A Construct-Based Evaluation of Kotlin-to-Swift and Swift-to-Kotlin Transpilers

Master's Degree Program: Media Informatics

Master's Thesis

by

Larissa Manon Elisabeth Katharina Schneider
born in Regensburg

delivered at the
Department of Information Technology-Electrical Engineering-Mechatronics, Friedberg (Hessen)

Supervisor: Prof. Dr. rer. nat. Dominik Schultes
Co-Supervisor: Lisa Ranold, M. Sc.

Friedberg, 2022

Abstract

With Google's Android and Apple's iOS holding almost the entire market share for mobile operating systems, third-party app developers should address both platforms for reaching the most users possible. However, the tools intended by Google and Apple for developing apps natively are specific to the two platforms. In addition to a particular software development kit, different programming languages are required, with Kotlin being the primary language for Android and Swift being the primary language for iOS and its operating system variants. Since this makes code sharing between the platforms impossible, developers are often at a crossroads when choosing the appropriate approach for developing their app. Multiple cross-platform tools and frameworks, aiming to reduce the double effort of considering both platforms, emerged over the years. Most of them achieve their goal by providing an architecture that allows running one non-native code base on multiple target platforms by employing a layer of abstraction. This, however, creates a dependency on the continuous development of the tool used, since it has to adapt its architecture to updates of the operating system.

An alternative approach would be to use a transpiler, translating the model and business logic parts of an application from one platform's native programming language to the other, making the output code bases independently maintainable. In order to gain insight into the current state of the art in this respect, this thesis evaluates the transpilers Gryphon (Swift-to-Kotlin), Kotlift (Kotlin-to-Swift), SequalsK (both directions) and SwiftKotlin (Swift-to-Kotlin) on their support of a set of basic constructs, taken from the overview chapters of the Kotlin and Swift documentations. In addition to the mere support of a construct, the readability of the output code is also examined by assessing it both manually and with a linter based on acknowledged style guidelines for that language. Moreover, the results on construct support are given more practical relevance by also considering the occurrence of a transpiler's unsupported constructs in open-source app projects. Data on construct occurrence was obtained by counting the appearance of the relevant constructs in those projects by using a tool specifically implemented for this thesis, referred to as the Construct Analyzer Tool. This tool identified the constructs of interest from a parse tree, i.e., a model representation of the input code file. Notably, constructs whose identification depended on declarations made in a project's external dependencies could not be recognized.

The results regarding construct support revealed that Gryphon and SequalsK can be classified as the most mature, with both supporting $\sim$74% of the considered Swift constructs and SequalsK supporting $\sim$78% of the Kotlin constructs. Meanwhile, SwiftKotlin shows only satisfactory results by supporting $\sim$68% of the Swift constructs and Kotlift merely sufficient

support by considering ∼54% of the Kotlin constructs. The evaluation of the valid output w.r.t. readability unveiled that all transpilers produce generally readable code, with Kotlift and SequalsK (for both directions of translation) achieving very good and Gryphon and SwiftKotlin achieving good compliance with the respective language's style guidelines.

For every construct unsupported by one or more transpilers, its occurrence within every project of the respective language, normalized with that project's logical lines of code, was calculated. The resulting average value of all projects was established as a construct's popularity value. By setting the popularity values of all constructs unsupported by a transpiler into relation with the popularity values of all constructs unsupported by one or more transpilers of the same translation direction, the score W was determined. W presents an indication of the manual effort required to correct a transpiler's output code and ranges from 0 to 1, with 0 being the worst possible score and 1 being the best possible score. W.r.t. the Kotlin-to-Swift transpilers, SequalsK proved to be far more applicable than Kotlift, scoring a value of 0.964 for W while Kotlift only achieved 0.013. Regarding the Swift-to-Kotlin transpilers, Gryphon achieved a W score of 0.680 and SequalsK of 0.622, while SwiftKotlin scored last with a value of 0.263 for W. While a discrepancy between W for SequalsK and Kotlift was to be expected, since Kotlift supports considerably less constructs than SequalsK, the results for the transpilers of the Swift-to-Kotlin translation direction prove that some constructs impact transpiler applicability more than others.

For further studies on construct popularity, the accuracy of the Construct Analyzer Tool would profit from including declarations from a project's external dependencies, since excluding those resulted in a considerable amount of false negatives for constructs whose identification depends on declarations made there. But regardless, the results obtained within this thesis on the transpilers' construct support only reflect their current situation, since their future releases might include support for currently disregarded constructs. Recommended directions for further studying Kotlin-to-Swift and Swift-to-Kotlin transpilers are to consider more constructs and to test the transpilers on the model and business logic parts of app projects from practice.

Acknowledgements

In the following, I would like to thank the people who have supported me during this thesis in various ways. Even when my path was occasionally paved with obstacles, they have helped me to get closer to my goals one step at a time.

First, I would like to express my sincere gratitude towards my supervisor, Prof. Dr. Dominik Schultes, for providing me with the opportunity to conduct a thesis in this field. Throughout the past months, his professional and moral support helped me again and again to find new motivation and continue working on this thesis. Moreover, his valuable feedback helped me to improve its overall quality. Furthermore, I would like to thank Ms. Lisa Ranold, M. Sc., for taking on the role of co-supervisor. I would also like to thank the Technische Hochschule Mittelhessen, which provided me with a device on which I could conduct the experiments for my research.

Besides, I would like to show my gratefulness towards my friends and family, who each individually played a role in enabling me to realize this thesis. I would especially like to thank my mother Gudrun Gauß-Schneider and my longtime friend Barbara Grotz, who both proofread this thesis. Some special words of gratitude also go to my fellow students Pascal Schüler, Niklas Spies and Sebastian Wolzenburg, with whom I was able to face the challenges of our Master's program together.

Finally, I would like to acknowledge the moral and emotional support of everyone else who patiently and helpfully stood by me during this demanding, but rewarding time. Thank you very much!

Selbstständigkeitserklärung

Ich erkläre, dass ich die eingereichte Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Ich versichere, dass diese Arbeit in gleicher oder ähnlicher Form noch keiner anderen Hochschule oder Prüfungsstelle vorlag.

Friedberg, Juli 2022

Larissa Manon Elisabeth Katharina Schneider

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Smartphones are everywhere. Touchscreen interface based mobile phones with so-called smart features, e.g., internet connection, integration of various sensors and the ability to install applications, have become vital to the digitization of everyday life. This is also reflected in the global number of smartphone users. With around 80% of the population owning a smartphone, North America and Europe lead in this regard [GSM21]. Demand for third-party smartphone applications, more commonly referred to as (mobile) apps, became even more popular in the past two years, when the COVID-19 pandemic resulted in stay-at-home orders [Sen21]. On top of that, app development was deemed crisis relevant, with governments worldwide commissioning apps offering services like tracking contacts or displaying proof of vaccination.

Still, when trying to cover the most users possible, mobile third-party app developers are faced with the particular challenge of platform fragmentation. Nowadays, the **operating system (OS)** market is almost exclusively divided into the Android OS from Google and Apple's iOS [Sta22b]. In conclusion, access to a smartphone app should be provided for both platforms for extensive market coverage. Notably, mobile phones are not the only target devices for Android or for iOS and its variants, like macOS, watchOS, or tvOS. Nowadays, the ecosystems of both platforms include, next to smartphones, tablets, smartwatches, computers and smart TVs. Consequently, third-party app development is also possible for those devices.

1.1 Motivation

Specialized development for a target platform, normally referred to as native development, is the approach favored and officially supported by OS distributors. It requires using a designated programming language, e.g., Kotlin or Java for Android and Swift or ObjectiveC for Apple platforms, as well as a **software development kit (SDK)** provided by Google or Apple. However, this results in the inability to directly share code snippets between platforms and thus doubles the development effort and needed expertise when addressing both.

Given these circumstances, multiple approaches to cross-platform app development have been established in recent years. While different taxonomies and architectural concepts exist

in the literature, the Hybrid, the Interpreted and the Web approach are among the most popular and widely discussed [BHGG19]. While this thesis is not a detailed comparison of existing approaches to cross-platform development, those three approaches are described briefly in the following for contextualizing their architectural background.

Mobile web apps are websites enhanced for smartphones. As every modern mobile OS has a browser, they are cross-platform by nature. Recently, progressive web apps even tackle multiple infamous problems concerning mobile web apps, like offline-access [Moz22]. Moreover, the standard web technologies HTML5, CSS3, and JS are often complemented by frameworks like ReactJS [Met22b] or Angular [Goo22b]. However, mobile web apps cannot be offered and downloaded via the OS specific marketplace. Hybrid apps are also developed with web technologies, with their source code running in a browser environment which accesses device functionalities through a layer of abstraction. This architecture is provided by frameworks like Cordova [The22a] or Capacitor [Cap22], which make it also possible to export the project in the OS app format. Thus, it can be published in the official Android and iOS stores. Lastly, the Interpreted approach enables targeting multiple platforms with its architecture, too. Frameworks like React Native [Met22a] or Flutter [Goo22d] use a bridge layer to translate between the JavaScript-/Dart-source-code and the native code, as well as OS specific components at runtime. Projects can also be exported to the OS app format and published in the OS specific marketplace.

In summary, those three approaches promise a reduced development effort by deploying a one-timely written, non-native code base to multiple target platforms. This is made possible by running the code in a cross-platform available environment, like a browser, and/or communicating with OS specific components via a layer of abstraction. However, all app projects relying on a framework's or library's specific non-native architecture or programming language ultimately create a dependency on the tool's continuous development, resulting in an unmaintainable code base once it reached its end of life [VGM20, Sch21]. Additionally, apps created with cross-platform tools might experience disadvantages compared to natively developed apps, like restricted communication with OS-specific components, worse performance [MLL+18, QGZ16] and a lack of the "look and feel" the OS is usually known for [RS12].

In an attempt to overcome these drawbacks of popular cross-platform development tools, there have been some efforts to translating native code to native code by using a so-called transpiler translating between mobile programming languages. Thereby, two independently maintainable code bases can coexist. However, while other cross-platform development tools cover the entirety of an app, transpilers usually prioritize the translation of non-platform specific code. I.e., when given a **model-view-controller (MVC)** pattern fairly often used as the base architecture of an app, only the model and business logic parts are translatable. Although separating presentation layer and business logic is a common pattern in mobile app design, neglecting such platform specific parts, especially view components, can be classified as a disadvantage. At the same time, the enforcement of abstraction layers and/or non-native interfaces is avoided.

While the J2ObjC [Goo21b] transpiler pioneered the transpiler approach in the field of mobile development by translating from Java to ObjectiveC, this thesis focuses on transpilers for the current primary programming languages for Android and iOS, Kotlin and Swift. For

examining the current state of the art in Kotlin-to-Swift and Swift-to-Kotlin transpilers, the four transpilers Gryphon [VGM20, Ven22a], Kotlift [Stu20], SequalsK [Sch21, Sch22] and SwiftKotlin [Oll20] were selected for further examination. While Gryphon and SwiftKotlin focus on the translation of Swift to Kotlin code, Kotlift enables the reverse process from Kotlin to Swift. SequalsK supports bidirectional translation.

1.2 Problem Definition

On a surface level, Swift and Kotlin appear to share many syntactic similarities, making them seemingly suitable for the transpiler approach [Mec17, Oll16]. Nevertheless, going beyond simple differences like constant keywords or different notations for various kinds of loops, it becomes quickly apparent that a simple search and replace is not sufficient [Sch21]. Even though more complex constructs are considered by the aforementioned transpilers to a varying degree, all authors admit to deficits in language coverage [Oll20, Ven22a, Stu20, Sch21]. Consequently, the question about the transpilers' current operational capabilities in a cross-platform development context arises. One way of approaching this question would be by analyzing and comparing the transpilers' translation capabilities. In light of independent code base maintainability, the readability of their output code should be taken into account as well. But to this date, no such study could be found in the literature, other than the preliminary work done for this thesis [SS22].

The aforementioned paper was the result of the development project preparing this thesis. There, the Gryphon, Kotlift and SequalsK transpilers were already tested on the language constructs presented in the overview chapters of the respective official documentations of Kotlin and Swift, named "Basic syntax" and [Jet22a] "A Swift Tour" [App22c]. Even though the constructs presented there do not show the full capabilities of both languages, they do provide an overview of commonly used features and some best practices. In general, the results suggested that language coverage of common constructs appears to be achievable. While SequalsK and Gryphon obtained good results, Kotlift's language support was merely sufficient. Some shortcomings affected all transpilers, such as "fundamental differences between Swift and Kotlin or constructs that were simply not considered" [SS22]. Nevertheless, the valid output code of all transpilers followed a majority of acknowledged style guidelines.

Regarding the deficits in language coverage, it must be taken into account that all of these projects are much smaller compared to popular cross-platform tools like React Native that is published by Meta. Consequently, those deficits are probably not just due to faults in the transpiler approach, but due to the restrictions created by little manpower. Therefore, besides profiting from an analysis of their translation capabilities, the transpiler projects themselves would also benefit from gaining insight into an unsupported construct's popularity in every-day programming projects. Ultimately, this would help to focus efforts on practically relevant areas. As no literature thoroughly analyzing language construct usage in neither Swift nor Kotlin exists thus far (see Chapter 3), the need for conducting such a study is created. Ultimately, this yields three research questions for this thesis:

- *RQ1*: From a set of basic constructs featured in the input programming language, which are supported by the transpilers?

- *RQ2*: Does the output code generated by the transpilers follow the language's style guidelines?

- *RQ3*: How does the popularity of a construct unsupported by a transpiler affect its applicability?

    RQ3.1: How popular is a certain construct in practice?

1.3 Objectives

The primary contribution of this thesis is to present a practically relevant evaluation of Gryphon, Kotlift, SequalsK and SwiftKotlin by judging their applicability despite their unsupported constructs by taking the popularity of those constructs into account. However, since considering all aspects of Kotlin and Swift is outside the scope of this thesis, the transpilers are tested on a set of basic but representative constructs instead. The popularity-metric may be obtained by analyzing a statistically relevant sized sample pool of Kotlin and Swift programming projects. At the same time, dealing with such massive amounts of code files results in the need for automated construct recognition, since manually counting constructs would simply not be feasible. Therefore, the results of this study regarding the popularity of the considered Kotlin and Swift constructs present a secondary contribution of this thesis. When taking those two main goals into account, the objectives of this thesis can be ultimately separated into four successive sections.

1.3.1 Repeating the Experiments on Construct Support

Since the transpilers are under ongoing development, the study regarding construct support conducted in the preliminary work should be repeated [SS22]. Again, the test case pool should be formed out of the code examples given in the overview chapters of the Kotlin and Swift documentations to test the transpilers on a set of basic constructs. However, by adding the SwiftKotlin transpiler to the once already evaluated transpilers Gryphon, Kotlift and SequalsK, even more insight can be gained on the current state of the art regarding the transpiler approach.

1.3.2 Mining for Kotlin/Swift App Projects

For extracting information on construct usage, a mining software repository study should be conducted as part of this work. In order to be able to fall back on a statistically relevant sample pool for this study, a sufficient amount of projects has to be collected. In the context of this work, it therefore makes sense to mine open-source projects from a source like GitHub [Git22b]. Since Gryphon, Kotlift, SequalsK and SwiftKotlin primarily advertise themselves as translation tools between Android and Apple platforms, the sample pool should be narrowed down to projects of this kind.

1.3.3 Automatic Analysis of Code Structures

To provide the study on construct support with more practical relevance, the popularity of the constructs unsupported by one or more transpilers should be further examined. Consequently, the files from the Kotlin and Swift project sample pools should be checked for the occurrences of those constructs. Due to the presumably large amount of code, manually counting each construct would likely be too time-consuming. Instead, an automated counting is aspired by recognizing constructs from the file's parse tree. The tool should be able to handle both Kotlin and Swift files, so implementations of abstract concepts can be shared. Furthermore, to reduce the development effort, the parser generator ANTLR should be used to create the parts necessary to build a parse tree.

1.3.4 Evaluating Transpilers Considering the Popularity of Their Unsupported Constructs

After determining the occurrence of the constructs, the transpilers are evaluated again. This time, however, this occurrence-metric is taken into account when comparing the shortcomings of the transpilers. All in all, the final results should represent an indication of the manual effort required to correct the transpilers' output.

1.4 Outline

The subsequent part of this thesis is divided into five parts — the theoretical background, related works from the literature, the methodology employed, a discussion of the results and the final summary of this work with recommendations for future works.

In the upcoming Chapter 2, the reader is familiarized with terminology and concepts important for understanding the underlying topics of this thesis. This includes basic information on Kotlin and Swift, the principles behind programming language parsing and a general definition of transpilers.

Then, this thesis' contribution is placed in the context of the current state of the art in Chapter 3. For this purpose, past works on native source-to-source translators for cross-platform Android and Apple OS development are introduced. This also includes the transpilers considered in this thesis, Gryphon, Kotlift, SequalsK and SwiftKotlin. In addition, work focused on the translatability of Kotlin and Swift and the adoption of those two languages is presented. Furthermore, information on other works with ANTLR is given.

The subsequent three chapters are guided by the research questions of this thesis and present the methodology used to attempt to answer them. Accordingly, Chapter 4 elaborates on the experiment setup for testing a transpiler's support of the basic constructs from Kotlin's and Swift's overview chapters. This also includes verifying the compliance of a transpiler's output code with general coding style guidelines. First, however, the test case pool is described in detail. Afterwards, the execution of the experiments is illustrated step-by-step, including the environment for the experiments and the transpiler versions used.

In the following Chapter 5, the mining and preparing of the Kotlin and Swift project sample pools is described. For this purpose, the criteria that classified a project as relevant

for this thesis are provided. It is also elaborated on how these criteria were identified. The chapter concludes by discussing how noise factors were removed and how metrics describing the sample pools were extracted.

Chapter 6 presents the Construct Analyzer Tool developed for automatically counting construct appearance within the aforementioned sample pools. Firstly, its basic workflow is presented. Secondly, the tool's implementation is described in detail. Lastly, the accuracy observed when testing the functionality of the tool on an unrepresentative number of files from the sample pools is reported.

Subsequently, Chapter 7 summarizes the results from the previous methodology chapters. For this purpose, the results regarding construct support and style guideline compliance are presented. Afterwards, the findings regarding the unsupported constructs' occurrences are disclosed. Ultimately, those results are used for evaluating the transpilers once again in a more practice oriented fashion, revealing how their shortcomings potentially affect the need for post-translation edits.

At last, Chapter 8 concludes this thesis. To this end, the reader is provided with a summary of all the key findings from this work. Since this thesis was not possibly able to fully examine all aspects of Kotlin-to-Swift and Swift-to-Kotlin transpilers, recommendations on how the transpiler approach can be further studied in the context of cross-platform app development are given. Even more so, propositions are made on how the Construct Analyzer Tool could be improved.

Chapter 2

Theoretical Background

This chapter provides an overview on the theoretical background this thesis is built upon. After giving general information on the Kotlin and Swift programming languages, the theory behind programming language parsing is explained in detail. This includes briefly introducing the parser generator ANTLR and describing its workflow for language parsing. Finally, transpilers, which are a certain type of language processor making use of the previously detailed parsing steps, are explored.

The goal of this chapter is to familiarize the reader with the principles and technologies used in the following chapters without diving too deeply into their application in this thesis or work related to this thesis. These points will be discussed later in Chapter 3.

2.1 Kotlin

Kotlin is a general-purpose, multi-paradigm and statically typed open-source programming language [Jet22d]. Although Kotlin is majorly used for Android development, it can be utilized in many different fields, like server-side applications, web fronted development, data science or even mobile multiplatform development [Jet21]. Furthermore, an alternative to the traditional way of defining **user interfaces (UIs)** for Android apps with XML layout files is made possible by writing declarative Kotlin code using Jetpack Compose [Goo22e].

Nevertheless, Kotlin was originally developed independently of the Android platform. According to JetBrains, the company behind Kotlin, they needed a programming language fulfilling their needs that were unaddressed by Java, which had been used by them thus far [Kri11]. Kotlin improves on Java by having a less verbose and more expressive syntax on the one hand, but offering features making the code more fail-safe, like null-checks before compiling and safely accessing nullable values, on the other. Due to the company's history with Java projects, Kotlin has full Java interoperability since the beginning. The language was first made publicly available as open-source in 2011 [Ore12]. Google has been offering first-class Android support for Kotlin since 2017 and promoted it to be the preferred language for native Android Apps in 2019 [Goo21a].

2.2 Swift

Similar to Kotlin, Swift is a general-purpose, multi-paradigm and statically typed programming language initially developed by Apple [App22d]. It was introduced at the 2014 WWDC and exclusively released in the same year [Dow16]. At the following WWDC 2015, Swift 2 was announced to be open-source.

Swift was developed with the intention of eventually replacing ObjectiveC, which had been the official, but aged, programming language for Apple systems for approximately 30 years. To prevent the need for total migration of ObjectiveC projects to Swift, Swift offers great ObjectiveC interoperability. Additionally, Swift is designed to be a more modern language with features like optional typing, functional programming patterns and safety measures, e.g., automatically preventing null objects or overflowing arrays. Furthermore, its syntax is concise and less verbose. Although the ultimate goal is for Swift to be compatible on as many platforms as possible [App22e], only Linux is currently supported next to the Apple platforms, namely iOS, macOS, watchOS, and tvOS. For the remainder of this thesis, these aforementioned Apple platforms will be summarized by the term Apple OS.

Swift can be utilized in a diverse set of projects, ranging from systems programming to cloud services. When using Swift for app development, it can also be used for declaratively defining the UI by using the SwiftUI framework [App22g]. This proposes an alternative to the traditional way of defining layouts with designated layout files.

2.3 Programming Language Parsing

After taking a closer look at the two programming languages relevant for this thesis, one might ask how translating between them is possible in the first place. For this, understanding how programming language processing works in general is vital. This section explains the steps necessary to parse a piece of programming language code, so it can be further processed.

Source code files boiled down are nothing other than text. Although those text files possess different file endings, e.g., *.kt* for Kotlin files or *.swift* for Swift files, and the way of defining programming routines might vastly differ from language to language, their content can be handled by any simple text editor. To decipher the instructions given to build a processable data structure, a special program, referred to as a parser, is used. Figure 2.1 (see page 9) visualizes the phases of programming language parsing. Firstly, the input source code, most likely represented by a text file, which is made up of characters, is scanned and processed into tokens in the lexical analysis phase. In the context of programming language parsing, a token is the basic lexical unit. The token stream is then analyzed in the syntax analysis phase on its validity in terms of how programs in that particular programming language are defined. The syntax analysis usually produces a model representation of the input source code called a parse tree. The semantic validity of this structure, including issues such as type compatibility, is then verified in the semantic analysis phase. However, this phase is sometimes skipped and further processing operates directly on the parse tree generated by the syntax analysis phase. The language processors ultimately handling the output parse tree

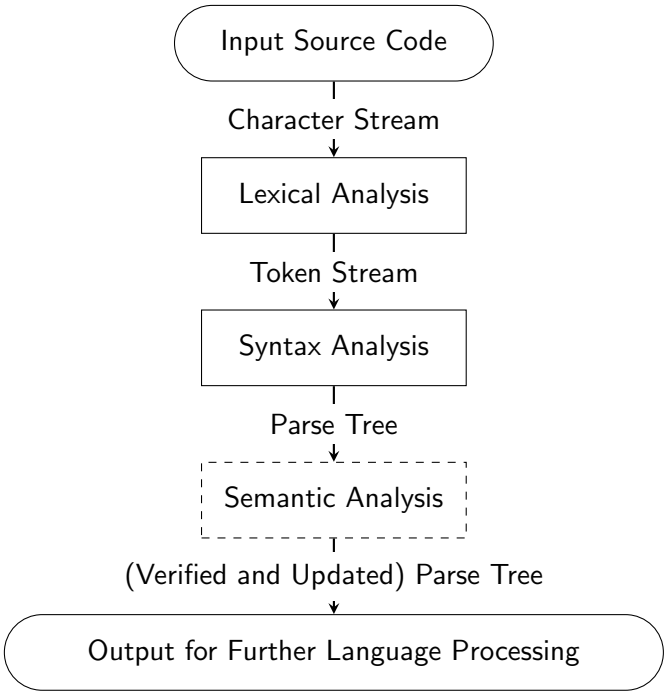


Figure 2.1: Phases of programming language parsing.

include, among others, compilers generating machine-readable code, interpreters executing operations directly in the source program and translators producing an output in an entirely different high-level language.

2.3.1 Grammar

Before the individual phases are explained in more detail, it is shown how the syntactic rules for a programming language are defined. For humans, learning how to read a natural language, like English or German, encompasses two areas. Firstly, the grammar of the language, which can defined as a set of rules to form syntactically and morphologically correct sentences [The20], has to be studied. Secondly, a vocabulary that can be used within that set of rules has to be acquired.

Even though programming languages are by definition formal languages that do not serve the purpose of communication, a programming language's grammar still describes how syntactically correct sentences, or statements, are formed. A variable assignment, for example, is an exemplary sentence from a lot of programming languages. However, unlike natural languages, grammars for programming languages contain vocabulary definitions as well. The grammar of a higher programming language like Kotlin or Swift can usually be described as a **context-free grammar (CFG)** in accordance to the Chomsky Type 2 definition [Cho56]. CFGs can be defined by the tuple notation presented in Equation 2.1 and described as follows [Lee17] [Aho07].

$$G = (N, T, P, S) \tag{2.1}$$

Let G be a CFG. T represents the basic symbols in a programming language, like keywords, operators, etc., and is referred to as a set of terminals or tokens. N represents nonterminals or syntactic categories/variables, like expressions or `if`-statements. Being independent of the context in which nonterminals are used is fundamental to a CFG. S is the start symbol of the grammar and a special, designated nonterminal. P represents productions following $n \rightarrow \alpha$ where $n \in N$ and $\alpha \in \{ N \cup T \} *$. The non-terminal n represents a construct, while α represents the written form of that construct. According to the ISO/IEC 2382 standard, a programming language construct is the "syntactically allowable part of a program that may be formed from one or more lexical tokens in accordance with the rules of a programming language" [ISO15]. In the terminology of this thesis, a construct is therefore one possible way of implementing a programming language feature. Programming languages usually consist of various features, like different kinds of loops, or object-oriented features like classes.

According to language theory, a language consists of sentences that are valid combinations of its vocabulary [KVE94]. In context of a CFG, the complete language of a grammar is defined by the terminals that can be derived from the start symbol [Aho07]. This process of derivation is basically a step by step replacement of nonterminals by other nonterminals and/or terminals until only a terminal is left. In conclusion, if it is not possible to derive a string of terminals and/or nonterminals from the start symbol, it is not a syntactically correct sentence in the given language.

2.3.2 Lexical Analysis

Typically, the process of tokenizing the direct text input is decoupled from the parser in a separate program, commonly referred to as a lexer [Aho07]. The analysis performed by the lexer can be divided into two steps: Firstly, the input character stream is scanned and possibly unneeded characters such as comments and spaces are removed. Secondly, a token stream is formed based on the input, in a process called lexical analysis. In lexical analysis, a token is a logical unit represented by a pair. It consists of a token name, defined by the language, and an optional attribute value. A lexeme is a string of characters that matches the pattern of a token. The pattern of a lexeme itself can be described by a regular expression. Per definition, regular expressions are the algebraic description of a certain set of strings [Hop11]. They define regular languages, which are of Type 3 grammar, according to the Chomsky hierarchy [Cho56]. Terminals of a CFG can usually be formulated as regular expressions.

Three basic operations can be applied to languages defined by regular expressions. When considering L and M as two regular languages, the union of the two, formally noted as $L \cup M$, produces a set of strings that is in either language or both. Concatenation, either noted with a dot operator $(L.M)$ or no operator at all (LM), produces a set of strings by taking any string from L and following it with any string from M. Lastly, the Kleene closure from L, noted as $L*$, represents concatenating any string from L zero or more times.

According to the Kleene's theorem, regular expressions describe finite state automata [K+56], more precisely **non-determenistic finite automata (NFA)** and **determenistic finite automata (DFA)**. DFA are usually the preferred implementation in lexers, as their simulation of a regular expression containing a union is more straightforward [Aho07]. In

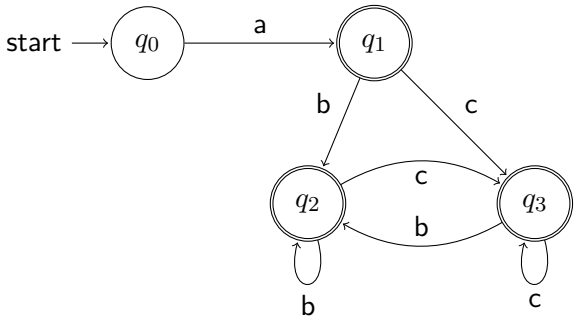


Figure 2.2: Graphical representation of a DFA implementing a regular expression.

practice, DFA are often converted from the easier to design NFA, which was proven possible by Rabin and Scott [RS59]. DFA can be defined by the following tuple notation presented in Equation 2.2 [Hop11].

$$A = (Q, \Sigma, \delta, q_0, F) \tag{2.2}$$

Let A be a DFA. Q is a finite set of states, while Σ represents a finite set of input symbols. The transition function δ is parameterized by a state and an input symbol, and ultimately returns a new state. The return value of δ marks the fundamental difference to a NFA, as they return a subset of Q instead. The start state is q_0 and F defines the set of accepting states belonging to Q. Figure 2.2 depicts the graphical representation of a DFA implementing the regular expression for a certain token. The DFA consists of the states q_0-q_3 with q_1-q_3 as accepting states. The regular expression describing a lexeme matching this token is `a(b|c)*`. Accordingly, a stream of symbols is accepted if an `a` is followed by any combination of zero or more `b` and/or `c`. Possible valid combinations are `a`, `abbb`, `acb`, `accbbc` and likewise.

