    age: number;
  },
  message: string
): string {
  return [message, user.name].join(" ");
}

const greeting: string = formatUserGreeting(user, prefix);
```

use type aliases to refer to a type by a name. These type variables act as an alias, which can be used in place of the type itself. The Listing 2.2 shows a refactored `formatUserGreeting` function of the previous Listing 2.1 using type aliases.

Listing 2.2 Type aliases

```
type User = {
  name: string;
  age: number
}

const prefix: string = "Hello world"
const user: User

function formatUserGreeting(
  user: User,
  message: string
): string {
  return [message, user.name].join(" ");
}
```

2.3.1 Primitive Types

A primitive value is data that is not an object and has no methods or properties. These primitives are immutable. Thus, they cannot be altered. The TypeScript type system provides a comprehensive representation of these primitives, as seen in Listing 2.3:

Some primitive values represent a singular data value, such as `null` or `undefined`, but many of these primitives can represent multiple values (`boolean` can represent either `true` or `false`), or even an infinite amount of values, like `number`, `bignumber` or `string`.

Listing 2.3 Primitive Types

```
type StringPrimitive = string
type NumberPrimitive = number
type BigintPrimitive = bigint
type BooleanPrimitive = boolean
type UndefinedPrimitive = undefined
type NullPrimitive = null
type SymbolPrimitive = symbol
```

2.3.2 Literal Types

Literal types are used to describe an exact value as a type. From the point of view of the type system, a literal type is a subset of one of the following primitive types: `string`, `number`, `bignumber` or `boolean`,[2] as seen in Listing 2.4.[3]

Listing 2.4 Literal Types

```
type Literal = "foo" | 42 | true | 100n;

// Valid code
const Valid: Literal = "foo"

// @ts-expect-error Type '"bar"' is not assignable to type 'Literal'
const Invalid: Literal = "bar"
```

2.3.3 Types for data structures

TypeScript also allows annotating data structures such as objects and arrays with four possible types, depending on the enumerability of items and their types. The syntax overview can be seen here in Listing 2.5.

- `tuple` type for describing an array with a fixed number of elements, possibly with a different type for each element,
- `array` type for describing an array with an unknown length, and the values are of the same type,
- `record` type for describing an object with an unknown number of keys, and the values are of the same type,
- `object` type or an `interface` for describing an object with a finite set of keys with values of different types per key.

TypeScript syntax offers two notations which can be used for describing objects with a finite set of key-value pairs in TypeScript: `object` and `interface`. There are some key differences between these two notations:

[2] Both `null` and `undefined` are literal types as well
[3] The following listing uses union types, described in Section 2.3.4

Listing 2.5 Data structures

```
interface ObjectStructure {
  foo: string;
  bar: number;
}

type ObjectStructure =
  | { foo: string, bar: number }

type RecordStructure
  | { [key: string]: number }
  | Record<string, number>

type TupleStructure = [number, string]

type ArrayStructure = number[]
```

1. The `object` type uses the type alias syntax, whereas an interface is defined using a special `interface` keyword.
2. TypeScript allows multiple declarations of `interface` later merged during interpretation. This can be especially useful when augmenting non-TypeScript modules [28].
3. Even though both support object merging, `interface` can be implemented by classes, ensuring that the class adheres to the structure defined by the interface. `object` types cannot be directly implemented by a class.
4. Merging `interface` declarations is more performant when merging multiple declarations than an intersection of `object` types [29].

TypeScript uses structured typing, which means that TypeScript only validates the shape of the data. Essentially, if the data has the same shape as the type, it is considered to be of that type, as seen in Listing 2.6. This is also known as duck typing, essentially: “If it walks like a duck and quacks like a duck, it is a duck.”

Listing 2.6 Structured typing

```
type DuckLike = { quack: () => void; type: string };

const Duck: DuckLike = {
  quack: () => console.log("duck!"),
  type: "duck",
};

// This will be still valid
const Goose: DuckLike = {
  quack: () => console.log("goose!"),
  type: "goose",
};
```

Structured typing does have some drawbacks, unlike in nominal type systems, where each type is unique, and the same data cannot be assigned across types, but these can be easily mitigated using literal types to act as brands, as seen in Listing 2.7.

Listing 2.7 Nominal typing in TypeScript

```
type DuckLike = { quack: () => void; type: "duck" };

const Duck: DuckLike = {
  quack: () => console.log("duck!"),
  type: "duck",
};

// This will not be valid
const Goose: DuckLike = {
  quack: () => console.log("goose!"),
  type: "goose",
};
```

2.3.4 Union and intersection types

Revisiting the notion of types as sets of values, as seen in Listing 2.4, when attempting to assign a value not permitted by the literal type, a type error occurs. In TypeScript, a type is "assignable", if it is either a "member of" the set of permitted values defined by the type (when describing relationships between a value and a type, or it is a "subset of" the sets (when describing relationships between two types).

Sometimes, we need to describe a type, which is a combination of multiple types, combining two sets of values into a single set. This is achievable by using the union operator represented by the `|` symbol to describe a type that represents a value, which may be any of one of the combined types referred to as "union members" [30]. Essentially, `X | Y` can be read as a type for a value that can either be of type `X` or `Y`.

Since a union type can contain a value from any of the member types, TypeScript permits only those operations that are valid for all member types within the union. If we want to perform an operation which valid for some of the union members, we must perform type narrowing, which refines a broader type to a more specific narrow one, capturing a subset of values of the original broader type.

An example can be seen in Listing 2.8, where the function `printUserId` can accept both a `string` or a `number` as an argument. To invoke `toUpperCase()`, a method valid only for values of `string` type, we must perform a check if the parameter is a `string`. Afterwards, TypeScript has the necessary information to infer that the type of the checked value must be necessary a `string` and permits the invocation of `toUpperCase()`.

An intersection of types can be represented by the `&` operator. Similarly to the union type, `X & Y` can be read as a type for a value that can simultaneously belong to type `X` and `Y`. These intersection types are of particular interest when working with object types, as an intersection of two object types has all properties of both object types, as an object with both of the properties can be assigned to both of the intersection member types. For this particular reason, intersection

Listing 2.8 Union types with simple narrowing

```
function printUserId(id: string | number) {
  if (typeof id === "string") {
    return id.toUpperCase()
  } else {
    return id
  }
}
```

types are commonly used to merge two object types, as seen in 2.9.[4]

Listing 2.9 Intersection types

```
type Intersection = { a: string } & { b: number }
const item: Intersection = { a: "a", b: 1 }
```

2.3.5 `keyof` type and indexed access types

The indexed access type is used to access a specific property type of a record or a tuple type. The syntax of indexed access types mirrors the syntax for accessing an object in JavaScript, as seen in Listing 2.10. We can also use unions as keys to get types of multiple properties of an object type.

Listing 2.10 Indexed access types

```
type User = { firstName: string; lastName: string; age: number }

type Age = User["age"]
type Names = User["firstName" | "lastName"]
```

The `keyof` keyword operator can be used to get all possible keys of an object type. This will return an union of all keys of the provided data structure type. These are especially useful when working with mapped types later on. An example can be seen in Listing 2.11.

2.3.6 Special data types

When working with unions and intersections, we need to be able to describe a type, which can describe a union of all possible types or a type, which is created by intersecting two types with no related properties. We refer to these types as universal supertypes and universal subtypes, respectively. Universal supertypes, also known as top types, are types that are a superset of all other types and are used to represent any possible value. Whereas universal subtypes, also known as bottom types, are types that are a subset of all other types and are often used to describe a type that has no permitted values.

TypeScript includes two top universal supertypes: `any` and `unknown`. In the case of `any`, every type is assignable to type `any` and type `any` is assignable to every type [31]. In general, `any` can be

[4]We can also use `extends` keyword to merge two interfaces

Listing 2.11 Usage of `keyof`

```
type User = { firstName: string; lastName: string; age: number }
type Keys = keyof User
//   ^? "firstName" | "lastName" | "age"
```

used as an escape hatch to opt out of type checking. This does have unintended consequences, as `any` is assignable to every type; it can be assigned to a different type without any warnings. This is especially problematic when dealing with external data as the return type of `JSON.parse()` is `any`. An example of assignability can be seen at Listing 2.12.

Listing 2.12 Assignability of any

```
let data: any = JSON.parse("...")

// All of these are valid TypeScript code
data = null
data = true
data = {}

// Still valid code, opting out of type checking
const a: null = data
const b: boolean = data
const c: object = data
```

`unknown` acts as a more restrictive version of `any`. Every type is assignable to type `unknown`, but `unknown` is not assignable to any other type, which can be seen at Listing 2.13. To assign `unknown` to a different type, we must narrow the types using either type guards, type assertions, equality checks or other assertion functions.

Listing 2.13 Assignability of unknown

```
let data: unknown = JSON.parse("...")

// All of these are valid TypeScript code
data = null
data = true
data = {}

// Not valid, as unknown is not assignable to any other type
const a: null = data
const b: boolean = data
const c: object = data
```

Finally, `never` is a bottom type, acting as a subtype of all other types, representing a value that should never occur. In the context of the theory of mathematical logic, `never` acts as a logical contradiction, describing a value that may never exist. No other type can be assigned to `never` nor `never` cannot be assigned to any other type. `never` can be found when attempting to intersect two types that have no properties in common, such as `string & number`.

`void` is a specific type used to signify a function which does not return a value. There is a notable difference between the usage of `void` when used in context, describing a type for a function with `void` return type, and when used in the function declaration, as seen in Listing 2.14. The former is used to describe a situation when an implementation of a "void function" does return a value but should be ignored. The latter does enforce that a function should not return a value at all.

Listing 2.14 Return type void

```
type voidFn = () => void

// Valid code
const fn1: voidFn = () => true

function fn2(): void {
  // @ts-expect-error Not valid, as void functions cannot return a value
  return true
}
```

2.3.7 Enumerations

`enum` type is a distinct subtype used to describe a set of named constants. Instead of using individual variables for each constant, an `enum` provides an organised way to express a collection of related values. `enum` is one of the few TypeScript features which introduce an additional code added to the compiler output, and enums refer to real objects at runtime.

An `enum` type consists of members and their corresponding initialisers for the runtime value of the member. There are two types of enums in TypeScript: numeric enums and string-based enums. In numeric enums, each member is assigned a numeric value, as seen in Listing 2.15. Each member can have an optional initialiser to specify an exact number corresponding to a member. If omitted, the value of the member will be generated by auto-incrementing from previous members.

Listing 2.15 Numeric enums

```
enum Direction {
  Up = 1,
  Down,
  Left,
  Right,
}
```

String-based enums are similar in nature, where each member is assigned a string value instead. Each member thus must have an initialiser with a string literal, as seen in Listing 2.16. The key benefit of string-based enums is that they tend to keep their semantic value well when serialising, which is especially helpful when debugging, as the values of numeric enums tend to be opaque.

Listing 2.16 String-based enums

```
enum Direction {
  Up = "UP",
  Down = "DOWN",
  Left = "LEFT",
  Right = "RIGHT",
}
```

2.3.8 Namespaces

TODO: Add description to namespaces

2.3.9 Generic Types

Sometimes we need to write code which needs to work and accept types we don't know in advance. Generic types allow the development of such reusable components that can work over a variety of types rather than a single one. Generic types are created by defining a type parameter that can be used as a placeholder for a specific type. The consumers can then replace the placeholder with their desired types when using the component. In TypeScript, generic types can be defined on interfaces, functions and classes.

To illustrate the point, consider the implementation of the built-in `Array` type found in the `lib.*.d.ts` files (a subset can be seen at Listing 2.17). The `Array<T>` is a generic type, which accepts a single type argument `T` and is used to describe the type of the elements in the array. The type argument `T` is later used both in arguments and return types of the methods of the `Array<T>` type: `push()` accepts only elements of the same type as the array while `pop()` will return an element of the same type.

Listing 2.17 Array type

```
interface Array<T> {
  push(...items: T[]): number;

  pop(): T | undedfined;
}

const strArr: Array<string> = []
const numArr: Array<number> = []

strArr.push("one", "two")
numArr.push(1, 2)

const a = strArr.pop()
//    ^? string

const b = numArr.pop()
//    ^? number
```

Generic types can be interpreted as functions in a meta-programming language found inside the TypeScript type system itself. The meta-programming language implements some of the key concepts found in the functional programming paradigm.

Generic types are considered first-class citizens in the language, being able to be passed as arguments into other generic types, similar to functions in a functional programming language. Generic types are also pure and cannot have any side effects during type checking. We also use recursion in the meta-programming language to break down complex problems into smaller ones and solve them independently.

There is a notable omission, however: generic types cannot receive other generic types as type arguments [32]. Thus, higher-order functions are not permitted.[5]

2.3.10 Type constraints with `extends`

When writing generic types, sometimes we need to be able to describe some expectations that a type argument must satisfy. For example, we might want to accept types which do have a certain property, such as `length` as seen in Listing 2.18. To achieve this, we use the `extends` keyword to describe our constraints to the type.

Listing 2.18 Type constraints with `extends`

```
function getLength<T extends HasLength>(obj: T): number {
  return obj.length
}

const a = getLength("hello")
const b = getLength([1, 2, 3])
const c = getLength({ length: 10 })

// @ts-expect-error
// Argument of type '{ foo: string; }' is not
// assignable to parameter of type 'HasLength'.
const d = getLength({ foo: "bar" })
```

The generic function will not be able to accept any types anymore, as desired and we must only pass types, which satisfy the constraints instead.

2.3.11 Conditional types

Within the TypeScript meta-language, developers can write conditions and branching logic using conditional types. Conditional types follow a syntax similar to the conditional ternary operators with another case of overloading the `extends` keyword: `Input extends Expect ? A : B`. This can be read as "If type Input is assignable to type Exepct, then the type resolves to type A, otherwise to type B." An example can be seen in Listing 2.19, where the `IsString<T>` type will resolve to `true` if the type argument `T` is assignable to `string` and to `false` otherwise.

Listing 2.19 Conditional types

