INSTITUTO
UNIVERSITÁRIO
DE LISBOA

SKot: a Web-based Structured Code Editor for Introductory Programming in Kotlin

Pedro Manuel Lima e Silva

Master in Computer Engineering

Supervisor:
Prof. André Leal Santos, Assistant Professor
Iscte - Instituto Universitário de Lisboa

November, 2022

Resumo

O uso de Editores Estruturados de Código como o Scratch para introdução à programação, especialmente em idades mais jovens, é prevalente nos sistemas de educação modernos. A transição para editores de código baseados em texto pode ser desafiante para iniciantes, principalmente no que diz respeito à aprendizagem da sintaxe de uma linguagem moderna.

Este trabalho explora a investigação atual sobre Editores Estruturados de Código e explica a implementação de um Editor Projecional que apresenta o código ao utilizador com a mesma aparência que teria num editor de texto numa linguagem de programação moderna, como Kotlin, enquanto preserva a correção sintática durante edições.

Um editor no web browser, chamado SKot (Structured Kotlin), foi implementado usando JavaScript (JSON, React e Redux). Uma técnica para implementar o Editor Projecional usando estas tecnologias é descrita em detalhe. Por fim, o SKot é comparado com outros editores de código modernos e descreve-se trabalho futuro relativo a melhoramentos no mesmo.

Palavras-Chave: Editor de Texto Estruturado, Projeção, Ambientes de Desenvolvimento, Programação, Pedagogia

Abstract

The use of Structured Code Editors such as Scratch for introduction to programing, especially at a younger age is prevalent in modern education systems. Transitioning to text-based editors sometimes proves to be challenging for beginners especially when it comes to learning the syntax of a modern language.

This work goes into the research on Structured Code Editors and delves into the implementation of a Projectional Editor that presents the code to the user as it would appear in a text-based editor in a modern programing language such as Kotlin, while preserving syntactical correctness during edits.

A web-based editor, named **SKot** for **S**tructured **Kot**lin was implemented using JavaScript (JSON, React and Redux). A technique for implementing the Projectional Editor using these technologies is described in detail. Then, SKot is compared to other modern code editors and further work is proposed to improve it.

Keywords: Structured Editors, Projection, IDE, Programming, Pedagogy

Contents

Resumo i

Abstract iii

Chapter 1. Introduction 1

1.1. Context 1

1.2. Research Questions 2

1.3. Goals 2

1.4. Research Method 3

Chapter 2. State of the Art 5

2.1. General Purpose Code Editors 5

2.2. Structured Code Editing 8

2.3. Pedagogical tools 10

2.4. Studies on Developers Text Editing Patterns 10

Chapter 3. SKot: a Structured Editor for Kotlin 13

3.1. Language Support 13

3.2. Limitations 20

Chapter 4. Implementation 23

4.1. The Abstract Syntax Tree 24

4.2. Components 27

4.3. User Interactions 30

4.4. AST Projections 32

Chapter 5. Discussion 35

5.1. Comparative Analysis 35

5.2. Future Work 36
5.3. Conclusion 38

References 39

CHAPTER 1

Introduction

Most software development involves editing text, usually in a general-purpose text editor (tools like VS Code, Sublime Text, Text Wrangler, Eclipse, Visual Studio). Changes performed in editors like the ones mentioned have the potential of creating syntax-related errors which produce error messages that are not very clear to the inexperienced programmer [1].

Other approaches to software development implement visual interfaces that represent code blocks like if-statements, loops, functions, and variables. (e.g., Scratch [2], OutSystems [3])

In the context of this work, we want to develop a tool that helps the user improve programing skills that are transferable to other languages and environments. We aim to familiarize the user with a modern programing language's syntax, while minimizing typing mistakes.

1.1. Context

By 2019, the most popular text editor among Stack Overflow users was Visual Studio Code (VSCode) which is a general-purpose code editor (GPCE) [4]. It has many extensions to its functionality designed to provide language-related capabilities like code snippets, linters, and formatters. These extensions don't limit the text inputs accepted but are either shortcuts to introduce a piece of common code (like declaring a function) or markup on syntax errors or bad formatting, like underlining a piece of code in red.

In cases where the IDE is designed for a Domain Specific Language (DSL), visual interfaces are more commonly available. Examples of this are OutSystems, used to develop smartphone and web apps; Blender's Shader Graphs, useful to create material shaders; Unreal Engine's Blueprints and Unity's Visual Scripting. In the later 3 examples, these

serve as abstractions that end up being transpiled or compiled to other languages and serve as a tool for an artist or developer who is not familiar with the language in question.

Between these two extremes, where the developer has full freedom to fail in the case of a general-purpose text editor (GPTE) and where the developer has no awareness of the underlying code syntax, there can be a middle ground of a Structured Code Editor (SCE) with textual representation, where the developer can see the full syntax of the language and is prevented from typing invalid programs.

When teaching students to code, block-based programing is often introduced, to focus on teaching the fundamental concepts of programing [5]. Despite knowing these fundamental concepts, when transitioning to text-based programing, students are often met with syntax errors that could be a detriment to the programing experience [1], [6], [7]. Structured Code Editors have been proposed to bridge this gap [8] by letting the student see the underlying syntax of the program being built with block-based interactions.

1.2. Research Questions

(1) What are the User Interface / Experience challenges faced in the designing an SCE?
(2) How does the editing effort in an SCE compare to modern Development Environments?

1.3. Goals

The main goal of this work is to investigate how an SCE can help with the process of learning a general-purpose programing language. Additionally, we aim to measure key performance indicators (KPIs) to compare typing efficiency in an SCE vs a GPTE with no language support vs a dedicated IDE with language support.

To achieve these goals, a prototype SCE will be built to partially support Kotlin language features. This prototype should include a reporting tool to register every keystroke and cursor change.

1.4. Research Method

The investigation method for this work will closely follow the Design Science Research Methodology (DSRM) Process Model, described by K. Peffers et al. [9].

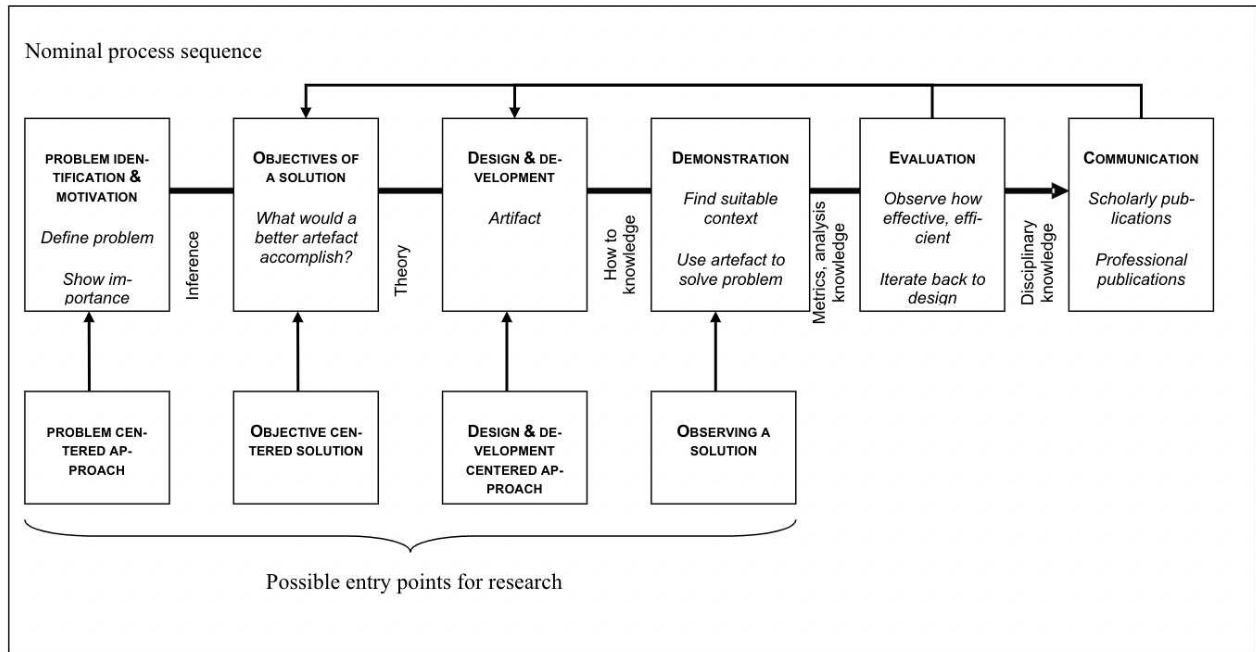

Figure 1. Design Science Research Process Model (taken from [9])