2.3.3 Syntax Analysis

The validity of a token stream produced by the lexer is checked by the next phase, referred to as syntax analysis [O'R16]. It is implemented by a program universally referred to as a parser, although the term parser often refers to the whole architecture behind programming language parsing as well. In this chapter, however, this term explicitly refers to the program performing syntax analysis.

In case the input turns out to be not parsable, the parser returns a syntax error. In addition to checking a token stream's validity, it might create some kind of model for further processing, which is usually in the form of a parse tree. Parse trees are also alternatively referred to as syntax trees. Their tree-like structure represents the derivation of a sentence of a language $L(G)$ of a given grammar G. While its nodes construct the sentence, the root is defined by the start symbol S of G. Ultimately, a syntactically correct program can be represented by a parse tree. Figure 2.3 (see page 12) depicts the parse tree for the statement `(y + x) * 8` in an exemplary grammar $G = (N, T, P, E)$ provided by Lee [O'R16]. N is defined as $\{E,\ T,\ F\}$ and T as $\{identifier,\ number,\ +,\ -,\ *,\ /,\ (,\ )\}$. P consist of

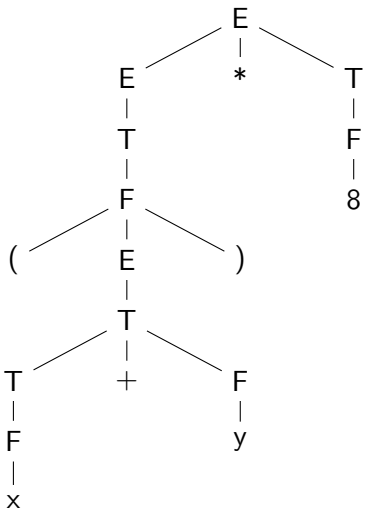


Figure 2.3: A parse tree example for a grammar provided by Lee [O'R16].

the production set defined in Equation 2.3.

$$\begin{aligned} &E \rightarrow E + T \mid E - T \mid T \\ &T \rightarrow T * F \mid T/F \mid F \\ &F \rightarrow (E) \mid identifier \mid number \end{aligned} \tag{2.3}$$

Parse trees are considered to be concrete if they represent a direct mapping of the grammar to the tree structure. Abstract syntax trees, commonly abbreviated with AST, however omit nodes unimportant for translation. Whether the parse tree the parser returns is concrete or abstract depends on the parser itself. Approaches to creating a parse tree can be generally divided into two categories; top-down parsers and bottom-up parsers.

Top-Down Parsers

Top-down parsers build the nodes of a parse trees from the root down by considering token after token [Aho07]. As they can be implemented as a set of mutually recursive functions, top-down parsers are sometimes called recursive descent parsers [O'R16]. Usually working on so called LL grammars, the parser scans the input from left to right and performs the left most derivation of that product. It is noteworthy, however, that recursive descent parsers can get stuck in an indefinite loop when discovering a production P where the leftmost symbol in α is the same as n. This phenomenon is called *left recursion*. In variations of the LL grammar, the decision which production is followed is based on looking a certain number of tokens ahead. A LL(1) parser for example considers one lookahead token, while a LL(k) parser considers an arbitrary number of k tokens.

Bottom-Up Parsers

Contrary to top-down parsers, the construction of a parse tree performed by a bottom-up parser starts at its lowest nodes and works up to the root [Aho07]. This process is generally performed by replacing substrings matching α of a production P by n of that production. This is implementable with a **pushdown automata (PDA)**, performing shift reduce parsing. Hence, those bottom-up parsers are occasionally referred to as shift reduce parsers [O'R16].

A PDA, which is a finite automaton with a stack, can be defined as shown in Equation 2.4 [Hop11].

$$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F) \tag{2.4}$$

Let P be a PDA. Q, Σ, q_0 and F are analogous to definitions given in Chapter 2.3.2 for a DFA. Γ represents a finite stack alphabet, which holds the set of symbols allowed to push onto the stack. The transition function δ takes three arguments: Firstly, q, a state in Q. Secondly, a, which is either an input symbol from Σ or an empty string, formally noted as ϵ. Lastly, the stack symbol X, which is a member of Σ. δ ultimately returns a finite set of pairs (p, γ), with p being the new state and γ being a string of stack symbols to replace X at the top of the stack. Finally, Z_0 denotes the start symbol, which is the initial content of the PDA's stack.

In the actual process of shift reduce parsing, tokens from the input buffer are shifted onto that stack. If the incoming token results in the stack forming a production P from the given grammar G partly or as whole, the part corresponding α is reduced to n at the top of the stack. Consequently, a program valid for a given grammar can be reduced up to S and results in an empty stack. Parsers implementing shift reduce parsing may be built on LR grammars, for which reduction is performed by scanning the input from left to right and executing the rightmost derivation of the product. Accordingly, for LR(k) grammars the decision which production to choose is based on k symbols of lookahead. In addition to the PDA, a LR parser built upon a LR grammar generally implements a parsing table, which size depends on the grammar and the number of lookahead tokens [Aho07]. Traditionally, the most used algorithm for a LR parser is LALR(1), as it covers the grammar of most programming languages [O'R16] while maintaining a sensible sized parsing table [Aho07]. The LA references to lookahead, which, in case of LALR(1), is one token.

2.3.4 Semantic Analysis

To verify if a program is executable, it needs to be both syntactically and semantically correct [Lee17]. The previously described syntactical checks are oblivious to how more logical aspects of the language work. Although the statement `a * b` might pass as syntactically valid, it might not be semantically valid, as the values behind the variables `a` and `b` are not intended to be part of a production in that particular programming language. As some semantic issues will only arise during runtime and can therefore be considered dynamic, a semantic analyzer program is considered to be concerned only with static semantic issues. Those issues may include type checking, scope and recognizing identifiers [Aho07].

The analysis is generally performed on the parse tree previously created by the parser, possibly in combination with a data structure called a symbol table. These tables keep

track of identifiers by noting information relevant to them, like their definition, location in the source code, typing and so on. Symbol tables can be constructed incrementally in the analysis phases by binding the identifier to the aforementioned values or by extracting that information from the parse tree. The semantic analysis may perform type conversion, referred to as coercion, on the existing parse tree as well. Ultimately, the verified and updated parse tree is the output of the semantic analyzer, ready for further processing. However, some language processors skip this phase entirely and operate only on the parse tree generated by the parser as it is.

2.3.5 Lexer/Parser Generators

While it is possible to implement the previously described lexer and parser models by hand, it is oftentimes impractical. In the case of lexers, handwritten lexers may offer more flexibility, but can become cumbersome to create and maintain, depending on the language. A lexer generator is a tool that dynamically implements finite state machines for lexical analysis, with lex [Les06] being a popular example for the C programming language. Lexer generators usually require a set of regular expressions representing the lexemes of all tokens.

At the same time, while top-down parsers are generally written by hand, the complexity behind the implementation of a bottom-up parser almost always results in the usage of a parser generator. For building the parser, the generator requires the language's grammar. Oftentimes, parser generators take care of lexer generation as well. Popular parser generators for CFGs include Yacc [Les06], using the LALR(1) algorithm for parsing, JavaCC [Jav22], using LL(1) and occasionally LL(k) and ANTLR4 [Par22], using the Adaptive LL(*) algorithm [PHF14].

ANTLR

ANother Tool for Language Recognition (ANTLR) is a flexible parser generator tool, capable of reading, processing, executing and translating structured text or binary files when given an input grammar meeting its requirements [Par22, Par12]. ANTLR's flexibility allows it to be applied in a wide area of projects, ranging from legacy code converters to information extractors for texts in natural languages. Large projects using ANTLR include Twitter's query parsing or NetBeans C++ parser. The latest installment, ANTLR4, was released in 2013 as a complete rewrite from older versions featuring the new parsing mechanism Adaptive LL(*), or ALL(*) for short. For the remainder of this thesis, ANTLR4 will simply be referred to as ANTLR.

ANTLR accepts any CFGs that do not include indirect left-recursion, which describe rules calling themselves through another rule, and/or hidden left-recursion, which occurs when an empty production results in left-recursion. Grammar rules are defined in extended Backus–Naur form, combining both lexical and syntactical rules. Listing 1 (see page 15) illustrates this notation style with a simple example, which defines a parser rule for concatenating any string of numbers from 0 to 9 with a plus sign at least one time. In addition, the grammar can be enriched by two ANTLR-specific constructs, written in the output parser's host language. Firstly, side-effecting actions, officially called mutators, can be used for infor-

Listing 1 ANTLR4 grammar example.

```
grammar Foo; // generates class FooParser

// parser rules
sum: INT (PLUS INT)* ;

// lexer rules
INT :   [0-9]+ ;
PLUS:   '+' ;
```

mation extraction or other similar use cases. Secondly, a production's semantic viability can be dictated by semantic predicates, which are side effect free Boolean-valued expressions. This opens the possibility of parsing grammars which do not meet ANTLR's initial requirements. As output, ANTLR generates both a lexer and a recursive-descent parser with Java, C#, Python 2 and 3, JavaScript, Go, C++, Swift, PHP and Dart as currently supported target languages.

The parser generated by ANTLR operates on the ALL(*) algorithm that moves grammar analysis to parse time [PHF14]. In case of multiple production options, an ALL(*) parser dynamically builds pseudo-parallel sub-parsers that use DFAs exploring each option. Ultimately, the surviving parser is chosen. If multiple parsers should survive in case of grammar ambiguity, the first alternative defined in the grammar is chosen. The previously described process illuminates the meaning behind the * in ALL(*), as the entire remaining input has to be considered for the decision-making in a worst-case scenario. According to Parr et al. this normally does not occur for common languages, and an overall linear complexity can be observed instead [PHF14]. Although ALL(*) itself does not support left recursion, ANTLR makes this possible by internally rewriting the rules prior to the parser generation.

Running the parser generated by ANTLR on an input text results in a parse tree output, accessible through specific classes provided by the ANTLR framework. Processing the information represented by the tree is facilitated by two tree walking mechanisms. The `ParseTreeListener` interface can be implemented for performing a depth-first walk on a specific node, while an implementation of `ParseTreeVisitor` allows explicitly visiting the nodes of the tree while it is walked.

2.4 Transpilers

A transpiler is a language processor program capable of automatically translating between programming languages of the same complexity, usually high level programming languages [KCH15]. This differentiates them from compilers, that translate high level programming code, which is usually written and understandable by humans, to machine code, which is difficult to understand by humans but efficiently to process by a machine.

Transpilers are used for a variety of purposes and have been around for years. As early as 1980, Albrecht et al. describe translating between compatible subsets of the programming languages Ada and Pascal [AGG+80]. Transpilers are also referred to as transcompilers [HSKY19] or source-to-source compilers/translators [KCH15, AGG+80] in the literature, but for the remainder of this thesis, only the term transpiler will be used.

The translation process from the input source code to the target output code is built upon the phases described in Chapter 2.3. After recognizing language constructs through parse tree traversal, the output is dynamically generated. Although many programming languages can be categorized into groups that follow the same core concepts and even use similar keywords and syntax, addressing all differences may prove difficult and too complex, if not entirely impossible. Nonetheless, complete language coverage is oftentimes not absolutely necessary. Although many programming languages do contain complex and therefore difficult to translate language constructs, those may be rarely used by programmers in practice [Yel88].

By reducing effort and human-made errors, transpilers are generally suitable for code migration in the world of software development [Yel88]. Next to translating from one particular source language to one particular target language, a transpiler may be capable of generating output code from different input languages. The .NET platform [Mic22a], for example, can handle C#, F# and Visual Basic as input and translates them to native output code. Translating between two versions of the same language is also a practical scenario, implemented for example by Babel [Bab22] creating ES5 conform JavaScript out of ES6. The extensibility of a language can be improved by transpilers as well, as shown by Sass [Sas22] or Typescript [Mic22b]. With the increasing importance of web applications that are subjects to data restrictions set by the internet connection, source code optimization becomes a very relevant topic for transpilers. Google Closure [Goo22c] or UglifyJS [Baz22], for example, reduce the file size of JavaScript by stripping the code of unneeded characters. However, the output of the aforementioned tools is aimed towards workability rather than readability, as they are used in a frequent translation cycles. Hence, the output code is not designed for being manually and independently developed any further. Nevertheless, projects focusing on the maintainability aspect of the output source code exist. Huijsman et al. developed a transpiler for a permanent Algol 60 to Ada migration, focusing on readability [HvKPT87]. Schaub and Malloy had the same goal for translating a Java subset into Python and C++ [SM16]. Recently, in the field of translating between native mobile programming languages, efforts to achieve independently maintainable output source code have been made as well [Sch22, Oll20, Stu20, Ven22a]. As they're particularly relevant to this thesis, they will be further explored in Chapter 3.1.

2.5 Summary

This chapter gave background on Kotlin and Swift, the core concepts of programming language parsing and transpilers. Kotlin and Swift are modern programming languages that have replaced their predecessors Java and ObjectiveC as the preferred languages for the Android and Apple platforms, respectively. In particular, they are characterized by their

less verbose and safer syntax. Next to app development, both languages can be used for a diverse set of projects, like server-side applications or systems programming.

However, while Kotlin and Swift as high level programming languages are suited to be understood by humans, their syntax cannot be easily processed by a machine. For converting the instructions to a processable model, programming language parsing becomes necessary. Three steps describe this process, although the last step may be omitted occasionally [Aho07].

Firstly, a lexer is used to for transforming the incoming character stream to a token stream. The tokens of a programming language are defined in its grammar, along with a set of rules defining how these tokens can be used to form syntactically valid sentences, i.e., statements. The grammar of a higher programming language can usually be described as a CFG in accordance to the Chomsky Type 2 definition [Cho56]. Tokens are identified from the input by matching lexemes, which are strings of characters corresponding to the pattern of a token. In this context, these patterns can be usually described by regular expressions, which can be implemented by using DFA.

Secondly, the output token stream from the lexer is picked up by a parser checking its syntactical validity. If a string cannot be completely derived from a special token known as a start symbol, it is no valid sentence in the given language. The output of a parser is usually a tree-like structure called parse tree, representing the derivation of the input token stream from the start symbol of the language. Generally, parsers are divided into the two categories of top-down and bottom-up parsers, working on LL or LR grammars respectively. While top-down parsers build the nodes of the parse tree from the root down, bottom-up parsers start at the lowest nodes and work their way up to the root. Both types might incorporate an arbitrary number of lookahead tokens to determine the rule of the grammar that is being followed by the input.

A potential last step, that is omitted by some language processors in favor of working directly on the parse tree produced by the parser, is the semantic analysis. Not all rules of the language can be defined in its grammar, as some of them do not arise before runtime. E.g., the statement `a * b` might pass as syntactically valid while not being semantically valid, as the typing of the values behind the variables `a` and `b` are not intended to be part of a production in that particular programming language. Therefore, in the semantic analysis phase, the parse tree is checked on its semantic validity conforming to rules w.r.t. type checking, scopes and recognizing identifiers. Usually, this involves a data structure called a symbol table, keeping track of identifiers and information relevant to them. Ultimately, the output of the semantic analysis phase is a verified parse tree.

To reduce development effort, several lexer and parser generators exist. One of them is ANTLR [Par22, Par12], a flexible and customizable parser generator tool for creating a lexer, a parser, and various helper classes for traversing the parse tree. ANTLR was also used for this thesis.

Transpilers are special language processor programs that usually translate between high level programming languages [KCH15]. While being suited for code migration, they are sometimes confronted with fundamental differences between their input and output language [Yel88]. However, complex and difficult to translate language constructs might be rarely used by programmers in practice. In the modern landscape, transpilers are, e.g., used for cross-

platform migration, legacy support, extension of a language or code optimization. However, transpilers do not always focus on the maintainability and the readability of the output code, but instead on its workability. Nevertheless, especially in cross-platform migration, it might be desirable to create an independently maintainable output code base. The Swift-to-Kotlin transpilers Gryphon [VGM20] and SwiftKotlin [Oll20], the Kotlin-to-Swift transpiler Kotlift [Stu20] and the bidirectional Swift and Kotlin transpiler SequalsK [Sch21] aim to produce readable output code that can be developed further without being dependent on the transpiler itself. As a result, they open up opportunities for cross-platform Android and Apple OS development. The current state of the art in regard to transpilers in this field is therefore discussed, among others, in the next chapter.

Chapter 3

State of the Art

This chapter presents the current state-of-the-art relevant to different aspects of this thesis. Towards the end of the previous chapter, Gryphon, Kotlift, SequalsK and SwiftKotlin were briefly introduced as transpilers for cross-platform Android and Apple OS development. Building on this, native source-to-source translators intended for that purpose, including the aforementioned four transpilers, are described further in the following. Part of this thesis includes uncovering deficits in Gryphon's, Kotlift's, SequalsK's and SwiftKotlin's language coverage. To understand the differences between Kotlin and Swift and potential obstacles for these transpilers better, previous work on the translatability between those languages is briefly discussed afterwards. Since the impact of deficits in language coverage might vary depending on their actual degree of usage in programming projects, existing studies conducted on Kotlin and Swift usage are examined next. Lastly, this thesis is placed into the context of other works using ANTLR.

On the one hand, this chapter aims to provide an overview of more concrete topics regarding this thesis and show previous work that it is based upon. On the other hand, it reveals why the current state of the literature is insufficient for evaluating Kotlift, SequalsK and SwiftKotlin, thus making the study conducted in this thesis necessary.

3.1 Native Source-to-Source Translation in the Context of Android and Apple OS Cross-Platform Development

The interest in translating between Android and Apple OS apps as described in Chapter 1 created varying approaches over the last years, including tools converting native code to native code. Notably, the tools included in this section range from more traditional, fully automated transpilers that operate as described in Chapter 2 to tools that perform translation, but still require some degree of manual verification. However, they all share common themes behind their motivation. The goal of these tools is usually to create an independently maintainable, native output code base that does not rely on the continuous development of a certain third-party-tool. Furthermore, some of them aim to decrease the necessity of learning the native language of the other platform. Ultimately, like all cross-

platform development tools, they want to reduce the development effort for targeting both platforms.

However, little literature exists considering the tools presented in this section when comparing other means of cross-platform development, which may be related to the novelty of the native source-to-source translation approach in this area. Therefore, the only evaluation of the tools available in the literature is often conducted within the works introducing them. Still, those results certainly show promise and make further examination of the native source-to-source translation approach interesting.

3.1.1 Java-to-ObjectiveC and ObjectiveC-to-Java

Pioneering the transpiler approach for Android's and Apple's natively supported languages, the J2ObjC transpiler from Google was started in 2012, with version 1.0 released in 2016 to the public [Goo21b]. The transpiler officially concentrates on the translation of client-side business logic code from Java to ObjectiveC. For the code conversion, it operates on an abstract syntax tree created from parsing the input. Due to the complexity of platform specific **application programming interfaces (APIs)**, J2ObjC only supports business logic code [Goo16]. J2ObjC is open-sourced and internally used by Google apps like Gmail, Google Docs or Google Drive. In addition, it is successfully incorporated in various works in the literature regarding cross-platform Android and Apple OS development [CLTC15, FW16]. Although J2ObjC aims for "generally easy to debug source output" [Goo16], its output code is noted as difficult to read by the literature, as the transpiler prioritizes working code over readable code [VGM20]. In their comparison of cross-platform development tools for mobile platforms, the advantages for J2ObjC and transpilers in general were stated as the reusability of code and the possibility of creating a native app from the output [EKAYW17].

A counterpart for translating ObjectiveC to Java code called objc2j exists, but is deprecated [Goo13].

3.1.2 Java-to-Swift and Swift-to-Java

Meanwhile, j2swift [Nie16], converts from Java to Swift by also analyzing the parse tree of the input. Nevertheless, the project itself appears to be largely abandoned, with the last commit six years ago and the supported language versions being Java 8 and Swift 1.2.

As part of their j2sInferer-tool for facilitating the porting of mobile applications, An et al. included code translation from Java to Swift [AMT18]. This is done by creating a parse tree from the Java input code and matching the lowest subtree representing the code block of interest to the entries of a database consisting of abstract string template mappings. Thus, the Swift output code is generated from an appropriate match by replacing the abstract parameters with concrete substrings based on the input. The authors evaluated j2sInferer by testing it on four applications that had both a Java and a Swift implementation. Ultimately, they obtained an average translation accuracy of 76%, which outperformed j2swift that was used on the same applications and obtained an average translation accuracy of only 57%.

Works surrounding the **Trans Compiler Android to IOS Conversion (TCAIOSC)**-project aim to produce a whole independently maintainable app project for the respective

other platform by transpiling from Java to Swift [SHK+19] and Swift to Java [MME+20]. As a proof of concept for their Android to iOS transpiler, they fully converted a simple Android app for solving a second degree polynomial to an iOS app [SHK+19]. The code translation from Swift to Java was evaluated with a **Bilingual Evaluation Understudy (BLEU)** score [PRWZ02], a metric for evaluating machine generated translation in comparison to human translation [MME+20]. Their test cases consisted of an app specifically developed for that evaluation, in addition to four other apps. The results show an average of 88% BLEU score of the translatable parts. Next to the business logic code, approaches for converting UI code and other parts of the app leveraging native APIs were presented in subsequent work [HSKY19]. The conversion is mostly done by, once again, leveraging the parse tree of the input code and recognizing the usage of platform specific APIs, like sensor usage. Likewise, views from XML layout files are mapped to corresponding code blocks from the input, so the layout file for the other platform can be generated. These approaches were enhanced further for the Swift-to-Java translation direction, as described in a follow-up paper, by generic library mapping to prevent the need for manually defining API specific translations inside the transpiler [MSSY21]. This is achieved by overriding the parser output based on JSON-files that describe the mapping of a function to the corresponding other language. Those JSON-Files are previously generated as output of a separate tool that semiautomatically finds the equivalent of a library function in another language. Their methodology was evaluated using BLEU. After previously mapping 91 functions of Swift's and Java's math library and 90 functions of their string library, the improved system leveraging library mapping achieved an average accuracy of 91.36% when translating the test cases from Ahmad et al. [MME+20]. In another work, TCAIOSC was proposed to support the translation of SwiftUI to Android UI classes and functions from the `android.widget` library [Goo22a] and likewise [EKSY21]. For this purpose, every considered UI component was mapped to its corresponding counterpart in a JSON-file. When using the extended tool based on the version of Ahmad et al. [MME+20, MSSY21] on five example applications, an average accuracy of 89% was achieved when performing a BLEU test. However, the conversion presented in this work did not include the generation of XML layout files.

While not exactly being a transpiler in the traditional sense, Native-2-Native still performs Java to Swift translation by leveraging popular online resources such as the popular Q&A site Stack Overflow [Sta22a] in an "automated code synthesis" [CBTR17, BCT15]. In practice, suitable translations are suggested via an Eclipse plugin while editing the source code. On the one hand, the selection requires a considerable amount of manual effort. On the other hand, native APIs that are not included in the actual grammar of the language can be easily covered. Similar to the transpilers presented thus far, Native-to-Native tokenizes the input Java code to generalize it for creating a search query and a meta-data object. Then, it tokenizes the Swift code of relevant results into meta-data objects, too, so they can be matched to the Java meta-data object. As a proof of concept, Native-2-Native was evaluated by its capability of translating 66 API functions for each direction of translation, including communication with sensors, network interfaces and canonical library classes and data structures [CBTR17]. Ultimately, 85% of these test cases retrieved at least one relevant answer when converting Java to Swift and 92% when converting Swift to Java.

3.1.3 Kotlin-to-Swift and Swift-to-Kotlin

For translating between Kotlin and Swift, the primary programming languages of Android and Apple OS, the four transpiler projects Gryphon, Kotlift, SequalsK and SwiftKotlin can be found when searching online [Ven22a, Stu20, Sch21, Oll20]. Notably, all of these tools currently only focus on the model with the business logic parts of an application.

Gryphon supports the translation of Swift to Kotlin while laying a heavy focus on both the independence and the manual maintainability of the output code [VGM20, Ven22a]. Next to a command line tool, Gryphon is complemented by an Xcode-plugin. In addition to the core transpiler, Gryphon provides two separate libraries for both Swift and Kotlin. Thereby, classes and functions defined there, e.g., Gryphon's own implementations of array and map types, can be used to improve the workability of the transpiler. Furthermore, manual translation can be provided inside the Swift input code. The authors evaluated their tool with five example applications from the Computer Languages Benchmark Catalog [FG22] and by bootstrapping Gryphon, i.e., translating the transpiler to Kotlin by using Gryphon itself. After collecting benchmark runtimes for both manually written and transpiled versions of the sample applications, they found the differences in the Gryphon versions to be acceptable, as they ranged from 0.14% speedup to 3.01% slowdown. According to Gryphon's official GitHub page, it currently claims to support Swift 5.2 to 5.5 [Ven22b].

Notably, no designated work for Kotlift exists in the literature. However, according to the official GitHub page of the project [Stu20], the code conversion is achieved by working with regular expressions and a simplified tree representing the structure of the code. In addition to the transpiler itself, Kotlift provides a file mapping the most basic Kotlin specific data type functions from Kotlin's and Swift's standard libraries. However, this file is outdated and cannot be used for current versions of Swift. The transpiler comes with two other options for manual extensions. Firstly, the JSON-file describing the translation mappings can be extended with custom translations. Secondly, custom rewrites can be set in code by using a certain comment structure. Currently, Kotlift claims support for Kotlin 1.0.1 and Swift 2.2 and can be called as a command line tool.

SequalsK is the implementation of a bidirectional Kotlin and Swift transpiler, working on the parse tree generated from the input code [Sch21, Sch22]. The translation process is focused mainly on aspects of the language that are defined in its grammar. However, when part of one language's API was part of the actual grammar of the other language, that API feature was considered still. The transpiler is accompanied by two support files for the Kotlin and Swift output, respectively. Those do not only provide type extensions to support the workability of the output code, but encompass the functionality for supporting a loss free re-translation to the source language, e.g., by defining annotations. As a proof of concept, the model part of a board game app written in Swift was translated to Kotlin, another game was added to the app and the Kotlin model was translated back to Swift. While the Swift to Kotlin translation produced 100% valid Kotlin output code, translating back to Swift required further optimizations of the transpiler to be translated 100% correctly. Those changes included 11% of the lines of code of the final transpiler. As the model part of the app translated from Swift to Kotlin made up 86% of the whole app project before adding the other game, the author observes a saved effort of 86% for this particular case

study by using SequalsK instead of manually migrating the board game app. SequalsK is available as a command line tool, a web tool and an Android plugin.

Like for Kotlift, no designated work about SwiftKotlin exists in the literature. Nevertheless, according to its official GitHub page [Oll20], the translation is performed by analyzing an abstract syntax tree. The author disregards the translation of parts of the language outside its grammar, like functions for string types, and encourages manual editing in these cases. SwiftKotlin can be used as a command line tool or a Mac application.

While their approaches seem promising, all authors admit to deficits in language coverage that are only partly documented. Given the complexity of Kotlin and Swift, this may be rooted in the effort required to cover them fully. In the preliminary work for this thesis, some of those shortcomings were uncovered when testing Gryphon, Kotlift and SequalsK on a set of basic constructs that was derived from the overview chapters of the Kotlin and Swift documentations [SS22]. Those included constructs for simple values and typing, strings, collection types, classes, functions and so on. While Kotlift was identified as less mature due to only supporting 55% of the test cases, Gryphon and SequalsK both achieved good support of the test cases, with Gryphon supporting 73% and SequalsK supporting 74% when translating Swift to Kotlin and 78% when translating Kotlin to Swift. Nevertheless, some obstacles were observed when translating between the languages, like constructs that have no counterpart in the other language, are difficult to identify and translate or make use of data type specific functions.

3.2 Translatability between Kotlin and Swift

The transpilers evaluated in this thesis work on the claim that Kotlin and Swift share similarities to an extent that makes translating between them possible. When comparing the two languages, they do share common traits at a first glance. Both are general-purpose, statically typed languages with aspects of both object-oriented and functional programming. Furthermore, they are strongly typed with safe handling of nullable values. Some developers even point out that learning Kotlin is easier when already having experience in Swift [OTE20].

Multiple articles online explore their similarities further by directly comparing the two side to side [Oll16, Mec17]. These similarities are especially visible on a surface level for constructs that can be either left as they are or only require a simple search and replace. At the same time, more gravitating differences become visible when approaching more complex features of the languages, as it was uncovered in the experiments of this thesis' preliminary work [SS22]. E.g., unlike Kotlin, Swift supports associated values for its enum type, so a workaround has to be used for Kotlin. At the same time, Kotlin offers the smart casting feature that makes explicit casting of a variable unnecessary if the typing of that variable has already been checked previously in the same scope. However, Swift insists on explicit casting in these cases. Consequently, simply translating from Kotlin to Swift without considering this difference would lead to a compiler error in Swift.

When those differences remain unaddressed by the transpiler, their workability is put at risk. At the same time, as constructs vary in their complexity, they also vary in the

Listing 2 Code examples for Schultes' four categories regarding translation complexity [Sch21].

```
/* Category 1 */
var a = 123 // Kotlin
var a = 123 // Swift

/* Category 2 */
val b = "Hello World!" // Kotlin
let b = "Hello World!" // Swift

/* Category 3 */
class Rectangle(val height: Double, val width: Double) // Kotlin
class Rectangle { // Swift
    let height: Double
    let width: Double
    init(height: Double, width: Double) {
        self.height = height
        self.width = width
    }
}
```

required effort for implementing their support. Hereby, Schultes differentiates between four categories [Sch21]. The three categories that are translatable are exemplified by concrete code examples in Listing 2. The first category includes identical keywords and constructs, like the keyword for starting a variable declaration. Meanwhile, the second category only requires a simple search and replace. A construct exemplifying this category is the declaration of a constant. The translation of the third category of language constructs is still possible, but in need of further transformation of the output code. E.g., Kotlin offers the possibility of describing a class' assignable properties in a so-called primary constructor that is declared directly after the class' identifier. Swift only supports a constructor placed in the class' body, so the properties declared have to be added to the class' body and Swift's class' constructor accordingly. Lastly, constructs of the fourth category are simply not translatable from one language to the other. These include, e.g., *Self Mutating Extensions* in Swift that are not allowed in Kotlin.