```
type IsString<T> = T extends string ? true : false
```

[5] There is a way to create such type using HOTScript, more on that later

We can use the `infer` keyword to deduce or extract a specific type within the scope of conditional types, essentially acting as a way to perform pattern matching. With `infer`, we introduce a new generic type variable, which can be later used within the true branch of the conditional type, as seen in the implementation of the `ReturnType<T>` utility type in Listing 2.20. The `ReturnType<T>` type will resolve to the return type of the type argument `T`.

Listing 2.20 Infer in conditional types

```
type ReturnType<T> = T extends (...args: any) => infer R ? R : never;
```

Since TypeScript version 4.7 [33], we can also add an additional type constraint for the inferred type, which will be checked before the conditional type is resolved. This is useful when we want to avoid an additional nested conditional type, as seen in Listing 2.21, where we want to return the first element of the tuple type only if it is a string.

Listing 2.21 Type constraints within infer

```
type FirstIfString<T> =
  T extends [infer S extends string, ...unknown[]]
    ? S
    : never;

// is equivalent to
type FirstIfString<T> =
  T extends [infer S, ...unknown[]]
    ? S extends string ? S : never
    : never;
```

When given a union type within the conditional type, the conditional type will be resolved for each member type in the union separately, essentially distributing the union type. To prevent such behaviour, we can wrap the type argument in a tuple or any other structure type.

Listing 2.22 Distributing union types

```
type ToArray<Type> = Type extends any ? Type[] : never;

// $ExpectType string[] | number[]
type A = ToArray<string | number>

type ToArrayNonDist<Type> = [Type] extends [any] ? Type[] : never;

// $ExpectType (string | number)[]
type B = ToArrayNonDist<string | number>
```

2.3.12 Mapped types

Sometimes we need to transform a type into another type. For example, we might want to create a new type, which is a copy of the original type, but with all properties being optional. This can be achieved using mapped types. Mapped types are types which are created using the syntax for index signatures, commonly used in JavaScript for properties not declared ahead of time. An example is shown in Listing 2.23, where the generic type `ToBoolean<T>` will create a new type which will take all properties from `T` and change their values to `boolean`.

We can also specify mapping modifiers to affect the mutability or optionality of a property: `readonly` and `?` respectively. Prefixing the modifier with either `+` or `-` will either add or remove the modifier to the property.[6] This can be seen in the `Optional<T>` type in Listing 2.23, which will create a new type, which is a copy of the original type, but with all properties being optional.

Listing 2.23 Mapped types

```
type ToBoolean<T> = {
  [K in keyof T]: boolean
}

type Optional<T> = {
  [K in keyof T]+?: T[K]
}
```

Introduced in TypeScript 4.1 [34], we can also use the `as` keyword to re-map keys in mapped types. This can allow us to create, transform or filter out keys when creating a new type. An example is shown in Listing 2.24, where the `Omit<T, Key>` creates a new object type based on type `T` with omitted properties which are assignable to `Key`.

Listing 2.24 Using as in mapped types

```
type Omit<T, Key> = {
  [K in keyof T as Exclude<K, Key>]: T[K]
}
```

2.3.13 Recursive Types

A recursive data type is a data type that includes a reference to itself within the type definition. Recursive types are useful for modelling complex or hierarchical data structures, such as linked lists or trees.

An example can be seen in Listing 2.25, where the `Tree<Value>` generic type represents an object with a value of type `Value` and optional left and right subtrees of the same type.

Listing 2.25 Modeling a binary tree with recursive types

```
type Tree<Value> = {
  value: Value,
  left?: Tree<Value>,
  right?: Tree<Value>
}
```

Using recursive types combined with generic types, we can implement typical recursive algorithms useful for this thesis. One such example can be seen at Listing 2.26, where we implement

[6] + is assumed by default if omitted

a `FromEntries<Entries>` generic type, converting a list of `[Key, Value]` tuples into a single object type.

First, we define an additional optional generic type parameter `Accumulator` with an initial type value of . For every tuple in a list, we create an object type containing the current key-value pair with `{ [K in Key]: Value }` and merge it with the accumulator using the `&` operator. The merged object type is then passed as the accumulator to the next iteration. Finally, when the list is empty, we return the accumulator, which will be the final object type.

Listing 2.26 Reduce example

```
type FromEntries<Entries, Accumulator = {}> =
  Entries extends [infer Entry, ...infer Rest]
    ? FromEntries<
        Rest,
        Entry extends [infer Key, infer Value]
          ? { [K in Key]: Value } & Accumulator
          : Accumulator
      >
    : Accumulator;
```

There are some limitations regarding recursive types. To prevent infinite recursion, TypeScript limits the instantiation depth to ensure a consistent and performant developer experience. As of writing, the limit is set to 100 levels for type aliases and 5 million type instantiations [35]. Thanks to the tail-recursion elimination optimisation, the limit is set to 1000 levels for tail-optimized recursion types. Thus, it is desired to use tail recursion whenever possible.

Another limitation related to recursive generic types is that the variables declared with `infer` do not inherit the constraints of the parent type, as seen in Listing 2.27. As the `Tail` type lost the type constraint of `Haystack`, we cannot pass the tail as the new haystack of the `FilterWrong` type. To remedy this issue, we need to add an additional type constraint to the inferred type.

Listing 2.27 Recursive types and type constraints

```
type FilterWrong<Haystack extends string[], Needle extends string> =
  Haystack extends [infer Head, ...infer Tail]
    ? Head extends Needle
      // $ExpectError Type 'Tail' does not satisfy the constraint 'string[]'.
      ? [Head, ...FilterWrong<Tail, Needle>]
      : FilterWrong<Tail, Needle>
    : [];

type FilterCorrect<Haystack extends string[], Needle extends string> =
  Haystack extends [infer Head, ...infer Tail extends string[]]
    ? Head extends Needle
      ? [Head, ...FilterCorrect<Tail, Needle>]
      : FilterCorrect<Tail, Needle>
    : [];
```

2.3.14 Template Literal Types

Finally, template literal types are based on the string literal types, allowing string interpolation and manipulation within the TypeScript type system. For this thesis, we use template literal types to create a parser of mathematical expressions. However, template literal types can be used to create fully typed string-based Domain Specific Languages (DSL).

Similar to the syntax of JavaScript template literal strings, we use backticks to create a new template literal type. When used with a string literal type, a template literal will create a new string literal type by concatenation [36]. For example, the type `` `Hello ${"World"}` `` will create a new string literal type `"Hello World"`.

Template literal types can be used with primitive types as well, the only limitation being that the primitive type must be stringifiable. That includes all of the primitive types except the `symbol` type. When created, these types are as a subset of their primitive type and can be used to work as a validation mechanism matching a string of an expected format. For instance, the type `` `localhost:${number}` `` will create a new string literal type that will match a string of the format `localhost:PORT`, where `PORT` is a number.

The distributive nature of union types applies to template literal strings as well: the type will be applied for every member type of the union to the template literal, as seen in the Listing 2.28, where we create a new `Style` type with all of the possible combinations of the `Variants` and `Weights` types. Generally, it is preferable to avoid combinations of big union types, as it can lead to worse type-checking performance or an error if a union type reaches 1 000 000 member types.

Listing 2.28 Distributive nature of unions in template literal types

```
type Variants = "primary" | "secondary"
type Weights = 100 | 200 | 300

type Style = `${Variants}-${Weights}`
//   ^? | "primary-100" | "primary-200" | "primary-300"
//      | "secondary-100" | "secondary-200" | "secondary-300"
```

Finally, we can use inference in template literal types to perform pattern matching within string literals with the combination of conditional types and the `infer` keyword. In Listing 2.29, we create a generic type `SplitString`, which splits a string literal type into a tuple of substrings with a space as the delimiter. We attempt to perform pattern matching a string with `Head`, containing the first character, and `Rest`, including the rest of the split string, as the two inferred types as a result. We also apply type constraints for the inferred types to ensure the types are assignable to `string`.[7] Both of the inferred types are used to create a new tuple type, with `Head` being the first element of the tuple and `Rest` used in a recursive call to split the rest of the string.

Listing 2.29 Pattern matching with template literal types

```
type SplitString<Input extends string> =
  Input extends `${infer Head extends string} ${infer Rest extends string}`
    ? [Head, ...SplitString<Rest>]
    : [Input];
```

[7] Albeit unnecessarily, as TypeScript automatically applies the `string` type constraint in this instance

2.4 Prior Art

There are multiple basic implementations of math operations in TypeScript. Tasks regarding basic math operations are even part of the TypeChallenges collection[37]. However, most of them only work on integers, as they work on tuple expansion, which will be further discussed in the implementation part of this thesis.

Nevertheless, multiple libraries in the NPM registry provide basic math calculations within the TypeScript type system, but none provide a fully typed parser of mathematical expressions. Some of the libraries found do provide type utilities that operate on floating-point numbers instead of integers, such as `type-fest` [38] or `typescript-lodash`[39]. The most comprehensive implementation of math operations can be found in the `ts-arithmetic` library [40], which provides a fully typed implementation of division.

Chapter 3

Implementation

This chapter delves into the implementation of the math expression evaluator using the TypeScript type system. The work being done in this thesis is realised into two major parts: the realisation of mathematical operations and parsing and evaluating string literals with a mathematical expression. The limitations and workarounds for TypeScript literal types are discussed, and by the end of this chapter, readers should gain a deeper understanding of the TypeScript type system when applied to non-trivial problem domains.

3.1 Type representation of numbers

As powerful as the type system in TypeScripts, there are certain limitations that need to be addressed when working with literal numerical types. Namely, although TypeScript type syntax includes literal number types, useful for representing specific numeric values, these types do not directly support mathematical operations, such as addition or subtraction. Due to these limitations, other methods of representing numbers are explored in this thesis.

One approach to representing numbers in TypeScript is to use tuples types. As described in 2.3.3, tuple types allow developers to describe a fixed-length JavaScript array where each element can have a specific type. As it represents a JavaScript array, the type includes all of the properties and methods found in an array, including `length` property, which contains the actual number of elements in the tuple. This feature can be used to represent a number, as the length of the tuple can represent the number itself, as seen in Listing 3.1. The actual type of a member item in a tuple is irrelevant, as the type system only cares about the length of the tuple, but for clarity purposes, the literal type `0` can be used as the element type of a tuple.

However, manually describing a tuple is tedious. Recursion can be employed to parse a literal number type to a tuple type, as seen in 3.2. The `ParseNumber<Value>` generic type accepts a mandatory type argument `Value` that should be the length of the final tuple and an optional type argument `Acc` used to preserve the state of the recursion.

First, a check is performed to see if the length of `Acc` is equal to the `Value` by checking the assignability of types. If that is the case, the tuple type found in `Acc` is returned. Otherwise, the list is extended with a new `0` element being prepended, and the function is called recursively. The function is called recursively until the length of `Acc` is assignable to `Value`.

It is possible to improve the number of recursions to create a tuple by expanding by a

Listing 3.1 Tuple representation of a number

```
type Zero = []
type Four = [0, 0, 0, 0]

// $ExpectType 0
type ZeroValue = Zero['length']

// $ExpectType 4
type ZeroValue = Four['length']
```

Listing 3.2 Parse a literal number type to a tuple type

```
type ParseNumber<
  Value extends number,
  Acc extends Array<0> = []
> = Acc["length"] extends Value ? Acc : ParseNumber<Value, [0, ...Acc]>
```

whole digit instead of by single increments. As seen in Listing 3.3, where `ParsedNumber2` will first perform stringification of the literal number type `T` and infer the first digit recursively. The accumulator type parameter `Rest` is first expanded ten times by the `ExpandArrayTenTimes` generic type, and then the parsed digit is spread into `Rest` as well. The recursion is performed until the stringified number is empty and the final `Rest` type is returned.

Listing 3.3 Parse by digit expansion

```
type ExpandArrayTenTimes<R extends Array<0>> = [
  ...R, ...R, ...R, ...R, ...R,
  ...R, ...R, ...R, ...R, ...R
]

type ParseNumber2<
  T extends number,
  Rest extends Array<0> = []
> = `${T}` extends `${infer Digit extends number}${infer R}`
  ? ParseNumber2<R, [...ExpandArrayTenTimes<Rest>, ...ParseNumber<Digit>]>
  : Rest
```

Even though this method of representing numbers is reasonably simple, it does come at a performance cost, as the tuple must contain the number of elements equal to the number itself. As such, the checking time of the addition and subtraction operations grows as the number grows. This issue alone poses a significant problem, primarily when representing large numbers, as TypeScript has an upper limit on the number of elements in a tuple to avoid performance degradation. As of writing, the limit is set to 10 000 elements[35], which is only enough for representing integer numbers no greater than 10 000.

Another approach is to represent each digit of a number type into a tuple. This approach does avoid the limitation of the tuple size imposed by TypeScript, as it is now possible to represent much larger numbers whilst reducing the performance overhead as the checking time can be reduced for some operations, which work on individual digits. The number type is parsed into object types beforehand to improve the developer experience when implementing arithmetic operations, keeping the sign, the integer and the fractional parts of a decimal representation number separate. An example can be seen in Listing 3.4, where two object types are created: `FloatNumber`, representing a number with both integer and fractional digits, and `SignFloatNumber`, which is used to provide number sign of an existing parsed number.

Listing 3.4 Interface representation of numbers

```
type Sign = "+" | "-"
type Digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

type FloatNumber<
  IntDigits extends Digit[] = Digit[],
  FracDigits extends Digit[] = Digit[]
> = {
  int: IntDigits
  frac: FracDigits
}

type SignFloatNumber<
  Sign extends "+" | "-" = "+" | "-",
  Float extends FloatNumber<Digit[], Digit[]> = FloatNumber
> = {
  sign: Sign
  float: Float
}
```

Parsing a number type into digits can be done by recursive types, as seen in Listing 3.5. First, `ParseSignFloatNumber` attempts to infer the sign of the stringified number literal type into a new `TSign` type. Afterwards, `ParseFloatNumber` generic type attempts to split the stringified literal into an integer and a fractional part. Both parts are later parsed separately in `ParseNumber`, matching if each string contains only digits.

Listing 3.5 Number parsing into objects

