

**Generating Coding Exercises for Language Concepts
by Searching, Simplifying, and Annotating Existing
Code**

by

Stacia Edina Johanna

S.B. Computer Science and Engineering
Massachusetts Institute of Technology, 2021

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2022

© Massachusetts Institute of Technology 2022. All rights reserved.

Signature of Author
Department of Electrical Engineering and Computer Science
May 6, 2022

Certified by
Rob Miller
Distinguished Professor of Computer Science
Thesis Supervisor

Accepted by
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Generating Coding Exercises for Language Concepts by Searching, Simplifying, and Annotating Existing Code

by

Stacia Edina Johanna

Submitted to the
Department of Electrical Engineering and Computer Science
on May 6, 2022, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Learning a new programming language is best done through coding exercises. However, manually creating coding exercises is time-consuming because there are many language syntax and concepts to cover. PraxisGen aims to alleviate the burden of problem creation by providing an interface to search, simplify, and annotate publicly available code to be used as exercises. By using the system, the user can create code examples for language concepts that represent real-life programs more efficiently because the system automates some parts of the process. The technical evaluation proves the ability of PraxisGen in finding a wide variety of language concepts, while the user study demonstrates that the system makes the process of creating high-quality exercise problems simpler.

Thesis Supervisor: Rob Miller

Title: Distinguished Professor of Computer Science

Acknowledgments

I want to say thank you to my thesis advisor, Rob Miller, for his guidance throughout the entire course of this work. Your advice truly helps me in solving many obstacles that I faced during the project. I learned a lot through working with you, and I will remember the lessons I learned from this experience in my future career.

I also want to thank 6.031 instructors, teaching assistants, lab assistants, and fellow research members in the Usable Programming group and Software Design group for all the helpful feedback on my work. They provide a lot of help in testing the platform during different stages of my work, and I am grateful for all the constructive comments I received that helped improve the platform.

Throughout my career at MIT, I always receive the biggest support from my family. I want to say thank you to my parents and sister, Johannes, Herlianlini, and Felicia, for all your love and support. Even if we are all far apart in different countries, all of your support and encouragement help me finish my study here.

A special thank you to my dearest friends Jasmine and Jiayi for the friendship and memories throughout my undergrad and MEng here at MIT. I cherish all the time we spent together and thank you for making all the stressful times more bearable.

Contents

1	Introduction	8
2	Related Work	12
2.1	Peer Exercise Writing	12
2.2	Using Existing Code as Examples	14
2.2.1	Code Example Search	14
2.2.2	Code Example Extraction	15
3	Design	17
3.1	Exercise Creation Workflow	17
3.2	Search Query Language	19
3.2.1	Flags	20
3.2.2	Identifier’s Name and Type	21
3.3	User Interface	22
3.3.1	Concept Search Sidebar	23
3.3.2	Code Simplification Interface	27
3.3.3	Exercise Code Annotation	31
4	Implementation	34
4.1	PraxisGen as a VSCode Extension	34
4.2	Concept Local Search	35
4.2.1	Sidebar Editor for User Input	35
4.2.2	User Query Parser	36

4.2.3	TypeScript Files and User Query Abstract Syntax Trees (AST) Matching	37
4.2.4	Search Result Display and File Preview	41
4.3	Code Simplification Guide	41
4.3.1	Preparing Independent File Copy for Simplify	42
4.3.2	User Action	43
4.3.3	Split Preview Window	48
4.3.4	Finish Simplification Process	49
4.4	Exercise Annotation	49
5	Evaluation	51
5.1	Technical Evaluation	51
5.1.1	Methodology	51
5.1.2	Result	52
5.2	User Study	54
5.2.1	Methodology	54
5.2.2	Result	56
6	Discussion	63
6.1	Limitation	64
6.2	Future Work	64
7	Conclusion	66
A	Tables	67

List of Figures

1-1	A screenshot of TypeScript Tutor exercise.	9
3-1	Exercise creation workflow	18
3-2	Screenshots of PraxisGen Search sidebar	23
3-3	Using the Ignore flag on <code>for..of</code> query.	24
3-4	Using the Strict flag on empty statement query.	25
3-5	Warning box for editing read-only file.	26
3-6	A view of PraxisGen Search Help page	26
3-7	One of the examples available in the Help page	27
3-8	A view of PraxisGen Simplify	28
3-9	A view of PraxisGen Simplify with Preview panel open	30
3-10	YAML template for annotating code	32
3-11	The view of Annotate tab on the sidebar.	33
4-1	An illustration of the AST of <code>for (const x of y)</code>	38
4-2	Examples of matching and non-matching pairs of trees.	39
4-3	An excerpt of the generated <code>tsconfig.json</code> file.	43
5-1	An excerpt of the first section of the user study	55
5-2	Some examples of the exercises that the users created manually	61
5-3	Some excerpts of the exercises that the users created using PraxisGen	62

List of Tables

5.1	Overall result of the technical evaluation	52
5.2	The breakdown of the reason on the search query creation failure . .	53
5.3	The differences on the queries written by users who read and did not read the Help page	57
5.4	The search queries written by the participants.	58
5.5	The number of exercises created using both methods by all four par- ticipants.	60
A.1	A list of 110 TypeScript concepts needed for 6.031	69
A.2	A list of TypeScript concepts that are not available in the repository used for the technical evaluation.	70
A.3	A list of TypeScript concepts that the PraxisGen query language can- not represent.	70

Chapter 1

Introduction

Learning a new programming language may happen multiple times in life, especially for people who are in tech industries. Different companies or different projects may require an engineer to learn a new programming language. Even through college, a student needs to learn several programming languages for different courses and internships. Because programming is a practical subject, one of the best ways to learn to program is through practice exercises. Through practice drills, a student can master the syntax and understand the concepts of the new language [5].

The exercise problems discussed in this thesis refer to exercises intended for a student who already has experience in coding and is learning a new language. In particular, the work presented by this paper is built on a framework that is created and currently used for MIT course 6.031 Software Construction. The framework is the Tutor platform, a platform for students to learn a new programming language [17]. The way the platform works is that the student chooses a concept they want to learn, and they can see the problems on the sidebar and a code file on the editor side. On the code file, there is at least one blank that the student must fill in with the code to answer the problems. After filling in the blanks, the student can check their answer by clicking on the ‘Check’ button, and the Tutor gives immediate feedback. The system also gives explanations or hints for some questions. The intended users of this platform are students who already have a programming background in another language and are now learning a new language.

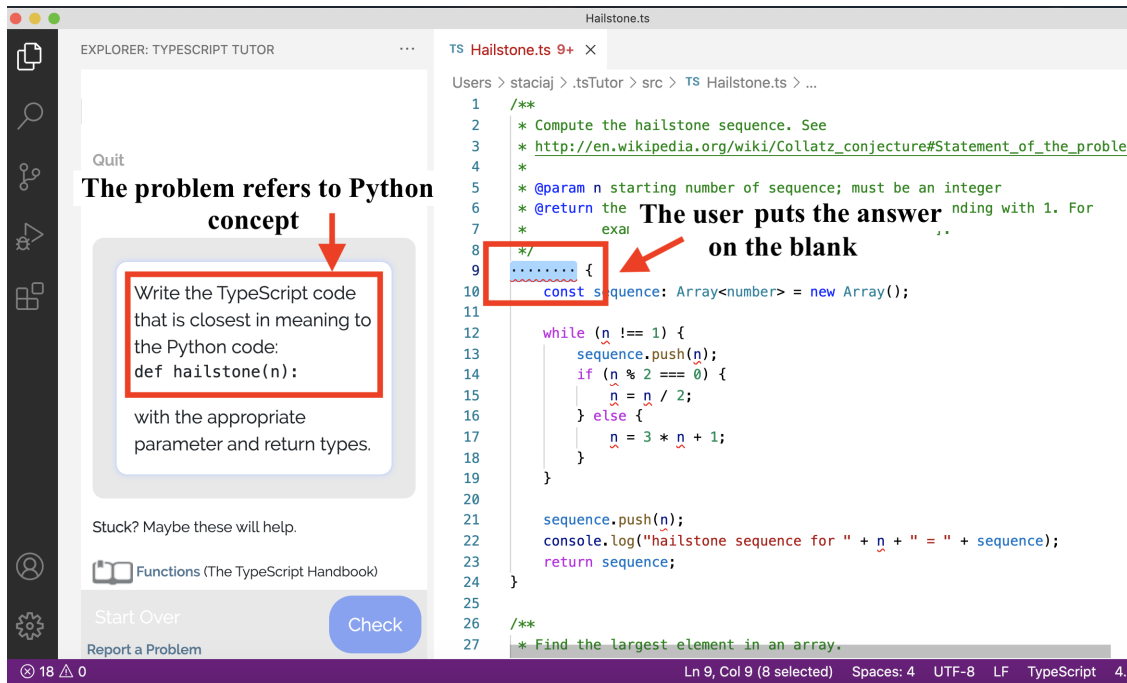


Figure 1-1: A screenshot of TypeScript Tutor exercise.

Creating programming practice is not an easy task. First, there are a lot of different concepts and language syntax to cover. As an illustration, there are 110 TypeScript concepts currently learned by the students in MIT’s 6.031 Software Construction class. Manually creating exercises for all of the concepts takes a lot of time. Moreover, a good practice suite should contain multiple problems for each concept so the student can have a variety of tasks. This multiplies the amount of exercises needed to be created. Other than that, certain concepts also have different variations in their usages. For example, the concept of for loop in TypeScript can be written in at least three different syntax: `for`, `for..in`, `for..of`. This further increases the number of exercise codes that need to be created.

A possible source that can be used for creating programming exercises is code from public repositories. Code from public repositories is a good representation of the concepts because it is the code that is used for real projects. However, this kind of code is usually very long and filled with unnecessary complex details that can confuse students and distract them from the actual concepts being learned. Therefore, it is necessary to provide exercise code that is balanced between being a good representa-

tive of the concepts while still being simple enough for students to be able to focus on the specific concepts being taught.

This thesis contributes by developing PraxisGen, a multi-part system that makes the problem creation process easier because the user does not have to write the code from scratch by searching, simplifying, and annotating existing code. First, code files from public repositories are collected to form a local repository. The files on the local repository are then used to create the code examples in the Tutor platform’s exercise problems. Then, the user specifies which concept they want to create an exercise for using a search query, and PraxisGen searches on the local repository and outputs a list of locations in which the concept is located. After that, the system provides an interactive guide for the user to semi-automatically reduce down the file to remove unrelated details while still keeping the relevant context for the exercise. Finally, the user writes the related prompts, hints, and explanations for the exercise code on the annotation interface.

In this iteration of the system, PraxisGen is in the form of Visual Studio Code (VSCode) extension. The exercise created by the system is a TypeScript exercise that is compatible with the Tutor platform, the coding practice platform used in 6.031 [17]. During its computation in each part of the system, PraxisGen heavily uses the information provided by the Abstract Syntax Tree (AST) of the TypeScript search query and files in the local repository. This information is acquired with the help of TypeScript Compiler API [12].

The technical evaluation of PraxisGen demonstrates its ability in providing the tool to create coding exercises of a wide range of TypeScript concepts. Among 110 TypeScript concepts tested, the query language used by the system can represent 94 of them. And for all 94 concepts searched on the selected local repository, the first ten result produced by the system accurately contains the concept.

The user study proves that the system helps the user in making the exercise problem creation more efficient. In the user study, the participants create some TypeScript code examples for three different concepts suitable for the Tutor platform. After the study participants create some exercises manually and using PraxisGen, it

can be seen that using the system has several advantages. Without using the system, the participants have to think of the variations on how the concept used themselves. However, by using the system, the users only need to choose from the available options in the repository. Also, because the system uses public repositories as resources, the resulting exercise created using PraxisGen contains enough context on how the concept is used in real life programs. On average, the code examples produced by the participants using PraxisGen have more variety than the ones created manually.

Chapter 2

Related Work

2.1 Peer Exercise Writing

An alternative to manual exercise creation is peer exercise writing, in which students write coding exercises that are then solved by other students. This approach distributes the amount of work needed to create the exercises originally intended for a few staff members to a much larger amount of people. Other than having a larger quantity of exercise problems created by a larger number of human resources, one advantage of peer exercise writing is that it encourages the student to fully understand the concept they are learning and express their ideas clearly [9]. This is because the problems they created are used and evaluated by other students, so they are usually more careful when writing the problems.

There are many systems which are based on peer exercise writing and are currently used in classes. One example is CodeWrite, a web-based system for students to write short programming exercises that are shared and practiced by other students in a shared repository [5]. In CodeWrite, the student provides a detailed description of the methods that the other students have to write. This description includes the purpose of the method, some sample inputs and outputs, and some test cases for the methods created. Then, the author of the problem must provide an implementation of the method they just designed. After all parts of the problem is written, the exercise is published for other students to practice on. The result in the paper shows that

the students create hundreds of questions using the full range of the syntax taught in the class. A large percentage of the students in the class also uses CodeWrite to learn various problems covering different topics, thus proving the effectiveness of peer exercise writing as a resource for students to do programming exercises.

Another type of the exercise created through peer exercise writing is coding-related questions. Some examples of platforms that support this type of exercise writing are StudySieve [9] and PeerWise [4]. In StudySieve, the student creates free-response questions in which other students can provide answers for and rate the quality of the questions. Similarly, PeerWise is a platform for a student to write multiple choice questions that other students can practice on and give feedback in the form of comments. The results in both papers show that there is a high level of engagement to the systems. Because the systems are used in large classes of around 400-500 students, the number of exercises created using the system is also high. That number of exercises is definitely much more difficult to reach if only less than 10 course staff members write them.