In conclusion, while the literature offers no thorough analysis of the common features and fundamental differences of Kotlin and Swift, previous works suggests that translating between them is generally achievable with exception to fundamental differences. What aspects of the language a transpiler needs to focus on in particular to be widely usable can be assessed by examining how the language is used.

3.3 Studies on the Language Adoption of Kotlin

The extent to which Kotlin is being adopted by Android developers and the problems encountered in various aspects of its use was the subject of Oliveira's et al. research [OTE20]. Their paper provides a more general overview on language usage and problems developers encounter. Next to general problems with the Android platform, their paper examines the Java-Kotlin interoperability and how intuitively functional paradigm aspects can be integrated with Kotlin. Furthermore, the authors evaluate developers' experiences when working with popular Kotlin **integrated development environments (IDEs)** like Android Studio and how developers use the language. For this purpose, automatically discovered topics from Stack Overflow that are grouped together with the **Latent Dirichlet Allocation (LDA)** algorithm [BNJ03], are cross-validated by semi-structured interviews with seven Android developers. The most questioned topics on Stack Overflow included general questions about the language, Java-Kotlin interoperability and language specifics like the correct definition of functions, properties and so on. Questions regarding the UI, the compilation process, integration of Google or Android components like Firebase, multithreading and dealing with data types and structures, including questions on data types like string, were grouped into mediumly asked questions. Finally, connectivity issues, multimedia handling, data storage, dependency injection and testing belonged into the lowest question group. Although the Java-Kotlin interoperability is stated as an advantage, they identified that concepts not available in Java can lead to problems. Moreover, none of the interviewees described a fully migrated code base from Java to Kotlin in their current projects. Next to comprehension issues when integrating functional paradigms in Kotlin, interviewees brought to attention that those paradigms can make code harder to read. Although being the official Android IDE, Android Studio poses a risk to the workability of projects, as minor upgrades can lead to problems. Lastly, in addition to finding out that developers think of Kotlin as improving productivity and quality of the source code, the interviews confirmed that the similarities with Swift help adopting Kotlin.

Mateus and Martinez concentrate their study on actual feature adoption by measuring the occurrence of certain constructs [MM20]. They examined whether those constructs were actually adopted, the degree of their adoption, when they were introduced to a project and how their usage evolved over time. Their dataset consisted of 387 open-source Android applications from a list they curated in a previous study [MM19]. Constructs of interest to their study are available in Kotlin, but not in Java, like *Type Inference*, *Smart Casts*, *Sealed Classes* and so on. Their occurrence in the applications' source code was detected by a custom tool searching for a representation of the construct in an abstract syntax tree provided by the Kotlin compiler API. For each construct, the number of applications containing at least one instance was determined, in addition to the total number of instances in the last commit on the corresponding Git repository. As their sample pool projects varied in size, the extracted numbers were normalized. This was done using either the lines of code of an application or a metric relevant to the construct, e.g., the relative occurrence of a *Data Class* to normal class definitions. Furthermore, the first use of a construct in the repository tree was noted. Their findings show that the most used feature is *Type Inference*, with 98% of the applications making use of that feature. 77% of variable declarations did not have an

explicitly declared type. *Lambdas* were found in 95% of the applications, while *Safe Calls* existed in 89%. Other popular constructs included the `when` control structure, *Unsafe Calls*, *Companion Objects* and *String Templates*. Meanwhile, *Type Alias*, *Super Delegation*, *Infix Functions*, *Inline Classes* and *Contracts* were found in less than 20% of the applications. While the least used constructs were generally added later in the lifespan of a project, the general usage of a construct tends to grow with the project's evolution.

As part of a study on Kotlin construct diffusion and adoption in Android, Zayat also took an automated approach to evaluate language feature usage [Zay20]. Their dataset of 33,267 open-source applications was retrieved via a custom Java application connecting to the GitHub API [Git22a] and filtering for projects containing the *AndroidManifest.xml* file as well as the `setContentView` method. The repositories had to be last updated in October 2017, the month and year Kotlin was officially supported as a language for Android. However, limitations of the GitHub API returning only 1000 results per search resulted in a total of 26 hours to get all Android projects using the Kotlin language. The constructs considered, namely *Data Classes*, *Nullability*, *Mandatory Casts* and *Argument Lists* were mapped to regular expressions and identified with the rigrep search tool [Gal22]. The results illustrate the percentage of applications containing a certain feature at least once. On the one hand, *Unsafe Casting* by making use of the `as` operator, exists in 71% of the applications. On the other hand, *Safe Casting* with `as?` is only found in 11% of the applications. Defining a *Default Value for an Optional* is found in 54.7% of the applications. The `vararg` keyword working as a placeholder for an argument list in Kotlin is used in 15% of the applications. The following results slightly differ from the observations by Mateus and Martinez. However, *Safe Calls* are, once again, identified as popular with 70.4% of the projects using them at least once. Meanwhile, *Unsafe Calls* with the null assertion operator `!!` were found in only 56.5% of the applications, opposing Mateus and Martinez result of 87.6%. *Data Classes* show a little lower occurrence, with 49% of applications using them, opposed to 65.1% in Mateus' and Martinez' study.

3.4 Studies on the Language Adoption of Swift

With Swift being pushed by Apple to eventually replace ObjectiveC, Rebouças et al. examined how developers deal with that transition [RPE+16]. They took a closer look at Swift's error handling mechanisms and *Optionals*, as those greatly differ from ObjectiveC. Despite not involving actual source code analyzation, this study provides a general overview on the experience of developing with Swift and what features are interesting to developers. The first, quantitative part of their methodology consisted of mining posts from Stack Overflow that are associated with Swift. Afterwards, those are automatically summarized into 25 topics with LDA. To validate those results with a qualitative study, they conducted 12 semi-structured interviews, with half of the interviewees claiming familiarity and the other half strong familiarity with Swift. Their results show that, while the core language seems to be easily adoptable, most of the questions on Stack Overflow center around libraries and frameworks used in combination with Swift. Dealing with *Optionals*, especially correctly handling *Optionals* that are introduced by frameworks, proofs to be a popular question topic compared

to other Swift features that do not exist in ObjectiveC. As the error handling mechanisms were newly introduced at the time of the paper, migrating from ObjectiveC to Swift and the difference between their approaches appeared to be a problem for some developers as well. As Swift was just a year old by the time of their study, the authors admit that deficiencies with the Swift tool set, like the Swift compiler that received its fair share of critic in both the online questions and the interviews, might improve over time. Nevertheless, to the best knowledge of the author of this thesis, no study exists with updates on how challenges for developers with regard to the Swift programming language as a whole might have changed.

However, Casse et al. further examined how developers use Swift's error handling mechanisms [CPCS18]. Their study centers around three research questions. Firstly, the degree of adopting recommendations provided by six popular online guides, including the official Swift documentation. Secondly, the occurrence of well-known antipatterns and thirdly, the differences between experienced and novice developers when writing error handling code. The quantitative aspects of their methodology consisted of mining open-source Swift repositories using GHTorrent [Gou13]. With the assumption that 50% of the projects found by GHTorrent included error handling, they retrieved a sample size of 9,000 projects. To exclude personal or naive repositories, they applied Reaper, a classification framework for identifying projects serving a general purpose [MKCN17]. For extracting syntactical information about error handling mechanisms, a custom tool, referred to as metric extractor, was developed. A parse tree representation of the code for the metric extractor to work on was generated by swift-ast [yan19]. The popular Swift dependency managers CacaoPods and Carthage were excluded from the analyzation process. Ultimately, 2,733 projects containing error handling mechanisms, including variants of `try` and `do`, types that implement the `Error` protocol and error throwing and catching declarations were extracted by their tool. Those results were complemented by a qualitative study that included interviewing four experienced and six novice Swift developers, as well as manually analyzing commits pushed by expert and novice Swift developers. Their findings exhibit that only half of the projects considered even use Swift's error handling mechanisms to consume or signal errors. While less than a third only consumes, the remaining fifth both consumes and signals errors at least once. `Try` is by far the most used error handling construct examined, although it and its variants are only used more than five times in little over a half of the error handling projects. Additionally, custom error types, e.g., with an `Enum`, are defined sparsely. In light of the interviews conducted, this is attributed to the counter-intuitiveness to novices, practical limitations, problems with the compiler as well as a lacking documentation of Swift's error handling mechanisms. Swallowing or ignoring errors with `try?` or leaving an empty generic `catch` block are among the antipatterns most observed.

In order to gain more insight into code smell occurrences in iOS applications, Rahkema and Pfahl also took an automated approach to code smell detection [RP20]. For the detection of the 36 code smells they identified from reputable sources in the literature, they generated a model of the code and entered it into a graph database, using their custom tool GraphifySwift. Notably, only application source code was considered for their study, not imported libraries. The code smells were defined as queries. The 273 sample applications stemmed from a collaborative list of open-source iOS applications from GitHub [dkh22]. Their findings show, that the most common code smells are *Lazy Class*, *Long Method*,

Message Chain, *Ignoring Low Memory Warning* and *Data Class*.

3.5 Works with ANTLR

ANTLR is used as a tool in the literature in an array of works surrounding language processing. Evidently, some tools presented in 3.1 work on a parse tree generated with the help of ANTLR [Nie16, AMT18, SHK+19, MME+20, Sch21]. Naturally, building a translator based on ANTLR is not exclusive for translating the native languages of Android and Apple OS. Examples from other areas of computer science include porting legacy software [SS21], translating between very purpose-specific programming languages, e.g., for sound generation and manipulation [Dem15] or translating SQL queries to the query languages of NoSQL systems [RLMW14]. This flexibility of ANTLR is achieved by its dynamic generation of the lexer and parser based on a grammar file. So theoretically, any language that is described by a grammar file valid to ANTLR can be processed. This allows integrating ANTLR into language independent frameworks that can be adapted for concrete use, e.g., for static code analysis [RBS13]. Works using ANTLR for code analysis through leveraging the parse tree is related the closest to the intended usage of ANTLR for this thesis. Other examples from the literature include the identification of code clones [AP21, SYCI18], proposing code recommendations based on recognized constructs [LYB+19], or extracting a set of metrics evaluating an implementation of a Smart Contract [APT20].

3.6 Summary

This chapter introduced the reader to the current state of the art regarding the topics relevant to this thesis.

Evidently, transpilers and similar tools for native source-to-source translation can be found in the literature that attempt to overcome the drawbacks of other cross-platform tools for Android and Apple OS. While J2ObjC pioneered this approach in that particular field by translating from Java to ObjectiveC, multiple projects can also be found for translating between Java and Swift and Kotlin and Swift. Although most of these projects show promise, achieving good results for translating between the languages and considering many aspects of migration between Android and Apple OS, they are usually not considered in works comparing different means of cross-platform development. Therefore, the only evaluation of these tools is most often within the works presenting them, due to a **deficiency of coverage regarding native source-to-source translation for mobile cross-platform development in the literature**. Thus, this thesis aims to improve upon this situation by evaluating Kotlin-to-Swift and Swift-to-Kotlin transpilers, as those languages are officially preferred for native development by Google and Android. The transpilers considered for this thesis, namely Gryphon, Kotlift, SequalsK and SwiftKotlin currently do not translate parts of an app outside the pure language and some standard libraries. Regarding the need for post-translation edits, the transpilers differ in their demands. While Gryphon and SequalsK aim for little manual editing of the output and value meeting the stylistic standards of the output

language, Kotlift and SwiftKotlin are less ambitious. However, all of them admit to **deficits in language coverage**.

In any case, for successfully translating between programming languages, a certain degree of similarity between those languages must exist. While Kotlin and Swift are very syntactically similar on a superficial level [Oll16, Mec17], differences that are more difficult to overcome exist, in addition to features that have no counterpart in the other language. Generally, the constructs of both languages can be divided into four categories regarding their translatability, namely **identical**, **replaceable**, **adaptable with a certain amount of effort** and **not translatable** [Sch21].

The impact of a transpiler's deficits and capabilities regarding language coverage can be better evaluated by looking at the way the language is adopted in practice. Thus far, little work on both Kotlin and Swift exists in this regard. Although some findings were already made that are also relevant to this thesis, only a limited selection of constructs was considered in related work. Nevertheless, some insights regarding certain constructs were given. E.g., *Safe Calls* were identified as popular constructs in Kotlin [Zay20, MM20]. W.r.t. Swift, findings showed that approximately 50% of the test applications did not use Swift's error handling mechanisms [CPCS18].

In conclusion, the contribution of this thesis can be summarized as follows: Although concepts regarding native source-to-source translation, especially by transpilers, appear to be an up-and-coming approach to cross-platform development for Android and Apple OS, they are hardly considered by the literature. Therefore, this thesis aims to built upon the preliminary work evaluating the transpilers Gryphon, Kotlift and SequalsK [SS22]. This subcategory of transpilers was chosen because Kotlin and Swift are the primary native programming languages of Android and Apple OS, respectively. To the best knowledge of the author, no other work directly comparing Kotlin-to-Swift and/or Swift-to-Kotlin transpilers exists in the literature, besides the preliminary work. For an even better insight on the current state-of-the art regarding those kinds of transpilers, the Swift-to-Kotlin transpiler SwiftKotlin is evaluated next to the three transpilers chosen for the preliminary work. Furthermore, as proposed there, the transpilers should be evaluated in a more practical fashion by considering the popularity of unsupported constructs. However, since there is insufficient data in the literature regarding the considered constructs' popularity, the necessity for conducting such a study for this thesis arises. For that study, parse trees created from the source code of a project sample pool should be analyzed, like it was done in some of the previously mentioned works on language adoption [MM20, CPCS18]. However, as this thesis deals with more than one language, the flexible ANTLR tool is chosen as a parser generator. Unlike language specific compilers, ANTLR can generate lexers and parsers from any grammar file meeting its requirements. ANTLR is widely used in the literature for different works requiring language processing. Works using ANTLR for code analysis indicate its suitability for this thesis [AP21, SYCI18, LYB+19, APT20].

The next chapters will further describe how the studies regarding construct support of the transpilers and language usage of Swift and Kotlin were conducted within the scope of this work.

Chapter 4

Testing Transpiler Construct Support

In the preliminary work for this thesis [SS22], the language support of the considered transpilers was taken as an indication of their capabilities. Constructs unsupported by a transpiler would likely be ignored by the translation process or be translated incorrectly. To better assess whether using a transpiler for a project is worthwhile, it is valuable to have this information in advance.

Nevertheless, which aspects of the language are supported by Gryphon, Kotlift, SequalsK and SwiftKotlin is only partly transparent at this point in time. As illustrated in Chapter 3, the authors of the transpilers do not claim full support of the input language and limit their project to a set of supported language constructs. Although the support of those constructs is proven by providing code examples, shortcomings are not described in detail. Moreover, given the number of features in both Kotlin and Swift, it is likely that unsupported constructs unknown to the developers of the transpiler projects exist.

Admittedly, testing the transpilers on all aspects of Kotlin and Swift is out of scope for this thesis as well. Therefore, analyzing the support of basic constructs from the input programming language is presented as a starting point for evaluating the transpilers. This builds on the study conducted in the preliminary work, by repeating the experiments with the most recent versions of the transpilers and adding SwiftKotlin to the examination.

Next to the actual language construct support, all four presented Kotlin-to-Swift and Swift-to-Kotlin transpilers have differentiating aspirations regarding post-translation edits and the conformity of the output code to acknowledged style guideline's of the output language. While Gryphon aims to produce output code "without the need for post-translation edits" that is "reasonably understandable" [Ven22a], Kotlift just promises "largely valid code" [Stu20]. SequalsK wants to "strive for code that most programmers of the target language will prefer" [Sch21] while SwiftKotlin's emphasizes that the translation "will require manual editing" [Oll20]. Consequently, as the independent maintainability of the output is stated as an advantage for all the aforementioned transpiler projects, its stylistic correctness becomes an additional subject of study.

The following chapter describes how the study from the preliminary work was repeated.

Firstly, the test case pool of language construct is described. Secondly, a step-by-step explanation on how the experiments were conducted is presented. Lastly, the specifics of the experiment environment are listed for reproduction purposes.

4.1 Description of the Test Case Pool

As previously stated, a thorough analysis of the transpilers' language coverage w.r.t. all aspects of Kotlin and Swift is outside the scope of this thesis. Instead, this work aims to test the transpilers on a set of basic constructs representing both languages on a surface level. Both the documentations for Kotlin and Swift possess an overview chapter, namely "Basic syntax" for Kotlin [Jet22a] and "A Swift Tour" for Swift [App22c], with example code snippets fitting that description. Although both chapters only present selected features, they introduce the reader to commonly used core features, some advanced features and ultimately best practices. Those include, among others, variable and class declarations, conditionals, loops and (optional) typing. To avoid a completely random selection of constructs, the code snippets presented in those chapters were therefore chosen as test constructs for the transpilers. Notably, this possibly gave SequalsK an advantage over Gryphon and SwiftKotlin, as it already targeted the constructs presented in "A Swift Tour" as test cases [Sch21].

Table 4.1 (see page 33) summarizes the constructs that were ultimately extracted from both documentations and reworked into test cases. At the time of the experiments, "Basic syntax" from the Kotlin documentation represented Kotlin 1.6.21 and "A Swift Tour" represented Swift 5.6. The goal for the test case pool was to take the code snippets from both documentations as literally as possible. Nevertheless, not all exemplary constructs could be used unchanged as a test case, as some of them were not compilable as top-level declarations and required to be placed in some form of context, i.e., a function.

Aside from this, the examples given by both documentations often included constructs that were primarily unrelated to the feature that was actually shown in that section of the documentation. Next to describing the basic principles of the language, these chapters have a representative function as well, which might lead to exemplifying a feature more complex than necessary to spark interest in the language. E.g., the *String Templating* code example in the Kotlin documentation includes the string function for replacing part of a string with another. In Kotlin, this can be achieved by calling `String.replace(...)`, while the corresponding Swift function is `String.replacingOccurrences(...)`. In case the transpiler does not support this translation and leaves the function as it is, the *String Templating* test case results in non-working output code. Yet, this outcome does not provide any meaningful insight on the transpiler's support of string templates in general. In order to better differentiate between the basic translatability of a feature and problems caused by unrelated directives that were part of the example code, simplified test cases of the feature in question were added to the test case pool, if necessary. Additionally, such constructs were listed separately in Table 4.1.

Table 4.1: Language constructs extracted for Swift(^{S}) and Kotlin(^{K}).

Construct	Construct
Comments	**Printing**
End-Of-Line Comment^{SK}	Print Inline^{SK}
Block Comment on Multiple Lines^{SK}	Print Line^{SK}
Nested Comment^{SK}	**Strings**
Simple Values and (Optional) Typing	String & Number Concatenation^{SK}
Variables & Constants^{SK}	Interpolating One Variable^{K}
Explicit Typing (for `Int` & `Double`)^{SK}	Interpolating Expression^{SK}
Type Casting (`String` $\rightarrow$ `Int`)^{SK}	Multiline-String^{SK}
Implicit Typing^{SK}	Uppercase[4], Replace, Starts With^{SK}
Type Handling Without Initialization^{SK}	**Custom Types and Instances**
Top Level Declaration^{SK}	Classes With Properties/Functions^{SK}
Deferred Assignment^{SK}	Empty Classes^{K}
Type Checking With `is`^{SK}/`!is`^{K}	Superclasses^{SK}
Automatic Casting^{K}	Subclasses (Overriding Prop./Func.)^{SK}
Optional Value & Func. Return Type^{SK}	Constructor[5]-Like^{SK}/Deconstructor-Like^{S}
Control Binding Structure^{SK}	Property `get` & `set`^{SK}
Smart Cast After Null Check ^{K}	`willSet` & `willSet` Access. Other Prop.^{S}
Default Value for Optional^{SK}	Object Instantiating^{SK}
Safe Accessing of Optional^{SK}	Properties & Function Accessing^{SK}
Collection Types	Enumeration^{SK} (W. Function^{SK})
Array-[1]/Map[2]-Like Definition^{SK}	Enum. With^{SK}/From^{S} Raw Value
Initializing Empty Array-/Map-like^{SK}	Enum. With Associated Value^{S}
Overwr. Array-/Map-Like W. Empty^{SK}	Shorthand Enumeration Accessing^{SK}
In Collection^{SK}	`Data Class`^{K}/`Struct`^{S}
Array-Like Filter, Sort By & Map^{SK}	Interface-Like Def. & Implementation^{SK}
Functions and Lambda-Likes	Overriding Interface-Like Function^{SK}
Function W. Return[3] & Typed Param.^{SK}	Extension^{SK} (Incl. Self-Mutating^{S})
Function Call^{SK}	**Control Flow and Ranges**
Function Labels (Included and Omitted)^{S}	Iterating Array-Like^{SK} (With Index)^{SK}
Function Body as Expression^{K}	Iterating Array-Like With `forEach`^{SK}
Return Void^{SK}, Expl. Void^{SK}, Func.^{SK}	Iterating Map-Like^{SK} (With Placeholder^{S})
Function as Argument^{SK}	While Loop & Do-While-Like Loop^{SK}
Assign Function to Variable^{K}	Switch-Like^{SK}
Nested Function^{SK}	Conditional in Typed Switch-Like^{SK}
Lambda-Like^{SK} (Without Return Type^{SK})	If-Else^{SK}
Error Handling and Generics	Within a Range^{SK}
Defining Exception-Like^{SK}	Out of a Range^{SK}
Throwing/Catching Exception-Like^{SK}	Iterating Over a Range^{SK}
Safe Variable Assignment^{SK}	Iterating Over a Progression^{SK}
`Defer`^{S}	**Misc.**
Gen. Func.^{SK}, Enum.^{S} & Requirements^{S}	$++/--^{K}$

[1] `listOf`, `mutableListOf`, `arrayListOf`, `arrayOf`, `mapOf` were considered for Kotlin.

[2] `mapOf`, `mutableMapOf` were considered for Kotlin.

[3] Including multiple returns for Swift.

[4] It was tested if either the deprecated `toUpperCase` or `uppercase` worked.

[5] Both primary and secondary constructors were considered for Kotlin.

4.2 Experiments

Figure 4.1 (see page 35) depicts the workflow of the experiments for one direction of translation, whose steps will be described in more detail in the following. In order to extend the test case pool for both languages and provide an expected translation to compare the transpiler output code to, the examples from both documentations were manually translated into the respective other programming language. This resulted in the coverage of even more features for each language. E.g., "A Swift Tour" covers enums while Kotlin's "Basic syntax" chapter does not. However, both languages contain features that, to the best knowledge of the author, are unparalleled in the other language. Those include, e.g., *Self Mutating Extensions* in Swift. Although the constructs representing those features were kept for their originating language to evaluate the transpilers' output, no manually translated example was provided. Furthermore, any duplicate constructs adding no additional valuable insight were excluded from the test case pool.

In summary, 104 constructs for the translation from Swift to Kotlin and 102 constructs for the translation from Kotlin to Swift are considered for the experiments in this thesis. The number of constructs was increased from the preliminary work due to further differentiation made necessary by including SwiftKotlin. Notably, the constructs for *Package Import* and *Program Starting Point* from the Kotlin chapter were deliberately taken out of the evaluation, as their translation could not be simply reduced to being correct or incorrect. While Kotlin uses a package and import logic like Java, import statements are reserved to frameworks in Swift and unnecessary for the content of other files in the project. As for the starting point of the program, Kotlin requires a `main`-function, while Swift starts at the first line of the project's *main.swift*-file. As reported in Section 4.1, the test case pool was initially composed of examples from both the language's overview chapter and manually translated examples from the other language's chapter. Adjustments to the code guaranteeing its compilability were made in that step as well. The resulting test cases were then used as input to the transpilers which would return files of the target programming language. This step introduced a level of automatization, by calling all transpiler's **command line interface (CLI)** tools relevant to the current translation direction from a separate shell script, to reduce the effort of addressing each tool separately. Notably, manual translation options offered by Gryphon and Kotlift were not used, except for replacing `Int32` with `Int` in Kotlift's configuration file to avoid return type mismatching when testing the output with specifically written unit tests.

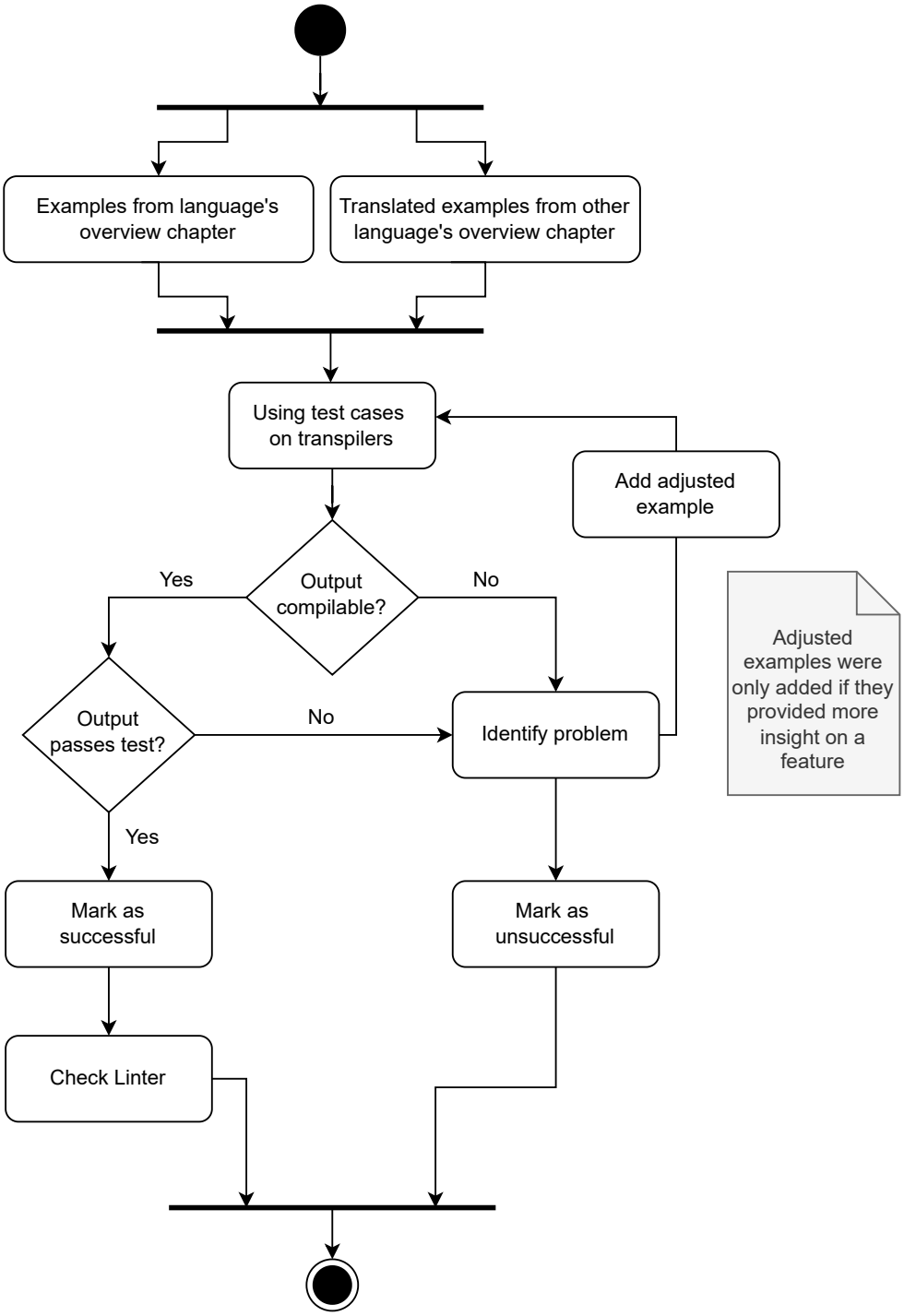


Figure 4.1: Workflow for identifying construct support of the transpilers.

Figure 4.2: Style warning in IntelliJ IDEA.

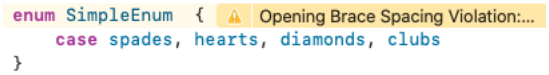


Figure 4.3: Style warning in Xcode with SwiftLint.

Support files provided by SequalsK and Gryphon were considered, as they are seen as part of the core transpiler. Although Kotlift also offers such a support file, it could not be used for the experiments as it used a deprecated Swift version using now invalid syntax.

The output created in the previous step was then further analyzed in an IDE native to the output programming language, namely IntelliJ IDEA [Jet22c] for Kotlin and Xcode [App22h] for Swift. If the output turned out to be not compilable, the problem was identified and an adjusted example was added to the test case pool, if further evaluation of the construct was deemed necessary. The current test case itself was then marked as **unsuccessful** for the transpiler in question.

If the output was compilable, a unit test corresponding to the test case was run in the environment of the aforementioned IDE. Those unit tests were manually written for each test case in the XCTest framework [App22i] for Swift and the kotlin.test library [Jet22g] for Kotlin. Ultimately, only output code both compilable and passing the unit test was marked as **successful**. In case the test should fail, the construct translation was marked as **unsuccessful**.