In this case, the application of DSRM will follow an Objective Centered Approach, since the goal is to produce a prototype SCE that improves on KPIs that are comparable with other IDEs. Following that approach, we'll start by defining which KPIs can be improved on in the “Objectives of a Solution” activity.

Then, the prototype will be designed and implemented in the “Design and Development” activity. This entails the definition of functionalities and architecture of the prototype to be implemented in this same phase.

Once a prototype that covers the main language features is achieved, we can proceed to the Demonstration phase, where we'll measure those KPIs defined in the “Objectives of a Solution” activity with the prototype and other text editors.

In the following “Evaluation” activity, the results obtained in the previous activity will be compared. Depending on the result of this comparison and time constraints, a

new iteration starting from the “Design and Implementation” can be initiated to improve on the prototype.

Eventually, when ready, the problem, research and prototype produced can be presented in the “Communication” activity.

The following chapters will delve into the development of a Structured Code Editor that uses the findings of the research outlined in Chapter 2 to guide its features. In Chapter 3, the functionality of the Structured Code Editor is laid-out and some design choices and limitations are discussed. Chapter 4 follows it, by discussing more technical aspects of the Code Editor’s implementation and architecture. Concluding this work, Chapter 5 discusses a brief comparative analysis of the custom editor compared to other editors and future work.

CHAPTER 2

State of the Art

2.1. General Purpose Code Editors

Developers use a wide variety of development environments, and these tools vary in complexity and feature sets widely. From text terminal programs like vim to fully integrated development environments like Android Studio or Eclipse [4]. This wide range of choice caters to different fields and technologies. XCode is used mainly by developers for the macOS ecosystem; Android Studio to develop Android apps.

General-purpose Text Editors (GPTE), such as Microsoft's Notepad or Apple's Text Editor app, are programs that can edit any type of text file. They can have very basic functionality, like opening text files to view, edit them and save changes on the file system.

General-purpose Code Editors (GPCE), such as Atom or Visual Studio Code, have the same capabilities as GPTEs but also include features that adapt to the type of file being edited. These include, but are not limited to, syntax highlighters, formatters, autocomplete and code snippets, and linters.

2.1.1. Syntax Highlighting

Syntax highlighting visually changes the text formatting to highlight language keywords and color-code parts of the text as meaningful words for the language of the text file.

Figure 2 shows the same file in the same GPCE (Visual Studio Code) with Syntax highlighting turned on and turned off. It is apparent that language keywords turned purple while names of variables turned light blue, for example.

2.1.2. Code formatting

Code formatting is a functionality where the Code Editor will change the spacing of words and indentation of lines to adhere to the language rules.

```
1  export const INDENT_CHAR = '  ';
2
3  export const PRINTER_EMPTY = () => '';
4
5  export function getIndent(indentations) {
6    if (!indentations) {
7      return '';
8    }
9    let indent = '';
10   for (let i = 0; i < indentations; i++) {
11     indent += INDENT_CHAR;
12   }
13   return indent;
14 }
15
```

```
1  export const INDENT_CHAR = '  ';
2
3  export const PRINTER_EMPTY = () => '';
4
5  export function getIndent(indentations) {
6    if (!indentations) {
7      return '';
8    }
9    let indent = '';
10   for (let i = 0; i < indentations; i++) {
11     indent += INDENT_CHAR;
12   }
13   return indent;
14 }
15
```

Figure 2. Syntax highlighting on and off in Visual Studio Code

Figure 3 (left) shows code that is not readable due to the lack of spaces and indentation. The same code is shown on the right, after running the formatting function, placing spaces where appropriate and indenting lines according to rules that can be customized by the user. The user can still delete the spaces that the formatting function created after running it.

```
5  export function getIndent(indentations){if
   (!indentations){return'';}let indent='';for
   (let i=0;i<indentations;i++){indent
   +=INDENT_CHAR;}return indent;}
6
```

```
5  export function getIndent(indentations) {
6    if (!indentations) {
7      return '';
8    }
9    let indent = '';
10   for (let i = 0; i < indentations; i++) {
11     indent += INDENT_CHAR;
12   }
13   return indent;
14 }
15
```

Figure 3. JavaScript with no spaces or tabs (left). After running the formatter, the code on the right is produced

2.1.3. Auto-complete and code snippets

While typing in GPCEs like VSCode or Sublime Text, the program can suggest the rest of the word being typed, and, when language support is available, suggest code snippets to save time. A simple auto-complete scenario is when the developer is typing a long function name and the editor suggests the rest of its name. One example of a code

snippet is shown in Figure 4, where the user typed “fun” and the editor suggested a code snippet for a function declaration. Upon selecting that option, the editor generates the text for a function declaration, placing the selection on the function name, so it can be replaced on the next keyboard input. This example is small, but code snippets can have dozens of lines, like when initializing an html document.

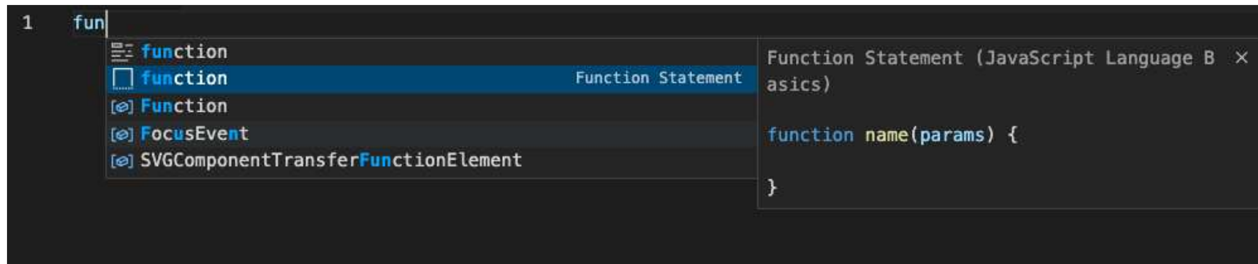

Figure 4. VSCode suggesting a function declaration

2.1.4. Syntax Errors

When a programing language’s grammar rule is broken, a syntax error will be generated when the code is run or compiled. To prevent this from happening at build time or runtime (for compiled or interpreted languages respectively), GPCEs parse the text as it is typed to check the program’s syntactical correctness. Figure 5 shows an example of syntax errors caused by a single character deletion (in this case, the first semicolon in the “for” cycle) which then caused the parser to highlight other places that break other grammar rules. In this situation, the editor pointed out the correct mistake “Missing semicolon”, but it ultimately is the job of the developer to find the inconsistency and how to resolve it.

Another common example would be not closing every parenthesis that is open (most common in the analysis done in [6]). GPCEs usually close parenthesis in front of the typing cursor when the developer inputs an open parenthesis but in the processing of editing the code, punctuation marks might be lost or misplaced, causing other syntax errors. The amount of freedom provided by text editors opens the possibility to these kinds of ambiguous mistakes that the developer then needs to decide on how to resolve.

