CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Programming Language for Implementing Data Structures and Algorithms

Master's thesis in Computer science and engineering

Georgios Nikolaou

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

Programming Language for Implementing Data Structures and Algorithms

Georgios Nikolaou

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Programming Language for Implementing Data Structures and Algorithms
Georgios Nikolaou

Supervisors: Peter Ljunglöf, Department of Computer Science and Engineering
Niklas Deworetzki, Department of Computer Science and Engineering
Examiner: Ana Bove, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Programming Language for Implementing Data Structures and Algorithms
Georgios Nikolaou
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Existing programming languages used in Data Structures courses have drawbacks, as they were designed for general-purpose or professional use rather than for education. The goal of this thesis is to design a programming language to be used in a Data Structures course and in which data structures and algorithms can be implemented in a straightforward way. We identify the data structures and algorithms typically covered in a Data Structures course and derive the language constructs. The language design is guided by a set of principles, including the domain-specific principle of enabling straightforward implementations of data structures and algorithms, as well as principles of educational programming language design. We also provide an implementation of the front end of our language. Our analysis with respect to the design principles indicates that the goal is achieved to a large extent.

Keywords: programming language design, educational programming languages, data structures, compiler construction

Acknowledgements

I would like to thank my supervisors, Peter Ljunglöf and Niklas Deworetzki, for their guidance throughout this thesis, and my family for their continuous support during my studies.

Georgios Nikolaou, Gothenburg, 2026-06-22

Contents

1

Introduction

Python is a common language used in introductory programming courses. Its readable syntax, dynamic typing, and automatic memory management are features that make it popular with beginners to programming. However, many of Python's characteristics can become drawbacks when students try to implement data structures and algorithms (DSA) at a detailed level. In particular, its dynamic typing and runtime type checking make it difficult to detect type errors and easier to make mistakes when developing larger programs. A statically typed language, on the other hand, allows such errors to be detected promptly or prevented altogether. Also, Python's built-in data structures, which are implemented in C [1], make it difficult for students to reason about program behavior, as the underlying implementations of those data structures are not accessible. Their availability may also result in students resorting to pre-defined solutions, instead of attempting to implement them on their own. Moreover, Python's lack of low-level features, such as fixed-size arrays, as well as the use of syntactic sugar for lists, dictionaries, and iterators, make it harder to reason about memory use and program behavior, and therefore make understanding of DSA difficult in general.

Alternative languages to use in a Data Structures course include Java and C/C++, which both support fixed-size arrays and compile-time type checking. However, these languages are also far from ideal for this purpose. Java's [2] complex type and class systems require the use of a lot of boilerplate code, which is unnecessary and counterproductive in the process of studying and implementing DSA. On the other hand, C [3] and C++ [4] are too low-level, exposing implementation details related to memory management, among others, which impose unnecessary concerns on users trying to implement DSA, for example, having to deal with segmentation faults and undefined behavior. Also, general-purpose languages may report errors in a way that is less accessible to students, using error messages that contain technical terminology or special error codes, as these languages were not designed to be used in education.

The absence of a dedicated programming language intended specifically for implementing DSA is the core problem addressed in this thesis. In particular, our goal is to design and implement a programming language that can be used to implement DSA in the scope of a Data Structures course.

1.1 Goals

The research question addressed in this thesis is: *What constructs does a programming language need in order to be able to implement DSA in a straightforward way?* This naturally raises the subsequent question: *What principles should guide the design of such a language?*

The goal of the thesis is to design and implement a programming language to be used in a Data Structures course. In particular, we aim for programs implementing DSA in our language to closely resemble their specifications.

Additional objectives include:

- The language should have a simple syntax, so users already familiar with programming should have no apparent difficulty learning it.
- The language should provide detailed error messages for type errors, to help users identify such errors in their programs.

1.1.1 Analysis

We will analyze the language with respect to the set of principles that guided its design, as described in Section 2.3. In order to assess the correctness of the implementation, namely the front end of a transpiler for our language, an extensive testsuite will be developed to validate each implemented compilation stage.

1.1.2 Limitations

The implementation of the transpiler back end is not covered in this thesis. The development of a code generator for our language is currently in progress and will be presented as part of a subsequent thesis.

1.2 Related Work

There have been multiple instances of programming language design and implementation with educational objectives in mind. An example of such a language is Pascal [5], which places emphasis on a simple and structured syntax, and is intended for teaching programming. Blue [6] is an educational programming language that was designed with the purpose of teaching object-oriented programming in an introductory programming course. MuLE [7] is a multi-paradigm language designed for educational purposes [8], in particular to be used for teaching procedural, object-oriented and functional programming. To the best of our knowledge, there has not been an implementation of a programming language designed specifically for the purpose of implementing data structures and algorithms.

Principles of educational programming language design have been described in the literature. Black et al. [9] propose principles that should be followed when designing a new educational programming language. Kölling [10] discusses the principles

according to which a programming language used to teach programming to novices should be designed. The design criteria of our language are largely based on Kölling's paper, as discussed in Section 2.2.

1.3 Structure of the Thesis

The remainder of the thesis is structured as follows. Chapter 2 provides the necessary context for the subsequent chapters, covering compiler construction, the design principles, and our approach to the language design. Chapter 3 presents the design of our language, including its constructs, syntax, and type system. Chapter 4 covers the implementation of the transpiler front end. Chapter 5 analyzes the language design with respect to the principles, and assesses the implementation using a test-suite. Chapter 6 concludes the thesis by discussing limitations and trade-offs in the language design and outlining directions for future work.

2

Background

This chapter provides the context required for subsequent chapters of the thesis. In particular, it introduces fundamental concepts of compiler construction, presents the principles that guide our language design based on Kölling's paper [10], and describes our approach to the design of our language.

2.1 Compiler Construction

A compiler is a program that takes as input a program written in a specific language (the source language) and translates it to another language (the target language). A compiler is itself written in a programming language (the implementation language) which may be different from both the source and the target languages. Compilers consist of two main parts, the front end, where analysis of the source program is performed, and the back end, where synthesis of the target program is performed.

The front end consists of lexing, parsing, and semantic analysis. The lexer reads the input program as a string of characters and divides it into meaningful tokens. The parser uses the token sequence produced by the lexer to check that the input program conforms to the source language's syntax, and creates its abstract syntax tree (AST), which is a representation of the program corresponding to its structure. The type checker performs semantic analysis, which involves ensuring that the source program is type correct and gathering type information for use in subsequent phases. This information is either integrated in the AST as type annotations or stored in a dedicated structure called a symbol table.

Syntax-directed translation is a common technique used in type checkers. The type system of the source language is typically defined using inference rules, and the type checker implementation is based on these rules. In particular, the rules correspond to recursive operations over the abstract syntax tree. These operations involve inferring the type of an expression and verifying that an expression has the expected type. Type checking also relies on symbol tables or typing contexts, which are used to store and retrieve type information about declared variables and functions. In our language, we describe the type system using inference rules (Section 3.6), while our implementation follows the syntax-directed approach; the type checker traverses the AST produced by the parser, performs type checking using typing contexts, and constructs a type-annotated AST.

The back end consists of code generation and, optionally, code optimization. The code generator produces code in the target language using the syntax tree and the type information collected during semantic analysis. Code optimization involves performing various transformations to the code in the target language, usually to improve its performance or reduce its size.

Commonly, the output of a compiler is machine code or byte code, so that it can be directly executed by the CPU or by a virtual machine. It is also possible, however, that both the source and target languages are high-level languages. In these cases, the term "transpiler" can be used instead of "compiler".

A more detailed description of the compilation process can be found in the literature [11, 12].

The implementation of our language relies on established compiler construction techniques for the development of the front end, including the use of formal grammars (particularly in Backus-Naur form; BNF) and parser generator tools. In particular, the BNF Converter (BNFC [13]) and its underlying Labelled BNF (LBNF) formalism [14] facilitate the implementation of the compiler front end. BNFC takes the LBNF grammar of the source language and produces a lexer and a parser generator file, which are used by the respective generator tools to produce a lexer and a parser for that language. The files it generates also include the AST representation of programs in the source language. The use of automated tools ensures that the generated components directly reflect the formal grammar, significantly reducing the likelihood of errors in the lexical and syntactic analysis stages, thus simplifying the front end development process.

2.2 Language Design Principles

Kölling [10] discusses principles for the design of programming languages intended for teaching programming. The paper argues that programming education is dominated by languages that are popular in industry, despite the fact that such languages are designed and maintained for purposes different from those of educational programming languages. According to Kölling, the features and evolution of industry languages over time, in particular the introduction of new constructs and overall complexity, make them less suitable as teaching languages. The paper concludes that a new dedicated educational language should be designed for teaching programming, instead of continuing to rely on industry-oriented languages.

While the purpose of our language is not to teach introductory programming, but rather to be used in the context of a Data Structures course, its design is largely based on the principles discussed by Kölling. In the following sections, we present these principles, dividing them into those we consider in our language design and those we do not. We also introduce an additional domain-specific principle, reflecting the main goal of our language, as described in Section 1.1.

2.2.1 Educational Principles We Follow

The following principles from Kölling's paper [10] are taken into consideration when designing our language:

Don't Invent: Educational languages should be created using simplified versions of widely adopted concepts and constructs, avoiding unnecessary innovation and creativity. Our language will therefore rely on established constructs.

Programming for All Versus Programming for Some: It must be taken into consideration that programming is not a necessary skill only for those who will eventually become software developers. Educational languages should expose the necessary programming concepts without trying to prepare learners to become professionals in the field. In our language, the criteria for choosing constructs are strictly related to the purpose of implementing DSA.

Piggy-Backing on Existing Infrastructure: Educational programming languages should build on existing infrastructure, otherwise the cost of implementing all the necessary features would become infeasible. In particular, Kölling mentions class libraries as a piece of infrastructure that should not be re-implemented from scratch, but instead language designers should attempt to incorporate ready and widely-used libraries in their languages. In the case of our language, we do not need to use external libraries. Apart from the language constructs and a few necessary built-in definitions, implementation of DSA should be achievable without the use of additional pre-defined functions or types.

Nonetheless, we do rely on existing infrastructure, though in a different way from the one emphasized by Kölling [10]. In particular, we do not enable the use of existing infrastructure by the users of our language, but we rely on it for the transpiler implementation by choosing a high-level target language and utilizing its runtime infrastructure. This way, we avoid the need to re-implement established mechanisms, such as automatic memory management, including garbage collection.

Enforcing Good Practice: Educational languages should encourage users to follow good practice, which implies enforcing restrictions on the language in order to avoid bad programming practices wherever possible. For example, we aim to enforce explicit block structure and forbid executable statements outside of top-level definitions.

Programming is Creating Language: Educational programming languages should be extensible by user-defined abstractions that can be used as part of the syntax without special notation. Our language will support user-defined types that enable the implementation of data structures.

Keeping It Simple: Educational languages should avoid redundancy, keeping their set of constructs minimal. They should aim to offer exactly one way to solve a particular task. Our language will include only the necessary constructs so that its goals can be achieved.

2.2.2 Principles We Do Not Follow

We do not consider the following principles in our language design, as they involve the implementation of an integrated development environment that supports special features, which is outside the scope of this thesis.

Keywords Versus Symbols: The main argument for using symbols instead of keywords in a language syntax is conciseness, while a keyword-based syntax offers better readability. As Kölling suggests, there are ways to enhance the readability of programs beyond relying only on text-based representations. In particular, symbols and keywords can be replaced by graphical elements, as in block-based and frame-based [15] languages, such as Scratch [16] and Stride [17]. According to Kölling, educational languages should start incorporating graphical program representations.

Given that the users of our language are expected to be familiar with programming, we choose instead to focus the use of keywords on non-trivial constructs, such as user-defined types, while more common constructs, such as assignments and block scoping, will follow similar syntax to widely-used languages.

Separating the Intrinsic from the Accidental: This principles states that trivial programming errors, such as certain syntax errors, should not be the user's concern; instead, they should be prevented or resolved automatically by the programming environment, thus allowing users to focus on more important parts of the development process. Implementing such an environment for preventing or automating the correction of insubstantial syntax errors in our language could be explored in future work.

2.2.3 Domain-Specific Principle

Given our goal to design a programming language for implementing data structures and algorithms in the context of a Data Structures course, we guide our language design by a domain-specific principle, in conjunction with the educational principles described above.

Straightforward DSA Implementations: Selected constructs must enable data structure and algorithms to be expressible in a similar way to their specifications, so that the programs in our language resemble the structure of the pseudocode used to describe the DSA.

2.3 Our Approach

The goal of this work is to design a language in which data structures and algorithms can be implemented in a straightforward way. We will identify the DSA typically covered in a Data Structures course (Section 3.1), relying on the course book [18] used in the Data Structures course at Chalmers University and the University of Gothenburg. We will then derive the required language constructs (Sections 3.2 and 3.3), guided by the set of principles outlined in the previous section.

We will analyze the language design with respect to the adopted principles. For the domain-specific principle, this will involve structurally comparing implementations of the DSA requirements as programs in our language with their corresponding pseudocode specifications given in the course book [18]. For the educational principles, we will identify which design decisions follow each principle, and which violate it.

In order to assess the correctness of the transpiler implementation, an extensive testsuite will be developed to validate each implemented component, namely parsing and type checking. Specifically, the testsuite will confirm that:

- Syntactically correct programs are parsed successfully, while incorrect programs are rejected.
- Type-correct programs are accepted by the type checker, while programs with type errors are rejected.

We will also analyze the type error messages produced for the unit tests in the testsuite that contain type errors.

3

Language Design

In this chapter, we describe the design of our language. We start by documenting the required DSA that need to be implementable as programs in our language. Then, we derive the language constructs based on the principles listed in Section 2.2. We also present the syntax and describe the type system of our language.

3.1 Required Data Structures and Algorithms

In order to derive the set of constructs that our language needs, we must first identify and document the requirements. That is, all data structures and algorithms that should be implementable in this language. We document all DSA that are typically covered in a Data Structures course. In particular, we base this documentation process on the course book [18] used by the course in Data Structures and Algorithms at Chalmers University and the University of Gothenburg. Tables 3.1 and 3.2 list algorithms and abstract data types (ADTs), along with their respective data structure implementations. The listed DSA have been implemented as programs in our language, and the implementations will be used in the analysis of the language with respect to the domain-specific principle, as described in Chapter 5.

3.2 Deriving the Language Constructs

In this section, we describe the constructs included in our language, which enable the implementation of the required data structures and algorithms.

3.2.1 Basic Types and Operations

The language supports integers, floating-point numbers, booleans, characters, and strings, all of which are immutable. Strings support indexing, as well as a `length` property. Numerical types support addition, subtraction, multiplication, division, modulo, and negation operations. Division between two integers is integer division; for example `5 / 3` evaluates to `1`. On the other hand, division involving at least one floating-point number is floating-point division; for example, `3 / 2.0` evaluates to `1.5`. Relational operations are also supported between numerical types, along with characters and strings, which are compared lexicographically. Booleans

Algorithm Category	Algorithms
Searching Algorithms	Sequential Search Binary Search
Simple Sorting Algorithms	Bubble Sort Selection Sort Insertion Sort
Divide and Conquer	Merge Sort Quick Sort
Graph Traversal	DFS Traversal BFS Traversal
Minimum Spanning Trees	Prim's Algorithm Kruskal's Algorithm
Shortest Path	Dijkstra's Algorithm Floyd's Algorithm
Topological Sort	Topological Sort using DFS Topological Sort using BFS

Table 3.1: DSA Requirements – Algorithms

support equality comparison (`==`, `!=`), conjunction (`and`), disjunction (`or`), and negation (`not`). Bit-wise conjunction (`&`), disjunction (`|`), and negation (`~`) are supported for integers, as they are needed for computing hash codes in data structures involving hash tables.

In some programming languages, such as C/C++ and Java, character literals can be used as integers. We do not allow this; `Char` is treated as a separate non-numeric type. Similarly, strings are distinct from character arrays. `Bool` is also a non-numeric type, unlike in other languages, such as Python, where `0` and `1` can be used instead of `false` and `true`. Those design decisions are based on the principle of Enforcing Good Practice, as blurring the distinction between the aforementioned types can lead to more error-prone programs, with cases of type mismatch being permitted by the type system and not reported as type errors.

3.2.2 Arrays

The language supports mutable fixed-size arrays. An array is allocated by specifying its base type and size. For example, `new Array[Int](4)` allocates an array of four integers. Array elements are accessed and modified using zero-based indexing. For example, `arr[2]` accesses the third element of array `arr`. Valid indices range from 0 up to, but not including, the size of the array. Array access using an invalid index will result in a runtime error. Arrays also support a `length` property, making information about the size of the array directly accessible. Otherwise, users would need to store it separately, which is less straightforward and more error-prone, thus violating the Enforcing Good Practice principle.

Abstract Data Types	Implementations
Collection	
Stack	Linked Stack Array Stack Dynamic Array Stack
Queue	Linked Queue Array Queue
Double-Ended Queue	Doubly Linked Deque
List	Linked List Array List Find Algorithm
Priority Queue	Binary Heap Heap Sort Algorithm
Map	BST Map AVL Map Skip List
Set	Linked Set Hash Set using Separate Chaining Hash Set using Open Addressing
Disjoint Sets (Union/Find)	Parent Pointer Tree
Graph	Adjacency Graph

Table 3.2: DSA Requirements – Data Types

In some languages, such as C/C++, the size of an array is part of its type, which implies that arrays with the same base type but different sizes are considered values of different types, forbidding assignment between them. In fact, C/C++ forbids array variable assignment altogether. In other languages, such as Java, the size of an array is not part of its type and array variable assignment is supported for arrays with the same base type, even if their sizes are different. In DSA implementations, there are cases where array variable assignment is needed. In particular, the resize operations of the Dynamic Array Stack and Hash Set implementations involve assigning a new array of different size to the internal array variable. Therefore, based on the domain-specific principle, the size of an array is not part of its type in our language, and array variable assignment is supported. In C/C++, such assignments would require pointers to be used instead of array variables.

3.2.3 Null

The `null` reference is used to represent the absence of a value. It is utilized in several DSA implementations, for example, as the child references of a leaf node in a tree or to represent the end of a linked list. In this language, the `null` reference is a valid value for any type. The language has a dedicated `Null` type, containing

`null` as its only value, and is a subtype of all other types in the language.