```
type ParseNumber<S extends string> =
  S extends `${infer TInt extends Digit}${infer Rest}`
    ? [TInt, ...ParseNumber<Rest>]
    : []

type ParseFloatNumber<S extends NumberLike> =
  `${S}` extends `${infer Int}.${infer Frac}`
    ? FloatNumber<ParseNumber<Int>, ParseNumber<Frac>>
    : FloatNumber<ParseNumber<`${S}`>, []>

type ParseSignFloatNumber<T extends NumberLike> =
  `${T}` extends `${infer TSign extends Sign}${infer Rest}`
    ? SignFloatNumber<TSign, ParseFloatNumber<Rest>>
    : SignFloatNumber<"+", ParseFloatNumber<T>>
```

The formatting of the object representation of a number is implemented in a similar fashion, where a digit is concatenated with a string-type accumulator, as seen in a short code snippet in

Listing 3.6.

Listing 3.6 Formatting of object types

```
type JoinDigit<T extends number[]> = T extends [
  infer A extends number,
  ...infer R extends number[]
]
  ? `${A}${JoinDigit<R>}`
  : ""
```

3.2 Addition and Subtraction

When representing the numbers as tuple lengths, some operations, such as addition and subtraction, can be easily implemented by spreading or inference, as seen in Listing 3.7. In the case of the addition operation, a new tuple type is created by spreading the elements of both tuples into a new tuple, which is then used to obtain the length representing the result.

Listing 3.7 Addition with tuple types

```
type Add<A extends number, B extends number> = [
  ...ParseNumber<A>,
  ...ParseNumber<B>
]['length']
```

The subtraction operation, assuming the first number is larger than the second one, is implemented with the idea that the tuple type of a first number contains all of the elements of the second number with a remainder. As seen in Listing 3.8, the `Subtract` generic type accepts two type arguments, `A` and `B`, which represent the numbers to subtract. A conditional type is used to check if `ParseNumber<A>` is assignable to a tuple that contains the elements of `ParseNumber<B>` followed by a remainder of the `number[]` type, inferred in a new type named `Remainder`. If true, the length of the `Remainder` is returned as the result of the subtraction operation. Otherwise, the `never` type is returned instead.

Listing 3.8 Subtraction with tuple types

```
type Subtract<
  A extends string | number,
  B extends string | number
> = ParseNumber<A> extends [
  ...infer Remainder extends number[],
  ...ParseNumber<B>
]
  ? Remainder["length"]
  : never
```

As described in the previous section, the final implementation of the addition operation based on object representation of numbers is the traditional schoolbook addition with carry. The algorithm adds the numbers digit by digit and keeps track of the carry as it moves from one digit to the next. This technique has a time complexity of $\Theta(n)$, where n is the number of digits in the number being added.

The core building block of the schoolbook addition and subtraction algorithm is the ability to obtain the next digit alongside the carry or borrow flag when performing the operation on single decimal digits. This can be purely done in the type system alone using tuple expansion and checking for the stringified length of the tuple, as seen in 3.9, but to improve the performance and avoid unnecessary type instantiations, a lookup table is used to obtain the next digit and the carry flag instead. The subtraction operation is implemented similarly, where a two-dimensional lookup table of tuples is used to obtain the next digit and the borrow flag.

The lookup table is created by iterating over all possible combinations of two digits and storing the result of the addition and the carry flag in a two-dimensional map. To improve the performance even further, the lookup tables of both the addition and subtraction operations are generated as a built step in JavaScript and stored in a separate file, which is later imported into the type system.

Listing 3.9 Lookup table for addition operation

```
type AddDigitsResult<A extends Digit, B extends Digit> =
    [...ParseNumber<A>, ...ParseNumber<B>]["length"] extends
    infer Length extends number
    ? `${Length}` extends `${Digit}${infer Value extends Digit}`
      ? [Value, true]
      : `${Length}` extends `${infer Value extends Digit}`
        ? [Value, false]
        : never
    : never

// This is generated by a build step
type AddMapCarry = {
  [A in Digit]: {
    [B in Digit]: AddDigitsResult<A, B>
  }
}
```

The schoolbook addition algorithm, seen in Listing 3.13, is implemented as three generic types. `AddWithCarry` accepts two digits named `Left` and `Right` and a carry flag as type arguments and is responsible for adding the two digits and propagating the carry flag to the next digit. It will first check if the `Carry` type is assignable to `true`, and if it is assignable, it will increment the `Left` digit. The `AddMapCarry` is used to obtain the result, and the `Or` generic type implements the binary disjunction operation to determine the carry flag in case of multiple additions due to `Carry` being true.

`AddArr` is responsible for adding two tuples of digits. `AddArr` will attempt to extract the rightmost digit from both tuples and add them using `AddWithCarry`. The `AddArr` will be called recursively with the remaining digits and the carry flag from the previous addition until both of the tuples are empty. Note that both of the digit tuples must have the same length to prevent premature bailouts.

Finally, `AddInt` will add two digit tuples by first padding them into tuples of the same length by prefixing them with zeroes and then calling `AddArr` to perform addition itself. If `Carry` is assignable to `true`, an extra `1` digit is prepended to the result.

These foundational blocks can be further chained to add support for fractional numbers and signed numbers. As seen in Listing 3.13, `AddFloatNumber` will first extract the integer and fractional parts of a number, performing integer addition on both parts separately. The carry flag is propagated appropriately from the fractional part to the integer part by recursively calling `AddFloatNumber` to increment the result.

Listing 3.10 Floating point addition

```
type AddFloatNumber<
A extends FloatNumber,
B extends FloatNumber
> = PadFloat<A, B> extends [
  FloatNumber<infer IntA, infer FracA>,
  FloatNumber<infer IntB, infer FracB>
]
? AddArr<FracA, FracB> extends [
    infer FracResult extends Digit[],
    infer FracCarry extends boolean
  ]
  ? AddArr<IntA, IntB> extends [
      infer IntResult extends Digit[],
      infer IntCarry extends boolean
    ]
    ? IntCarry extends true
      ? FracCarry extends true
        ? AddFloatNumber<
            FloatNumber<[1, ...IntResult], FracResult>,
            FloatNumber<[1], []>
          >
        : FloatNumber<[1, ...IntResult], FracResult>
      : FracCarry extends true
      ? AddFloatNumber<
          FloatNumber<IntResult, FracResult>,
          FloatNumber<[1], []>
        >
      : FloatNumber<IntResult, FracResult>
    : never
  : never
: never
```

When working with subtraction, underflows are resolved by implementing digit comparison. Similarly to addition and subtraction, the comparison operation is performed per digit, utilising an additional two-dimensional lookup table with all possible digit comparison results represented as a number from the following set: $\{-1, 0, 1\}$. Based on the comparison result, the operation can be decided by using a map object type with the comparison result as the key and the operation as the value, seen in Listing 3.11.

Finally, to simplify dealing with signed operations, an object type with all possible sign pairs can be used to determine whether to invoke addition or subtraction, as seen in Listing 3.12.

Listing 3.11 Subtraction switching

```
type SubOperatorSwitch<A extends FloatNumber, B extends FloatNumber> = {
  [-1]: SignFloatNumber<"-", SubFloatNumber<B, A>>
  [0]: SignFloatNumber<"+", FloatNumber<[0], []>>
  [1]: SignFloatNumber<"+", SubFloatNumber<A, B>>
}[CompareAbsNumbers<A, B>]
```

Listing 3.12 Signed number addition and subtraction

```
type AddSignFloatNumber<
  A extends SignFloatNumber,
  B extends SignFloatNumber
> = {
  "+": {
    "+": SignFloatNumber<"+", AddFloatNumber<A["float"], B["float"]>>
    "-": SubOperatorSwitch<A["float"], B["float"]>
  }
  "-": {
    "+": SubOperatorSwitch<B["float"], A["float"]>
    "-": SignFloatNumber<"-", AddFloatNumber<A["float"], B["float"]>>
  }
}[A["sign"]][B["sign"]]
```

Listing 3.13 Addition algorithm

```
type AddWithCarry<
  Left extends number,
  Right extends number,
  Carry extends boolean
  > = Carry extends true
  ? AddMapCarry[Left][1] extends [
      infer LeftTmp extends number,
      infer LeftCarry extends boolean
    ]
    ? AddWithCarry<LeftTmp, Right, false> extends [
        infer Result extends number,
        infer RightCarry extends boolean
      ]
      ? [Result, Or<LeftCarry, RightCarry>]
      : never
    : never
  : AddMapCarry[Left][Right]

type AddArr<
  A extends number[],
  B extends number[],
  Tmp extends [number[], boolean] = [[], false]
> = [A, B, Tmp] extends [
  [...infer ARest extends number[], infer ARight extends number],
  [...infer BRest extends number[], infer BRight extends number],
  [infer Result extends number[], infer Carry extends boolean]
]
  ? AddWithCarry<ARight, BRight, Carry> extends [
      infer Digit extends number,
      infer Carry extends boolean
    ]
    ? AddArr<ARest, BRest, [[Digit, ...Result], Carry]>
    : never
  : Tmp

export type AddInt<A extends Digit[], B extends Digit[]> = PadStartEqually<
  A,
  B
> extends [infer PA extends Digit[], infer PB extends Digit[]]
  ? AddArr<PA, PB> extends [
      infer Rest extends Digit[],
      infer Carry extends boolean
    ]
    ? Carry extends true
      ? [1, ...Rest]
      : Rest
    : never
  : never
```

3.3 Multiplication

A naive implementation of the multiplication algorithm can be created by repeatably adding the multiplicand when numbers are represented by tuple length, as seen in Listing 3.14. `Multiply` generic type has two mandatory type parameters: `A` and `B` representing the multiplicand and multiplier respectively. The optional type parameter `Left` is used to track how many iterations are left before the recursion terminates. This method is considered ineffective, as the number of recursion calls is proportional to the size of the multiplicand, and the method can easily reach the instantiation depth limit with large multiplicands.

Listing 3.14 Naive multiplication algorithm

```
type Multiply<
  A extends number,
  B extends number,
  Left extends number = B
> = Left extends 0 ? 0 : Multiply<Add<A, B>, B, Subtract<B>>
```

Because of this reason, the library implements the long multiplication method instead. Similarly to the addition and subtraction algorithm, a two-dimensional lookup object type is used to obtain the resulting multiplication digit and the appropriate carry number. First, `MultiplyInt` will iterate on multiplier digits from right to left and multiply each digit with the multiplicand by invoking the `MultiplySingleInt` generic type. The result of each multiplication, appropriately offset with zeroes to account for the position of the digit in the multiplier, is then added together to obtain the final result. An example can be seen in Listing 3.15.

Listing 3.15 Long multiplication

```
type MultiplyInt<
  X extends Digit[],
  Y extends Digit[],
  Tmp extends { result: Digit[]; offset: Digit[] } = { result: [0]; offset: [] }
> = Y extends [...infer Rest extends Digit[], infer Single extends Digit]
  ? MultiplySingleInt<X, Single> extends infer SingleResult extends Digit[]
    ? AddInt<
        Tmp["result"],
        [...SingleResult, ...Tmp["offset"]]
      > extends infer Result extends Digit[]
      ? MultiplyInt<X, Rest, { result: Result; offset: [0, ...Tmp["offset"]] }>
      : never
    : never
  : Tmp["result"]
```

With the core building block for integer multiplication, extending the algorithm to floating-point numbers and signed numbers is straightforward.

The `MultiplyFloat` generic type, as seen in Listing 3.16, converts the floating point number to an integer by concatenating the integer part of a number with the fractional part, preserving the precision, number of digits in the fractional part, as the length of a tuple. The precision is encoded as a tuple because the precision of the multiplication is the sum of the multiplicand

and multiplier precisions. This can be done by spreading the tuples representing the precisions instead of calling expensive per-digit addition recursive types.

Listing 3.16 Float multiplication

```
type ExpandIntFloat<X extends FloatNumber> = IntFloat<
  [...X["int"], ...X["frac"]],
  ExpandNumberToArray<X["frac"]["length"]>
>

type MultiplyFloat<
  X extends FloatNumber,
  Y extends FloatNumber
> = ExpandIntFloat<X> extends infer A extends IntFloat
  ? ExpandIntFloat<Y> extends infer B extends IntFloat
    ? CompressIntFloat<
        IntFloat<
          MultiplyInt<A["mantissa"], B["mantissa"]>,
          [...A["precision"], ...B["precision"]]
        >
      >
    : never
  : never
```

The result of the integer multiplication is then converted back to a floating-point number by shifting the integer part right, as seen in Listing 3.17. This is done by iteratively taking the elements from the tuple representing the precision, acting as a counter, and prepending the rightmost digit of the integer part to the fractional part. The recursion terminates when the precision tuple is empty.

Listing 3.17 Conversion of an integer number back to a fractional number

```
type Compress<
  Count extends Array<0>,
  Left extends Digit[],
  Right extends Digit[] = []
> = Count extends [0, ...infer RestCount extends 0[]]
  ? Left extends [...infer LeftRest extends Digit[], infer End extends Digit]
    ? Compress<RestCount, LeftRest, [End, ...Right]>
    : Compress<RestCount, Left, [0, ...Right]>
  : [Left, Right]
```

3.4 Division and modulo

The implementation of the division algorithm is split into two main parts: the Euclidean division and the long division algorithm. Given two integers, a dividend x and a divisor y, the Euclidean division aims to find a quotient q and a remainder r, which satisfies the following equation:

$$x = y \cdot q + r \qquad \text{if } 0 \leq r < |b|$$

The Euclidean algorithm finds the quotient and the remainder using repeated subtraction as seen in 3.18. The `DivisionResult` contains both the temporary quotient and remainder values passed to the next iteration. The `EuclideanDivision` generic type first checks if the remainder is greater than or equal to the divisor. If that is the case, the quotient is incremented by one using `AddInt` generic type and the remainder is subtracted by the divisor using `SubDigit`. The process is repeated until the remainder is less than the divisor, at which point the computed quotient and remainder are returned.

Listing 3.18 Euclidean division