If the error message were not clear, an innexperienced programer would have trouble finding the source of the problem [1].

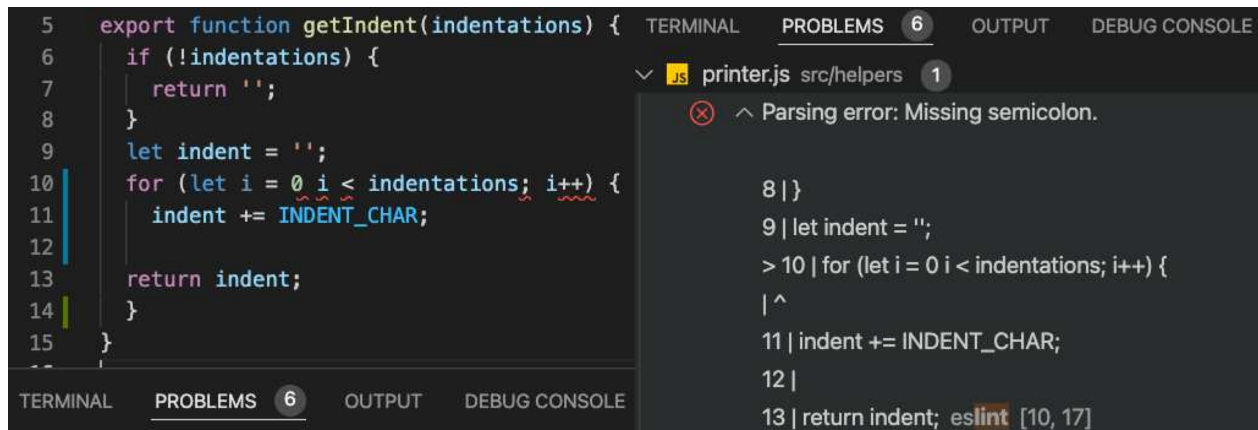

Figure 5. JavaScript code with a syntax error

2.2. Structured Code Editing

Ko et al. 2005 [10] ascribed SCEs' resilience to syntax errors to their ability to "avoid the parsing process altogether, instead allowing the direct editing of the abstract syntax tree that represents a program". SCEs use other visual means other than editable lines of text, to represent the program structure. These can be completely abstracted from the code that will eventually run (like in the case of Scratch [2]) or something resembling the actual code, as seen in the next two examples.

Tillmann et al. 2012 [11] created TouchDevelop for a code editor that runs on a smartphone, as an SCE for its own tailor-made language. The program created by the user is still visually represented as text but only names and literals are editable as normal text would. It achieves this by having buttons with possible code structures to be inserted (a conditional statement, or a function with a name, arguments, and body). Upon tapping a button, the corresponding code structure is inserted into the program. The user can tap on the generated statement to focus on it and edit its components. In the case of an if-else-block, an expression can be inserted as its condition and then blocks inside their branches using the interface buttons.

Javardise [12] uses a similar approach for the Java language on a desktop. Instead of buttons, however, it uses typing and autocomplete to infer which code structure is

being inserted (much like a GPCE's code snippet functionality). It then replaces the typed characters with a component containing placeholders in the fields that require attention from the developer. Unlike GPCEs with autocomplete, where the whole text generated can be edited, on Javardise only the text inside the placeholders can. This helps the user keep syntax errors from occurring while editing.

This is very close to what we aim to achieve with this work. Instead of focusing on Java, we aim to implement an editor for the Kotlin language, for its simpler syntax and being an industry favorite in some sectors. Another differentiating factor will be its availability, by making it run on the browser, much like other code sharing services like CodePen or JSfiddle, that require no installation, nor download, nor particular operating system to run.

2.2.1. Projectional Editing

Projectional Editing is a technique SCEs can use to avoid syntax errors by directly manipulating the abstract syntax tree (AST). Its potential is not limited to avoiding syntax errors, however. Projectional editing opens the possibility to combine various ways of visualizing code, such as text, tables, equations, and graphics [13].

Figure 6 illustrates what the user sees and interacts with in a Projectional Editor (ProjE). A Projection Engine relates user interactions with edits to the AST. The rendering engine then renders the AST in the UI the notation required. This is a common approach for graphical interfaces that use the Model-View-Controller design pattern, where the user's actions update the model which then updates the view.

Figure 6. Parser-Based Editor (left) Projectional Editor (right). Taken from [13]

ProjEs face challenges in the current development environment, as well. One such challenge is to save files in a way that is friendly to version control systems without losing the readability of the Projection Engine. Another challenge is to not lose the flexibility of GPCEs when rendering the AST as text. If the interactions with the text

require the user to be aware of the structure of the AST, it might be a detriment to the developer's performance. The former challenge falls out of scope for this work, which is more focused on the interactions while editing text in a Structured Code Editor.

2.3. Pedagogical tools

When introducing children to programing, it is most common to use block-based interfaces such as Scratch or Alice [2], [5], [14]. Eventually, these same students will need to learn other languages that do not come with a visual interface and to help with that transition, Kölling et al. 2019 [5] proposed using Stride in the BlueJ editor. Stride is a simplification of the Java programing language that can help a student coming from a block-based interface familiarize themselves with text-based programing.

BlueJ is an IDE, that allows the user to manipulate the code as blocks (while seeing the language's syntax) when editing a Stride class. In this mode, the editor behaves as an SCE and there are keyboard shortcuts to create code blocks, that can be filled with other code blocks. This way, syntax errors are near-impossible to create, much like in other block-based environments. Figure 7 shows a Stride Class open in the BlueJ editor, where the list of shortcuts is visible. This list is content-aware and will show different shortcuts depending on the position of the cursor (blue horizontal line)

However, when editing a Java class, the editor behaves like a GPCE, with the addition of color-coding whole blocks of text. Using this, the user can correlate the colored text blocks in the Java editor with the blocks in Stride while getting comfortable with the capabilities of a GPCE.

It would be interesting to compare transitioning from code blocks like Scratch, directly to an SCE with the full Java syntax instead of Stride, but that analysis falls out of scope for this project.

2.4. Studies on Developers Text Editing Patterns

A particular focus of our work will require researching common text editing patterns employed by experienced developers and ensuring they are possible to perform in this work's artifact. Ko et al. 2005 [10] studied the habits of Java developers with findings that are transportable to the Kotlin language. One such finding of note was that even in

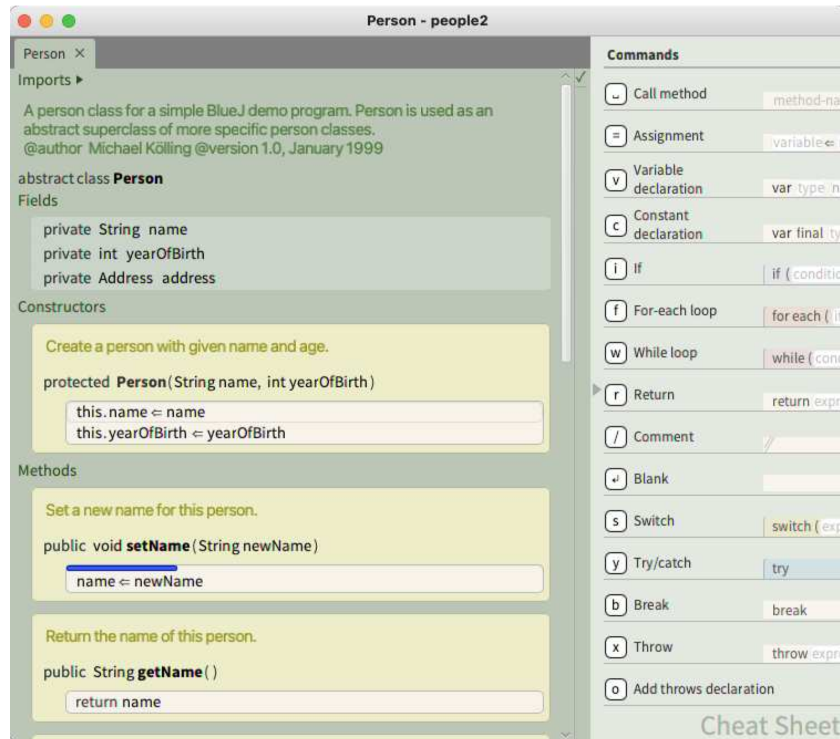

Figure 7. A Stride Class as seen in the BlueJ editor

unstructured editors, selecting text across structures was not common and therefore it would not be a loss if that type of interaction were not possible, as long as the intended edit is.