One drawback of this approach is that peer problem creation increases course loads as it is a much more involved process than just answering questions [1]. This approach also increases the instructor's workload because they have to design a procedure for the peer exercise writing so that the created have adequate quality for it to be used as exercise. A good procedure is needed to prevent questions that are too simple or a duplicate of each other, which is the problem found during the evaluation of StudySieve [9].

To prevent added burden to students in their learning experience, PraxisGen focuses on programming exercises created by the instructor. So, to reduce the amount of work that needs to be done by the instructor, the system needs to make the problem creation process more efficient, and it is done by automating some parts of the process.

2.2 Using Existing Code as Examples

Code examples have been used for various purposes. Instead of writing those examples manually, many platforms take code snippets from public domains. Some platforms, such as Bing Developer Assistant [21], XSnippet [19], and Blueprint [2] gather code snippets directly from the public domain to help a developer in their work. Using those platforms, developers can refer to the code examples for recalling how certain libraries and methods are used. Other platforms like APIMiner [14], MUSE [15], and UsETeC [22] first gather public resources into curated internal repositories before using them. All three platforms extract code examples from the files in the local repositories to solve the problem of lack of concrete examples in API documentation. Similarly, PraxisGen gathers the files from public repositories such as Github and collects them into a local repository before using them for exercise problem creation.

To use the files gathered from various resources, the platforms mentioned above provide different ways for the user to search the specific example they want and different methods in which the platforms acquire relevant code to be used as the example. Both of these processes are described in more detail below.

2.2.1 Code Example Search

There are several ways in which a user can search for specific example code. First is by natural language query, which is the same type of query if a user is searching for code examples from Google. A few examples of platforms that use this type of query language are Bing Developer Assistant and Blueprint. Both of them work by connecting the platforms to established search engines, and processing the result acquired from those engines. They also augmented the user query with the context the query is sent, such as the programming language the user is using, the error received by the user, and the APIs the user is using. While this type of query is easy to use for the user, it is not suitable for PraxisGen because the system is searching code files in a repository. The implementation for this type of query would be complicated because the system has to parse the natural language and convert it into a code.

Another type of search query is by processing user code selection. In Sporq [16] and XSnippet, the user selects parts of the code in the repository that exemplifies the concept they want to request. The system then searches all code locations in that repository which has a similar structure to the selected code. This type of query is similar to the one used by PraxisGen, but the way the user inputs the query is different. In Sporq and XSnippet, the user has to specifically look for a code that contains the concept before they can search. In PraxisGen, the user can write the code examples that represent their request themselves, so they do not have to manually look for those concepts in the repository.

2.2.2 Code Example Extraction

There are various methods used by all of those platforms to get the relevant code snippets from their resources. One method is by converting the code they have into a form of graph, then traversing through it starting from the method requested by the user. This method is used by API Synthesize [3] and XSnippet in their implementation. In API Synthesize, the algorithm first created a graph that depicts method call sequences. Then, the algorithm uses data-flow analysis to gather important details, such as type declarations and method usages. Similarly, in XSnippet, the system created a graph representation of the Java code, then traverses the graph using BF-SMINE, an algorithm they designed that focuses on gathering information on how certain methods are initialized.

Other than constructing a graph to traverse on, a more widely used method for extracting code examples is by program slicing. CodeScoop [8], APIMiner, MUSE, and UseTeC uses slicing principles to get the relevant code parts in their platform. In CodeScoop, the user extracts examples by first selecting the code lines they want to get and indicating that they want to keep the line. Then, the system creates a copy of that line into a separate pane and automatically produces a skeleton code that wraps that line. The system then interactively gives suggestions on which other parts of the code that the user may want to add to the skeleton, and the user just needs to click buttons to add or remove them. In the implementation of CodeScoop,

it uses an extension of program slicing technique by making the process interactive.

The design of the PraxisGen Simplify is similar to the one used in CodeScoop. In PraxisGen, the user also has to select which lines they want to keep or remove and specifically indicate it by clicking on buttons. However, the system also allows the user to manually edit the file while simplifying it, unlike CodeScoop in which the user has to rely only on the suggestions the system gives. This give PraxisGen's users more flexibility in how they want the final code looks like.

Chapter 3

Design

PraxisGen provides the facility for the user to create coding exercises from start to end. In the current iteration, the exercise created using this system can be used as an exercise to learn the TypeScript language in the TypeScript Tutor. This section describes the detail of the workflow of exercise creation and gives a preview of the interface in which the user engages.

3.1 Exercise Creation Workflow

Coding exercise creation using PraxisGen consists of five components. First, the user collects TypeScript files to be used as resources in the local repository. The files collected should have satisfactory quality: they should be acquired from trusted repositories with high-quality code. This is because the resulting exercise code that is used by the student should be a representative of what real-life good coding is, so the resources used must also have adequate quality.

After the local repository has enough TypeScript files containing various TypeScript concepts, the user starts creating individual exercises by searching for a certain concept in that repository. The user then can choose a file from the resulting list, and this file will be used for the exercise.

Step three of the exercise creation is simplifying the TypeScript file. Because the file in the local repository is acquired from the public domain, most of them are longer

than necessary and contain a lot of information not related to the concept taught to the students. So, PraxisGen provides a tool to reduce the length of the file so that the exercise code is short enough for it to not be confusing but still has enough context surrounding the learned concept.

After the file is simplified, the user adds all the necessary information about the problems. This includes the prompts, answer explanation, correct answers, and hints.

Finally, the annotated exercise file is submitted by the system to the TypeScript Tutor server, so the exercise can be playtested directly on the Tutor system. Because both PraxisGen and TypeScript Tutor are VSCode extensions, the change between the two systems is seamless: the user only needs to change the sidebar view from PraxisGen to TypeScript Tutor. Playtesting directly at the Tutor system also allows the user to make sure that all parts of the new exercise are correctly rendered on the same system that is used by the students for learning.

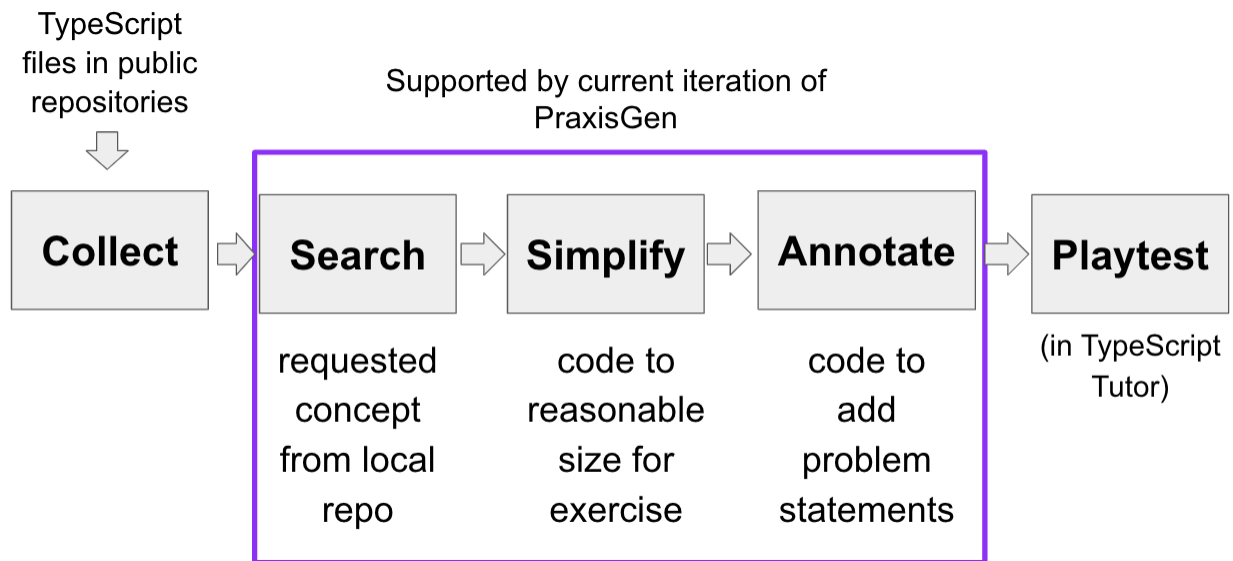


Figure 3-1: Five parts system for semi-automatically creating code for exercise using PraxisGen.

In the current iteration of PraxisGen, the system provides the tool for the user to search for a specific concept in a local repository, simplify a specific file, and annotate the file with all the information related to the exercise. However, gathering TypeScript resources from the public domain and playtesting the exercise in TypeScript Tutor is

considered out of scope for this thesis.

To work around that, the TypeScript files can be acquired manually by looking at Github repositories that have a large number of TypeScript files and a high number of engagement, such as a high number of forks and stars. The user can clone those repositories and use them as the local repository.

For playtesting the exercise in TypeScript Tutor, the user can upload all the created exercises directly to the Mongo database used by the Tutor, and access the exercise in the Tutor system.

3.2 Search Query Language

To look for a specific TypeScript concept, the user must put in a search query at the code editor on the sidebar. The search query language used in PraxisGen is in the form of a simple example of the concept in TypeScript code. This format is chosen based on the assumption that the user who is going to create the TypeScript exercise is likely to be fluent in TypeScript themselves. So, the user does not have to learn a new query language and can directly use their experience in using TypeScript to use PraxisGen Search.

While it has to resemble a TypeScript code, the query does not have to be free from compile errors. However, it has to include the core parts of the concept syntax. These core parts consist of all code keywords that usually appears in the concept and identifiers that may appear between them. The ordering of these parts should also be the same as when the user write the code for the concept. Some examples of the queries and the expected results:

Concept	Search Query	Example Code ... in the result	
		included	not included
for..of	<code>for(const _x of _y)</code>	<code>for(const e of arr)</code>	<code>for(let e of arr)</code>
if::else	<code>if() else</code>	<code>if (x > 5) else</code>	<code>if (x > 5)</code>
print	<code>console.log('a' + 'b')</code>	<code>console.log('i' + 'am');</code>	<code>console.log(var_x)</code>

For example, if the user wants to look for code that contains `for..of` concept, one way to write the query is: `for (const _x of _y) {}`. In this example, because `_x` and `_y` are not declared, the code contains compilation errors. However, the code contains the core syntax of `for..of` concept, which is the keywords `for`, `of` with the parenthesis around `of`. Thus, the query contains enough information for the system to deduce that the user is looking for code examples for `for..of`.

3.2.1 Flags

To improve the result of the search query and to be able to represent more diverse concepts, there are several flags that the user can use to annotate their query. There are the Ignore flag and the Strict flag.

The Ignore flag is used to indicate parts of the query code that the system can safely overlook. Using the example of `for..of` from section 3.2, if the user adds the Ignore flag on `const _x` and `_y`, PraxisGen result contains all location of `for..of` location with `let`, `var`, or `const` on the variable declaration inside the parenthesis instead of variable declaration with only `const`. Some other examples of the queries with the Ignore flags can be seen below. The strike-through on the query indicates parts of the code that is annotated by the Ignore flag.

Concept	Search Query	Example Code ... in the result	
		included	not included
for..of	for(const _x of _y)	for(let elem of arr)	for(const idx in arr)
arithmetic	_a * _b	str.length * 3	-
print	console.log(!a² + !b²)	console.log(var_x + 'a')	console.log(var_x + s.length)

The Strict flags are used to indicate parts of the query that have to be matched strictly. This means that the parts of the output with the strict marking cannot contain code that is not written in the query. For example, to search for the empty statement concept, the query of `'{}'` must be annotated by this flag. When the Strict flag is added, PraxisGen only includes code in which there is no code between the curly braces in the output. Another example involving the Strict flag is in the table

below. The parts of the code annotated by the Strict flag is indicated by the yellow highlight.

Concept	Search Query	Example Code ... in the result	
		included	not included
empty-statement	<code>}</code>	<code>class C {}</code>	<code>class C { constructor() {...} }</code>
static-typing	<code>let <u>x</u>: any</code>	<code>const s: string</code>	<code>const s = (s.length < 5) ? x : y</code>

3.2.2 Identifier's Name and Type

There is also an additional feature that PraxisGen provides for identifiers in the user's query. An identifier is an element in code, such as a variable, a function, and a class, and it is usually the part of the code that is not a TypeScript keyword. In PraxisGen, by default, all identifier names and types are matched exactly like how they are written in the query. For example, if the variable's name is `x`, PraxisGen only looks for variables with the name `x` in the local repositories. If the identifier has a type of `any`, it is matched with all possible types of identifier.

However, matching the identifier's name exactly as it is written in the query can be too restricting, and the user may only care about the variable's type but not its exact name. To allow any identifier name, PraxisGen processes all identifiers whose name starts with `'_'` by matching them to identifiers with any name.

Concept	Search Query	Example Code ... in the result	
		included	not included
string::length	let _s: string _s.length	x.length (in which x is string)	a.length (in which a is an array)
			s.substring() (in which s is a string)
number::equality	let _x: number let _y: number _x == _y	x == y (in which both x and y are numbers)	s.substring(..) == x

In the first example in the table above, the main part of the query is the concept the user is searching for, which is `_s.length`. The user uses `_s` instead of `s` because they want the identifier to match variables with any name, not just variables with the name `s`. Then, to make sure that `_s` is of type `string` instead of `any`, the user declares `_s` as `string`. Because this line is not the main part of the concept, the user adds the Ignore flag on this whole line. This is because the user also wants to accept string identifiers from any source (e.g. variable declaration, function parameter). Then, during the matching, the system only produces output that contains `.length` of a string identifier.