In our language, `null` is also the default value for uninitialized variables and array elements. Some languages, such as C [3, §4.9] and C++ [4, §6.3.5.1], do not specify a default value for uninitialized local variables. On the other hand, Java uses type-dependent default values [19, §4.12.5]; however, local variables must be explicitly initialized before use, otherwise a compilation error is reported [19, §14.4.2]. Yet, Java may reject programs where it could be established at compile time that variables are always assigned a value before use [19, ch. 16]. Since in our language `null` is a valid value for all types, we chose it as the default value for all variables that have not been initialized by the user. This results, however, in our language permitting the use of variables without explicitly initializing them first, which is considered bad programming practice.

Equality comparison (`==`, `!=`) of an expression with the `null` reference represents checking whether the expression has a value or not. Based on the domain-specific principle, the language supports such checks for expressions of any type, even those for which regular equality comparison is not supported. These checks are used, for example, to determine the position of an element in the Hash Set implementation using open addressing.

3.2.4 Variables and Assignment

Variables are declared by specifying their name and type. The value of a variable is updated using the assignment statement. Array element assignments have the form `arr[idx] = val`, where `arr` is an array expression, `idx` is the index of the updated element and `val` is the new value. The example in Listing 1, which computes the average of a two-element array, contains declaration and assignment statements. As shown in the listing, the language supports code comments; single-line comments begin with `//`, while multi-line comments begin with `/*` and end with `*/`.

```
Array[Float] arr;          // declaration
Float avg = 0;             /* declaration
                              with initialization */
arr = new Array[Float](2);
arr[0] = 1.5;
arr[1] = 2.5;
avg = (arr[0] + arr[1]) / arr.length;
```

Listing 1: Declaration and assignment

3.2.5 Statements and Expressions

Our language distinguishes between statements and expressions. A statement performs an action that controls program execution or has side effects, such as declaring a new variable or assigning a value to an existing variable. Expressions, when evaluated, produce a value (given that the evaluation terminates). For example, `arr.length` in Listing 1 is an expression whose value is equal to the size of array `arr`. A statement may contain expressions. For example, the assignment statement

updates the value of a variable based on an expression. Function calls are the only expressions that can also be used as statements. We allow this because functions may also have side effects. For example, a function implementing a sorting algorithm updates the input array.

3.2.6 Control Flow and Blocks of Statements

Conditional statements are needed in many algorithms to direct control flow based on a boolean expression. Our language supports `if` statements, with an optional `else` branch. In the example in Listing 2, a conditional statement is used to compute the maximum of two integers.

```
Int x = -4;
Int y = 5;

Int max;
if (x > y) {
    max = x;
} else {
    max = y;
}
```

Listing 2: Using a conditional statement to compute the maximum integer

According to the principle of Enforcing Good Practice, it must be explicitly specified which statements belong to a conditional branch. Some languages, such as Python, use indentation-sensitive syntax to specify branches. We choose to use blocks of statements, enclosed in curly brackets (`{}`).

We prefer not to enforce the use of indentation, despite the fact that unindented code is considered bad programming practice. Had we decided to make indentation part of the syntax, then the language would need to restrict the use of whitespace characters, by giving special meaning to tabs or a fixed number of spaces. Depending on invisible characters to define program correctness may lead to confusion, since two code segments that look identical may have different meanings or it may even be the case that one of them is syntactically incorrect. Furthermore, programmers would have to configure their code editors to make sure that the whitespace characters produced match the language's syntactic requirements.

Other languages that use curly brackets to define blocks, such as Java and C/C++, also allow single statements to be used instead. Based on the principle of Enforcing Good Practice, we do not allow this, as it makes the branch specification implicit and the programs more error-prone. For example, programmers may need to modify a single-statement `else` branch by adding new statements but forget to enclose the branch in curly brackets, causing the added statements to always execute after the conditional.

Our language also supports loops, so that parts of programs can be executed repeatedly. Any form of repetition can be expressed using `while` loops, by continuing execution as long as a boolean condition holds. However, in many algorithms, `for`

loops are preferred for certain use cases, such as traversing arrays or a sequence of consecutive integers. Including `for` loops is therefore in accordance with the domain-specific principle. Moreover, the use of `for` loops for traversing (fixed-size) arrays makes it significantly easier to reason about algorithmic complexity, as the number of repetitions a `for` loop performs is directly bounded by the length of the array it traverses. Consequently, our language supports both condition-based `while` loops, as well as `for` loops for traversing arrays and strings, which are fixed-size and indexable. Listing 3 shows two ways to compute the sum of an integer array, using a `while` loop and a `for` loop, respectively. Similarly to conditionals, loop bodies are blocks of statements enclosed by curly brackets.

```
Int sum = 0;
Int i = 0;
while (i < intArray.length) {
    sum = sum + intArray[i];
    i = i + 1;
}
```

```
Int sum = 0;
for (Int n in intArray) {
    sum = sum + n;
}
```

Listing 3: Computing the sum of an integer array using a `while` and a `for` loop

3.2.7 Functions

The types of the function's parameters, as well as its return type, need to be specified in its definition. As mentioned in subsection 3.2.5, function calls are expressions that can also be used as statements. The return statement is used inside the function body to pass a value back to its caller. When the function's return type is `Null`, the return value can be omitted. When the execution of a function's body is finished and no return statement is executed, the return value is `null`. Listing 4 contains two examples of function definitions, namely for computing the absolute value of an integer and swapping two elements of an integer array.

```
function abs(Int x) -> Int
{
    if (x >= 0) {
        return x;
    } else {
        return -x;
    }
}
```

```
function swap(Array[Int] a, Int i, Int j) -> Null
{
    Int tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
    // optional: return; or return null;
}
```

Listing 4: Absolute value and array element swap functions

3.3 User-Defined Types

The language supports user-defined types so that data structures can be implemented as types. In particular, it supports abstract types to represent ADTs as well as concrete type definitions for their implementations.

3.3.1 Abstract Types

An ADT specifies a set of operations, without providing implementations. Similarly, an abstract type in our language is defined using only method signatures containing their parameter types and return type.

```
abstract CollectionOfFloats {
    function toArray() -> Array[Float];
    function getSize() -> Int;
    function isEmpty() -> Bool;
}

abstract SetOfFloats extends CollectionOfFloats {
    function add(Float) -> Null;
    function remove(Float) -> Null;
    function contains(Float) -> Bool;
}
```

Listing 5: Abstract inheritance

One abstract type can extend another, inheriting its method signatures and possibly adding new ones. In Listing 5, `SetOfFloats` extends `CollectionOfFloats`, which enables objects of the former to call methods of the latter. Abstract inheritance introduces a subtype relation between the abstract types. In DSA implementations, we can use abstract inheritance to express that ADTs, such as Sets and Lists, are Collections, sharing common operations related to their sizes, such as `getSize` and `isEmpty`, as well as a `toArray` operation, which enables traversing their elements using `for` loops.

3.3.2 Classes

Data structure implementations require compound data types that can hold complex information. For example, an Array Stack needs an integer to store its size and an array to store its elements. The language supports classes for this purpose, so that operations associated with the data structures are part of the type definition as class methods. All class methods and instance variables (also called fields) are public. Access modifiers are not supported, based on the Keeping It Simple principle, as the DSA implementations do not specify private members. According to the same principle, we do not support class variables; each class object has its own copy of the fields. Each class must define a special constructor method named `init`, which is invoked automatically when a new object is allocated, to initialize its fields and perform any preparatory work needed before the object is ready to be used.

In class method bodies, the `self` keyword is used to access fields and methods of the current class instance. In some languages, such as Java [19, §8.4, §15.8.3] and C++ [4, §16.2.10], the equivalent `this` keyword is not required to be explicitly specified as a class method parameter, nor is it required for accessing class members. On the other hand, languages such as Rust [20, §5.3] and Python [21, §3.2.8.2] require that `self` is explicitly specified as a method's first parameter, and its use is

```
abstract Vehicle {
    function isStreetLegal() -> Bool;
}
abstract RoadVehicle extends Vehicle {
    function wheelCount() -> Int;
}
class Car { /* fields, methods */ }

instance Car of RoadVehicle {
    // implementation of Vehicle's method
    function isStreetLegal() -> Bool {
        return self.wheelCount() >= 2;
    }
    function wheelCount() -> Int {
        return 4;
    }
}
```

Listing 6: Instance declaration

always required for member access.[1] Our approach is a combination of the two. In particular, our language does not require that `self` is explicitly declared. However, we always enforce its use, based on the principle of Enforcing Good Practice, in order to avoid cases where top-level functions and class fields are shadowed by class methods and local method variables respectively.

3.3.3 Instance Declarations

The language also makes it possible to declare that a class implements an abstract type, using an instance declaration. An instance declaration specifies the associated class and abstract type and provides implementations for all method signatures in the abstract type and its supertypes, as shown in Listing 6. Instance declarations provide the way for data structures to be declared as implementations of abstract data types in our language. For example, to express that a Linked Queue is a Queue, the `LinkedQueue` class can be declared an instance of the abstract `Queue` type. An instance declaration makes the class a subtype of the abstract type it implements.

The methods defined in instance declarations are treated as methods of the associated class. They have access to fields and other methods of the class, including those defined in other instance declarations of the same class. Separating instance declarations in dedicated syntactic constructs makes it easier to distinguish methods that are related to internal implementations of data structures from methods that are part of their ADT interface. For example, a Hash Set class, which implements the abstract type `Set`, may define methods for computing hash indices or resizing the hash table, while the methods for insertion and removal of elements, which are operations of the Set ADT, are implemented in a dedicated instance declaration.

[1] In fact, `self` in Python is just a conventional name for a method's first parameter rather than a keyword.

3.3.4 On Class Inheritance

The language does not support class inheritance, as it offers no significant benefits in implementing the required DSA, even in data structures where multiple operations may have similarities. This is the case for the `BSTMap` [18, §11.1.1] and `AVLMap` [18, §13.2] types, which implement the Map ADT using a binary search tree and an AVL tree, respectively. While class inheritance can be utilized here, not much code of the former can be used directly in the latter's implementation. In particular, code for the `get` operation can indeed be shared between the two classes; however, the insertion and removal of elements in an AVL tree involve frequent height updates between steps, forbidding potential reuse of the methods defined in the `BSTMap` class in the `AVLMap` implementation. Moreover, class inheritance introduces unnecessary complexity, as users would need to reason about program behavior not directly related to the DSA implementations, such as keeping track of which method implementation is invoked when methods are overridden. Therefore, the decision not to support class inheritance may deviate from the domain-specific principle in one DSA implementation, but it is based on the Keeping It Simple principle.

3.3.5 Generics and Type Constraints

Generic types are types parameterized by type variables. They provide a way to define reusable data structures that work for multiple data types. For example, instead of having to define a set of integers and a set of strings separately, it suffices to define a generic `Set<<T>>`, where `T` is a type variable, and then instantiate it each time with the desired element type. Methods and internal implementations may be defined with respect to those type parameters. Similarly, generic top-level function definitions are also supported.

While generic definitions provide flexibility, it is often necessary to limit which types are allowed to be used to instantiate them. Therefore, constraints on type variables need to be introduced in these cases. For example, functions implementing sorting algorithms should require that the base type of the array to be sorted is comparable. Specifying type constraints is in accordance with the principle of Enforcing Good Practice, as it prevents invalid instantiations of generic definitions.

Type constraints are expressed in the language as abstract types. For example, a type `T` can be declared comparable if it is an instance of the abstract type `Cmp<<T>>`, which specifies a method for comparing the calling object with another object of type `T`.

Class types can be declared to satisfy a constraint in the same way they are declared as instances of abstract types, i.e. with an instance declaration providing the required method implementations.

Generic abstract types do not need type constraints, because any necessary constraints can be enforced by the concrete types implementing them. In some cases, different implementations may even need to impose different constraints. For example, the implementation of `Set<<T>>` using a linked list requires that `T` supports

equality comparison, while implementations using hash tables require that `T` is hashable.

3.4 Language Syntax

A program in this language is a non-empty list of top-level definitions. A top-level definition is either an abstract type definition, a class definition, an instance declaration, or a function definition. Each top-level construct begins with an appropriate keyword (`abstract`, `class`, `instance`, `function`).

An example function definition is shown in Listing 7, namely an implementation of the factorial function.

```
function fact(Int n) -> Int {
    if (n <= 0) {
        return 1;
    } else {
        return n * fact(n - 1);
    }
}
```

Listing 7: Definition of the factorial function

Generic definitions are specified by including a (non-empty) list of type parameters enclosed in `<< >>` after the type or function name. Generic classes and functions may also have constraints on their type variables. Listing 8 contains the definition of a generic `Pair` class where both type variables must support equality comparison.

```
class Pair<<X,Y>> where X implements Eq<<X>> and Y implements Eq<<Y>> {
    X fst;
    Y snd;

    function init(X fst, Y snd) -> Null {
        self.fst = fst;
        self.snd = snd;
    }
}
```

Listing 8: Generic class with type constraints

A method definition is similar to a top-level function definition, except that it does not bind new type parameters or constraints. Instead, it uses those of the enclosing class definition.

Similarly, an instance declaration does not bind any type parameters, but instead uses the ones bound by the class definition. As discussed in subsection 3.3.3, instance declarations are associated with class definitions; the methods they define are treated as methods of the corresponding class. In Listing 9, the `Pair` class is declared as an instance of the built-in `Eq` abstract type.

```
instance Pair of Eq<< Pair<<X,Y>> >> {
    function eq(Pair<<X,Y>> other) -> Bool {
        // the constraints on X and Y allow the use of ==
        return self.fst == other.fst and self.snd == other.snd;
    }
}
```

Listing 9: Instance declaration

Note that `and` is used both as the boolean conjunction operator and as a separator in lists of constraints, as shown in the previous listings. Other reserved keywords in our language include `extends`, `while`, `for`, `in`, `true`, `false`, `null`, `new`, `not`, `or`.

Table 3.3 shows the precedence and associativity of operations. Unary operators are non-associative.

Precedence	Operator	Description	Associativity
1	`[]` `.`	indexing member access	left

2	`not`
	`-`
	`~`
		logical negation
		numerical negation
		bit-wise negation
			-

3	`*`, `/`, `%`	multiplicative operators	left
4	`+`, `-`	additive operators	left
5	`==`, `!=`, `<`, `<=`, `>`, `>=`	comparison operators	left
6	`&`	bit-wise AND	right
7	`|`	bit-wise OR	right
8	`and`	logical AND	right
9	`or`	logical OR	right

Table 3.3: Operator precedence and associativity

The syntax for accessing array elements and string characters is `e[index]`. Accordingly, array types are specified using `[]`. For example, an array of integers has type `Array[Int]`. On the other hand, generic types are specified using `<< >>`. For example, a pair of integers according to the definition of the `Pair` class in Listing 8 has the type `Pair<<Int,Int>>` (note that `Int` satisfies the constraint `Eq<<Int>>`). This syntactic distinction exists to reflect that arrays are fundamental constructs with special operations, such as indexing, rather than ordinary generic classes.

The syntax of our language is similar to that of Java, although there exist several differences. Most notably, our language allows top-level function definitions, while Java requires all functions to be defined as methods of a type. Abstract types in our language closely resemble Java interfaces, with the exception that abstract types can only contain method signatures; neither fields nor default method implementations are allowed. Moreover, Java classes are declared to implement an interface in the class declaration construct, providing all method implementations in the same block.

Our language, on the other hand, uses a separate block for method implementations related to each instance declaration.

3.5 Built-In Definitions

In this section, we describe the built-in definitions in our language, including the functions `floor`, `random`, `ord`, `chr`, and `range`, as well as the abstract types `Eq`, `Cmp`, and `Hashable`.

The `floor` function, with signature `floor(Float) -> Int`, converts decimals to integers.

The `random` function takes no arguments and generates a random floating-point number in the interval $[0, 1)$.

The functions `ord` and `chr` are used to convert characters to and from integers, respectively. Their signatures are `ord(Char) -> Int` and `chr(Int) -> Char`.

The `range` function is used to generate sequences of consecutive integers. It has signature `range(Int,Int) -> Array[Int]`. For integers `a` and `b`, `range(a,b)` returns the sequence `a`, `a+1`,...,`b-1` if `b` is greater than `a`; otherwise, it returns the sequence `a`, `a-1`,...,`b+1`.

The built-in abstracts `Eq`, `Cmp`, and `Hashable` are used to specify comparable and hashable types. Conceptually, they correspond to the definitions given in Listing 10.

```
abstract Eq<<T>> {
    function eq(T) -> Bool;
}
abstract Cmp<<T>> extends Eq<<T>> {
    function lt(T) -> Bool;
}
abstract Hashable<<T>> {
    function hashCode() -> Int;
}
```

Listing 10: Built-in abstract types

The equality comparison operators are overloaded for types `T` that are subtypes of `Eq<<T>>` (and similarly for relational operators and `Cmp<<T>>`). Basic types `Int`, `Float`, `Bool`, `Char`, and `String` are subtypes of `Cmp<<Int>>`, `Cmp<<Float>>`, `Eq<<Bool>>`, `Cmp<<Char>>`, and `Cmp<<String>>`, respectively, reflecting the supported comparison operations described in subsection 3.2.1. The types `Int`, `Float`, `Bool`, `Char`, and `String` are also hashable.

3.6 Type System

The language is statically typed; type errors are detected and reported at compile time [22, ch. 1]. Static typing ensures that the programs that get to execute do not contain certain kinds of errors [22, §1.2]. In particular, programs containing type

errors will be rejected by the type checker [23]. Empirical evidence also shows that static typing improves maintainability [24, 25].

Furthermore, the language is explicitly typed; types of variables and function signatures need to be specified in declarations. Therefore, type annotations are part of the syntax, allowing them to potentially serve as a form of documentation [22, §1.2]. In the context of implementing DSA, function signatures may be indicative of their behavior. A study shows that type names alone improve the usability of APIs [26].

In this section, we describe the type system of the language. The notation is largely based on the one used in the Featherweight Java paper [27]. A more detailed presentation can be found in Appendix B.

3.6.1 Abstract Syntax

Listing 11 contains the abstract syntax of our language. T, R, and S range over types, while X and Y range over type variables. We assume that the set of variables contains self, which cannot be used as a regular variable name.

We write $\overline{\mathrm{T}}$ for the possibly empty sequence T_1,...,T_n (and similarly for $\overline{\mathrm{X}}$, $\overline{\mathrm{O}}$, $\overline{\mathrm{e}}$). We express the concatenation of sequences using commas, e.g. the type sequence $\overline{\mathrm{O}}$,$\overline{\mathrm{T}}$,T in rule T-Function (treating T as a sequence of length 1). We write $\overline{\mathrm{M}}$ for M_1...M_n without commas (and similarly for $\overline{\mathrm{s}}$). We also write "$\overline{\mathrm{T}}$ $\overline{\mathrm{fd}}$;" for "T_1 f_1;...T_n f_n;" and similarly "$\overline{\mathrm{T}}$ $\overline{\mathrm{x}}$;" for "T_1 x_1;...T_n x_n;".