This study and others like it [5], [12], [15] will be fundamental to inform which kinds of interactions should be prioritized in the implementation of this work.

CHAPTER 3

SKot: a Structured Editor for Kotlin

The focus of this work is be to deliver a Structured Code Editor that differentiates itself by presenting the program being written as text in a particular language, in this case, Kotlin. This should allow the user to benefit from the guarantee of syntactical soundness that a Structured Editor provides and still have the means to familiarize themselves with a text-based programing language's syntax.

As mentioned before, this Structured Code Editor should run in the browser, much like other text editors that already do, such as CodePen, or even VSCode. The latter is available as a web app (accessible at https://vscode.dev), with full functionality. The former is often used by web developers as a code sharing tool for small proof-of-concept artifacts.

Being a web app removes some barriers to entry like having to install a separate application that might not work in some operating system or machine arquitecture. It also opens the possibility to integrate it in other contexts, for example, as an exercise field in a written tutorial.

The Structured Editor developed will henceforth be referred to as **SKot** for **S**tructured **Kot**lin, and focuses on basic concepts of the Kotlin language.

3.1. Language Support

Due to time constraints, the scope of the of language supported by SKot was reduced to Kotlin's fundamental concepts. These are limited to: Functions, If and Else clauses, While loops, declarations of variables and values, Expressions, Assignments, Function calls, Return statements. Further work can be done to increase the number of concepts supported, but for a proof of concept, these should be a good starting point for novices.

The following sections provide a brief description of how SKot provides interactions for each Kotlin language concept.

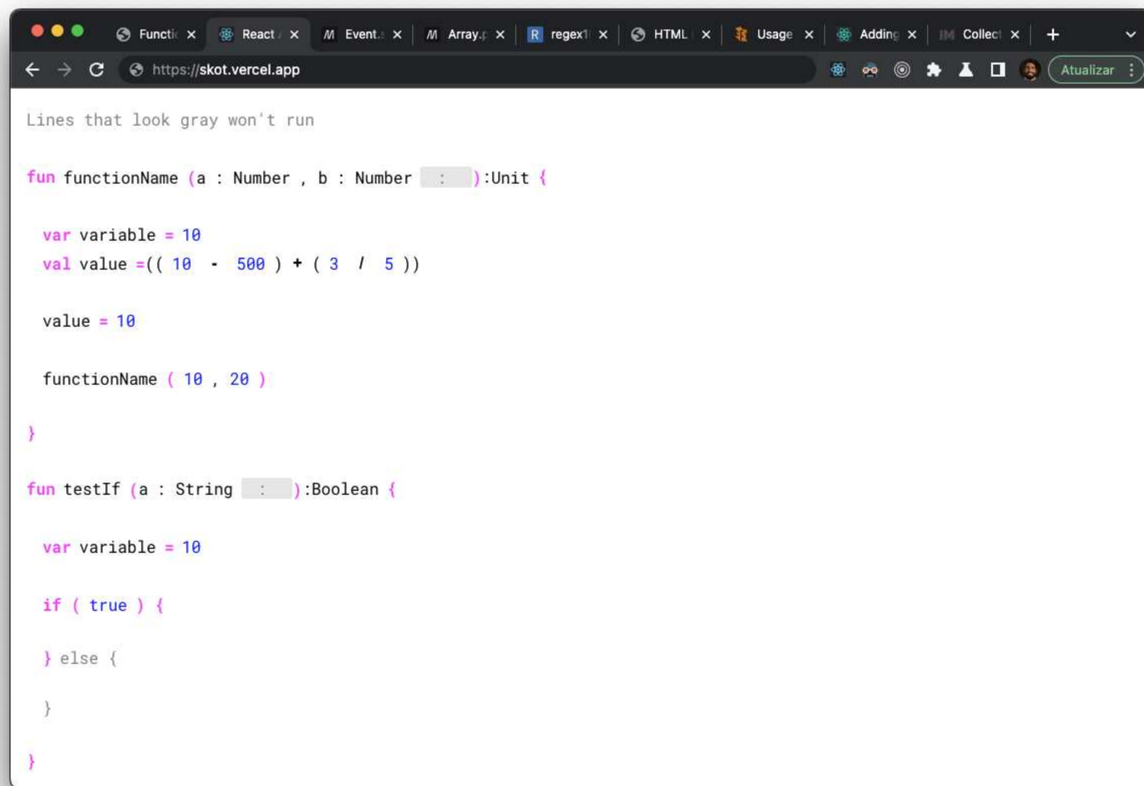

Figure 8. SKot running in a brower

3.1.1. Functions

The last step of Figure 9 illustrates what a Function looks like in SKot. To recreate it, the user would have to type the string “fun” followed by pressing the space bar. At that point, SKot would recognize that a function is being typed and fill in all the necessary fields for a valid function: a name, arguments (enclosed by parenthesis), a return type and a function body (enclosed by curly braces). All these fields can be edited but none can be deleted. The user cannot delete the parenthesis in front of the function’s name, for example.

If the user keeps typing, SKot replaces the pre-filled function name with whatever is being typed. Pressing right or clicking inside the parenthesis focuses the first argument which was pre-created by SKot. Pressing Backspace or Delete with the argument’s name

```
0:  fun

1:  fun functionName( : ):Unit {
    }

2:  fun toDays ( : ):Unit {
    }

3:  fun toDays (years : Type  : ):Unit {
    }

4:  fun toDays (years : Number  : ):Number {

    }

5:  fun toDays (years : Number  : ):Number {
      return
    }

6:  fun toDays (years : Number  : ):Number {
      return
    }

7:  fun toDays (years : Number  : ):Number {
      return( years  *  0 )
    }

8:  fun toDays (years : Number  : ):Number {
      return( years  *  365 )
    }
```

Figure 9. An example of the steps taken to build a simple function in SKot

selected deletes the argument and turns the function into a nonary function. Selecting the grayed-out field allows the user to add a new argument to the function.

Pressing right focuses the function's return type. SKot allows any alphanumeric character here as long as it does not start with a number.

Pressing down or right again focuses the function's body, where new statements can be created.

3.1.2. If and Else clauses

```
if ( true ) {

} else {

}
```

Figure 10. An if-else block in SKot

Figure 10 illustrates what If and Else clauses look like in SKot. To recreate them, the user would have to type the string "if" followed by pressing the space bar or opening parenthesis. At that point, SKot would recognize that an if clause is being typed and fill in all the necessary fields to ensure syntactical soundness: a condition (enclosed by parenthesis) where an Expression can be typed, and a body (enclosed by curly braces). An optional else block is also presented and is grayed-out until there is content inside its body. Else-ifs can be achieved by creating another if clause inside the else.

After SKot recognizes the if clause, the condition is focused and the user can start editing it. Pressing down or right focuses the if clause's body, where new statements can be created. Pressing down again until the line inside the else body is focused allows the user to type statements inside the else body.