3.3 User Interface

PraxisGen is an interface in the form of a Visual Studio Code (VSCode) extension. This form is chosen instead of other forms such as a web interface or a separate application because TypeScript coding is usually done in an IDE. Therefore, having a familiar environment to work in helps the user in having one less thing to get used to. VSCode is chosen in particular because it is one of the most popular IDEs for TypeScript and JavaScript development [7][20].

In VSCode, the user can access the extension by clicking on the PraxisGen icon at the Activity Bar on the left side of the IDE. Then, each part of the system can be accessed by clicking on different tabs on the sidebar. A screenshot of the sidebar

can be seen in figure 3-2a . The sections below detail the user interface for Search, Simplify, and Annotate parts of exercise code creation using PraxisGen.

3.3.1 Concept Search Sidebar

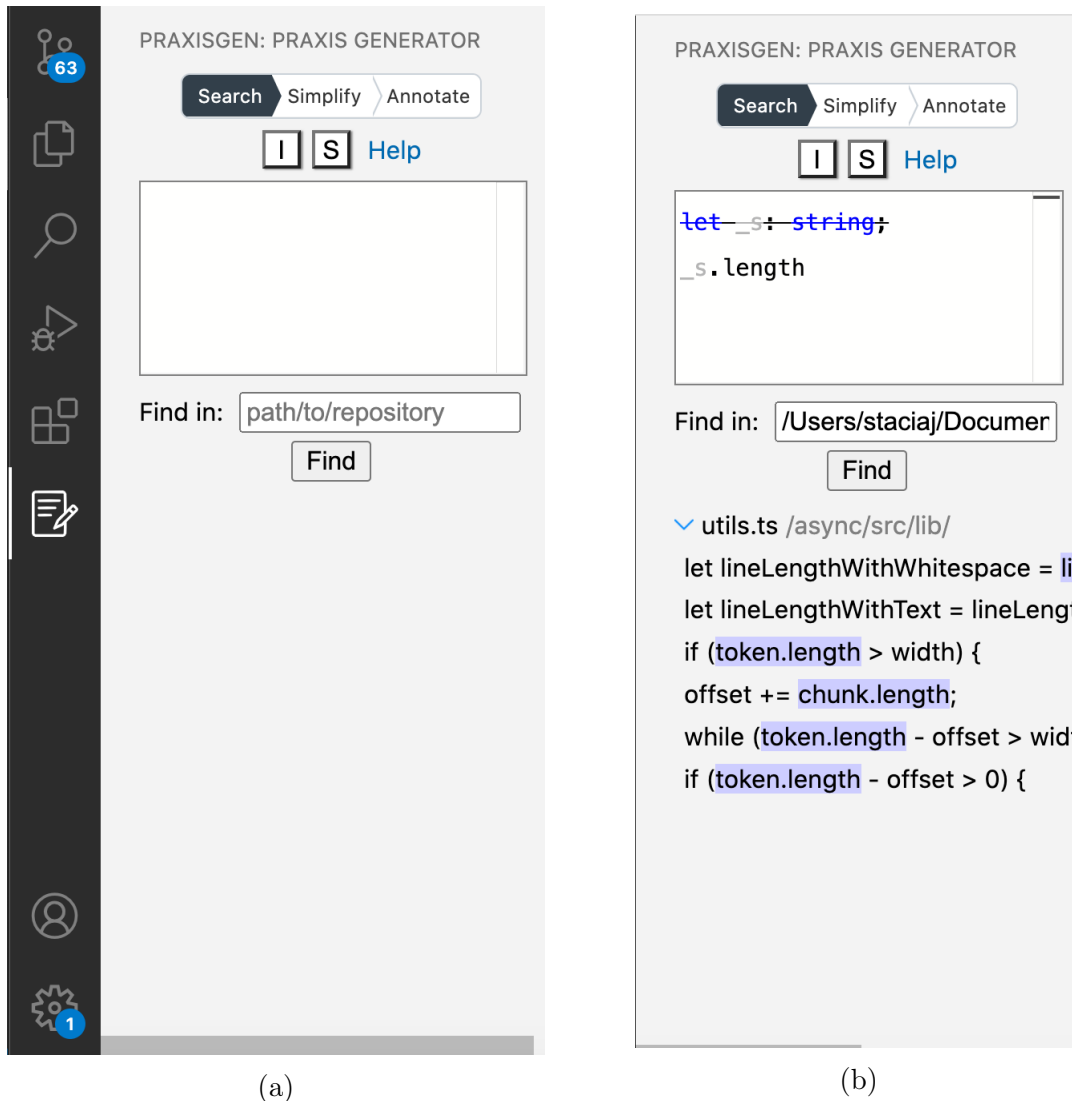


Figure 3-2: Screenshots of PraxisGen Search sidebar. (a) The view of the sidebar when it is empty. (b) The view of the sidebar with the search query and result.

When the user first clicks on the PraxisGen icon at VSCode's Activity Bar, the Search tab is automatically open. On the sidebar, there is a mini code editor where the user puts their query, an input box to put the file path to the local repository, and some buttons related to Search functionalities. Screenshots of the Search sidebar can be

seen in figure 3-2. The following subsections will describe how to use PraxisGen Search in detail.

3.3.1.1 Search Query Interface

The user can write their query on a mini editor located on the sidebar. The sidebar also contains an 'I' button that indicate the Ignore flag and an 'S' button that indicate the Strict flag. The way the user can use these flags is by selecting the part of the code that they want to annotate and clicking on the corresponding button. To remove the annotation, they can select the part and click on the button again.

On the mini code editor, the parts of the code that are ignored are annotated by a strikethrough.

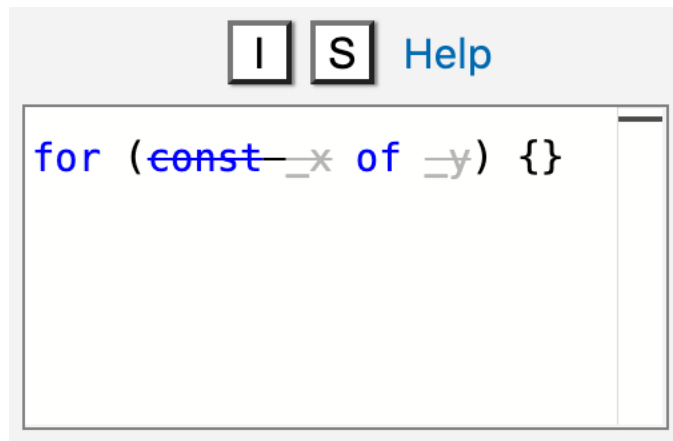


Figure 3-3: Using the Ignore flag on for..of query.

To easily differentiate with identifiers whose name is matched exactly, the identifiers with '_' at the front will be greyed out in the sidebar editor.

The code annotated by the Strict flag is shown with a highlighted background and a superscript S.



Figure 3-4: Using the Strict flag on empty statement query.

3.3.1.2 Search Result and File Preview

Once the user finishes writing the query and clicks on the Find button, the system takes the code query and the file path to the local repository to the back end. Then, PraxisGen searches all TypeScript files that are on the local repository and looks for all locations in which the requested concept appears. The list of locations is sent back to the front end, and the extension displays the result on the sidebar. Multiple results that are located on the same file are grouped, and the user can click on the filename to make the group collapse and expand. For each result, the entire line of code in which the result appears is reported on the sidebar; however, the specific concept that is requested is highlighted so that the user can easily identify the concept on the result. To see what the sidebar looks like with the search result, see figure 3-2b.

The user can see what the file in the result looks like by clicking on it. The file chosen then opens at the VSCode editor in the form of a read-only file preview. The extension scrolls the file directly to where the concept is located. The opened file is read-only to prevent the user from accidentally editing the original file in the local repository while browsing for a good TypeScript file that will be used as a coding exercise in the next step. If the user tries to edit the file, an error box pops up containing a warning about editing the file and a button that can be clicked for the user to go to the next step, which is simplifying the file. The error box that appears

can be seen in figure 3-5. Another way to move on from the file selection step to file simplification is by clicking on a pencil icon that is located on the top right of the editor.

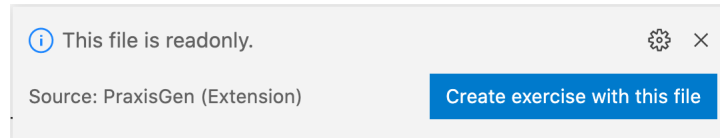


Figure 3-5: Warning box for editing read-only file.

3.3.1.3 Search Help Page

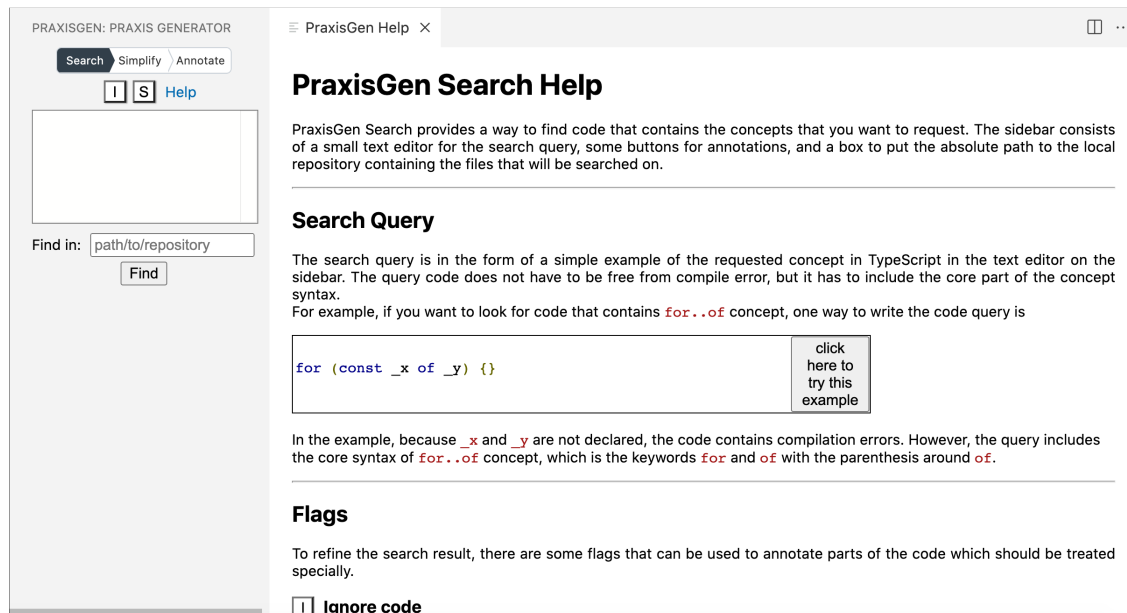


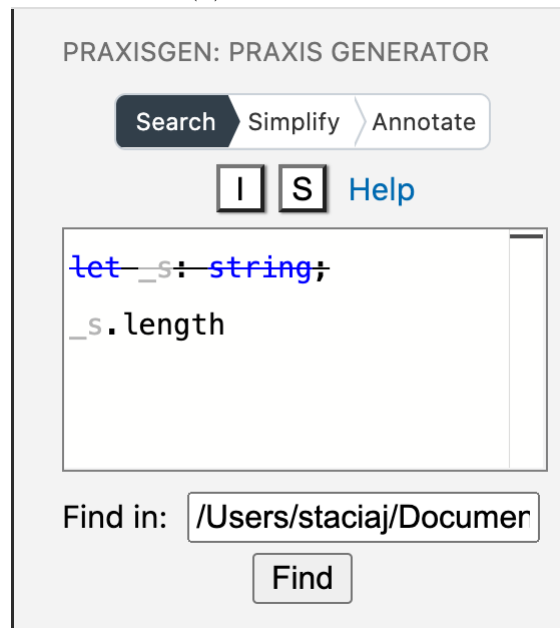
Figure 3-6: A view of PraxisGen Search Help page

To aid the user in using PraxisGen Search, the system provides a Help page. It contains guidance on the format of the Search query, which is the minimum TypeScript code that contains the core syntax of the concept. The Help page also describes the functionalities of the Ignore and Strict flags, and how to indicate whether the identifiers' name and type need to be matched in the query. On the page, PraxisGen also provides several examples that the user can refer to. The system also provides a button on the example that the user can click to automatically copy the example query and the file path of a sample repository to PraxisGen Search on the sidebar.

```
let _s: string;  
_s.length
```

click here to try this example

(a)



(b)

Figure 3-7: (a) One of the examples available in the Help page. (b) The view of the Search sidebar when the button on the example is clicked.

3.3.2 Code Simplification Interface

The next step in the coding problem creation is reducing the length of the file to an appropriate size for an exercise. PraxisGen provides the tool to do this on the Simplify tab. As seen in figure 3-8, there are two actions that the user can use: Keep and Trash. Other than Keep and Trash buttons, there is also an Open Preview button that the user can click to see what the simplified file looks like. Finally, the Finish button is clicked after the user simplifies the file, and the user can go to the next part of the exercise creation, which is adding problem statements. The way Keep, Trash, and Preview buttons work is described in more detail below.



Figure 3-8: A view of PraxisGen Simplify. Code that is highlighted green indicates that the code is Kept, and code that is highlighted orange indicates that it is Trashed.

