Tool to explore finite monoidal categories

Pradnesh Sanderan

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2023

Abstract

In this report, the design, implementation and evaluation of a tool that validates the multiplication tables of strict finite monoidal categories are presented. The tool implemented has a graphical user interface where it is able to accept multiplication tables inputted by the user, extract all necessary information from the multiplication table, rebuild each component in the program, validate if it is a valid category and then validate if it is a valid monoidal category and then display the results. The tool was built from scratch due to the lack of previous work done in the area of research and many design decisions was made to make it as functional as possible.

The tool is built with Java, and uses an object-oriented programming approach and the user interface is built with Java Swing. Each component of the tool is designed, implemented and evaluated separately and its results are conveyed in this report. Finally, the achievements and limitations are discussed and future improvements are proposed.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Pradnesh Sanderan*)

Acknowledgements

I would like to thank my supervisor Dr Chris Heunen for his continuous guidance and support throughout this project. I would also like to thank Jesse Sigal for his support and assistance.

Many thanks to my parents and family for their advice and encouragement. Lastly many thanks to my friends for helping me get through the long nights I spent in the Library and in Appleton Tower.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Category theory is the general theory of mathematical structures and has vast applications in almost all areas of mathematics and some areas of computer science. Monoidal categories is a subsection of category theory which generalises finite monoids. While monoidal categories are more generally represented as a collection of objects, morphisms and a tensor product, if we regard them as a combinatorial object, they can be represented purely in terms of a multiplication table.

This has applications in cryptography where it has been used in the development of quantum-resistant cryptography, and computer science, particularly in the area of programming language design and others. However, there has not been a tool that could validate if a given multiplication table is a valid monoidal category or not.

Such a tool would be beneficial in a variety of situations, from pure research in mathematics to practical applications in fields like cryptography and computer science.

1.2 Project Goals

The main aim of this project is to build a tool from scratch with a graphical user interface that when given 2 multiplication tables (1 for the category and another for the tensor product), is able to validate if the category is a valid strict monoidal category or not. For this project to be successful, the following contributions were necessary:

- **Extensive background research.** Extensive background research must be done to improve our understanding of category theory and monoidal categories and to get inspiration from any past projects done in this field

- **Deep understanding of category theory.** Designing the tool requires a concrete understanding of category theory for it to work. Without it it is possible the tool would be incomplete or faulty.

- **Designing of the tool beforehand.** Since this tool is being built from scratch, every design decision matters and would affect the outcome of the project. The design must be carefully considered to avoid any mishaps in the future

- **Implementation of table parser.** Since we are dealing with multiplication tables, an algorithm would need to be designed to extract all required information from these tables.
- **Implementation of validation algorithm.** Each property that needs to be checked would need to be implemented in the program. Hence an algorithm would need to be designed to represent each of these properties.
- **Implementation of a user-friendly Interface.** The graphical user interface would need to be easy to understand yet serve its functionality.
- **Extensive testing.** To ensure the program functions correctly, extensive testing is required. Without so we cannot be sure of its correctness.

1.3 Dissertation Structure

This dissertation is divided into 6 chapters and is structured as follows:

- In **chapter 2**, we discuss categories and monoidal categories as it is represented in category theory to build our understanding of the topic. We also review existing literature on category theories and past projects.
- In **chapter 3**, we discuss the design decisions made for this tool. It gives a justification for the design considerations made at each stage of the development process.
- In **chapter 4**, we discuss how the different components of the tool were developed and tested and the various challenges faced in doing so.
- In **chapter 5**, we discuss the results of the tests conducted on the tool, the limitations of the tests and how they could be solved.
- In **chapter 6**, we present the achievements we accomplished, limitations faced and some future work for this project. We also give a brief summary of the project overall at the end.

Chapter 2

Background

The study of finite monoidal categories has been a fascinating area of research that has seen significant attention in recent years. This chapter aims to provide an overview of the background and related work in the field and discuss the key concepts and ideas that underpin the study of finite monoidal categories. The chapter begins with an introduction to the basic concepts of category theory, including categories, functors, and natural transformations. From there, we delve into the definition and properties of monoidal categories, exploring their connections to algebraic structures such as monoids. The next section focuses on the special case of finite monoidal categories, which are defined by giving their multiplication table. The chapter concludes with a survey of previous work in the field, including existing algorithms and tools for checking the legality of monoidal categories. The chapter sets the stage for the subsequent chapters, which will detail the implementation and evaluation of the tool for exploring finite monoidal categories.

2.1 Categories

In mathematics, a **category** is a collection of **objects**, together with **morphisms** between them. The objects can be anything such as numbers, sets or functions and the morphisms represent the relationship between the objects.

2.1.1 Categories in a nutshell

Category theory is based on the abstraction of the arrow,

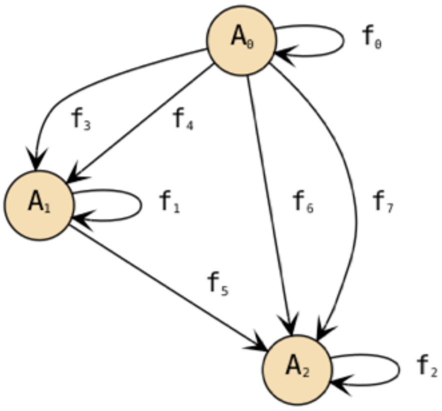
$$f : A \to B$$

In category theory, A and B are called **objects** and the arrow relating these 2 objects is called a **morphism**. In this case, the source of the morphism is *object A* and the target is *object B*. We have seen similar directional structures before as they occur widely in other mathematical concepts and topics, for example, set theory, algebra, topology and logic. For example in set theory, A and B may be sets and f could be a function with A as the domain and B as the codomain, and in logic, A and B may be propositions and f would be a proof for $f : A \rightarrow B$.

2.1.2 Categories as graphs

In category theory, a category is often described as a graph, mainly a **directed graph** (directed multigraph) and for this project, it would be easier to view the categories as said graphs. Based on this, we are able to give a formal definition of a category.

> **Definition 1:** A graph is a pair N, E of classes (of nodes and edges) together with a pair of mappings $s, t : E \rightarrow N$ called source and target respectively. We write $f : A \rightarrow B$ when f is in$E$ and $s(f) = A$ and $t(f) = B$. An example of a graph is shown below:



> **Definition 2:** A category is a **graph (O, M, s,t)** whose **nodes O** we call objects and whose **edges M** we call **morphisms**. Just like the graphs, these morphisms have **sources** and **targets**. For example in$f : A \rightarrow B$, f would represent the morphism f, the source of the morphism is object A and the target of the morphism is object B.

Although we often represent the graphs as categories in category theory, not all directed graphs can be considered categories. A graph would need an additional structure which is the collection of objects and the collection of morphisms where each morphism would require a well-defined source and target object. Furthermore, to be considered a valid category, the directed graph would need to satisfy certain properties:

- **P1:** For every object, A, in the category, there must be a morphism ID_A which has object A as the source and target object (it maps the category from object A to object A), is both left-associative and right-associative and obeys the unitality condition where for any morphism f, composing it with the identity morphism does nothing:

$$f \cdot IDA = f = IDA \cdot f$$

- **P2:** For any 3 $Objects\ A, B, C$, if there is a $morphismf$, from A to B ($f : A \rightarrow B$) and a $morphismg$, from B to C ($g : B \rightarrow C$) then there must also exist a morphism h, from A to C ($g \cdot f : A \to C$) ("$\cdot$" represents the composition of two morphisms)

- **P3:** For any 3 morphisms, h, g, f in the category, $(h \cdot g) \cdot f = h \cdot (g \cdot f)$ whenever either side is defined.
- **P4:** If $f : A \to B$, then $f \cdot id_A = id_B \cdot f = f$.

2.1.3 Terminology

Now that we have a basic understanding of categories, before we move any further we will set some terminology that will be used throughout the dissertation to prevent any further confusion. Given a **category C**, the term $ob(C)$ refers to a class whose elements are objects in C and $hom(C)$ refers to the class of morphisms in C. We will refer to the mappings between objects as **morphisms** rather than functions or arrows. We cannot always consider morphisms as functions as if we were to look at a monoid as a category, then the elements in the category would be the morphism and not a function. Each morphism will have a **domain** and **codomain** rather than source and target as this seems to suit the concept of categories more. It allows us to distinguish between categories and non-category-directed graphs which use source and target.
We will denote **Objects** in the category with uppercase letters and morphisms with lowercase letters. For example, as shown below, f is the morphism with object A as the domain and object B as the codomain.

$$f : A \to B$$

If we have a composition of morphisms, we would write both morphisms together and only state the domain and codomain of the composition. For example if, $f : A \to B$ and $g : B \to C$ then $fg : A \to C$. Lastly, $HomC(A,B)$ or $Hom(A,B)$ is the notation used to represent the set of all Morphisms in a category C that has A as their domain and B as their codomain. If the category is obvious from the context, $Hom(A,B)$ can be used instead.

2.1.4 Morphisms

As said, above, a category also consists of a composition of morphisms. This means that 2 or more morphisms can be joined to relate 3 or more objects. The relations between morphisms are depicted using commutative diagrams which we will look into later to represent the coherence conditions for the monoidal categories 2.2.4.
As mentioned, morphisms are used to represent the relations between objects in a category. Given a morphism $F : A \to B$, it can have some of these properties:

- **Monomorphism:** a monomorphism is a type of morphism between two objects that behaves like an injective function in set theory (it is left cancellative) whenever two morphisms composed with the monomorphism result in the same morphism, those two morphisms themselves must have been the same.

  If $f \cdot g = f \cdot h$, then $g = h$ for all morphisms $g, h : X \to A$.