Lastly, the maintainability aspect of working output code was assessed manually by the author of this thesis by judging its readability. This was done by examining the output code on its formatting and similarity in naming compared to the input code. In order to test conformity with existing style guidelines, a linter was run on the code separately. Linters are programs for static code analysis that flag problems in the code, ranging from semantic and syntactical issues to stylistic preferences. Those preferences may include spellings, formatting or naming conventions. For Kotlin, IntelliJ IDEA's built-in style guide for Kotlin following official coding conventions was used. For Swift, the SwiftLint [Rea22] plug-in for Xcode was used, which is based on generally accepted guidelines from the Swift community. If a **style warning** should appear for the translation, this was noted accordingly. Figure 4.2 depicts how a warning was displayed in the Kotlin code in IntelliJ IDEA, while Figure 4.3 shows a warning within the Swift code in Xcode. Suggestions that were already noted in the source

code, such as shortened variable declarations, were not considered.

4.3 Environment

The experiments were conducted on macOS Catalina, version 10.15.7. The output from the transpilers was evaluated using IntelliJ IDEA 2022.1 and Xcode version 12.0. The SwiftLint plugin was in version 0.47.1. Table 4.2 shows the version of the transpilers used for the experiments.

Table 4.2: Transpiler versions.

	Version	**Last Updated**
Gryphon	0.18.1	9 Nov 2021
SequalsK	0.8.5.4	12 Jan 2022
Kotlift	N/A[1]	26 Mar 2020
SwiftKotlin	0.2.5	17 May 2020

[1] Kotlift lacked any information on its versioning, but the most current available version was pulled from GitHub.

4.4 Summary

This chapter described the methodology used for assessing construct support for Gryphon, Kotlift, SequalsK and SwiftKotlin.

The experiments conducted built upon the preliminary work done for this thesis [SS22]. To evaluate the transpilers' translating capabilities, construct-based experiments were presented. The test case pool for these experiments was assembled from the overview chapters of the Swift and Kotlin documentations by transforming the code snippets provided there into compilable test cases. Then, they were manually translated by the author of this thesis to the respective other programming language and ultimately used as additional test cases. In the end, 104 distinct constructs for Swift and 102 constructs for Kotlin could be derived from the test cases.

The experiments for each direction of translation were conducted as follows: Firstly, the test case pool derived from the input language's overview chapter and the manual translations of the target language's overview chapter were used as input for the transpilers. Secondly, the compilability of the output code was tested. If it failed, the problem was identified and the construct was marked as unsuccessful for the transpiler, concluding the experiment. If further examination of the problem seemed appropriate, an adjusted example

was added to the test case pool. If the output was compilable, a specifically written unit test was run on the output code. Only if the outcome was valid, the construct was ultimately marked as successfully translated. Additionally, the readability and therefore maintainability aspect of the output code was assessed using both manual evaluation and a linter following the best practices of the target language. Any notable warnings that were produced by the linter were written down. Afterwards, the experiment was concluded.

The intention behind the experiments presented in this chapter was to test transpiler support of a presumably basic set of constructs. While conducting the experiments, every construct was treated equally and two possible outcomes regarding their translatability were expected – either **supported** or **unsupported**. However, a look at the test case pool shows that some constructs differ significantly in terms of their complexity and presumably translatability. At the same time, it can be assumed that developers do not use all aspects of a language in equal measure. In conclusion, the impact of unsupported constructs on the applicability of the transpilers differs as well. Analyzing the popularity of a language construct would add more practical relevance to the study presented thus far.

Chapter 5

Mining Open-Source Applications

As previously described in Chapter 4, code examples from the two overview chapters of the Kotlin and Swift documentations, namely "A Swift Tour" and "Basic syntax" for Kotlin, were chosen to create a test case pool roughly representative of the languages. These chapters are the starting point of the respective language's documentation, and it can be assumed that the features described there are widely known amongst developers. At the same time, previous studies suggested that the constructs used to exemplify those features might not garner the same popularity regarding their usage [CPCS18, MM20, Zay20]. Nevertheless, most of the test constructs presented in Chapter 4 are not covered by previous work. Undoubtedly, a transpiler's applicability would suffer from neglecting a construct that is popular amongst developers from the input programming language's community. In addition, treating the support of each construct equally is not representative of how different constructs affect transpiler applicability, as some constructs might be used more often than others. In conclusion, a construct's popularity should be applied as a metric to the results from Chapter 4 for evaluating Gryphon, Kotlift, SequalsK and SwiftKotlin in a more practice-oriented fashion.

Moreover, not only developers who want to use one of the transpilers would benefit from such an extension of the study. During the experiments illustrated in Chapter 4, if a construct from the documentation included a directive that caused an incorrect translation, this directive was viewed separately and a simplified construct was potentially added to the test case pool. On the one hand, this provided more insight on the transpiler's general support of a feature. On the other hand, this increased the already existing dissonance in translation complexity between the entries in the result table. Chapter 3 already showed potential difficulties for translating between Kotlin and Swift. This included a categorization for classifying the translatability of language constructs by Schultes [Sch21], ranging from identical translation to not translatable. Thus, varying implementation effort is required when adding the support of another construct to a transpiler. In this context, it is important to know to what extent this effort is worthwhile to the authors of Gryphon, Kotlift, SequalsK and SwiftKotlin.

Determining the popularity of the constructs from the test case pool of Chapter 4 requires a large enough sample pool of relevant Swift and Kotlin projects. Consequently, source code

was mined from third party sources in a **mining software repository (MSR)** study [KCM07] for this thesis. MSR studies are conducted to gain insight into various aspects of software, e.g., how developer behavior evolves over time or to what extent certain constructs are used.

The following chapter describes how the sample pool for the MSR study conducted in this thesis was assembled and prepared for further processing. Firstly, a step by step explanation presents how relevant repositories were found using GitHub Search [DAB21], followed by a description of the criteria applied to the projects to identify them as Android and Apple OS app projects. Next, it is outlined how potential noise created by third party dependency managers was removed from Swift projects. Subsequently, the usage of the cloc tool [Dan22] for extracting general metrics relevant to the later evaluation process is elaborated.

5.1 Finding Relevant Kotlin and Swift Repositories

Source code collaboration tools, and especially their most popular representative by far, GitHub [Git22b], are widely popular amongst developers for various reasons [Jet22h]. GitHub does not only make version control and setting up continuous integration pipelines easy, but offers a safe environment for collaboration on a project through multiple branches, project forking and reviewing of merge requests by outside contributors. In addition, public projects can be published for free. GitHub prides itself with being an open-source friendly platform, with millions of open-source projects existing there [Git22d].

With regard to the MSR study conducted for this thesis, it could be assumed that a sensible size of open-source Kotlin and Swift projects reside on GitHub. In order to achieve statistical significance and diminish the influence of outliers, it was decided to collect as many relevant projects as possible illustrating construct usage. Still, GitHub is not built with mining its repositories in mind and searching for a representative amount of repositories fulfilling research criteria is not straight-forward. One way of achieving this is by leveraging the GitHub **representational state transfer (REST)** API [Git22a], or using tools incorporating it.

5.1.1 The GitHub REST API

The GitHub REST API, for the remainder of this thesis simply referred to as the GitHub API, allows "to create integrations, retrieve data, and automate your workflows" [Git22a]. Generally, REST APIs allow the retrieval or manipulation of data via HTTP by sending either GET, POST, PUT or DELETE as a request method to an endpoint of the API. In the case of the GitHub API, JSON is used for sending and receiving data. However, the number of requests that can be sent to the GitHub API is limited to 60 requests per hour for unauthenticated clients and 5,000 requests an hour for authenticated clients. Further restrictions include a maximal number of 1,000 results when using the GitHub API for finding specific items in repositories, accompanied by not being able to send more than 30 requests per minute as an authenticated request and 10 requests per minute as an unauthenticated request. Notably, these circumstances lower the efficiency of finding data suited to the criteria of a MSR study.

5.1.2 GitHub Search

In order to improve upon the applicability of the GitHub API for MSR studies, Dabic et al. developed the **GitHub Search (GHS)** tool [DAB21]. GHS facilitates filtering repositories according to certain criteria including programming language, date-based filters, filters on activity and the repository's popularity. Notably, the programming language filter returns projects which are written mainly in the sought after language. Unlike when using the GitHub API, no restrictions regarding limited requests per hour are imposed. Furthermore, GHS only mines projects having at least 10 stars, i.e., projects that have been marked as a favorite by 10 or more GitHub accounts. Although the authors admit that stars are no reliable indicator of the quality and/or relevance of a project, they claim that this restriction "provides a reasonable compromise between the quality of data and the time required to mine and continuously update all projects" [DAB21]. After the completion of the querying process, the user can download the results from GHS as an XML-, JSON- or CSV-file. These files include all information that is normally visible by visiting the project's GitHub page, like its full name, number of commits, contributors, default branch and the number of watchers.

5.1.3 Filtering for Relevant Kotlin and Swift Projects

When using GHS for finding Kotlin and Swift projects, 22,471 results can be obtained for Swift and 13,576 for Kotlin as of 22 May 2022. Traditionally, MSR studies have a proneness to being skewed by noise, as pointed out by Barros et al. [BHWS21]. When basing a study on source control management systems, they recommend filtering for relevant repositories beforehand and extracting only necessary data from those repositories. As described in the previous subsection, GHS already excludes projects with less than ten stars in order to reduce unwanted results. Still, upon further inspection of these projects, some appeared to being abandoned for quite some time, possibly using old versions of Kotlin/Swift. As a consequence, a filter regarding the last commit to the project was applied.

Kotlin

According to the timestamp visible in the "Basic syntax" chapter from the Kotlin documentation, the last update was performed on 13 September 2021 [Jet22a]. However, according to the publicly available changelogs, these changes merely contained updating the chapter with information on a Kotlin course [Jet22b]. As the chapter itself was added to the documentation on 11 February 2021, it can be assumed that most of the features presented in "Basic syntax" already existed before its release. Notably, the only changes since then included fixing typing errors and switching the deprecated string function `toUpperCase` to `uppercase`, in accordance with the release of Kotlin 1.5.0. However, since the latter is a non-breaking change and `toUpperCase` is still usable, most developers would not feel urged to update their code accordingly right away. Hence, the date this change was made is not adequate as a filter for narrowing down the GHS results.

To approximate a more relevant date, the Markdown-file that is used for rendering the "Basic Syntax" chapter in the web front-end was examined further [Jet22f]. Listing 3 (see page 42) shows an excerpt of the file *basic-syntax.md*. The Kotlin code exemplifying a

Listing 3 Excerpt of *basic-syntax.md* from the Kotlin documentation [Jet22f].

````
{kotlin-runnable="true" kotlin-min-compiler-version="1.3"}

A function body can be an expression. Its return type is inferred.

```kotlin
//sampleStart
fun sum(a: Int, b: Int) = a + b
//sampleEnd
fun main() {
    println("sum of 19 and 23 is ${sum(19, 23)}")
}
```
````

feature is located inside ```` ```kotlin [...] ``` ```` ````. The directives outside `//sampleStart` and `//sampleEnd` remain invisible to the viewer of the "Basic syntax" chapter, so only the function `sum` will appear as an exemplary code snippet on the website. In the whole file, every subsection containing a piece of runnable exemplary code is preceded by a directive stating the minimal Kotlin compiler version as Kotlin 1.3. When comparing the release notes since Kotlin 1.3 to the test cases directly derived from "Basic syntax" and the manually translated "A Swift Tour" test cases, all constructs existed at Kotlin 1.3 release. Kotlin 1.3 itself was released on 29 October 2018 and introduced features that are relevant to the test case pool from Chapter 4, like the automatic casting of variables that were already type checked. Consequently, the GHS results were filtered for projects having at least one commit after 29 October 2018. Ultimately, 11,605 of the previous 13,576 projects remained.

Swift

The revision history for the Swift documentation dates the last modification of "A Swift Tour" to 21 March 2016 [App22a]. Since the fast evolving Swift programming language underwent three major releases in the meantime from then until 2022, this was met with suspicion and examined further. The WayBack Machine [Int22] is an online service archiving the state of billions of websites over time. The earliest available state of "A Swift Tour" implementing the changes noted in the 21 March 2016 entry of the Swift documentation's revision history can be found in the recording of the WayBack Machine from 23 April 2016 [App16]. When comparing the state of the site then to the most recent state of the site, some deviations in code examples became apparent indeed. Most likely, changes to "A Swift Tour" since 21 March 2016 have not been noted explicitly in the revision history. Next to minor differences, like comments or printing, and an example for variable argument size for functions that is missing in the current version of "A Swift Tour", some differences affecting the test cases described in Chapter 4 exist. Table 5.1 (see page 43) lists these

differences, adds the Swift version the construct was released and the date of the state they are accessible for the first time via the WayBack Machine. The changes surrounding arrays and dictionaries solely modify existing examples by adding or changing directives to language constructs that have been present since Swift 1.0. Nonetheless, changes in the argument label syntax and the enumeration cases notation were introduced in Swift 3.0 Similarly, *Multiline Strings* were introduced in Swift 4.0. With this in mind, only repositories with a commit after the release of Swift 4.0 have a chance of incorporating the constructs that are described in the most current version of "A Swift Tour".

Table 5.1: Missing constructs in the 23 April 2016 version of "A Swift Tour".

	Since Swift	**Present in "A Swift Tour"[1]**
Multiline Strings	4.0	30 June 2017
Appending Array Items	1.0	20 September 2018
Different Syntax for Initializing Empty Arrays/Dictionaries	1.0	30 June 2021
Placeholder for Iterating a Dictionary	1.0	27 February 2021
Different Syntax for Argument Labels	3.0	21 February 2017
Lowercased Enumeration Cases	3.0	21 February 2017

[1] This date is an approximation done by searching through the WayBack Machine and possibly does not reflect the real date of change in the Swift documentation.

In addition to the test cases directly derived from "A Swift Tour", the test case pool described in Chapter 4 consists of manually translated examples from the "Basic syntax" chapter of the Kotlin documentation. Therefore, those were reviewed on any language constructs that were added to Swift after the 4.0 release. In fact, *Implicit Returns*, that were used as a translation of Kotlin's *Function Body as Expression*, were added in Swift 5.1. As a result, a repository from the sample pool must have at least one commit after 10 September 2019, the day Swift 5.1 released. This reduced the number of results from GHS to 10,876.

5.2 Identifying App Projects

Gryphon, Kotlift, SequalsK and SwiftKotlin primarily are advertised as tools for cross-platform mobile development [Ven22a, Stu20, Sch21, Oll20]. However, as pointed out in Chapter 1, the Android and Apple OS ecosystems comprise a multitude of platforms, from mobile devices like smartphones and tablets to smart TVs. As some projects target multiple

hardware anyway and other platforms than mobile do present possible use cases for the transpilers, all app projects developed for Android and Apple OS platforms were considered relevant for the MSR study performed in this thesis. Yet, identifying them from the projects collected by GHS thus far required analyzing the content files of the repositories. For this purpose, certain criteria were defined for both Android and Apple OS apps.

5.2.1 Criteria for Android App Projects

Zayat proposes a way of differentiating Android projects from libraries and other utilities by a set of certain properties [Zay20]. According to their thesis, the manifest file *AndroidManifest.xml*, that must exist in every Android app project, and the usage of the `setContentView` function in an `Activity` class are characteristic to Android projects. In Android, an `Activity` serves as a container class for processing the input from user interaction. The `setContentView` function is used to load an XML layout file describing the respective UI file for the current `Activity`. Therefore, a project using `setContentView` at least once can be assumed to be an app with an UI. However, this is not true for Android apps whose UI is solely built with Jetpack Compose [Goo22e]. Jetpack Compose is an alternative, declarative way of defining an UI in Kotlin, opposed to the traditional XML layout files.

While building on Zayat's methodology to identify Android projects by searching for the existence of an *AndroidManifest.xml*, this thesis proposes an extension of the recognition of existing app views. In addition to searching Java and Kotlin files for `setContentView`, Kotlin files should be searched for characteristics of Jetpack Compose as well. When using Jetpack Compose, the building blocks of the UI are instantiated in the block of `setContent`. Therefore, the existence of a `setContent` statement suggests the existence of an UI.

5.2.2 Criteria for Apple OS App Projects

For identifying Apple OS app projects from the 10,876 Swift repositories found by GHS, a similar approach to the one described for Android was taken. When creating a project with Xcode, which is the native IDE for developing Apple OS app projects, a property list file named *Info.plist* must exist for all executable bundles of the project [App22b]. Therefore, the first criterion for a Swift project to potentially be an Apple OS app project is to possess an *Info.plist* file, something a library project would lack. However, this criterion alone would include frameworks. Consequently, the existence of at least one view describing an UI was the next logical condition for identifying an Apple OS app project.

In an Apple OS app project, a view can either be defined by an XML like structured layout file or alternatively in Swift in a declarative fashion by using SwiftUI [App22g]. The lifecycle for apps building upon XML structured layout files is usually handled by a class implementing either `UIApplicationDelegate` for apps for iOS-based devices, `WKExtensionDelegate` for watchOS for Apple Watch apps or `NSApplicationDelegate` for macOS apps. The implementing class can be seen as the root object of the app and is, among others things, responsible for managing the app's views. Therefore, the existence of either of those protocols in a Swift or ObjectiveC file suggests an app project with an UI. Opposite to apps

whose UI are defined by XML layout files, the views defined in SwiftUI usually do not use a separate controller class. Instead, the response to user interaction is handled directly in the object describing the structure of the view, which implements SwiftUI's `View` protocol. Consequently, the existence of a Swift file importing SwiftUI and declaring an object implementing the `View` protocol suggests a SwiftUI based app project with an UI.

5.2.3 Filtering the GHS Results for App Projects

In a first attempt to filter the list of Kotlin projects found by GHS, requests were sent to the GitHub API to get information about the content of the repositories. However, this had multiple downsides. The GitHub API is protected against abuse by inflicting various rating limits against clients if too many requests are sent in a short period of time. Generally, they are two types of rate limits: A **primary rate limit** is triggered, when the client exhausted the allowed number of requests that can be sent within a certain time period. The condition for hitting a **secondary rate limit** is to "repeatedly request data that is computationally expensive" [Git22c]. However, this definition is somewhat vague, and the response triggered by a secondary rate limit does not contain any information on when the request can be repeated. The client has no choice but to wait for an arbitrary number of seconds before trying again, while still potentially hitting another secondary rate limit. When using the GitHub API for the purposes of this study, which can be considered costly as they required searching through the code in the repository, the process was notably slow as the secondary rate limit was hit regularly.

Furthermore, some inconsistencies were observed when sending requests to the GitHub API. In some cases, the API response suggested that the project did not meet the search criterion, although manual verification showed that the project was indeed an Android app project. When repeating the request, the API would most often return the expected result. Although this behavior could not be reproduced reliably, it does not seem to be exclusive to this study, as proven by a community post [Ale22]. The participants of the thread describe the same problem when using both Python and Ruby and suspect a bug within the GitHub API itself.

In addition to these drawbacks, the search query does not allow regular expressions and connection of query strings with a logical AND. However, the criterion defined for detecting a SwiftUI based app requires looking for both the import statement of the framework and the implementation of `View` in the same file.

After weighing up these disadvantages in conjunction with the time required, it was decided to download the repositories as ZIP-archives instead and then to verify them locally. On the one hand, the worst-case of this methodology would include downloading and analyzing repositories that are of considerable size but are not app projects. On the other hand, downloading the first 600 Kotlin repositories as archives and verifying them took approximately 13 minutes, while just verifying the same amount using the GitHub API took approximately 1 hour and 18 minutes. Downloading the repositories as ZIP-archives was implemented as a Python script. Listing 4 (see page 46) shows the function used for requesting the ZIP-file contents. The `fetch_zip`-function was called inside a `for`-loop that iterated all entries of the CSV-file exported by GHS. It was parametrized with the current

Listing 4 Function for cloning the repositories as ZIP-archives.

```
def fetch_zip(repo_name, default_branch):
  response = requests.get("https://github.com/" + repo_name +
  ↪  "/zipball/refs/heads/" + default_branch)
  if response.status_code != 200:
    print(response.status_code)
    raise RepoNotValid
  else:
    return response.content
```

Listing 5 Function for validating the ZIP-Archives as app projects.

```
def check(pattern, path, include_file):
    stdout = os.popen('zipgrep -Elzw ' + pattern + ' ' + path + ' ' +
    ↪  include_file).read()
    return len(stdout) > 0
```

entry's repository name and default branch. Firstly, inside `fetch_zip`'s body, a call was made to GitHub requesting the default branch of a repository as a ZIP. If the response had a HTTP status code of 200 and was therefore fulfilled, its content was returned and then written to a ZIP-file. If the response had a status code other than 200, the custom exception `RepoNotValid` was raised. As the file download from GHS and the download of the ZIP files had a time discrepancy of a few days, some of the repository's naming changed in the meantime. In the case of the Swift repositories, one was even deleted. Therefore, `RepoNotValid` was handled by writing down the repository's name in a separate CSV-file, so it could be corrected manually, and then the download process could be repeated for those projects.

The verification of the projects as Android and Apple OS app projects was done using `zipgrep` [Gai22]. `Zipgrep` allows searching the contents of a ZIP-archive for regular expression matches. Listing 5 shows the function used for checking the downloaded ZIP-archive of a repository's default branch. The `pattern` argument passed to the function `check` describes a regular expression for the searched term, e.g., `''setContentView''`. The `path` argument contains the location of the ZIP-archive, while `include_files` describes what kind of files should be included in the search. E.g., when trying to find the characteristics of a SwiftUI app, only *.swift* files should be included. By using the `popen` function of the Python module `os`, which opens a pipeline to the command interpreter, the `zipgrep` command was issued. The `E` option declares the conformity of the search pattern to POSIX Extended Regular Expressions, while the `l` option defines that only the name of the first matching file is sent to the output and the search will be stopped afterwards. The `z` option defines that newline characters are omitted from the input. Finally, the `w` option declares that a match

must either be at the beginning of the line or preceded by a non-word constituent character, or at the end of the line or followed by a non-word constituent character. The output of `zipgrep` is saved to `stdout` and as `stdout`'s length would be zero in case no matching file was found, `check` is only true when `stdout`'s length is greater than zero. Ultimately, 7,483 Swift app projects and 7,417 Kotlin app projects remained.

5.3 Removing Dependencies

When comparing the findings of their study on iOS code smells with related work, Rahkema and Pfahl noted that keeping project dependencies as part of the analyzed source code leads to different results [RP20]. As developers usually do not concern themselves with the source code of these dependencies, they decided to remove them from their source code analysis. Furthermore, as those dependencies might be used in several projects, they could potentially skew the perception of the importance of a certain construct. Therefore, it was decided to ignore dependencies when counting the constructs for this study, too.

In their study on the usage of error handling mechanisms in Swift projects, Cassee et al. removed the default download directories for the popular Swift third-party dependency managers CocoaPods [Coc22] and Carthage [Car22] from the project prior to the source code analyzation process [CPCS18]. This methodology was adopted for this study. However, as the recognition of some constructs relied on information from other files, as it is further elaborated in Chapter 6, they were not removed permanently. Still, they are not counted into the total number of analyzed files, and they were not examined for construct usage.

Another popular way of adding dependencies to Swift projects is by using the Swift Package Manager [App22f]. However, the source files of remote dependencies are not saved directly to the project, but cached to a local folder by default. In the Android development ecosystem, Gradle [Gra22] is a frequent choice for dependency management, as it is already configured by default when creating a new Android project with Android Studio, the native IDE for developing Android projects. Apache Maven [The22b] is another poplar tool for managing dependencies in that context. Like Swift Package Manager, both of them cache dependencies outside the actual project directory by default. Therefore, no further actions were taken regarding the Android app projects. Notably, third party libraries that were added manually to the project's folder still remain as part of the source code. However, given the sheer amount of projects in the sample pool, it was not feasible to remove all of them manually. Consequently, the results have to be evaluated with this in mind.

5.4 Extracting General Metrics With cloc

For finalizing the sample pool, the metrics on the number of files, blank lines, lines with comments, and lines of code for Kotlin or Swift respectively were extracted from the repositories. This was implemented using the cloc CLI tool [Dan22]. Cloc offers writing the results in CSV-format. After being called by a Python script, the CSV output from cloc was appended to the existing CSV-file containing the results of GHS.

5.5 Summary

This chapter outlined how the sample pool for conducting a MSR study for gaining insight on the popularity of the constructs collected as described in Chapter 4 was created. The projects were taken from GitHub, which is a popular source code collaboration tool where many open-source projects reside. Because of downsides including slow response time and limited search options, relevant repositories were searched for by using the GitHub Search (GHS) tool instead of the official GitHub API, which is not as suited for assembling a sample pool for a MSR study. The relevancy of a project of said sample pool was defined by, firstly, being **mainly written in either Kotlin or Swift**, secondly having at least **one commit contributed after the construct** from the test case pool (see Chapter 4) that **was last introduced** to the language **was released**, and thirdly, being an **app project**. The filters regarding the programming language and the limit for the last commit were applied by configuring GHS accordingly, with 29 October 2018, the day Kotlin 1.3 was released as the limit for Kotlin, and 10 September 2019, the release of Swift 5.1, as the limit for Swift. Ultimately, 11,605 results were found for Kotlin and 10,876 for Swift. The results were downloaded in form of a CSV-file, that included various information about each project like commits, stars, contributors, in addition to its name. Afterwards, those projects were downloaded as ZIP-archives and analyzed on predefined characteristics of Android and Apple OS app projects. Table 5.2 describes these criteria, that can be summed up as the possession of a file that must exist in an app project and is absent from other kind of projects, e.g., libraries, and a function indicating the implementation of an UI.

Table 5.2: Criteria for identifying app projects.

	Android App Projects	**Apple OS Projects**
App Project File	AndroidManifest.xml	Info.plist
User Interface Implementation	`setContentView` in Java or Kotlin files (XML layout) **and/or** `setContent` in Kotlin files (JetPack Compose)	`UIApplicationDelegate`, `WKExtensionDelegate` or `NSApplicationDelegate` in ObjectiveC or Swift files (XML layout) **and/or** `import SwiftUI` and `View` in Swift files (SwiftUI)

This resulted in the final list of relevant projects, with 7,417 Kotlin based Android and 7,483 Swift based Apple OS app projects. In order to remove noise and potential redundancies from the sample pool, the default folders of popular third party dependency managers CocoaPods and Carthage for Swift were excluded from construct analyzation. No folders were excluded for Android projects, as they usually rely on Android Studio's built-in dependency manager Gradle, which caches libraries outside the project by default. For finalizing the sample pool, metrics regarding each project's used programming languages and their lines of code in the

project were extracted using the cloc tool and added to the CSV describing the sample pool. All in all, the sample pool consists of 7,417 Kotlin based Android and 7,483 Swift based Apple OS app projects, with a total of 640,231 Kotlin files and 429,512 Swift files[1]. For a visualization of the metrics regarding files per project, stargazers, commits, and contributors, please refer to Appendix A Figure A.1 for the Kotlin projects and A.2 for the Swift projects. For a comprehensive list including all collected information about the Kotlin and Swift sample pools, please refer to the corresponding CSV-files in Appendix B.

Ultimately, the files from both sample pools became the input for a tool for automatic construct recognition, described in the next chapter, that detected the total occurrences of the unsupported constructs from Chapter 4.

[1]The files included by the aforementioned dependency managers CocoaPods and Carthage are excluded from this number.

Chapter 6

Automatic Construct Recognition

Although a sample pool as large as the one described in Chapter 5 helps the study for examining construct usage to attain statistical relevancy, manual counting of relevant language constructs becomes not feasible. Therefore, the usage of a tool for automatic construct recognition became necessary. While automatic construct recognition is part of the methodology of several related works on Kotlin and Swift [Zay20, MM20, CPCS18, RP20], a majority of the constructs considered in this thesis were not part of those studies. Therefore, the tools developed for automatic construct recognition in related work, even if they were made public, would have needed to be extended and adapted to fit the needs of this thesis. In addition, this work looks at two programming languages opposite to related work, that normally just focuses on either Kotlin or Swift and uses a language specific parser.

Consequently, a tool specifically for the MSR study conducted in this thesis was developed. For the remainder of this thesis, this tool will be referred to as the **Construct Analyzer Tool (CAT)**. The following chapter presents the concept behind this tool firstly and its implementation secondly. Thirdly, the validation of the tool is illustrated.

6.1 Concept

The goal behind the implementation of CAT was to create a program that took the **sample pool of Kotlin and Swift projects**, **automatically counted** the number of occurrences of the unsupported constructs and output the results in a **suitable data format** for further processing and evaluation. This general workflow is summarized by the depiction in Figure 6.1 (see page 52). Further specifications regarding these three basic requirements are described in the following.

6.1.1 Accepting of Both Swift and Kotlin Projects

Since this thesis deals with both Kotlin-to-Swift and Swift-to-Kotlin transpilers, and therefore with constructs from both Kotlin and Swift, it seemed effort reducing for CAT to be able to handle both languages. Developing a single program allowed for shared code between the

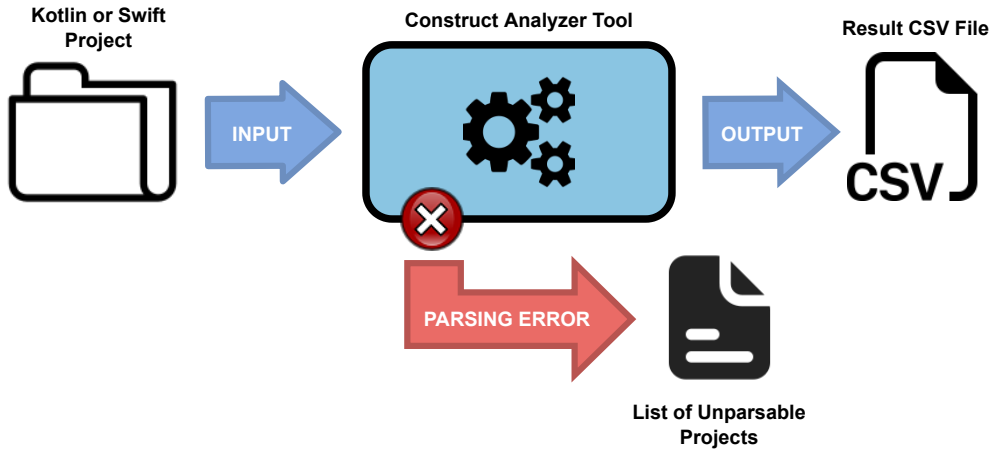


Figure 6.1: Workflow of the Construct Analyzer Tool.