3.3.2.1 Keep Action

To keep the parts of the file that contains the requested concept, the user selects the block in the editor, then clicks on the Keep button. The system highlights the selected part in green and added `/*[start keep]*/` and `/*[end keep]*/` tags at the end of the block. Both of them are indicators that the code in the green block must appear in the final version of the simplified file. To undo the action, the user can select the code in the green block and click on the Keep button again.

PraxisGen automatically computes which other parts of the code are needed by the green block. Parts of code that do not contain any code that is necessary for the Kept code are then greyed out by the system. A code block X is needed by another block of code Y if X contains the declarations of any variables or methods in Y, or if X is a header of a block of code needed in Y. An example of the latter is when Y contains a `return` statement, then X must include the function or method block containing that `return` statement; this is because a `return` statement without the method surrounding it can produce compile errors.

If only the header of a block is selected by the user, the system automatically

includes the whole block. This is so that the user does not have to select the whole block themselves. All code parts that are greyed out by the system are not included in the final simplified file. Code that still has its color is included in the final file, and the user does not need to manually add the Keep indicator themselves on those parts. An example of a file with user action highlights can be seen in figure 3-8.

3.3.2.2 Trash Action

When the user keeps a certain code part, PraxisGen automatically includes the whole block that surrounds the part. This sometimes causes the code file to still be large in size if the block is too long. It is possible that there are certain parts inside the block that are not actually needed by the part that contains the requested concept. To address this, the user can use the Trash action to reduce the code further. The user can select the part of the code that wants to be removed and click on Trash button. That section is then highlighted orange, greyed out, and added `/*[start trash]*/` and `/*[end trash]*/` tags at the end. Both of them are indicators that the block of code must be removed in the final version of the simplified file. To undo the action, the user can re-select the block of code and clicks the Trash button again. An example of the file with the orange highlights can be seen in figure 3-8.

If the user trashed a declaration of a variable or a method, the system looks for all locations in the file in which the variable is used and grays out all of them. This is because if the declaration is removed, not removing the variables in other places can cause a compile error. If the user accidentally does the Trash action on the declaration of code that is actually necessary for the part of the code that has Keep tags, the system produces compile error. This compile error can be handled by undoing either the Trash action or the Keep action done on the code.

While it is possible for the user to do the Trash action on code that is greyed out, it is unnecessary because all greyed out code is automatically removed. This Trash action is most effective if used in code that still has its original color once the Keep action is used. At the end of the Simplify step, the system removes all code parts that are greyed out or have orange highlighting on them.

3.3.2.3 Manual File Edit

Other than editing the file using the Keep and Trash action, the user can edit the file directly on the VSCode editor. This can be used as an alternative to using the Trash action. The user can also use this feature to improve the code based on how they want the exercise code to look by editing the file directly. The user can also manually edit the file and add or remove the Keep and Trash tags. This action produces the same result as selecting the block of code and clicking on the action buttons. When the user edits the file, PraxisGen automatically recalculates all the Keep and Trash, so the user can use the combination of manual edit and Keep and Trash buttons to simplify the file.

3.3.2.4 Final File Preview

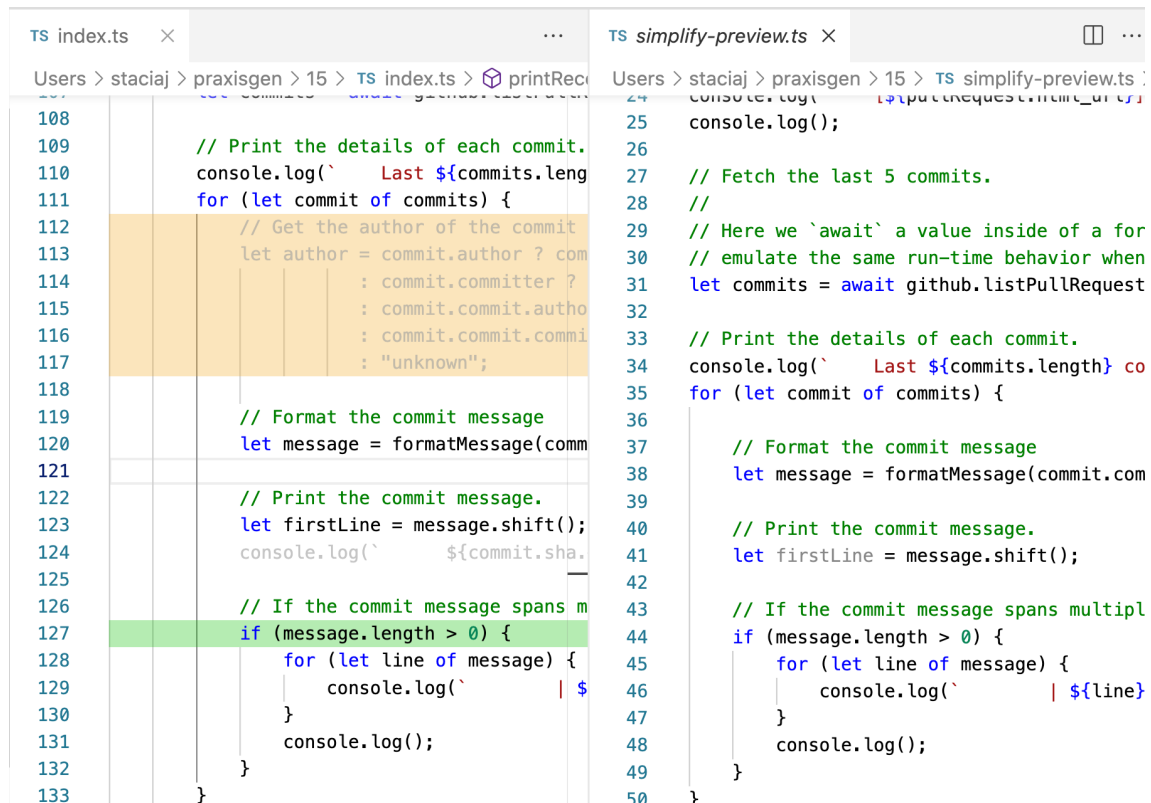


Figure 3-9: A view of PraxisGen Simplify with Preview panel open. The left panel is the original file and the right panel is the file preview.

To see what the final simplified file looks like without having to go to the next step of exercise creation, the user can use the file preview feature offered by PraxisGen. When the “Open Preview” button is clicked, a new panel opens up on the editor at the side of the original file. This panel contains a view of how the file looks like if it is simplified. This means that all code in the original file that is greyed out or highlighted orange is thrown away.

A view of the PraxisGen Simplify with the preview panel open can be seen in figure 3-9. In the screenshot, the left panel is the file that is currently being simplified, while the right panel is the preview of the final file. As seen in the figure, the code equivalent of line 127 of the original file is line 44 at the file preview, because many parts of the code before that line are removed after the user does Keep action on line 127.

The preview panel can be left open, and when the user edits the original file further, either manually or using Keep and Trash, the file preview is automatically updated. This can aid the user in having an estimate of what the final exercise file looks like.

3.3.3 Exercise Code Annotation

After the user is satisfied with the simplified code, the next step in problem creation is writing the prompts, explanations, and answers. This can be done on the Annotate tab of PraxisGen. The system uses YAML comments for adding problem statements and indicating answers so that it is compatible with the currently available Tutor system.

First, the user presses the “Add YAML header to file” button to add the YAML template to the file. Then, the user provides all the annotation information in the provided space. Below is the template given to the user:

```

//<yaml>
//  - id: <TODO>
//    conceptIds:
//      - <TODO>
//    prompts:
//      - '<TODO>'
//    explanation: <TODO>
//    blanks:
//      - code: <TODO>
//        transformers:
//          - <TODO>
//        triggeredHints:
//          - <TODO>
//</yaml>

```

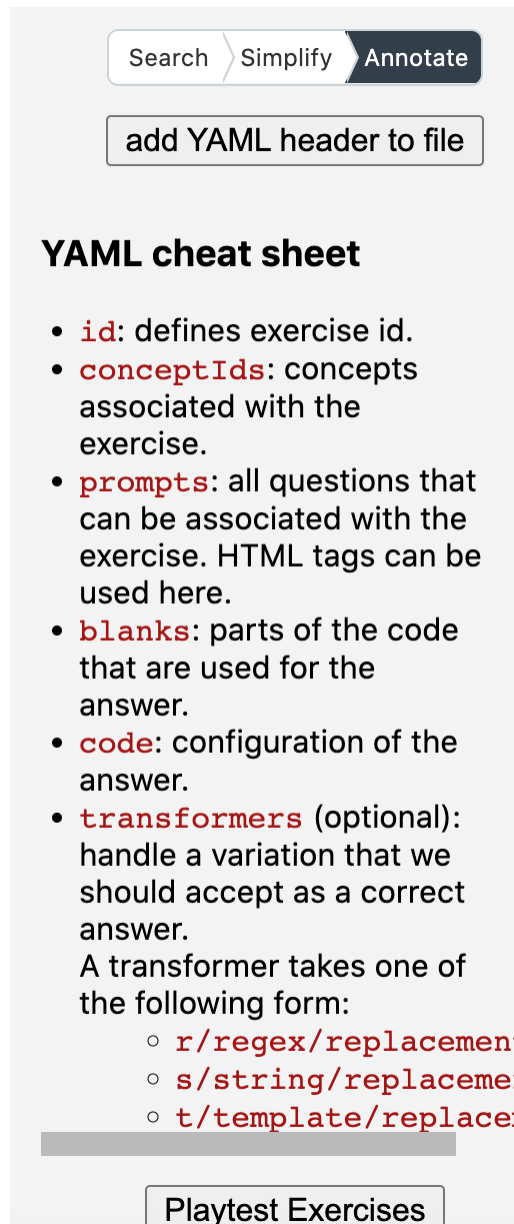
Figure 3-10: YAML template for annotating code

The user can put the problem statements in the **prompts** section. Then they can put the explanation of the answer on the **explanation**. In **blanks**, the user provides the part of the code that is substituted with dotted blanks during the exercise. The user can provide all possible answer combinations in the **transformers** section. The user can also include triggered hints for the exercise. Triggered hints are hints that appear if the student put in a predicted incorrect answer. For example, in teaching the concept of equality comparison in TypeScript, the syntax desired is using triple equal (`===`). A student may put double equal (`==`) as the answer because it is syntactically correct. However, using double equal is not ideal for TypeScript, because it does automatic type conversion, which may result in an unexpected result. So, using triggered hints, if the student puts `==` as the answer, the TypeScript Tutor regards this answer as incorrect and gave the corresponding hint written during this annotation step.

It is also possible for the user to include multiple YAML comment blocks for

different exercises that use the same file. The summarized instructions to fill the YAML comment are also provided on the sidebar.

Once the user finishes annotating the code, they can click on the “Playtest Exercises” button. PraxisGen then uploads the exercise to the Tutor server, and the user can playtest the exercise directly by visiting TypeScript Tutor on VSCode Explorer panel.



The image shows a sidebar interface with three tabs: "Search", "Simplify", and "Annotate". The "Annotate" tab is active and highlighted in dark grey. Below the tabs is a button labeled "add YAML header to file". Underneath is a section titled "YAML cheat sheet" containing a bulleted list of YAML fields: **id**, **conceptIds**, **prompts**, **blanks**, **code**, and **transformers** (optional). Below the list, it explains that a transformer takes one of three forms: `r/regex/replacement`, `s/string/replacemer`, or `t/template/replacem`. At the bottom of the sidebar is a button labeled "Playtest Exercises".

Search > Simplify > **Annotate**

add YAML header to file

YAML cheat sheet

- **id**: defines exercise id.
- **conceptIds**: concepts associated with the exercise.
- **prompts**: all questions that can be associated with the exercise. HTML tags can be used here.
- **blanks**: parts of the code that are used for the answer.
- **code**: configuration of the answer.
- **transformers** (optional): handle a variation that we should accept as a correct answer.
A transformer takes one of the following form:
 - `r/regex/replacement`
 - `s/string/replacemer`
 - `t/template/replacem`

Playtest Exercises

Figure 3-11: The view of Annotate tab on the sidebar.

Chapter 4

Implementation

This section discusses the implementations of Search, Simplify, and Annotate parts of PraxisGen. The front-end of all these parts is written using HTML, CSS, JavaScript, and TypeScript. To have the system integrated as a VSCode Extension, we use VSCode Extension API. The back-end of the system is written in TypeScript. Additionally, for the back-end of Search and Simplify sections, PraxisGen uses TypeScript Compiler API to convert the code into Abstract Syntax Tree (AST) and get the information needed for calculations in both sections [12].

4.1 PraxisGen as a VSCode Extension

PraxisGen uses VSCode Extension API, in particular the WebView API. The WebView API allows the system to embed custom views on either the sidebar or the editor side of VSCode [13]. In PraxisGen, the system uses the API to insert a web page created using HTML, CSS, and JavaScript on the sidebar. This page contains the input boxes and buttons that the user uses for exercise code creation. Also, this page interacts with both the back end and the editor side of VSCode. The way this sidebar HTML page communicates with the back end is through message passing; the WebView API provides `webview.postMessage()` which can send any JSON data. The back end of the system then accepts the message through event listeners. The same method is used for communication from the back end to the HTML sidebar.

4.2 Concept Local Search

The first step in exercise creation using PraxisGen is searching for the desired concept on the local repository full of TypeScript files. This part consists of the front end where the user puts the TypeScript code query and sees the result, and the back end converts the query and the TypeScript files into ASTs and matches them.