We assume that sequences of field, variable, class, abstract type, function, and class method declarations do not contain duplicate names. This is not the case for type sequences, which may contain duplicates. However, we assume that generic definitions do not bind the same type variable name more than once.

We write $\mathrm{P} \lhd \mathrm{O}$ to express that abstract type P extends abstract type O. We write $\mathrm{N} \lhd \overline{\mathrm{O}}$ to express that N implements abstract types O_1,...,O_n. We write $\overline{\mathrm{X}} \lhd \overline{\mathrm{O}}$ for $\mathrm{X}_1 \lhd \overline{\mathrm{O}}$,..., $\mathrm{X}_\mathrm{n} \lhd \overline{\mathrm{O}}$. Type substitution is denoted with $[\overline{\mathrm{T}}/\overline{\mathrm{X}}]\mathrm{S}$. Applying the substitution $[\overline{\mathrm{T}}/\overline{\mathrm{X}}]$ to a type S replaces each occurrence of X_i in S by T_i.

If the sequence of type arguments is empty, we omit $\langle\langle\rangle\rangle$ when specifying functions or user-defined types. In the abstract syntax, method signatures are represented using method definitions with empty bodies, while the argument names are omitted. If a variable is initialized to null at declaration, then the initialization can be omitted (we can write "T x;" instead of "T x = null;"). When the else branch of a conditional statement contains no statements, the "else {}" part can be omitted. Return statements inside functions with Null return type may omit the returned expression; in that case, "return;" is treated as "return null;".

3.6.2 Contexts and Typing Judgments

Θ is the type variable context, mapping type variables to a set of abstract types (their constraints). We write $\Theta = \overline{\mathrm{X}} <:$ for the type variable context with domain $\overline{\mathrm{X}}$ and $\Theta(\mathrm{X}) = \emptyset$, for all $\mathrm{X} \in \mathit{dom}(\Theta)$.

T	::=		*types:*
		Int | Float | Bool | Char | String	*basic types*
	|	Array[T]	*array type*
	|	Null	*null type*
	|	X	*type variable*
	|	N | O	*named types*
N	::=	$C\langle\langle\overline{T}\rangle\rangle$	*class type*
O	::=	$A\langle\langle\overline{T}\rangle\rangle$	*abstract type*
K	::=	abstract $A\langle\langle\overline{X}\rangle\rangle$ $\{\overline{M}\}$	*abstract definition*
	|	abstract $A\langle\langle\overline{X}\rangle\rangle \lhd A\langle\langle\overline{X}\rangle\rangle$ $\{\overline{M}\}$	*inherit. abstract definition*
	|	class $C\langle\langle\overline{X} \lhd \overline{O}\rangle\rangle \lhd \overline{O}$ $\{\overline{T}\ \overline{fd}; \overline{M}\}$	*class definition*
M	::=	T m($\overline{T}$ $\overline{x}$) $\{\overline{s}\}$	*method definition*
F	::=	T $f\langle\langle\overline{X} \lhd \overline{O}\rangle\rangle(\overline{T}\ \overline{x})$ $\{\overline{s}\}$	*function definition*
s	::=		*statements:*
		$\{\ \overline{s}\ \}$	*block of statements*
	|	T x = e;	*variable declaration*
	|	x = e;	*variable assignment*
	|	e[e] = e;	*array element assignment*
	|	e.fd = e;	*field assignment*
	|	return e;	*return statement*
	|	if (e) $\{\overline{s}\}$ else $\{\overline{s}\}$	*if statement*
	|	for (T x in e) $\{\overline{s}\}$	*for loop*
	|	while (e) $\{\overline{s}\}$	*while loop*
	|	e;	*expression*
e	::=		*expressions:*
		x	*variables*
	|	i	*integer literals (e.g. 3, 42)*
	|	d	*float literals (e.g. 3.14, 6.7)*
	|	true | false	*boolean literals*
	|	c	*character literals (e.g. 'c', '=')*
	|	str	*string literals (e.g. "ab", "aa bb")*
	|	null	*null literal*
	|	$f\langle\langle\overline{T}\rangle\rangle(\overline{e})$	*function call*
	|	new T($\overline{e}$)	*allocation*
	|	e[e]	*indexing*
	|	e.fd	*field access*
	|	e.m($\overline{e}$)	*method call*
	|	$\circ$ e	*unary expression*
	|	e $\bowtie$ e	*binary expression*
$\circ$	::=	not| $-$ | $\sim$	*unary operators*
$\bowtie$	::=	$+$| $-$ | $*$ | / | % | & | ! | and | or	*binary operators*
	|	== | != | $<$ | $<=$ | $>$ | $>=$	

Listing 11: Abstract syntax

Γ is the variable context. It is structured as a stack of blocks Δ, each mapping variable names to types. A block is written as a sequence of mappings x : T separated by commas. $\Gamma.\Delta$ denotes the extension of context Γ with block Δ. The empty block is denoted with (). Γ(x) denotes the type of the top-most occurrence of variable x in context Γ.

The typing judgment for statements has the form $\Theta; \Gamma \vdash^{\mathrm{T}} \mathrm{s} \Rightarrow \Gamma'$. It means that under type variable context Θ, variable context Γ, and inside the body of a function with return type T, statement s is well-typed and updates context Γ to Γ'.

For expression typing judgments, we write $\Theta; \Gamma \vdash \mathrm{e} : \mathrm{T}$, which means that expression e has type T under contexts Θ and Γ.

We write $\Theta; \Gamma \vdash \overline{\mathrm{e}} : \overline{\mathrm{T}}$ for the collection of typing judgments $\Theta; \Gamma \vdash \mathrm{e}_1 : \mathrm{T}_1$,...,$\Theta; \Gamma \vdash \mathrm{e}_\mathrm{n} : \mathrm{T}_\mathrm{n}$.

3.6.3 Subtyping and Well-Formed Types

We write $\Theta \vdash \mathrm{S} <: \mathrm{T}$ to express that S is a subtype of T under type variable context Θ. We assume that there are no cycles in the subtype relation. The subtyping relation is reflexive and transitive. Listing 12 shows the subtyping rules for type variables, abstract type and class definitions. The full subtyping relation can be found in Appendix B.

A type variable X is a subtype of all abstract types specified as constraints on X in the given type variable context. Abstract types are subtypes of the abstract types they extend, while class types are subtypes of the abstract types they implement, as shown in Listing 12.

Subtyping

$$\frac{\mathrm{O} \in \Theta(\mathrm{X})}{\Theta \vdash \mathrm{X} <: \mathrm{O}} \quad \text{(S-TVar)} \qquad \frac{\text{abstract } \mathrm{A}\langle\langle \overline{\mathrm{X}} \rangle\rangle \lhd \mathrm{O}\ \{...\}}{\Theta \vdash \mathrm{A}\langle\langle \overline{\mathrm{T}} \rangle\rangle <: [\overline{\mathrm{T}}/\overline{\mathrm{X}}]\mathrm{O}} \quad \text{(S-Abs)}$$

$$\frac{\text{class } \mathrm{C}\langle\langle \overline{\mathrm{X}} \lhd \overline{\mathrm{P}} \rangle\rangle \lhd \overline{\mathrm{O}}\ \{...\} \qquad \mathrm{O} \in \overline{\mathrm{O}}}{\Theta \vdash \mathrm{C}\langle\langle \overline{\mathrm{T}} \rangle\rangle <: [\overline{\mathrm{T}}/\overline{\mathrm{X}}]\mathrm{O}} \quad \text{(S-Class)}$$

Listing 12: Subtyping

We write $\Theta \vdash \mathrm{T}$ ok to express that type T is well-formed under type parameter context Θ. In particular, class types are well-formed if their type arguments are well-formed and also satisfy the constraints specified in the class definition, as shown in Listing 13.

We write $\Theta \vdash \overline{\mathrm{T}}$ ok for the collection of typing judgments $\Theta \vdash \mathrm{T}_1$ ok,...,$\Theta \vdash \mathrm{T}_\mathrm{n}$ ok (similarly for $\Theta \vdash \overline{\mathrm{S}} <: \overline{\mathrm{T}}$).

Well-formed class types

$$\frac{\text{class } \mathrm{C}\langle\langle \overline{\mathrm{X}} \lhd \overline{\mathrm{P}} \rangle\rangle \lhd \overline{\mathrm{O}}\ \{...\} \qquad \Theta \vdash \overline{\mathrm{T}} \text{ ok} \qquad \Theta \vdash \overline{\mathrm{T}} <<: [\overline{\mathrm{T}}/\overline{\mathrm{X}}]\overline{\mathrm{P}}}{\Theta \vdash \mathrm{C}\langle\langle \overline{\mathrm{T}} \rangle\rangle \text{ ok}} \quad \text{(WF-Class)}$$

Listing 13: Well-formed class types

We write $\Theta \vdash \overline{S} <: \overline{T}$ for $\Theta \vdash S_1 <: \overline{T}_1, ..., \Theta \vdash S_n <: \overline{T}_n$, where $\Theta \vdash S <: \overline{T}$ stands for $\Theta \vdash S <: T_1, ..., \Theta \vdash S <: T_n$.

3.6.4 Function and Method Typing

Listing 14 contains typing rules for function and class method definitions. The typing judgments have the forms "F ok" and "M ok in $C\langle\langle \overline{X} \lhd \overline{P} \rangle\rangle$", respectively.

A function definition is ok if the types of its arguments, its return type, as well as the abstract types specified in its constraints are well-formed under the function's type variable context. Additionally, the function body must be well-typed under that type variable context and the variable context mapping the arguments to their types.

Similarly, a class method is ok if the argument and return types are well-formed under the class's type variable context. In addition, the method body must be well-typed under that type variable context and a variable context that maps the arguments to their types and self to the class type.

Function and method typing

$$\frac{\Theta = \overline{X} <: \overline{O} \qquad \Theta \vdash \overline{O}, \overline{T}, T \text{ ok} \qquad \Theta; (\overline{x} : \overline{T}) \vdash^{T} \overline{s} \Rightarrow \Gamma}{T\ f\langle\langle \overline{X} \lhd \overline{O} \rangle\rangle (\overline{T}\ \overline{x})\ \{\overline{s}\}\ \text{ok}} \quad \text{(T-Function)}$$

$$\frac{\begin{array}{c}\Theta = \overline{X} <: \overline{P} \qquad \Theta \vdash \overline{T}, T \text{ ok} \\ \Theta; (\overline{x} : \overline{T},\ \text{self} : C\langle\langle \overline{X} \rangle\rangle) \vdash^{T} \overline{s} \Rightarrow \Gamma \qquad \text{class } C\langle\langle \overline{X} \lhd \overline{P} \rangle\rangle \lhd \overline{O}\ \{...\}\end{array}}{T\ m(\overline{T}\ \overline{x})\ \{\overline{s}\}\ \text{ok in } C\langle\langle \overline{X} \lhd \overline{P} \rangle\rangle} \quad \text{(T-CMethod)}$$

Listing 14: Function and class method typing

3.6.5 Abstract and Class Typing

As shown in Listing 15, an abstract type definition is ok if its methods are ok and its supertype (if there is one) is well-formed under the abstract type's type variable context.

A class definition is ok if the types of its fields, its supertypes, as well as the abstract types specified in its constraints are well-formed under the class's type variable context. Additionally, the class methods must be ok, including an init method with Null return type, and the class must be a valid instance of the abstract types it implements.

A class is a valid instance of an abstract type if for all methods specified in the definition of the abstract type or inherited from its supertypes, there exists a method definition with the same name and type signature in the class definition.

Abstract and class typing

$$\frac{\Theta = \overline{X} <: \qquad \Theta \vdash O \text{ ok} \qquad \overline{M} \text{ ok in } A\langle\langle\overline{X}\rangle\rangle}{\text{abstract } A\langle\langle\overline{X}\rangle\rangle \lhd O\ \{\overline{M}\}\ \text{ok}} \quad \text{(T-AbsExt)}$$

$$\frac{\begin{array}{c}\Theta = \overline{X} <: \overline{P} \qquad \Theta \vdash \overline{O},\overline{P},\overline{T} \text{ ok} \qquad \overline{M} \text{ ok in } C\langle\langle\overline{X} \lhd \overline{P}\rangle\rangle \\ \text{Null init}(\overline{S}\ \overline{x})\ \{\overline{s}\} \in \overline{M} \qquad \Theta \vdash \text{instance } C\langle\langle\overline{X}\rangle\rangle \text{ of } \overline{O} \text{ ok}\end{array}}{\text{class } C\langle\langle\overline{X} \lhd \overline{P}\rangle\rangle \lhd \overline{O}\ \{\overline{T}\ \overline{fd};\ \overline{M}\}\ \text{ok}} \quad \text{(T-Class)}$$

Listing 15: Abstract and class typing

3.6.6 Statement and Expression Typing

In this section, we present the typing rules for selected statements and expressions (Listing 16). The full set of statement and expression typing rules can be found in Appendix B.

In rule T-For, *base* is an auxiliary function defined over array types and the String type. It maps each array type to its base type and String to Char. A for loop statement is well-typed under contexts Θ, Γ if the type of the loop variable matches the base type (according to the *base* function) of the expression that is being traversed. The loop body must also be well-typed under Θ and Γ extended by a block mapping the loop variable to its type.

For statement typing

$$\frac{\Theta \vdash S \text{ ok} \qquad \Theta;\Gamma \vdash e : S \qquad \mathit{base}(S) = T \qquad \Theta;\Gamma.(x{:}T) \vdash^{R} \overline{s} \Rightarrow \Gamma'}{\Theta;\Gamma \vdash^{R} \text{for } (T\ x \text{ in } e)\ \{\overline{s}\} \Rightarrow \Gamma} \quad \text{(T-For)}$$

Function call and class object allocation

$$\frac{\mathit{ftype}(f) = \langle\langle\overline{X} \lhd \overline{O}\rangle\rangle\overline{R} \to R \qquad \Theta \vdash \overline{T} \text{ ok} \qquad \Theta \vdash \overline{T} <: [\overline{T}/\overline{X}]\overline{O} \qquad \Theta;\Gamma \vdash \overline{e} : [\overline{T}/\overline{X}]\overline{R}}{\Theta;\Gamma \vdash f\langle\langle\overline{T}\rangle\rangle(\overline{e}) : [\overline{T}/\overline{X}]R} \quad \text{(T-FCallExp)}$$

$$\frac{\Theta \vdash N \text{ ok} \qquad \mathit{mtype}_{\Theta}(\text{init},N) = \overline{T} \to \text{Null} \qquad \Theta;\Gamma \vdash \overline{e} : \overline{T}}{\Theta;\Gamma \vdash \text{new } N(\overline{e}) : N} \quad \text{(T-NewClass)}$$

Equality comparison typing

$$\frac{\Theta;\Gamma \vdash e_1, e_2 : T \qquad \Theta \vdash T <: \text{Eq}\langle\langle T\rangle\rangle \qquad \odot \in \{==, !=\}}{\Theta;\Gamma \vdash e_1 \odot e_2 : \text{Bool}} \quad \text{(T-Eq)}$$

$$\frac{\Theta;\Gamma \vdash e_1 : S \qquad \Theta;\Gamma \vdash e_2 : T \qquad \text{Null} \in \{S,T\} \qquad \odot \in \{==, !=\}}{\Theta;\Gamma \vdash e_1 \odot e_2 : \text{Bool}} \quad \text{(T-NullEq)}$$

Listing 16: Statement and expression typing

Constraint satisfaction for generic functions and class types is checked at function calls and class object allocation expressions, respectively.

In rule T-FCallExp, *ftype* maps each function name to its signature, including its type constraints. A function call is well-typed if its type arguments are well-typed and satisfy the function's constraints. Additionally, the argument expressions must be well-typed and their types must match the instantiated function signature.

Similarly, *mtype* maps method names for given types to their type signatures. In rule T-NewClass, *mtype* is used to look up the signature of the class type's init method. A class object allocation is well-typed if the class type is well-formed and the argument expressions are well-typed, with their types matching the signature of the init method.

Equality comparison between two well-typed expressions is well-typed if at least one of the expressions has type Null or if both expressions have the same type T, which is a subtype of Eq⟪T⟫.

4

Implementation

In this chapter, we describe the implementation of our language[1], namely the implementation of the front end of the transpiler. The development of a code generator is left for future work. The intended design is that the transpiler takes source files in our language and produces modules in the target language that can be used within an external program. This will enable users to implement DSA as programs in our language, compile them to the target language, and then use the external program to import and invoke the compiled functions, as well as to handle input and output.

We chose Haskell as the implementation language for the transpiler. Multiple compilation phases involve the traversal and transformation of abstract syntax trees. For example, semantic analysis in our language involves traversing the AST and adding type annotations, resulting in a new AST. Haskell's algebraic data types offer straightforward AST representations, while Haskell's pattern matching facilitates AST traversal and transformations, which are implemented as recursive functions. Moreover, prior experience with Haskell in compiler construction projects was also a factor for choosing it as the implementation language for the transpiler.

4.1 Lexing and Parsing

We have written the grammar of our language in Labelled Backus-Naur (LBNF [14]) form, which can be found in Appendix A. We give this grammar to the BNF Converter (BNFC [13]) to produce the specification files for the lexer and parser of the language. Since Haskell is the implementation language, we instruct BNFC to generate the files in the format used by the Alex [28] lexer generator and the Happy [29] parser generator. BNFC also produces the AST representation of programs in the source language as a Haskell data type, which we utilize in our transpiler implementation. In particular, the parser produces its output in that format.

4.2 Semantic Analysis

The type checker uses the AST produced by the parser to build the global context and perform type checking, reporting any semantic errors it detects at any point of its execution.

[1]The implementation is available at: `https://github.com/grnkl/dsalang`

4.2.1 Contexts

In this section, we describe how the contexts are represented in our implementation. Notably, the implementation utilizes a global context Σ to store information about user-defined types and functions. In the type system rules, this information is represented using auxiliary functions, some of which are mentioned in subsection 3.6.6, and all are described in detail in Appendix B.

In particular, the global context Σ contains information about:

- *Classes.* For each class, Σ contains its name, its type parameter context, a mapping of field names to their types, and a mapping of method names to their signatures.
- *Abstracts.* For each abstract type, Σ contains its name and type parameters, and a mapping of method names to their signatures.
- *Functions.* For each function, Σ contains its name, its type parameter context, and its signature.
- *Subtype relations.* In particular, Σ contains a mapping of abstract type names to the abstract types they extend, as well as a mapping from class names to the list of abstract types they implement.