Kotlin and Swift analyzer parts, w.r.t. abstract concepts applying to both. This included, e.g., models mapping language components like classes or variables.

6.1.2 Automatic Recognition of a Predefined Set of Constructs

As previously stated, the automation of the counting process was a necessity for the feasibility of the study. The general idea behind CAT was analyzing a parse tree for recognizing subtrees representing constructs. Next to constructs that could be simply identified by the existence of a certain node, this allowed for the identification of more complex constructs that were dependent on the context they were implemented in. In addition, the symbol table concept present in other language processing tools like compilers or transpilers was adopted (see Chapter 2). Thus, constructs whose identification depended on previous declarations, could be identified, too.

6.1.3 Result Output as CSV-File

After the project was analyzed, the results had to be noted in a file format that allowed further processing and creation of diagrams in an uncomplicated fashion. The CSV-format was a fitting choice, as CSV-files are not only more compact than other comparable formats like JSON, but can be easily imported in statistics programs and programming languages for data processing.

Nevertheless, it had to be assumed that some files would result in a parsing error, either due to errors in the input code files or to deficiencies in CAT. These files and their corresponding projects were noted in a separate lists.

6.1.4 Form

It was decided that CAT should be accessible from the command line. Since the author of this thesis was the only user of the tool, the implementation of a graphical interface was

not necessary. Furthermore, this allowed for the tool to be easily called by other automated processes. From the command line, the user should be able to pass the location of the project to be analyzed, the location of the output file and the input language.

6.2 Implementation

As previously stated, CAT was created as a standalone CLI tool. Most importantly, it depended on the ANTLR parser generator [Par22] previously introduced in Chapter 2, namely version 4.9.3. With exception to the Java classes and interfaces generated by ANTLR, the tool was entirely written in Kotlin. For creating the output CSV-file, the CSV generator from Apache Commons CSV [The21] was used.

Figure 6.2 (see page 54) depicts the general architecture of CAT in a package view. The architecture was composed of three main packages, with `Main` representing the entry point of the application, handling the user input from the command line. Dependent on the arguments passed to the command line, the components of the `constructAnalyzer` package analyzed the Kotlin or Swift input files accordingly and produced the output CSV-file. The analyzation of the input was performed by leveraging the classes created by ANTLR for language parsing, included in the `parsers` package. The models for mapping constructs to the symbol table were combined in the package `languageModels`.

In the following, the collaboration of the three main packages will be described in more detail.

6.2.1 Parsers

For creating parser and lexer classes, ANTLR expected a grammar file in the *.g4* format. For Kotlin, the official grammar source file was used [Jet22e]. For Swift, an unofficial *.g4* file was used [TY21, TYM21]. However, the website of that file also stated that it might be incomplete and possibly contained "ambiguities or wrong rules" [TYM21].

An instance of the lexer class for the respective language, subclassing ANTLR's `Lexer` class, processed the input character stream from the current Kotlin or Swift file. The specific parser class for the respective language subclassing ANTLR's `Parser` class was instantiated by using the lexer class as an input token stream. With the instance of the parser class, the parse tree was created by addressing the root node of the tree. Figure 6.3 (see page 55) depicts an exemplary parse tree for a Kotlin file containing the constant declaration `val a = 5`. Every node of the parse tree represented a specific rule from the input grammar or a terminal node at the end of a branch. For walking the parse tree, ANTLR's parse tree visitor concept for explicitly visiting the nodes of a parse tree by calling them was leveraged. For that purpose, ANTLR created an interface extending `ParseTreeVisitor`. This interface is implemented by a base class extending `AbstractParseTreeVisitor`, so a custom visitor class can extend this base class.

In order to easily map nodes to the language models used in the symbol tables, the functionality of the parser classes created by ANTLR was extended by using Kotlin's extension feature.

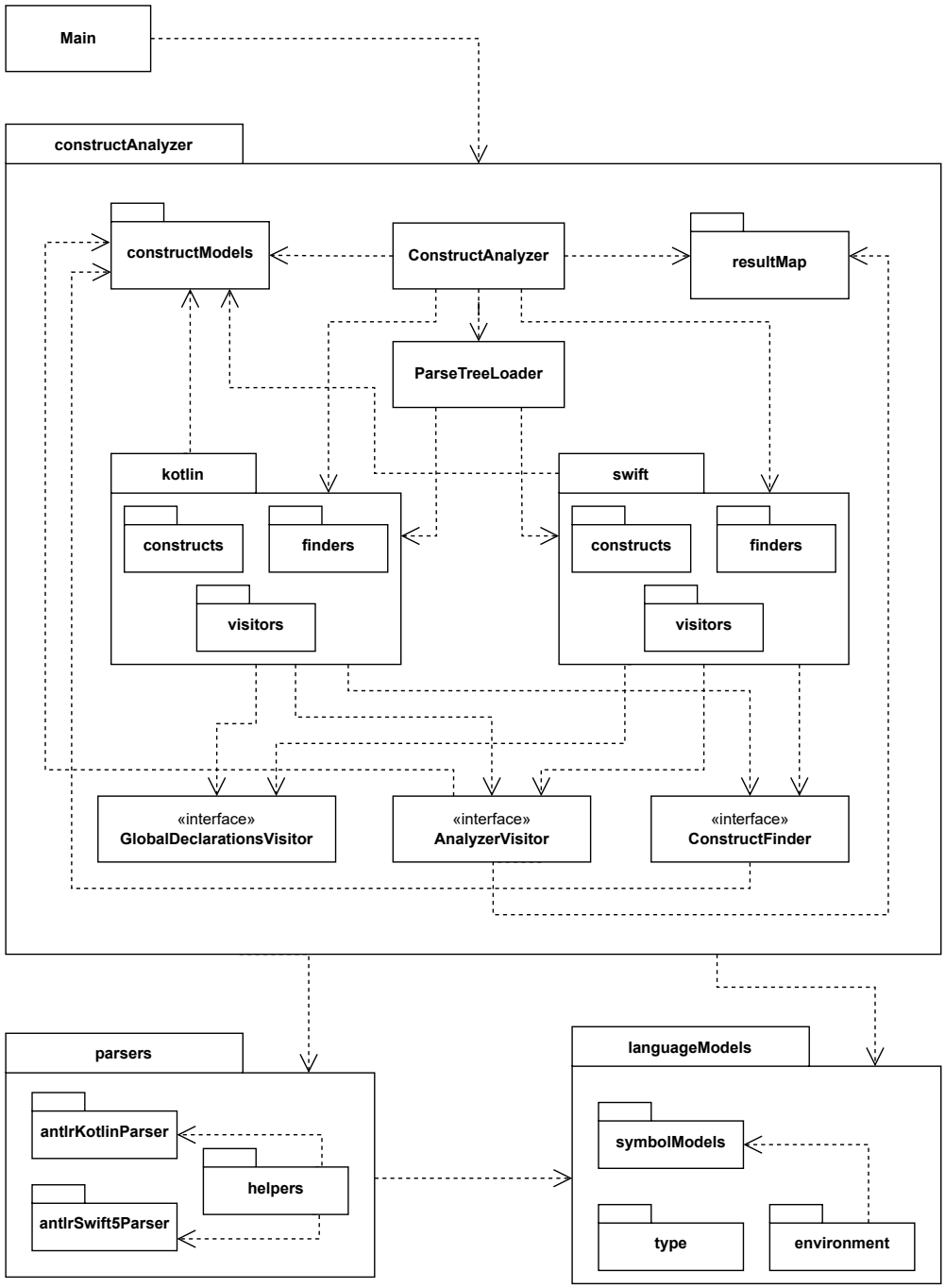


Figure 6.2: Construct Analyzer Tool package view.

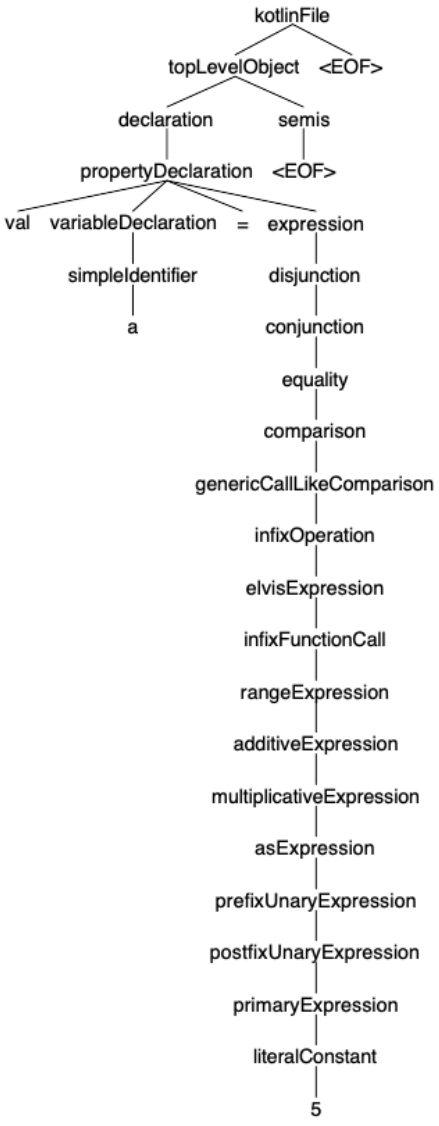


Figure 6.3: Kotlin parse tree generated with ANTLR.

Listing 6 Function extending Kotlin parser class.

```
fun KotlinParser.MultiVariableDeclarationContext.toModel(environment:
↪  Environment): List<VariableModel> {
    val p = this.parent as KotlinParser.PropertyDeclarationContext
    val expression = p.expression()
    val declarations = this.variableDeclaration()
    val models = mutableListOf<VariableModel>()

    for (d in declarations) {
        models.add(
            VariableModel(
                expression.text,
                d.type()?.toType(environment),
                true,
                d.simpleIdentifier().text
            )
        )
    }

    return models
}
```

E.g., Listing 6 shows the transformation of a Kotlin rule declaring multiple variables to a list of `VariableModels` by extending the `MultiVariableDeclarationContext` class with the `toModel` function.

6.2.2 Language Models

As various constructs required implementation context, the concept of symbol tables was incorporated into the implementation of CAT. This made accessing information about previously declared variables, functions and other programming components possible.

The basic implementation of a chained symbol table, represented by the `Environment` class, was based on the Java implementation by Aho et al. [Aho07]. By stacking an instance of `Environment` into another instance, the concept of block scoping was implemented, as illustrated by Figure 6.4 (see page 57). Next to an `Environment` instance representing the outer scope saved to the property `prev`, an `Environment` instance holds a list of `SymbolModel` implementations in an instance of `SymbolTable`.

Figure 6.5 (see page 58) depicts the classes mapping the language symbols from the subpackage `symbolModels` of the `languageModels` package. It's noteworthy, that only symbols needed for the analyzation process were actually implemented and that the models implemented for CAT do not represent the full extent of Kotlin and Swift features. All

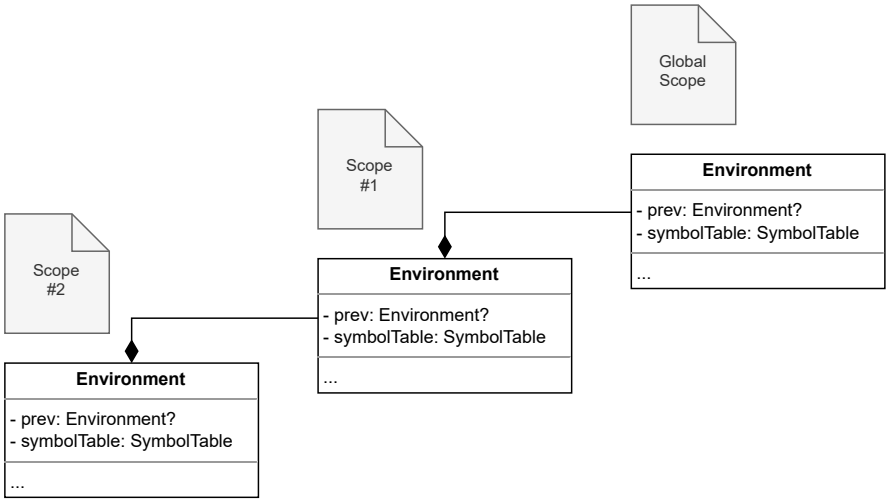


Figure 6.4: Chained symbol tables for implementing scopes.

concrete symbol classes implement the `SymbolModel` interface, giving access to the symbol's identifier and symbol type. The `SymbolType` enum consists of six members, namely `VARIABLE`, `CLASS`, `INTERFACE`, `ENUM`, `DATA_STRUCTURE` and `FUNCTION`. For some symbols, like interfaces or variables, the behavior is similar enough to describe them with the same model for both Kotlin and Swift. However, some differentiating concepts made separate classes necessary. E.g., functions in Swift can have multiple returns, while in Kotlin only one type can be returned. Next to the primitive types existing in both Kotlin and Swift, like integers, strings or booleans, custom types can be defined by declaring, e.g., classes or interfaces. To make these declarations easily accessible for typing variables, symbols that fell under this category extended `AbstractTypeModel` that implemented the `Type` interface. The primitive types of both languages were declared in enums also implementing `Type`.

6.2.3 Construct Analyzer

As described before and depicted in Figure 6.2 (see page 54), the actual analysis is triggered within the `constructAnalyzer` package. From the starting point of the application described in `Main`, the function `analyze` of an instance of the `ConstructAnalyzer` was called.

Firstly, all files from the project were parsed with the language specific parser and the parse trees created were stored into a `HashMap` with the name of the file as key. For improving parsing decision performance, ANTLR caches decision made thus far [PHF14]. In the case of CAT, this feature took a considerable amount of **random-access memory (RAM)** for a large number of input files, leading to a Java `OutOfMemoryException`. In addition, the overall parsing process was slowed down with less and less RAM available. After some trial and error, the solution that worked best for preventing this problem was by clearing the lexer and the parser caches every 400 files by using their `reset` function and the `clearDFA` function for the parser interpreter class.

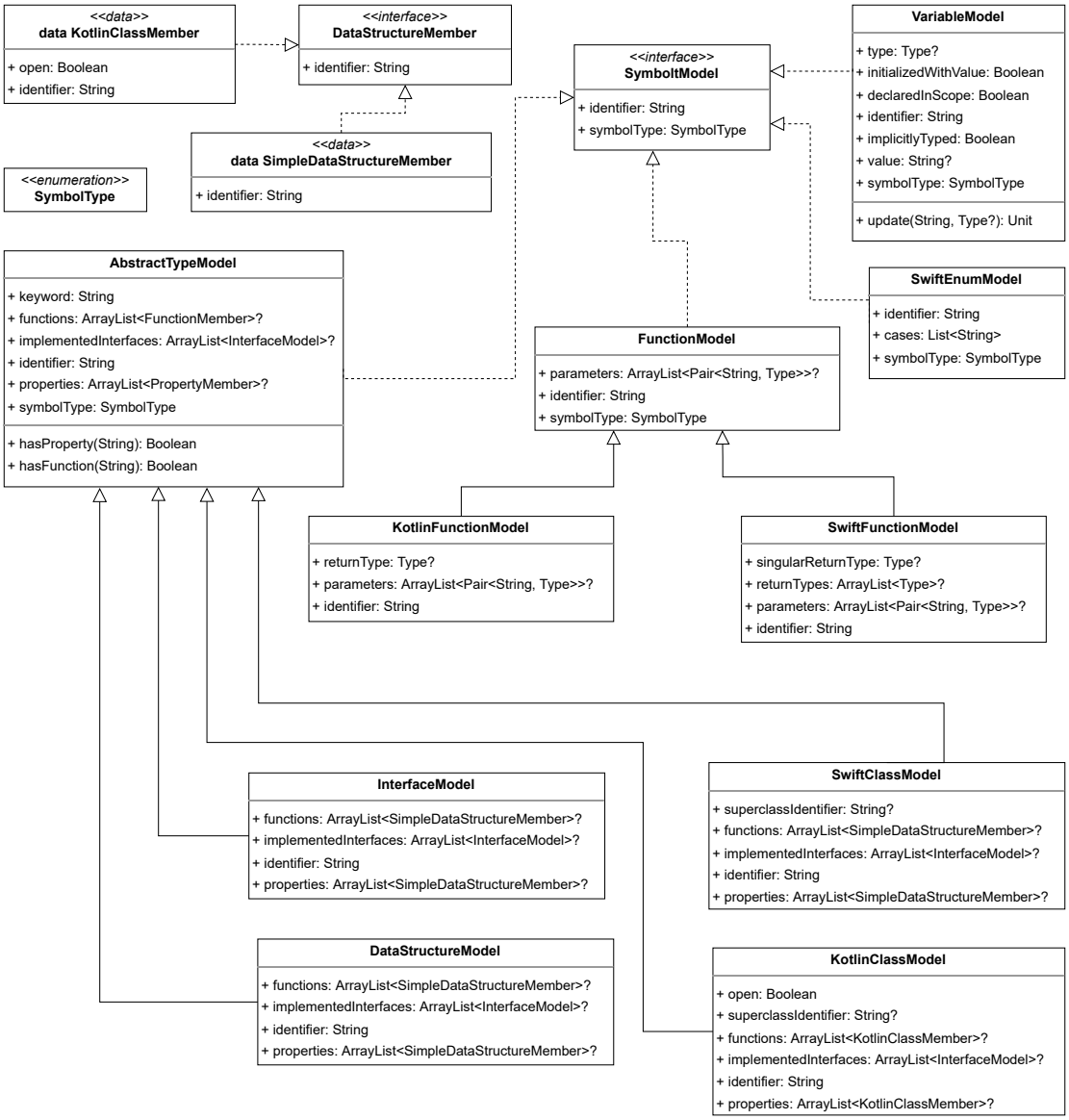


Figure 6.5: Simplified UML class diagram of the symbol models.

Still, some files took a considerable amount of time for parsing. Although the performance of the parser in general could be possibly improved upon by reworking the grammar, implementing and testing such changes was outside the scope of this thesis. At the same time, allowing the parsing process to take whatever time it needed was not feasible with the large amount of files that had to be parsed. Hence, the lexical analysis and the parsing process were limited to three minutes each and stopped when taking more. This timeout was implemented by the classes `ParserATNSimulatorWithTimeOut` and `LexerATNSimulatorWithTimeOut` from the `helpers` subpackage of the `parsers` package. Those were used as the value for `interpreter` for the parser and lexer classes, respectively. This solution was strongly based on a Stack Overflow post experiencing a similar problem [Kil19]. Files that were neglected due to the aforementioned time limit or files that were simply not parsable due to possibly incorrect source code were noted in a separate output file.

Before the parse tree of each parsable file could be analyzed for constructs, a global `Environment` incorporating all global declarations was built. For Kotlin, the specific imports for a file could be traced back by examining the subtree representing the `import` statements from the top of the file. In addition, all parse trees from files residing in the same directory had to be considered, too, as those would not need an explicit import. For Swift, import statements are only necessary for frameworks, and all declarations made inside the whole project are accessible from anywhere. Therefore, all files had to be added to the global `Environment` instance. All in all, it is noteworthy that declarations from dependencies outside the scope of the project were not included, as those would require downloading the source files of each library separately and analyzing them. In addition, declarations made in Java or ObjectiveC files that were incorporated into the Kotlin or Swift files through the interoperability of those languages, were not analyzed.

As previously described, constructs were found in the parse tree by making use of ANTLR's parse tree visitors. The corresponding base class was inherited by specific implementations for each Kotlin and Swift, namely the `KotlinAnalyzerVisitor` and `SwiftAnalyzerVisitor` classes, residing in the `visitors` package in the corresponding language's subpackage. Both of these classes also implemented the `AnalyzerVisitor` interface, that contained fields for the `Environment`, the list of constructs to be considered and a `ResultMap` instance. Within the `ResultMap` class, the occurrence of a construct was stored within a `HashMap`. This `HashMap` had objects implementing the `Construct` interface as keys and an `ArrayList` of instances of `Entry` as value. An `Entry` object would contain the name of the file the construct was found in and the line, so this information could later be listed in the output file.

The `Construct` interface, residing in the `constructModels` subpackage, was implemented by two specific enums for each language, namely `KotlinConstruct` and `SwiftConstruct` residing in the `constructs` package in the corresponding language's subpackage. Their members would list all constructs that were detectable by CAT.

In general, all constructs had a corresponding implementation, except for Swift's *Superclass Definition*, Swift's *String And Number Concatenation* and any constructs describing comments from both Kotlin and Swift. Since all Swift classes can function as superclasses without further specifications, all classes could be categorized as superclasses. However, this

Listing 7 Tier I construct example.

```
override fun visitDeinitializer_declaration(ctx:
↪  Swift5Parser.Deinitializer_declarationContext?) {
    noteResult(ctx!!,
        nodeConstruct = SwiftConstruct.CLASS_DEINIT
    )

    super.visitDeinitializer_declaration(ctx)
}
```

is likely not representative of the count of classes actually used as superclasses. So instead, classes implementing a custom superclass were searched for to still get a metric on superclass definition importance. W.r.t. *String And Number Concatenation*, the problematic part of the construct was the string type casting of an integer instead of the concatenation itself. So, the casting from an integer to string became the construct of interest. Comments were omitted by the parser, so it was impossible for CAT to recognize them in the parse tree. For the constructs that were represented in the parse tree, the complexity of identifying them could be divided into three tiers.

Tier I

Constructs from Tier I could be identified by the existence of a certain node in the parse tree. Therefore, their appearance was added to the results whenever the respective function for visiting that node was called. Listing 7 shows the overridden function for visiting the declaration of a *Deinitializer* in a Swift class from `SwiftAnalyzerVisitor`. Since this function is called upon for every *Deinitializer* declaration, the appearance of this construct is added to the `SwiftAnalyzerVisitor`'s `ResultMap` by the function `noteResult` every time.

Tier II

For a Tier II construct, nodes in proximity had to be analyzed as well to clearly identify the construct. The identification of constructs belonging to Tier II and III was outsourced to separate classes implementing the `ConstructFinder` interface. Listing 8 (see page 61) provides an example for such a class. A class conforming to `ConstructFinder` would firstly override the `ctx` property with a class extending `ParserRuleContext`, representing a subtree from the parse tree. Secondly, the `construct` property held the member of either `KotlinConstruct` or `SwiftConstruct` representing the construct of interest. Thirdly, the function `check` would contain the logic for identifying the construct in the tree. Classes implementing the `ConstructFinder` interface were instantiated in the body of the visit function for the node serving as the root of the subtree passed for `ctx`. In the case of the class described by Listing 8, this was the node for a property declaration.

Listing 8 Tier II construct example.

```
class FunctionAssignedVariableFinder(override val ctx:
↪  KotlinParser.PropertyDeclarationContext): ConstructFinder {
    override val construct: KotlinConstruct =
    ↪  KotlinConstruct.FUNCTION_ASSIGNED_VARIABLE

    override fun check(): Boolean {
        return ctx.expression()?.start?.type == KotlinParser.FUN
    }
}
```

Listing 9 Tier III construct example.

```
class ClassOverridingProtocolFunctionFinder(override val ctx:
↪  Swift5Parser.Class_memberContext, val environment: Environment) :
↪  ConstructFinder {
    override val construct: SwiftConstruct =
    ↪  SwiftConstruct.CLASS_OVERRIDING_PROTOCOL_FUNCTION

    override fun check(): Boolean {
        if (ctx.declaration()?.function_declaration() != null) {
            val identifier = (ctx.parent.parent.parent as
            ↪  Swift5Parser.Class_declarationContext).class_name().text
            val classModel = environment.get(identifier,
            ↪  SymbolType.CLASS, null) as SwiftClassModel?

            if (classModel?.implementedInterfaces != null &&
            ↪  classModel.functions != null) {
                val functionName =
                ↪  ctx.declaration().function_declaration()
                    .function_name().text
                for (protocol in classModel.implementedInterfaces) {
                    if (protocol.hasFunction(functionName)) return true
                }
            }
        }
        return false
    }
}
```

Tier III

For identifying a construct from Tier III, it was necessary to involve symbol tables additionally. The classes for identifying such constructs were instantiated and called upon similarly as for constructs of Tier II. However, the current state of the `Environment` was passed so that definitions from its accessible symbol tables could be called upon. During the analyzation process, the formerly created global `Environment` was dynamically amended with declarations of the current scope. Constructs that required the symbol tables included string- and array-specific functions, the overriding of an inherited member like a superclass or interface function and dependencies on formerly made declarations and their typing. Listing 9 (see page 61) shows the finder for identifying the overriding of a `protocol` function by a Swift class. As a first step, the corresponding class model of the function member was taken from the `Environment`. Then, it was checked if any of the implemented protocols included a function of the same name as the class's function member.

6.3 Tool Validation

The possible inaccuracy of CAT posed a great threat to the validity of the results when determining the constructs' popularity. However, a thorough test of CAT was outside the scope of this thesis. To still gain insight on its capabilities and limitations, 20 random files were chosen from each language's project sample pool and manually examined. Then, the results were compared to CAT's results. While such a small sample size is not sufficient for calculating reliable results regarding CAT's accuracy, some trends became visible nevertheless.

The overall precision of CAT when analyzing the Kotlin files amounted to 96.38%. However, vast differences were disclosed between the accuracy for the Tier I and II constructs and the Tier III constructs. The accuracy for Tier I constructs was 100%, while Tier II constructs were detected with an accuracy of 99.14%. Shortcomings of the Tier II detection mainly included contexts in which a relevant construct occurred that were previously unknown. Tier III constructs were discovered with an accuracy of only 20.83%. While interpreting symbol tables did work successfully in some cases and such constructs were recognized, the most obvious problem was a lack of information on declarations made in external dependency files. Naturally, Android apps make use of a multitude of classes, interfaces, and functions from the Android SDK. This was most evident in the overriding of functions from a superclass, a construct most often not found by CAT due to the extension of an unknown superclass. Still, as for constructs of Tier I and II, all constructs that were recognized were true positives.

When analyzing the Swift files, CAT achieved an overall accuracy of 63.5%. Tier I precision was noted at 100% again, while Tier II precision amounted to 94.18%. Constructs belonging to Tier III were recognized with an accuracy of 49.47%. While still struggling with dependencies outside the project and typing that could not be determined definitely, CAT profited from including dependency files from CocoaPods and Carthage that existed within the projects. Like for the Kotlin side, no false positives were discovered by CAT.

In conclusion, it was assumable that the results for Tier III constructs were more unreliable than for the Tier I and Tier II constructs. Being limited to declarations made inside

the project poses a weakness for the accuracy of CAT and as a consequence more Tier III constructs possibly existed in the projects than determinable by the tool.

6.4 Summary

This chapter described the Construct Analyzer Tool (CAT) for automatically detecting the constructs that were unsupported by one or more of the considered transpilers in the Kotlin and Swift project sample pools (see Chapter 5). The tool was implemented as an CLI tool written mainly in Kotlin, taking both Kotlin and Swift projects as input. For each project, it would produce a CSV-File listing the occurrence of each relevant construct in the project. However, it was likely that not every file would be parsable by CAT, either due to incorrect input or because of CAT itself. In particular, the grammar files for Kotlin and Swift dictated the allowed input characters and language coverage. In some edge cases, the structure of the file resulted in such inefficient lexical analysis or parsing that a timeout was required to prevent the analyzation of all projects of becoming infeasible. This possibly led to some files being excluded by the tool despite being syntactically and semantically correct. In conclusion, being **highly dependent on the given grammar files** presented a limitation of CAT.

Constructs were identified by analyzing the parse trees generated by lexer and parser classes created by ANTLR. However, comments were omitted in the parsing process, therefore **excluding the recognition of any comment related constructs** from CAT's capabilities and presenting another limitation of the tool. The core parser functions were extended to create models for language components such as variables and classes. These models were saved to symbol tables, so their properties could be called upon when necessary. The symbol tables were implemented following the concept of chained symbol tables presented by Aho et al. [Aho07], allowing for block scoping to be represented. To include global declarations, a global environment including such declarations in a symbol table was created prior to the construct recognition. For finding constructs within the parse tree, its relevant nodes were visited by extending ANTLR's corresponding utility classes. In general, constructs were divided into three tiers:

- **Tier I**: Represented by a single node
- **Tier II**: Required analyzing other nodes in proximity
- **Tier III**: Required the symbol table(s)

For revealing general shortcomings of CAT, 20 files from each the Kotlin and Swift sample pools were manually analyzed and then compared to CAT's results. While constructs of Tier I were identified with an accuracy of 100%, Tier II constructs were still recognized reliably with an accuracy of 99.14% for Kotlin and 94.18% for Swift. However, constructs of Tier III were only recognized in 20.83% of the Kotlin cases and 49.47% of the Swift cases. When building a symbol table, CAT only included Kotlin or Swift files made within the project, i.e., excluding Java and ObjectiveC files and external dependencies. However, missing information on declarations made there greatly impacted the accuracy of identifying Tier III

constructs, like *Overriding a Superclass Function*. Therefore, **not including declarations from Java/ObjectiveC files and external dependencies** poses another limitation of CAT, affecting especially the detection accuracy of Tier III constructs. Still, no false positives were found by CAT when comparing its results to the manually evaluated results. Ultimately, its overall accuracy for Kotlin was calculated to be 96.38% and 63.5% for Swift, although a sample size as small as previously described does not yield a reliable representation of the tool's general accuracy.