4.2.1 Sidebar Editor for User Input

In the sidebar of PraxisGen Search tab, there is an input box in which the user can put in the concept query they want to search. Because the search query is in TypeScript, instead of using a plain input box, the one used in the system is a code editor. This way, the user can utilize the syntax highlighting and auto-complete the editor provides. The code editor in the sidebar is implemented using Monaco Editor API [11]. This editor is the same code editor that is used by VSCode.

To accommodate PraxisGen's needs, there are several modifications on the sidebar Monaco editor. First, the system customizes errors by removing compile errors caused by uninitialized variables and the grey color code decoration that usually appears on unused variables. This is done to prevent confusion because of misleading error messages.

Other than custom error messages, the editor also has new decorations for Ignore and Strict flags. The system keeps track of a list of locations and types of the flags currently on the editor. If there is any change in the flag's location on the editor, the system automatically detects and updates the list. The editor also keeps track of the location of the mouse cursor; if it is located at the area where any of the flags is present, the corresponding flag button can be seen as pressed.

PraxisGen also adds custom code decoration for the identifiers. For Identifiers whose names are not matched (i.e. identifiers that start with `'_'`), the system automatically greys them out. The way these part works is that every time there is a change in the editor, the extension sends the code and the flag decorations to the back end. Then, the system transforms the code into AST using TypeScript Compiler

API and traverses the AST using depth-first search. During the search, the system notes all nodes that have the type of identifier and whose name starts with `'_'`. The back end then sends the list of all these nodes to the front end, in which the front end attaches the gray decoration to appropriate parts of the code.

When the user clicks the Find button, the extension sends the code and the flag decoration information to the user query parser.

4.2.2 User Query Parser

After getting the code and the name and location of each of the flag decorations from Monaco Editor, the system parses them so that they can be matched to the local repository TypeScript files' ASTs. The user code query is converted into an AST, then converted again to a custom query tree, in which each node contains more information about the query, including whether the name and/or type of the identifier has to be matched, or whether the node is a part of the Strict flag.

To convert the query into the query tree, the system uses a combination of TypeScript Compiler API and DFS. First, PraxisGen converts the query into AST using TypeScript Compiler API. Then, the system constructs Strict trees: for all parts of the code input that contains the Strict flag, an AST is constructed. These Strict trees are used in the matching stage to check whether the file's AST contains subtrees that have exact same structure as these Strict trees. A separate AST is used to make the matching implementation easier: the start and end of the subtree that need to be matched is more clear.

After creating all the Strict trees, now the query's AST is converted into the custom query tree. This tree is created by doing DFS on the query's AST. During the traversal, for each node, the system first checks whether the node is in the range that is marked by an Ignore flag. If it contains the Ignore flag, then the node is not included in the query tree. Then, the system checks whether the node is an identifier or not and whether the name and/or type of the identifier need to be checked during the matching stage. If any of that information needs to be checked, the information is noted on the corresponding node on the final query tree.

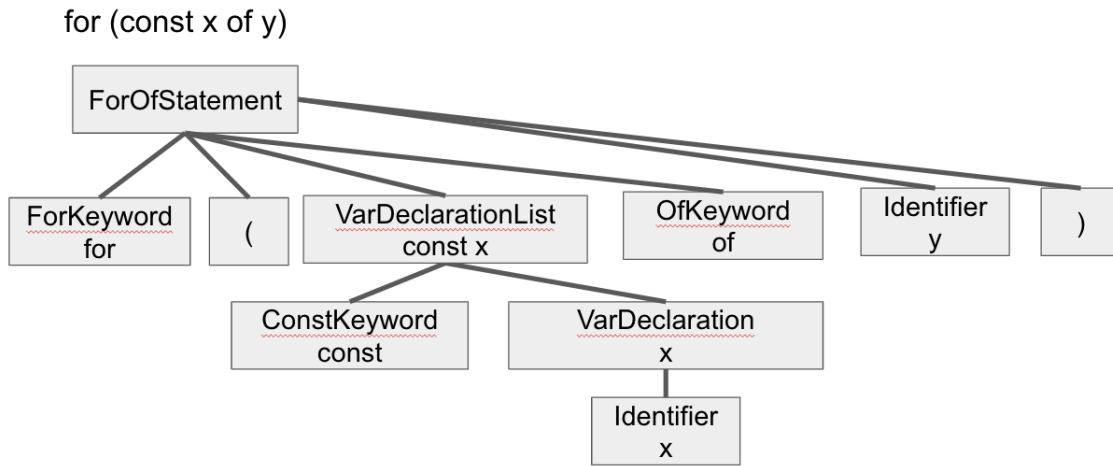
During the construction of the query tree, there are several special cases that need to be addressed in particular ways. For ignored nodes, if the name of the node is in the form of `X-Keyword`, in which `X` is a type of node, the parser looks at the ancestors of these nodes and removes all nodes with `X` in the name. For example, in the AST for `For` loop, the root has the name of `ForStatement` which encompasses the whole `For` loop block. In the descendant, the tree has a node called `ForKeyword` which encompasses `For`. So, if `For` has an ignore flag on it, the `ForKeyword` node will be thrown out. If the code does not have `For` in it, then it is incorrect to call the whole code `ForStatement`. That is why, the node `ForStatement` is also removed from the query tree, and the subtree of that node is connected to a dummy ancestor node instead. For an illustration of this example, see 4-1.

Other than the user-specified Ignore nodes, there are some nodes in the AST produced by the TypeScript Compiler API that the system automatically ignored. This is because the node is usually a duplicate of their subtrees, so including them in the final query tree just makes the tree more complicated and slows down the matching step. Some of these nodes are `SyntaxList`, `FirstStatement`, `ExpressionStatement`, and `VariableStatement`.

After the query tree is created, it is then used for the matching stage of the PraxisGen Search.

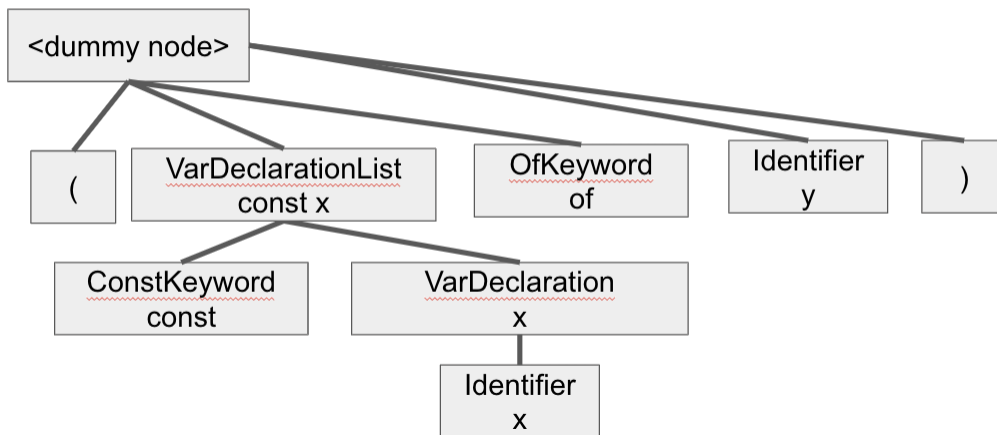
4.2.3 TypeScript Files and User Query Abstract Syntax Trees (AST) Matching

During the matching stage, the user query tree is matched to subtrees of all ASTs created from the TypeScript files in the local repository. By default, the information that is matched is the AST node's name. See figure 4-1a for an example of an AST with all the nodes' names.



(a)

~~for~~ (const x of y)



(b)

Figure 4-1: (a) An illustration of the AST of `for (const x of y)`. The code that is represented by the node is written on the second line. (b) An example of the resulting query tree if `for` has an Ignore flag.

4.2.3.1 The Type of Tree Pairs that is Considered a Match

There are several conditions for a subtree of a file AST to be considered a ‘match’ to the query tree. First, the file subtree must contain all the nodes with the same name as the query tree. Then, the vertical structure of the subtree must be similar to that of the query tree. This means that if x is descendant of y in the query tree, x should also be a descendant of y in the file subtree. Moreover, the nodes’ order of appearance between both trees should be the same; if x appears before y in the query tree, x should also appear before y in the file subtree. This ordering is important because the AST represents the code, and the code has a specific order for it to work correctly. For example, if node ‘{’ appears before ‘}’ in the query tree, the system also wants the same order of appearance in the subtree. Finally, the file subtree may have extra nodes compared to the query tree; this is fine as long as all the mentioned rules are followed. Figure 4-2 illustrates some pairs of trees that are matching and not matching.

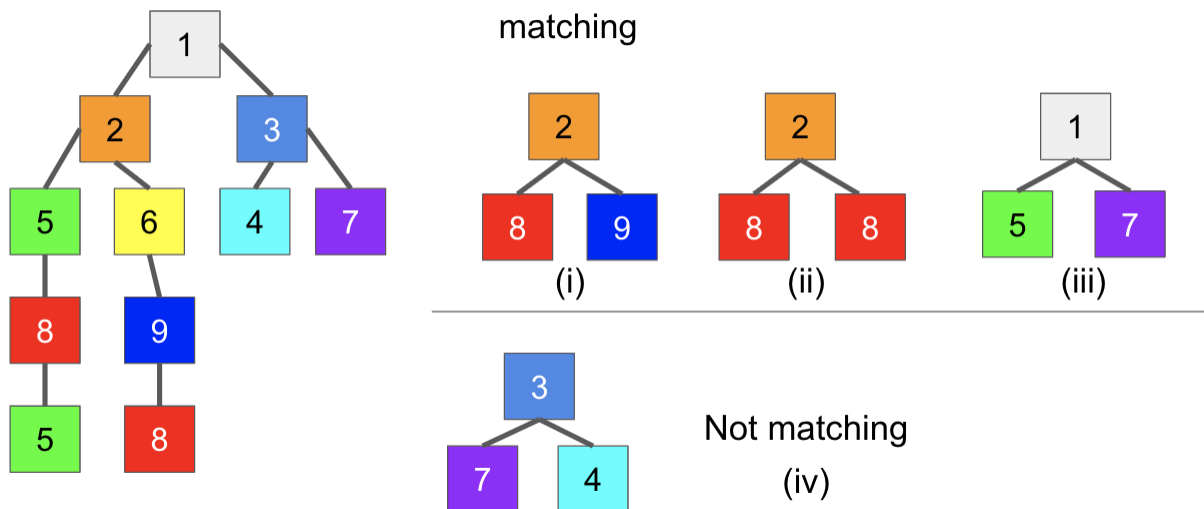


Figure 4-2: Examples of matching and non-matching pairs of trees. The nodes’ name is written as numbers for simplicity. The left tree represents the file AST, and each of the trees on the right represents different query trees. (i), (ii), and (iii) have matching subtrees on the file AST. For example, (i) matches the subtree with root ‘2’ and leaves of ‘8’ and ‘9’; even if the subtree contains ‘5’, ‘6’, the system still considers them as a match. However, (iv) does not have a match; the most similar subtree has the root of ‘3’ and leaves of ‘4’ and ‘7’, but the leaves have a different order.

4.2.3.2 Matching Process

To do the matching, the system does DFS on both the query AST and the file AST at the same time. First, from the path to the local repository that is provided by the user, the system gets all files that have the extension of `.ts`, `.d.ts`, or `.tsx`. Each file is then converted into an AST using the TypeScript Compiler API and matching between the user query tree and the AST is done. The output of this process is in the form of an array of Result Tree. Result Tree is a tree that is similar to the file subtree which matched the query tree with additional information in the nodes. Some information includes the corresponding code line number the subtree is located, the first line of the code in the subtree, height and order number of the node. The first line of code information is used for the result preview, and the height and order number of the node are used during the matching to follow the rule mentioned in section 4.2.3.1.

During each iteration of the DFS, there are several possible cases:

- Current file node is not chosen as a part of the Result Tree: in this case, the system traverses down the file AST without doing anything to the current file node.
- The node name of the current file node and current query node is the same: the system can use this file node as a part of the Result Tree.

If the name of both nodes is 'Identifier', PraxisGen also checks whether the identifier's name and/or type are the same. This depends on whether the user requested that the name and/or type need to be identical. The identifier's name can be acquired by getting the actual code of the node, while the type of the identifier can be obtained through the type checker provided by the Compiler API.

In the file AST, there is a possibility that certain types of nodes appear multiple times. The system first collects all possible combinations of Result Trees. Then, it removes all trees which do not fulfill all the restrictions relating to the vertical and

horizontal relationship between the nodes mentioned in section 4.2.3.1. The array of Result Trees is then cut down further by removing Trees that contain at least one node already contained by other Result Trees. This is done to reduce the number of duplicates in the result. After that, if the user uses the Strict flag anywhere in their query, the reduced list of Result Trees is cut down further by only keeping Trees that contain subtrees with the exact same structure as the Strict Trees obtained during the parsing in section 4.2.2.

4.2.4 Search Result Display and File Preview

After the matching process, the system has a list of Result Trees containing the necessary information for the result, including the file paths and the line numbers in which the concept requested by the user is located. PraxisGen then sends that list to the front-end to display the result and file preview to the user using `postMessage`. The system first groups the results that are contained within the same file. Then, the system figures out parts of the first line of the result code that needs to be highlighted. Each result also has `EventListener` attached. The listener is used to track when the user clicks on the result at the sidebar, a read-only preview of the file appears.