3.1.3. While loops

```
while ( false ) {

}
```

Figure 11. A While loop in SKot

Very similar to an If-Else block, the while loop, as illustrated in Figure 11 has a keyword (in this case while instead of if) followed by a condition above a body.

To recreate it in SKot, the user would have to type the string “while” followed by pressing the space bar or opening parenthesis. At that point, SKot would recognize that a function is being typed and fill in all the necessary fields for a valid While loop: a condition and a body.

After SKot recognizes the while loop, the condition is focused and the user can start editing it. Pressing down or right focuses the while loop’s body, where new statements can be created.

3.1.4. Expressions

```
var age = 49
val ageLimit = 50

while (( age >= ageLimit )) {
  age =( age + 1 )
}
```

Figure 12. Highlighted examples of Expressions in SKot

In Figure 12, highlighted in green for the purpose of illustration, are examples of expressions in SKot. Expressions represent a value in one way or another: either by being a primitive (like a number, string or boolean) or an instance of a known class; or by being a value stored in a variable or an array position or object property; or by representing an arithmetic operation over other expressions, a composed expression.

An Expression can always be found on the right side of an equals sign; as a condition for an If clause or a While loop; as a return value.

Composed expressions are created by typing an operator symbol while editing a simple expression. The operator can then be altered into another operator and members of the composed expression can also be deleted, deleting the operator preceding it too.

3.1.5. Declarations

Figure 12 includes two examples of declarations in the first and second line. Declarations can initialize either values (val) or variables (var). The former is a constant and cannot be reassigned while the latter can.

To create a declaration, the user would type either "var" or "val" followed by a space. SKot would then recognize a declaration and prepare fields for the declaration, leaving the user's focus on the variable's (or value's) name field and pre-filling the value after the equals sign with 0.

If the user changes their mind after creating the declaration, they can click "var" or "val" to toggle between the declaring a variable or a value.

3.1.6. Assignments

Figure 12 also includes an example of an assignment in the second to last line, where age is assigned a value one unit higher than it was.

After writing anything that could be a variable's name followed by the equals sign, SKot would recognize an assignment and focus the field for the assignment's value.

3.1.7. Function calls

```
fun toDays (years : Number   :   ):Number {
  return( years  *  365 )
}

var ageInDays = toDays ( 28 )
```

Figure 13. A function call right after its declaration in SKot

A Function call can be written directly on a line, usually when the function has side effects and the return value is not relevant, or on an expression. In the latter case, the return value is used for the value of the expression.

To write a function call in SKot, after writing the function's name, opening parenthesis indicates that the user is trying to call a function. At that point, SKot would recognize

what is being written as a function call and close the parenthesis after pre-filling fields for the function's arguments, if the editor found a matching function name.

3.1.8. Return Statements

Return statements, as seen in Figure 13, are a simple keyword followed by an expression, which can be empty.

To indicate a return statement, the user would type "return" in an empty line followed by a space.

3.1.9. Inputs

In the web browser, there is always one (and only one) element in focus, and it is usally represented with a rounded blue outline (see Figure 11) though some web pages might render it in a different way. Being the element that is focused means that is the field where the keyboard cursor is and if the user does anything on the keyboard, that focused element might react in some way.

A keystroke might be a letter, number or other symbol, but it also might be a backspace (delete), a return, or an arrow key.

3.1.9.1. *Symbols* Pressing a key on an input in SKot, in most cases, needs to be validated to make sure what is being typed belongs in the field that input element is for. This is done with a customizable regex that depending on the needs of the field, should allow or not adding the symbol typed to the field.

One such example is where a variable or function name should consist of a limited alphabet of characters and never start with a number. Figure 14 is the regular expression for that input validation.

```
const EXPRESSION_REGEX = /[a-zA-Z_$]+[a-zA-Z0-9()_$[\].]*/gm;
```

Figure 14. This regular expression accepts a letter, underscore or dolar sign symbols as first caracters, followed by 0 or more of characters like those (and some more options too)

3.1.9.2. *Backspace* Pressing backspace on a filled input will delete characters from it. The amount of characters depends on the position of the text cursor and the size of the current selection. If, for example, the user focuses the input field using the arrow

keys, the content of the input will be fully selected, allowing the user to either replace or delete the selected value in the next keystroke.

When the text cursor is at the first position in the input (or it is empty) and the backspace key is pressed, depending on the context of the input, something different might happen.

In an if's condition, for example, if the user deletes the condition and proceeds to press backspace, the if header is deleted (along with the closing bracket) and the contents of the if are moved to the statement's parent.

In some situations, if an input is unfocused and its content is empty, depending on the context of the input, SKot might decide on a default value for the field (a function or variable name), or delete the field altogether (a function's argument or a simple expression).

3.1.9.3. *Arrow Keys* When a text field is in focus, the position of the cursor matters to where a character is added or which is deleted. The browser can manage the cursor inside a text field as expected (Pressing left or right moves the cursor along the characters in the field) but when the cursor is at the edges and the user presses the arrow that points towards it the focused element should change in SKot.

Inputs are one of the few things in SKot that are focusable and when the Input Component detects a keystroke of an arrow key, it first checks if the keyboard's cursor is at the edge the arrow pressed is pointing to. Then the component needs to look up the closest focusable component to it in that direction and focuses it.

3.2. Limitations

Compared to a text editor, a structured editor struggles with tasks that might seem trivial for an experienced programmer.

One simple example is to cut and paste text: On a structured editor, there might be a mechanism to move or copy statements around the program, but at all stages of the edit, the program has to be syntactically correct. On a text editor, the user has full control (to the character) of what is being cut and can place it wherever in the file. This

might create an error due to some misplaced character but an experienced programmer can fix it in less time than the amount saved by doing the risky edit.

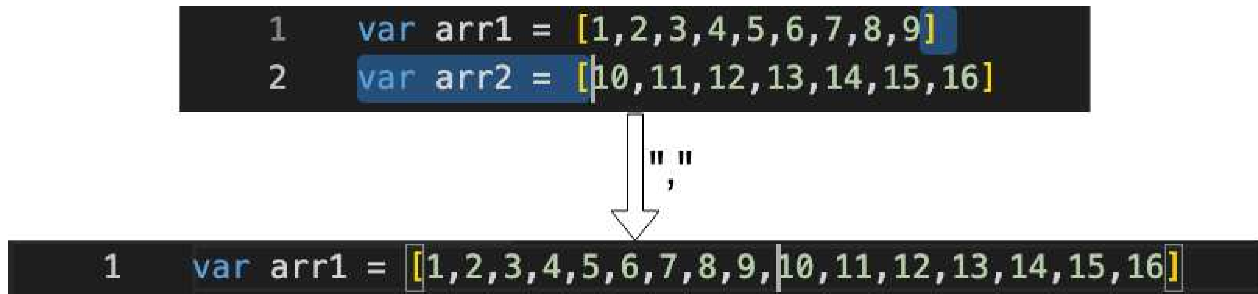

Figure 15. With a single stroke of “,” the text editor merges 2 arrays

Another example of an action that is trivial in a text editor but not in a Structured editor (particularly SKot) is given in Figure 15. In a single stroke, the text editor adds the content of the second array to the first. For a structured editor, it can either tackle the problem by creating a user experience that deviates from what is available in a text editor, or programmatically by figuring out what the user selection corresponds to in the underlying data structure of the program and approximate the next keystroke to a change in that structure.

Limitations like the ones described by these examples can be worked around by developing another user experience or present difficult problems for the developer of the editor to mimic a functionalities that is a given in all aspects of modern text editors.