Chapter 7

Results

This chapter summarizes how the research questions *RQ1-RQ3* introduced in Chapter 1 were answered by the experiments conducted within this thesis. Firstly, the results regarding the experiments for evaluating construct support (see Chapter 4) are shown for answering *RQ1 (From a set of basic constructs featured in the input programming language, which are supported by the transpilers?)* and *RQ2 (Does the output code generated by the transpilers follow the language's style guidelines?)*. Secondly, the construct occurrence found by CAT (see Chapter 6), used on the project sample pools acquired with the methodology of Chapter 5, is reported. This provides the groundwork for answering *RQ3 (How does the popularity of a construct unsupported by a transpiler affect its applicability?)* by presenting the frequency of use of the considered constructs in accordance to *RQ3.1 (How popular is a certain construct in practice?)*. Lastly, the final evaluation of the transpilers and therefore this thesis' answer to *RQ3* is described by considering the average occurrence of a transpiler's unsupported constructs within each project normalized with that project's **logical lines of code (LLOC)**.

7.1 Construct Support

In an attempt to answer *RQ1* and *RQ2*, this section reveals Gryphon's, Kotlift's, SequalsK's and SwiftKotlin's support of the constructs highlighted in Chapter 4. For this purpose, the results are presented in tabular form, showing both a construct's support by a transpiler and the accordance to style guidelines of supported constructs.

A construct is marked as working when the corresponding output code is both compilable and passes the corresponding unit test (✓). Non-working constructs did not fulfill those requirements (✗). Since in almost all cases a non-compilable translation was the cause of the error, no symbolic distinction was made between not compilable and not passing the unit test. Furthermore, if the translation was inadequate due to a failed unit test, manual correction would often require the same effort as for comparable non-compiling translations. Listing 10 (see page 66) illustrates this claim and shows an excerpt of a function making use of Kotlin's `when`. Please consider `items: List<String>` to be a parameter originally passed to the function.

Listing 10 Construct translation failing unit test.

```
// Original Kotlin code
...
var str = "Nothing to say."
when {
    "orange" in items -> str = "juicy"
    "apple" in items -> str = "apple is fine"
}
...

// SequalsK Swift output code
...
var str = "Nothing to say."
...
```

Listing 11 IntelliJ IDEA preferred spelling suggestion.

```
// Original Swift code
let appleSummary = "I have \(apples) apples."

// Gryphon Kotlin output code
internal val appleSummary: String = "I have ${apples} apples."

// Suggested spelling by the editor
internal val appleSummary: String = "I have $apples apples."
```

Since SequalsK most likely does not support `in`-expressions in `when`-statements yet, it completely neglects the construct while the rest of the code is still translated. Inserting a suitable `switch`-statement would probably require about the same effort as correcting a non-compilable `switch`-statement. Usually, unsupported constructs were just ignored by the transpiler and adopted to the output code literally. For Gryphon, some unsupported constructs resulted in a transpiling error. Gryphon was configured to still continue the translation process and would therefore insert `<<Error>>` at the corresponding position in the output code.

Overall, all transpilers achieved at least good results regarding the readability of their validly translated output code. The manual evaluation by a person with little to average experience in Kotlin and Swift yielded understandably formatted output code. Nevertheless, translations not meeting the style guidelines imposed by IntelliJ IDEA or SwiftLint were noted in the table (✻). However, the changes those warnings suggested were oftentimes only a minimal improvement on readability. This is illustrated by Listing 11.

For clarity, the constructs were divided into categories, although multiple constructs were occasionally grouped in the same row for saving space (✓✗).

7.1.1 Kotlin-to-Swift Construct Support

Table 7.1 shows the construct support of Kotlift and SequalsK w.r.t. the 102 considered Kotlin constructs. Notably, Kotlift officially supports Kotlin version 1.0.1 and Swift version 2.2, while the test cases are based on the latest available versions of the respective introductory chapters representing Kotlin 1.6.21 and Swift 5.6. Any results that would have been different for older versions of the two languages were therefore marked with a footnote. SequalsK achieved good results, covering around 79% of the constructs. Nevertheless, Kotlift showed only sufficient support by considering around 54% of the constructs. While the support and negligence of constructs was distributed across all categories, both transpilers seemed to neglect string and collection-type functions to a high degree.

Although the poorer performance of Kotlift was partly due to its outdated language support, most of the deficits arose from constructs that were simply not considered. E.g., next to the collection constructors featured in the "Basic syntax" chapter of the Kotlin documentation, `arrayListOf` and `arrayOf` were considered additionally since Kotlift supported none of the others. However, while Kotlift supports `arrayListOf`, despite it being introduced in Kotlin 1.1, it ignores `listOf` available since Kotlin 1.0.

W.r.t. the compliance of code style conventions, both transpilers showed very good results for meeting SwiftLint's requirements. The few warnings that occurred were limited to incorrect formatting and redundant keywords.

Table 7.1: Findings for Kotlift and SequalsK.

Construct	Kotlift	SequalsK
Comments and Printing		
End-of-line/Block/Nested Comments	✓✓✓	✓✓✓
Print Line/Inline	✓✗	✓✗
Simple Values and Typing		
Variables and Constants	✓✓	✓✓
Explicit Typing (Int/Double)	✓✓	✓✓
Type Casting (String→ Int/Int→ String)	✗✗	✗✓
Implicit Typing	✓	✓
Type Handling without Initialization	✓	✓
Top Level Declaration	✓	✓
Deferred Assignment	✓	✓
Type Checking With `is` and `!is`	✗✗	✗✗
Automatic Casting	✗	✗
Strings		
String and Number Concatenation	✗	✗
Interpolating One Variable/Expression	✓✓	✓✓
Multiline String	✓	✗

Table 7.1: Findings for Kotlift and SequalsK.

Construct	Kotlift	SequalsK
Replacement/Starts With/Uppercase	✗✗✗	✗✗✗
Arrays and Maps		
Array With `listOf`/`mutableListOf`	✗✗	✓✓
Array with `arrayListOf`/`arrayOf`	✓✗	✗✓
Map With `mapOf`/`mutableMapOf`	✗[1] ✗[1]	✓✓
Initializing Empty Array/Map	✗✗	✓✓
Overwriting Array/Map With Empty	✗✗	✓✓
`in` Operator	✗	✗
Array Filter/Map/Sort By	✓✓✗	✓✓✗
Classes and Instances		
Class Definition With Property/Function	✓✓	✓✓
Empty Class Definition	✗	✓
Superclass/Subclass Definition	✓✓	✓✓
Overriding Superclass Function/Property	✓✗	✗✓
Primary Constructor	✓	✓
Secondary Constructor	✗	✓
Property Involving Other Properties	✗	✓
Property `get` and `set`	✓	✓
Instantiating Object (With Prop.)	✓✗	✓✓
Accessing Properties/Functions	✓✓	✓✓
Enumeration and Data Classes		
Enumeration (With Function)	✓✓	✓(✻) ✓(✻)
Enumeration With Raw Values	✗	✓
Shorthand Enumeration Accessing	✓	✓
`Data Class`	✗	✓
Interfaces and Extensions		
Interface Definition	✗	✓
Class/Data Class Interface Implementation	✓✗	✓✗
Overriding Interface Function/Property	✓✗	✓✓
Extension Properties	✗	✓
Functions and Lambdas		
Function With Parameters and Return	✓	✓
Function Call (With Arguments)	✓✗	✓✓
Function Body as Expression	✗	✓
Return Void/Unit/Function	✓✓(✻) ✓	✓✓✓
Function as Argument	✓	✓
Assign Function to Variable	✗	✗
Nested Function	✓	✓
Lambda With Return Type	✗	✗
Lambda With Omitted Return Type	✗	✓

Table 7.1: Findings for Kotlift and SequalsK.

Construct	Kotlift	SequalsK
Control Flow		
Iterating Array (With Index/`forEach`)	✓✓✓	✓✓✓
Iterating Map	✓	✓
While/Do-While	✓✗	✓✓
`when` statement	✓(✻)	✓(✻)
Boolean Expression in Typed `when`	✗	✓
If-Else	✓	✓
Range		
Within/Out of Range	✗✗	✗✗
Iterating Over Range	✓	✓
`step`	✗	✓
Optionals		
Optional Value/Return	✓✓	✓✓
`?.let` Structure	✗	✓
Smart Cast After Null Check (One Var./Conjunction)	✓✗	✗✗
Default Value for Optional	✓	✓
Property Accessing for Optional	✓	✓
Error Handling and Generics		
Throwing/Catching an `Exception` Type	✗✓	✓✓
Variable Assignment With `try`	✗	✗
Generic Function	✓	✓
Misc.		
`++/--`	✗[2]	✓
Total	55/102	80/102

[1] Supports initialization with `Pair`, which was deprecated in Kotlin 1.1.

[2] `++/--` worked until Swift 3 (deprecated in Swift 2.2).

7.1.2 Swift-to-Kotlin Construct Support

The results regarding the Swift construct support of Gryphon, SequalsK and SwiftKotlin are shown in Table 7.2 (see pages 70-72). In general, Gryphon and SequalsK showed good results by supporting ∼74% of the considered constructs. SwiftKotlin proved slightly less mature, achieving only satisfactory results by considering ∼68% of the constructs. Like for the Kotlin to Swift translation, a lack of support for string- and collection-type functions was noticeable. Constructs not directly translatable to Kotlin corresponding to Schultes' fourth category regarding translation complexity [Sch21] (see Chapter 3.2) like *Self-Mutating Extensions* or *Addressing Closure Parameters By Number* were oftentimes ignored or led to an error in the case of Gryphon.

Although all transpilers were able to translate array and dictionary definitions to valid Kotlin code, it is noteworthy that those translations might still lead to not compilable output. As visible in Table 7.1 (see pages 67-69), various array and map types exist in Kotlin for implementing mutable and immutable collection types. I.e., if a collection type in Swift was to be translated to an immutable collection type in Kotlin, although that collection would be mutated later, a compilation error would occur. This problem is a.o. visible in SwiftKotlin, which consistently translates to Kotlin's immutable collection types. Likewise, errors regarding typing may also arise when an immutable collection is expected, but the translation resulted in a mutable collection type. This may still occur for SequalsK's approach of consistently translating to Kotlin's mutable collection types. For this reason, Gryphon proposes using its own `MutableMap`/`MutableList` type defined in its support file. Otherwise, Gryphon consistently translates to immutable collection types like SwiftKotlin.

For their validly translated constructs, SequalsK showed very good compliance w.r.t. the official Kotlin code style conventions and Gryphon and SwiftKotlin showed good compliance.

Table 7.2: Findings for Gryphon, SequalsK and SwiftKotlin.

Construct	Gryphon	SequalsK	SwiftKotlin
Comments and Printing			
End-of-line/Block/Nested Comments	✓✗✗	✓✓✓	✓✓✓
Print Line/Inline	✓✓	✓✓	✗✗
Simple Values and Typing			
Variables and Constants	✓✓	✓✓	✓✓
Explicit Typing (Int/Double)	✓✓	✓✗	✓✗
Type Casting (String→ Int)	✓	✗	✗
Implicit Typing	✓(✻)	✓	✓
Type Handling without Initialization	✓	✓	✓
Top Level Declaration	✓	✓	✓
Deferred Assignment	✓	✓	✓
Type Checking With `is`	✓	✗	✓
Strings			
String and Number Concatenation	✓	✗	✗
Interpolation	✓(✻)	✓(✻)	✓(✻)
Multiline String	✓	✗	✓
Replacement/Starts With/Uppercase	✗✗✓(✻)	✗✗✗	✗✗✗
Arrays and Dictionaries			
Array	✓	✓	✓
Dictionary	✓	✓	✓
Initializing Empty Array/Dictionary	✗✗	✓✓	✓✓(✻)
Overwriting Array/Dictionary With Empty	✓✓	✓✓	✓✓
In Collection	✓	✗	✓
Array Filter/Map/Sort By	✓✓✗	✓✓✗	✓✓✗
Classes and Instances			
Class Definition With Property/Function	✓✓	✓✓	✓✓

Table 7.2: Findings for Gryphon, SequalsK and SwiftKotlin.

Construct	Gryphon	SequalsK	SwiftKotlin
Superclass/Subclass Definition	✓✓	✓✓	✗✓(✻)
Overriding Superclass Function/Property	✓(✻) ✓(✻)	✓✓	✗✓
Initializer	✓(✻)	✓	✓(✻)
Deinitializer	✗	✗	✗
Property `get` and `set`	✓	✓	✓
Property `willSet`	✗	✓	✓
`willSet` Accessing Other Variables	✗	✗	✗
Instantiating Object (With Prop.)	✓✓	✓✓	✓✓
Accessing Properties/Functions	✓✓	✓✓	✓✓
Enumeration and Structures			
Enumeration (With Function)	✓✓	✓(✻) ✓(✻)	✓✗
Enumeration With Raw Values	✓	✓(✻)	✓(✻)
Enumeration Type From Raw Value	✓	✓	✓
Enumeration With Associated Value	✗	✗	✓(✻)
Shorthand Enumeration Accessing	✓	✗	✗
`Struct`	✓	✓	✓(✻)
Protocols and Extensions			
Protocol Def. (With Mutating Function)	✓✓	✓✓	✓✗
Class/Struct Protocol Implementation	✓✗	✓✗	✓✗
Overriding Protocol Function/Property	✗✗	✓✓	✗✗
Extension	✓	✓	✓
Self-Mutating Extension	✗	✗	✗
Functions and Closures			
Function With Parameters and Return	✓	✓	✓(✻)
Function Call (With Arguments)	✓✓	✓✓	✓✓
Function Labels (Omitted)	✓✓	✓✓	✓✓
Omitting return keyword	✓	✓	✓
Return impl. Void/expl. Void/Function	✓✓✗	✓✓✓	✓✓✗
Multiple Function Returns	✗	✗	✗
Function as Argument	✗	✓	✗
Nested Function	✓	✓	✓
Closure With Return Type	✓(✻)	✓(✻)	✓(✻)
Closure With Omitted Return Type	✓(✻)	✓(✻)	✓(✻)
Addressing Parameters By Number	✗	✗	✗
Control Flow			
Iterating Array (W. Enumerated/`forEach`)	✓✗✓	✓✗✓	✓✗✓
Iterating Dictionary (With Placeholder)	✓✗	✓✗	✓✓
While/Repeat	✓✗	✓✓	✓✗
`switch` statement	✓	✓	✓
`where` in Typed `switch`	✗	✗	✗
If-Else	✓(✻)	✓	✓

Table 7.2: Findings for Gryphon, SequalsK and SwiftKotlin.

Construct	Gryphon	SequalsK	SwiftKotlin
Range			
Within/Out of Range	✓✓(✻)	✓✓(✻)	✓(✻) ✓(✻)
Iterating Over Range	✓	✓	✓
Iterating Over Progression	✗	✓	✗
Optionals			
Optional Value/Return	✓✓(✻)	✓✓(✻)	✓✓(✻)
`if let` Structure	✓	✓	✓
`as?` in `if let` structure	✓	✗	✓
Default Value for Optional	✓	✓	✓
Property Accessing for Optional	✓	✓	✓
Error Handling and Generics			
Error Enum	✓	✗	✗
Throwing/Catching an `Error` Type	✓✓	✓✓	✓✗
Optional Conversion With `try?`	✗	✓	✓(✻)
`defer`	✓	✗	✗
Generic Function	✓	✓	✓
Generic Enum	✗	✗	✗
Requirements for Generic	✗	✗	✗
Total	77/104	77/104	71/104

7.2 Construct Popularity

Before being able to determine how the lacking support of certain constructs affects a transpiler's applicability and therefore answering *RQ3*, the frequency of use of the constructs unsupported (or constructs comparable to them) has to be analyzed in accordance to *RQ3.1*. For this purpose, this section shows the results of using CAT on the sample pools of Kotlin and Swift projects (see Chapter 5).

Notably, not all files were parsable by CAT due to faulty syntax or unsupported input characters, timeout restrictions or because they utilized grammar rules not covered by the ANTLR grammar used. However, next to possibly containing Tier I and Tier II constructs, their neglection also impacted the recognition of constructs belonging to Tier III, as declarations made in those files were not added to the global symbol table. For each programming language, the following further elaborates on the number of files ultimately considered. Subsequently, CAT's results are presented by discussing the percentage of project's containing a certain construct at least once. Afterwards, the occurrence of the considered constructs in each project, normalized with the LLOC of that project, is described. Within a project p, the normalized occurrence o_c of a certain construct c can be calculated as shown in Equation 7.1.

$$o_c(p) = \frac{count_c(p)}{lloc_p} \tag{7.1}$$

While $count_c$ is the total number of appearances that were counted for that construct, $lloc_p$ represents the total number of LLOC of the currently considered project. Notably, constructs that appeared multiple times on the same line of a file were counted just once. Furthermore, all files of a project that were determined to be non-parsable were excluded from $lloc_p$.

7.2.1 Kotlin

The original Kotlin sample pool included 7,417 projects with a total of 640,231 files. However, out of those files, 33,118 from 164 projects were non-parsable due to the aforementioned reasons. More precisely, it was discovered that many files were not parsable because they included unsupported input characters in the code, e.g., Chinese characters in variable and function names. This offers a possible explanation why a relatively small amount of projects included non-parsable files, as those characters were likely to be used repeatedly within the project.

Figure 7.1 (see page 74) visualizes the percentage of projects including a certain construct at least once. Notably, the *Lambda With Omitted Return Type* construct occurred in ∼98% of the projects. In their study, Mateus and Martinez observed a similar result in the occurrence of *Lambas* generally, as they appeared in ∼95% of the applications from their sample pool [MM20]. The percentage of projects including a *Data Class* construct was also alike in the study of Mateus and Martinez (∼65%) and the study conducted for this thesis (∼69%). At the same time, these results differ from Zayat's results for *Data Classes* being included in 49% of the applications considered [Zay20]. However, differences to Mateus' and Martinez' study exist in the occurrence of *Automatic Casting*, i.e., implicitly casting to a specific type when that type has been previously verified in a condition. They found out that ∼65% of the applications considered by them featured such casts, while this study noted ∼39% of projects including the *Automatic Casting* construct. These differences may be funded in the different ways the construct is identified by CAT and by the tool implemented by Mateus and Martinez. For this thesis, an *Automatic Casting* construct had to contain an `is` check for a variable in the condition of an `if` and use the variable within the control structure's body. When comparing the application coverage of the *Type Checking With* `is` construct of ∼61% to Mateus' and Martinez' results for *Automatic Casting*, a more similar result becomes apparent indeed.

The construct used in most applications was *Function Call With Arguments* (∼99.9% of the projects). This result was to be expected, since calling functions with arguments is a basic building block of many programming languages. On the opposite side of the scale, the construct present in the least projects was `step` (∼1%). Likewise, the *Array Sort By* construct was also featured in only ∼1% of the projects. Admittedly, this construct was part of the Tier III constructs, which were assumed to frequently fall victim to false negatives (see Chapter 6.3). In fact, string and array functions were gathered at the lower and of the scale, with *String Replacement* being the most presented and appearing in ∼11% of the applications. *Overriding Superclass Function* was the most popular construct from Tier III, used in ∼31% of the projects.

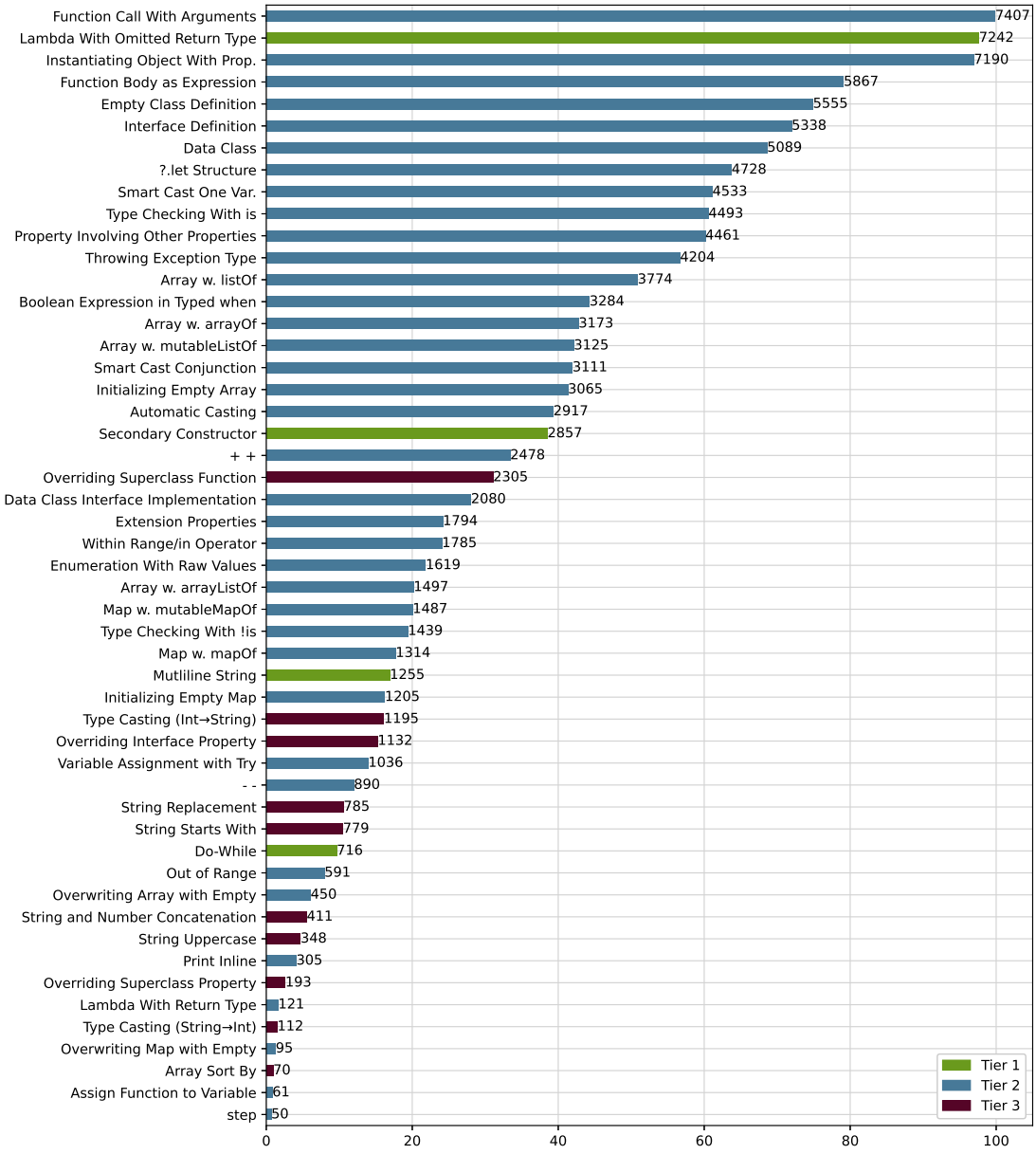


Figure 7.1: Percentage (and total number) of Kotlin projects including a certain construct.

Figure 7.2: Results for $o_c(p)$ for the Kotlin projects.

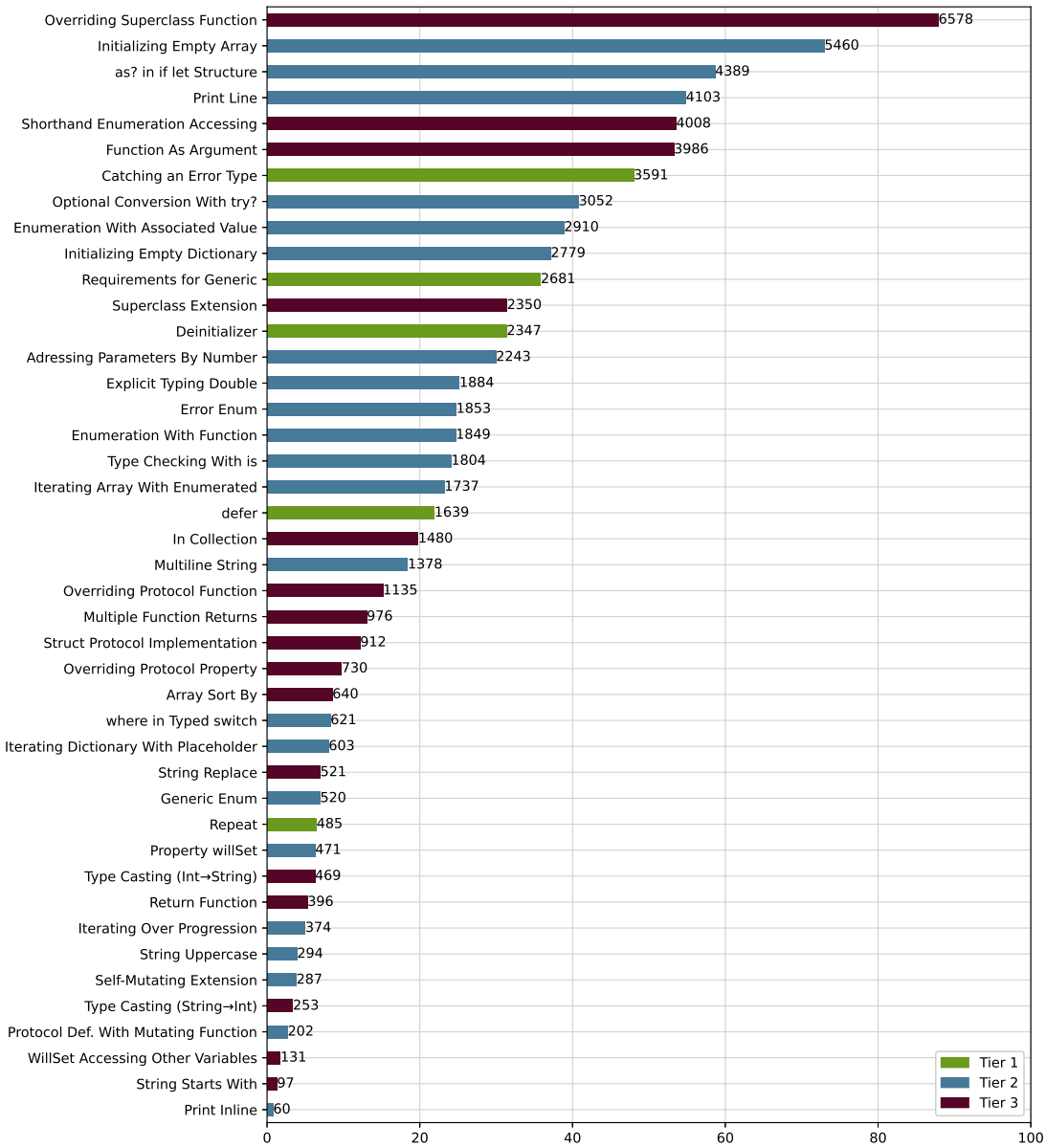


Figure 7.3: Percentage (and total number) of Swift projects including a certain construct.

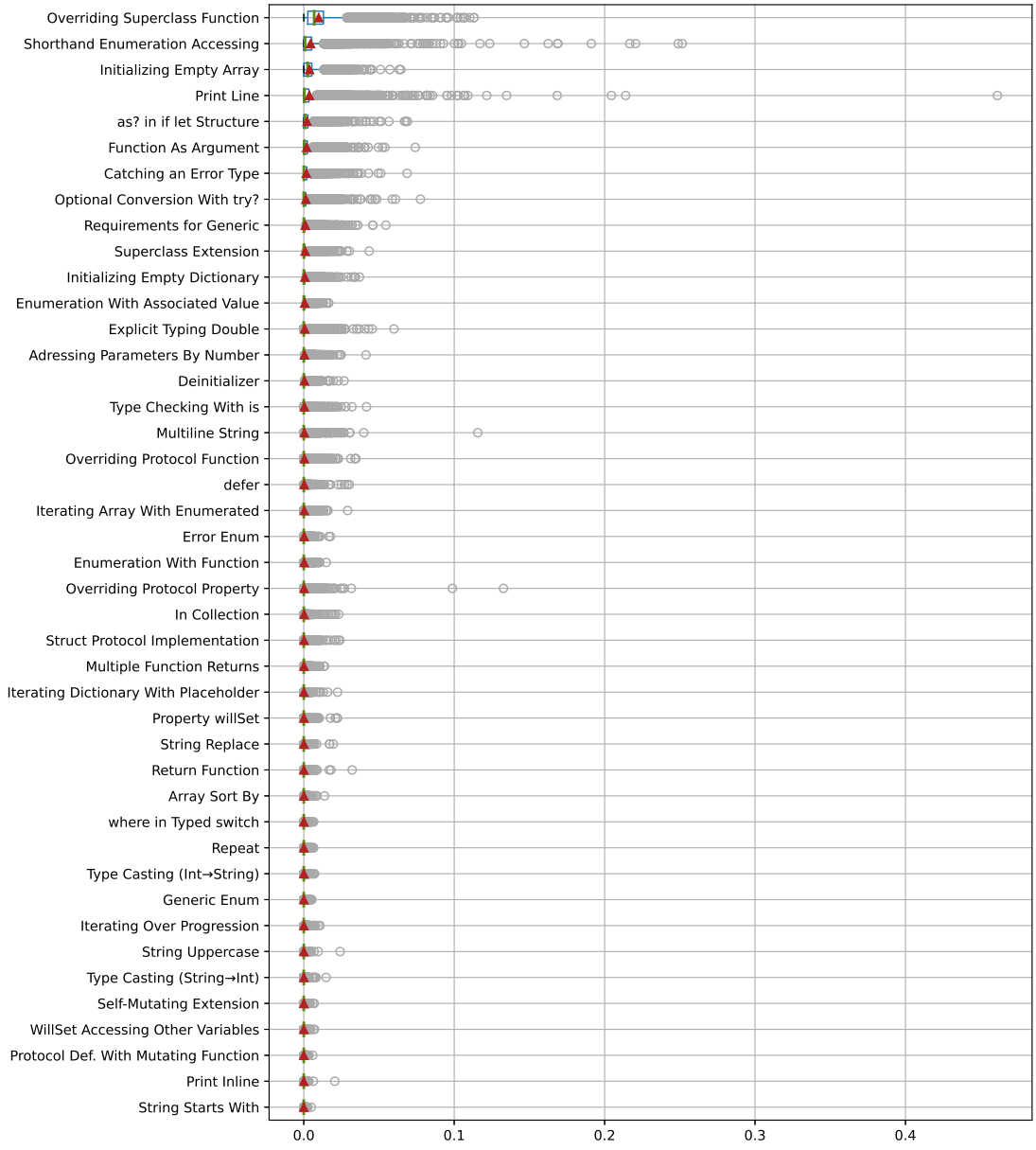


Figure 7.4: Results for $o_c(p)$ for the Swift projects.