- **Epimorphism:** Epimorphism is a type of morphism between two objects that behaves like a surjective function in set theory. Similar to monomorphism but is instead right cancellative.

$$\text{If } g \cdot f = h \cdot f, \text{ then } g = h \text{ for all morphisms } g, h : B \to X.$$

- **Bimorphism:** A morphism that has the properties of a monomorphism and an endomorphism is a bimorphism.
- **Isomorphism:** an isomorphism is a type of morphism between two objects that behaves like a bijective function in set theory. If f is an isomorphism then there exists a morphism $g(g : B \to A)$ that is the inverse of f.

  $$f \cdot g = id_A \text{ and } g \cdot f = id_B, \text{ where } id_A istheidentitymorphismofobjectA$$

- **Endomorphism:** An endomorphism that has an Object X as the domain and Codomain. Then $f : X \to X$. the identity morphism is a type of endomorphism. The significance of endomorphisms in category theory lies in their ability to define fundamental algebraic structures like monoids, groups, and rings. Moreover, the set of all endomorphisms of an **object X** in **category C** forms a monoid under the composition of morphisms, referred to as the endomorphism monoid of X. The study of endomorphism monoids provides insights into the internal structure of the category and can help classify objects based on their endomorphism monoids.

2.1.5 Functors

A **functor** is a mapping between categories that preserves the structure of the categories. Given a *Functor F* in *Category C*, where $F : C \to D$, it assigns to each *object X* in C, an $objectF(X)$ in *category D*. It also assigns to each morphism $g : X \to Y$in C, $F(g) : F(X) \to F(Y)$ in such a way that :

- **The functor preserves the composition:** for any composable pair of morphisms $f : X \to Y$ and $g : Y \to Z$ in C, we have $F(g \cdot f) = F(g) \cdot F(f)$ in D.
- **F preserves the identity morphisms**. For any object X in C, we have $F(id_X) = id_F(X)$

The concept of functors plays a big role in category theory as they provide a way of comparing and relating different categories and allow us to import techniques and concepts from one category to another. This concept will be revisited later in the chapter on bifunctors.

2.1.6 Natural transformations

In category theory, natural transformation is a relation between two functors. It is a way of transforming one functor into another while respecting the structure of the categories involved. Supposed we have two functors F and G between the categories C and D. The natural transformation a where $a : F \to G$is a family of morphisms $aX : F(X) \to G(X)$, one for each object X in C such that for every morphism f, in C where $f : X \to Y$, the following diagram commutes.

$$\begin{tikzcd}
F(X) \arrow[r, "a(x)"] \arrow[d, "F(f)"'] & G(X) \arrow[d, "G(f)"] \\
F(Y) \arrow[r, "a(Y)"'] & G(Y)
\end{tikzcd}$$

Natural transformations allow us to compare and relate different functors and it provides a way of measuring how similar or different two functors are.

2.1.7 Finite vs infinite categories

A category is considered to be finite if it has a finite number of objects and morphisms. To be more precise, a category is finite if there exists a finite set of objects $Ob(C)$ and a finite set of morphisms $Hom(C)$ such that the composition of any two morphisms in $Hom(C)$ is again a morphism in $Hom(C)$. An example of a finite category is given below.

$$\begin{tikzcd}
& a \arrow[dr, "g"'] \arrow[dr, bend left, "h"] & \\
b \arrow[ur, "f"] \arrow[rr, "k"'] & & c
\end{tikzcd}$$

Here *a,b and c* are objects and *f, g and h* are morphisms with domains and codomains as depicted and the existence of an identity morphism for each object is assumed.
For this project, we will be only working with finite categories as the category would need to be finite to be able to generate a valid multiplication table. Otherwise, we would be left with an infinite multiplication table.

2.2 Monoidal categories

Since the project revolves around monoidal categories, finite strict monoidal categories to be precise, in this chapter, we will be looking into the formal definition of a monoidal category, its properties and how it can be represented as a multiplication table.

2.2.1 Monoids

A monoid is a set with a binary operation and an identity element that satisfies certain axioms. The binary operation needs to be associative (the order of applying the operation does not matter) and the identity element, which is an element, when combined with any other element using the binary operation must return the other element unchanged.

A more formal definition would be that a monoid is a triple (M,OP,e) where M is a set, op is the binary operation and e is the identity element, such that it satisfies these properties:

- **Associativity:** For all a,b and c in M, $(aOPb)opc = aOP(bOPc)$
- **Identity:** there exists an element e such that for all a in M $eOPa = a = aOPe$.

In Haskell, a monoid can be expressed as a type class as follows:

```
Class Monoid m where
    mappend :: m -> m -> m
    empty :: m

mappend :: m -> (m -> m )
```

in Haskell, the definition of mappend is curried. It can be interpreted as the mapping of every element of m to a function as shown above. The definition of a monoid as a single-object category mentioned in (), where the elements of the monoid are represented by endomorphisms$(m \rightarrow m)$, arises from this interpretation.

2.2.2 Formal definition

A **Monoidal Category** is a Category C, equipped with a monoidal structure. The monoidal structure consists of a bifunctor called the tensor product, an object *I* called the monoidal unit and three natural isomorphisms. A deeper explanation of the monoidal structure and its properties is explained in the next few subsections

2.2.3 Tensor product

A **bifunctor** is a functor whose domain is t a product category. We refer to them as the tensor product in monoidal categories and in monoidal categories, they take 2 arguments each from a different category and produce a result as shown :

$$\otimes : C \times D \rightarrow E$$

Specifically, C and D are categories and E is a monoidal category. We use the infix notation so that the value at (A,B) is written as $A \otimes B$.

2.2.4 Natural isomorphisms

A monoidal category must have 3 **natural isomorphisms** *a,r and l* for each object A, B, and C, which are subject to certain coherence conditions. These conditions ensure that the tensor operation:

- **Is Associative**. There exists a natural isomorphism a, called the associator whereby $a(A,B,C) : A \otimes (B \otimes C) \rightarrow (A \otimes B) \otimes C$;
- **Has *I* as left and right identity:** there are natural isomorphisms *r* and *l* which are right and left unitors respectively and satisfy:

1. $rA : A \otimes I \rightarrow A$
2. $lA : I \otimes A \rightarrow A$

If the monoidal category had such natural isomorphisms and they satisfy the said conditions, then they must make the following diagrams commute.

$$\begin{tikzcd}
A \otimes (I \otimes B) \arrow[rr, "a\,(A, I, B)"] \arrow[dr, "A \otimes l\,(B)"'] & & (A \otimes I) \otimes B \arrow[dl, "r\,(A) \otimes B"] \\
& A \otimes B &
\end{tikzcd}$$

The triangle diagram depicts the concept of the unit law, which declares the existence of certain objects (represented by I) that serve as identity elements for the ⊗ operation. It consists of three vertices, each of which is labelled with an object of the category, and three edges labelled with ⊗ operations. One of the vertices is marked with the distinctive object I, and the other two vertices are linked to it by edges labelled with ⊗ operations. The diagram affirms that if we begin at any vertex and follow the edges towards the vertex marked with I, we obtain the same result.

$$\begin{tikzcd}
& A \otimes (B \otimes (C \otimes D)) \arrow[dl, "A \otimes a(B, C, D)"'] \arrow[dr, "a\,(A, B, C \otimes D)"] & \\
A \otimes B \otimes C \otimes D \arrow[d, "a\,(A,B \otimes C, D)"'] & & A \otimes B \otimes C \otimes D \arrow[d, "a\,(A \otimes B,\ C, D)"] \\
A \otimes B \otimes C \otimes D \arrow[rr, "a\,(A, B, C) \otimes D"'] & & A \otimes B \otimes C \otimes D
\end{tikzcd}$$

The pentagram diagram illustrates the principle of associativity, which asserts that the order in which we group three elements using the ⊗ operation is immaterial. It comprises five vertices, each of which denotes an object of the category, and ten edges labelled with ⊗ operations. These edges combine to form a pentagon, where each vertex is connected to its adjacent vertices by edges labelled with ⊗ operations. The diagram affirms that if we traverse the edges around the pentagon in either direction starting from any vertex, we obtain the same outcome. These diagrams provide a way to visualize and reason about the coherence conditions that must hold in a monoidal category.

2.2.5 Strict vs free categories

In category theory, a monoidal category can either be strict or free. A strict monoidal category is a category in which the composition of morphisms satisfies the associative and unitality properties stated above in 2.1.2. A free monoidal category on the other hand Is a category built from a set of generators (the elements of a given set S, which is usually referred to as the set of generating objects or generating morphisms) and relations, subject to the requirements of composition and identity. For example, the free category F(S, R) with a set S of generators and a set R of relations, has one object whose morphisms are formed by concatenating generators from S and the inverses of generators from S, subject to their relations in R. For this project we will only be dealing with strict monoidal categories as it has been widely accepted that every monoidal category is monoidally equivalent to a strict monoidal category.

2.3 Previous work

In our research, we were not able to find any previous work about this specific project or topic per se, but we were able to find previous work on the broader topic of category theory, monoidal categories and strict monoidal categories. Our initial readings were Chapters 1 and 2 of Bar and Wells's "Category Theory for computer science ", Bartosz Milewski's "category theory for programmers", and Fong and Spivak's "An Invitation to applied category Theory. These 3 papers gave us a good understanding of the basics of category theory.

”Category theory for computer science” [1] provided a good introduction to the concept of categories where it was compared to set theory and graph theory, something more relatable than the abstract idea of categories. The paper talks about sets in mathematical settings, how it is used, their notations and their specifications. It also talks about functions and their properties and their relation to morphisms. For example, it talks about how if a function is injective then it has a one-to-one relation between the domain and the codomain. It then talks about directed graphs, mainly directed multigraph with loops which is the main graph used in category theory and its properties.

"Category theory for programmers" [5], gives a more in-depth view into morphisms. It explains how the arrows in graph theory are related to morphisms in category theory and the various properties that these morphisms would have to satisfy. "An Invitation to applied category Theory" [4] builds on this idea of morphisms and introduces us to identities and objects. It also talks about free categories which are not touched on by previous papers and refer to a category whose objects are the vertices V and whose morphisms from object 1 to object 2 are the paths from object 1 to object 2. The paper also talks about isomorphisms and pre-orders as categories. Knowledge of both which are used by the tool being developed to complete the multiplication table.