```
interface DivisionResult<
  Quotient extends Digit[] = Digit[],
  Remainder extends Digit[] = Digit[]
> { quotient: Quotient; remainder: Remainder }

type EuclideanDivision<
  Dividend extends Digit[],
  Divisor extends Digit[],
  Tmp extends DivisionResult = DivisionResult<[0], Dividend>
> = CompareDigits<Tmp["remainder"], Divisor> extends 1 | 0
  ? EuclideanDivision<
      Dividend,
      Divisor,
      DivisionResult<
        AddInt<Tmp["quotient"], [1]>,
        SubDigit<Tmp["remainder"], Divisor>
      >
    >
  : DivisionResult<TrimStart<Tmp["quotient"]>, TrimStart<Tmp["remainder"]>>
```

The long division algorithm, seen in 3.19 as `LongDivisionDigit` generic type, implemented in this thesis builds upon the foundation of the Euclidean division. In each iteration, the left-most digit is popped from the dividend and pushed to the end of the accumulated remainder. Subsequently, pass the newly created tuple as the remainder for the Euclidean division, together with the divisor. The next invocation of `LongDivisionDigit` takes the resulting dividend, the divisor and the updated accumulator of `DivisionResult` type. The updated `DivisionResult` instance has the remainder copied and the quotient concatenated from the result of the Euclidean division. The process is repeated until all digits in the dividend have been used. Finally, the quotient and remainder are returned, with the leading zeros removed.

When conducting division operations involving two numbers with fractional components, the digit tuples of fractional parts are padded with zeroes to ensure equal lengths for both tuples. Afterwards, the fractional part is concatenated behind the integer part, creating an integer number compatible with the long division algorithm. Further digit shifting is not necessary, as the orders of magnitude get cancelled out during the division process, and the division itself will return a `FloatNumber`. An example of how the numbers are processed can be seen in Figure 3.1.

Since both the long division and Euclidean division algorithms exhibit greater complexity and are prone to deep recursion, it is likely that when used, the instantiation depth limit imposed by TypeScript will be exceeded. As a workaround, it is possible to defer the evaluation of a type by rephrasing it as a distributive conditional type. This workaround will be remarkably useful

Listing 3.19 Long division

```
type LongDivisionDigit<
  Dividend extends Digit[],
  Divisor extends Digit[],
  Acc extends DivisionResult = DivisionResult<[], []>
> = Dividend extends [
  infer Head extends Digit,
  ...infer RestDividend extends Digit[]
]
  ? EuclideanDivision<
      [...Acc["remainder"], Head],
      Divisor
    > extends infer IntDivision extends DivisionResult
    ? LongDivisionDigit<
        RestDividend,
        Divisor,
        DivisionResult<
          [...Acc["quotient"], ...IntDivision["quotient"]],
          IntDivision["remainder"]
        >
      >
    : never
  : DivisionResult<TrimStart<Tmp["quotient"]>, TrimStart<Tmp["remainder"]>>
```

$$\begin{array}{rcrcl} 123.456 & = & 123.456 & = & 123456 \times 10^{-3} \\ 2.5 & = & 2.500 & = & 2500 \times 10^{-3} \\ \frac{123.456}{2.5} & = & \frac{123.456}{2.500} & = & \frac{123456 \times 10^{-3}}{2500 \times 10^{-3}} = \frac{123456}{2500} \end{array}$$

Figure 3.1 Preprocessing of fractional numbers for long division

when multiple complex arithmetic operations are chained together, as the n-th root operation will exemplify.

Modulo operation builds on top of the division, multiplication and subtraction algorithm by calculating the floor of the division result obtained when dividing the dividend by the divisor. Subsequently, the result is multiplied by the divisor and finally subtracted from the dividend to obtain the final result of the modulo operation.

3.5 Other operations

3.5.1 Comparison

Some operations require an additional type-level operation for comparing two numbers, such as the Euclidean division, for deciding whether to continue recursion. For that purpose, a type-level three-way comparison operator has been implemented, also known as the “spaceship operator” in the C++ programming language [41].

The spaceship operator for comparing two numbers x and y, denoted by $x <=> y$, is defined in Figure 3.2 as follows:

$$x <=> y = \begin{cases} -1 & \text{if } x < y \\ 0 & \text{if } x = y \\ 1 & \text{if } x > y \end{cases}$$

Figure 3.2 spaceship operator

It is possible to implement the operator entirely within the TypeScript type system by decomposing each number into a tuple of elements, where the size of the tuple is equal to the number itself. As seen in Listing 3.20, the `CompareTuples` attempts to remove the first element of both tuples until one or both of the tuples are empty. The generic type returns the appropriate value depending on which tuple is empty first.

Listing 3.20 Type-level comparison operation of single digit

```
type CompareTuples<X extends Array<0>, Y extends Array<0>> =
  X extends [0, ...infer XRest extends Array<0>]
  ? Y extends [0, ...infer YRest extends Array<0>]
  ? CompareTuples<XRest, YRest>
  : 1
  : Y extends [0, ...Array<0>]
  ? -1
  : 0

type Compare<X extends number, Y extends number> =
  CompareTuples<ParseNumber<X>, ParseNumber<Y>>
```

As is the case for addition, subtraction and multiplication, it is desirable to precompute these values for every combination of digits and store them in a lookup table.

The comparison of digit tuples is implemented by first ensuring the two tuples are of equal length by padding the shorter tuple with zeroes at the beginning. The first elements of both tuples are extracted into two type variables, `XHead` and `YHead`, and are compared using the lookup table. If the digits are equal, the recursion continues with the rest of the tuples, named `XRest` and `YRest`. Otherwise, the result of the last digit comparison is returned. The full implementation can be seen in Listing 3.21.

Listing 3.21 Digit tuple comparison

```
type CompareArr<X extends Digit[], Y extends Digit[]> =
  PadStartEqual<X, Y> extends [
    [infer XHead extends Digit, ...infer XRest extends Digit[]],
    [infer YHead extends Digit, ...infer YRest extends Digit[]]
  ]
    ? CmpMap[XHead][YHead] extends infer Result extends number
      ? Result extends 0
        ? CompareArr<XRest, YRest>
        : Result
      : never
    : 0
```

3.5.2 Numeric rounding operations

The library implements four operations performing numeric rounding. Truncation is the simplest of the four implementations, where the parsing of numbers into a structured object type is doing the heavy lifting. The truncation itself is done by replacing the fractional part of a number with an empty tuple, as seen in Listing 3.22

Listing 3.22 Truncation function

```
type Truncate<Number extends SignFloatNumber> =
  SignFloatNumber<Number["sign"], FloatNumber<Number["float"]["int"], []>>
```

Ceiling and flooring are more complex operations. In the case of the ceiling operation, the number is first truncated and then checked to see if the input number is greater than the truncated number. If that is the case, the truncated number is incremented by one and returned. This behaviour is done to obtain the same result when flooring a negative number. For flooring, the process is similar, but the truncated number is decremented by one if the original number is less than the truncated number. The implementation can be seen in 3.23.

Listing 3.23 Floor function

```
type Floor<Number extends SignFloatNumber> =
  TruncateSignFloatNumber<Number> extends
    infer TrucateNumber extends SignFloatNumber
    ? CompareSignNumbers<Number, TrucateNumber> extends -1
      ? SubSignFloatNumber<
          TrucateNumber,
          SignFloatNumber<"+", FloatNumber<[1], []>>
        >
      : TrucateNumber
    : never
```

Rounding is the most complex of the four rounding operations. The fractional part's first digit is checked to determine whether it is assignable to the union of rounding up digits ($\{5, 6, 7, 8, 9\}$). If that is the case, the truncated number is incremented by one and returned. Otherwise, the truncated number is returned as is, seen in Listing 3.24.

3.5.3 Exponentiation

A naive implementation of exponentiation would be based on repeated multiplication. This is an inefficient approach, as the complexity of such an algorithm would be $O(M(x) \cdot 10^n) = O(n^2 \cdot 10^n)$, where n is the number of digits and $M(x)$ is the complexity of multiplication algorithm, in this instance $O(n^2)$.

A more efficient exponentiation method is to perform binary exponentiation instead, as seen in Figure 3.3.

It can be shown that the complexity of the algorithm is $O(n^2 \cdot \log_2(10^n))$, a notable improvement over the naive approach.

Parity checks done by `IsEventInt` as seen in Listing 3.25 are done by checking the last digit of the exponent. Once again, the even digits are represented by a union type of number literal

Listing 3.24 Round function

```
type RoundSignFloatNumber<Number extends SignFloatNumber> =
  Number["float"]["frac"] extends [infer Head extends Digit, ...Digit[]]
  ? Head extends 5 | 6 | 7 | 8 | 9
    ? SignFloatNumber<
        Number["sign"],
        AddFloatNumber<
          FloatNumber<Number["float"]["int"], []>,
          FloatNumber<[1], []>
        >
      >
    : SignFloatNumber<
        Number["sign"],
        FloatNumber<Number["float"]["int"], []>
      >
  : Number
```

$$x^n = \begin{cases} x \cdot (x^2)^{\frac{n-1}{2}} & \text{if } n > 0 \text{ is odd} \\ (x^2)^{\frac{n-1}{2}} & \text{if } n > 0 \text{ is even} \\ 1 & \text{if } n = 0 \\ (\frac{1}{x})^n & \text{if } n < 0 \end{cases}$$

Figure 3.3 Exponentiation by squaring

types. Notably, the conditional type is not a type itself. Developers need to write `true` and `false` types explicitly.

Listing 3.25 Parity check of digits

```
type IsEvenInt<X extends Digit[]> = X extends [
  ...Digit[],
  infer Tail extends Digit
]
  ? Tail extends 0 | 2 | 4 | 6 | 8
    ? true
    : false
  : false
```

The implementation shown in Listing 3.26 does require trimming of excess zeroes in the exponent to ensure the correctness of a fast assignability check for termination conditions. The implementation differs from the algorithm in Figure 3.3 in that the `PowerAuxInt` includes an optional type argument `Y` used to convert the method to a tail-recursive generic type, bypassing the need for deferring the instantiation to avoid the depth limit.

3.5.4 n-th root extraction

There are some cases where an operation is so complex that the type instantiation limit is reached, and TypeScript will prematurely abort type checking of the entire file. One such example is the

Listing 3.26 Auxilary exponentiation by squaring

```
type PowerAuxInt<
  X extends SignFloatNumber,
  N extends Digit[],
  Y extends SignFloatNumber = SignFloatNumber<"+", FloatNumber<[1], []>>
> = TrimEnd<N> extends [0]
  ? Y
  : IsEvenInt<N> extends true
  ? PowerAuxInt<
      MultiplySignFloat<X, X>,
      LongDivisionDigit<N, [2]>["quotient"],
      Y
    >
  : PowerAuxInt<
      MultiplySignFloat<X, X>,
      LongDivisionDigit<SubDigit<N, [1]>, [2]>["quotient"],
      MultiplySignFloat<X, Y>
    >
```

n-th root extraction of a number. The implementation uses the Newton-Raphson method.

The Newton-Raphson method [42] is an iterative numerical method for estimating the roots of real-valued functions. Assuming the function $f(x)$ is derivable on $x \geq 0$ and an initial guess for root is x_0, then:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

Thus, to estimate the n-th root of a number, declared by the function $f(x) = x^n - \alpha$, where α is the target number to apply n-th root and n is the degree of the root, the following definition for the next approximation is used:

$$\begin{aligned}
x_{k+1} &= x_k - \frac{f(x_k)}{f'(x_k)} \\
&= x_k - \frac{x_k^n - \alpha}{n \cdot x_k^{n-1}} \\
&= \frac{1}{n}\left((n-1) \cdot x_k + \frac{\alpha}{x_k^{n-1}}\right) \\
&= \underbrace{\frac{n-1}{n}}_{\text{L}} x_k + \underbrace{\frac{\alpha}{n}}_{\text{R}} \frac{1}{x_k^{n-1}} \\
&= L \cdot x_k + R \cdot \frac{1}{x_k^{n-1}}
\end{aligned}$$

A naive implementation can be done by intimately mirroring the algorithm and nesting the generic types for readability, shown in Listing 3.27. However, as it turns out, TypeScript will bail out due to the depth limit. Instead, to bypass the limit, the final implementation seen in Listing 3.28 uses `infer` keyword to defer instantiation as much as possible, essentially treating `infer` as a way to assign intermediate values to variables.

Even so, it is not desired for the algorithm to run indefinitely; instead, the iteration is cut off after seven iterations, as more iterations will cause the type checker to reach the instantiation limit when evaluating.

Listing 3.27 n-th root - wrong version

```
type RootDigit<
  Alpha extends SignFloatNumber,
  N extends Digit[],
  Step extends SignFloatNumber = SignFloatNumber<"+", FloatNumber<[1], []>>,
  StepCnt extends Array<0> = []
> = StepCnt["length"] extends 5
  ? Step
  : RootDigit<Alpha, N, MultiplySignFloat<
      SignFloatNumber<"+", DivideInt<[1], N>>,
      AddSignFloatNumber<
        MultiplySignFloat<RootNSubOne<N>, Step>,
        DivideSignFloatNumber<Alpha, PowerSignFloatNumbers<Step, RootNSubOne<N>>>
      >
    >, [...StepCnt, 0]>
```

Listing 3.28 n-th root - right version