Figure 7.2 (see page 75) depicts the occurrence of the considered constructs within the files, normalized with the project's LLOC. Most constructs appear quite seldom, with *Array Sort By* being the least frequent construct on average, identifiable only every ∼0.000003 lines. *Function Call With Arguments* appears the most frequent on average, being found every ∼0.16 lines.

7.2.2 Swift

The sample pool of Swift projects comprised 7,483 projects and a total of 429,512 files. Nevertheless, 18,888 files from 2,969 projects were not parsable by CAT due to the reasons described in the introduction to this section. It became especially apparent that the Swift ANTLR grammar used by CAT does not support the SwiftUI syntax, possibly excluding many files with actually correct Swift syntax.

Figure 7.3 (see page 76) depicts the percentage of projects including a certain construct at least once. Compared to the 2018 study of Casse et al., an increase in the `catch` clause from ∼39% to ∼48% became noticeable [CPCS18]. Furthermore, Casse et al. found only ∼5% of their considered projects featured an enum implementing the `Error` protocol, while ∼25% of the Swift projects analyzed for this thesis featured at least one enum implementing `Error`. This evolution hints that Swift developers possibly became more accustomed to the best practices for exception handling over time.

In contrary to the Kotlin results, the *Overriding Superclass Function* construct was largely detected in the considered projects despite it being a Tier III construct. In fact, it is the construct featured in most projects, i.e., ∼88% of the projects. The detection rate of this construct was possibly improved by the inclusion of third party dependencies from CocoaPods and Carthage in the global symbol table.

On the lower end of the scale, the *Print Inline* construct scored last place, with being featured in only ∼0.8%. This is possibly due to the unintuitive way of formulating this construct, as it requires passing an empty String to the `terminator` parameter of `print`. Quite possibly, printing inline is also less popular than the *Print Line* construct, which was present in ∼55% of the projects.

All considered string functions and the considered array `sorted` function were observed in <10% of the projects. However, as they belong to the Tier III category of constructs, it's quite possible not all their occurrences were recognized by CAT.

Figure 7.4 (see page 77) depicts the occurrence of the considered constructs within the projects, normalized with the project's LLOC. Noticeably, none of the constructs occur in great frequency within the projects. When sorted according to their average usage, the *Overriding Superclass Function* construct appeared most frequently every ∼0.0098 lines. The construct that was encountered least compared to the LLOC was *String Starts With* every ∼0.000005 lines.

7.3 Transpiler Evaluation

While the percentage of projects using a certain construct is a metric suited to assess the overall popularity of a construct within a programming language's community, developers

aiming to use a transpiler are more likely interested in how many lines they have to manually correct in the transpiler's output. Therefore, the average occurrence of a construct normalized with the LLOC from its origin project is a more fitting metric for evaluating how an unsupported construct affects the development workflow when using a transpiler.

The average occurrence a_c of a certain construct c can be calculated as depicted in Equation 7.2, by dividing the sum of all values of o_c (see Equation 7.1, page 72) for all projects by the total number of projects n_p of the sample pool of the currently considered language. The entirety of those projects is described by the set P, with $P = \{p_1, p_2, p_3, ..., p_{n_p}\}$.

$$a_c = \frac{\sum_{i=1}^{n_p} o_c(p_i)}{n_p} \tag{7.2}$$

The entirety of all average occurrences of all constructs unsupported by one or more transpilers is described by the set A of either Kotlin or Swift, with $A = \{a_{c_1}, a_{c_2}, a_{c_3}, ..., a_{c_m}\}$ and m being the total number of those constructs. For the Kotlin-to-Swift transpilers, $m_{KotlinConstructs}$ is 51 and for the Swift-to-Kotlin transpilers, $m_{SwiftConstructs}$ is 43.

Equation 7.3 incorporates the elements of A of a transpiler for calculating the score W, indicating a transpiler's applicability despite its unsupported constructs. The result of W ranges from 0 to 1, with 1 representing the highest possible score for applicability and 0 representing the lowest possible score. While A contains the average occurrence of every construct considered, the set U only includes the values a_c of constructs the currently considered transpiler does not support.

$$W = 1 - \frac{\sum U}{\sum A} \tag{7.3}$$

This section will describe the results and implications of W for the Kotlin-to-Swift (W_{KS}) and Swift-to-Kotlin transpilers (W_{SK}).

7.3.1 Kotlin-to-Swift Transpilers

Table 7.3 suggests that large differences exist between SequalsK and Kotlift in terms of the average number of lines of code that need to be changed after the translation process. While SequalsK achieved a W_{KS} score of 0.964, Kotlift scored only 0.013. However, it was to be expected that Kotlift would be less applicable, since it supported less constructs than SequalsK. Even more so, the constructs not considered by Kotlift included, among others, the three constructs with the highest average occurrence, namely *Function Call With Arguments*, *Lambda With Omitted Return Type* and *Instantiating Object With Properties*, all of which SequalsK supported.

Table 7.3: Results for W_{KS} for the Kotlin-to-Swift transpilers.

	Kotlift	**SequalsK**
W_{KS}	0.013	0.964

7.3.2 Swift-to-Kotlin Transpilers

As seen in Table 7.4, the least amount of manual editing of the output code w.r.t. the considered constructs is likely to be required for Gryphon compared to the other Swift-to-Kotlin transpilers. Still, Gryphon's result of 0.680 for W_{SK} is close to the result of SequalsK of 0.622, although the transpilers show quite a few differences in the constructs they support. However, in contrary to Gryphon, SequalsK does not support the construct with the second-highest average occurrence, *Shorthand Enumeration Accessing*, and the fifth-highest, `as?` *in* `if let` *Structure*. Although Gryphon does not consider the third-highest ranking construct w.r.t. average occurrence, *Initializing Empty Array*, it otherwise covers four of the five constructs with the highest occurrence, while SequalsK covers three.

Despite considering only a few less constructs than Gryphon and SequalsK, SwiftKotlin places last by far with a W_{SK} score of 0.263. Notably, SwiftKotlin disregarded some constructs leading w.r.t. their average occurrence, like *Overriding a Superclass Function* and *Print Line*. Meanwhile, both of these constructs were supported by Gryphon and SequalsK.

Table 7.4: Results for W_{SK} for the Swift-to-Kotlin transpilers.

	Gryphon	**SequalsK**	**SwiftKotlin**
W_{SK}	0.680	0.622	0.263

7.4 Summary

In summary, this chapter presented the results for the underlying research questions *RQ1-RQ3* of this thesis.

To answer *RQ1 (From a set of basic constructs featured in the input programming language, which are supported by the transpilers*?) and *RQ2 (Does the output code generated by the transpilers follow the language's style guidelines?*), the study described in Chapter 4 and previously conducted in the preliminary work for this thesis [SS22] was repeated with the newer versions of Gryphon, Kotlift and SequalsK in addition to the SwiftKotlin transpiler. Similar results to before were acquired, classifying **Gryphon and SequalsK (for both directions of translation) as more mature with a good coverage** of the considered constructs, while **SwiftKotlin only achieved satisfactory results** and **Kotlift only sufficient**. Nevertheless, all transpilers were affected by shortcomings spread across all categories of constructs, including fundamental differences between the languages and constructs that were simply not considered alike. W.r.t. code maintainability, the manual evaluation yielded no outlandish formatting or other obstacles to code readability. At the same time, all transpilers generally followed most of the acknowledged style conventions, with **Kotlift and SequalsK (for both directions of translation) achieving very good results** and **Gryphon and SwiftKotlin achieving good results**.

As a next step for ultimately dealing with *RQ3*, the results for *RQ3.1 (How popular is a certain construct in practice?*) were discussed. When being used on the app project

sample pools (see Chapter 5), the results of CAT regarding the Kotlin files suggested that the *Function Call With Arguments* construct was found in most (∼99.9%) of the projects, in addition to being the construct with the highest average usage every ∼0.16 lines. At the same time, `step` was found in the fewest projects (∼1%). W.r.t. average line occurrence, *Array Sort By* was found the least, appearing only every ∼0.000003 lines on average. In general, the results for the Kotlin construct usage is possibly largely affected by the exclusion of declarations made in external dependencies and therefore worsening the accuracy for Tier III constructs.

On the contrary, the results for the Swift sample pool did not only propose that *Overriding a Superclass Function*, a Tier III construct, was featured in the most projects (∼88%) but that it had also the highest average occurrence, appearing every ∼0.0098 lines. In this case in particular, the precision of CAT profited from the inclusion of the CacaoPods and Carthage dependencies existing inside the project. While *Print Inline* was found in the least projects (∼0.8%), *String Starts With* had the fewest average occurrences, being identified only every ∼0.000005% lines.

Finally, *RQ3 (How does the popularity of a construct unsupported by a transpiler affect its applicability?)* was examined by determining the score W suggesting a transpilers applicability despite its unsupported constructs, ranging from 0 to 1 with 0 being the worst possible outcome and 1 being the best possible outcome. It was calculated by using the previously determined results regarding construct occurrence normalized with the projects' LLOC. I.e., W represented an indication of the manual effort required to edit a transpiler's output code.

For the Kotlin-to-Swift transpilers, **SequalsK proved to be far more applicable than Kotlift**, scoring 0.964 while Kotlift only achieved 0.013. Regarding the Swift-to-Kotlin transpilers, Gryphon achieved a score of 0.680 and SequalsK of 0.622. Therefore, **Gryphon's output is likely to require slightly less manual editing than SequalsK's** w.r.t. the considered constructs. **SwiftKotlin scored last by far** with a value of 0.263, although it supported only six constructs less than SequalsK. All in all, these results confirm the previously made assumption that the impact different constructs have on the applicability of a transpiler greatly varies.

Chapter 8

Summary and Future Work

To conclude this thesis, the upcoming chapter summarizes the key aspects of each chapter to remind the reader of the central themes of this thesis and present its scientific contribution. However, as this thesis could not cover all aspects of its underlying problems, the following also presents recommended directions for future work, including improvements on the proposed solutions from this thesis.

8.1 Summary

This thesis provided insight into the current state of the art in Kotlin-to-Swift and Swift-to-Kotlin transpilers in the field of cross-platform development for Android and Apple OS by examining transpiler support of a set of basic constructs. Moreover, this study was given more practical relevance by also including the occurrence of unsupported constructs in real programming projects in the final evaluation of the transpilers.

The demand for cross-platform development tools like transpilers arises especially from the current situation of the mobile OS market. There, Google's Android and Apple's iOS are the dominant players, holding almost the entire market share together [Sta22b]. However, when developing third-party applications for these platforms, developers have to use a certain programming language that differs between those OS, in addition to a designated SDK contributed by either Google or Apple. This situation created various tools for cross-platform mobile development over the years. They usually pursue the goal of reducing development effort by providing concepts that only require the development of one code base. For most of them, this is achieved by employing a layer of abstraction to run non-native code on the device. However, apart from losses in performance [MLL+18, QGZ16] and possibly failing to accomplish the "look and feel" the OS is known for [RS12], this creates a dependency on the continuous development of the tool to stay up to date with new releases of the OS [VGM20, Sch21]. An alternative approach would be using a transpiler, translating from one platform's native programming language to the other, making the output code bases independently maintainable. Notably, most transpilers focus on the translation of the pure language, i.e., neglecting platform-specific and other libraries. Consequently, when considering that an app is designed according to the MVC pattern, only the model with the

business logic parts are translatable.

For the primary native languages of Android and Apple OS, Kotlin and Swift, the four transpilers Kotlift [Stu20], translating from Kotlin to Swift, Gryphon [Ven22a] and SwiftKotlin [Oll20], translating from Swift to Kotlin, and SequalsK [Sch21], supporting bidirectional translation, can be found online. The transpilers work on the underlying assumption that Kotlin and Swift are translatable to each other. In fact, Kotlin and Swift share general common traits, like being modern programming languages that are both characterized by their less verbose and safer syntax compared to their native programming language predecessors, Java and ObjectiveC. Moreover, their syntactical similarity is explored in numerous articles online [Oll16, Mec17], although more complex differences exist other than constructs that are either identical or replaceable. While some of them are still adaptable with a certain amount of effort, others are simply not translatable [Sch21].

However, to the best knowledge of the author, no other study examining the capabilities of Kotlin-to-Swift and Swift-to-Kotlin transpilers exists in the literature, besides the preliminary work done for this thesis [SS22]. Even more so, there appears to be a general lack of works on the suitability of transpilers and other source-to-source translation tools for Android and Apple OS cross-platform development. While various works introducing own implementations of such tools suggest that source-to-source translation might become an alternative approach to cross-platform development, their abilities are oftentimes only evaluated within the works presenting them. W.r.t. Gryphon, Kotlift, SequalsK and SwiftKotlin, all transpilers admit to deficits in language coverage while also stating different demands regarding the need for post-translation edits, therefore making it worthwhile to compare their capabilities.

Consequently, this thesis elaborated on the construct support study conducted in the preliminary work not only by repeating the study with the newest versions of the transpilers, but by adding SwiftKotlin to gain even more insight on the current state of the art. Like in the preliminary work, this study was founded on construct-based experiments based on the exemplary code snippets from the overview chapters of the Kotlin and Swift documentations, namely "Basic syntax" [Jet22a] and "A Swift Tour" [App22c]. Those were transformed into compilable test cases and manually translated to the respective other language for extending the test case sample pool. All in all, 102 distinct constructs were tested for Kotlin and 104 distinct constructs for Swift. Only if the translation of a construct was compilable and passed a corresponding unit test verifying its functionality, a construct was marked as supported by the transpiler. If the transpiler output failed to meet those requirements, the construct was marked as unsupported. In some cases, it seemed appropriate to test the transpiler on a simplified version of that construct, since the overview chapters sometimes exemplified a feature with a more complicated construct than necessary. Since the maintainability and therefore readability of the output code is one of the central advantages of using a transpiler as a cross-platform development tool, the compliance of the valid output code w.r.t. generally accepted style guidelines of the target language was also assessed using both manual evaluation and a linter. For Kotlin, the built-in Kotlin style guide from the IntelliJ IDE was used [Jet22c] and for Swift, the output code was checked using SwiftLint [Rea22], a third-party Xcode plug-in based on generally accepted Swift conventions.

Since the results from this study only yielded whether a construct was supported or

not, disregarding that some constructs are more frequently used than others by developers, therefore making it more cumbersome to manually correct them in the output code if they were unsupported, the study was extended to be more practice oriented by considering the popularity of the unsupported constructs. While various works on language adoption existed in the literature, even considering some constructs relevant to this thesis, no popularity metric could be found for all the constructs unsupported by one or more of the transpilers. Still, these related works provided insight on aspects like general developer experience, difficulties when incorporating Java-Kotlin interoperability [OTE20], the adoption of features introduced in Kotlin but not available in Java [MM20, Zay20], code smells in iOS applications [RP20] or the challenges of adopting Swift features vastly differing from ObjectiveC like *Optionals* and error handling mechanisms [RPE+16, CPCS18]. Nevertheless, this created the need to conduct an own study on construct usage within Kotlin and Swift projects from practice, which was conducted as a MSR study. As a basis for this study, it was necessary to put together a statistically relevant sized sample pool of Kotlin and Swift projects. It was assumable that an appropriate amount of open-source projects were residing on the version control website GitHub [Git22b]. However, since the official GitHub API [Git22a] did not offer the necessary means for filtering the results appropriately, GHS [DAB21] was used for filtering the projects according to the primary language used and their last commit. For Kotlin, a project had to have at least one commit after 29 October 2018, the release of Kotlin 1.3, so all constructs from the test case pool could have theoretically been implemented in the project. For Swift, 10 September 2019, the release of Swift 5.1, was the corresponding equivalent. Since the considered transpilers primarily target app projects, the sample pool was further reduced to Kotlin based Android and Swift based Apple OS projects. These were identified by looking for the *AndroidManifest.xml*-file in Kotlin and the *Info.plist*-file in Swift projects and by searching their code for statements signaling the implementation of an UI. Ultimately, the Kotlin sample pool consisted of 7,417 projects with a total of 640,231 files. Meanwhile, after excluding third-party dependencies located within the projects, the Swift sample pool consisted of 7,483 projects and 429,512 files.

The occurrence of the considered constructs within those projects was counted by using a tool specifically developed for this thesis, referred to as Construct Analyzer Tool (CAT). Identifying constructs was achieved by recognizing them within the parse tree of the code file. Generally speaking, a parse tree is a model representation of the directives from a piece of code, created by a parser. The input of a parser is a token stream, created by a lexer from the input file's character stream. Those tokens, in addition to rules on how they can be combined to form statements, are defined within the grammar of a language. For the implementation of CAT, ANTLR [Par22, Par12] was used for generating Kotlin and Swift lexers, parsers and various helper classes for traversing the parse tree. ANTLR is a commonly utilized tool in the literature for tasks requiring language recognition, e.g., transpilers [Nie16, AMT18, SHK+19, MME+20, Sch21, Dem15, RLMW14, SS21] or studies on source code analyzation [RBS13, AP21, SYCI18, LYB+19, APT20].

CAT was implemented as a CLI tool taking either a Kotlin or a Swift project as input and producing a CSV-file listing the occurrences of each construct as output. Files that were not parsable due to faulty syntax, unsupported input characters or grammar rules or due to an over excessive lexical analysis or parsing time were listed in a separate file. The complexity

of identifying a construct from the parse tree was divided into three tiers – while Tier I constructs were identifiable by the existence of a node in the parse tree, Tier II constructs required a look at the nodes in proximity to be surely recognized. Moreover, constructs belonging to Tier III required consulting symbol tables, which were initially built with global declarations before the analyzation process and then dynamically expanded as the parse tree was traversed.

For gaining insight on CAT's accuracy, 20 random files from different projects from each the Kotlin and the Swift sample pools were manually analyzed, and the results were then compared to CAT's. Notably, an unsatisfactory accuracy regarding Tier III constructs was detected for both Kotlin and Swift, largely due to the exclusion of declarations made in the projects' external dependencies. Nevertheless, a 100% accuracy for both Kotlin and Swift was observed for Tier I constructs. For Tier II, 99.14% of the Kotlin constructs were found and 94.18% of the Swift constructs. Nevertheless, no false positives could be found for neither language.

Ultimately, three main restrictions became visible for CAT: Firstly, it was highly dependent on the language grammar file used for generating the ANTLR classes, laying restrictions upon the allowed input characters and grammar rules as well as influencing parsing performance. Secondly, any comment related constructs had to be omitted from the analyzation, since comments were omitted from the generated parse tree. Lastly, any declarations made in external dependencies or Java-/ObjectiveC-files within the projects were excluded.

The final results on construct support unveiled that Gryphon and SequalsK (for both directions of translation) can be classified as more mature, achieving good coverage of the considered constructs. Meanwhile, SwiftKotlin showed only satisfactory results and Kotlift merely sufficient. The evaluation of the valid output w.r.t. readability revealed that all transpilers produced generally readable code, with Kotlift and SequalsK (for both directions of translation) achieving very good results and Gryphon and SwiftKotlin achieving good results regarding the compliance of style guidelines. The results of CAT used on the project sample pool revealed the differences between the constructs in terms of their popularity within the Kotlin and Swift sample pool projects. However, especially for Kotlin, Tier III constructs were generally recognized more seldom than constructs from the other tiers. This can likely be attributed to the exclusion of third-party dependencies from the analyzation process, since a better overall recognition for Tier III constructs could be achieved for the Swift projects including third-party dependencies from CacaoPods and Carthage. The impact of an unsupported construct on a transpiler's applicability was ultimately measured by its average occurrence, which was calculated by setting the number of lines it was found on within the project into relation with the total LLOC of the project. The average occurrence of all constructs unsupported by a transpiler was used to calculate the score W, indicating the required manual effort to correct any invalid translations of the considered constructs within the transpiler's output code.

W.r.t. the Kotlin-to-Swift transpilers, SequalsK proved to be far more applicable than Kotlift, scoring a value of 0.964 for W while Kotlift only achieved 0.013. Regarding the Swift-to-Kotlin transpilers, Gryphon achieved a W score of 0.680 and SequalsK of 0.622, while SwiftKotlin scored last with a value of 0.263 for W. Evidently, these results prove the hypothesis that the impact different unsupported constructs have on the applicability of a

transpiler varies indeed, since especially the Swift-to-Kotlin transpilers supported a similar amount of constructs.

8.2 Future Work

All in all, the results of this thesis present only a record of the current situation regarding Kotlin-to-Swift and Swift-to-Kotlin transpilers as well as the Kotlin and Swift programming languages. The transpilers are still under development, so their support of language constructs may increase over time. At the same time, Kotlin and Swift are constantly receiving new features – furthermore, constructs that were introduced recently might become more popular over time.

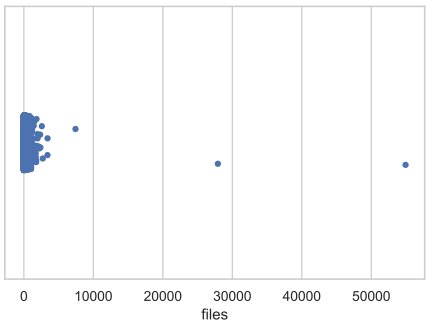
Admittedly, the tool developed for automatically recognizing constructs within a project's source files, CAT, is also left with room for improvement. In its early development stages, it was planned to include all constructs considered within the construct support study. Due to the limited scope of this thesis, this was ultimately scrapped. However, implementing the search for the supported constructs (and other constructs) would possibly provide more insight on construct usage that would ultimately benefit the transpilers' evaluation. Firstly, the popularity of constructs supported by the transpiler and those that were unsupported could be set into relation. Secondly, like proposed by Mateus and Martinez [MM20], certain constructs could be used for normalization. E.g., determining how many class definitions in Kotlin are *Data Classes* might put the importance of that construct in a different light than simply calculating its occurrence in relation to the LLOC of a project. When examining the tool's accuracy, it became quickly apparent that the recognition of constructs dependent on previously made declarations is unsatisfactory due to the exclusion of the project's external dependencies. At the same time, parsing is the most costly operation performed when analyzing a project and including large libraries would likely require a considerable amount of time. One possible solution for considering those dependencies, while still preventing the need to parse them fully for every project they are included in, would be by parsing frequently used libraries, like the `android` library, once and to save the resulting parse tree and import it when necessary. Admittedly, this possibly still results in inaccuracies due to different library versioning. Nevertheless, the existing implementations of construct finders not achieving 100% accuracy might still be generally improvable, especially w.r.t. finders not requiring symbol tables. Lastly, modifications to the used ANTLR grammars for Kotlin and Swift could possibly enhance the overall accuracy as well as CAT's performance. Several code files had to be excluded from analyzation, not only due to containing faulty syntax, but due to containing unaccepted input characters, grammar rules or simply taking an unfeasible amount of lexical analysis or parsing time. Changes to the grammar possibly fix those problems. Furthermore, CAT is currently not able to recognize different types of comments or comments per se. Perhaps either the grammar files could be configured to include those to the parse tree, or the cloc tool [Dan22] used to obtain the file number and LLOC of a project could be customized in a way that it could differentiate between various kinds of comments.

In addition to CAT, there are various ways how the study on transpiler constructs sup-

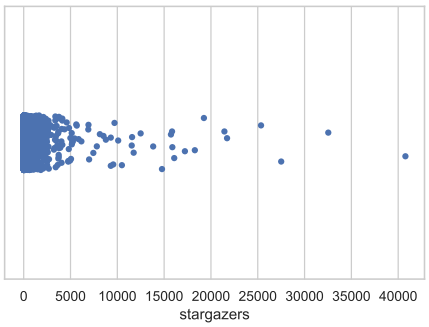
port itself could be extended. The most obvious improvement would be to consider more constructs. Although the constructs from the overview chapters of the Kotlin and Swift documentations were chosen on purpose to attain insight on the transpilers' support of general language aspects, they neither represent the language fully nor illustrate all varying ways that the features shown there can be implemented. While this study assessed the readability of the output code both by manual evaluation and by using a linter, another way of comparing it to human-written code would be by using BLEU [PRWZ02] or, more appropriately, an adapted version for programming code [RGL+20]. Furthermore, besides testing the transpilers on isolated constructs, using the transpilers on the model and business logic parts of Kotlin based Android and Swift based Apple OS applications would provide further insight on the suitability of Gryphon, Kotlift, SequalsK and SwiftKotlin for cross-platform development.

Appendix A

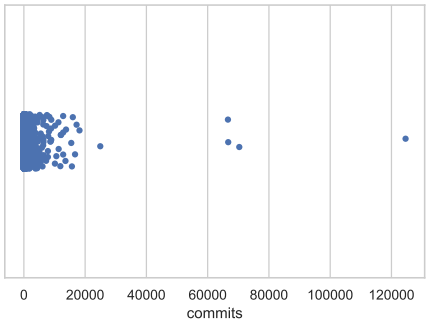
Project Sample Pool Metrics



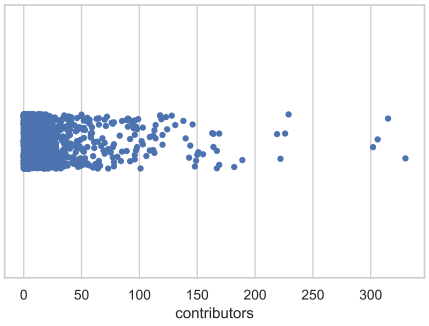
(a) Files per project.



(b) Stargazers per project.

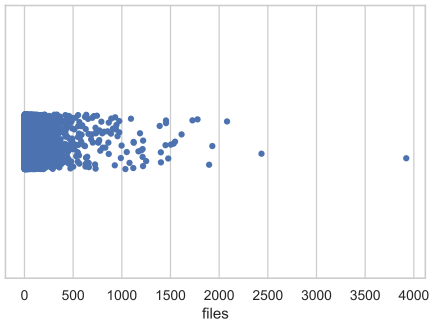


(c) Commits per project.

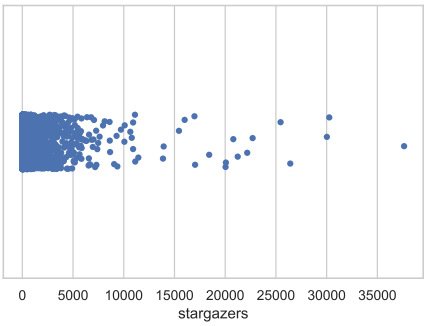


(d) Contributors per project.

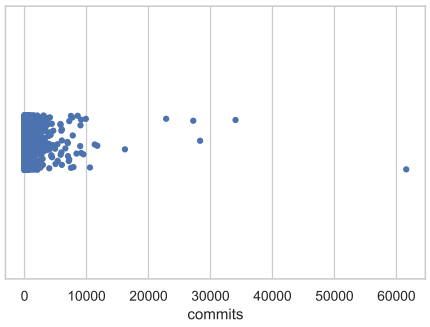
Figure A.1: Metrics for the Kotlin projects sample pool.



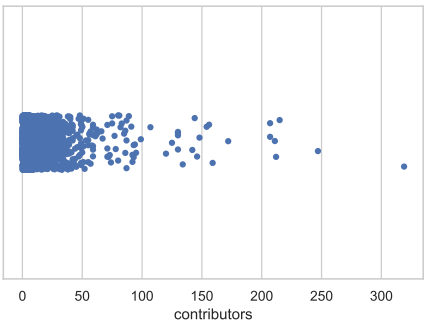
(a) Files per project.



(b) Stargazers per project.



(c) Commits per project.



(d) Contributors per project.

Figure A.2: Metrics for the Swift projects sample pool.