The Wikipedia article on category theory [7] gives a better understanding of the main concepts of category theory and it was our main reference point when completing this project. It gives us the formal definitions of categories, morphisms, and the axioms of composition. It also explains the role of morphisms and functors in category theory. The Wikipedia article on monoidal category [8] and strict 2-category [9] was also a main reference point when it came to those 2 topics. Unlike the previous articles, it gave a formal definition of those categories and the properties that make up the category such as the bifunctors, the covariance and contravariance of functors and the conditions that each morphism would have to hold in these categories.

Building on that, the paper “Finite tensor categories” by Pavel Etingof and Viktor Ostrik [3] gives a deeper insight into monoidal categories. Although tensor categories are not exactly monoidal categories as stated in the textbook “Tensor Categories” by Etingof, Gelaki, Nikshych, and Ostrik, [6] they do possess similar structures especially when the mathematical structures of both categories are concerned. The paper gives a better understanding of finite tensor categories which in turn gave a better understanding of finite monoidal categories.

In terms of the technical side of this project (coding, data structures and frameworks), not much past research relating to this project was found. However, we did find a research paper that helped with the development of the project which is the paper DisCoPy: Monoidal Categories in Python by Giovanni de Felice, Alexis Toumi, and Bob Coecke [2]. The paper is about an open-source toolbox for computing monoidal categories in Python. The data structures and methods implemented in this toolbox such as the implementation of the Diagram class in monoidal.py which is the implementation of the arrows of a free monoidal category and the Tensor class in tensor.py would mean that we would not have to reinvent the wheel, or we could just restructure our implementations similarly when we are implementing it into our project. The only downfall to this paper is that it does not touch on providing valid solutions for the categories nor does it talk about solving the multiplication tables, hence it cannot really be considered past work on the same topic, but it is a good place to start for the project.

Chapter 3

Design

The design chapter of this dissertation aims to outline the process and decisions made in the development of this project. This chapter will detail the various design considerations, such as the choice of programming language, style of programming, and algorithms used in the implementation of the tool. The chapter will begin with an overview of the project requirements and the challenges and limitations that were encountered during the design process. It will then delve into the specific design choices made and the reasons behind them. Overall, the design chapter aims to provide a comprehensive account of the design process, from initial concept to final implementation, and to justify the choices that were made at each stage of development.

3.1 Requirements

The requirements section of this dissertation outlines the key features and functionality that are essential for the successful implementation of the tool. In this section, we will discuss the technical requirements, and design requirements that must be taken into account when developing the tool.

3.1.1 Backend

3.1.1.1 Validating the monoidal categories

- **Speed**: The tool should take no more than 0.5 Seconds to validate a category. This is to ensure that the tool is responsive and no downtime is noticed by the users.
- **Error handling**: The tool should not just throw every error encountered without any indication of what is causing it. Instead, it should pinpoint the property the category does not satisfy and the morphism or object causing it.
- **Correctness**: The tool must only validate a category if the category given is a valid monoidal category and must state it is not if an invalid or incomplete category is given.

- **Capability**: Given any complete multiplication table, the tool should be able to determine if it is a valid category or not, subjective to time and space constraints.

3.1.2 User interface

- **Responsiveness**: The interface must be responsive to the user without any noticeable delays.
- **Intuitiveness**: The interface should be easy to understand and the user should be able to use it without any prior training.
- **Functionality**: The interface should be able to display tables according to the size given to the user, be able to send the data inputted to the backend and then be able to display the results of the validating process to the user.

3.1.3 Tool overall

- **Robustness**: The tool must not crash under normal circumstances.
- **Functionality**: Given the multiplication table the tool must be able to correctly determine if it is a valid Monoidal category or not.
- **Ease of Use**: The tool should be easy to be used and responsive.
- **Maintainability**: The code should have proper documentation, the variables and methods should have meaningful names and the tool overall should be easy to understand by another developer. This is to allow the tool to be easy to maintain and improve in the future.

3.2 Design choices

In this section, decisions made before the start of implementation are explained

3.2.1 Programming language

When starting the project, we contemplated using Java, Python and Haskell as the main programming language for the background. we had chosen these languages because of the previous work done in these languages that are related to this project. In the end, we chose Java because of certain reasons. Firstly, Java is an object-oriented programming language (OOP) which is more well-suited for building complex programs. A more in-depth reasoning for choosing OOP-style programming is given in the next chapter. Another reason for choosing Java was because the project could then be compiled and run on a Java Virtual Machine(JVM). Although it was not used, the usage of the JVM was considered an important factor because of its cross-platform capabilities, its excellent memory management through garbage collection to reduce the risk of memory-related errors and its usage of a just-in-time compiler which would optimize the performance of the tool at runtime. Lastly, the large developer community and extensive libraries were a big factor in choosing it. The various Graphical user interface libraries such as Java Swing were a big factor in choosing it as it would make building the frontend part of the project simpler. Furthermore, the large community would mean that for any problem we run into, there would probably be a blog post or discussion on it already. We did not choose Python because, it is an interpreted language, which means that it may be slower than a compiled language like Java for performing complex computations. Additionally, Python's dynamic typing may make it harder to catch errors in the code. Although functional programming languages such as Haskell have strong support for algebraic data types and pattern matching, which can be useful for representing the structure of monoidal categories, we were not very familiar with the language and it seemed to be a waste of time, trying to learn a new language when we are already fairly proficient in one. JavaScript and C++ were also considered for their performance and popularity but were rejected for the same reasons as Haskell.

3.2.2 Programming style

Functional programming had a couple of advantages for this project that we considered at the start when designing this program. The advantages mainly were that there already exists a partial tool, written in Haskell that we could build on, that developing the tool may be simpler if done in a functional programming language as most books and research papers on category theories and monoidal categories had explained them in terms of Haskell code. Finally, it was that functional programming languages have strong support for algebraic data types and pattern matching which would make it easier to represent the monoidal categories. However, we decided to choose OOP because of a variety of reasons.

- It provides us with a way to organize and encapsulate the data and functionality associated with monoidal categories. It seemed logical to use OOP to create objects to represent the elements of the monoidal categories (multiplication table, morphisms, tensor products) and to define their behaviour through methods. Encapsulating these elements in objects, provides us with a way to organize and manage the complexity of the monoidal categories in a modular fashion, making it easier to maintain and modify.

- Another reason is that OOP allows us to reuse code and implementations through inheritance. This would become useful in cases when more than 1 category or morphism is involved.

- We chose not to use functional programming because firstly it is immutable. Data structures created with functional programming cannot be modified later which makes it less suitable for usage with monoidal categories which often require elements to be modified.

- Next, Functional programming languages tend to favour pure functions and immutable data structures, which can sometimes result in slower performance compared to languages like Java, which are designed for performance-critical applications.

- Lastly, the type system is more complex and would require a significant amount of time and effort to learn. It seemed more logical to use Java and channel that time and effort into improving the program

3.2.3 Usage of previous work

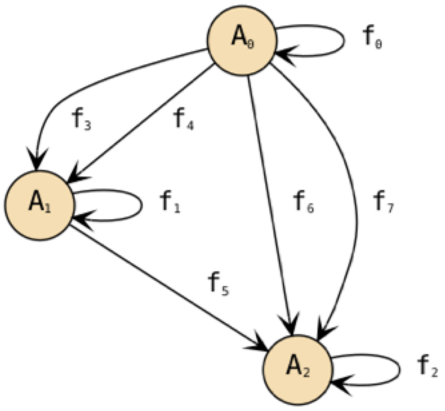
we chose not to use the previous work done (DiscoPy and the partial tool done in Haskell) mainly because we preferred building the tool from scratch as we would then be more familiar with it and easier to implement. We had chosen not to use the DiscoPy toolbox because firstly, it only works with free monoidal categories but we would be working with strict ones. Next, the current toolbox does not have any relation to multiplication tables or validating categories hence we would have to implement a significant majority of the code myself. We chose not to build upon the current tool that was built in Haskell with an SAT solver firstly because We are not familiar with Haskell or functional programming as mentioned above. Next, as mentioned above, we believed that an OOP approach would be most suitable for this project rather than a functional programming approach.

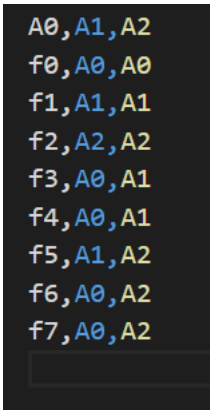
3.3 First implementation

Since it was our first time working with categories and it was also the first time categories were being represented as multiplication tables, we found it easier to build a tool that would take a category as input and then check if it is a valid category. If it is then it would output the various multiplication tables that could be formed from this one valid category. We found that doing this reverse-engineering approach first would help build our understanding of the multiplication table and how categories can be checked to see if they are valid or not.

3.3.1 Input CSV

The tool would take a CSV file as input and this file would have all the information required to represent the category. The first line in the file would be the name of all the objects in the category and the following lines would be the names of each morphism and their respective domain and codomain objects. For example for the following category, the CSV file is listed below it.





```
A0,A1,A2
f0,A0,A0
f1,A1,A1
f2,A2,A2
f3,A0,A1
f4,A0,A1
f5,A1,A2
f6,A0,A2
f7,A0,A2
```

3.3.2 File reader

The file reader was designed to read the CSV file and save each object as a State type and each morphism as a Morphisms type. It would also check each morphism and determine if it is an identity morphism or not.

3.3.3 Validating the category

3.3.3.1 Checking identity morphisms

The method used to check the identity morphisms was designed such that it would check if the identity morphisms identified by the file reader possess the properties of an identity morphism in a valid category which is that it has the same object as its domain and codomain, and it satisfies the unitality condition. If a morphism does not satisfy these properties, we do not throw an error. Instead, it just removes the morphism from the list of identity morphisms.