By the time students learn to code, they will, most likely, be acquainted with other text editors to create documents and presentations and be familiar with mechanisms like copy and paste. Not having access to those tools in a program that looks like every other text editor might be detrimental to the user experience, and delay the acquisition of skills that would be transferable to a text-based editor.

CHAPTER 4

Implementation

In order to realize the specifications outlined in Chapter 3, the implementation followed the same architecture as a Projectional editor, where the Abstract Syntax Tree (AST) informs the app what to render and the app will perform edits to the AST through well defined user interactions.

As mentioned before, SKot will run in the browser, so web technologies were used to implement it. One relevant aspect of the web browser for the development of this editor is its rendering engine. HTML documents are already structured text. There are elements, inside divs, inside sections, inside a body, forming a distinct hierarchy. This is relevant to the way web development is done.

Most front-end frameworks (like Angular, Vue or React [16]) implement the notion of components, which are pieces of the User Interface (UI) that can be instantiated and repeated across the web app and in some cases, inside each other. For a Structured Code Editor, using components as the code blocks makes sense since they are also pieces of the UI that will be repeated throughout the page.

For this work, React was used, for the following reasons:

- React is a well established framework, that has plenty of documentation and code examples.

- It has other frameworks and libraries that are compatible with it, like Redux, which will be useful to simplify the implementation of the AST.

Using a framework works as a shortcut to a component system that works and is well documented. Since that is not the focus of this work, spending time creating a custom-made rendering system would be time-consuming and less efficient.

The React framework itself, however, is not equipped to handle interactions that affect the state of the application in an effective manner. A component can easily affect

itself and its children by changing their properties (props), and it can affect its parent by triggering an event. The parent can then affect other children or propagate the event to its parent, but that can become hard to maintain as the application is developed and grows in scope.

One solution is to add State by using another framework known as Redux. Redux State can be seen as the Single Source of Truth to the web app and if the state changes, the relevant components that are being rendered are updated with those changes.

The solution implemented by this work has its foundation on three pillars. The Abstract Syntax Tree (AST), components as code blocks and user interactions.

4.1. The Abstract Syntax Tree

The AST is the data structure that represents the current state of the code being written. It must always represent a syntactically valid program, and will be used to render an interface for the user to interact with.

Figure 16 provides a high-level view of the AST Model. In summary, the Module, being the root of the AST contain many functions, that in turn have parameters and other Statements in the body.

Statements are an abstract concept that can be either Assignments, Declarations, While loops, If-Else clauses or Return statements. While loops and If-Else clauses can, in turn, contain other statements. All statements, then can include an Expression either in their condition (in the case of If-else clauses) or value (in the case of Declarations, Assignments or Returns)

Expressions can contain other Expressions (in which case it is designated a Binary Expression), and can also be a Function Call, which refers to a function by name and can take other Expressions as arguments.

In this work's implementation, the Redux State is the perfect candidate to work as the AST. Redux can store the AST as a JSON object and provide the React components portions of that data structure to render them for the user.

Module

Function
name: string
returnType: string

Parameter
name: string
type: string

<<Abstract>>
Statement

Declaration
name: String
isVariable: boolean

Assignment
name: String

Return

IfElse

While

Expression

FunctionCall

Binary Expression
operator: Operator

body
then
else
condition
Value
args
1
0..1
2
*

Figure 16. A UML representation of the Abstract Syntax Tree

4.1.1. Code in the AST

4.1.1.1. *Composed expressions* can be represented as a binary tree, as exemplified in Figure 17, where every node represents an operation, and each edge points to either another operation or a value.

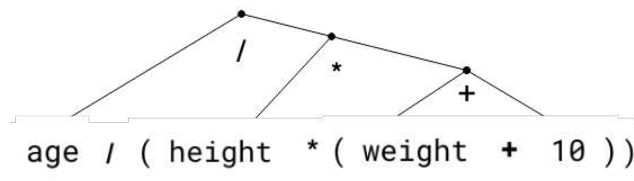

Figure 17. An expression represented as a Binary Tree

```
{
  "_type": "BINARY_EXPRESSION",
  "operator": "/",
  "content": [
    {
      "_type": "EXPRESSION",
      "content": "age"
    },
    {
      "_type": "BINARY_EXPRESSION",
      "operator": "*",
      "content": [
        {
          "_type": "EXPRESSION",
          "content": "height"
        },
        {
          "_type": "BINARY_EXPRESSION",
          "operator": "+",
          "content": [
            {
              "_type": "EXPRESSION",
              "content": "weight"
            },
            {
              "_type": "EXPRESSION",
              "content": "10"
            }
          ]
        }
      ]
    }
  ]
}
```

Figure 18. A composed expression in the AST

As a data structure in the AST, the composed expression from Figure 17 resembles something like the code snippet in Figure 18. For brevity and legibility, certain values in the JSON object were omitted, such as the unique identifier for each nested object.

```
{
  "_type": "IF",
  "condition": {}, // An Expression would go here
  "elseBlock": [
    {
      "_type": "RETURN",
      "content": {
        "_type": "EXPRESSION",
        "content": "10"
      }
    }
  ],
  "statements": [
    {
      "_type": "RETURN",
      "content": {
        "_type": "EXPRESSION",
        "content": "3.14"
      }
    }
  ]
}
```

Figure 19. An If Clause in the AST

4.1.1.2. *If Clauses* need to store at least two data structures in the AST: Their condition, which is an expression just like described in Figure 18 and their statements, meaning the code that will run when the condition is true.

Additionally, they can store statements for the else block, the same way the other statements are stored.

Figure 19 includes a snippet of the AST that represents a simple if clause with two return statements. For the sake of brevity, some parts of the structure were ommited, such as the unique identifiers and the condition, which would look very similar to what is present in Figure 18.

4.2. Components

The components provide the user with a visual representation of the AST, making it look like text in the Kotlin language and provide ways to change it.

Components can be thought of as pieces of HTML code on the page that have behaviour associated with them. For example, it can be something as simple as a button that changes color after being clicked a certain number of times or a whole table that renders data coming from an external source.

4.2.1. Statement Blocks

Components can render other components (or other instances of themselves) inside them, just like a For Loop can be inside another For Loop inside a Function. This is essential in order to represent nesting in a program.

In this work's implementation, the concept of nesting is handled by a specificic componentent called the Statement Block Component. This code receives a list of statements as a property (called "prop" in React) and maps each statement from the AST to a component based on the statement's type.

4.2.2. Expression Blocks

As referred to in 3.1.4 expressions represent a value and can be either simple or composed.

4.2.2.1. *Simple expressions* Simple expressions are straight-forward in their implementation. A simple Input Component will suffice for the user to type a value or variable into.

The code snippet in Figure 20 shows how the Expression component would render a simple Expression, using the Input component with a specialized Regular expression to type a value into it, and callbacks like `onTurnIntoFunctionCall` to handle when the value being typed becomes a function call by opening parenthesis.

4.2.2.2. *Composed Expressions* When the user is typing a simple expression, if a character that can indicate an operator is pressed, the Expression Component might decide to break the expression into a composed expression, storing the current simple expression that was being typed in a new binary expression object's array of expressions, creating another simple expression and adding an operator. Each expression of that binary expression will behave as a simple expression, listening to possible operator inputs and splitting into other binary expressions.

```
return (
  <div className='Expression' onInput={this.handleInput}>
    <Input
      id={this.props.id || expression._id}
      content={expression.content}
      inline
      regex={EXPRESSION_REGEX}
      onUpdate={this.handleInputUpdate}
      onDelete={() =>
        this.props.deleteExpression({
          id: this.props.id || expression._id,
          path: this.state.path,
        })
      }
      onTurnIntoFunctionCall={(value) =>
        this.props.turnIntoFunctionCall({
          path: this.state.path,
          value,
        })
      }
    />
  </div>
);
```

Figure 20. The portion of the Expression Component that renders an Input Component

4.2.3. IfElse Clause

The IfElse Clause Component uses both Expression and Statementent Block Components to render portions of the AST such as the one represented in Figure 19.