```
type OneSignFloatNumber = SignFloatNumber<"+", FloatNumber<[1], []>>

type RootDigitIter<
  NSubOne extends SignFloatNumber,
  L extends SignFloatNumber,
  R extends SignFloatNumber,
  Step extends SignFloatNumber = OneSignFloatNumber,
  StepCnt extends Array<0> = []
> = StepCnt["length"] extends 7
  ? Step
  : MultiplySignFloat<L, Step> extends infer LStep extends SignFloatNumber
  ? PowerSignFloatNumbers<
      Step,
      NSubOne
    > extends infer StepPowNSubOne extends SignFloatNumber
    ? DivideSignFloatNumber<
        R,
        StepPowNSubOne
      > extends infer RStep extends SignFloatNumber
      ? AddSignFloatNumber<
          LStep,
          RStep
        > extends infer Sum extends SignFloatNumber
        ? RootDigitIter<NSubOne, L, R, Sum, [...StepCnt, 0]>
        : never
      : never
    : never
  : never

type RootDigit<
  Alpha extends SignFloatNumber,
  N extends Digit[]
> = SignFloatNumber<
  "+",
  FloatNumber<N, []>
> extends infer N extends SignFloatNumber
  ? SubSignFloatNumber<
      N,
      OneSignFloatNumber
    > extends infer NSubOne extends SignFloatNumber
    ? DivideSignFloatNumber<NSubOne, N> extends infer L extends SignFloatNumber
      ? DivideSignFloatNumber<Alpha, N> extends infer R extends SignFloatNumber
        ? RootDigitIter<NSubOne, L, R>
        : never
      : never
    : never
  : never
```

3.6 Statement parser and evaluator

The generic types for mathematical operations are well suited for simple expressions. However, the proposed interface can be too verbose when describing complex formulas. A more elegant solution is to represent both input and output as a literal string type and let a compiler do the parsing and evaluation of the expression. The input literal string type will contain a mathematical expression in infix notation, and the output literal string type will contain the result of the expression. The compiler is built in three parts: the lexer, the parser and the evaluator, which will be described in the following sections.

3.6.1 Lexer

The lexical analyser (lexer) is responsible for dividing the input literal string type into a sequence of meaningful units called tokens. The goal of a lexer is to remove whitespaces and inconsistencies to simplify the input stream, which is helpful for later stages of parsing. One such example is the parsing of numbers: consuming a single number token is easier than parsing each digit of a number, which can unnecessarily complicate the design of a parser.

The following section will provide an in-depth look into the handwritten lexer implementation. Namespaces have been used to describe the object types representing tokens, ensuring proper isolation between different type aliases and preventing naming clashes without the need to resort to prefixing. An example can be seen in 3.29, where `Plus` and `Minus` are type aliases for the object types, whereas `_` is a union type for when a placeholder for a token is needed. Also, instead of utilising the `never` type for errors, a string enum is used to prevent unintended matches when performing assignability checks, as `never` is a subtype of all types.

Listing 3.29 Lexer token namespace

```
enum Error {
  Lexer = "LexerError",
  Parser = "ParserError",
}

namespace Token {
  export type Plus = { type: "Plus" }
  export type Minus = { type: "Minus" }
  export type _ = Plus | Minus
}
```

The lexing itself is done by a generic type, which accepts a literal string type as a type argument and returns either a string enum as an error or an object type containing the matched token and the remaining unparsed input. As seen in Listing `lst:lexer-structure`, the `HandleToken` attempts to perform pattern matching on the first character of the input string literal type `T`. If succeeded, the matched token is returned wrapped in an object type defined by `LexResult` generic type, passing both the matched token and the remaining input to the next iteration of the `HandleToken` generic type. If the pattern matching fails, the `Error.Lexer` string enum is returned instead. This structure can be chained together to create a lexer for multiple token types, such as function keywords or numbers.

Listing 3.30 Lexer Structure

```
type LexResult<Rest extends string, Result extends Token._> = {
  result: Result
  rest: Rest
}
type HandleToken<T extends string> =
  T extends `${infer Head}${infer Rest}`
  ? Head extends "+"
    ? LexResult<Rest, Token.Plus>
    : Error.Lexer
  : Head extends "-"
    ? LexResult<Rest, Token.Minus>
    : Error.Lexer
  : Error.Lexer
```

3.6.2 Parser

Infix notation is the most common way of writing mathematical expressions and is more intuitive for humans to read and write. However, it is not ideal for computers due to the complexity of parsing algorithms, which must adequately evaluate parentheses and operator precedence rules. Postfix notation addresses the shortcomings of infix notation by explicitly stating the order of computation, making the evaluation unambiguous.

Various methods exist for converting an expression in infix notation. However, this thesis focuses on implementing a top-down LL(1) parser for mathematical expression. The main reason for choosing the LL(1) parser is the extendibility and suitability for supporting other LL(1) grammars more readily. Some other common parsers were considered for this thesis, including the Shunting-Yard algorithm, using two stacks for operators and output operands, and the Pratt parser, a recursive descent parsing algorithm utilising a precedence table for extendability.

In order to explain the LL(1) parser, the following sections will describe the core concepts of grammar and parsing. Grammar is a set of rules that defines the syntax of a language. The grammar $G = (\Sigma, N, R, S)$ consists of a set of terminals Σ, a set of non-terminals N, a set of production rules R and a start symbol S. Terminals are the basic unit of the language, while non-terminals are placeholders for other terminals and non-terminals. Production rules define how non-terminals can be expanded into a sequence of other non-terminals and terminals while the start symbol defines the starting non-terminal. The parsing itself will create a derivation, which is a sequence of production rules applications transforming a string of symbols, usually starting from the start symbol of the grammar.

A derivation can be visualised as a tree, also known as the derivation tree or parse tree. Each node of the tree represents a symbol in the string and each edge represents a production rule application. The root of the tree is the start symbol of the grammar and the leaves are the terminal symbols of the string.

There are multiple ways to construct a derivation tree: either by replacing the leftmost non-terminal symbol with the righthand side of a production rule or by replacing the rightmost non-terminal symbol with the righthand side of a production rule. The former is known as the leftmost derivation while the latter is known as the rightmost derivation. Finally, an ambiguous grammar is a grammar that can produce multiple derivation trees for the same string.

As an example, consider the given simplified grammar for mathematical expressions applied for the expression $3 + 4 * 5$. As can be seen in Figure 3.4, the given grammar is ambiguous, as there are multiple possible derivation trees for the given input string.

1. `E` → `E "+" E .`
2. `E` → `E "*" E .`
3. `E` → `"number" .`

Figure 3.4 An example of ambiguous grammar and the parsing tree for $3 + 4 * 5$

LL(1) parsers are a class of top-down parsers that read the input string from left to right and construct a leftmost derivation of the input. They use a single token of lookahead when parsing a sentence, meaning that the parser can only see the next token before parsing. LL(1) parsers recognise LL(1) grammars, which are a special case of context-free grammars. The grammar must be unambiguous, without any left recursion and common prefixes among the alternatives of any expansion rule to be deterministic.

The parser relies upon two important concepts: the FIRST(α) and FOLLOW(A) sets. Assuming a context-free grammar $G = (\Sigma, N, R, S)$, the FIRST(α) set is a set of terminals, that can appear as the first symbol in a string derived from α. Formally, FIRST(α) can be defined as follows:

$$\mathrm{FIRST}(\alpha) = \{a | \alpha \Rightarrow^{\star} a\beta, a \in \Sigma, \alpha, \beta \in (N \cup \Sigma)^{\star}\} \cup \{\varepsilon | \alpha \Rightarrow^{\star} \varepsilon\}$$

The FOLLOW(A) set is a set of terminals, that can appear as the next symbol in a string derived from a given non-terminal symbol A. Formally, FOLLOW(A) can be defined as follows:

$$\mathrm{FOLLOW}(A) = \{a | S \Rightarrow^{\star} \alpha A \beta, a \in \mathrm{FIRST}(\beta)\}$$

Given both the FIRST(α) and FOLLOW(A) sets, a parsing table can be constructed. The parsing table is created as follows: for each of the production rule $A \rightarrow \alpha$ found in the grammar, do the following:

1. For each terminal a found in the FIRST(α), add the production role $A \rightarrow \alpha$ to the parsing table at the position $[A, a]$.
2. If the ε token, determining the end of input, is present in the FIRST(α) set, add $A \rightarrow \alpha$ to the parsing at position $[A, b]$ for each terminal b in the FOLLOW(A) set.

When designing a LL(1) grammar for mathematical expressions, operator precedence must be taken into consideration, as the expression found in the input string literal type is written in the infix notation. Left or right associativity is a key constraint as well, with exponentiation being an operator with right associativity instead of left associativity as the other operators. The precedence and associativity rules for the operators can be seen in Table 3.1.

Precedence	Operator Type	Associativity
1	Addition, Subtraction	left-to-right
2	Multiplication, Division, Remainder	left-to-right
3	Factorial	non-associative
4	Unary plus, Unary negation	non-associative
5	Exponentiation	right-to-left
6	Function call, grouping	non-associative

Table 3.1 Associativity and precedence rules for math expressions

The final grammar used for this thesis can be seen in Figure 3.5. The operator precedence rules is baked into the grammar itself, where the non-terminals representing the higher precedence operations are expanded later. The associativity of operators have been taken into consideration as well, by changing the position of the non-terminal from the lefthand side, essentially switching from left recursion to right recursion and vice versa. An example can be seen in 3.2, where both the previous context-free grammar and the appropriately modified LL(1) version of the grammar is shown to demonstrate the difference between these two grammars.

1. START → ADD .
2. ADD → MUL ADDx .
3. ADDx → "+" MUL ADDx .
4. ADDx → "-" MUL ADDx .
5. ADDx → ε .
6. MUL → FACT MULx .
7. MULx → "*" FACT MULx .
8. MULx → "/" FACT MULx .
9. MULx → "%" FACT MULx .
10. FACT → UNARY FACTx .
11. FACTx → "!" FACTx .
12. FACTx → ε .
13. UNARY → "-" UNARY .
14. UNARY → "+" UNARY .
15. UNARY → POW .
16. POW → TERM POWx .
17. POWx → "^" POW .
18. POWx → ε.
19. TERM → "unary" "(" ADD ")" .
20. TERM → "binary" "(" ADD "," ADD ")" .
21. TERM → "(" ADD ")" .
22. TERM → "number" .

Figure 3.5 LL(1) grammar for mathematical expressions

Left associativity	Right associativity
ADD → TERM . ADD → ADD "+" TERM .	ADD → TERM . ADD → TERM "+" ADD .
ADD → TERM ADD' . ADD' → "+" TERM ADD' . ADD' → .	ADD → TERM ADD' . ADD' → "+" ADD . ADD' → .

Table 3.2 Grammar comparison between left-associativity and right-associativity

A custom code generation tool has been developed to generate a parser running entirely in the TypeScript type system from the provided LL(1) grammar, using the aforementioned algorithm for creating the parsing table and appropriate recursive descent parser. The interface of a parser is defined as a generic type `Parser`, accepting a tuple of lexer tokens and a possible output AST node type as type parameters, seen in Listing 3.31. The generic type returns an object type, with an additional `head` property for simplifying the matching of the current lookahead token needed by the LL(1) parser. `ConsumeParser` is a generic type for consuming a token from the input stream and returning a new object type with the rest of the token stream.

Listing 3.31 Core parser interface

```
interface Parser<T extends Token._[] = Token._[], A extends AST._ = AST._> {
  tokens: T
  head: T[0]
  return: A
}

type ConsumeParser<
    Match extends Token._,
    TParser extends Parser
  > = TParser["head"] extends Match
    ? TParser["tokens"] extends [Token._, ...infer Rest extends Token._[]]
      ? Parser<Rest, TParser["return"]>
      : []
    : Error.Parser
```

With the following building blocks, it is possible to write a recursive descent parser based on the obtained parser table. An example can be seen in Listing 3.32, where a non-terminal `POW` and `POWx` is transformed into generic types accepting a type instance of `Parser` as the type parameter. The generic type attempts to match a lexer token by performing an assignability check and if succeeded, either the token can be consumed by using `ConsumeParser`, yielding a new parser to work with, or the parser can be passed on to the next generic type. The `ReturnParser` generic type reassigns the AST node, essentially acting as a way to return a value from a generic type.

3.6.3 Evaluator

Finally, the evaluator takes the AST returned by the parser as the input and returns a string literal type containing the result of the expression.

As the AST does already take operator precedence and associativity into account, the evaluator itself only recursively traverses the tree, visiting each of the AST nodes and performing the appropriate operation by pattern matching. A shortened example can be seen in Listing 3.33.

The evaluator itself is not required per se, and the expression can be evaluated directly in the parser, but to avoid the instantiation depth limit and to simplify debugging and unit testing, the parser emits an AST as a temporary result, and the evaluation is performed in a separate step. This does have the additional benefit of simplifying testing of the entire parsing mechanics, as the AST can be easily inspected and compared to the expected result.

Listing 3.32 Implementation of exponentiation parser

```
type POWx<T extends Parser> = T["head"] extends Token.Power
  ? ConsumeParser<Token.Power, T> extends infer T extends Parser
    ? POW<T> extends infer R extends Parser
      ? ReturnParser<R, AST.Binary<T["return"], "^", R["return"]>>
      : Error.Parser
    : Error.Parser
  : T["head"] extends
      | Token.EOF | Token.Factorial | Token.Multiply
      | Token.Divide | Token.Modulo | Token.Plus
      | Token.Minus | Token.RightBracket | Token.Comma
  ? T
  : Error.Parser

type POW<T extends Parser> = T["head"] extends
  | Token.UnaryFunction | Token.BinaryFunction | Token.LeftBracket | Token.Number
  ? TERM<T> extends infer T extends Parser
    ? POWx<T> extends infer T extends Parser
      ? T
      : Error.Parser
    : Error.Parser
  : Error.Parser
```

3.7 Higher kinded types

Higher kinded types (HKT), also known as higher-order types, are a powerful type system language feature that enables describing expressive generic types by allowing accepting other generic types as type arguments. To demonstrate, consider the following Listing 3.34. As can be seen, all three generic types do essentially the same type instantiation, only with different type constructors.

With HKTs, it is possible to define a single higher-order generic type, that accepts a type constructor as an argument. The type constructor is then applied to each property of the object type. The result is shown in Listing 3.35.

With higher kinded types, it is possible to declare a monad generic type [43] or applicative functors [44], design patterns commonly found in functional programming languages such as Haskell or Scala.

However, as of writing, higher-kinded types are not natively supported by TypeScript [45]. Fortunately, it is possible to emulate the behaviour of higher kinded types.

There are two ways to achieve the behaviour of HKT. One such way can be achieved by implementing lightweight higher-kinded polymorphism [46] and defunctionalisation of kinds [47], a technique for translating higher-order programs into a first-order language. The main idea is to create a mapping of unique names of type constructors to their implementations. Afterwards, a `Kind` utility converts the name and the appropriate type argument to the corresponding higher-kinded type. An example can be seen in Listing 3.36.

This method is historically used in libraries for typed functional programming such as `fp-ts` [48]. Unfortunately, this method requires a central registry of URIs that are used to identify the appropriate type constructor and extendability based on module augmentation is limited.

Listing 3.33 Evaluator example