3.3.3.2 Checking composition

The tool was designed to fill up the multiplication table with morphisms while checking its composition. As the method traverses the multiplication table, it fills it up with the appropriate morphism or fills it up with "-" if the composition does not exist. While filling up the table, the method checks If the domain of the first morphism, the codomain of the first morphism and the codomain of the second morphism are different (the codomain of the first morphism and the domain of the second morphism must be the same for there to be a composition). If they are different then it means that there are 3 different objects involved and the following property must hold:

$$\text{If } f : A \to B \text{ and } g : B \to C \text{ then there must be } h : A \to C$$

The method then checks every morphism until it finds a morphism with object A as the domain and object C as the codomain. If it does not find such a morphism. It prints out "This is not a valid category" and the tool ends as the category does not satisfy the composition of morphisms property.

3.3.3.3 Checking associativity

If the category passes the composition property check then, it checks the associativity property. For this check, the method takes 3 morphisms and checks if the composition exists. If it does then the following property must hold:

$$h \cdot (g \cdot f) = (h \cdot g) \cdot f$$

For any morphism, if the property does not hold then it returns that it is not a valid category.

3.3.3.4 Outputting the various tables

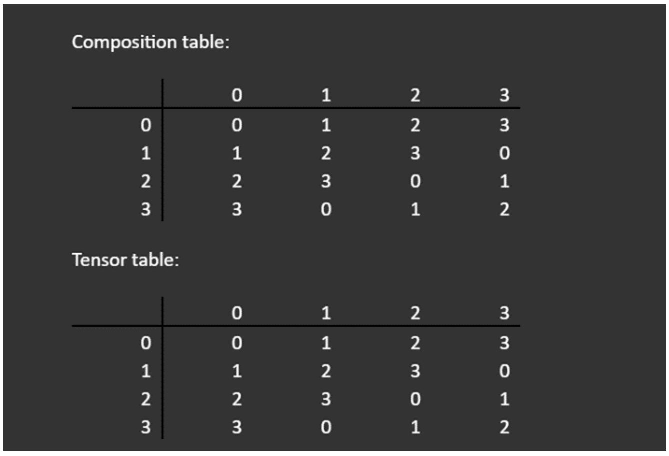
While filling up the table, the method keeps track of all the entries in the table that could be replaced by a different morphism (this happens when there is more than 1 morphism with the same domain and codomain). The method printTable then prints the original table the previous method had filled and also different variations of the table with different morphisms where applicable. Hence it outputs every possible multiplication table that the category could produce.

3.4 Second implementation

In the second implementation, we changed the methods to better suit the project whereby in this implementation, the tool would read a multiplication table, rebuild the table in the program, create the appropriate morphisms and objects and then check the properties of the category followed by checking the monoidal properties of the category.

3.4.1 Input CSV

The tool was designed to take 2 CSV files; one to represent the multiplication table of the category and another to represent the multiplication table of the tensor product. The CSV file is structured such that it reflects a multiplication table with columns and rows. An example of a category and tensor product multiplication table is given below along with a screenshot of the CSV file.



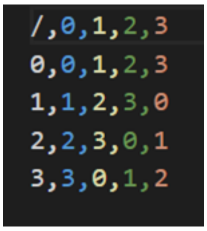
Composition table:

	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

Tensor table:

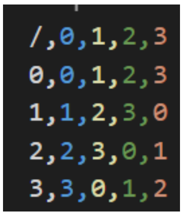
	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

Category CSV:



```
/,0,1,2,3
0,0,1,2,3
1,1,2,3,0
2,2,3,0,1
3,3,0,1,2
```

Tensor product CSV



```
/,0,1,2,3
0,0,1,2,3
1,1,2,3,0
2,2,3,0,1
3,3,0,1,2
```

3.4.2 File reader

Unlike in the first implementation where the file reader would read the files and format the morphisms and objects, this implementation was designed to just read the CSV files. The file reader was designed to read the CSV files and then store each line in the readlines variable. It would also store the row titles and column titles for the table. The same file reader would be used to read the category CSV file and for reading the tensor product CSV File. The only difference would be that for the tensor product, each line would be stored in the readTensor list. Once the lines were read, a different method would format the morphisms and objects and recreate the category table and tensor product table in the program.

3.4.3 Validating the category

Using the knowledge gained from the first implementation, designing the methods used to validate the categories was simple as it was the same properties that needed to be checked as before; the presence of valid identity morphisms, the composition of morphisms and associativity of morphisms. The methods were designed to make use of the reconstructed category multiplication table created which would be more efficient than cycling through each morphism as what was done in the previous implementation. The pseudocode for each check can be found below.:
The method used to validate the composition property of the category:

Algorithm 1 checkComp(Table table)

```
for $i$ to $morphisms.size()$ do
    for $j$ to $morphisms.size()$ do
        if $table.getMorphims(morphisms[i], morphisms[j]\textbf{is}null$ then
            continue
        end if
        a1 $\leftarrow statesA.get(morphisms[i])$
        b1 $\leftarrow statesB.get(morphisms[i])$
        a2 $\leftarrow statesA.get(morphisms[j])$
        b2 $\leftarrow statesB.get(morphisms[j])$
        if not$(a1 \neq a2\textbf{or}a2 \neq b2\textbf{or}a1 \neq b1)$ then
            $found \leftarrow false$
            for $k$ to $morphisms.size()$ do
                curr $\leftarrow morphism.get(k)$
                aState $\leftarrow statesA.get(curr)$
                bState $\leftarrow statesB.get(curr)$
                if $aState = a1\textbf{and}bState = b2$ then
                    $found = true$
                    break
                end if
            end for
            if not$found$ then
                return false
            end if
        end if
    end for
end for
return true
```

The method used to validate the associativity property of the category:

Algorithm 2 checkAssoc(Table table)

```
for $i$ to $morphisms.size()$ do
    for $j$ to $morphisms.size()$ do
        m1 $\leftarrow morphisms.get(i)$
        m2 $\leftarrow morphisms.get(j)$
        if $table.getMorphism(m1,m2)$ is $null$ then
            continue
        end if
        for $k$ to $morphisms.size()$ do
            m3 $\leftarrow morphisms.get(k)$
            if $table.getMorphism(m2,m3)$ is $null$ then
                continue
            end if
            l1 $\leftarrow table.getMorphism(m1,m2)$
            r1 $\leftarrow table.getMorphism(m2,m3)$
            check1 $\leftarrow table.getMorphism(m1,r1)$
            check2 $\leftarrow table.getMorphism(l1,m3)$
            if $tcheck1$ is $null$ or $check2$is$null$ then
                continue
            end if
            if $check1$ not equals $check2$ then
                returns  false
            end if
        end for
    end for
end for
return true
```

3.4.4 Validating the tensor product

To check if the category is a valid monoidal category, there are 6 properties that the category must satisfy. For that, we created 6 different methods that would each check one of those 6 properties. The pseudocodes for the methods designed are shown below. An explanation for each method and the properties it satisfies can be found in 4.6.

Algorithm 3 checkDomain(Table tensorTable, Table table)

```
for $i$ to $morphisms.size()$ do
    for $j$ to $morphisms.size()$ do
        f $\leftarrow morphisms.get(i)$
        g $\leftarrow morphisms.get(j)$
        leftSide $\leftarrow tensorTable.getMorphism(f.name, g.name).stateA$
        rightSide $\leftarrow tensorTable.getMorphism(f.stateA.getIdentity(), g.stateA.getIdentity()).stateA$
        if $leftSide \neq rightSide$ then
            return false
        end if
    end for
end for
return true
```

Algorithm 4 checkCodomain(Table tensorTable, Table table)

```
for $i$ to $morphisms.size()$ do
    for $j$ to $morphisms.size()$ do
        f $\leftarrow morphisms.get(i)$
        g $\leftarrow morphisms.get(j)$
        leftSide $\leftarrow tensorTable.getMorphism(f.name, g.name).stateB$
        rightSide $\leftarrow tensorTable.getMorphism(f.stateB.getIdentity(), g.stateB.getIdentity()).stateB$
        if $leftSide \neq rightSide$ then
            return false
        end if
    end for
end for
return true
```

Algorithm 5 checkIdentitiesMonoidal(Table tensorTable)

```
for i to $identities.size()$ do
    for j to $identities.size()$ do
        a $\leftarrow identites.get(i)$
        b $\leftarrow idenitites.get(j)$
        leftSide $\leftarrow tensorTable.getMorphism(a, b)$
        if $identites$ does not contain $leftSide$ then
            return false
        end if
    end for
end for
return true
```

Algorithm 6 bifunctoriality(Table tensorTable,Table table)

```
for k to $morphisms.size()$ do
    for h to $morphisms.size()$ do
        if $table.getMorphism(k,h)$ is $null$ then
            continue
        end if
        for $g$ to $morphims.size()$ do
            for $f$ to $morphisms.size()$ do
                if $table.getMorphism(g,f)$ is $null$ then
                    continue
                end if
                kh ← $table.getMorphism(k,h)$
                gf ← $table.getMorphism(g,f)$
                left ← $tensorTable.getMorphism(kh,gf)$
                kg ← $tensorTable.getMorphism(k,g)$
                hf ← $tensorTable.getMorphism(h,f)$
                right ← $table.getMorphism(kg,hf)$
                if $right$ is $null$ then
                    continue
                end if
                if $left$ not equals $right$ then
                    return false
                end if
            end for
        end for
    end for
end for
return true
```

Algorithm 7 checkAssoc(Table tensorTable)

```
for $i$ to $morphisms.size()$ do
    for $j$ to $morphisms.size()$ do
        for $k$ to $morphisms.size()$ do
            a ← $morphisms.get(i)$
            b ← $morphisms.get(j)$
            c ← $morphisms.get(k)$
            ab ← $getTensor(a,b,tensorTable)$
            abc ← $getTensor(ab,c,tensorTable)$
            bc ← $getTensor(b,c,,tensorTable)$
            bca ← $getTensor(a,b,tensorTable)$
            if $abc$ not equals $bca$ then
                return false
            end if
        end for
    end for
end for
return true
```

Algorithm 8 checkUniqueIden(Table tensorTable)

```
for i to identityNames.size() do
    currIden ← identityNames.get(i)
    valid ← true
    for j to morphisms.size() do
        currMorphism ← morphisms.get(j).name
        c1 = tensorTable.getMorphism(currIden, currMorphism)
        if c1 ≠ currMorphism then
            valid ← false
        end if
        c2 = tensorTable.getMorphism(currMorphism, currIden).
        if c2 ≠ currMorphism then
            valid ← false
        end if
    end for
    if valid then
        return true
    end if
end for
return false
```

3.5 User interface

The Graphical User Interface (GUI) was implemented last so it was only designed after a working prototype for the backend was completed. The GUI was designed to query the user for the size of the table and then display 2 empty tables that the user could fill up. The user could then click the validate button which would send the table to the backend to be validated. The results will then be displayed by the GUI.