The render function represented in Figure 21 receives a statement from its parent form a property (`this.props.statement`) that contains a condition, statements and an elseBlock, as prescribed by the AST representation of an If clause in Figure 19. It then returns some HTML elements with Kotlin syntax and other instances of components such as the Expression component, that will take care of rendering the condition, and the Statementent block, that takes care of rendering the statments and the else block statments.

```
render() {
  const { condition, statements, elseBlock } = this.props.statement;

  return (
    <section className='IfClause'>
      <header>
        <strong>if (</strong>
        <Expression
          expression={condition}
          path={this.state.path}
          stateKeys={['condition']}
          onDelete={() =>
            this.props.removeBlock({
              path: this.state.path,
            })
          }
        />
        <strong>) {`{`}</strong>
      </header>
      <StatementBlock statements={statements} path={this.state.path} />
      <header>
        <strong>{'} else {'}</strong>
      </header>
      {/* Else Statement block goes here*/}
      <strong>{'}'}</strong>
    </section>
  );
}
```

Figure 21. The render function of an If Else Component. The else statement block was ommited for brevity.

The Expression Component also receives a callback named removeBlock that will run when the Expression Component is completely deleted (when the user presses backspace and there is no content in the expression).

4.3. User Interactions

User interactions can be split into two categories:

(1) Editor controls like moving the focus.

(2) Edits to the AST like creating a function or changing a variable name.

4.3.1. Moving the focus

User interactions like moving the focus in SKot can be handled as internal component events that don't affect the AST. These are usually handled in the Input Component that itself can look for other focusable elements and change the focus to them.

A simple search algorithm based on the order that all focusable components appear in the editor was implemented by getting all focusable elements using a Query selector, searching for the element with a matching id width the current input's and requesting focus to the one that is next to it.

This has the limitation that pressing up or down might lead to an input that is located to the left or right of the current input. An alternative entails using that list of focusable inputs and search the nearby inputs to find the one that is rendered closest to the current input according to the x axis if the user pressed a horizontal arrow, or the y axis if the user pressed a vertical arrow. This latter implementation was skipped to focus on more complex features.

4.3.2. Edits to the AST

Edits to the AST happen in a single component but need to be registered in the AST as the program being built needs to be updated, even if it is something as simple as updating a variable's name.

HTML elements emit events, that the React components can listen to, to indicate changes in the element itself. An input element might emit a "change" event that includes the new value of the input, for example. React components can listen to events like these to trigger methods in the component that perform other changes in the internal state of the component or call other functions or methods.

Redux also adds Actions as a mechanism to change the State of the app. These actions can be triggered from the components by the user and are made available as functions to be called when appropriate with data, if necessary.

One example of a function given to a component to trigger a Redux action can be seen in Figure 20 by the name `onTurnIntoFunctionCall` which will cause a set of changes to the AST to turn an expression's content into a function call.

In order to locate the component issuing the Action, these methods usually include a path which is a list of ids and object keys from the root of the AST to the component itself. The part of the Redux State that listens to actions then receives those parameters, follows the path given and performs the changes in the AST.

There are other solutions for finding the issuer of the Action out there that require more setup than this simple, if possibly less performant, implementation.

4.4. AST Projections

For the purpose of making the editor interactable, the HTML produced by the AST in SKot is not directly copyable to another text editor and usable as Kotlin code. SKot's rendering engine might introduce elements that are decorative and bear no meaning to the Kotlin compiler. See, for example the lonesome colon present in front of "Boolean" in Figure 22. It serves only as a visual aid for the user to indicate a possible second argument for the function, but it is not syntactically correct.

This implies that the AST needs to be rendered with a different strategy for the code to be readable by a Kotlin compiler or transferable to another text editor. The same way SKot's rendering engine uses the JSON that represents the AST to render html components, a different function can use that same object to create plain text. Figure 22 represents how SKot would render the same part of the AST as Kotlin text and in the SKot Editor.

With this same notion, the AST could also, theoretically, be used to transpile code into other languages by relating concepts that are represented in the AST, to the syntax of other languages apart from Kotlin.

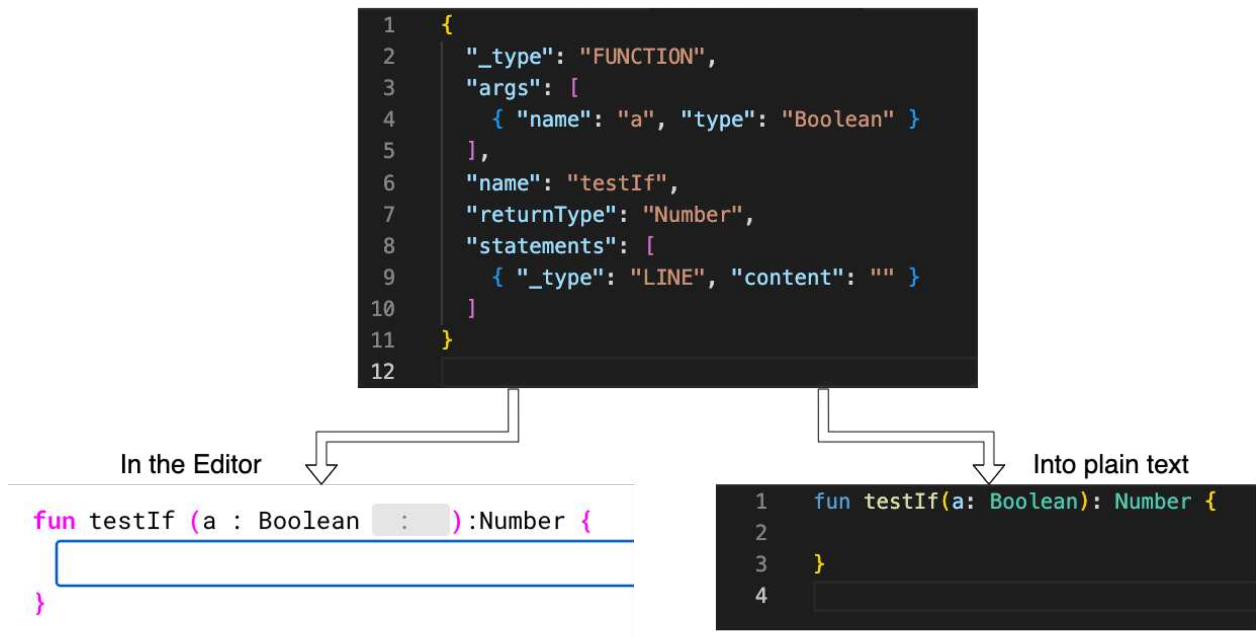

Figure 22. The same portion of the AST projected in the SKot Editor or in text

CHAPTER 5

Discussion

5.1. Comparative Analysis

Comparing the number of keystrokes needed to perform the same simple task in SKot and in a GPCE like VSCode can used as an indicator for the second research question posed in Section 1.2 regarding editing effort.

Task	VSCode with Language support	SKot
Recursive Factorial Function	96	87
Changing If to While	4	34

Table 1. Comparative analysis of different tasks in VSCode with language support and SKot

```
fun factorial(n: Number): Number {
  var next = n - 1
  if(n == 1){
    return n
  }else{
    return n*factorial(next)
  }
}
```

```
fun factorial(n: Number): Number {
  var res = n
  if(n >= 1){
    res = res * (n - 1)
    n = n - 1
  }
  return res
}
```

Figure 23. On the left, the Recursive Factorial Function. On the right, the task is to turn the if clause into a while loop

In the first exercise listed in Table 1, SKot over-performs the text editor marginally. VSCode, depending on the extensions installed, can auto-complete some structures as they are being typed. In this case an auto-complete option to create a unary function was taken, saving a considerable number of keystrokes.