The global context contains the signatures of built-in functions `randInt`, `floor`, `ord`, `chr`, and `range`. Information about the built-in `Eq`, `Cmp`, and `Hashable` abstract types is also part of the global context.

The type parameter context Θ is a mapping of each type parameter to a list of abstract types, corresponding to its constraints. Θ contexts are local to class, abstract, and function definitions. Instance declarations use the Θ context of the relevant class definition.

The local variable context Γ is structured as a stack of blocks Δ, which map variable names to types.

4.2.2 Building the Global Context

The Σ context is built as follows. First, the type checker extracts all relevant information from the top-level definitions and checks for name conflicts between them. Then it ensures that abstract inheritance and instance declarations involve valid types. It is also checked that there are no abstract inheritance cycles before the mapping from abstracts to their supertypes is built. The type checker also checks for duplicate class fields and duplicate methods in classes, abstracts and instance declarations.

The Θ context for each class, abstract, and function definition is constructed by first ensuring that there are no duplicate type parameters in each definition, and then by checking that type constraints are valid as abstract types and are applied to bound parameters.

The type checker then proceeds to check the validity of types in class fields and method signatures. For each abstract, it adds the inherited method signatures to its method mapping, checking for name conflicts. Finally, it checks that methods in instance declarations match the respective abstract method signatures and adds them to the respective class method mappings, making sure that there are still no name conflicts.

4.2.3 Type Checking

Type checking involves the following operations: checking that a statement is well-typed, confirming that an expression has the expected type, and inferring the type of expressions. All computations are based on the typing rules presented in Appendix B. The last two operations are mutually recursive. The type checker traverses the AST performing these operations and recursively builds the type-annotated AST.

4.2.4 Error Messages

The type checker may detect a type error at some point during the traversal of the AST. It utilizes the information available at that point to construct an error message and returns it to the checker of the parent AST node. The error message is propagated up to the root of the AST, and additional information is gradually added to it, providing context regarding the location of the error in the program.

The type checker detects several kinds of errors. Errors such as duplicate definitions or cyclic abstract inheritance are detected while building the global context, as described in subsection 4.2.2. The type checker reports an error if a class does not define an `init` method with `Null` return type.

When checking functions and class methods, an error is reported if there are duplicate argument names or if a local variable is declared more than once in the same

```
abstract Abs<<T>> {}

class C1<<A>> {
    function init() -> Null {}
}

class C2<<T>> where T implements Abs<<C1<< Abs<<T,T>> >>>> {
    function init() -> Null {}
}

TYPE ERROR
While checking the validity of constraints in class C2:
While checking the validity of type arguments of abstract Abs<<C1<<Abs<<T, T>>>>
>>:
While checking the validity of type arguments of class C1<<Abs<<T, T>>>>:
Wrong number of type arguments for abstract Abs<<T, T>>
Expected: 1
Got:      2
```

Listing 17: Arity mismatch in abstract type

```
class C<<T>> where T implements Cmp<<T>> {
    function init() -> Null {}
}

class D {
    function init() -> Null {}
}

function main() -> Int {
    C<<D>> x;
    return 1;
}
```

```
TYPE ERROR
In (top-level) function main:
In statement 'C<<D>> x;'
While checking validity of type C<<D>>:
While checking if the type arguments of class C<<D>> satisfy C's constraints:
Class D implements no abstracts so it cannot be a subtype of Cmp<<D>>, required
by C
```

Listing 18: Constraint not satisfied by type argument

scope. References to out-of-scope variables, including the use of `self` outside of class methods, as well as invalid field access and calls to unknown functions or methods, are also detected. Typing errors occur when invalid types are used in variable declarations and allocation expressions, or when there is an arity or type mismatch in function and method calls or in the application of operators.

Type error messages produced by the transpiler contain information regarding the location of the error as well as a description of the nature of the error. As shown in Listing 17, they also disclose intermediate steps, corresponding to the series of checks that led to the detection of the error, and not just its location in the source code. In this example, simply reporting the location of the error and the arity mismatch may not be particularly helpful for the user in identifying which part of the complicated type constraint is erroneous.

```
function main() -> Int {
    if (true) {
        if (false) {
            Int x;
            return 1;
        }
        else {
            while (false) {
                return x;
            }
            return 2;
        }
    }
}
```

```
TYPE ERROR
In (top-level) function main:
In statement 'if (true) {...}'
In statement 'if (false) {...} else {...}'
In statement 'while (false) {...}'
In statement 'return x;'
Variable x not in scope
```

Listing 19: Error in nested statement

Error messages use natural language descriptions, rather than cryptic error message codes or unintuitive technical terminology. Listing 18 shows an error message that explains why the type argument of a generic class instantiation does not satisfy the required constraint.

When an error occurs inside a statement, the error message includes that statement, as shown in Listing 18. In the case of nested statements, the message also includes the headers of the enclosing statements, in order to help users identify the location of the error. The program in Listing 19 contains a return statement accessing an out-of-scope variable inside a `while` loop inside two nested conditional statements. The transpiler outputs the conditional and `while` statement headers before the return statement, where the error occurs.

Appendix C presents the complete set of templates used for the error messages reported by the type checker.

5

Analysis

In this chapter, we analyze our language with respect to the design principles we adopted, as described in Section 2.2. Regarding the domain-specific principle, we analyze the structural differences between the DSA pseudocode in the course book [18] and DSA implementations in our language. Moreover, we go through the educational principles described in subsection 2.2.1, identifying the parts of the language design that are based on each principle as well as parts that violate it. We also test our transpiler implementation using a testsuite and analyze the type error messages produced for the negative typecheck tests in the testsuite, discussing the information they provide and their limitations.

5.1 Analysis of the Language Design

In this section, we revisit the principles we used to guide our design decisions, as described in subsections 2.2.1 and 2.2.3, and identify which design decisions follow them and which violate them.

5.1.1 Principle: Straightforward DSA Implementations

We have implemented the 34 data structures and algorithms listed in Tables 3.1 and 3.2 as programs in our language.[1] In this section, we structurally compare the pseudocode for the required DSA with implementations in our language. In particular, we identify constructs used in the pseudocode and discuss what our language uses in the corresponding DSA implementations.

5.1.1.1 ADTs

The pseudocode uses the `interface` construct to express abstract data types. This construct may declare fields, such as `size` in the `Collection` and `Graph` interfaces, method signatures, or even method implementations, such as the `isEmpty` method in `Collection`.

Our language uses the `abstract` construct to express abstract data types, which contains only method signatures. In DSA implementations, instead of the `size` property for Collections, a `getSize` method is used. Definitions for the `getSize`

[1] Available at: `https://github.com/grnkl/dsalang/tree/main/DSA%20Implementations`

and `isEmpty` methods are given in concrete data structure implementations of Collections.

Both the `interface` and the `abstract` constructs support inheritance, which is used to express that ADTs, such as Queue, Stack, Set, and Map, are Collections.

5.1.1.2 Compound Data Types

The pseudocode uses a `datatype` construct to define compound data types, including internal variables and functions. Our language uses classes, which define fields and methods. In the pseudocode, internal variables may be initialized in the datatype declaration, while in our language, class fields may be initialized in the `init` method that must be defined in every class. In the pseudocode, only 3 out of 17 data structure implementations (as specified in Table 3.2) contain a `datatype` that specifies a constructor method for handling internal variable initializations.

In 16 out of 17 data structure implementations, it is declared that the data structures implement an ADT, providing implementations for the operations of the ADT. In the pseudocode, this declaration happens in the `datatype` construct, for example `datatype ArrayStack implements Stack`. All methods of the `datatype` are defined in the same construct, including those which implement the ADT interface. In our language, a separate construct is used, namely instance declarations. When a class `C` implements an abstract type `A`, all class methods that implement the abstract type's method signatures are defined in a separate `instance C of A` block.

The pseudocode uses inheritance for `datatype` in one occasion, namely the AVL Map implementation, which reuses parts of the BST Map implementation. In our language, the AVL Map is implemented using separate class definitions.

5.1.1.3 Conditional Branching

The pseudocode uses `if-else` statements which allow any number of `else if` branches at the same level. Our language supports conditional statements with at most two branches, namely `if` and an optional `else` branch. Multi-branch conditionals are expressed by nesting `if-else` statements inside the `else` branch of the enclosing conditional, as shown in the example in Listing 20, taken from the Binary Search algorithm implementations. This difference in structure occurs in 4 out of 34 DSA implementations.

5.1.1.4 Loops

Both the pseudocode and our language use condition-based `while` loops. In Quick Sort's partitioning algorithm, the pseudocode uses a `break` statement to exit the loop (whose condition is simply `true`) when the left and right pointers have passed each other. In our language, which does not support `break` statements, the `while` loop in the corresponding implementation uses a boolean variable as the condition instead, initialized to `true`. Once the condition to exit the loop is satisfied, the variable is assigned the value `false`. Moreover, the subsequent code in the loop

```
if arr[mid] < val:
    low = mid + 1
else if arr[mid] > val:
    high = mid - 1
else:
    return mid
```

```
if (arr[mid] < val) {
    low = mid + 1;
} else {
    if (arr[mid] > val) {
        high = mid - 1;
    } else {
        return mid;
    }
}
```

Listing 20: Multi-branch conditionals (pseudocode vs. our language)

body is structured as the `else` branch of the preceding conditional statement, so it is not executed after the exit condition has been reached.

The pseudocode uses `for` loop constructs for traversing arrays, integer sequences, and collections. In our language, `for` loops are used for array traversal. The `range` function is used to specify sequences of consecutive integers as arrays. In 13 out of 34 DSA implementations, traversal of integer sequences is expressed in our language using for loops along with the `range` function. Collection traversal is expressed using a `toArray` method in the `Collection` abstract type definition, which returns the elements of the collection as an array, so it can be traversed using our language's `for` loop construct. The use of the `toArray` method in `for` loops occurs in 9 out of 34 DSA implementations in our language (7 of which are graph algorithm implementations).

In 3 out of 34 DSA implementations, the pseudocode uses a `repeat` loop construct to execute a block of statements a fixed number of times. In these cases, our language uses a for loop with the `range` function.

5.1.1.5 Variable Scope

In the `remove` operation of the Linked List implementation, the pseudocode accesses a variable that is introduced inside the preceding conditional statement. In our language, the variable needs to be declared outside the conditional statement.

5.1.1.6 Assignment

In one DSA implementation, namely Double-Ended Queue, the pseudocode contains chained assignment of the form `x = y = e`. Our language uses separate assignments (`y = e; x = y;`) to express this.

In the implementation of the Binary Heap, the pseudocode uses multiple assignment to implement swapping (`heap[i], heap[j] = heap[j], heap[i]`). Our language uses a temporary variable to swap the array elements.

5.1.1.7 Special Values

In 15 out of 34 DSA implementations, both the pseudocode and our language use `null` to express the absence of a value. In the Set implementation using Open Addressing, the pseudocode uses an additional special value `DELETED`, to indicate that hash table slots are no longer occupied. The corresponding implementation in our language keeps an additional boolean array `deleted` as a class field, to record which hash table slots are deleted slots. Searches and updates to the hash table, such as resizing or adding and removing elements, are accompanied by corresponding actions on this auxiliary boolean array.

In the same implementation of Open Addressing Hash Set, the pseudocode uses a `throw error` statement to stop program execution when the `hashAndProbe` function fails to find an unoccupied slot for the given element. Our language handles this case by returning `null`, which leads to a runtime error, since all calls to `hashAndProbe` are followed by a hash table look-up using the returned value as the array index.

5.1.1.8 Implicit Assumptions on Types

In 32 out of 34 DSA implementations, the types of the elements stored in arrays and data structures are not specified. Our language relies on generics, and the pseudocode also uses type variables for `interface` and `datatype` definitions. However, 25 out of 34 DSA implementations require that the element types have special properties, such as supporting comparison or being hashable. The pseudocode does not make these assumptions explicit, while our language uses constraints on type variables in function and class definitions.

The Parent Pointer Tree data structure that stores `n` elements assumes that those elements are the integers `0,...,n-1`. Kruskal's algorithm [18, §13.7] uses a Parent Pointer Tree, therefore the same assumption holds for the vertices of the given graph. Furthermore, the minimum spanning tree in Kruskal's algorithm is represented as a Set of edges. The pseudocode does not specify which Set implementation is used, although all provided implementations require that the edge type supports equality comparison, by comparing individual vertices. However, Kruskal's algorithm also sorts the edges by weight, which requires that the edge type is also comparable by weight. Our language uses two wrapper classes, each implementing a different comparison criterion.

In Floyd's algorithm [18, §13.10] implementation, the pseudocode stores pairs of vertices `(v,w)` as keys in a Map data structure, assuming that both the pair type and the type of the vertices are comparable. Our language uses a `Pair` class, which is declared as comparable.

5.1.2 Principle: Don't Invent

The language includes common constructs, such as variable declarations, assignment statements, loops and conditionals, functions and return statements. The type system also relies on established mechanisms, such as static typing, explicit type annotations, parametric polymorphism using generics, and bounds on type

parameters expressed using type constraints. The language adopts common object-oriented concepts, such as classes containing fields and methods, as well as interfaces, by providing the `abstract` construct.

However, the use of instance declarations as a separate construct to specify that a class implements an abstract type is not a commonly used concept in existing programming languages. While its structure is similar to Haskell's type classes, the concept of a class type implementing an abstract interface is more commonly expressed as part of the class definition. For example, Java specifies interface implementation as `class C implements Interface`.

Another uncommon feature of our language is that `null` is a valid value for any type. Some languages, such as Java, make `null` a valid value for all reference types, excluding primitives. Dynamically typed languages may allow a special value, such as Python's `None`, to be assigned to any variable, but this is because variables can store values of different types throughout the program, rather than because the special value is part of every type. While this design decision contradicts the Don't Invent principle, it is based on the domain-specific principle, as several DSA implementations require a way to represent the absence of a value in variables and arrays of any type.

5.1.3 Principle: Programming for All Versus Programming for Some

The language was designed to be used in a Data Structures course and not as a language that prepares students for a career in software development. Its constructs were included with the purpose of enabling straightforward DSA implementations, for example, fixed-size arrays, classes for data structure implementations, abstract types for expressing ADTs, and generics with type constraints for enabling data structures that are independent of the stored element type, while still specifying its required properties, such as comparability.

Furthermore, the language does not include features that are unnecessary for implementing the required DSA, but are useful in software development. Examples include modules or packages, access modifiers for class members, class variables, exception handling, and the possibility to use external libraries.

5.1.4 Principle: Keeping It Simple

The language omits features that are not needed for the required DSA implementations or that introduce unnecessary complexity. For example, it provides a single conditional construct, namely `if-else`, rather than supporting additional constructs, such as `switch` statements. Class definitions do not support access modifiers for fields and methods, or class variables. The language also does not support class inheritance, as mentioned in subsection 3.3.4, although the pseudocode utilizes this concept in one data structure implementation, as discussed in subsection 5.1.1.

Generic abstract types cannot specify constraints on their type variables. Support-

ing this would be unnecessary, since in DSA implementations, the concrete data structures determine the required properties of the element types, rather than the ADT interface.

However, not all redundancy is avoided. In particular, comparisons in our language can be performed using both method calls to `eq` and `lt` of the built-in `Eq` and `Cmp` abstract types, as well as comparison operators, including `==` and `<`. This redundancy is a result of the design decision to support user-defined comparable types, based on the domain-specific principle.

Moreover, the language provides both `while` and `for` loops. Although supporting just `while` loops would suffice to express any form of repetition, omitting the `for` loop construct would violate the domain-specific principle, since many DSA implementations utilize `for` loops for traversing arrays, sequences of integers, or the elements of a data structure.

5.1.5 Principle: Piggy-Backing on Existing Infrastructure

The implementation of our language utilizes existing tools and infrastructure. In particular, the BNF Converter is used to produce configuration files for lexer and parser generator tools, which generate the lexer and parser for our language. Avoiding manual implementation of these components makes the development of the front end faster and less error-prone.

Furthermore, the decision to implement a transpiler from our language to a high-level language avoids the need to implement complex mechanisms from scratch. Had we instead chosen a low-level target language or compiled to machine code, we would have to put effort into implementing mechanisms to handle issues such as memory allocation and garbage collection. Relying on a target language that handles these automatically allows us to focus on the primary goal of our language.

However, our language does not provide users with the possibility to use external libraries implemented in another language in their programs. Since no external libraries are necessary for the required DSA implementations, providing this feature would be redundant, and therefore a violation of the Keeping It Simple principle. In contrast, transpilation to a high-level language enables DSA implementations written in our language to be used as libraries in the target language.

5.1.6 Principle: Enforcing Good Practice

Many of our design decisions have been made with the purpose of preventing users from adopting bad programming practices, including restricting the possibility of writing error-prone code.

The language uses static typing, which prevents many erroneous programs from executing, since type errors are detected during compilation. In particular, the explicit specification of type constraints in generic definitions prevents them from being instantiated with types that do not have the required properties.

Similarly, the language does not treat `Char` and `Bool` as numeric types. Numbers cannot be used as character or boolean literals. Characters, however, can be explicitly converted to and from integers using the built-in functions `ord` and `chr`. Permitting implicit conversions between these types would result in fewer type errors being detected and reported during compilation.

Additionally, variable declaration is required before use. This prevents accidental use of undeclared or out-of-scope variables, for example, due to misspelling their names. Also, the language does not enable arbitrary transfer of control flow using `goto`, or early exit from loops using `break`. Instead, control flow is directed using constructs such as conditional statements, loops, and function calls. Furthermore, conditional and loop statements require that branches and bodies are always enclosed by curly brackets to make it explicit which statements belong to each branch or body, and to eliminate the risk of users forgetting to enclose a multi-statement block in brackets, as discussed in subsection 3.2.6.

The language also requires that the `self` keyword is explicitly used for accessing class members. This eliminates the risk of accidental shadowing of class fields by local variables and global functions by class methods. Moreover, requiring the definition of the `init` method makes omitting class field initialization a deliberate user choice and not a result of forgetting to define a class constructor.

However, some design decisions contradict this principle. In particular, all class fields and methods are public in our language, which means that information hiding is not supported. This decision is based on the Keeping It Simple principle, since DSA implementations do not require data structures to specify private members. Also, our language does not enforce indentation in control structures, which is considered good practice, as it makes the code more readable and the program structure more explicit. Additionally, the language does not enforce initialization of variables or class fields before use, which means that errors resulting from uninitialized variables can only be detected at runtime.