The GUI was designed to be a simple and easy-to-use frontend to the tool and would use Java Swing framework to implement. Although we had considered using React.Js with a server for the backend at first, due to a lack of time we decided to go with the Java Swing design.

With the GUI implemented, the design of the tool was changed slightly so that it would no longer read a CSV file as stated in 3.4. Instead, it passes the inputted tables to the backend and each line of these tables would be stored in the readLines and readTensor variables. The process from then on would be similar to that as in Implementation 2, starting from when the identity morphisms get formatted.

Chapter 4

Implementation

This chapter will discuss how different components of the tool were developed and tested and the challenges faced in doing so.

4.1 Tool overview

The sequence of steps that are followed in order to input the multiplication tables and to receive confirmation of its validity as a Monoidal Category are listed below:

1. When the tool is launched, A window pops up with a text box and a submit button
2. The user is supposed to input the number of morphisms in the category to determine the size of the category multiplication table and tensor product table (the category table and tensor product table are of the same size)
3. Once they click the submit button, 2 tables appear of (size inputted +1) (the size is bigger to allow an extra column and row for the labels of the rows and columns.
4. The user then Inputs the category multiplication table into the top table and the tensor product multiplication table into the bottom table.
5. The User clicks on the validate button which sends the 2d arrays of string to the backend part of the program.
6. In the backend, the data received is first formatted into Morphisms and Objects. (How they are formatted is explained in ()).
7. Once they are formatted appropriately, the category is first checked if it satisfies the properties of a valid category.
8. If it passes the Category Validity Check, the monoidal properties of the category is checked.
9. The results of these checks are then sent to the GUI where it is displayed. If it passes all checks, a small window pops up saying that it is a valid monoidal category. Otherwise, the original window says that it is not and shows which property of the category or the monoidal property it did not pass.

4.2 Multiplication tables

In this project, we will regard strict monoidal categories as combinatorial objects, purely in terms of their multiplication tables. A combinatorial object is a structured concept composed of components and an object that can be put into a one-to-one correspondence with a finite set of integers.

The multiplication tables used in this project are composed of the morphisms in the category. Each row and column in the table corresponds to a different morphism in the category and the entry in that cell shows the morphism that corresponds to the composition of the row and column morphism of that cell. For example, given the morphisms:

$$\begin{gathered} f : A \rightarrow B \\ g : B \rightarrow B \\ h : B \rightarrow C \\ k : A \rightarrow C \end{gathered}$$

Then the cell with morphism f as the row and column g as the column would have g as an entry as it is the composition of the two morphisms. Next, the cell with morphism f as the row and morphism h as the column will have morphism k as the entry.

As for the table representing the tensor product, each cell represents the tensor product value of the row morphisms and column morphism of that cell. All cells in the tensor product table must be filled as for it to be a valid monoidal category, the tensor product of every morphism must equate to a non-null value. However, for the table representing the composition of the category, some cells will be filled with ”-” to show that a composition is not possible. For example, when the row morphism is f and the column morphism is k, the entry to the cell would be ”-”.

From these tables, we are able to identify the objects, morphisms and the domains and codomains of each morphism. This is all we would need to verify if a category is a valid monoidal category or not. We are able to identify the objects by checking if each morphism satisfies the properties of an Identity morphism. If it does then it must represent an object as the properties of a category state that each object in a category must possess 1 unique identity morphism.

Once we have identified all the identity morphisms, it is a simple task to identify the domains and codomains of each morphism. Firstly we would look at each morphism with an identity morphism as the row or column morphism of the cell. Having an identity morphism as a row or column for the cell in the table would mean that the morphism has the object that the identity morphism corresponds to as the domain or codomain respectively. Once these are identified, for every other morphism we are able to identify their domains and codomains just based on their composition and match it with every other morphism that we have already identified (9).

4.3 Code architecture

4.3.1 Reasoning behind the classes

- **Main**: The Main Class is the entry point of the program. When the tool is launched, it calls the CategoryValidatorGUI method in CategoryValidatorGUI Class which then initiates the front end of the program. The class also houses the methods used to format the morphisms, the tensors and the multiplication tables. (formatIdentities(),createTable(),createMorphs(),formatMorphs() and formatTensor()). The Main class also receives the 2d string array from the GUI and then formats the morphisms and tensor and then checks the validity of the category by calling the various checking methods implemented in the Validator class. It then sends the results of the check to the GUI.
- **Table**: This class is used to represent a multiplication table in the tool and it makes it easier to certain rows and columns in the table.
- **State**: This class is used to represent an object in a category as a Java Object. Each object has a name, a boolean if it has an identity morphism and if it does it is linked to its identity morphism.
- **Morphisms**: This class is used to represent a morphism as a Java object. Each Morphism object has a name, a domain state referred to as stateA and a codomain state referred to as stateB.
- **CategoryValidatorGUI**: This class represents the GUI of the program. It is built using JFrame and Java Swing. The class houses just the layout of the GUI and the actionListener methods which determine what happens when the submit button and validate button are clicked.
- **Validator**: This class houses all methods that are used to validate the category and monoidal properties. It contains 2 methods used to check the validity of the category as a category (checkAssoc() and checkComp()) and 6 methods used to check the monoidal properties of the category(checkUniqueIden() , checkIdentitiesMonoidal(), checkCodomain(), checkDomain(), bifunctoriality(), checkAssociativity()).

4.4 Formatting the components

Implementing the backend to the tool was the first step in developing it. To do this, the first thing needed would be the methods that would format the inputted 2d arrays into the respective components. The first component that was formatted was the morphisms, specifically the identity morphisms. This was because the identity morphisms and the number of objects would need to be known before we are able to format the other morphisms and determine their domains and codomains. Each morphism was checked if they possess the properties of an identity morphism as mentioned in (). Once they were identified, The different objects in the category were identified by subjecting each identity morphism to its own object (State object). The other morphisms were then created (each morphism represented as a Morphisms object) and their domains and codomains were set based on their composition as stated in 4.2. The pseudocode to this implementation can be found in 9. After that, the tool rebuilds the multiplication table as a 2d array of Morphisms objects. This is used to represent the category table. Another 2d array of Morphisms objects is built after to represent the tensor product table.

4.5 Validating the category

The next part of the tool is the validation part and the first step in the validation process is validating if it is a valid category or not. For the category to be considered a valid category, it must satisfy the identity morphism property(each object must have a unique identity morphism), the associativity property and the composition property. The identity morphism property is already checked when formatting the morphisms. We first check to make sure that the number of identity morphisms is greater than 0, then we check the morphisms whereby if we encounter a morphism that we are unable to set its domain or codomain then we know that the multiplication table is not valid.

The Associativity of the morphism is checked as described in 3.3.3.3. The method checkAssoc() iterates through each morphism with 3 nested for loops to check the associativity of the morphisms.

The Composition of the morphism is checked as described in 3.3.3.2. The method checkComp() also iterates through each morphism with nested for loops to check if they all satisfy the composition properties. If either of these checks fails, the tool immediately states that it is not a valid category and displays the check(s) that fail. In the command line, the user is able to see specifically the name of the morphisms that caused the check to fail.

4.6 Validating the monoidal properties

Once the tool validates the property of the categories, it checks the monoidal properties of the categories. There are 6 properties that must be satisfied by the category to be considered a valid monoidal category. These Properties are:

- **The domain property** 3:

  $$dom(f \otimes g) = dom(f) \otimes dom(g)$$

  To satisfy this equation, we would need the tensor product table of Objects instead of morphisms, which we have. Although it would be possible to reconstruct it from the given table, it would be easier instead to use the identity morphism of the object. In this case, we would use $ID[dom(f)] \otimes ID[dom(g)]$. Where ID[o] represents the identity morphism of the object o.

- **The codomain property** 4:

  $$codomain(f \otimes g) = codomain(f) \otimes codomain(g)$$

Each morphism in the tensor product multiplication table would need to satisfy this property. Just as in the Domain Property, we do not reconstruct a tensor product table for the objects so we would use its identity morphisms instead. Hence the equation would be:

$$codomain(f \otimes g) = ID[codomain(f)] \otimes ID[codomain(g)],$$

where ID[O] represents the identity morphism of object O.

- **Associativity property** 7:

  $$(f \otimes g) \otimes h = f \otimes (g \otimes h)$$

  This check is similar to the check that was done for the Category and as explained by the Pentagram diagram in 7. The only difference would be that this check would apply to every combination of morphism as opposed to only those that have a valid composition in the category check.

- **The distributivity of tensor over composition property** 6:

  $$(k \cdot h) \otimes (g \cdot f) == (k \otimes g) \cdot (h \otimes f),$$
  $$\text{if } cod(h) = dom(k) \text{ then } cod(f) = dom(g)$$

  This property is the only property that involves both multiplication tables.