```
export type Evaluate<T> = T extends AST.Binary<
  infer Left,
  infer Op,
  infer Right
>
  ? Op extends "+"
    ? Evaluate<Left> extends infer LeftStr extends NumberLike
      ? Evaluate<Right> extends infer RightStr extends NumberLike
        ? Add<LeftStr, RightStr>
        : never
      : never
    : never
  : T extends AST.Number<infer Value extends string>
  ? Value
  : never
```

Listing 3.34 Duplicate generic types

```
type Foo<O> = O extends string ? `Foo<${O}>` : never
type Bar<O> = O extends string ? `Bar<${O}>` : never
type Baz<O> = O extends string ? `Baz<${O}>` : never

type MapValuesWithFoo<O> = { [K in keyof O]: Foo<O[K]> }
type MapValuesWithBar<O> = { [K in keyof O]: Bar<O[K]> }
type MapValuesWithBaz<O> = { [K in keyof O]: Baz<O[K]> }
```

The other possible method for implementing HKTs is by utilising the properties of type unification with `this`. This method is thoroughly used in HOTscript [49] and a simplified implementation can be seen in Listing 3.37.

The most popular implementation of the latter method, HOTScript, exposes most of the core functionality of the library as a public facing API. Thus, an additional public facing API for mathematical operations has been exposed for users of HOTScript, extending the library with an advanced mathematical expression evaluator implemented in this work.

Listing 3.35 Proposed HKT syntax in TypeScript

```
type MapValuesWith<O, T<~>> = { [K in keyof O]: T<O[K]> }

type MapValuesWithFoo<O> = MapValuesWith<O, Foo>;
type MapValuesWithBar<O> = MapValuesWith<O, Bar>;
type MapValuesWithBaz<O> = MapValuesWith<O, Baz>;
```

Listing 3.36 HKT emulation using lightweight higher-kinded polymorphism

```
type URItoKind<A> = { "Foo": Foo<A>; "Bar": Bar<A>; "Baz": Baz<A> }
type URI = keyof URItoKind<unknown>
type Kind<F extends URI, A> = URItoKind<A>[F];

type MapValuesWith<O, Type extends URI> = Kind<Type, O>
```

Listing 3.37 Type unification for emulating HKTs

```
interface Fn { input: unknown; output: unknown; }

type Call<fn extends Fn, input> = (fn & { input: input })["output"];

interface Foo extends Fn {
  output: this["input"] extends infer O extends string ? `Foo<${O}>` : never;
}

interface Bar extends Fn {
  output: this["input"] extends infer O extends string ? `Bar<${O}>` : never;
}

interface Baz extends Fn {
  output: this["input"] extends infer O extends string ? `Baz<${O}>` : never;
}

type MapValuesWith<O, Wrap extends Fn> = {
  [K in keyof O]: Call<Wrap, O[K]>
}
```

Chapter 4

Development Tooling and Testing

4.1 Testing and development

During the development of the type-level mathematical expression evaluator, several invaluable tools were discovered and utilised that significantly contributed to the implementation. This section is devoted to discussing these tools and their impact on the overall development process.

The core of the development experience is underpinned by TypeScript Standalone Server, also known as `tsserver`. `tsserver` encapsulates both the compiler and the accompanying language services for use in editors and IDEs, communicating via LSP to add support for code completion, auto-importing, symbol renaming etc. `tsserver` also provides the ability to see the inferred types of any symbol by hovering on top the symbol, as seen in Figure 4.1. This service is invaluable when developing a type-level library, as it allows the developer to break down complex types into smaller pieces, achieving better readability.

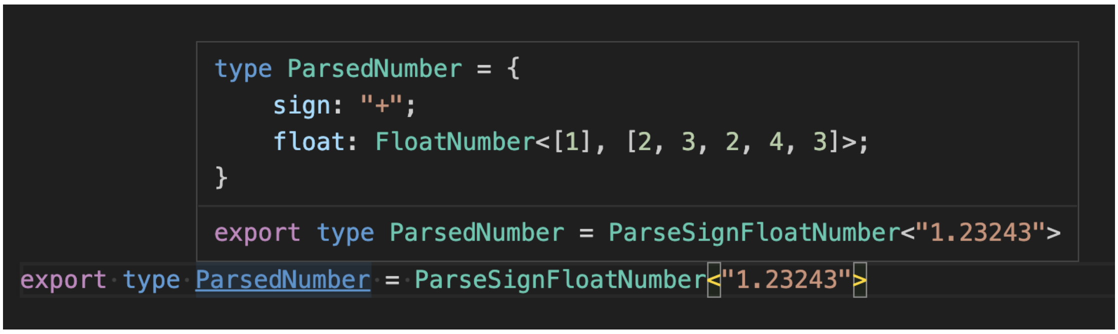

Figure 4.1 Inferred type on hover in VSCode

Another critical tool used when developing the implementation is `vscode-twoslash-plugin` extension [50]. In order to avoid hovering the mouse over a symbol to see the inferred type, developers can write the `// ^?` comment, with the caret pointing to the targeted symbol. The plugin will then display an inlay hint with the inferred type of the selected symbol, as seen in Figure 4.2.

Finally, Pretty TypeScript Errors [51] attempts to parse and reformat the TypeScript error messages to be more human-readable in VSCode. This is especially helpful when dealing with

Figure 4.2 Twoslash syntax of `vscode-twoslash-plugin`

complex object types, where the error messages can become unreadable since the error message and the serialised type is printed out on a single line, as seen in Figure 4.3.

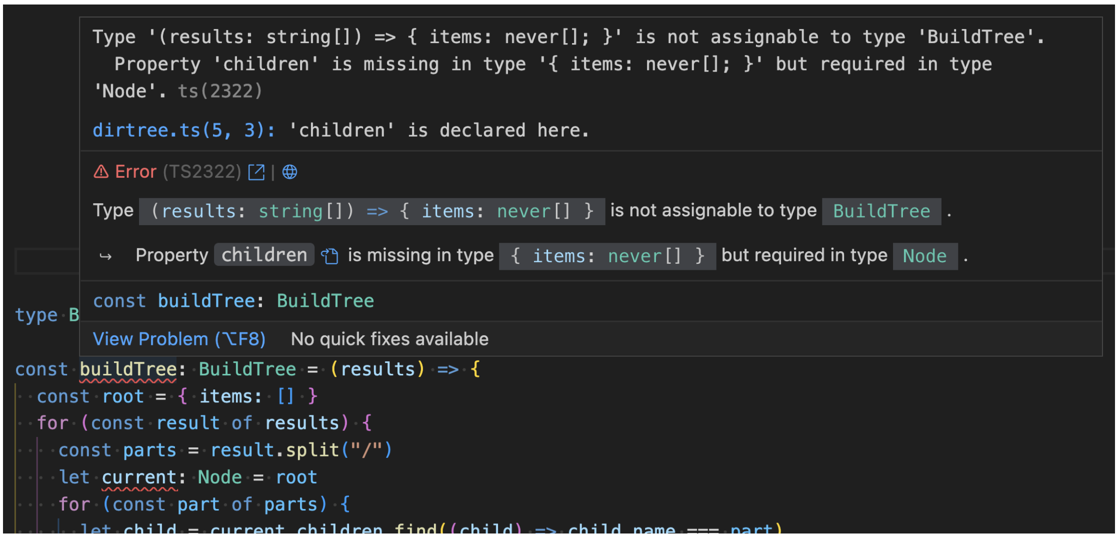

Figure 4.3 Formatting errors with Pretty TS Errors extension

Some generic types include an accompanying unit test to ensure correctness and prevent regression. Testing is backed with `eslint`[52], a static code analyser for JavaScript and TypeScript. Configuration-wise, `@typescript-eslint/parser` has been set up as the parser used by ESLint for properly analysing TypeScript code, and `eslint-plugin-expect-type` has been added for writing type assertions as comments. `eslint-plugin-expect-type` enables writing `$ExpectType`, `$ExpectError` and twoslash type assertions (`// ^?`). An example test assertion can be seen in Listing 4.1.

Listing 4.1 Type assertion with `$ExpectType`

```
// $ExpectType "0.3619047620"
type EvaluateCase = Evaluate<
  RecursiveParser.Parse<Lexer<"3.1 + 2.5 * (1 - 5.6) / 4.2">>
>
```

4.2 CI/CD workflow and release management

Continuous integration (CI) and Continuous delivery are the two key parts of the software development process that help developers deliver high-quality software. Continuous integration (CI) is the practice of automating the integration of code changes into a version control repository [53], encouraging developers to merge their changes to the main branch as often as possible. CI establishes an automated method for building, packaging and testing the software. The main benefit of this approach is to avoid major integration challenges when releasing a version by continuously integrating more minor changes during the development instead of doing all the integration on the release day.

Whereas Continuous delivery (CD) is an extension of continuous integration, where the code changes are automatically deployed to the production environment after the build and test stage. CD aims to simplify the deployment as much as possible, making it a routine process that can be performed as many times as needed, even multiple times during a day [54]. Note that there is a distinction between Continuous Delivery and Continuous Deployment where the former requires human intervention to deploy changes to production, and the latter is fully automated without any manual steps.

Both Continuous Integration and Continuous Delivery are set up in the implementation part of the thesis. The core of the CI/CD setup is the Github Action platform. The GitHub Actions platform allows developers to automate the build, test, and deployment pipeline within an existing GitHub repository [55]. The main components of GitHub Actions include workflows, jobs and actions defined using YAML files saved in the `.github/workflows`. This project uses two workflows, one for running unit and integration tests and a second one for performing Continuous delivery (CD) to the NPM registry.

The first workflow, found in `.github/workflows/main.yml`, runs both `yarn run test` and `yarn run build` after every push to the repository event, regardless of branch or reference. The second workflow, found in `.github/workflows/publish.yml`, is responsible for handling releases to the NPM registry using Changesets [56]. Changesets allow developers to keep track of the release history of a package and automate both versioning and release note generation.

The Changesets tool works by separating versioning into two stages: adding a changeset, describing the changes made in a commit or a branch, and combining created changesets with a version incrementing.

Creation of a changeset is done by running `yarn changeset`, which will ask the developer to provide the appropriate version bump type (either MAJOR, MINOR or PATCH, following the Semver versioning) and a message describing the changes. The changeset will be saved as a Markdown file with a unique identifier in the `.changesets` folder. The file will be committed to the Git repository. These changesets are preserved in the repository until the release is ready to be published by merging the changesets into the `main` branch.

After a push event to the `main` branch occurs, the release process, defined as a GitHub Actions job, is launched. The release process itself works by running `yarn changeset publish` command and works as such; When new changesets are found in the `main` branch, Changesets will automatically create a new pull request, which will perform all of the key steps for releasing a package: incrementing the version, updating the `CHANGELOG.md` file and removing the accumulated changelogs. When the pull request is merged, Changesets will automatically publish the new version to the NPM registry, using the granular access token provided as a secret variable for CI, and create an appropriate Git tag for the release. The final package is published to the NPM registry under the name `ts-math-evaluate` [57].

4.3 Performance testing

Advanced utility types do have a significant strain on type checking and can have a negative impact on the developer experience with worse latency of language services and longer build times when building with `tsc`. The performance test suite has been created to measure the impact of various implemented math operations on type-checking performance.

Two metrics are measured in the performance test suite: the "check time" obtained from extended diagnostics when compiling via `tsc` and the number of type instantiations performed when evaluating utility types. These metrics can be obtained from the `tsc` CLI with the `--extendedDiagnostics` flag. However, the TypeScript API does expose an internal `performance` singleton, which, combined with internal `extendedDiagnostics` flag and Compiler API, can be used to obtain the same metrics programmatically, as seen in Listing 4.2.

Listing 4.2 Programmatic access to internal extended performance metrics

```
import * as ts from "typescript"
const performance = (ts as any).performance

performance.enable()
const program = ts.createProgram(fileNames, {
  noEmit: true,
  incremental: false,
  extendedDiagnostics: true,
})
program.emit()