Whenever there is a file opened on the editor of VSCode, the extension checks whether the file is a result preview or not. If it is a result preview, the extension notes the opened file as read-only. VSCode does not provide any functionalities to prevent users from modifying a certain editor. So, a custom read-only feature is developed. Whenever the user tries to type anything that can change the opened editor, the extension shows an information box stating that the file is read-only and provides a button that the user can click to move on to the Simplify step of exercise code creation. This is done using `vscode.window.showInformationMessage`.

4.3 Code Simplification Guide

After the user picks on the file they want to create an exercise from, they move on to the next step of the exercise problem creation, which is simplifying the file to an

appropriate length. In this step, the extension creates a copy of the chosen file, then handles user action during the file simplification, and prepared the simplified file so that it is ready for the annotation step. This section explains how PraxisGen does each of those in detail.

4.3.1 Preparing Independent File Copy for Simplify

To prevent users from modifying the original TypeScript files in the local repository, PraxisGen provides a copy of the files that can be used for the user to simplify. However, it is not enough to only copy the one file chosen, because the file may be dependent on other files or NPM modules. An easy way to work around this is by copying the whole project folder that contains the chosen file; but, some files may be unnecessary and just waste the memory. So, the system gets all the possible file and module dependencies using the chosen file as the root file.

The way the system gets the dependencies is by turning the file into AST using the TypeScript Compiler API, traversing the tree using DFS, and finding the import nodes. Through the import nodes, the system keeps track of imports that come from other files, which is characterized by the string that begins with '.', '/', or '~', and imports that come from the NPM modules. Other than dependencies through imports, the system also looks for dependencies through triple-slash directive (///). This type of dependency has to be handled separately because, by default, the Compiler API's AST does not consider comments as separate nodes. So, the triple-slash directive has to be searched using the regex `///\\s*<\\s*reference\\s*path\\s*=`. After getting all the imports from the chosen file, the system also needs to get the imports from the dependencies too. This is because the system makes sure that the resulting copied file is free from compile error. So, PraxisGen recursively gets all the network of dependencies starting from the chosen file as the root.

After getting information on the list of files dependencies and NPM modules imports, PraxisGen creates copies of all the files in a different folder. Then, the system automatically runs `npm install` command on all the imported NPM modules. The system also adds a minimum `tsconfig.json` file so that the copied files can work

properly. The excerpt of the configuration can be seen in figure 4-3. Finally, the copied files are ready to be simplified, and the copy of the root file is opened in VSCode editor.