On the other hand, performing the other task was not something that was included in the spec of SKot, so despite being able to delete the if clause without deleting its

contents, there was no mechanism to move those lines into a new while loop, forcing the user to re-write the contents of the clause.

Besides a mechanism to move lines, functionality to per-mutate code structures into similar ones could be developed for SKot.

5.2. Future Work

Even in its own specifications, SKot works only as a proof of concept, so what follows are suggestions on where it can be improved or other areas projectional editors can be taken.

5.2.1. User Tests

Despite being short, the comparative analysis performed in 5.1 reveals at least two shortcomings that might be addressed with more research and development.

In addition to more typing effort comparisons, user tests would be beneficial for user experience feedback and to help come up with different interactions that might be useful for real users.

5.2.2. Multi-language support

As mentioned in Section 4.4, supporting multiple languages both in the editor an in the text output would be an interesting endeavor as a playground to compare languages with custom code. Even if an editor like SKot turns out to not be adopted by students transitioning from fully structured editors like Scratch to Text Editors, further work can be done to explore the use of Projectional Editors to generate code in more than one language at the same time, or presenting it with different visual languages.

Supporting multiple languages in the editor will require a rendering system that takes into account the language selected to render the components, something that SKot is not ready for. It might also make sense to change the way certain user interactions affect the AST.

Supporting multiple languages in the text output is rather simpler. The same way object types in the AST can be mapped to text, as long as there is overlap in concepts between the languages, it should be straightforward to translate the concept to the new

language. A good first candidate for this would be Java, given its compatibility and similarity with Kotlin.

5.2.3. Converting text to Structured Code

Code editors like the one available for Unreal Engine present the possibility of creating full games and shaders with a structured editor - called Blueprints - and allow the user to convert Blueprint code to C++. The reverse is not possible (at the time of writing) but there is interoperability of classes where classes written in C++ can be used in Blueprints and vice-versa.

Converting text-based code to an AST is not impossible, since most code editors that have parsers for the language of the file being edited can show an outline of the program, but certain concepts might not be supported in the structured editor (as is the case with SKot), requiring more research and development for a better user experience than using the text editor.

5.2.4. Other possible implementations

Instead of a Projectional Editor like SKot, a possible alternative would be to create a text editor (or build on an existing text editor, like a VSCode extension) that validates every input so that the syntax validity is maintained. This, however, poses other problems.

To guarantee full syntactical validity, a text-based editors like VSCode need to parse at least parts of the code that was edited. In the case of VSCode, if, for example a curly brace is improperly removed an error appears in the editor indicating a missing character.

To follow the specifications outlined in Chapter 3, particularly of not letting the user perform inputs that would break the syntax of the program, the extension will need to intercept keystrokes before they reach the viewport and decide, based in the location of the attempted input whether to allow it (and, if appropriate add more code) or deny it. In order to make that decision a parser needs to run at least every so often to create an AST and relate parts of the text file to portions of the AST.

Depending on the kind and location of the keystroke, the AST might become outdated, and if the parser is running during the next keystroke, the extension might have to block

the input for some time until the parser finishes running. The time between pressing a key and seeing the effect on screen is called latency and in devices like a keyboard, users can detect high latency at about 90ms [17]. Making the user wait for the parser to run between inputs might introduce noticeable latency that will be detrimental to the user experience.

5.3. Conclusion

Creating a Projectional Editor like SKot requires equal parts technological development and user experience research to keep up with the editing power available in a text-based code editor. This work focused on the development a proof of concept for a projectional editor on the web that would maintain syntactical soundness during edits with support for a subset of features in the Kotlin language.

By the very nature of it being a Projectional Editor, SKot can easily ensure that whatever edits are performed will not disrupt syntactical validity. However, the limitations in edits require more research and iteration with user tests for an editor like SKot to be viable as a pedagogical tool.

Our hope is that this work can serve as a starting point for other projectional editor endeavors and/or to perform user tests with SKot to save time with the technical proof of concept.

References

[1] M. Hristova, A. Misra, M. Rutter, and R. Mercuri, "Identifying and correcting java programming errors for introductory computer science students," p. 420, 2003.

[2] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The scratch programming language and environment," *ACM Trans. Comput. Educ.*, vol. 10, 4 Nov. 2010. doi: 10.1145/1868358.1868363. [Online]. Available: https://doi.org/10.1145/1868358.1868363.

[3] D. Golovin, "Outsystems as a rapid application development platform for mobile and web applications," 2017.

[4] S. Overflow, *Stack overflow developer survey 2019*, 2019. [Online]. Available: https://insights.stackoverflow.com/survey/2019.

[5] M. Kölling, N. C. C. Brown, and A. Altadmri, "Frame-based editing: Easing the transition from blocks to text-based programming," 2015, pp. 29–38.

[6] A. Altadmri and N. C. Brown, "37 million compilations: Investigating novice programming mistakes in large-scale student data," Association for Computing Machinery, Inc, Feb. 2015, pp. 522–527, isbn: 9781450329668. doi: 10.1145/2676723.2677258.

[7] G. Rößling, A. D. Library., and A. for Computing Machinery. Special Interest Group on Computer Science Education., *Understanding the Syntax Barrier for Novices*. ACM, 2011, p. 398, isbn: 9781450306973.

[8] M. Kölling, N. C. Brown, H. Hamza, and D. McCall, "Stride in bluej - computing for all in an educational ide," Association for Computing Machinery, Inc, Feb. 2019, pp. 63–69, isbn: 9781450358903. doi: 10.1145/3287324.3287462.

[9] K. Peffers, T. Tuunanen, C. E. Gengler, *et al.*, "Design science research process: A model for producing and presenting information systems research," *CoRR*,

vol. abs/2006.02763, 2020. [Online]. Available: https://arxiv.org/abs/2006.02763.

[10] A. J. Ko, H. H. Aung, and B. A. Myers, "Design requirements for more flexible structured editors from a study of programmers' text editing," 2005, pp. 1557–1560.

[11] N. Tillmann, M. Moskal, J. D. Halleux, M. Fahndrich, and S. Burckhardt, "Touchdevelop: App development on mobile devices," 2012, pp. 1–2.

[12] A. L. Santos, "Javardise: A structured code editor for programming pedagogy in java," Association for Computing Machinery, Mar. 2020, pp. 120–125, isbn: 9781450375078. doi: 10.1145/3397537.3397561.

[13] M. Voelter, J. Siegmund, T. Berger, and B. Kolb, "Towards user-friendly projectional editors," 2014, pp. 41–61.

[14] I. Ouahbi, F. Kaddari, H. Darhmaoui, A. Elachqar, and S. Lahmine, "Learning basic programming concepts by creating games with scratch programming environment," *Procedia - Social and Behavioral Sciences*, vol. 191, pp. 1479–1482, 2015, The Proceedings of 6th World Conference on educational Sciences, issn: 1877-0428. doi: https://doi.org/10.1016/j.sbspro.2015.04.224. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877042815024842.

[15] A. J. Ko and B. A. Myers, "Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors," 2006.

[16] E. Saks, "Javascript frameworks: Angular vs react vs vue.," *HAAGA-HELIA University of Applied Sciences*, 2019. [Online]. Available: https://www.theseus.fi/bitstream/handle/10024/261970/Thesis-Elar-Saks.pdf.

[17] J. Deber, R. Jota, C. Forlines, and D. Wigdor, "How much faster is fast enough? user perception of latency & latency improvements in direct and indirect touch," 2015, pp. 1827–1836.