Appendix B

Contents of the Attached CD

1. Construct test case files and corresponding unit tests
2. CSV-files describing the Kotlin and Swift project sample pools
3. Source code of the Construct Analyzer Tool
4. Automation scripts
5. Online references
6. PDF-version of this thesis

Glossary

Apple OS Umbrella term for iOS and any of its variants used as operating systems for Apple devices. E.g., Apple Watches use watchOS and Apple TV uses tvOS. 5, 8, 18–20, 22, 28, 29, 40, 43, 44, 46, 48, 49, 83–85, 88

construct Shorthand for programming language construct. A programming language feature is implemented by a construct. i, ii, 3–6, 10, 16, 17, 23–29, 31, 32, 34, 36–40, 42, 43, 47–49, 51–53, 56, 59, 60, 62–67, 69, 70, 72, 73, 78–81, 83–88

feature Shorthand for programming language feature. Programming languages consist of various features, like different kind of loops or object-oriented features like classes. 3, 7, 8, 10, 22–27, 29, 31, 32, 34, 39, 41, 42, 53, 56, 84, 85, 87, 88

lexer Produces an output stream of tokens from an input stream of chars corresponding to a given grammar. 10, 11, 14, 15, 17, 28, 29, 53, 57, 59, 63, 85

linter Program for statical code analysis that flags problems with the code, like semantic/styntactical issues and stylistic violations. i, 36, 38, 84, 88

parse tree Tree structure representing the sentence of a given grammar. i, 5, 8, 11–17, 20–22, 27–29, 52, 53, 57, 59, 60, 63, 85–87

parser Produces a model, often a parse tree, from an input stream of tokens corresponding to a given grammar. 8, 10–15, 17, 21, 28, 29, 51, 53, 57, 59, 60, 63, 85

parser generator Tool for automatically creating a parser from a given grammar. Sometimes also includes lexer generation. 5, 7, 14, 17, 29, 53

regular expression Form of algebraic description of regular languages, often used for pattern matching strings. 10, 11, 14, 17, 22, 26

symbol table Data structure used by a language processor to note information relevant to identifiers, like their location in the source code, typing, etc. 13, 14, 17, 52, 53, 56, 62, 63, 72, 78, 86, 87

token Basic lexical unit in programming language parsing. 8, 10–14, 17, 53, 85

transpiler Program capable of translating source code from one high level programming language to another. Also referred to as transcompiler or source-to-source compiler/-translator. i, ii, 2–7, 15–24, 28, 29, 31, 32, 34, 36–39, 44, 51, 52, 63, 65–67, 70, 72, 79–81, 83–88

Acronyms

ANTLR ANother Tool for Language Recognition. 5, 7, 14, 15, 17, 19, 28, 29, 53, 57, 59, 63, 72, 78, 85–87

API application programming interface. 20–22, 25, 26, 40, 41, 45, 48, 85

BLEU Bilingual Evaluation Understudy. 21, 88

CAT Construct Analyzer Tool. i, ii, 6, 51–53, 56, 57, 59, 60, 62–65, 72, 73, 78, 81, 85–87, 91

CFG context-free grammar. 9, 10, 14, 17

CLI command line interface. 34, 47, 53, 63, 85

DFA determenistic finite automata. 10, 11, 13, 15, 17

GHS GitHub Search. 40–48, 85

IDE integrated development environment. 25, 36, 44, 84

LDA Latent Dirichlet Allocation. 25, 26

LLOC logical lines of code. ii, 65, 72, 73, 78, 79, 81, 86, 87

MSR mining software repository. 4, 40, 41, 44, 48, 51, 85

MVC model-view-controller. 2, 83

NFA non-determenistic finite automata. 10, 11

OS operating system. 1, 2, 29, 83

PDA pushdown automata. 13

RAM random-access memory. 57

REST representational state transfer. 40

SDK software development kit. 1, 62, 83

TCAIOSC Trans Compiler Android to IOS Conversion. 20, 21

UI user interface. 7, 8, 21, 25, 44, 45, 48, 85

Bibliography

[AGG+80] Paul F. Albrecht, Philip E. Garrison, Susan L. Graham, Robert H. Hyerle, Patricia Ip, and Bernd Krieg-Brückner. Source-to-source translation: Ada to Pascal and Pascal to Ada. *ACM SIGPLAN Notices*, 15(11):183–193, November 1980. ISSN 0362-1340. URL https://doi.org/10.1145/947783.948658.

[Aho07] Alfred V. Aho. *Compilers : principles, techniques, tools*. Boston [a.o.], 2nd edition, 2007. ISBN 0321547985.

[Ale22] AlexAtUni. GitHub API Search code. https://github.community/t/github-api-search-code/223992 (Accessed: 2022-05-25), 2022.

[AMT18] Kijin An, Na Meng, and Eli Tilevich. Automatic Inference of Java-to-Swift Translation Rules for Porting Mobile Applications. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 180–190, 2018. URL https://doi.org/10.1145/3197231.3197240.

[AP21] Sanjay B. Ankali and Latha Parthiban. Detection and Classification of Cross-language Code Clone Types by Filtering the Nodes of ANTLR-generated Parse Tree. *International Journal of Intelligent Systems and Applications*, 13(3):43–65, 2021. ISSN 20749058. URL https://doi.org/10.5815/ijisa.2021.03.05.

[App16] Apple Inc. The Swift Programming Language (Swift 2.2): A Swift Tour. https://web.archive.org/web/20160423163358/https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/GuidedTour.html (Accessed: 2022-05-25), April 2016.

[App22a] Apple Inc. Document Revision History — The Swift Programming Language (Swift 5.6). https://docs.swift.org/swift-book/RevisionHistory/RevisionHistory.html (Accessed: 2022-05-25), 2022.

[App22b] Apple Inc. Information Property List | Apple Developer Documentation. https://developer.apple.com/documentation/bundleresources/information_property_list (Accessed: 2022-05-25), 2022.

[App22c] Apple Inc. A Swift Tour. https://docs.swift.org/swift-book/GuidedTour/GuidedTour.html (Accessed: 2022-04-29), 2022.

[App22d] Apple Inc. Swift – Apple. https://www.apple.com/swift/ (Accessed: 2022-02-23), 2022.

[App22e] Apple, Inc. Swift.org. https://swift.org/about (Accessed: 2022-03-15), 2022.

[App22f] Apple Inc. Swift.org - Package Manager. https://swift.org/package-manager (Accessed: 2022-05-25), 2022.

[App22g] Apple Inc. SwiftUI Overview - Xcode - Apple Developer. https://developer.apple.com/xcode/swiftui/ (Accessed: 2022-05-25), 2022.

[App22h] Apple Inc. Xcode 13 Overview - Apple Developer. https://developer.apple.com/xcode/ (Accessed: 2022-04-29), 2022.

[App22i] Apple Inc. XCTest. https://developer.apple.com/documentation/xctest (Accessed: 2022-05-03), 2022.

[APT20] Giuseppe Antonio Pierro and Roberto Tonelli. PASO: A Web-Based Parser for Solidity Language Analysis. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 16–21, 2020. URL https://doi.org/10.1109/IWBOSE50093.2020.9050263.

[Bab22] Babel. Babel · The compiler for next generation JavaScript. https://babeljs.io/ (Accessed: 2022-03-14), 2022.

[Baz22] Mihai Bazon. UglifyJS 3. https://github.com/mishoo/UglifyJS (Accessed: 2022-03-14), 2022.

[BCT15] Antuan Byalik, Sanchit Chadha, and Eli Tilevich. Native-2-Native: Automated Cross-Platform Code Synthesis from Web-Based Programming Resources. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2015, page 99–108. Association for Computing Machinery, New York, NY, USA, 2015. ISBN 9781450336871. URL https://doi.org/10.1145/2814204.2814210.

[BHGG19] Andreas Biørn-Hansen, Tor-Morten Grønli, and G. Ghinea. A Survey and Taxonomy of Core Concepts and Research Challenges in Cross-Platform Mobile Development. *ACM Computing Surveys (CSUR)*, 51:1 – 34, 2019. URL https://doi.org/10.1145/3241739.

[BHWS21] Daniel Barros, Flavio Horita, Igor Wiese, and Kanan Silva. A Mining Software Repository Extended Cookbook: Lessons learned from a literature review. In *Brazilian Symposium on Software Engineering*, pages 1–10, 2021. URL https://doi.org/10.1145/3474624.3474627.

[BNJ03] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent Dirichlet Allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003. URL https://dl.acm.org/doi/10.5555/944919.944937.

[Cap22] Capacitor. Capacitor by Ionic - Cross-platform apps with web technology. https://capacitorjs.com/ (Accessed: 2022-07-19), 2022.

[Car22] Carthage. Carthage. https://github.com/Carthage/Carthage (Accessed: 2022-05-25), 2022.

[CBTR17] Sanchit Chadha, Antuan Byalik, Eli Tilevich, and Alla Rozovskaya. Facilitating the development of cross-platform software via automated code synthesis from web-based programming resources. *Computer Languages, Systems Structures*, 48:3–19, 2017. ISSN 1477-8424. URL https://doi.org/10.1016/j.cl.2016.08.005.

[Cho56] Noam Chomsky. Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124, 1956. URL https://doi.org/10.1109/TIT.1956.1056813.

[CLTC15] Zi-Yik Cheah, Yik-Shu Lee, Thong-Yun The, and Ji-Jian Chin. Simulation of a pairing-based identity-based identification scheme in IOS. In *2015 IEEE International Conference on Signal and Image Processing Applications (ICSIPA)*, pages 298–303, 2015. URL https://doi.org/10.1109/ICSIPA.2015.7412208.

[Coc22] CocoaPods. CocoaPods.org. https://cocoapods.org/ (Accessed: 2022-05-25), 2022.

[CPCS18] Nathan Cassee, Gustavo Pinto, Fernando Castor, and Alexander Serebrenik. How Swift Developers Handle Errors. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 292–302. IEEE, 2018. URL https://doi.org/10.1145/3196398.3196428.

[DAB21] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. Sampling Projects in GitHub for MSR Studies. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*, pages 560–564. IEEE, 2021. URL https://doi.org/10.1109/MSR52588.2021.00074.

[Dan22] Albert Danial. cloc. https://github.com/AlDanial/cloc (Accessed: 2022-02-03), 2022.

[Dem15] Abdullah Onur Demir. MEPHISTO: A source to source transpiler from pure data to faust. Master's thesis, Enformatik Enstitüsü, 2015. URL https://hdl.handle.net/11511/24717.

[dkh22] dkhamsing. Dkhamsing/open-source-IOS-apps: Collaborative List of open-source IOS apps. https://github.com/dkhamsing/open-source-ios-apps (Accessed: 2022-04-15), 2022.

[Dow16] Eric Downey. *Practical Swift*. Springer, Berkeley, CA, 1st edition, 2016. ISBN 9781484222805.

[EKAYW17] Wafaa S. El-Kassas, Bassem A. Abdullah, Ahmed H. Yousef, and Ayman M. Wahba. Taxonomy of cross-platform mobile applications development approaches. *Ain Shams Engineering Journal*, 8(2):163–190, 2017. URL https://doi.org/10.1016/j.asej.2015.08.004.

[EKSY21] Shaymaa Sayed El-Kaliouby, Sahar Selim, and Ahmed H Yousef. Native Mobile Applications UI Code Conversion. In *2021 16th International Conference on Computer Engineering and Systems (ICCES)*, pages 1–5. IEEE, 2021. URL https://doi.org/10.1109/ICCES54031.2021.9686093.

[FG22] Brent Fulgham and Isaac Gouy. The Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/ (Accessed: 2022-07-19), 2022.

[FW16] Xiaochao Fan and Kenny Wong. Migrating User Interfaces in Native Mobile Applications: Android to IOS. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, MOBILESoft '16, page 210–213. Association for Computing Machinery, New York, NY, USA, 2016. ISBN 9781450341783. URL https://doi.org/10.1145/2897073.2897101.

[Gai22] Jean-loup Gailly. zipgrep(1) - Linux man page. https://linux.die.net/man/1/zipgrep (Accessed: 2022-05-25), 2022.

[Gal22] Andrew Gallant. Burntsushi/ripgrep. https://github.com/BurntSushi/ripgrep (Accessed: 2022-05-25), 2022.

[Git22a] GitHub, Inc. GitHub REST API. https://docs.github.com/en/rest (Accessed: 2022-05-24), 2022.

[Git22b] GitHub, Inc. GitHub: Where the world builds software · GitHub. https://github.com/ (Accessed: 2022-05-25), 2022.

[Git22c] GitHub, Inc. Resources in the REST API. https://docs.github.com/en/rest/overview/resources-in-the-rest-api (Accessed: 2022-05-25), 2022.

[Git22d] GitHub, Inc. Where open source communities live · GitHub. https://github.com/open-source (Accessed: 2022-07-19), 2022.

[Goo13] Google. objc2j. https://code.google.com/archive/p/objc2j/ (Accessed: 2022-07-19), 2013.

[Goo16] Google. J2ObjC. https://developers.google.com/j2objc/guides/why-use-j2objc (Accessed: 2022-06-06), 2016.

[Goo21a] Google. Android's Kotlin-first approach, 2021. URL https://developer.android.com/kotlin/first (Accessed: 2022-03-15).

[Goo21b] Google. J2ObjC. https://github.com/google/j2objc (Accessed: 2022-02-19), 2021.

[Goo22a] Google. android.widget — Android Developers. https://developer.android.com/reference/android/widget/package-summary (Accessed: 2022-07-19), 2022.

[Goo22b] Google. Angular. https://angular.io/ (Accessed: 2022-07-19), 2022.

[Goo22c] Google. Closure Compiler. https://developers.google.com/closure/compiler (Accessed: 2022-03-14), 2022.

[Goo22d] Google. Flutter - Build apps for any screen. https://flutter.dev/ (Accessed: 2022-07-19), 2022.

[Goo22e] Google. Jetpack Compose. https://developer.android.com/jetpack/compose (Accessed: 2022-05-25), 2022.

[Gou13] Georgios Gousios. *The GHTorrent dataset and tool suite*. MSR '13. IEEE Press, Piscataway, NJ, USA, 2013. ISBN 9781467329361. 233–236 pp. URL http://dl.acm.org/citation.cfm?id=2487085.2487132.

[Gra22] Gradle Inc. Gradle Build Tool. https://gradle.org/ (Accessed: 2022-05-25), 2022.

[GSM21] GSM Association. The mobile economy 2021. Technical report, 2021. 8-9 pp. URL https://www.gsma.com/mobileeconomy/wp-content/uploads/2021/07/GSMA_MobileEconomy2021_3.pdf (Accessed: 2022-01-05).

[Hop11] John E. Hopcroft. Einführung in die Automatentheorie, formale Sprachen und Berechenbarkeit, 2011. ISBN 9783868940824.

[HSKY19] Rameez B. Hamza, David I. Salama, Martina I. Kamel, and Ahmed H. Yousef. TCAIOSC: Application Code Conversion. In *2019 Novel Intelligent and Leading Emerging Sciences Conference (NILES)*, volume 1, pages 230–234, 2019. URL https://doi.org/10.1109/NILES.2019.8909207.

[HvKPT87] R. D. Huijsman, J. van Katwijk, C. Pronk, and W. J. Toetenel. Translating Algol 60 Programs into Ada. *Ada Lett.*, VII(5):42–50, 1987. ISSN 1094-3641. URL https://doi.org/10.1145/36077.36080.

[Int22] Internet Archive. Wayback Machine. https://web.archive.org/ (Accessed: 2022-05-25), 2022.

[ISO15] Information technology – Vocabulary – language construct. Standard, International Organization for Standardization, Geneva, Switzerland, 2015. URL https://www.iso.org/standard/63598.html (Accessed: 2022-05-15).

[Jav22] JavaCC Community. JavaCC. https://javacc.github.io/javacc/ (Accessed: 2022-03-15), 2022.

[Jet21] JetBrains. Kotlin Programming - The State of Developer Ecosystem in 2020 Infographic. https://www.jetbrains.com/lp/devecosystem-2020/kotlin/ (Accessed: 2022-03-15), 2021.

[Jet22a] JetBrains. Basic syntax. https://kotlinlang.org/docs/basic-syntax.html (Accessed: 2022-04-29), 2022.

[Jet22b] JetBrains. History for docs/topics/basic-syntax.md - JetBrains/kotlin-website. https://github.com/JetBrains/kotlin-web-site/commits/master/docs/topics/basic-syntax.md (Accessed: 2022-05-25), 2022.

[Jet22c] JetBrains. IntelliJ IDEA: The Capable Ergonomic Java IDE by JetBrains. https://www.jetbrains.com/idea/ (Accessed: 2022-04-29), 2022.

[Jet22d] JetBrains. Kotlin Programming Language. https://kotlinlang.org/ (Accessed: 2022-01-23), 2022.

[Jet22e] JetBrains. Kotlin Programming Language - Grammar. https://kotlinlang.org/docs/reference/grammar.html (Accessed: 2022-07-19), 2022.

[Jet22f] JetBrains. kotlin-web-site/basic-syntax.md at master · JetBrains/kotlin-website · GitHub. https://github.com/JetBrains/kotlin-web-site/blob/master/docs/topics/basic-syntax.md?plain=1 (Accessed: 2022-05-25), 2022.

[Jet22g] JetBrains. kotlin.test. https://kotlinlang.org/api/latest/kotlin.test/index.html (Accessed: 2022-05-03), 2022.

[Jet22h] JetBrains. Team Tools - The State of Developer Ecosystem in 2021 Infographic. https://www.jetbrains.com/lp/devecosystem-2021/team-tools/ (Accessed: 2022-05-24), 2022.

[K+56] Stephen C Kleene et al. Representation of events in nerve nets and finite automata. *Automata studies*, 34:3–41, 1956. URL https://doi.org/10.1515/9781400882618-002.

[KCH15] Rohit Kulkarni, Aditi Chavan, and A Hardik. Transpiler and it's Advantages. *International Journal of Computer Science and Information Technologies*, 6(2): 1629–1631, 2015. ISSN 0975-9646. URL https://ijcsit.com/docs/Volume%206/vol6issue02/ijcsit20150602159.pdf.

[KCM07] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of software maintenance and evolution: Research and practice*, 19(2):77–131, 2007. URL https://doi.org/10.1002/smr.344.

[Kil19] KillerAllKillerAll. ANTLR4 - how to stop the parser when it takes too long. https://stackoverflow.com/questions/56761502/antlr4-how-to-stop-the-parser-when-it-takes-too-long (Accessed: 2022-07-19), 2019.

[Kri11] Paul Krill. JetBrains readies JVM-based language. *InfoWorld*, 2011. URL https://www.infoworld.com/article/2622405/jetbrains-readies-jvm-based-language.html (Accessed: 2022-03-15).

[KVE94] Peter Knauber, Stefan Vorwieger, and Reinhard Eppler. Terminologie des Übersetzerbaus. Technical Report 255, Fachbereich Informatik, 1994. URL http://nbn-resolving.de/urn:nbn:de:hbz:386-kluedo-49764.

[Lee17] Kent D. Lee. *Foundations of Programming Languages*. Springer, 2017. ISBN 9783319707907. URL https://doi.org/10.1007/978-3-319-70790-7.

[Les06] Lesk, M. and Schmidt, E. Lex. http://dinosaur.compilertools.net/ (Accessed: 2022-03-06), 2006.

[LYB+19] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. Aroma: Code Recommendation via Structural Code Search. *Proc. ACM Program. Lang.*, 3(OOPSLA), 2019. URL https://doi.org/10.1145/3360578.

[Mec17] Gautier Mechling. Swift is like Kotlin. http://nilhcem.com/swift-is-like-kotlin/ (Accessed: 2022-03-21), 2017.

[Met22a] Meta Platforms, Inc. React Native · Learn once, write anywhere. https://reactnative.dev/ (Accessed: 2022-07-19), 2022.

[Met22b] Meta Platforms, Inc. React – a JavaScript library for building user interfaces. https://reactjs.org/ (Accessed: 2022-07-19), 2022.

[Mic22a] Microsoft. .NET | Free. Cross-platform. Open Source. https://dotnet.microsoft.com/en-us/ (Accessed: 2022-03-14), 2022.

[Mic22b] Microsoft. TypeScript: JavaScript With Syntax For Types. https://www.typescriptlang.org/ (Accessed: 2022-03-14), 2022.

[MKCN17] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating GitHub for Engineered Software Projects. *Empirical Software Engineering*, 22(6):3219–3253, 2017. URL https://doi.org/10.1007/s10664-017-9512-6.

[MLL+18] Y. Ma, X. Liu, Y. Liu, Y. Liu, and G. Huang. A Tale of Two Fashions: An Empirical Study on the Performance of Native Apps and Web Apps on Android. *IEEE Transactions on Mobile Computing*, 17:990–1003, 2018. ISSN 1558-0660. URL https://doi.org/10.1109/TMC.2017.2756633.

[MM19] Bruno Góis Mateus and Matias Martinez. An empirical study on quality of Android applications written in Kotlin language. *Empirical Software Engineering*, 24:3356 – 3393, 2019. URL https://doi.org/10.1007/s10664-019-09727-4.

[MM20] Bruno Gois Mateus and Matias Martinez. On the adoption, usage and evolution of Kotlin features in Android development. *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020. URL https://doi.org/10.48550/arXiv.1907.09003.

[MME+20] Ahmad A. Muhammad, Amira T. Mahmoud, Shaymaa S. Elkalyouby, Rameez B. Hamza, and Ahmed H. Yousef. Trans-Compiler based Mobile Applications code converter: swift to java. In *2020 2nd Novel Intelligent and Leading Emerging Sciences Conference (NILES)*, pages 247–252, 2020. URL https://doi.org/10.1109/NILES50944.2020.9257928.

[Moz22] Mozilla Foundation. Making PWAs work offline with Service workers. https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Offline_Service_workers (Accessed: 2022-07-19), 2022.

[MSSY21] Ahmad Ahmad Muhammad, Abdelrahman Mohamed Soliman, Sahar Selim, and Ahmed H Yousef. Generic Library Mapping Approach for Trans-Compilation. In *2021 International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC)*, pages 62–68. IEEE, 2021. URL https://doi.org/10.1109/MIUCC52538.2021.9447641.

[Nie16] Pat Niemeyer. Patniemeyer/J2SWIFT: Basic java to swift syntax converter. https://github.com/patniemeyer/j2swift (Accessed: 2022-07-19), 2016.

[Oll16] Angel G. Olloqui. Swift vs Kotlin for real iOS/Android apps. https://angelolloqui.github.io/blog/38-Swift-vs-Kotlin-for-real-iOS-Android-apps (Accessed: 2022-07-19), 2016.

[Oll20] Angel G. Olloqui. SwiftKotlin. https://github.com/angelolloqui/SwiftKotlin (Accessed: 2022-03-21), 2020.

[O'R16] Gerard O'Regan. *Introduction to the history of computing: a computing history primer*. Springer, 2016. ISBN 9783319331386. URL https://doi.org/10.1007/978-3-319-33138-6.

[Ore12] Ann Oreshnikova. Kotlin Goes Open Source! | The Kotlin Blog. *The JetBrains Blog*, 2012. URL https://blog.jetbrains.com/kotlin/2012/02/kotlin-goes-open-source-2/ (Accessed: 2022-03-15).

[OTE20] Victor Oliveira, Leopoldo Teixeira, and Felipe Ebert. On the Adoption of Kotlin on Android Development: A Triangulation Study. *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 206–216, 2020. URL https://doi.org/10.1109/SANER48275.2020.9054859.

[Par12] Terence Parr. *The definitive ANTLR4 reference*. The Pragmatic programmers. Dallas, Tex. [a.o.], 2012. ISBN 1934356999.

[Par22] Terence Parr. ANTLR. https://www.antlr.org/ (Accessed: 2022-03-19), 2022.

[PHF14] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive LL (*) parsing: the power of dynamic analysis. *ACM SIGPLAN Notices*, 49(10):579–598, 2014. URL https://doi.org/10.1145/2714064.2660202.

[PRWZ02] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002. URL https://doi.org/10.3115/1073083.1073135.

[QGZ16] P. Que, X. Guo, and M. Zhu. A Comprehensive Comparison between Hybrid and Native App Paradigms. In *2016 8th International Conference on Computational Intelligence and Communication Networks (CICN)*, pages 611–614, 2016. ISSN 2472-7555. URL https://doi.org/10.1109/CICN.2016.125.

[RBS13] Gordana Rakić, Zoran Budimac, and Miloš Savić. Language Independent Framework for Static Code Analysis. BCI '13, page 236–243. Association for Computing Machinery, New York, NY, USA, 2013. ISBN 9781450318518. URL https://doi.org/10.1145/2490257.2490273.

[Rea22] Realm. SwiftLint. https://github.com/realm/SwiftLint (Accessed: 2022-04-28), 2022.

[RGL+20] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. *CoRR*, abs/2009.10297, 2020. URL https://doi.org/10.48550/arXiv.2009.10297.

[RLMW14] Julian Rith, Philipp S Lehmayr, and Klaus Meyer-Wegener. Speaking in tongues: SQL access to NoSQL systems. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 855–857, 2014. URL https://doi.org/10.1145/2554850.2555099.

[RP20] Kristiina Rahkema and Dietmar Pfahl. Empirical study on code smells in iOS applications. In *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*, pages 61–65, 2020. URL https://doi.org/10.1145/3387905.3388597.

[RPE+16] Marcel Rebouças, Gustavo Pinto, Felipe Ebert, Weslley Torres, Alexander Serebrenik, and Fernando Castor. An Empirical Study on the Usage of the Swift Programming Language. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, volume 1, pages 634–638. IEEE, 2016. URL https://doi.org/10.1109/SANER.2016.66.

[RS59] Michael O. Rabin and Dana Scott. Finite automata and their decision problems. *IBM journal of research and development*, 3(2):114–125, 1959. URL https://doi.org/10.1147/rd.32.0114.

[RS12] C. P. Rahul Raj and Seshu Babu Tolety. A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach. In *2012 Annual IEEE India Conference (INDICON)*, pages 625–629, 2012. ISSN 2325-9418. URL https://doi.org/10.1109/INDCON.2012.6420693.

[Sas22] Sass Team. Sass: Syntactically Awesome Style Sheets. https://sass-lang.com/ (Accessed: 2022-03-14), 2022.

[Sch21] Dominik Schultes. SequalsK—A Bidirectional Swift-Kotlin-Transpiler. *2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft)*, pages 73–83, 2021. URL https://doi.org/10.1109/MobileSoft52590.2021.00017.

[Sch22] Dominik Schultes. SequalsK. https://transpile.iem.thm.de/ (Accessed: 2022-03-21), 2022.

[Sen21] Sensor Tower. Combined global Apple App Store and Google Play app downloads from 1st quarter 2015 to 3rd quarter 2021 (in billions). https://www.statista.com/statistics/604343/number-of-apple-app-store-and-google-play-app-downloads-worldwide/ (Accessed: 2022-01-05), 2021.

[SHK+19] David I. Salama, Rameez B. Hamza, Martina I. Kamel, Ahmad A. Muhammad, and Ahmed H. Yousef. TCAIOSC: Trans-Compiler Based Android to iOS Converter. In *International Conference on Advanced Intelligent Systems and Informatics*, pages 842–851. Springer, 2019. URL https://doi.org/10.1007/978-3-030-31129-2_77.

[SM16] Stephen Schaub and Brian A. Malloy. The Design and Evaluation of an Interoperable Translation System for Object-Oriented Software Reuse. *J. Object Technol.*, 15(4):1:1–33, 2016. URL https://doi.org/10.5381/jot.2016.15.4.a1.

[SS21] Markus Schnappinger and Jonathan Streit. Efficient Platform Migration of a Mainframe Legacy System Using Custom Transpilation. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 545–554, 2021. URL https://doi.org/10.1109/ICSME52107.2021.00055.

[SS22] Larissa Schneider and Dominik Schultes. Evaluating Swift-to-Kotlin and Kotlin-to-Swift Transpilers. In *2022 IEEE/ACM 9th International Conference on Mobile Software Engineering and Systems (MobileSoft)*, pages 102–106, 2022. URL https://doi.org/10.1145/3524613.3527811.

[Sta22a] Stack Exchange Inc. Stack Overflow - Where Developers Learn, Share, & Build Careers. https://stackoverflow.com/ (Accessed: 2022-02-03), 2022.

[Sta22b] StatCounter. Mobile operating system market share worldwide. https://gs.statcounter.com/os-market-share/mobile/worldwide (Accessed: 2022-07-19), 2022.

[Stu20] Studo. Kotlift. https://github.com/studo-app/Kotlift (Accessed: 2022-03-21), 2020.

[SYCI18] Yuichi Semura, Norihiro Yoshida, Eunjong Choi, and Katsuro Inoue. Multilingual Detection of Code Clones Using ANTLR Grammar Definitions. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 673–677, 2018. URL https://doi.org/10.1109/APSEC.2018.00088.

[The20] The Editors of Encyclopedia Britannica. Grammar. https://www.britannica.com/topic/grammar (Accessed: 2022-07-19), 2020.

[The21] The Apache Software Foundation. Commons csv – home. https://commons.apache.org/proper/commons-csv/ (Accessed: 2022-07-19), 2021.

[The22a] The Apache Software Foundation. Apache Cordova. https://cordova.apache.org/ (Accessed: 2022-07-19), 2022.

[The22b] The Apache Software Foundation. Maven – Welcome to Apache Maven. https://maven.apache.org/ (Accessed: 2022-05-25), 2022.

[TY21] Bernal Tarazona and Steven Yeisson. *ANTLR 4 grammar of the Swift 5 programing language*. Universidad de los Andes, 2021. URL http://hdl.handle.net/1992/53289.

[TYM21] Bernal Tarazona, Steven Yeisson, and Martin Mirchev. Grammars-V4/swift/SWIFT5 at master · ANTLR/grammars-V4. https://github.com/antlr/grammars-v4/tree/master/swift/swift5 (Accessed: 2022-07-19), 2021.

[Ven22a] Vinícius Jorge Vendramini. Gryphon. https://vinivendra.github.io/Gryphon/ (Accessed: 2022-03-21), 2022.

[Ven22b] Vinícius Jorge Vendramini. Gryphon. https://github.com/vinivendra/Gryphon (Accessed: 2022-03-21), 2022.

[VGM20] Vinícius Jorge Vendramini, A. Goldman, and G. Mounié. Improving mobile app development using transpilers with maintainable outputs. *Proceedings of the 34th Brazilian Symposium on Software Engineering*, 2020. URL https://doi.org/10.1145/3422392.3422426.

[yan19] yanagiba. Swift Abstract Syntax Tree. https://github.com/yanagiba/swift-ast (Accessed: 2022-02-03), 2019.

[Yel88] Daniel M. Yellin. Translating between programming languages. In *Attribute Grammar Inversion and Source-to-source Translation*, Lecture Notes in Computer Science, pages 118–143. Springer, Berlin, Heidelberg, 1988. ISBN 978-3-540-39079-4. URL https://doi.org/10.1007/3-540-19072-4_6.

[Zay20] Hussein Zayat. Kotlin and Android applications: diffusion and adoption of characteristic constructs. Master's thesis, Politecnico di Torino, 2020. URL http://webthesis.biblio.polito.it/id/eprint/14526.