console.log(`Instantiation count: ${program.getInstantiationCount()}`)
console.log(`Check time: ${performance.getDuration("Check")}`)
performance.disable()
```

Together with `ts-morph` library [58] and the insights from the Compiler API, a benchmarking tool was created, found in `scripts/bench.ts`. `ts-morph` is a wrapper around the Compiler API that provides convenient methods for setup, navigation and manipulation of the TypeScript AST. The benchmarking tool accepts a path to a benchmarking file and parses the file into an AST. Each test case of a benchmark file is denoted as an exported type alias, which is read by the benchmarking tool. The tool then creates a new separate valid TypeScript code for each test case, containing just the benchmark type alias, omitting all other unnecessary types and constructs. The evaluation of a test case follows the same logic as described in Listing 4.2. The tool performs multiple measurement iterations, and both the mean and variance are calculated for each metric. At the time of writing, the benchmarking tool performs twelve iterations in total, with two iterations being warmup iterations. The idea of warmup iterations is to increase the likelihood of the JavaScript engine deciding to optimise the interpreted code.

In order to measure the impact of the library on type checking, some mathematical operations were selected for benchmarking: `Add`, `Multiply`, `Divide` and `Root`, ordered by the increasing computational complexity. As shown in Figure 4.4, the number of type instantiations proportionally increases with the digit length. As expected, `Root`, the most complex operations of the selected few, creates an order of magnitude more type instantiations than other operations.

However, when comparing the actual time spent by the type checker, there does not seem to be a strong indication of performance degradation when comparing check times between `Add`, `Multiply` and `Divide`. Only the `Root` operation does seem to have a significant negative impact on the type-checking performance. This requires further investigation out of the scope of this thesis. However, there does seem to be a significant performance hit when the number of type instantiations is in the scale of millions. For full benchmarking results, refer to the Table A.1,

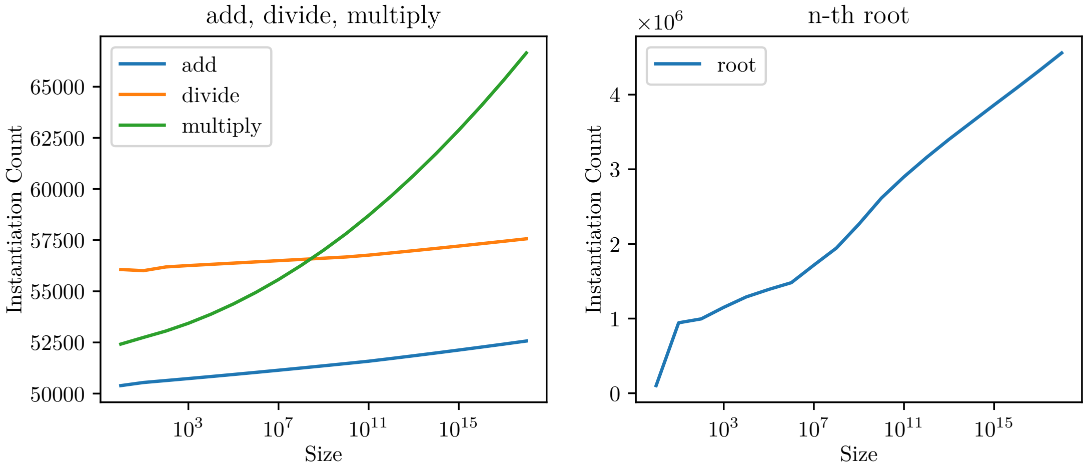

Figure 4.4 Comparison of instantiation count for selected operations

Table A.2, Table A.3, and Table A.4.

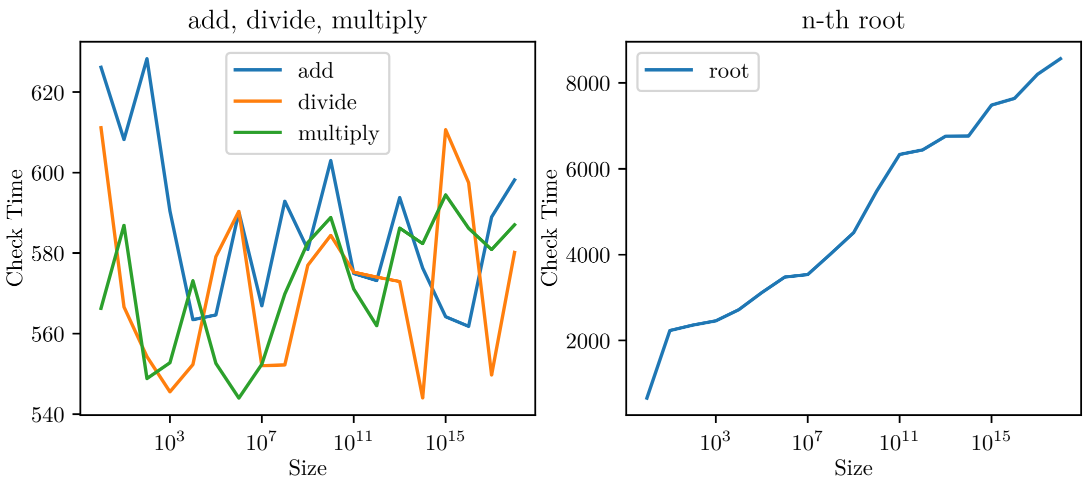

Figure 4.5 Comparison of time spent type checking between selected operations

Chapter 5
Conclusion

This thesis set out to implement a mathematical expression evaluator entirely written in the type system of TypeScript. Core concepts and techniques of TypeScript type-level programming were introduced and explained. The implementation of the expression evaluator was described in detail, and the implementation was evaluated in terms of correctness and performance using type-level unit tests and benchmarking suites.

The created evaluator is a proof-of-concept, demonstrating the capabilities of the TypeScript type system while addressing some of the limitations of the type system by applying workarounds.

This thesis also provides a comprehensive guide to the TypeScript syntax and type system and can be used as a reference for beginners to the type-level programming in TypeScript. Additional tools and utility types were introduced to aid the development of the mathematical evaluator, namely the benchmarking tool and the LL(1) parser generator.

The rest of this chapter will discuss both the practicality of the created types and the limitations of the type system found during development. Finally, the future work will be outlined.

5.1 Practical usage

The TypeScript type system is powerful for static type checking and inference. However, it is not without its limitations. These advanced types are considered to be extreme and are generally not recommended to be used in production code, as they can severely impact the compilation time and the in-editor developer experience.

Nevertheless, there are some possible practical use cases for these advanced types. Literal types are often used to describe a design system and accompanying design tokens. Namely, numeric literal types are used to describe the spacing and sizing of components. When the spacing is defined in other units, such as `rem` or `em`, developers often need to manually convert the values into pixels. A utility type can be introduced to convert values in `rem` or `em` units into pixels and reverse. This can be further expanded to allow more type transformations, such as converting a Tailwind CSS class name into a CSS string without any TypeScript editor plugins.

The parser and the accompanying parser generator can accept any LL(1) grammar and can be used to parse more complex formats, such as JSON. Finally, the benchmarking tool can be used to benchmark any type-level code in isolation, keeping all the test cases in a single file.

5.2 Limitations of the TypeScript type system

When developing the implementation of the evaluator, some limitations of the TypeScript type system were discovered.

In general, error messages in TypeScript are suboptimal. They tend to be displayed in one line without any formatting, and if they include complicated types, the types are truncated, which leads to a suboptimal debugging experience. Even with `noErrorTruncation` flag turned on in `tsconfig.json`, the type message is still truncated due to a hard limit. The limit can be artificially raised by patching the TypeScript source code, namely by increasing the `defaultMaximumTrucationLength` limit, but this is not a viable solution for production code. The only other option is to manually recreate intermediate types when debugging complicated types.

The type checker itself contains many hardcoded constraints to prevent performance degradation, ranging from the maximum tuple size to the limit on both instantiation count and depth. Some checks can be bypassed using various workarounds, often at a performance cost, discussed in previous chapters. However, these workarounds are poorly documented in the official TypeScript documentation and can break with new TypeScript releases without further notice.

Even though the TypeScript type system is powerful for complex types, some highly requested features are still missing as of the writing of this thesis, such as the lack of partial type argument inference [59] or lack of built-in utility types for type-level assertions. Some of these features can be partially emulated, such as the lack of higher kinded types, but the behaviour can also change with new TypeScript releases.

Finally, as the type checker itself is written in TypeScript to dogfood the language, it can be inherently slow when working with larger TypeScript codebases. Some of these performance issues are being solved by rewriting the type checker in a different programming language, such as Rust [60], but the project is still under active development.

5.3 Future work

Most of the future work is geared towards the underlying tooling and utilities rather than the mathematical expression evaluator itself, which can be further extended by adding additional mathematical operations based on the existing utility types implemented in this thesis.

For instance, the LL(1) parser generator is not flexible enough, as it can only generate code for LL(1) grammars, which, while being sufficient for mathematical expressions and other simple formats such as JSON, is not enough for more complex grammars. Future work could include creating a more generic Look-Ahead LR parser generator, which would be able to parse more complex grammars and programming languages.

The benchmarking utility itself can be extended and packaged both as an NPM package and as a GitHub Action. This is especially useful for library maintainers, which can use the additional CI step to monitor potential performance regressions when reviewing pull requests from contributors.

Appendix A

Performance measurements

	Instantiation Count	Check Time (ms)
`Add<"1", "1">`	50389	626.1480 ± 2732.4835
`Add<"1", "10">`	50541	608.1964 ± 1490.2754
`Add<"1", "100">`	50637	628.2837 ± 1187.9201
`Add<"1", "1000">`	50734	590.4544 ± 1323.4114
`Add<"1", "10000">`	50833	563.4551 ± 503.1991
`Add<"1", "100000">`	50934	564.6383 ± 326.1892
`Add<"1", "1000000">`	51037	590.2288 ± 800.6683
`Add<"1", "10000000">`	51142	566.8994 ± 1322.4848
`Add<"1", "100000000">`	51249	592.8983 ± 2703.7284
`Add<"1", "1000000000">`	51358	580.8956 ± 732.7624
`Add<"1", "10000000000">`	51469	602.9816 ± 2772.3885
`Add<"1", "100000000000">`	51582	574.9316 ± 443.2602
`Add<"1", "1000000000000">`	51714	573.1765 ± 288.5311
`Add<"1", "10000000000000">`	51849	593.7831 ± 815.4532
`Add<"1", "100000000000000">`	51987	576.2892 ± 1581.0826
`Add<"1", "1000000000000000">`	52128	564.1985 ± 226.5942
`Add<"1", "10000000000000000">`	52272	561.8088 ± 202.4868
`Add<"1", "100000000000000000">`	52419	588.9620 ± 1605.4446
`Add<"1", "1000000000000000000">`	52569	598.1670 ± 2184.9193

Table A.1 Instantiation count and check time for `Add`

	Instantiation Count	Check Time (ms)
`Multiply<"1", "1">`	52418	566.2856 ± 1304.8233
`Multiply<"1", "10">`	52742	586.9069 ± 2054.9946
`Multiply<"1", "100">`	53057	548.8409 ± 291.8180
`Multiply<"1", "1000">`	53436	552.7645 ± 1010.9699
`Multiply<"1", "10000">`	53879	573.1269 ± 1982.9075
`Multiply<"1", "100000">`	54383	552.6068 ± 404.8264
`Multiply<"1", "1000000">`	54948	543.9872 ± 156.1134
`Multiply<"1", "10000000">`	55574	552.3896 ± 391.9594
`Multiply<"1", "100000000">`	56261	569.8469 ± 1450.4120
`Multiply<"1", "1000000000">`	57009	582.6160 ± 2282.8004
`Multiply<"1", "10000000000">`	57818	588.8492 ± 2708.3371
`Multiply<"1", "100000000000">`	58705	571.0721 ± 1108.1628
`Multiply<"1", "1000000000000">`	59654	561.9530 ± 502.5156
`Multiply<"1", "10000000000000">`	60665	586.2230 ± 1929.5835
`Multiply<"1", "100000000000000">`	61738	582.3424 ± 1791.3684
`Multiply<"1", "1000000000000000">`	62873	594.4613 ± 2638.3184
`Multiply<"1", "10000000000000000">`	64070	586.1715 ± 1270.7923
`Multiply<"1", "100000000000000000">`	65329	580.8701 ± 641.9969
`Multiply<"1", "1000000000000000000">`	66650	587.0313 ± 1519.6371

Table A.2 Instantiation count and check time for `Multiply`

	Instantiation Count	Check Time (ms)
`Divide<"1", "3">`	56065	611.1183 ± 3712.1221
`Divide<"10", "3">`	56007	566.6266 ± 1312.9779
`Divide<"100", "3">`	56190	554.2810 ± 178.3204
`Divide<"1000", "3">`	56257	545.5406 ± 1368.3422
`Divide<"10000", "3">`	56317	552.3237 ± 1120.0729
`Divide<"100000", "3">`	56377	579.1138 ± 841.8474
`Divide<"1000000", "3">`	56437	590.3914 ± 2730.2088
`Divide<"10000000", "3">`	56497	552.0362 ± 983.1479
`Divide<"100000000", "3">`	56557	552.2182 ± 1549.2122
`Divide<"1000000000", "3">`	56617	576.9744 ± 1538.6618
`Divide<"10000000000", "3">`	56677	584.4138 ± 1771.3998
`Divide<"100000000000", "3">`	56767	575.2797 ± 2817.7110
`Divide<"1000000000000", "3">`	56875	574.0339 ± 1187.0906
`Divide<"10000000000000", "3">`	56985	572.9396 ± 1845.7412
`Divide<"100000000000000", "3">`	57097	544.0291 ± 187.0663
`Divide<"1000000000000000", "3">`	57211	610.6468 ± 1628.6978
`Divide<"10000000000000000", "3">`	57327	597.4917 ± 751.0289
`Divide<"100000000000000000", "3">`	57445	549.7234 ± 404.5449
`Divide<"1000000000000000000", "3">`	57565	580.1838 ± 1396.0654

Table A.3 Instantiation count and check time for `Divide`

	Instantiation Count	Check Time (ms)
`Root<"1", "2">`	101781	657.9258 ± 4637.2670
`Root<"10", "2">`	943579	2231.2624 ± 26354.7449
`Root<"100", "2">`	995709	2358.4428 ± 46642.8882
`Root<"1000", "2">`	1150338	2457.2318 ± 7974.3022
`Root<"10000", "2">`	1290084	2717.1363 ± 22569.9561
`Root<"100000", "2">`	1390432	3114.9099 ± 89929.3402
`Root<"1000000", "2">`	1480678	$3476.8778 \pm 132963.1171$
`Root<"10000000", "2">`	1715701	$3535.6702 \pm 217462.3269$
`Root<"100000000", "2">`	1944285	4017.5913 ± 22143.6082
`Root<"1000000000", "2">`	2264897	4512.2763 ± 45757.7304
`Root<"10000000000", "2">`	2614395	5477.2640 ± 64109.5688
`Root<"100000000000", "2">`	2898459	6333.6284 ± 9891.4644
`Root<"1000000000000", "2">`	3157392	6439.0302 ± 36044.4666
`Root<"10000000000000", "2">`	3400798	6758.4044 ± 84619.0587
`Root<"100000000000000", "2">`	3630962	6763.4433 ± 32105.0016
`Root<"1000000000000000", "2">`	3861157	7483.2405 ± 52541.9327
`Root<"10000000000000000", "2">`	4087750	$7638.8097 \pm 163773.6981$
`Root<"100000000000000000", "2">`	4319968	$8199.8814 \pm 306798.3177$
`Root<"1000000000000000000", "2">`	4558096	$8563.6442 \pm 363040.7800$

Table A.4 Instantiation count and check time for `Root`

Bibliography

1. JSWORLD CONFERENCE (director). *Fred K. Schott - Type-safety Is Eating the World* [online]. 2023. [visited on 2023-03-25]. Available from: https://www.youtube.com/watch?v=DqYxbjTM2vw.
2. *The State of JS 2022: Usage* [online]. [visited on 2023-03-25]. Available from: https://2022.stateofjs.com/en-US/usage/.
3. *Prisma/Prisma: Next-generation ORM for Node.Js & TypeScript — PostgreSQL, MySQL, MariaDB, SQL Server, SQLite, MongoDB and CockroachDB* [online]. [visited on 2023-03-25]. Available from: https://github.com/prisma/prisma.
4. MCDONNELL, Colin. *Zod* [online]. 2023. [visited on 2023-03-25]. Available from: https://github.com/colinhacks/zod.
5. *tRPC* [online]. tRPC, 2023. [visited on 2023-03-25]. Available from: https://github.com/trpc/trpc.
6. *Stack Overflow Developer Survey 2022* [online]. Stack Overflow. [visited on 2023-01-29]. Available from: https://survey.stackoverflow.co/2022/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2022.
7. SCHUMACHER, Toni. Concepts of Programming Languages: - Static vs. Dynamic Typing. 2015. Available also from: https://www.isp.uni-luebeck.de/sites/default/files/lectures/ws_2015_2016/CS%203702%20BachSemInf%2C%20%2C%20CS%203703%20BachSemMI%2C%20%2C%20CS%205480%20SemSSE%2C%20%2C%20CS%205840%20SemiEngl/schumacher-typing-slides.pdf.
8. *ECMAScript Proposal: Type Annotations* [online]. Ecma TC39, 2023. [visited on 2023-01-29]. Available from: https://github.com/tc39/proposal-type-annotations.
9. *Elm - Delightful Language for Reliable Web Applications* [online]. [visited on 2023-01-31]. Available from: https://elm-lang.org/.
10. *The Elm Architecture · An Introduction to Elm* [online]. [visited on 2023-01-31]. Available from: https://guide.elm-lang.org/architecture/.
11. *Prior Art — Redux* [online]. 2022-02-07. [visited on 2023-01-31]. Available from: https://redux.js.org/understanding/history-and-design/prior-art.
12. *BuckleScript & Reason Rebranding* [online]. ReScript Blog. [visited on 2023-01-29]. Available from: https://rescript-lang.org/blog/bucklescript-is-rebranding.

13. *Efficient and Insightful Generalization* [online]. [visited on 2023-02-12]. Available from: https://okmij.org/ftp/ML/generalization.html.
14. *History — ReScript* [online]. 2022-02-12. [visited on 2023-02-12]. Available from: https://github.com/rescript-lang/rescript-compiler/blob/master/CREDITS.md.
15. *Reconstructing TypeScript, Part 0: Intro and Background* [online]. [visited on 2023-01-24]. Available from: https://jaked.org/blog/2021-09-07-Reconstructing-TypeScript-part-0.
16. CHAUDHURI, Avik; VEKRIS, Panagiotis; GOLDMAN, Sam; ROCH, Marshall; LEVI, Gabriel. Fast and Precise Type Checking for JavaScript. *Proceedings of the ACM on Programming Languages* [online]. 2017, vol. 1, 48:1–48:30 [visited on 2023-01-29]. Available from DOI: 10.1145/3133872.
17. *Flow* [online]. Meta, 2023. [visited on 2023-01-29]. Available from: https://github.com/facebook/flow.
18. *TypeScript Design Goals* [online]. GitHub. [visited on 2023-01-29]. Available from: https://github.com/microsoft/TypeScript/wiki/TypeScript-Design-Goals.
19. *TypeScript: JavaScript With Syntax For Types* [online]. [visited on 2023-02-01]. Available from: https://www.typescriptlang.org/.
20. *Documentation - TypeScript for JavaScript Programmers* [online]. [visited on 2023-01-14]. Available from: https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html.
21. *Babel/Babel* [online]. Babel, 2023. [visited on 2023-02-01]. Available from: https://github.com/babel/babel.
22. *Esbuild - An Extremely Fast Bundler for the Web* [online]. [visited on 2023-02-01]. Available from: https://esbuild.github.io/.
23. *SWC - Rust-based Platform for the Web* [online]. [visited on 2023-02-01]. Available from: https://swc.rs/.
24. *Visual Studio Code - Code Editing. Redefined* [online]. [visited on 2023-02-01]. Available from: https://code.visualstudio.com/.
25. *Octoverse 2022: The State of Open Source* [online]. The State of the Octoverse. [visited on 2023-01-29]. Available from: https://octoverse.github.com/.
26. *Standalone Server (Tsserver)* [online]. GitHub. [visited on 2023-03-27]. Available from: https://github.com/microsoft/TypeScript/wiki/Standalone-Server-(tsserver).
27. VANDERKAM, Dan. *Effective TypeScript: 62 Specific Ways to Improve Your TypeScript*. First edition. Beijing [China] ; Sebastopol, CA: O'Reilly Media, 2019. ISBN 978-1-4920-5374-3.
28. *Documentation - Declaration Merging* [online]. [visited on 2023-04-10]. Available from: https://www.typescriptlang.org/docs/handbook/declaration-merging.html#module-augmentation.
29. *Performance* [online]. GitHub. [visited on 2023-04-02]. Available from: https://github.com/microsoft/TypeScript/wiki/Performance.
30. *Documentation - Everyday Types* [online]. [visited on 2023-03-27]. Available from: https://www.typescriptlang.org/docs/handbook/2/everyday-types.html.

31. *The Top Types ‘any‘ and ‘unknown‘ in TypeScript* [online]. [visited on 2023-03-26]. Available from: `https://2ality.com/2020/06/any-unknown-typescript.html`.

32. *Type Inference for Higher-Order, Generic Curried Function Breaks down When the Function Is Applied to Another Generic Function · Issue #49312 · Microsoft/TypeScript* [online]. GitHub. [visited on 2023-03-28]. Available from: `https://github.com/microsoft/TypeScript/issues/49312`.

33. ROSENWASSER, Daniel. *Announcing TypeScript 4.7* [online]. TypeScript, 2022-05-24. [visited on 2023-03-29]. Available from: `https://devblogs.microsoft.com/typescript/announcing-typescript-4-7/`.

34. ROSENWASSER, Daniel. *Announcing TypeScript 4.1* [online]. TypeScript, 2020-11-19. [visited on 2023-03-29]. Available from: `https://devblogs.microsoft.com/typescript/announcing-typescript-4-1/`.

35. *Implementation of Checker.Ts* [online]. Microsoft, 2023. [visited on 2023-03-31]. Available from: `https://github.com/microsoft/TypeScript/blob/55867271933d603f6c29b8eb7399960a71e96ccc/src/compiler/checker.ts`.

36. *Documentation - Template Literal Types* [online]. [visited on 2023-04-02]. Available from: `https://www.typescriptlang.org/docs/handbook/2/template-literal-types.html`.

37. *Type-Challenges/Type-Challenges* [online]. Type Challenges, 2023. [visited on 2023-04-02]. Available from: `https://github.com/type-challenges/type-challenges`.

38. SORHUS, Sindre. *Sindresorhus/Type-Fest* [online]. 2023. [visited on 2023-04-02]. Available from: `https://github.com/sindresorhus/type-fest`.

39. KAWAYILINLIN. *kawayiLinLin/Typescript-Lodash* [online]. 2023. [visited on 2023-01-15]. Available from: `https://github.com/kawayiLinLin/typescript-lodash`.

40. ARIEL. *Type Level Arithmetic* [online]. 2023. [visited on 2023-01-15]. Available from: `https://github.com/arielhs/ts-arithmetic`.

41. SUTTER, Herb. Consistent Comparison. 2017.

42. GERLACH, Juergen. Accelerated Convergence in Newton’s Method. *Siam Review - SIAM REV.* 1994, vol. 36. Available from DOI: `10.1137/1036057`.

43. WADLER, Philip. Monads for Functional Programming. In: BROY, Manfred (ed.). *Program Design Calculi.* Berlin, Heidelberg: Springer, 1993, pp. 233–264. NATO ASI Series. ISBN 978-3-662-02880-3. Available from DOI: `10.1007/978-3-662-02880-3_8`.

44. MCBRIDE, Conor; PATERSON, Ross. Applicative Programming with Effects. *Journal of Functional Programming* [online]. 2008, vol. 18, no. 1, pp. 1–13 [visited on 2023-04-25]. ISSN 1469-7653, ISSN 0956-7968. Available from DOI: `10.1017/S0956796807006326`.

45. *Documentation - TypeScript for Functional Programmers* [online]. [visited on 2023-04-25]. Available from: `https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes-func.html`.

46. YALLOP, Jeremy; WHITE, Leo. Lightweight Higher-Kinded Polymorphism. In: CODISH, Michael; SUMII, Eijiro (eds.). *Functional and Logic Programming.* Cham: Springer International Publishing, 2014, pp. 119–135. Lecture Notes in Computer Science. ISBN 978-3-319-07151-0. Available from DOI: `10.1007/978-3-319-07151-0_8`.

47. REYNOLDS, John C. Definitional Interpreters for Higher-Order Programming Languages. In: *Proceedings of the ACM Annual Conference - Volume 2* [online]. New York, NY, USA: Association for Computing Machinery, 1972, pp. 717–740 [visited on 2023-04-25]. ACM '72. ISBN 978-1-4503-7492-7. Available from DOI: `10.1145/800194.805852`.
48. *Gcanti/Fp-Ts: Functional Programming in TypeScript* [online]. [visited on 2023-04-25]. Available from: `https://github.com/gcanti/fp-ts`.
49. VERGNAUD, Gabriel. *Higher-Order TypeScript (HOTScript)* [online]. 2023. [visited on 2023-04-25]. Available from: `https://github.com/gvergnaud/hotscript`.
50. THEROX, Orta. *Vscode-Twoslash-Queries* [online]. 2023. [visited on 2023-03-26]. Available from: `https://github.com/orta/vscode-twoslash-queries`.
51. BALASIANO, Yoav. *Pretty TypeScript Errors* [online]. 2023. [visited on 2023-04-27]. Available from: `https://github.com/yoavbls/pretty-ts-errors`.
52. *ESLint* [online]. ESLint, 2023. [visited on 2023-04-28]. Available from: `https://github.com/eslint/eslint`.
53. ATLASSIAN. *Continuous Integration vs. Delivery vs. Deployment* [online]. Atlassian. [visited on 2023-04-28]. Available from: `https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment`.
54. *What Is CI/CD, Continuous Integration and Continuous Delivery?* [online]. Cisco. [visited on 2023-04-28]. Available from: `https://www.cisco.com/c/en/us/solutions/data-center/data-center-networking/what-is-ci-cd.html`.
55. *Understanding GitHub Actions* [online]. GitHub Docs. [visited on 2023-04-28]. Available from: `https://ghdocs-prod.azurewebsites.net/en/actions/learn-github-actions/understanding-github-actions`.
56. *Changesets/Changesets* [online]. changesets, 2023. [visited on 2023-04-26]. Available from: `https://github.com/changesets/changesets`.
57. *Ts-Math-Evaluate* [online]. npm, 2023-04-12. [visited on 2023-04-29]. Available from: `https://www.npmjs.com/package/ts-math-evaluate`.
58. SHERRET, David. *Ts-Morph* [online]. 2023. [visited on 2023-05-01]. Available from: `https://github.com/dsherret/ts-morph`.
59. *Implement Partial Type Argument Inference Using the _ Sigil by Weswigham · Pull Request #26349 · Microsoft/TypeScript* [online]. GitHub. [visited on 2023-05-01]. Available from: `https://github.com/microsoft/TypeScript/pull/26349`.
60. *Stc* [online]. Dudy, 2023. [visited on 2023-04-29]. Available from: `https://github.com/dudykr/stc`.

Contents of the attached media