5.1.7 Principle: Programming is Creating Language

The language provides abstract types and classes as means for users to define their own types. These types can be used in programs in the same way as built-in types. For example, they can be used in arrays or as type arguments to generic definitions, and they can satisfy type constraints. Class objects are allocated using the same syntax as array allocation. Furthermore, classes can use the comparison operators if they are appropriately declared as instances of the built-in `Eq` and `Cmp` abstract types, providing implementations for the `eq` and `lt` methods.

However, our language does not offer the possibility to overload any other operators, such as arithmetic and unary operators. Moreover, users cannot extend the language's syntax by defining their own constructs. These decisions are based on the Keeping It Simple principle. For example, DSA implementations do not require the use of arithmetic operators for any types other than the ones supported by default, and therefore introducing this feature would be redundant.

5.2 Transpiler Implementation

In order to assess the correctness of our implementation, an extensive testsuite has been developed. In particular, there are three categories of tests: positive tests, negative parse tests, and negative typecheck tests.

5.2.1 Positive Tests

Positive tests are syntactically correct and well-typed programs, and should therefore be accepted by the parser and type checker. Once the code generator is implemented, the transpiler should be able to successfully generate a target program for each of these tests, which produces the expected output when executed.

The positive tests cover supported language constructs and features, including primitive operations, strings, arrays, functions, conditional statements, loops, block scoping and variable shadowing, classes, abstract types, instance declarations, subtyping, generics and constraint satisfaction, as well as built-in functions and abstract types.

Every positive test defines a `main` function, which returns the output of the program. This will make code generation testing easier to automate, since a testing script can import each generated module, run its `main` function, and compare the result produced with the expected output.

5.2.2 Negative Parse Tests

Negative parse tests are programs that contain lexical or syntax errors, and should be rejected by the transpiler at the parsing stage. In particular, the error message printed by the transpiler should contain `SYNTAX ERROR` as its first line.

The negative parse tests contain malformed definitions of functions, classes, and abstract types, as well as instance declarations. They also contain invalid syntax for types and variable declarations, use of reserved keywords as variable or function names, invalid specification of type constraints, and lexical errors, such as unclosed strings and unclosed multi-line comments.

5.2.3 Negative Typecheck Tests

Negative typecheck tests are syntactically correct programs that contain semantic errors, and should be rejected by the type checker. In particular, the error message printed by the transpiler should contain `TYPE ERROR` as its first line.

The negative typecheck tests contain semantic errors, such as arity mismatches in generic instantiations and function calls, references to unknown types, variables, functions, fields, or methods, type mismatches in statements and expressions, application of operators to unsupported types, unsatisfied or cyclic type constraints, duplicate definitions or variable declarations, missing or incorrectly typed `init` class methods, use of `self` outside of class methods, cyclic abstract inheritance, and redefinition of built-in functions and abstract types.

5.2.4 Results

All tests in every category passed successfully. The results are summarized as follows:

- Positive tests: 74/74 passed
- Negative parse tests: 51/51 passed
- Negative typecheck tests: 123/123 passed

5.3 Error Messages

In this section, we analyze the type error messages reported for the negative typecheck tests in our testsuite.

The error messages contain information regarding the location of the reported error. In particular, the top-level definition in which the error occurs is reported, such as a function or class. For errors inside function and class method bodies, the statement and, when applicable, the relevant expression that contains the error are also printed. For cases of type or arity mismatch, the messages report the expected and actual types or number of arguments, respectively.

Type error locations are reported by utilizing the structure of the AST. However, source position information is not retained for the semantic analysis phase. As a result, line and column numbers are not reported for type errors. This is a limitation, especially in cases where erroneous statements or expressions are repeated in the same block of code.

Moreover, the messages do not provide hints on how to resolve some potentially trivial errors. For example, when the user tries to access a class field inside a class method but forgets to write the `self` keyword, the type checker reports that the referenced variable is not in scope, given that there is no local variable with the same name. This is less informative than also reporting that the enclosing class definition includes a field with the same name and suggesting that the user may have forgotten to include the `self` keyword while trying to access it. Similarly, unknown name errors for types, variables, or functions due to simple misspellings are not reported in conjunction with potential spelling corrections.

The type checker reports errors using natural language descriptions, but in some cases technical terms are still used. In particular, terms such as “constraint”, “subtype” and “inheritance” occur in error messages to describe relevant errors. Since those terms are not part of the language syntax, unlike terms such as “class”, “abstract” and “instance”, users need to be familiar with them in order to understand the related errors in their programs.

In the case of cyclic constraint errors, the error messages contain a duplicate line, reflecting the constraint checks that led to the detection of the cycle. While the structure of the messages may be indicative of the process that resulted in the detection of the error, the duplicate line could potentially cause confusion.

```
function main() -> Int {
  -foo();
  Int x = "10";
  return 1;
}

function foo() -> Bool {
}
```

```
TYPE ERROR
In (top-level) function main:
In statement '-foo();'
Invalid use of expression '-foo()' as
statement
Only function and method calls can be
used as statements
```

Listing 21: Program containing multiple errors

Error messages for errors in statements inside two-branch conditionals do not specify which branch contains the error. The erroneous statement itself is printed regardless, but the lack of branch-level location information is a limitation, especially in cases where both branches contain similar statements, given that the message does not report a line number.

Only one type error is reported at a time. For example, the unit test shown in Listing 21 contains multiple type errors: a negation expression used as a statement, numerical negation applied to a boolean, and a type mismatch in a variable declaration with initialization. However, only the first detected error is reported. Therefore, in programs with multiple type errors, users must fix one error and recompile the program in order to access the next one. On the other hand, reporting as many errors as possible at once introduces the risk of listing multiple errors with the same underlying cause.

The printed statements and expressions in type error messages are not copied from the source, but are reconstructed using the program's abstract syntax tree. In cases where the user's code contains multiple spaces, unnecessary parentheses, or line breaks, the printed statement or expression may not perfectly resemble it.

6

Conclusion

In this thesis, we designed a programming language for implementing data structures and algorithms in the scope of a Data Structures course. The design of this language was guided by a set of principles, including the domain-specific principle of straightforward DSA implementations, as well as educational design principles described by Kölling [10]. We have also implemented the front end of the transpiler for our language, including semantic analysis and reporting of type error messages.[1]

6.1 Discussion

The goal of this thesis was to design a programming language in which data structures and algorithms can be implemented in a straightforward way. Our analysis shows that this goal has been achieved to a large extent. The language provides constructs that enable the implementation of data structures and algorithms, such as abstract types and classes for expressing ADTs and their respective data structure implementations, generics for reusable DSA implementations, type constraints, functions, fixed-size arrays, and conditional and loop statements.

However, the analysis with respect to the domain-specific principle highlights some structural differences between the DSA implementations in our language and the pseudocode given in the course book [18].

Some of those differences could have been avoided. For example, our language supports two-branch conditional statements, while some DSA implementations use multi-branch conditionals, utilizing an `else if` branch. One way to support multi-branch conditionals is to treat each `else if` branch as syntactic sugar for a nested conditional statement inside an `else` branch. Our language enforces the use of curly brackets to specify conditional branches, based on the principle of Enforcing Good Practice. Had we allowed conditional statements to be used as branches without being enclosed in curly brackets, the multi-branch conditional difference would not have been present.

Similarly, the use of instance declarations to separate the method implementations of ADT operations in our language is a notable difference from the pseudocode in most data structure implementations. An alternative design, closer to that of Java where interface implementations are declared as part of the class definition, would have

[1]The implementation is available at: `https://github.com/grnkl/dsalang`

avoided this difference in structure. Furthermore, expressing iteration over integer sequences using the `for` loop construct for array traversal is less straightforward than relying on a separate construct, although this would result in introducing a third kind of loop in our language, which contradicts the Keeping It Simple principle.

Other differences resulted from the need to explicitly express mechanisms and properties of DSA implementations that are implicit or assumed to hold in the pseudocode. For example, the Collection type in the pseudocode is assumed to be iterable without specifying how the iteration is performed by each data structure implementation. Our language offers traversal by first explicitly turning collections into arrays using the `toArray` method and then by utilizing the `for` loop construct. Similarly, assumptions about the types of data structure elements, such as comparability, are left implicit in the pseudocode. In one case, namely Kruskal's algorithm, these assumptions are even conflicting, as edges need to be compared using different criteria in different parts of the algorithm. Another notable difference concerns the implementation of the Open Addressing Hash Set, where the pseudocode assumes that a special `DELETED` value can be assigned to hash table slots. Instead, the implementation in our language uses a separate boolean array to keep track of deleted slots, as discussed in subsection 5.1.1.7. The need to use a separate array could have been avoided if the language provided an option type, also called a maybe type in some languages such as Haskell. This way, the ability to express two options in conjunction with the use of `null` to express absence would enable the encapsulation of all three slot states in the same array. Yet, the structural differences with the pseudocode implementation would not disappear completely, as wrapping and unwrapping values in the option type would still require special handling.

The design of our language was also guided by educational design principles. Our analysis indicates that the language design mostly follows them, but there are cases where design decisions based on one principle violate another, resulting in trade-offs either between multiple educational design principles or between the domain-specific principle and educational ones.

For example, treating `null` as a valid value for every type is not a common feature, and thus deviates from the Don't Invent principle; however, the decision to do so was based on the domain-specific principle. In particular, it allows generic definitions to express the absence of a value even when instantiated with built-in types, such as `Int` and `Float`. This is utilized in implementations such as the Open Addressing Hash Set, where empty hash table slots are represented using `null`, or in the dummy node of the Skip List implementation, whose key and value are set to `null`. Other languages may handle these cases differently. For example, Java does not allow `null` for primitive types. However, it provides additional wrapper classes for primitives, such as `Integer` and `Float`, that enable the use of `null` and can also be used in generics, unlike primitives [19, §4.5]. Adopting a similar approach in our language would likely lead to confusion and violate the Keeping It Simple principle, as having two forms for the same type introduces redundancy.

Another trade-off involves the overloading of comparison operators for class types. Many DSA implementations involve comparing or sorting elements, using compar-

ison operators in the pseudocode. Following the domain-specific principle and the Programming Is Creating Language principle, the language supports user-defined comparable types by providing the possibility of declaring classes as instances of the built-in `Eq` and `Cmp` abstract types. However, this results in the language offering two ways to perform comparison, namely using both methods and operators, which violates the Keeping It Simple principle, as it introduces multiple ways to solve the same problem. This redundancy could have been resolved by forbidding calls to the `eq` and `lt` methods of the respective built-in abstract types. However, this would likely lead to confusion, as these methods would have had special behavior without appearing different from regular class methods. A similar trade-off between the domain-specific and the Keeping It Simple principles occurs with supporting two loop constructs, as mentioned in subsection 5.1.4.

A design decision that puts the Keeping It Simple and Enforcing Good Practice principles into contradiction is the lack of support for access modifiers for class fields and methods. In our language, all class fields and methods are public, as no private members are specified in any DSA implementation. On the contrary, multiple implementations involve access to member variables and operations, for example, accesses and updates to the `next` node in DSA implementations using linked list structures, and the method call `heap.siftDown(0)` in the implementation of Heap Sort. Therefore, while not having the ability to specify private class members prevents information hiding which is good programming practice, introducing access modifiers in this language would be redundant.

The implementation of our language involves transpiling to a target language. However, our language does not depend on the target language itself, nor do users need to be familiar with the target language in order to use our language. In the context of a Data Structures course, where assignments involve implementing data structures and algorithms, students could write their implementations in our language, compile their programs, and then use an external program provided by the course teachers that imports and runs the implemented algorithms to determine their correctness. Moreover, multiple back ends could be implemented for our language. We are not restricted to choosing just one target language. For example, both Python and Java are languages that provide automatic memory management, so choosing either would be in accordance with the Piggy-Backing on Existing Infrastructure principle.

The implemented transpiler front end reports type error messages in detail. The messages include information about the location of the error, such as which function or method of which class it appears in, as well as the series of checks that led to its detection. Such information is provided in order to help users identify the cause of the error in their code. However, the absence of line and column numbers from type error messages is a limitation that should be addressed. This issue is also relevant for the code generator implementation; runtime errors should be handled by mapping the location of each runtime error in the generated code back to the original location in the source program. Relying on error handling performed by the target language would result in reporting the line number of the generated code, which would confuse users rather than help them identify the error in their program.

6.2 Future Work

The implementation of a code generator for the transpiler is not covered in this thesis; it is currently in progress and will be presented as part of a subsequent thesis. One aim for the code generator is to preserve the structure of the users' source code as much as possible, so that users interested in seeing how their code is translated to the target language can read the transpiler's output. The code generator can also include runtime checks, such as out-of-bounds indexing or division by zero, and report appropriate error messages. In particular, the location of the detected error should correspond to the source program, and not the generated code. Runtime checks would require additional code to be injected into the users' programs, which complicates the aforementioned goal of achieving straightforward correspondence between the source and generated code. However, those checks could be disabled by users with a compile-time flag, if the preference for a particular compilation is readability over source-level runtime error reporting.

Currently, our transpiler does not report detailed error messages about the detected syntax errors. As future work, it should be explored what kind of information the parser could collect and report to help users identify potential causes of such errors. For example, the symbols or tokens that were expected when the syntax error was encountered, or the grammar rules being parsed at that point, may provide useful insight into which syntactic constructs in the user's program are malformed. As Kölling suggests, many syntax errors may not be worth reporting and should be resolved automatically or prevented altogether. This is something that could be investigated in our language, for example, through the development of a programming environment that guides program construction in a way that helps avoid trivial errors, such as missing semicolons or mismatched brackets, or attempts to resolve them whenever possible.

Future work could also explore ways for the language to facilitate reasoning about program behavior and algorithmic complexity. One possibility is to perform static analysis on programs to provide an estimate of their complexity and infer properties, such as termination guarantees, when possible. Another possibility is to collect empirical execution statistics by running programs multiple times with inputs of different sizes and measuring properties such as the number of comparisons performed or the number of assignments to selected variables. Such information could provide insight to students on how algorithms behave asymptotically and help them understand differences between alternative implementations.

Another feature that the language would benefit from is multiple-file program compilation. Several DSA implementations depend on other data structures. For example, graph algorithms may use queues, stacks, maps, or sets. It would therefore be useful to allow data structure implementations to be placed in separate files and imported when needed.

Finally, a crucial direction for future work involves testing our language with students. The language should be used in the context of a Data Structures course to get feedback from students that will help identify the strengths and weaknesses of its

design. Such user-based evaluation of our language will be valuable for determining the appropriate revisions and improvements to its design and implementation.

Bibliography

[1] *The Python Standard Library*, https://docs.python.org/3/library/index.html. Accessed: Jun. 22, 2026.

[2] K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Language*. Addison Wesley Professional, 2005.

[3] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Pearson Education, 1988.

[4] B. Stroustrup, *The C++ Programming Language*. Pearson Education, 2013.

[5] N. Wirth, "The Programming Language Pascal," *Acta informatica*, vol. 1, no. 1, pp. 35–63, 1971.

[6] M. Kölling and J. Rosenberg, "Blue—a language for teaching object-oriented programming," *SIGCSE Bull.*, vol. 28, no. 1, pp. 190–194, Mar. 1996, ISSN: 0097-8418. DOI: 10.1145/236462.236537.

[7] N. Dümmel, B. Westfechtel, and M. Ehmann, "MuLE: a Multiparadigm Language for Education. The Object-Oriented Part of the Language," in *Proceedings of the 4th European Conference on Software Engineering Education*, ser. ECSEE '20, Seeon/Bavaria, Germany: Association for Computing Machinery, 2020, pp. 32–41. DOI: 10.1145/3396802.3396806.

[8] N. Dümmel, B. Westfechtel, and M. Ehmann, "Work in progress: Gathering requirements and developing an educational programming language," in *2019 IEEE Global Engineering Education Conference (EDUCON)*, 2019, pp. 1–4. DOI: 10.1109/EDUCON.2019.8725073.

[9] A. Black, K. B. Bruce, and J. Noble, "Panel: designing the next educational programming language," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, ser. OOPSLA '10, Reno/Tahoe, Nevada, USA: Association for Computing Machinery, 2010, pp. 201–204. DOI: 10.1145/1869542.1869574.

[10] M. Kölling, "Principles of Educational Programming Language Design," *Informatics in Education*, vol. 23, no. 4, pp. 823–836, 2024, ISSN: 1648-5831. DOI: 10.15388/infedu.2024.29.

[11] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006, ISBN: 0321486811.

[12] A. Ranta, *Implementing Programming Languages. An Introduction to Compilers and Interpreters*. College Publications, 2012, ISBN: 1848900643.

[13] M. Pellauer, M. Forsberg, and A. Ranta, "BNF Converter: Multilingual front-end generation from labelled BNF grammars," *Computer Science Chalmers University of Technology and Gothenburg University*, 2004.

[14] M. Forsberg and A. Ranta, "The Labelled BNF Grammar Formalism," *Department of Computing Science, Chalmers University of Technology and the University of Gothenburg*, 2005.

[15] M. Kölling, N. C. C. Brown, and A. Altadmri, "Frame-based editing," English, *Journal of Visual Languages and Sentient Systems*, vol. 3, pp. 40–67, Jul. 2017, ISSN: 2575-3916.

[16] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The Scratch Programming Language and Environment," *ACM Trans. Comput. Educ.*, vol. 10, no. 4, Nov. 2010. DOI: 10.1145/1868358.1868363.

[17] M. Kölling, N. C. C. Brown, and A. Altadmri, "Frame-Based Editing: Easing the Transition from Blocks to Text-Based Programming," in *Proceedings of the Workshop in Primary and Secondary Computing Education*, ser. WiPSCE '15, London, United Kingdom: Association for Computing Machinery, 2015, pp. 29–38. DOI: 10.1145/2818314.2818331.

[18] P. Ljunglöf and A. Gerdes, Eds., *Data Structures and Algorithms*. 2025. Accessed: Jun. 22, 2026. [Online]. Available: https://chalmersgu-data-structure-courses.github.io/dsabook/html.

[19] J. Gosling et al. "The Java Language Specification, Java SE 26 Edition," Accessed: Jun. 22, 2026. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se26/html/.

[20] S. Klabnik, C. Nichols, and C. Krycho, *The Rust Programming Language*. Accessed: Jun. 22, 2026. [Online]. Available: https://doc.rust-lang.org/stable/book/.