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "ES2020",
    "lib": ["ES2020", "dom"],
    "sourceMap": true,
  },
}
```

Figure 4-3: An excerpt of the generated `tsconfig.json` file.

4.3.2 User Action

After the copy of the chosen file is opened on the VSCode editor, the user can start simplifying it either manually through the editor, or using the help of PraxisGen Simplify. There are two actions that the system provides: Keep and Trash, and both implementations use AST in their calculations.

When the user selects a part of the code and clicks on any of the action buttons, the system edits the file and puts the start and end action tags on the first and last line of the selected code. These tags are added to help PraxisGen to keep track of the location of kept and trashed code parts. This is because the way the systems work is that if there is any change on the file, it removed all the code decorations and re-calculates all the Keep and Trash actions that are already done. All the actions are recomputed instead of saved in a data structure in order to maintain the simplicity of the system.

4.3.2.1 Keep Action

After the user clicks on the ‘Keep’ button, the system first gets the position of the user selection on the file. Then, it puts `/*[start keep]*/` tag at the end of the first line of the selection and `/*[end keep]*/` tag at the end of the last line. This is done using `vscode.TextEditor.edit`. The system then automatically saves the file. The file has to be saved on every changes because when the file is converted into AST using TypeScript Compiler API, the API grabs the whole file, so unsaved changes would not be included into the AST. Then, the system clears out all the editor decorations: all the green and orange highlights and grey code color from the previous change of the code, if any. Because PraxisGen does not store any information regarding the location of the tags, the next step for the system is getting all locations of the pairs of tags. The system also makes sure that the pairs of tags are valid, and this is done by using the same algorithm for solving the balanced brackets problem using stack [6].

To process the Keep action, the system creates an object of class `KeepCode`. The class contains two bucket sets: `Necessary` and `Keep`. In the `Keep` bucket, the system store the line numbers of code that are selected by the user. The `Necessary` bucket keeps track of the line numbers of code that the system deemed as parts of code that are needed by the code in the `Keep` bucket. These buckets are only used in one iteration of change and are recomputed whenever there is a new change.

After the `KeepCode` object is created, the first step is filling in all the buckets. First, the system iterates through all the locations of the `Keep` tags, and put all the line numbers of code which are covered by those tags into the `Keep` bucket. Then, the system fills the `Necessary` buckets from computation done on each pair of the `Keep` tags. In this process, PraxisGen first convert the code file into an AST using TypeScript Compiler API. Then, it acquires the highest nodes in the AST whose code are within the currently iterated `Keep` tags range. This is done by traversing the AST using Breadth-First Search (BFS) and stopping the search once the first nodes that are within the range are encountered.

For each of these top nodes, the system then looks for parts of code that are necessary for them. This is done by DFS starting from those nodes. For each traversed node, if the type is identifier, the system gets the declaration of the identifier using TypeScript's `TypeChecker` and checks whether the declaration is already within the range of the top nodes. If it is within the range, then the system ignores them so that there is no duplicate declarations in the Necessary bucket. The system also checks whether the identifier is user-defined identifier or TypeScript built-in ones by checking the file where the declaration is located. If it is located on user-created file, the systems then grabs the nodes above the declaration nodes that contains the whole declaration line. This is because sometimes the declaration nodes only consists of uncompileable nodes, so the system needs to look into the ancestor of these nodes for the full statement instead. For example, it is possible that an identifier node's declaration is in a function parameter. In this case, the system must look for the nodes that contains method header for the code to be free from compile errors. Also, when the system is traversing the ASTs and keeping track of the declaration nodes, it is also recursively traversing these nodes to get all the necessary variable or function declaration needed for the final code to run correctly. All the lines in these nodes are then included into the Necessary bucket.

In the final exercise code, the user may want to provide the context in which the requested concept is used. Therefore, whenever the user uses the Keep action on a part of code, the system searches the smallest block containing the kept code. For example, if the code that is kept by the user is inside a `for` loop, then the `for` loop block is also included in the Necessary bucket. The way PraxisGen does this is by traversing up the AST starting from the top nodes and searches for nodes that indicates a block. However, there are several special cases that the system needs to handle in a particular way:

- If any nodes in the subtree of the main block contains a `return` statement, the system must find the closest `function` or `method` block.
- If the subtree contains `break` statement, the system must look for closest `for`

loop, `while` loop, `do-while` loop, or `switch` statement.

- If the subtree contains `continue` statement, the system must find the closest `for`, `while`, and `do-while` loop.

These special cases are added to make sure that the final file is free from compile errors. PraxisGen also gets all the declarations of the identifiers contained in this main block by using the same algorithm as described before. All the lines in all these nodes are also included in the Necessary bucket.

Finally, PraxisGen also needs to handle a special case regarding nodes that are related to `class`. For example, if any of the code that is currently in the Keep or Necessary bucket contains a `class constructor`, the system must also include the `class` header in the final file. The system first iterates all the nodes acquired from all the steps above and checks whether any of those nodes are related to `class`. These nodes can be a `constructor`, `property` declaration, or `method` declaration. If any of these nodes are related, the system traverses up the AST to find the closest `class` declaration and include the lines in the Necessary bucket.

Once the system finishes filling the Keep and Necessary bucket, PraxisGen continues processing the user's Trash action.

4.3.2.2 Trash Action

When the user clicks on the Trash button, similar to the Keep action, PraxisGen also adds the start and end tags at the beginning and end of the code block. The system also automatically saves the file and clears all the code decorations.

The next step for PraxisGen after filling the Keep and Necessary bucket in Keep-Code class is handling the user's Trash action through TrashCode class object. In TrashCode class, the system keeps track of the Trash bucket which contains the line numbers in which the user selects for the Trash action. Additionally, the class keeps track of the compile error bucket, which contains the line numbers of user conflicting Keep and Trash actions. To fill the Trash bucket, PraxisGen iterates through all the line numbers in the range of the start and end Trash tags. It also checks that none

of the lines is in the Keep buckets; if there is any, that line is included in the error bucket. This is because the system disallows the user to Trash the lines in which they already indicate as Keep. If any of the line in the Trash bucket is in the Necessary bucket, that line is removed from the Necessary bucket.

PraxisGen then needs to look for potential action conflicts in other places and removes some code from the Necessary bucket. This is because if there is any code declarations that are trashed, all occurrences of the variable must also be removed from the Necessary bucket.

The first step is similar to the Keep action, which is getting all the top nodes in the AST that is within the Trash range for all pairs of the Trash action tags. Then, for all these top nodes, the system looks for all declarations in the subtrees. This is done by doing DFS on starting on each top node. Whenever the system finds an identifier, it checks whether the declaration is included inside the subtree of the top node. The system keeps track of these declarations. PraxisGen then traverses the AST of the entire code file and finds all occurrences of the variables with the same declaration. This is done by comparing the the TypeScript Symbol for both the declarations and each of the nodes in the AST. Symbol for both the declarations and each of the nodes in the AST. For each of these occurrences, the system then iterates all the line numbers of the code. If any of them is in the Necessary bucket, they are removed from the bucket. However, if any of them is contained in the Keep bucket, the system keeps the line number in the error bucket.

4.3.2.3 Code Decoration Update

The final step in computing the user action is updating the code decoration, which is the green and orange highlights that indicate the Keep and Trash action, and the gray color code that indicates code that is removed in the final file. There are three possible cases for updating the highlight:

- All of Keep, Necessary, and Trash bucket is empty: the system does not do anything to the file.

- Only the Trash bucket is not empty: the system only adds the orange highlighting. This is because it does not make sense to grey out all the code when there is no code to be kept.
- Otherwise, the system adds all the possible decorations based on the line numbers stored in each buckets.

PraxisGen adds the decoration with the help of VSCode's API `vscode.window.createTextEditorDecorationType`, in which the CSS of the decoration is the input to the method. So, the highlight comes from `backgroundColor` parameter, and the gray color of the code comes from `color` parameter. To put the decoration to the text editor, the system first gets all the lists of the line numbers. Then, the decoration is set using `vscode.TextEditor.setDecorations`.

Finally, the system applies the compile errors to the editor. This is done having a variable that keeps track of the custom VSCode diagnostics. This variable can be initialized using `vscode.languages.createDiagnosticCollection()`. Then, for all the line numbers that is kept in the error bucket, it is set as compile errors in the diagnostic collections.

4.3.3 Split Preview Window

When the user opens the file preview panel on PraxisGen Simplify, a new panel containing a preview of what the final simplified file looks like pops up beside the original file. First, the system creates a temporary file `simplify-preview.ts` in the same folder as the original file. Then, it gets all the lines in the code of the root file that is currently simplified except for lines that are indicated by grey color or orange highlight. After that, the system also removes the start and end tags of the Keep action for the preview. The result of this process is then written into `simplify-preview.ts`, and the file is then opened in a different panel using `vscode.window.showTextDocument`. Whenever there is a change to the original file being simplified, after the Keep and Trash actions are calculated, the file `simplify-preview.ts` is also re-written from scratch using the steps described above.

4.3.4 Finish Simplification Process

Once the user finishes simplifying the file, they can press the Finish button. The system creates a new folder that is filled with the simplified file and other files that are needed by it. After creating the folder, the system looks into the original root file and gets all the lines in Necessary and Keep buckets. Other than that, the system also grabs all the lines that contain triple-slash directives. This is because there is no simple way to know whether there is a variable or methods in the code that is imported from other files through triple-slash directives, so they are always included to reduce the possibility of getting a compile error in the final file. The system then removes all the start and end action tags within the lines gained in the previous process.

Since the output file is shorter than the original file, there may be some imports that were needed in the original file which are now not needed anymore. So, PraxisGen recomputes the other files and NPM modules that are needed to be included in the new folder. The way the system acquires all the files needed for the final root file and installs all the necessary NPM modules is the same as the method described in section 4.3.1. The only difference is that for the root file, PraxisGen also re-formats the file using Prettier API [18]. This is done because when the system gets the lines in Keep and Necessary buckets, it grabs the whole line, so some of the spacings may be in disarray.

After creating the new simplified file and installing all the necessary NPM modules, PraxisGen then opens the file in the editor. The system now moves on to the Annotate step of exercise problem creation.

4.4 Exercise Annotation

In the Annotate step of PraxisGen, the user can add problem statements, answers, and other information related to the coding exercise. When the user clicks on the “add YAML header to file”, the system first checks if there is a file currently opens on the editor. If not, the system throws an error. If the file that is created in the

previous section is currently open on VSCode editor, the system edits the file and adds the YAML header using VSCode's `TextEditor.edit`. Once the user finishes adding all the exercise problem information into the YAML header, they can click on the "Playtest Exercises" button. PraxisGen then sends all the finished file to the Tutor platform's server so that the user can playtest them directly on TypeScript Tutor.

Chapter 5

Evaluation

This section discusses the assessment of PraxisGen, which consists of two parts: technical evaluation and user study. The technical evaluation assesses the Search section of coding exercise creation, and the user study tested the usability of the system in writing problems.

5.1 Technical Evaluation

The technical evaluation of PraxisGen tests the system in its ability in parsing the user's queries and producing the correct result. The system must output the locations in that the requested concept appears in the local repository. It is also evaluated on the number of TypeScript concepts that the user is able to create queries for.

5.1.1 Methodology

PraxisGen is evaluated based on the TypeScript concepts that the MIT students of course 6.031 Software Construction needed to learn for the class. In total, there are 110 different concepts that the course staff needs to create exercises for, and the system is tested on all of them.

The way the testing works is by going through all 110 TypeScript concepts in the list and use the Search function to get the location of the concept in the local

repository. For this testing purposes, the local repository used is cloned from public TypeScriptSamples repository [10]. This repository is created by Microsoft and is used as the official repository for TypeScript sample code. There are 92 TypeScript files in the repository, and it is forked and starred by thousands of Github users, so both the quantity and quality is good enough to be used for the evaluation.

While manually going through all the concept in the list, the system is tested on whether it is possible to create a query for the concept. Other than that, from all the result that is produced by PraxisGen, the first ten result is analyzed on how many of the location in the output actually contains the concept. Incidents during the testing are investigated. For example, the reason for why certain concepts do not have queries are noted. Also, for concepts that does not produce any result, we search the existence of the concept on the local repository using VSCode Search feature to make sure that the repository actually does not contain them.

5.1.2 Result

	Number of concepts
Total number of concepts	110
Number of queries created	94
First 10 results contain the concept	73
Result not found in repository	21

Table 5.1: Overall result of the technical evaluation

The technical evaluation demonstrates PraxisGen’s capability in searching TypeScript concepts in the local repository. From 110 concepts tested, the user can create queries for 94 of them. Some of these concepts include concepts that are commonly used in code, such as binary comparison, conditionals, loops, and classes. Other rarer concepts such as ternary expression, empty statements, and anonymous classes are also supported by PraxisGen search query language. This proves that the system is able to accept a wide variety of TypeScript concepts, and the query language

designed for the system is effective in representing those concepts. In all of the 94 queries written, all of them can be written in five lines or less, and this shows the simplicity of the query language used in the system.

Another thing that the evaluation demonstrates is the ability of the system to parse the query correctly and produce the intended result. Among all 94 queries searched using PraxisGen, the system produces the correct result for 100% of the concepts that are available in the local repository used for the testing. The repository does not contain 21 concepts that are searched, so the system produces no result found for those queries. All of these 21 concepts are searched manually in the repository using VSCode search feature, and it is confirmed that none of them are contained in the local repository. Some of these concepts include concepts that are rarely used in regular TypeScript coding, such as big integers, lists using the syntax of `Array<T>`, and list destructuring. A few of them are commonly used data structures, such as set and map; however, none of the code in the repository uses those specific data structures.

Reason	Number of concept
Concept cannot be syntactically represented	5
Related information is not provided by the API	7
Concept is within string literal	4
Total number of concept that doesn't have a query	16

Table 5.2: The breakdown of the reason on the search query creation failure

Among 110 TypeScript concepts that are contained in the TypeScript Tutor, PraxisGen's query language cannot represent 16 of them. The reason for the inability to create the queries can be categorized into three groups. Because PraxisGen searches the query based on the structure of the code and the node name of the AST, it cannot be used in representing concepts that are not obvious syntactically. For example, the concept of index out of bound in list does not have specific syntax. Another example is the global constant concept. While a person can distinguish whether a constant is local or global by looking at the location in which the constant is created, the system

does not parse the search query based on the location, hence it is impossible to create the query for this concept.

For other concepts, some of their query cannot be created because the information is difficult to obtain in the TypeScript Compiler API. Most of them are related to comments, such as documentation tags `@param` or `@returns`, and single-line comment using `//`. In the AST produced by the API, comments do not have separate nodes and are combined with other nodes nearest to the where the comments are related. So, concepts related to comments have to be processed separately, and they cannot be easily represented by the query unless it is explicitly requested.

The last group is concepts that are within the string literal. This includes the concept of escaping newlines, tabs, backslash, and quotes. In the AST produced by the Compiler API, all strings have the name of 'string literal'. PraxisGen only matches the node with the name 'string literal' and not the value of the string itself, so queries related to these concepts cannot be created.

5.2 User Study

The user study portion on PraxisGen evaluation is done in the PraxisGen Search and Simplify section. In the study, the usability and effectiveness of the system in aiding the user in coding problem creation are assessed.

5.2.1 Methodology

For the study, four staff members from course 6.031 Software Construction are invited. The staff members from this specific course are recruited because the exercise created by the system is intended to be used for students in 6.031, so the teaching assistants (TA) or the lab assistants (LA) of this class are the primary intended users of the system. Moreover, because the course is taught in TypeScript, all the staff is proficient in the language too, so they can use the system more seamlessly.

The study requires the user to create several TypeScript exercises using the system. There are three different kinds of exercise problems that the user must create within

one hour. The three concepts chosen have a different range of difficulty in the search query creation. The local repository is already prepared, and it is a smaller part of the TypeScriptSamples repository used in section 5.1. A subset of the repository is used instead of all of them to save the processing time during the study.

In the first section of the study, the user can get used to the system by creating an exercise containing `if` concept. This concept is chosen as the first one because the search query for `if` concept is expected to be simple enough: the user can either directly write `'if'`, `'if() {}'`, or other short queries similar to these two. A high-level step-by-step instruction is given to the user and no further instruction is given unless the user is stuck and asks some questions. The user is also free to use the Help page to guide them, but no specific direction is given to them. The exercise problem creation is stopped once the Simplify process finishes because, in the Annotate section, the system uses a YAML format that is already established in TypeScript Tutor, so it is not included in the evaluation.

1. Search for `if` statement concept using PraxisGen sidebar (get the filepath in Stickies app)
2. Pick a code file you want to use for the exercise
3. Reduce down the length of the file using PraxisGen Simplify
4. Stop after clicking the 'Finish' button in Simplify tab

Figure 5-1: An excerpt of the first section of the user study

For the second section of the study, the user creates another exercise that contain number equality concept. This concept involves comparing the equality of two objects of type `number`. This concept tests whether the system's flag and identifier name or type matching features are intuitive to use or too confusing for the user. One of the expected search queries for this exercise is declaring two variables as `number` and comparing both of them while adding the Ignore flags on the declarations. The user can use the step-by-step instruction given during the first exercise, and no additional

instruction is given.

In the last part of the study, the user has to create multiple exercises using two different methods: manual exercise creation and PraxisGen. The concept chosen for this section is ternary expression. This concept requires the user to have basic understanding that the box in the sidebar that they use to write the query with is a code editor, and they have to input a short TypeScript code that exemplifies the ternary expression. Therefore, before starting this section, if the user has not opened the Help page at all, they are suggested to read it.

During the manual exercise creation, the user creates up to five exercises but spending no more than twelve minutes, whichever comes first. The user is allowed to use any methods possible other than using PraxisGen. They are also provided the local repository used throughout the study as a reference. If they want to use the files, they have to search the concept and simplify the file manually. It can be done by using VSCode search function and deleting parts of the code file that the user deemed unnecessary. The user can also search for the example of the concept in Google and copy-paste it. The only requirement provided for the exercises is that they have to have the length of around 15 – 30 lines of code, which is the average length of the code files in TypeScript Tutor. The manually created file also needs to use proper coding formatting and appropriate variable names.

After the manual problem creation, the user creates another up to five exercises within twelve minutes using PraxisGen. All the requirements about the length and the quality of the files created using the system is the same as the manual one.

5.2.2 Result

In the user study, all four participants successfully created TypeScript exercises using PraxisGen. They managed to create the search query for `if`, number equality, and ternary expression concept, then simplified the chosen files using PraxisGen Simplify Keep and Trash actions. While there are a few incidents in which the system can be improved, the overall sentiment is that the system can help users in creating TypeScript concepts more efficiently. Detailed explanation on the result of the Search,

Simplify, and the overall system is explained further below.

5.2.2.1 Using PraxisGen Search

When creating the first and second exercise, the users were not explicitly directed to read the Help page before writing the query and using the Search feature. This produces several interesting results; while the two users who skimmed through the Help page were able to write a more sophisticated search query, the other two users who did not read the Help page were still able to create search queries, and the system was able to process some of them. Table 5.3 contains several example queries written by the users who read and does not read the Help page.

Concept Searched	Queries written	
	User read Help page	User didn't read Help page
if	"if" or "if(){}"	"if statement"
number equality	let <code>_n1: number;</code> let <code>_n2: number;</code> <code>_n1 == _n2</code>	"==" , then look into the result one-by-one for <code>number == number</code>

Table 5.3: The differences on the queries written by users who read and did not read the Help page. The red cell indicate the query that PraxisGen cannot parse.

As seen from the table, the two users who read the Help page before starting to write the query understand that the query is supposed to be in the form of TypeScript code. During the search for number equality, they initially did not use any of the flags. Once the system did not produce any result, the user quickly scanned through the Help page once again and was finally able to add the Ignore flag to improve their query. For the users who did not read the Help page, they mentioned that they thought the Search feature works similarly as Google search and that's why they put "If statement" as the query. Another misconception is they thought the Search feature works like a simple string matching, and that is why they only put double equals as the query for number equality.

This part of the study demonstrates that the query language used for the Search

why the Keep and Trash buttons were still disabled when they wanted to start the Simplify process.

One of the strengths of Simplify is the Keep action. The participants said that they like how the system automatically provides them with the information on which parts of the code is relevant to the one they selected. One of them also mentioned that the start and end tags are helpful because it gives them multiple options on doing the actions, either by using the buttons or manually writing the tags.

However, there was also a slight misunderstanding in how the Keep and Trash button works because there is a lack of instruction or explanation in the system. All of the participants selected the whole block of code they want to keep instead of just the first line. While both of these methods work, the system is designed to be able to automatically keep the whole block even if the user only selects the first line because sometimes the code can be long, so selecting the whole block is not efficient. So, this specific feature is not utilized properly by the users. One user also missed some parts of the instruction on how to use Simplify that is available on the sidebar. They manually select all the grayed out code and click the Trash button even if the grayed out would automatically be thrown out by the system. Another user also admitted that they were hesitant in using the Trash action on the colored code because they thought that all code that still has its color is necessary for the final code to work properly.

Another feature that the participants liked is the side-by-side file preview. Three of them explicitly mention that it is helpful to have the panel open while editing the file because it is hard to estimate the length of the file without the feature. They also liked that it is automatically updated, so they did not have to repeatedly open and close the panel to preview the file.

The participants also appreciated the fact that they can also manually edit the file while using the Keep and Trash buttons. Unfortunately, only two out of four participants realized that they could type and delete the code on the file. However, these two participants then used the combination of this manual edit and the system's automated action in all of their exercise creation.

5.2.2.3 Using the System in Creating Exercises

In general, the participants think that PraxisGen helps them in creating exercise problems more efficiently compared to manual creation. Based on the data in table 5.5, the quantity of exercises created using both methods are similar. However, the advantage of using PraxisGen can be seen from the difference of the quality of the exercises they created using both methods. From figure 5-2, it can be seen that the manually created exercises are mostly very short and lack the context in which the concept is used. Other than that, some of them use very short and obscure variable names, so they are not a good representative on how the concept is used in real life coding. Compared to the manually created exercises, the ones created using PraxisGen in figure 5-3 are longer and resemble real life programs better.

Participant	Number of exercises created	
	Manually	Using PraxisGen
A	2	5
B	5	5
C	5	4
D	4	5

Table 5.5: The number of exercises created using both methods by all four participants.

<pre> 1 function check(a: number, b: number) { 2 if (a === b) { 3 console.log('a = b'); 4 } else if (a > b) { 5 console.log('a > b'); 6 } else { 7 console.log('a < b'); 8 } 9 } 10 11 // Refactor into ternary 12 function checkTernary(a:number, b:number) { 13 const c = (a === b ? 'a = b' : (a > b) ? 'a > b' : 'a < b'); 14 console.log(c); 15 } </pre>	<pre> 1 function f1() {} 2 function f2() {} 3 function f3() {} 4 function f4() {} 5 6 function multilineTernary(bar: any) { 7 bar === 'a' ? f1() : // if 8 bar === 'b' ? f2() : // else if 9 bar === 'c' ? f3() : // else if 10 f4(); // else 11 } </pre>
<pre> 1 const nums = [0, 1, 2, 3, 4, 5] 2 for (const n of nums) { 3 let cond: boolean = "hello".length > "goodbye".length; 4 let z = cond ? n : -1; 5 } </pre>	
<pre> 1 const nflTeams = ["bucs", "patriots", "seahawks"] 2 const teamName = "nets" 3 console.log("do the ", teamName, " play in the NFL?") 4 console.log((nflTeams.includes(teamName)) ? "yes" : "no") </pre>	
<pre> 1 function renderApp() { 2 // omitted 3 } 4 function renderLogin() { 5 // omitted 6 } 7 8 const authenticated = true; 9 const pageToRender = authenticated ? renderApp() : renderLogin(); </pre>	

Figure 5-2: Some examples of the exercises that the users created manually. The highlighted code is the main concept, which is ternary expression.