- **The tensor of identities property** 5:

  $$id(A) \otimes id(B) = id(A \otimes B)$$

- **The identity property** 8:

  $$\exists I : ID_I \otimes ID_A = ID_A = ID_A \otimes ID_I$$

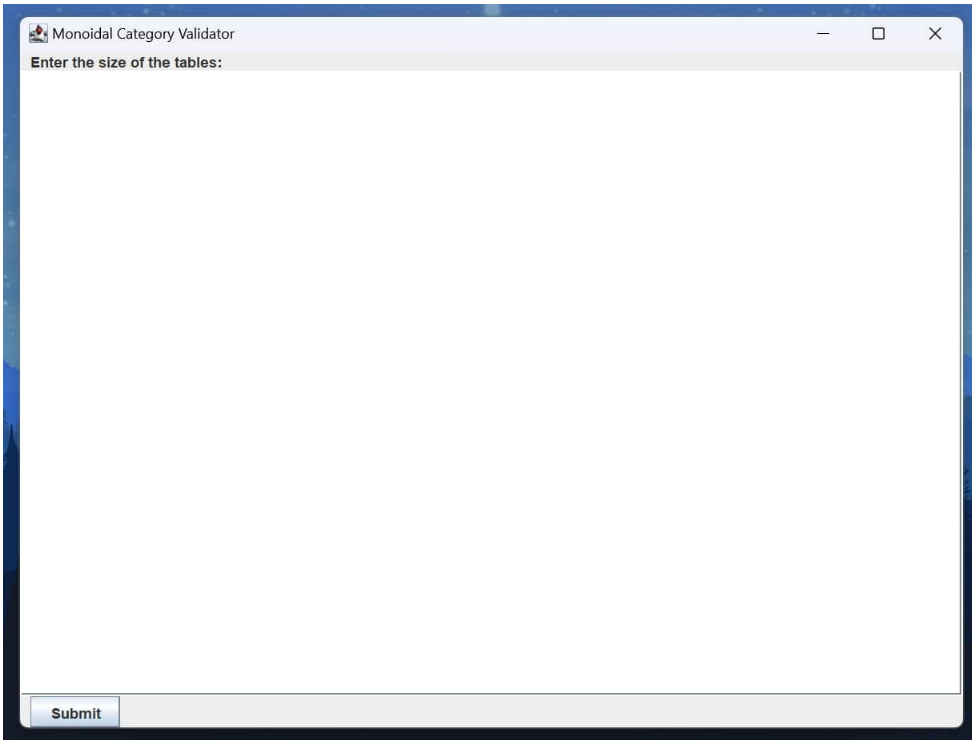
We have developed a method for each of these properties in the tool that checks the category separately. This made it easier to pinpoint which properties the category did not satisfy when it fails. Furthermore, the methods are implemented such that when a property fails, it prints which property failed and the morphisms that caused it to fail in the command line. This was implemented to give better clarity to the user as to why a property is not a monoidal category and also it makes it easier for a developer who may use this tool in the future, to maintain and improve and debug the program.

The main issue faced when trying to validate the monoidal properties of the category was developing the methods from mathematical equations. We were only given the mathematical equations that represent the properties that the monoidal category would have to satisfy and it was up to us to find a way to make the tool check these properties. However, once we had developed the method for the first property which was the Domain Property, it seemed simple enough to use the same concept for the other properties.

4.7 User interface

As mentioned in 3.5, the user interface was built using Java Swing. The GUI implemented is rather basic but serves its functions. The lack of design and extra functionality was due to its minimal requirements and a lack of time.

In the implementation, the GUI first prompts the user to input the size of the table as shown in section 4.7. Once the size has been inputted and the user clicks the submit button, the GUI displays 2 tables as shown in section 4.7. The table on the top is meant for the category multiplication table and the table at the bottom is meant for the tensor product multiplication table. The tables will be of the same size as the tensor product multiplication table would always be the same size as the category multiplication table. The GUI would then send the tables to the backend of the tool which would send the result of the validating process back to the GUI which then displays if the category is a valid monoidal category or not as shown in section 4.7.



Enter the size of the tables:

Figure 1: The user inputs the size of the table

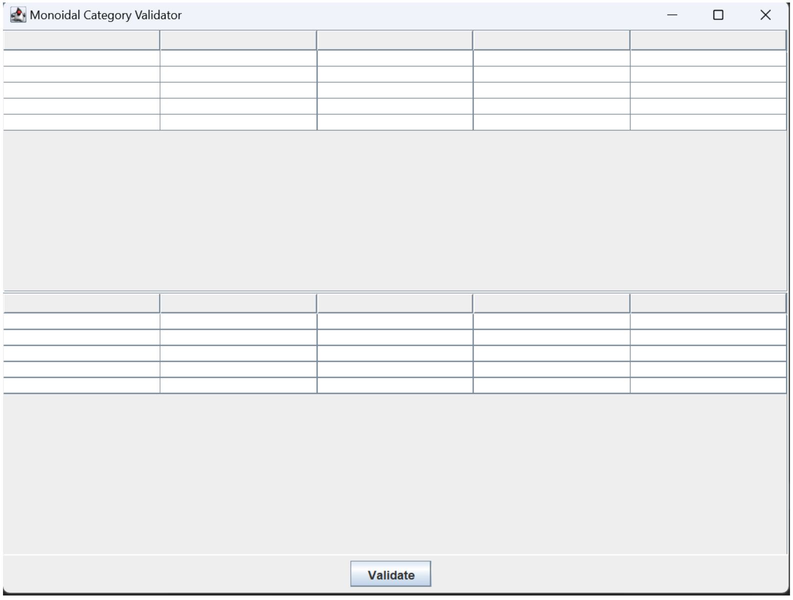
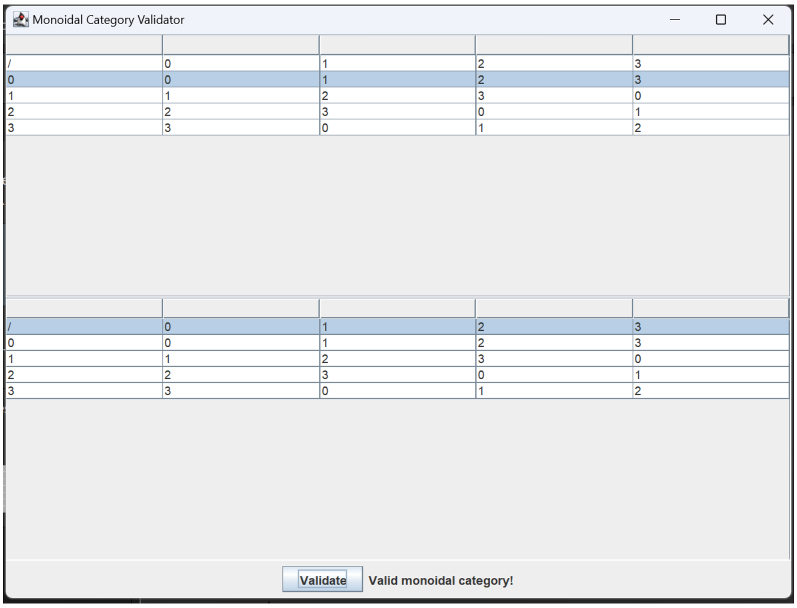


Figure 2: The GUI displays the tables to input the multiplication tables



/	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

/	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

Valid monoidal category!

Figure 3: The results of the validation process are displayed

4.8 Testing

Testing played a vital role in developing this tool as it was important to verify the functionality and correctness of a component before developing the next part of the program. This was essential as each part of the tool depends on the functionality of the other. We based our approach on testing according to Mike Cohen's Testing Pyramid which is shown below. The testing pyramid suggests that a good test suite should be designed with a majority of unit tests, followed by integration tests and a smaller number of end-to-end tests. This approach provides fast feedback, helps catch issues early, and reduces the risk of defects in the system.

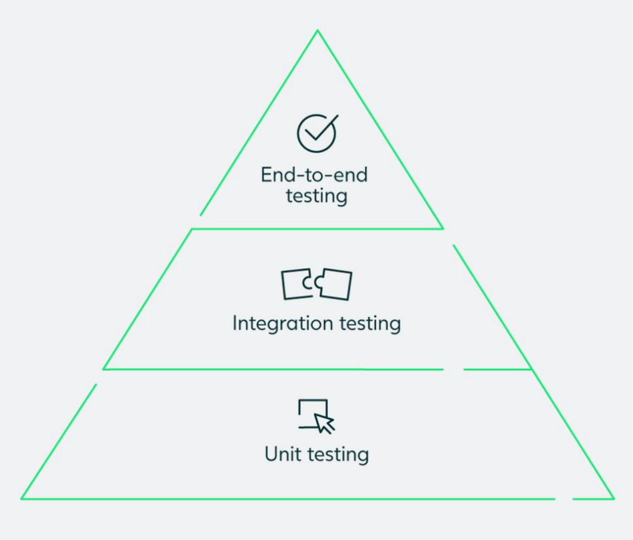
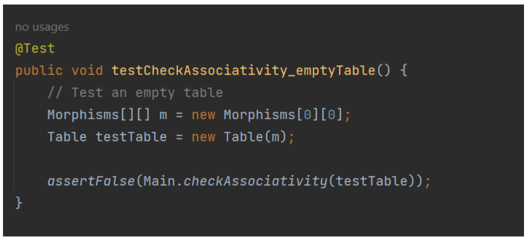


Figure 4: Mike Cohen's Testing Pyramid

4.8.1 Unit tests

Upon completion of each Major part of the tool (formatting, validating the category, validating the monoidal properties, creating the GUI), we tested each newly added component to ensure it behaved correctly. We did this by developing unit tests for each of these newly added methods. The unit tests were developed with Junit framework. We decided to use JUnit over TestNG as we had used it before and were familiar with it. The aim of each unit test was to cover all possible interactions with the program; standard interactions and invalid inputs. Using unit tests allowed me to detect bugs in the code much earlier and saved a lot of time debugging compared to if we had waited until we had developed the entire tool before testing it. A code snippet of some of the unit tests can be found below:



```
no usages
@Test
public void testCheckAssociativity_emptyTable() {
    // Test an empty table
    Morphisms[][] m = new Morphisms[0][0];
    Table testTable = new Table(m);

    assertFalse(Main.checkAssociativity(testTable));
}
```

4.8.2 Integrated testing

After Verifying that each unit of the tool works correctly by itself, we needed to test the functionality of the system as a whole and check if it functions properly when interacting with other components.