```
.changeset
  config.json
.eslintrc
.github
  workflows
    main.yml
    publish.yml
.gitignore
.npmignore
.prettierrc
.vscode
  settings.json
assets
  cover.svg
CHANGELOG.md
package.json
README.md
scripts
  dirtree.ts
  generate.ts
  parser.ts
src
  expression
    enum.ts
    evaluator.test.ts
    evaluator.ts
    lexer.test.ts
    lexer.ts
    parser.test.ts
    parser.ts
  hotscript.test.ts
  hotscript.ts
  index.ts
  math
    abs.test.ts
    abs.ts
    add.test.ts
```

```
│ │ │
│ │ ├── add.ts
│ │ ├── ceil.test.ts
│ │ ├── ceil.ts
│ │ ├── comparison.test.ts
│ │ ├── comparison.ts
│ │ ├── divide.test.ts
│ │ ├── divide.ts
│ │ ├── factorial.test.ts
│ │ ├── factorial.ts
│ │ ├── floor.test.ts
│ │ ├── floor.ts
│ │ ├── modulo.test.ts
│ │ ├── modulo.ts
│ │ ├── multiply.test.ts
│ │ ├── multiply.ts
│ │ ├── negate.test.ts
│ │ ├── negate.ts
│ │ ├── power.test.ts
│ │ ├── power.ts
│ │ ├── root.test.ts
│ │ ├── root.ts
│ │ ├── round.test.ts
│ │ ├── round.ts
│ │ ├── subtract.test.ts
│ │ ├── subtract.ts
│ │ ├── truncate.test.ts
│ │ └── truncate.ts
│ └── utils
│     ├── array.ts
│     ├── bits.ts
│     ├── boolean.ts
│     ├── map.ts
│     ├── parse.test.ts
│     └── parse.ts
└── thesis
    ├── .gitignore
    ├── .gitlab-ci.yml
    ├── assignment-include.pdf
    ├── ctufit-thesis.acn
    ├── ctufit-thesis.cls
    ├── ctufit-thesis.glo
    ├── ctufit-thesis.glsdefs
    ├── ctufit-thesis.ist
    ├── ctufit-thesis.pdf
    ├── ctufit-thesis.tex
    ├── indentconfig.yml
    ├── LICENSE
    ├── README.md
    ├── sources.bib
    └── text
        ├── analysis
        └── appendix.tex
```

```
│   ├── conclusion
│   │   └── conclusion.tex
│   ├── implementation
│   ├── introduction
│   │   └── introduction.tex
│   ├── medium.tex
│   └── testing
│       └── testing.tex
├── tsconfig.json
└── yarn.lock
```