```

16 private getReflectionColor(thing: Thing, pos: Vector, normal: Vector, rd: Vector, scene: Scene) {
17     return Color.scale(thing.surface.reflect(pos), this.traceRay({ start: pos, dir: rd }, scene)
18 }
19
20 private getNaturalColor(thing: Thing, pos: Vector, norm: Vector, rd: Vector, scene: Scene) {
21     var addLight = (col, light) => {
22         var ldis = Vector.minus(light.pos, pos);
23         var livec = Vector.norm(ldis);
24         var neatIsect = this.testRay({ start: pos, dir: livec }, scene);
25         var isInShadow = (neatIsect === undefined) ? false : (neatIsect <= Vector.mag(ldis));
26         if (isInShadow) {
27             return col;
28         } else {
29             var illum = Vector.dot(livec, norm);
30             var lcolor = (illum > 0) ? Color.scale(illum, light.color)
31                 : Color.defaultColor;

```

```

3 class Ship {
4     column = 0;
5     row = 0;
6     isVertical = true;
7     element: HTMLElement;
8     constructor(public size: number) {
9         this.element = $("<div class='ship'></div>")[0];
10    }
11    updateLayout() {
12        var width = "9.9%";
13        var height = "" + (this.size * 9.9) + "%";
14        this.element.style.left = "" + (this.column * 10) + "%";
15        this.element.style.top = "" + (this.row * 10) + "%";
16        this.element.style.width = this.isVertical ? width : height;
17        this.element.style.height = this.isVertical ? height : width;
18    }
19 }
20

```

```

2 import * as url from "url";
3 export class GitHubClient {
4     private token: string;
5     private prepareRequest(method: string, requestUrl: string, query: any) {
6         let parsedUrl = url.parse(url.resolve("https://api.github.com/", requestUrl), true);
7         let hostname = "api.github.com";
8         let headers: any = {
9             "User-Agent": "github-api (NodeJS v4.0.0)",
10            "Accept": "application/vnd.github.v3+json"
11        };
12        if (this.token) {
13            headers["Authorization"] = `token ${this.token}`;
14        }
15
16        let pathname = parsedUrl.pathname;
17        let search = querystring.stringify(Object.assign({}, parsedUrl.query, query));
18        let path = search ? pathname + "?" + search : pathname;
19        return { method, hostname, path, headers };

```

Figure 5-3: Some excerpts of the exercises that the users created using PraxisGen. The highlighted code is the main concept, which is ternary expression.

Chapter 6

Discussion

The evaluation of the system demonstrates that PraxisGen improves the user's experience in creating exercise problems. There are several difficulties encountered by the user when the exercises are created manually from scratch. While creating one exercise for a certain concept is easy, creating multiple exercises of the same concept is hard to do because the user has to think of many versions of how the concept is written syntactically and how it is used. When the user looks for examples of certain concepts on Google, they are usually short or lacking in context because the main purpose of the example is just for showing the syntax of the concept. Other than that, if the user chose to reference code from a local repository, it is hard to use VSCode search function because some concepts can have complicated regexes. Moreover, simplifying by manually deleting a very long file is tedious.

Creating coding exercises using PraxisGen helps the user with all the problems mentioned above. By using public repositories as the resources, the user does not have to think about the variations of how the concepts are used because the repositories already provide diverse examples and the user just needs to pick the ones they want. Because the repositories are gathered from public, they contain actual programs, so the context surrounding the concept represents how the concept is used in real life. Also, since the search query is in the form of simple TypeScript code, any user who is familiar with TypeScript can easily search for the concept they want without having to write intricate regexes.

6.1 Limitation

While in general, the system can aid the user in creating exercise problems, there are several limitations to the current iteration of PraxisGen. One of them is that the system cannot search for concepts that cannot be represented syntactically. Because PraxisGen depends heavily on the AST of the query and the repository files, if the concept does not have any specific syntax, exercise problems related to the concept cannot be created using the system. A few examples of this type of concept include array index out of bound and global constants.

Another part of PraxisGen that needs to be developed further is the Simplify section. From the user study, it can be seen that there is some confusion about the actions because they are not intuitive enough, especially the Trash action. The Simplify part also lacks instructions and a Help page, so the user does not have any reference on how the system is supposed to be used.

Finally, the time performance in all parts of PraxisGen needs to be improved. Currently, searching for a more complex query on the local repository used in the technical evaluation section can take up to 30 seconds. This is not ideal because the user may want to iterate on their query to get better results, but currently there is no way to stop the search mid-way, so they need to spend a notable amount of time just waiting for the system to process the query. Also, in the Simplify section, when Keep and Trash actions are done on a piece of code that coincidentally has a lot of dependencies, the processing time can take up to five seconds. Because the user needs to edit the file using multiple actions, the lag can be frustrating.

6.2 Future Work

Further work can be done on PraxisGen to solve the limitations described above. The query language needs new features or flags so that the 16 concepts that do not have a query in the technical evaluation section can be represented by the query. The new features can expand the variations of the exercises that can be created by the system.

Improvements to the Simplify section can be in the form of adding a tutorial for the step or removing some of the actions. Currently, there are only short snippets of explanation on what each action does on the sidebar, and from the user study, it can be seen that the explanation is not enough. Thus, a more detailed tutorial with examples is needed so that the user can see how the system can be used before using it themselves.

Another improvement to Simplify can be made by removing the Trash action entirely. This is because most of the confusion on the Simplify step is on how the Trash action works. Instead of the Trash function, the user can just delete the parts of the code they think is unnecessary manually. So, removing the Trash action can make the Simplify step easier for the user to use.

The algorithm used in the Search and Simplify function can also be sped up. Currently, most of the algorithm uses basic DFS for all the AST traversals. During the AST matching step in Search, brute force algorithm is used in matching all the nodes in the user query's AST and the files' ASTs. Other than that, in the Simplify step, every time there are any changes on the file, the Keep and Trash actions are recomputed. The time performance of the system can definitely be improved by changing the algorithms used on these parts. Another possible way to improve the performance is by caching the AST of repository files, so that the system does not have to convert the files into ASTs everytime the user searches for a new concept.

Chapter 7

Conclusion

This thesis presents PraxisGen, a multi-part system to create coding exercises for language concepts using existing code. The system provides a way for the user to search a wide variety of TypeScript concepts on a local repository using simple TypeScript code as the query language. PraxisGen also provides a guide so that the user can simplify the chosen code file by a few button clicks. Finally, the system allows the user to add problem statements, explanations, answers, and hints to annotate the code so that it can be used as coding exercise.

The result of the evaluation demonstrates that the system effectively helps the user in the problem creation process. The system supports problem creation of 94 different TypeScript concepts. Other than that, from the user study, it is shown that the exercises created using the system have better quality and variation compared to the problems written manually. The system also makes the exercise writing process simpler because the user only needs to pick from the available options instead of thinking from scratch. By using highly rated public repositories, the quality of the resulting code is also better than manually written code, and it truly represents how the concepts are used in real life programs.

Appendix A

Tables

print	constructor::call
print::for-debugging	function
while	return
comparison	boolean
arithmetic	boolean::literal
if	object::to-string
if::else	string::concatenation::with-object
if::else-if	string::literal::escape::newline
method::call	string::literal::escape::tab
string::literal	string::literal::escape::backslash
block-structure	string::literal::escape::quotes
comment::single-line	variable::local::scope
number	constant::local
number::int::division	constant::global
number::equality	constant::instance
number::not-equals	constant::static
number::increment-decrement	naming-conventions
number::bigint	continue
number::bigint::literal	for::three-part
string	switch
string::index	break
string::substring	ternary
string::length	empty-statement
string::concatenation::with-number	set
static-typing	map
variable::local::declaration	import
variable::local::initializer	undefined-value
object::equality::reference	union-type
string::equality	union-type::typeguard
boolean::and-or-not	nullish-coalescing
for::range	assert
for::iterable	comment::block
list	comment::documentation
list::equality	comment::documentation::param
list::type-shortcut	comment::documentation::return
list::length	comment::documentation::throws
list::index	exception::throw
list::index::out-of-bounds	exception::catch
list::destructuring	exception::catch::use-exception
record	exception::finally
record::literal	visibility::private
record::destructuring	visibility::private::runtime

visibility::public
constructor::declaration
field::reference
method::static::declaration
method::static::call::using-class-name
method::instance::declaration
variable::instance::declaration
variable::instance
variable::static::declaration
class::self-variable
variable::instance::initializer
variable::instance-vs-static
class
interface
interface::implement
class::anonymous
enumeration
enumeration::string
literal-type
exception::class
instanceof
object::to-string::implement
object
object::typecast
method::instance::override
constructor::default

Table A.1: A list of 110 TypeScript concepts needed for 6.031

number::bigint
string::substring
list (using Array<T> format)
list::destructuring
record::literal
record::destructuring
constant::instance
constant::static
continue
set
map
union-type
union-type::typeguard
nullish-coalescing
assert
exception::catch::use-exception
exception::finally
visibility::private::runtime
enumeration
literal-type
exception::class

Table A.2: A list of TypeScript concepts that are not available in the repository used for the technical evaluation.

string::literal::escape::newline
string::literal::escape::tab
string::literal::escape::backslash
string::literal::escape::quotes
comment::block
comment::documentation
comment::documentation::param
comment::documentation::return
comment::documentation::throws
comment::single-line
naming-conventions
constant::local
constant::global
list::index::out-of-bounds
object
object::equality::reference

Table A.3: A list of TypeScript concepts that the PraxisGen query language cannot represent.

Bibliography

- [1] Roy Ballantyne, Karen Hughes, and Aliisa Mylonas. Developing procedures for implementing peer assessment in large classes using an action research process. *Assessment amp; Evaluation in Higher Education*, 27(5):427–441, 2002. doi:10.1080/0260293022000009302.
- [2] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. Example-centric programming: integrating web search into the development environment. *Proceedings of the 28th international conference on Human factors in computing systems - CHI '10*, 2010. doi:10.1145/1753326.1753402.
- [3] Raymond P. Buse and Westley Weimer. Synthesizing api usage examples. *2012 34th International Conference on Software Engineering (ICSE)*, 2012. doi:10.1109/icse.2012.6227140.
- [4] Paul Denny, John Hamer, Andrew Luxton-Reilly, and Helen Purchase. Peer-wise: Students sharing their multiple choice questions. *Proceeding of the fourth international workshop on Computing education research - ICER '08*, Sep 2008. doi:10.1145/1404520.1404526.
- [5] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. Codewrite: supporting student-driven practice of java. *Proceedings of the 42nd ACM technical symposium on Computer science education - SIGCSE '11*, Mar 2011. doi:10.1145/1953163.1953299.
- [6] GeeksForGeeks. Check for balanced brackets in an expression (well-formedness) using stack, Jun 2021. URL: <https://www.geeksforgeeks.org/check-for-balanced-parentheses-in-an-expression/>.
- [7] Sacha Greif, Raphael Benitte, and Michael Rambeau. The state of javascript 2018: Other tools. *The State of JavaScript 2018*, 2018. URL: <https://2018.stateofjs.com/other-tools/>.
- [8] Andrew Head, Elena L. Glassman, Björn Hartmann, and Marti A. Hearst. Interactive extraction of examples from existing code. *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018. doi:10.1145/3173574.3173659.

- [9] Andrew Luxton-Reilly, Beryl Plimmer, and Robert Sheehan. Studysieve - a tool that supports constructive evaluation for free-response questions. *Proceedings of the 11th International Conference of the NZ Chapter of the ACM Special Interest Group on Human-Computer Interaction on ZZZ - CHINZ '10*, Jul 2010. doi:10.1145/1832838.1832849.
- [10] Microsoft. Microsoft/typescriptsamples: Community driven samples for typescript. URL: <https://github.com/microsoft/TypeScriptSamples>.
- [11] Microsoft. Monaco editor. URL: <https://microsoft.github.io/monaco-editor/>.
- [12] Microsoft. Using the compiler api · microsoft/typescript wiki. URL: <https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API>.
- [13] Microsoft. Webview api, Nov 2021. URL: <https://code.visualstudio.com/api/extension-guides/webview>.
- [14] Joao Eduardo Montandon, Hudson Borges, Daniel Felix, and Marco Tulio Valente. Documenting apis with examples: Lessons learned with the apiminer platform. *2013 20th Working Conference on Reverse Engineering (WCRE)*, 2013. doi:10.1109/wcre.2013.6671315.
- [15] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. How can i use this method? *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015. doi:10.1109/icse.2015.98.
- [16] Aaditya Naik, Jonathan Mendelson, Nathaniel Sands, Yuepeng Wang, Mayur Naik, and Mukund Raghothaman. Sporq: An interactive environment for exploring code using query-by-example. *The 34th Annual ACM Symposium on User Interface Software and Technology*, 2021. doi:10.1145/3472749.3474737.
- [17] Casey O'Brien, Max Goldman, and Robert C. Miller. Java tutor: bootstrapping with python to learn java. *Proceedings of the first ACM conference on Learning @ scale conference*, 2014. doi:10.1145/2556325.2567873.
- [18] Prettier. Api · prettier. URL: <https://prettier.io/docs/en/api.html>.
- [19] Naiyana Sahavechaphan and Kajal Claypool. Xsnippet: mining for sample code. *ACM SIGPLAN Notices*, 41(10):413–430, 2006. doi:10.1145/1167515.1167508.
- [20] ThemeSelection. Which is the best ide for javascript development in 2022? *DEV Community*, Jan 2022. URL: https://dev.to/theme_selection/what-is-the-best-ide-for-javascript-development-in-2021-1pmn.

- [21] Hongyu Zhang, Anuj Jain, Gaurav Khandelwal, Chandrashekar Kaushik, Scott Ge, and Wenxiang Hu. Bing developer assistant: Improving developer productivity by recommending sample code. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016. doi:10.1145/2950290.2983955.
- [22] Zixiao Zhu, Yanzhen Zou, Bing Xie, Yong Jin, Zeqi Lin, and Lu Zhang. Mining api usage examples from test code. *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014. doi:10.1109/icsme.2014.52.