To do this, we first tested the process of validating a category. This process would check if the formatting of the objects, morphisms and category multiplication table was correct and if the checks placed to validate the properties of a valid category work well with each other. To do this we developed tests with sample categories 6.4; some valid and some invalid and checked if the current tool would pick up on their validity or invalidity. We also placed tests to check that the ability of the tool to pinpoint what properties the category did not pass and which morphisms caused this, was working well.

We then checked and tested the process of validating the monoidal properties through the same method. The difference was this time we would use valid and invalid monoidal categories along with their tensor product to test it. The main issue faced by these tests was the lack of monoidal categories to test the tool with. Creating invalid monoidal categories was simple but creating a valid monoidal category along with the tensor product was difficult which resulted in the tool not being tested as extensively as we would like it to be.

4.8.3 End-to-end testing

We did the least amount of tests for the end-to-end tests as the user interface was rather basic and its requirements were minimal. Furthermore, the primary goal of the project was to build an implementation that would validate monoidal categories hence the backend part was the most important part.

To test the User Interface, we wrote tests for every possible interaction with the interface;

standard and invalid inputs. We also wrote some tests to test its responsiveness and the speed of the tool overall(we had aimed for the tool to complete within 0.5 seconds). Due to time constraints, we were not able to get user feedback from a large sample of users. However, we were able to have some peers test it which gave some indication as to how well the tool meets its requirements.

Chapter 5

Results and evaluation

This chapter discusses the results of the tests conducted on the tool. The limitations encountered by the tests are discussed and some possible alternatives are suggested where applicable.

5.1 User Interface

5.1.1 Responsiveness

The average time taken for the tool to validate the category and display the results was 0.05 seconds(The timer was started after the user clicks the send button after they input the tables). This is lower than the target time of 0.5 seconds. However, we cannot be sure that the tool is fast and efficient as the examples used to test the tool contained categories with a maximum of 4 objects and 10 morphisms. Hence categories with a much higher number of objects and morphisms may take a much longer time than expected. Nevertheless, for the examples we had tested it on, there was no noticeable delay.

5.1.2 Error handling

Although the tool does not crash under normal circumstances, steps were taken to ensure that common errors were handled appropriately. Input checks were added to the GUI to ensure that if the user were to leave some cells empty when filling up the table, they would be prompted to complete it instead of the tool just ending. Next, when validating the category, instead of throwing an error when the category cannot be validated, the tool instead would display the properties the category does not satisfy or the reason the category was invalid to the user and then would print the morphisms or objects that caused this error on the command line to ensure the user understands exactly why the category is not valid.

5.2 Validating

5.2.1 Validating the category

The current implementation of the tool is able to validate all the categories that were given to it and had passed all the unit tests. It validates the categories based on the properties of a category in category theory and we do not see any issues that might arise from the current implementation due to the thorough testing it had undergone.It could however be optimised and improved. The current methods implemented use a number of nested for loops and nested if-else statements which could be simplified further to make the tool more efficient.

5.2.2 Validating the monoidal properties

The validation process of the monoidal properties is not as error-free as the other validation process. This is mainly due to the fact that rigorous and extensive testing could not be performed due to the lack of valid monoidal categories in the testing set. Next is because we decided to be ”clever” in some parts of the testing, mainly in the domain and codomain check in section 4.6where we used the identity morphisms of the object rather than the object itself as stated in the equation. As of current it works for the monoidal categories we tested it on, but we are unsure if this will hold for other test sets. Besides that issue, we are confident that other property checks would function well even with other monoidal categories as not only did it pass all of the unit tests, but we also followed the equations strictly when implementing it.

5.2.3 Issue faced

The first issue faced was the difficulty in formatting the morphisms and extracting information, mainly the domain and codomain from the tensor tables. It was difficult to determine the domain and codomain at the start of the table because of the little information given. Hence as shown in the pseudocode 9, when formatting the morphisms, there were many cases that were considered multiple rechecks of the table before determining the domain and codomain of a morphism. The next issue was creating methods for each monoidal property category. Doing so required a concrete understanding of the concepts which we did not have at the start.

Chapter 6

Conclusions

6.1 Project achivements

The achievements of this project can be summarised as follows:

- Extensive literature covering Category Theory, monoidal categories and past projects related to monoidal categories
- Creation of a responsive and easy to user-friendly user interface with the functionality of inputting multiplication tables and displaying if a category is valid or not and why it is not.
- Represented a category along with its components(Objects, morphisms and tensor tables) in a Java Program
- Implemented an algorithm that is able to extract all required information about a category just from a multiplication table of the composition of morphism
- Developed various Java methods to check the properties of a category
- Understood the mathematical equations representing the properties of a monoidal category and developed Java methods to check each one of the properties
- Developed a tool to check if a set of multiplication tables represent a valid monoidal category or not.

6.2 Project limitations

Despite the overall success of the project, there were some limitations. Those limitations are summarised below

- Generating monoidal valid monoidal categories is not an easy task. Hence the number of monoidal categories that we were able to test the project on was rather limited. This meant that although the tool gave positive results for the categories we tested it on, we were not able to test the tool as extensively as we wanted to. Hence there may be some test cases that we left out.

- Due to the time constraint we were not able to gain user feedback on the user interface of the program. This led to a user interface that although functional, does not have the best design.

- Due to the time constraint, optimization was not carried out on the program. Having a low runtime was not the main priority of the project hence little attention was given to the time complexity of the program. At the moment, with the test cases we developed, the tool takes on average 0.05 seconds to execute but we are unsure of how this will fair for larger categories. If optimization is carried out, the tool would run much faster and this may not be an issue anymore.

- Due to a lack of time we were unable to extend the solver to fill up partial tables. Another way to enhance the tool's functionality would be to extend it such that it is able to determine if a partially filled table can be filled to a valid monoidal category or not. This can be done by using the validating checks mentioned in 4.6 and a backtracking search. This functionality was part of the original plan design for the tool. However, due to a lack of time, we were unable to implement it even though we had already designed the algorithm.

6.3 Future work

Some Possible extensions to this project are discussed below:

- **User testing and evaluation** User Testing is a core part of any software engineering process. Not much user testing was carried out due to time constraints. A more thorough evaluation with looser time constraints would result in a more user-friendly product.

- **Optimization and speed improvement.** Not much optimization was carried out as it was not given significant importance. As of current, we are unsure of how the tool would function with much larger categories. Optimizations to the code or changing the algorithm to a much faster one would solve this issue.

- **Testing with more sample categories.** The integrated tests and end-to-end tests carried out were not as extensive as planned due to the difficulty in generating valid monoidal categories to test it on. With a looser time constraint we should be able to generate more monoidal categories of varying sizes to test the tool on, hence improving its accuracy and functionality.

- **Updating the User Interface to React.JS.** The initial plan of the project was to create a prototype in Java since We are most familiar with Java and then re-write the entire tool in Javascript which will allow me to build the user interface with React.Js, hence turning this project into a web app. However, due to the time constraint, We were not able to do so. Therefore we decided to use Java Swing instead to build the GUI

- **Extending the project to accept and validate finite 2-Categories.** A finite 2-category is a category enriched over the category of categories, meaning that for every pair of objects in the category, there is a category of morphisms between them. Besides the composition of morphisms, the finite 2-category has the horizontal composition property which allows 2 parallel morphisms to be composed, subject to certain coherence conditions such as the interchange law. finite 2-Categories can be thought of as a more structured finite monoidal category hence it would make sense to extend the tool to accept and validate these categories also.

- **Extend the project to check whether two elements of a given multiplication table are adjoint.**The project can be extended to check if 2 multiplication tables are adjoint by checking if there exist 2 morphisms that satisfy the adjointness properties. This could be done by implementing an algorithm or by using category-theoretic methods such as the Hom-Tensor adjunction theorem. Overall this extension would enhance the tool's functionality

6.4 General remarks

This report discusses the implementation of a tool that validates if a multiplication table of morphisms represents a valid finite monoidal category. All goals of the project were achieved to an excellent degree. The areas where it could be improved are mainly in the test set where more valid categories could be generated to test the tool and the GUI could be improved to be more user-friendly and better designed. Most parts of the project led to very successful results as the main functionality of the tool is met where it is able to determine if a given multiplication table represents a valid strict finite monoidal category or not. Minimal changes to the current implementation would be necessary if the project were to be extended as mentioned in 6.3. Overall we would consider the project a success given the lack of prior knowledge of category theory, the lack of previous work done related to this topic and the difficulty in representing the concepts in category theory in code form.

Bibliography

[1] Michael Barr and Charles Wells. *Category Theory for Computing Science*. Third edition.

[2] Giovanni de Felice, Alexis Toumi, and Bob Coecke. DisCoPy: Monoidal categories in python. *Electronic Proceedings in Theoretical Computer Science*, 333:183–197, feb 2021.

[3] Pavel Etingof and Viktor Ostrik. Finite tensor categories, 2003.

[4] Brendan Fong and David I Spivak. Seven sketches in compositionality: An invitation to applied category theory, 2018.

[5] Bartosz Milewski. *Category Theory for Programmers*. First edition, 2017.

[6] Dmitri Nikshych Pavel Etingof, Shlomo Gelaki and Victor Ostrik. *Tensor Categories*. the American Mathematical Society, 2015.