[21] *The Python Language Reference*, https://docs.python.org/3/reference/index.html. Accessed: Jun. 22, 2026.

[22] B. C. Pierce, *Types and Programming Languages*. The MIT Press, 2002, ISBN: 0262162091.

[23] L. Cardelli, "Type Systems," in *The Computer Science and Engineering Handbook*, A. B. Tucker, Ed., CRC Press, 1997, ch. 103, pp. 2208–2236, ISBN: 0-8493-2909-4.

[24] S. Kleinschmager, R. Robbes, A. Stefik, S. Hanenberg, and E. Tanter, "Do static type systems improve the maintainability of software systems? An empirical study," in *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, 2012, pp. 153–162. DOI: 10.1109/ICPC.2012.6240483.

[25] S. Hanenberg, S. Kleinschmager, R. Robbes, É. Tanter, and A. Stefik, "An empirical study on the impact of static typing on software maintainability," *Empirical Software Engineering*, vol. 19, Oct. 2013. DOI: 10.1007/s10664-013-9289-1.

[26] S. Spiza and S. Hanenberg, "Type names without static type checking already improve the usability of APIs (as long as the type names are correct): An empirical study," in *Proceedings of the 13th International Conference on Modularity*, ser. MODULARITY '14, Lugano, Switzerland: Association for Computing Machinery, 2014, pp. 99–108. DOI: 10.1145/2577080.2577098.

[27] A. Igarashi, B. C. Pierce, and P. Wadler, "Featherweight Java: a minimal core calculus for Java and GJ," *ACM Trans. Program. Lang. Syst.*, vol. 23, no. 3, pp. 396–450, May 2001, ISSN: 0164-0925. DOI: 10.1145/503502.503505.

[28] *Alex User Guide — Alex documentation*, https://haskell-alex.readthedocs.io/en/latest/. Accessed: Jun. 22, 2026.

[29] A. Gill and S. Marlow, *happy: Happy is a parser generator for Haskell*, https://hackage.haskell.org/package/happy. Accessed: Jun. 22, 2026.

A

Language Grammar in LBNF

```
entrypoints Prog ;

Program. Prog ::= [Definition] ;
separator nonempty Definition "" ;

FunDef. Definition ::=
  "function" Name MaybeTypeVars "(" [Arg] ")" "->" Type MaybeTypeConstraints Blk;

TypeVars. MaybeTypeVars ::= "<<" [Name] ">>" ;
NoTypeVs. MaybeTypeVars ::= ;
separator nonempty Name "," ;

Argument. Arg ::= Type Name ;
separator Arg "," ;

Block. Blk ::= "{" [Stmt] "}" ;
separator Stmt "" ;

ClassDef. Definition ::=
  "class" Name MaybeTypeVars MaybeTypeConstraints "{" [MVDecl] "}" ;

VDecl. MVDecl ::= Type Name ";" ;
MDecl. MVDecl ::= Method;

separator MVDecl "";

AbsDef. Definition ::=
  "abstract" Name MaybeTypeVars MaybeExtends "{" [MethodSig] "}" ;

Extends. MaybeExtends ::= "extends" Name MaybeTypeVars ;
NoExt.   MaybeExtends ::= ;

separator MethodSig "";

InstanceDef. Definition ::= "instance" Name "of" Abstract "{" [Method] "}" ;

Abstract. Abstract ::= Name MaybeTypes ;

separator Method "";

Int.   Type ::= "Int"    ;
Float. Type ::= "Float"  ;
Bool.  Type ::= "Bool"   ;
```

```
Chr.   Type ::= "Char"   ;
Str.   Type ::= "String" ;
Null.  Type ::= "Null"   ;
Array. Type ::= "Array" "[" Type "]" ;
TName. Type ::= Name MaybeTypes ;

Types.   MaybeTypes ::= "<<" [Type] ">>" ;
NoTypes. MaybeTypes ::= ;
separator nonempty Type "," ;

TypeConstraints. MaybeTypeConstraints ::= "where" [TypeConstraint] ;
NoTypeConstr.    MaybeTypeConstraints ::= ;

TypeConstr. TypeConstraint ::= Name "implements" [Abstract] ;

separator nonempty TypeConstraint "and" ;
separator nonempty Abstract "," ;

Methd. Method ::= "function" Name "(" [Arg] ")" "->" Type Blk ;

MSig. MethodSig ::= "function" Name "(" [SigArg] ")" "->" Type ";" ;

SigArg. SigArg ::= Type;
separator SigArg "," ;

SBlock.  Stmt ::= Blk ;
SDecl.   Stmt ::= Type Name MaybeInit ";" ;
SAssgn.  Stmt ::= Exp8 "=" Exp ";" ;
SReturn. Stmt ::= "return" MaybeExp ";" ;
SIf.     Stmt ::= "if" "(" Exp ")" Blk ;
SIfElse. Stmt ::= "if" "(" Exp ")" Blk "else" Blk ;
SWhile.  Stmt ::= "while" "(" Exp ")" Blk ;
SFor.    Stmt ::= "for" "(" Type Name "in" Exp ")" Blk ;
SExp.    Stmt ::= Exp ";" ;

Init.   MaybeInit ::= "=" Exp ;
NoInit. MaybeInit ::= ;

IsExp. MaybeExp ::= Exp;
NoExp. MaybeExp ::= ;

EVar.      Exp9 ::= Name   ;
ELitInt.   Exp9 ::= Integer ;
ELitFloat. Exp9 ::= Double  ;
ELitTrue.  Exp9 ::= "true"  ;
ELitFalse. Exp9 ::= "false" ;
EChar.     Exp9 ::= Char    ;
EString.   Exp9 ::= String  ;
ENull.     Exp9 ::= "null"  ;
ESelf.     Exp9 ::= "self"  ;
EApp.      Exp9 ::= Name MaybeTypes "(" [Exp] ")" ;
ENew.      Exp9 ::= "new" Type "(" [Exp] ")" ;
EIdx.      Exp9 ::= Exp8 "[" Exp "]" ;

EDotF.   Exp8 ::= Exp8 "." Name ;
EDotM.   Exp8 ::= Exp8 "." Name "(" [Exp] ")" ;
```

```
ENot.   Exp7 ::= "not" Exp8 ;
ENeg.   Exp7 ::=  "-"  Exp8 ;
EBWNot. Exp7 ::=  "~"  Exp8 ;

EMul. Exp6 ::= Exp6 MulOp Exp7 ;
EAdd. Exp5 ::= Exp5 AddOp Exp6 ;
ERel. Exp4 ::= Exp4 RelOp Exp5 ;

EBWAnd. Exp3 ::= Exp4  "&"  Exp3 ;
EBWOr.  Exp2 ::= Exp3  "|"  Exp2 ;
EAnd.   Exp1 ::= Exp2 "and" Exp1 ;
EOr.    Exp  ::= Exp1 "or"  Exp ;

coercions Exp 9 ;
separator Exp "," ;

Plus.  AddOp ::= "+" ;
Minus. AddOp ::= "-" ;

Times. MulOp ::= "*" ;
Div.   MulOp ::= "/" ;
Mod.   MulOp ::= "%" ;

LTH. RelOp ::= "<"  ;
LE.  RelOp ::= "<=" ;
GTH. RelOp ::= ">"  ;
GE.  RelOp ::= ">=" ;
EQU. RelOp ::= "==" ;
NE.  RelOp ::= "!=" ;

comment "//" ;
comment "/*" "*/" ;

token Name (letter (letter | digit | '_')*) ;
```

B

Type System Formalization

In this appendix, we describe the type system of our language. We present the contexts and provide the typing rules for the subtyping relation, well-formed types, definitions, statements, and expressions.

The abstract syntax for our language is given in Listing 11. Table B.1 summarizes the notation for the judgments introduced in Section 3.6, which are used in the rules throughout the appendix.

Notation	**Meaning**
$\Theta \vdash \mathrm{S} <: \mathrm{T}$	S is a subtype of T
$\Theta \vdash \mathrm{T}$ ok	type T is well-formed
M ok in $\mathrm{C}\langle\langle\overline{\mathrm{X}} \lhd \overline{\mathrm{P}}\rangle\rangle$	class method definition M is ok
F ok	function definition F is ok
K ok	abstract type or class definition K is ok
$\Theta \vdash$ instance N of O ok	class N is a valid instance of abstract type O
$\Theta; \Gamma \vdash^{\mathrm{T}} \mathrm{s} \Rightarrow \Gamma'$	statement s is well-typed
$\Theta; \Gamma \vdash \mathrm{e} : \mathrm{T}$	expression e has type T

Table B.1: Notation for the judgments used in the inference rules

B.1 Contexts and Auxiliary Functions

Θ is the type variable context, mapping type variables to a set of abstract types (their constraints). We write $\Theta = \overline{\mathrm{X}} <<:$ for the type variable context with domain $\overline{\mathrm{X}}$ and $\Theta(\mathrm{X}) = \emptyset$, for all $\mathrm{X} \in \mathit{dom}(\Theta)$.

Γ is the variable context. It is structured as a stack of blocks Δ, each mapping variable names to types. A block is written as a sequence of mappings $\mathrm{x} : \mathrm{T}$ separated by commas. $\Gamma.\Delta$ denotes the extension of context Γ with block Δ. The empty block is denoted with $()$. $\Gamma(\mathrm{x})$ denotes the type of the top-most occurrence of variable x in context Γ.

The typing rules use several auxiliary function definitions, as shown in Listing 22.

Method names

$$\frac{\mathrm{M} = \mathrm{T}\ \mathrm{m}(\overline{\mathrm{T}})\ \{\}}{\mathit{mname}(\mathrm{M}) = \mathrm{m}}\ (\text{N-Method}) \qquad \frac{\text{abstract}\ \mathrm{A}\langle\!\langle\overline{\mathrm{X}}\rangle\!\rangle\ \{\overline{\mathrm{M}}\}}{\mathit{mnames}(\mathrm{A}\langle\!\langle\overline{\mathrm{T}}\rangle\!\rangle) = \mathit{mname}(\overline{\mathrm{M}})}\ (\text{N-Abs})$$

$$\frac{\text{abstract}\ \mathrm{A}\langle\!\langle\overline{\mathrm{X}}\rangle\!\rangle \lhd \mathrm{O}\ \{\overline{\mathrm{M}}\}}{\mathit{mnames}(\mathrm{A}\langle\!\langle\overline{\mathrm{T}}\rangle\!\rangle) = \mathit{mnames}(\mathrm{O}),\mathit{mname}(\overline{\mathrm{M}})}\ (\text{N-AbsExt})$$

Base type

$$\mathit{base}(\mathrm{Array[T]}) = \mathrm{T}\ (\text{B-Array}) \qquad \mathit{base}(\mathrm{String}) = \mathrm{Char}\ (\text{B-Str})$$

Field lookup

$$\frac{\text{class}\ \mathrm{C}\langle\!\langle\overline{\mathrm{X}} \lhd \overline{\mathrm{P}}\rangle\!\rangle \lhd \overline{\mathrm{O}}\ \{\overline{\mathrm{S}}\ \overline{\mathrm{fd}};\ \overline{\mathrm{M}}\}}{\mathit{fields}(\mathrm{C}\langle\!\langle\overline{\mathrm{T}}\rangle\!\rangle) = [\overline{\mathrm{T}}/\overline{\mathrm{X}}]\overline{\mathrm{S}}\ \overline{\mathrm{fd}}}\ (\text{L-Field}) \qquad \frac{\mathrm{T} \in \{\mathrm{Array[R]}, \mathrm{String}\}}{\mathit{fields}(\mathrm{T}) = \mathrm{Int\ length}}\ (\text{L-Length})$$

Method type lookup

$$\frac{\text{class}\ \mathrm{C}\langle\!\langle\overline{\mathrm{X}} \lhd \overline{\mathrm{P}}\rangle\!\rangle \lhd \overline{\mathrm{O}}\ \{\overline{\mathrm{S}}\ \overline{\mathrm{fd}};\ \overline{\mathrm{M}}\} \qquad \mathrm{R}\ \mathrm{m}(\overline{\mathrm{R}}\ \overline{\mathrm{x}})\ \{\overline{\mathrm{s}}\} \in \overline{\mathrm{M}}}{\mathit{mtype}_\Theta(\mathrm{m},\ \mathrm{C}\langle\!\langle\overline{\mathrm{T}}\rangle\!\rangle) = [\overline{\mathrm{T}}/\overline{\mathrm{X}}](\overline{\mathrm{R}} \to \mathrm{R})}\ (\text{L-ClassMethod})$$

$$\frac{\text{abstract}\ \mathrm{A}\langle\!\langle\overline{\mathrm{X}}\rangle\!\rangle\ \{\overline{\mathrm{M}}\} \qquad \mathrm{R}\ \mathrm{m}(\overline{\mathrm{R}})\ \{\} \in \overline{\mathrm{M}}}{\mathit{mtype}_\Theta(\mathrm{m},\ \mathrm{A}\langle\!\langle\overline{\mathrm{T}}\rangle\!\rangle) = [\overline{\mathrm{T}}/\overline{\mathrm{X}}](\overline{\mathrm{R}} \to \mathrm{R})}\ (\text{L-AbsMethod})$$

$$\frac{\text{abstract}\ \mathrm{A}\langle\!\langle\overline{\mathrm{X}}\rangle\!\rangle \lhd \mathrm{O}\ \{\overline{\mathrm{M}}\} \qquad \mathrm{m} \notin \overline{\mathrm{M}}}{\mathit{mtype}_\Theta(\mathrm{m},\ \mathrm{A}\langle\!\langle\overline{\mathrm{T}}\rangle\!\rangle) = \mathit{mtype}_\Theta(\mathrm{m},\ [\overline{\mathrm{T}}/\overline{\mathrm{X}}]\mathrm{O})}\ (\text{L-SupMethod})$$

$$\frac{\mathrm{O} \in \Theta(\mathrm{X}) \qquad \mathit{mtype}_\Theta(\mathrm{m},\ \mathrm{O}) = \overline{\mathrm{R}} \to \mathrm{R}}{\mathit{mtype}_\Theta(\mathrm{m},\ \mathrm{X}) = \overline{\mathrm{R}} \to \mathrm{R}}\ (\text{L-TVMethod})$$

$$\frac{\Theta \vdash \mathrm{T} <: \mathrm{Eq}\langle\!\langle\mathrm{T}\rangle\!\rangle}{\mathit{mtype}_\Theta(\mathrm{eq},\ \mathrm{T}) = \mathrm{T} \to \mathrm{Bool}}\ (\text{L-Eq}) \qquad \frac{\Theta \vdash \mathrm{T} <: \mathrm{Cmp}\langle\!\langle\mathrm{T}\rangle\!\rangle}{\mathit{mtype}_\Theta(\mathrm{lt},\ \mathrm{T}) = \mathrm{T} \to \mathrm{Bool}}\ (\text{L-Cmp})$$

$$\frac{\Theta \vdash \mathrm{T} <: \mathrm{Hashable}\langle\!\langle\mathrm{T}\rangle\!\rangle}{\mathit{mtype}_\Theta(\mathrm{hashCode},\ \mathrm{T}) = \epsilon \to \mathrm{Int}}\ (\text{L-Hashable})$$

Function type lookup

$$\frac{\mathrm{F} = \mathrm{R}\ \mathrm{f}\langle\!\langle\overline{\mathrm{X}} \lhd \overline{\mathrm{O}}\rangle\!\rangle(\overline{\mathrm{R}}\ \overline{\mathrm{x}})\ \{\overline{\mathrm{s}}\}}{\mathit{ftype}(\mathrm{f}) = \langle\!\langle\overline{\mathrm{X}} \lhd \overline{\mathrm{O}}\rangle\!\rangle\overline{\mathrm{R}} \to \mathrm{R}}\ (\text{L-Function})$$

Listing 22: Auxiliary functions

The *mname* function is defined over method definitions and returns the name of the given method. We write $\mathit{mname}(\overline{\mathrm{M}})$ for $\mathit{mname}(\mathrm{M}_1)$,...,$\mathit{mname}(\mathrm{M}_\mathrm{n})$. Similarly, the *mnames* function is defined over abstract types and returns the names of the given type's methods, including those defined in its supertype.

The *base* function is defined over Array types and the String type, returning the Array base type in the former case and Char in the latter.

The *fields* function is defined over class types, Array types, and the String type. For classes, it returns their field types and names, while for String and Array types, it returns the integer length field.

The *mtype* function is defined for class types, abstract types, (bound) type variables, and built-in types that are either hashable or support comparison. It maps a method name of a given type to its type signature. In rule L-HASHABLE, ϵ is used to express that the function has no arguments (and therefore no argument types). $\mathrm{M} \in \overline{\mathrm{M}}$ means that the sequence $\overline{\mathrm{M}}$ of method definitions contains definition M, while $\mathrm{m} \notin \overline{\mathrm{M}}$, as shown in rule L-SUPMETHOD, means that $\overline{\mathrm{M}}$ contains no method named m. The *ftype* function maps function names to their signatures, containing the type variables and their constraints.

In order for the *fields*, *mtype*, and *ftype* functions to be well-defined, several sanity conditions need to hold. In particular, user-defined types must have distinct names, there must be no duplicate function definitions, duplicate fields, or duplicate methods in the same class or abstract type, including its supertypes.

B.2 Subtyping

The judgment $\Theta \vdash \mathrm{S} <: \mathrm{T}$ means that type S is a subtype of T under type variable context Θ. As shown in Listing 23, the subtyping relation is reflexive and transitive. Type variables are subtypes of the abstract types listed in the given type variable context. Abstract types are subtypes of the abstract types they extend, while class types are subtypes of the abstract types they implement. Int is a subtype of Float, and the Null type is a subtype of all types. A type T supports equality comparison if it is a subtype of $\mathrm{Eq}\langle\langle \mathrm{T} \rangle\rangle$ (similarly for general comparison and $\mathrm{Cmp}\langle\langle \mathrm{T} \rangle\rangle$). Rule S-CMPEQ reflects that comparable types support equality comparison. Built-in types Int, Float, Bool, Char, and String are hashable.

B.3 Well-Formed Types

We write $\Theta \vdash \mathrm{T}$ ok to express that type T is well-formed under type parameter context Θ. We write $\Theta \vdash \overline{\mathrm{T}}$ ok to abbreviate the judgments $\Theta \vdash \mathrm{T}_1$ ok,...,$\Theta \vdash \mathrm{T}_\mathrm{n}$ ok (similarly for $\Theta \vdash \overline{\mathrm{S}} <: \overline{\mathrm{T}}$). We write $\Theta \vdash \overline{\mathrm{S}} <<: \overline{\mathrm{T}}$ for $\Theta \vdash \mathrm{S}_1 <: \overline{\mathrm{T}}_1$,...,$\Theta \vdash \mathrm{S}_\mathrm{n} <: \overline{\mathrm{T}}_n$, where $\Theta \vdash \mathrm{S} <: \overline{\mathrm{T}}$ stands for $\Theta \vdash \mathrm{S} <: \mathrm{T}_1$,...,$\Theta \vdash \mathrm{S} <: \mathrm{T}_\mathrm{n}$. In rule WF-ABSEXT (Listing 24), $\overline{\mathrm{Y}} \subseteq \overline{\mathrm{X}}$ means that the type variable sequence $\overline{\mathrm{X}}$ contains all members of the type variable sequence $\overline{\mathrm{Y}}$.

According to the rules in Listing 24, the basic types are well-formed; in the case of Array types, the base type must be well-formed. Type variables are well-formed if they are bound in the given type variable context. Class and abstract types require that their type arguments be well-formed. In the case of classes, they also need to satisfy the constraints declared in the class definitions.

Subtyping

$$\Theta \vdash \mathrm{T} <: \mathrm{T} \quad \text{(S-Refl)} \qquad \frac{\Theta \vdash \mathrm{R} <: \mathrm{S} \qquad \Theta \vdash \mathrm{S} <: \mathrm{T}}{\Theta \vdash \mathrm{R} <: \mathrm{T}} \quad \text{(S-Trans)}$$

$$\frac{\mathrm{O} \in \Theta(\mathrm{X})}{\Theta \vdash \mathrm{X} <: \mathrm{O}} \quad \text{(S-TVar)} \qquad \frac{\text{abstract } \mathrm{A}\langle\langle \overline{\mathrm{X}} \rangle\rangle \lhd \mathrm{O}\ \{...\}}{\Theta \vdash \mathrm{A}\langle\langle \overline{\mathrm{T}} \rangle\rangle <: [\overline{\mathrm{T}}/\overline{\mathrm{X}}]\mathrm{O}} \quad \text{(S-Abs)}$$

$$\frac{\text{class } \mathrm{C}\langle\langle \overline{\mathrm{X}} \lhd \overline{\mathrm{P}} \rangle\rangle \lhd \overline{\mathrm{O}}\ \{...\} \qquad \mathrm{O} \in \overline{\mathrm{O}}}{\Theta \vdash \mathrm{C}\langle\langle \overline{\mathrm{T}} \rangle\rangle <: [\overline{\mathrm{T}}/\overline{\mathrm{X}}]\mathrm{O}} \quad \text{(S-Class)} \qquad \Theta \vdash \mathrm{Null} <: \mathrm{T} \quad \text{(S-Null)}$$

$$\Theta \vdash \mathrm{Int} <: \mathrm{Float} \quad \text{(S-Num)} \qquad \Theta \vdash \mathrm{Bool} <: \mathrm{Eq}\langle\langle \mathrm{Bool} \rangle\rangle \quad \text{(S-EqBool)}$$

$$\Theta \vdash \mathrm{Cmp}\langle\langle \mathrm{T} \rangle\rangle <: \mathrm{Eq}\langle\langle \mathrm{T} \rangle\rangle \quad \text{(S-CmpEq)} \qquad \frac{\mathrm{T} \in \{\mathrm{Int}, \mathrm{Float}, \mathrm{Bool}, \mathrm{Char}, \mathrm{String}\}}{\Theta \vdash \mathrm{T} <: \mathrm{Hashable}\langle\langle \mathrm{T} \rangle\rangle} \quad \text{(S-Hash)}$$

$$\frac{\mathrm{T} \in \{\mathrm{Int}, \mathrm{Float}, \mathrm{Char}, \mathrm{String}\}}{\Theta \vdash \mathrm{T} <: \mathrm{Cmp}\langle\langle \mathrm{T} \rangle\rangle} \quad \text{(S-Cmp)}$$

Listing 23: Subtyping

Well-formed types

$$\frac{\mathrm{T} \in \{\mathrm{Int}, \mathrm{Float}, \mathrm{Bool}, \mathrm{Char}, \mathrm{String}, \mathrm{Null}\}}{\Theta \vdash \mathrm{T}\ \mathrm{ok}} \quad \text{(WF-Basic)}$$

$$\frac{\Theta \vdash \mathrm{T}\ \mathrm{ok}}{\Theta \vdash \mathrm{Array}[\mathrm{T}]\ \mathrm{ok}} \quad \text{(WF-Array)} \qquad \frac{\mathrm{X} \in \mathit{dom}(\Theta)}{\Theta \vdash \mathrm{X}\ \mathrm{ok}} \quad \text{(WF-TVar)}$$

$$\frac{\text{abstract } \mathrm{A}\langle\langle \overline{\mathrm{X}} \rangle\rangle\ \{...\} \qquad \Theta \vdash \overline{\mathrm{T}}\ \mathrm{ok}}{\Theta \vdash \mathrm{A}\langle\langle \overline{\mathrm{T}} \rangle\rangle\ \mathrm{ok}} \quad \text{(WF-Abs)}$$

$$\frac{\text{abstract } \mathrm{A}_1\langle\langle \overline{\mathrm{X}} \rangle\rangle \lhd \mathrm{A}_2\langle\langle \overline{\mathrm{Y}} \rangle\rangle\ \{...\} \qquad \overline{\mathrm{Y}} \subseteq \overline{\mathrm{X}} \qquad \Theta \vdash \overline{\mathrm{T}}\ \mathrm{ok}}{\Theta \vdash \mathrm{A}_1\langle\langle \overline{\mathrm{T}} \rangle\rangle\ \mathrm{ok}} \quad \text{(WF-AbsExt)}$$

$$\frac{\text{class } \mathrm{C}\langle\langle \overline{\mathrm{X}} \lhd \overline{\mathrm{P}} \rangle\rangle \lhd \overline{\mathrm{O}}\ \{...\} \qquad \Theta \vdash \overline{\mathrm{T}}\ \mathrm{ok} \qquad \Theta \vdash \overline{\mathrm{T}} <<: [\overline{\mathrm{T}}/\overline{\mathrm{X}}]\overline{\mathrm{P}}}{\Theta \vdash \mathrm{C}\langle\langle \overline{\mathrm{T}} \rangle\rangle\ \mathrm{ok}} \quad \text{(WF-Class)}$$

Listing 24: Well-formed types

B.4 Method and Function Typing

Listing 25 contains the typing rules for methods and functions.

A method definition is ok in a class definition if the types of its arguments and its return type are well-formed. Additionally, its body must be well-typed under the type variable context corresponding to the type constraints of the class, and under

Method typing

$$\frac{\Theta = \overline{X} <: \overline{P} \qquad \Theta \vdash \overline{T}, T \text{ ok} \qquad \Theta; (\overline{x} : \overline{T},\ \text{self} : C\langle\langle\overline{X}\rangle\rangle) \vdash^{T} \overline{s} \Rightarrow \Gamma \qquad \text{class } C\langle\langle\overline{X} \lhd \overline{P}\rangle\rangle \lhd \overline{O}\ \{...\}}{T\ m(\overline{T}\ \overline{x})\ \{\overline{s}\}\ \text{ok in } C\langle\langle\overline{X} \lhd \overline{P}\rangle\rangle} \quad \text{(T-CMethod)}$$

$$\frac{\Theta = \overline{X} <: \qquad \Theta \vdash \overline{T}, T \text{ ok} \qquad \text{abstract } A\langle\langle\overline{X}\rangle\rangle\ \{...\}}{T\ m(\overline{T}\ \overline{x})\ \{\}\ \text{ok in } A\langle\langle\overline{X}\rangle\rangle} \quad \text{(T-AMethod)}$$

$$\frac{\Theta = \overline{X} <: \qquad \Theta \vdash \overline{T}, T \text{ ok} \qquad \text{abstract } A\langle\langle\overline{X}\rangle\rangle \lhd O\ \{...\}}{T\ m(\overline{T}\ \overline{x})\ \{\}\ \text{ok in } A\langle\langle\overline{X}\rangle\rangle} \quad \text{(T-AextMethod)}$$

Function typing

$$\frac{\Theta = \overline{X} <: \overline{O} \qquad \Theta \vdash \overline{O}, \overline{T}, T \text{ ok} \qquad \Theta; (\overline{x} : \overline{T}) \vdash^{T} \overline{s} \Rightarrow \Gamma}{T\ f\langle\langle\overline{X} \lhd \overline{O}\rangle\rangle(\overline{T}\ \overline{x})\ \{\overline{s}\}\ \text{ok}} \quad \text{(T-Function)}$$

Listing 25: Method and function typing rules

the variable context mapping the method's arguments to their types and self to the class type.

Similarly, a method is ok in an abstract type definition if the types of its arguments and its return type are well-formed. Recall that abstract methods do not have bodies.

A function definition is ok if its constraints, as well as the types of its arguments and its return type, are well-formed. Additionally, its body must be well-typed.

B.5 Abstract and Class Typing

Listing 26 contains the typing rules for abstract types and classes.

An abstract type definition is ok if the abstract methods are ok and its supertype (if there is one) is well-formed.

A class definition is ok if all of its methods are ok, including the special init method with Null return type. Additionally, the types of its fields as well as its supertypes and constraints must be well-formed, and the class must be a valid instance of all abstract types it implements.

A class type N is a valid instance of an abstract type O, denoted by the judgment $\Theta \vdash$ instance N of O ok, if for all abstract methods of O, including inherited methods, there exists a class method definition in N with the same signature.

In rules T-Class and T-Ins, we write $\Theta \vdash$ instance $C\langle\langle X\rangle\rangle$ of $\overline{O}$ ok to abbreviate the judgments $\Theta \vdash$ instance $C\langle\langle X\rangle\rangle$ of O_1 ok,..., instance $C\langle\langle X\rangle\rangle$ of O_n ok (similarly for $\Theta \vdash$ N implements $\overline{m}$ of O).

Abstract and class typing

$$\frac{\overline{\mathrm{M}}\ \text{ok in}\ \mathrm{A}\langle\langle\overline{\mathrm{X}}\rangle\rangle}{\text{abstract}\ \mathrm{A}\langle\langle\overline{\mathrm{X}}\rangle\rangle\ \{\overline{\mathrm{M}}\}\ \text{ok}}\quad \text{(T-Abs)}$$

$$\frac{\Theta = \overline{\mathrm{X}} <:\qquad \Theta \vdash \mathrm{O}\ \text{ok}\qquad \overline{\mathrm{M}}\ \text{ok in}\ \mathrm{A}\langle\langle\overline{\mathrm{X}}\rangle\rangle}{\text{abstract}\ \mathrm{A}\langle\langle\overline{\mathrm{X}}\rangle\rangle \lhd \mathrm{O}\ \{\overline{\mathrm{M}}\}\ \text{ok}}\quad \text{(T-AbsExt)}$$

$$\frac{\begin{array}{c}\Theta = \overline{\mathrm{X}} <: \overline{\mathrm{P}}\qquad \Theta \vdash \overline{\mathrm{O}},\overline{\mathrm{P}},\overline{\mathrm{T}}\ \text{ok}\qquad \overline{\mathrm{M}}\ \text{ok in}\ \mathrm{C}\langle\langle\overline{\mathrm{X}} \lhd \overline{\mathrm{P}}\rangle\rangle \\ \text{Null init}(\overline{\mathrm{S}}\ \overline{\mathrm{x}})\ \{\overline{\mathrm{s}}\} \in \overline{\mathrm{M}}\qquad \Theta \vdash \text{instance}\ \mathrm{C}\langle\langle\overline{\mathrm{X}}\rangle\rangle\ \text{of}\ \overline{\mathrm{O}}\ \text{ok}\end{array}}{\text{class}\ \mathrm{C}\langle\langle\overline{\mathrm{X}} \lhd \overline{\mathrm{P}}\rangle\rangle \lhd \overline{\mathrm{O}}\ \{\overline{\mathrm{T}}\ \overline{\mathrm{fd}};\ \overline{\mathrm{M}}\}\ \text{ok}}\quad \text{(T-Class)}$$

$$\frac{mtype_\Theta(\mathrm{m},\mathrm{N}) = mtype_\Theta(\mathrm{m},\mathrm{O})}{\Theta \vdash \mathrm{N}\ \text{implements m of O}}\quad \text{(T-MImpl)}$$

$$\frac{\begin{array}{c}mnames(\mathrm{O}) = \overline{\mathrm{m}} \\ \Theta \vdash \mathrm{N}\ \text{implements}\ \overline{\mathrm{m}}\ \text{of O}\end{array}}{\Theta \vdash \text{instance N of O ok}}\quad \text{(T-Ins)}$$

Listing 26: Abstract and class typing rules

B.6 Statement Typing

Listing 27 contains the typing rules for statements in our language.

An empty sequence of statements, denoted by ϵ in rule T-EmptySeq, is trivially well-typed. A non-empty sequence is well-typed under contexts Θ and Γ if the first statement is well-typed under these contexts, changing Γ to Γ_1, and the subsequent statements are well-typed under Θ and Γ_1.

A block of statements is well-typed under Θ and Γ if the statement sequence it encloses is well-typed under Θ and the variable context Γ extended by an empty block.

A variable declaration is well-typed under Θ and Γ if the variable name is not a member of the top block of the variable context Γ. The statement extends that block by adding the mapping of the variable to its type. If the statement also initializes the variable with an expression, then the type of the expression must match the type of the variable.

Similarly, an assignment of an expression to a variable is well-typed if the type of the expression matches the type of the variable, and a class field assignment is well-typed if the type of the expression matches the type of the instantiated class field. Array element assignments are well-typed if the type of the assigned expression matches the base type of the array expression, and the index expression has type Int.

A conditional statement is well-typed if its branches are well-typed and the condition expression is boolean.

Similarly, a while loop is well-typed if the condition expression is boolean and the loop body is well-typed.

Statement typing

$$\Theta;\Gamma \vdash^{\mathrm{T}} \epsilon \Rightarrow \Gamma \quad \text{(T-EmptySeq)} \qquad \frac{\Theta;\Gamma \vdash^{\mathrm{T}} \mathrm{s} \Rightarrow \Gamma_1 \qquad \Theta;\Gamma_1 \vdash^{\mathrm{T}} \overline{\mathrm{s}} \Rightarrow \Gamma_2}{\Theta;\Gamma \vdash^{\mathrm{T}} \mathrm{s}\ \overline{\mathrm{s}} \Rightarrow \Gamma_2} \quad \text{(T-Sequence)}$$

$$\frac{\Theta;\Gamma.() \vdash^{\mathrm{T}} \overline{\mathrm{s}} \Rightarrow \Gamma'}{\Theta;\Gamma \vdash^{\mathrm{T}} \{\overline{\mathrm{s}}\} \Rightarrow \Gamma} \quad \text{(T-Block)} \qquad \frac{\Theta \vdash \mathrm{T}\ \mathrm{ok} \qquad \mathrm{x} \notin \mathit{dom}(\Delta) \qquad \Theta;\Gamma.\Delta \vdash \mathrm{e} : \mathrm{T}}{\Theta;\Gamma.\Delta \vdash^{\mathrm{T}} \mathrm{T}\ \mathrm{x} = \mathrm{e};\ \Rightarrow \Gamma.(\Delta, \mathrm{x}{:}\mathrm{T})} \quad \text{(T-Init)}$$

$$\frac{\Theta;\Gamma \vdash \mathrm{e} : \Gamma(\mathrm{x})}{\Theta;\Gamma \vdash^{\mathrm{T}} \mathrm{x} = \mathrm{e};\ \Rightarrow \Gamma} \quad \text{(T-VarAssgn)}$$

$$\frac{\Theta;\Gamma \vdash \mathrm{e}_1 : \mathrm{Array[T]} \qquad \Theta;\Gamma \vdash \mathrm{e}_2 : \mathrm{Int} \qquad \Theta;\Gamma \vdash \mathrm{e}_3 : \mathrm{T}}{\Theta;\Gamma \vdash^{\mathrm{R}} \mathrm{e}_1[\mathrm{e}_2] = \mathrm{e}_3;\ \Rightarrow \Gamma} \quad \text{(T-ArrAssgn)}$$

$$\frac{\Theta;\Gamma \vdash \mathrm{e}_1 : \mathrm{N} \qquad \mathit{fields}(\mathrm{N}) = \overline{\mathrm{T}}\ \overline{\mathrm{fd}} \qquad \Theta;\Gamma \vdash \mathrm{e}_2 : \mathrm{T_i}}{\Theta;\Gamma \vdash^{\mathrm{R}} \mathrm{e}_1.\mathrm{fd_i} = \mathrm{e}_2;\ \Rightarrow \Gamma} \quad \text{(T-FldAssgn)}$$

$$\frac{\Theta;\Gamma \vdash \mathrm{e} : \mathrm{Bool} \qquad \Theta;\Gamma \vdash^{\mathrm{T}} \{\overline{\mathrm{s}}_1\} \Rightarrow \Gamma \qquad \Theta;\Gamma \vdash^{\mathrm{T}} \{\overline{\mathrm{s}}_2\} \Rightarrow \Gamma}{\Theta;\Gamma \vdash^{\mathrm{T}} \mathrm{if}\ (\mathrm{e})\ \{\overline{\mathrm{s}}_1\}\ \mathrm{else}\ \{\overline{\mathrm{s}}_2\} \Rightarrow \Gamma} \quad \text{(T-IfElse)}$$

$$\frac{\Theta \vdash \mathrm{S}\ \mathrm{ok} \qquad \Theta;\Gamma \vdash \mathrm{e} : \mathrm{S} \qquad \mathit{base}(\mathrm{S}) = \mathrm{T} \qquad \Theta;\Gamma.(\mathrm{x}{:}\mathrm{T}) \vdash^{\mathrm{R}} \overline{\mathrm{s}} \Rightarrow \Gamma'}{\Theta;\Gamma \vdash^{\mathrm{R}} \mathrm{for}\ (\mathrm{T}\ \mathrm{x}\ \mathrm{in}\ \mathrm{e})\ \{\overline{\mathrm{s}}\} \Rightarrow \Gamma} \quad \text{(T-For)}$$

$$\frac{\Theta;\Gamma \vdash \mathrm{e} : \mathrm{Bool} \qquad \Theta;\Gamma \vdash^{\mathrm{T}} \{\overline{\mathrm{s}}\} \Rightarrow \Gamma}{\Theta;\Gamma \vdash^{\mathrm{T}} \mathrm{while}\ (\mathrm{e})\ \{\overline{\mathrm{s}}\} \Rightarrow \Gamma} \quad \text{(T-While)} \qquad \frac{\Theta;\Gamma \vdash \mathrm{e} : \mathrm{T}}{\Theta;\Gamma \vdash^{\mathrm{T}} \mathrm{return}\ \mathrm{e};\ \Rightarrow \Gamma} \quad \text{(T-ERet)}$$

$$\frac{\Theta;\Gamma \vdash \mathrm{f}\langle\!\langle\overline{\mathrm{R}}\rangle\!\rangle(\overline{\mathrm{e}}) : \mathrm{S}}{\Theta;\Gamma \vdash^{\mathrm{T}} \mathrm{f}\langle\!\langle\overline{\mathrm{R}}\rangle\!\rangle(\overline{\mathrm{e}});\ \Rightarrow \Gamma} \quad \text{(T-FCallStmt)} \qquad \frac{\Theta;\Gamma \vdash \mathrm{e.m}(\overline{\mathrm{e}}) : \mathrm{S}}{\Theta;\Gamma \vdash^{\mathrm{T}} \mathrm{e.m}(\overline{\mathrm{e}});\ \Rightarrow \Gamma} \quad \text{(T-MCallStmt)}$$

Listing 27: Statement typing rules

A for loop is well-typed under context Γ if the type of the loop variable matches the base type of the expression being traversed. Additionally, the loop body must be well-typed under context Γ extended by a block mapping the loop variable to its type.

A return statement is well-typed if the type of the returned expression matches the return type of the enclosing function.

Function and method calls are valid statements if the corresponding call expressions are well-typed.

B.7 Expression Typing

The typing rules for expressions are given in Listings 28, 29, and 30.

If an expression has type R and R is a subtype of type T, then the expression also

Expression typing (subtyping and base expressions)

$$\frac{\Theta;\Gamma \vdash \mathrm{e} : \mathrm{R} \qquad \Theta \vdash \mathrm{R} <: \mathrm{T}}{\Theta;\Gamma \vdash \mathrm{e} : \mathrm{T}} \quad \text{(S-Exp)}$$

$$\Theta;\Gamma \vdash \mathrm{x} : \Gamma(\mathrm{x}) \quad \text{(T-Var)} \qquad \Theta;\Gamma \vdash \mathrm{i} : \mathrm{Int} \quad \text{(T-IntLit)}$$

$$\Theta;\Gamma \vdash \mathrm{d} : \mathrm{Float} \quad \text{(T-FloatLit)} \qquad \Theta;\Gamma \vdash \mathrm{true} : \mathrm{Bool} \quad \text{(T-True)}$$

$$\Theta;\Gamma \vdash \mathrm{false} : \mathrm{Bool} \quad \text{(T-False)} \qquad \Theta;\Gamma \vdash \mathrm{c} : \mathrm{Char} \quad \text{(T-CharLit)}$$

$$\Theta;\Gamma \vdash \mathrm{str} : \mathrm{String} \quad \text{(T-StrLit)} \qquad \Theta;\Gamma \vdash \mathrm{null} : \mathrm{Null} \quad \text{(T-NullLit)}$$

Listing 28: Expression typing rules (subtyping and base expressions)

Expression typing (allocation and calls)

$$\frac{ftype(\mathrm{f}) = \langle\!\langle \overline{\mathrm{X}} \lhd \overline{\mathrm{O}} \rangle\!\rangle \overline{\mathrm{R}} \to \mathrm{R} \qquad \Theta \vdash \overline{\mathrm{T}}\ \mathrm{ok} \qquad \Theta \vdash \overline{\mathrm{T}} <<: [\overline{\mathrm{T}}/\overline{\mathrm{X}}]\overline{\mathrm{O}} \qquad \Theta;\Gamma \vdash \overline{\mathrm{e}} : [\overline{\mathrm{T}}/\overline{\mathrm{X}}]\overline{\mathrm{R}}}{\Theta;\Gamma \vdash \mathrm{f}\langle\!\langle \overline{\mathrm{T}} \rangle\!\rangle(\overline{\mathrm{e}}) : [\overline{\mathrm{T}}/\overline{\mathrm{X}}]\mathrm{R}} \quad \text{(T-FCallExp)}$$

$$\frac{\Theta \vdash \mathrm{N}\ \mathrm{ok} \qquad mtype_{\Theta}(\mathrm{init},\mathrm{N}) = \overline{\mathrm{T}} \to \mathrm{Null} \qquad \Theta;\Gamma \vdash \overline{\mathrm{e}} : \overline{\mathrm{T}}}{\Theta;\Gamma \vdash \mathrm{new}\ \mathrm{N}(\overline{\mathrm{e}}) : \mathrm{N}} \quad \text{(T-NewClass)}$$

$$\frac{\Theta \vdash \mathrm{Array[T]}\ \mathrm{ok} \qquad \Theta;\Gamma \vdash \mathrm{e} : \mathrm{Int}}{\Theta;\Gamma \vdash \mathrm{new\ Array[T](e)} : \mathrm{Array[T]}} \quad \text{(T-NewArr)}$$

$$\frac{\Theta;\Gamma \vdash \mathrm{e} : \mathrm{Array[Char]}}{\Theta;\Gamma \vdash \mathrm{new\ String(e)} : \mathrm{String}} \quad \text{(T-NewStr)} \qquad \frac{\Theta;\Gamma \vdash \mathrm{e}_1 : \mathrm{T} \qquad \Theta;\Gamma \vdash \mathrm{e}_2 : \mathrm{Int}}{\Theta;\Gamma \vdash \mathrm{e}_1[\mathrm{e}_2] : base(\mathrm{T})} \quad \text{(T-Idx)}$$

$$\frac{\Theta;\Gamma \vdash \mathrm{e} : \mathrm{T} \qquad fields(\mathrm{T}) = \overline{\mathrm{T}}\ \overline{\mathrm{fd}}}{\Theta;\Gamma \vdash \mathrm{e.fd_i} : \mathrm{T_i}} \quad \text{(T-Field)}$$

$$\frac{\Theta;\Gamma \vdash \mathrm{e} : \mathrm{T} \qquad mtype_{\Theta}(\mathrm{m},\mathrm{T}) = \overline{\mathrm{R}} \to \mathrm{R} \qquad \Theta;\Gamma \vdash \overline{\mathrm{e}} : \overline{\mathrm{R}}}{\Theta;\Gamma \vdash \mathrm{e.m}(\overline{\mathrm{e}}) : \mathrm{R}} \quad \text{(T-MCallExp)}$$

Listing 29: Expression typing rules (allocation and calls)

has type T.

Top-level function calls require that the type arguments used to instantiate the function satisfy the constraints on its type parameters. In addition, the expressions used as function arguments must match the function's instantiated signature. Its return type is also the type of the function call expression.

New class object allocation expressions require that the arguments match the signature of the class's init method.

Array allocation requires an integer argument for the length, while String allocation

Expression typing (operators)

$$\frac{\Theta;\Gamma \vdash e : \text{Bool}}{\Theta;\Gamma \vdash \text{not } e : \text{Bool}} \text{ (T-Not)} \qquad \frac{\Theta;\Gamma \vdash e : T \qquad T \in \{\text{Int,Float}\}}{\Theta;\Gamma \vdash \text{-}e : T} \text{ (T-Neg)}$$

$$\frac{\Theta;\Gamma \vdash e : \text{Int}}{\Theta;\Gamma \vdash \sim e : \text{Int}} \text{ (T-BWNeg)} \qquad \frac{\Theta;\Gamma \vdash e_1, e_2 : \text{Bool} \qquad \odot \in \{\text{and, or}\}}{\Theta;\Gamma \vdash e_1 \odot e_2 : \text{Bool}} \text{ (T-BoolOp)}$$

$$\frac{\Theta;\Gamma \vdash e_1, e_2 : \text{Int} \qquad \odot \in \{\&,|\}}{\Theta;\Gamma \vdash e_1 \odot e_2 : \text{Int}} \text{ (T-BWOp)}$$

$$\frac{\Theta;\Gamma \vdash e_1, e_2 : T \qquad T \in \{\text{Int,Float}\} \qquad \odot \in \{+,-,*,/,\%\}}{\Theta;\Gamma \vdash e_1 \odot e_2 : T} \text{ (T-Arith)}$$

$$\frac{\Theta;\Gamma \vdash e_1, e_2 : T \qquad \Theta \vdash T <: \text{Eq}\langle\langle T \rangle\rangle \qquad \odot \in \{==, !=\}}{\Theta;\Gamma \vdash e_1 \odot e_2 : \text{Bool}} \text{ (T-Eq)}$$

$$\frac{\Theta;\Gamma \vdash e_1, e_2 : T \qquad \Theta \vdash T <: \text{Cmp}\langle\langle T \rangle\rangle \qquad \odot \in \{<,<=,>,>=\}}{\Theta;\Gamma \vdash e_1 \odot e_2 : \text{Bool}} \text{ (T-Cmp)}$$

$$\frac{\Theta;\Gamma \vdash e_1 : S \qquad \Theta;\Gamma \vdash e_2 : T \qquad \text{Null} \in \{S,T\} \qquad \odot \in \{==, !=\}}{\Theta;\Gamma \vdash e_1 \odot e_2 : \text{Bool}} \text{ (T-NullEq)}$$

Listing 30: Expression typing rules (operators)

requires an argument of type Array[Char].

Indexed expressions for arrays and strings require that the index expression has type Int, and their type is the array's base type and Char, respectively.

Field access is well-typed if the type of the expression being accessed contains the accessed field. For arrays and strings, field access is supported for the length property.

Method calls require that the expressions used as arguments match the method's signature. Its return type is the type of the method call expression.

Boolean conjunction, disjunction, and negation are supported for boolean expressions. The equivalent bit-wise operations are supported for integer expressions.

Numerical negation is well-typed if applied to Int and Float expressions. Similarly for binary arithmetic operations (+, -, *, /, %).

Equality (==, !=) and comparison (<, <=, >=, >) operators are overloaded for all types T implementing Eq⟪T⟫ and Cmp⟪T⟫, respectively.

Equality comparison between two well-typed expressions is also well-typed if at least one of them has type Null.

C

Error Message Templates

This chapter presents the templates for the type error messages reported by the transpiler during semantic analysis. Section C.1 describes the notation used to describe the error messages. The remaining sections describe different kinds of error message segments, namely segments providing context about the location of the errors and checks that led to their detection, as well as descriptions of the detected errors themselves.

C.1 Notation

Table C.1 presents the notation and meta-variables used in the error message templates in subsequent tables.

Notation	Meaning
`An`	abstract type name
`Cn`	class name
`Tv`	type variable name
`x`	variable or field name
`m`	method name
`f`	function name
`Nm`	any name (class, function, variable, etc.)
`Aty`	abstract type
`Cty`	class type
`Ty`	any type
`n`	number
`st`	statement
`exp`	expression
$\overline{\texttt{Tv}}$, $\overline{\texttt{Nm}}$, ...	non-empty sequences

Table C.1: Notation used in error message templates

C.2 Context Information

Type error messages contain information regarding the location of the error. In particular, the top-level definition where the error occurs, such as a function or class. For errors inside function or method bodies, the erroneous statement is reported as well. Table C.2 presents the message segments related to the location of the error in the source program. Note that when a template contains alternatives separated by /, only one alternative appears in concrete error messages.

Location Context
`In (top-level) function f:`
`In method m of class Cn:`
`In instance Cn of An:`
`In definition of class / abstract type / function Nm:`
`In declaration of field x of class Cn:`
`In statement 'st':`

Table C.2: Location context

Type error messages also include intermediate steps, corresponding to the sequence of checks that led to the detection of the error. Table C.3 shows how different checks are reported, providing additional context to the overall error description.

Intermediate Checks Leading to Error Detection
`While checking the validity of type arguments of constraint Aty:`
`While checking if type arguments of class Cty satisfy Cn's constraints:`
`While checking the validity of constraints in class/function Nm:`
`While checking if Aty1 is a subtype of Aty2:`
`While checking array base type: Ty`
`While checking the validity of type arguments of class/abstract Ty:`
`While checking the validity of argument types in function f / method m of class/abstract Nm:`
`While checking the return type of function f / method m of class/abstract Nm:`
`While checking the type arguments of Aty:`
`While checking validity of type: Ty`
`While checking expression 'exp':`
`Function/method Nm is called with invalid arguments:`

Table C.3: Intermediate reported checks

C.3 Error Descriptions

The tables in this section present the error message segments providing the description of the type errors, grouped by category.

Table C.4 shows error message segments describing unknown type errors; for example, errors resulting from the use of undefined abstract types in type constraints and instance declarations.

Unknown Type Errors
`Tv implements unknown abstract: An`
`Class/Function Nm contains constraints on unbound type variables:` $\overline{\texttt{Tv}}$
`Unknown type: Ty`
`Abstract An1 extends undefined abstract An2`
`Abstract An extends abstract An with invalid type parameters` `Type variables not in scope: Tv`
`The following classes are undefined, but are declared to instantiate an abstract:` $\overline{\texttt{Cn}}$
`The following abstracts are undefined, but are instantiated by classes:` $\overline{\texttt{An}}$

Table C.4: Unknown type errors

Table C.5 presents the error message segments describing unknown name errors, namely errors caused by referencing undefined functions, out-of-scope variables, undeclared fields and methods.

Unknown Name Errors
`Call to unknown function: f`
`Variable Nm not in scope`
`Class Cn has no field named Nm`
`Class/Abstract type Nm has no method named Nm`
`Ty has no method named m`
`Type variable Tv implements no abstracts containing a method named Nm`
`Type Null/Array[Ty] has no method named foo` `(Array types and Null have no methods)`

Table C.5: Unknown name errors

Tables C.6 and C.7 present message segments describing arity and type mismatch errors. The former are caused by instantiating a generic, or calling a function or type constructor with different number of arguments than expected. The latter occurs when an expression or type argument does not match the expected type.

Arity Mismatch Errors

`Wrong number of type arguments for abstract/class/function Nm`
`Expected: n1`
`Got: n2`

`Abstract An extends abstract An with wrong number of type parameters`
`Expected: n1`
`Got: n2`

`Wrong number of arguments in new Array/String allocation`
`Expected: n1`
`Got: n2`

`Function f expects n1 argument(s), but was called with n2.`

Table C.6: Arity mismatch errors

Type Mismatch Errors

`Type argument mismatch when instantiating abstract An`
`Expected: Ty1`
`Got: Ty2`

`Return statements in non-Null functions must specify a return value`

`Expression with unexpected type: exp`
`Expected: Ty1`
`Got: Ty2`

`init method of class Cn has invalid signature.` `Expected Null return type, got Ty instead.`

Table C.7: Type mismatch errors

Table C.8 contains error message segments corresponding to invalid abstract type, class, and instance declarations. In particular, errors reported when there is a cycle in the subtype relation, when a class definition is missing its constructor method, or when an instance declaration does not provide exactly the required method definitions.

Class and Instance Validity Errors
`Detected cyclic abstract inheritance involving An`
`No init method provided for class Cn`
`Instance Cn of An defines extra methods:` $\overline{\texttt{m}}$
`Missing implementations for the following methods:` $\overline{\texttt{m}}$

Table C.8: Class and instance validity errors

Constraint Satisfaction Errors
`Abstract An1 does not implement abstract An2`
`Cyclic constraint: Tv implements Aty`
`Type variable Tv does not implement constraint Aty`
`Class Cn1 implements no abstracts so it cannot be a subtype of Aty, required by Cn2`
`Class Cn1 does not implement Aty required by Cn2`
`Null cannot satisfy constraint Aty` `(Null implements no abstracts)`
`Array[Ty] cannot satisfy constraint Aty` `(Array types implement no abstracts)`

Table C.9: Constraint satisfaction errors

Unsupported or Invalid Operation Errors
`Indexed assignment not supported for Strings (due to immutability)`
`Assignment to special 'length' field of Strings and Arrays not supported`
`Illegal assignment statement` `Assignment is supported for variables, class fields, and array elements`
`Invalid use of expression 'exp' as statement` `Only function and method calls can be used as statements`
`Use of 'self' outside of class method`
`New object allocation not supported for type Ty` `Only String, Array, and class types are supported`
`Indexing used for type other than String or Array: Ty`
`Invalid Array/String field: x` `The only valid Array/String field is 'length'`
`Field access not supported for type: Ty`

`Comparison/Additive/Multiplicative operations not supported between Ty1 and Ty2`
`Left operand (Ty1): exp1`
`Right operand (Ty2): exp2`

`Comparison/Equality comparison not supported for type Ty`
`Left operand: exp1`
`Right operand: exp2`
`(Type Ty does not implement Eq<<Ty>>/Cmp<<Ty>>)`

Table C.10: Unsupported or invalid operation errors

Constraint satisfaction errors, shown in Table C.9, occur when a generic type or function is instantiated by types that do not satisfy its constraints.

Table C.10 contains descriptions of errors caused by invalid use of operations, such as comparing elements of different types or using the `self` keyword outside of class methods.

Uniqueness errors, presented in Table C.11, are caused by duplicate declarations of types, functions, and variables, or by a type variable shadowing a class or abstract type.

Uniqueness Errors
`The following type constraints appear more than once:` `Tv implements An`
`The following fields in class Cn are declared more than once:` $\overline{\texttt{x}}$
`The following methods in class Cn are declared more than once:` $\overline{\texttt{m}}$
`Duplicate type parameters in definition of class / abstract type / function Nm: Tv`
`In definition of class/abstract type/function Nm:` `Type variable Tv shadows the class/abstract type Nm`
`Abstract An defines the following methods more than once:` $\overline{\texttt{m}}$
`The following methods in abstract An were already defined in its supertype hierarchy:` $\overline{\texttt{m}}$
`Instance Cn of An defines duplicate methods:` $\overline{\texttt{m}}$
`The following method(s) defined in class Cn are also defined in abstract An (or its supertype hierarchy):` $\overline{\texttt{m}}$
`Class Cn implements multiple abstracts defining method(s):` $\overline{\texttt{m}}$
`In function f / method m of class Cn the following parameter names are used more than once:` $\overline{\texttt{x}}$
`Redefinition of built-in function/abstract Nm`
`The following functions/classes/abstracts/instances are defined more than once:` $\overline{\texttt{Nm}}$
`The following names are used both as class names and abstract names:` $\overline{\texttt{Nm}}$
`Variable x already defined in the inner scope`
`Type variable Tv implements multiple abstracts containing method Nm:` $\overline{\texttt{An}}$

Table C.11: Uniqueness errors