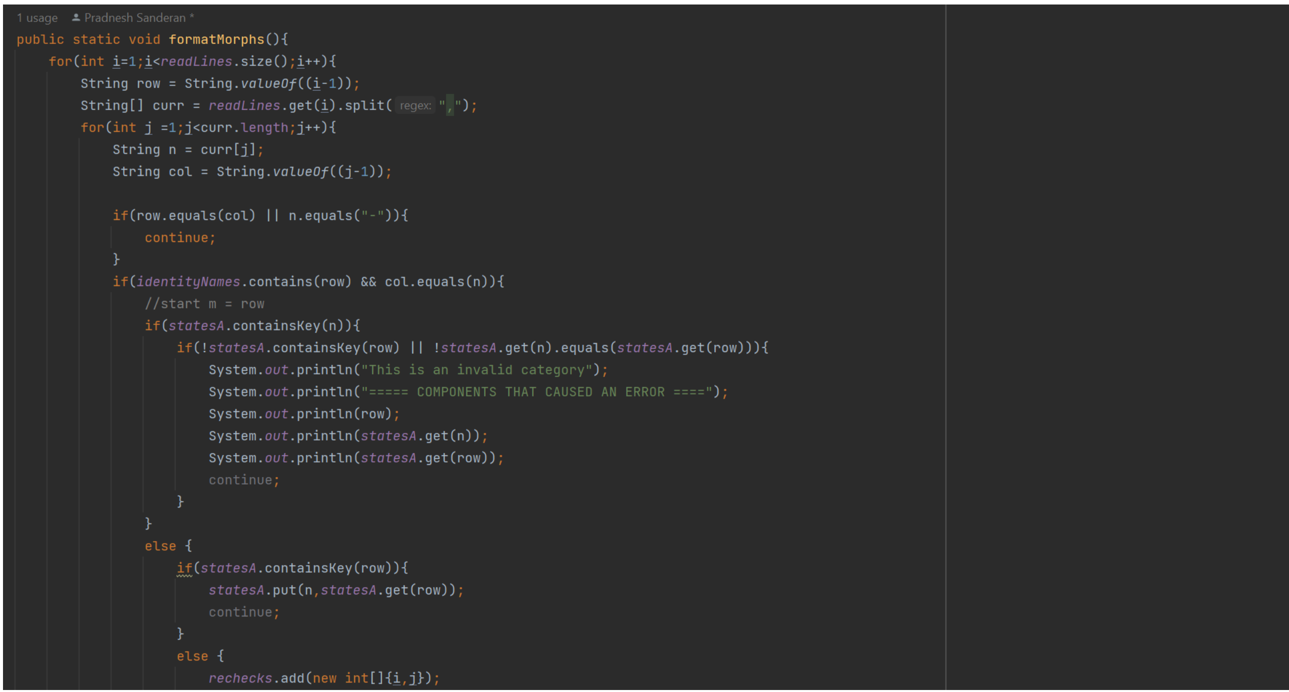
[7] Wikipedia. Category theory — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Category%20theory&oldid=1146714429`, 2023. [Online; accessed 13-April-2023].

[8] Wikipedia. Monoidal category — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Monoidal%20category&oldid=1141753927`, 2023. [Online; accessed 13-April-2023].

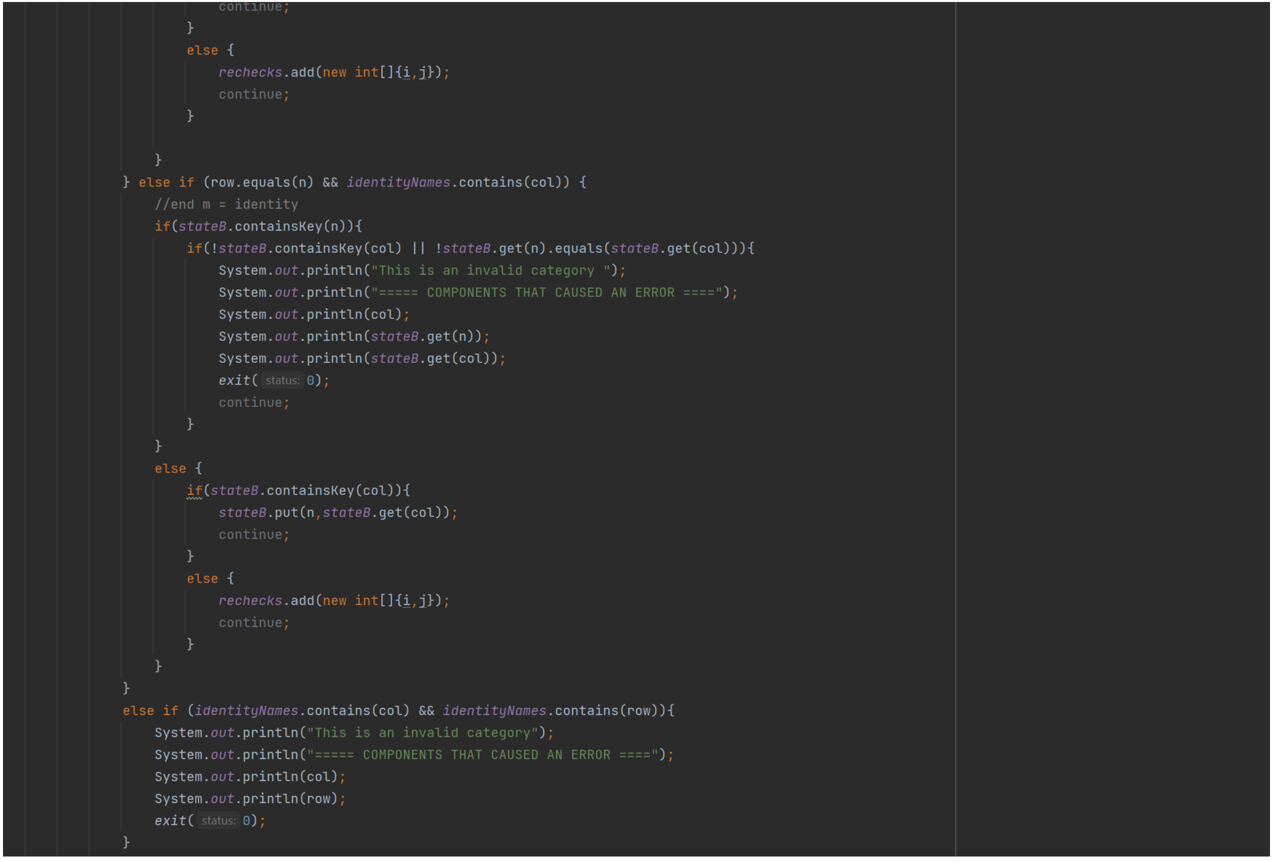
[9] Wikipedia. Strict 2-category — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Strict%202-category&oldid=1140599124`, 2023. [Online; accessed 13-April-2023].

Algorithm 9 formatMorphs()

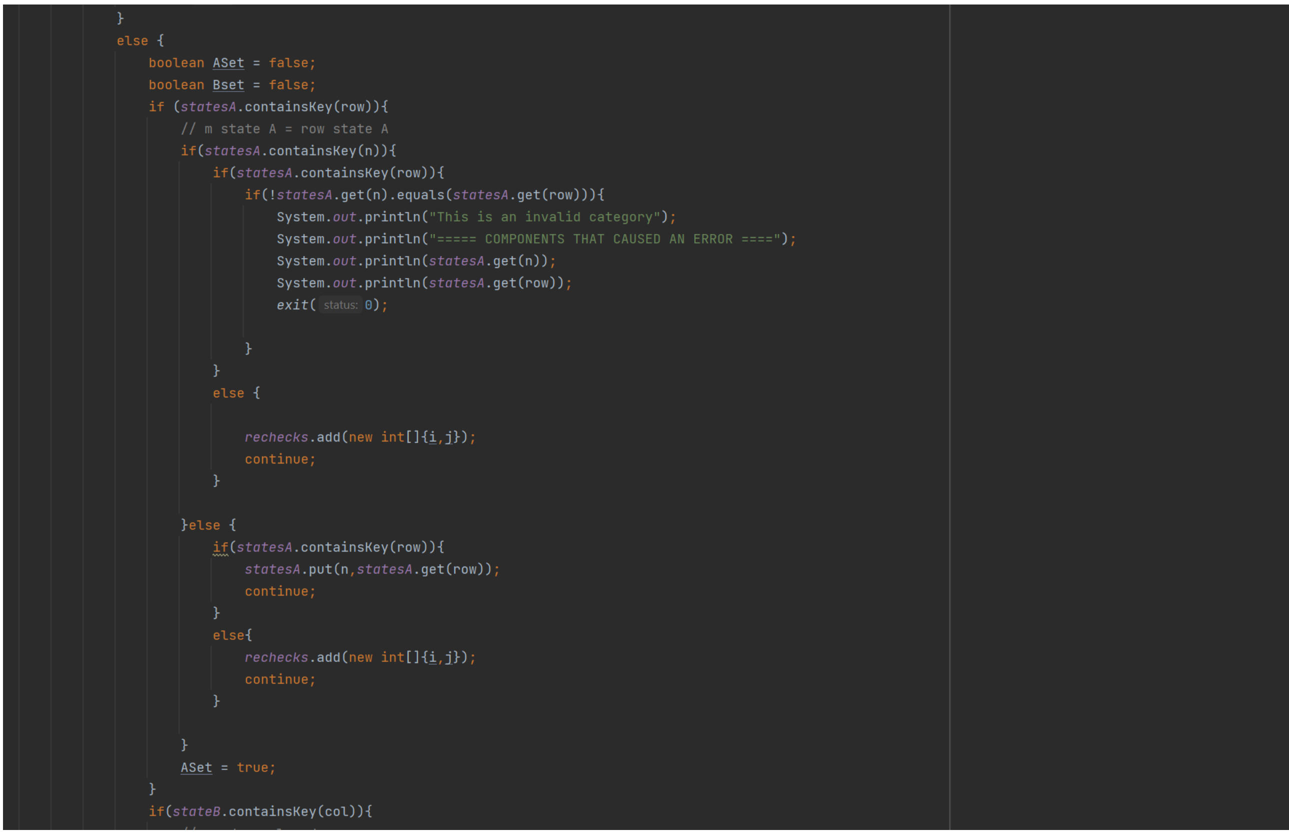
```
for $i$ to $readLines.size()$ do
    row ← $(i-1)$
    curr ← $readlines.get(i).split()$
    for $j$ to $curr.length()$ do
        n ← $curr[j]$
        col ← $(j-1)$
        if $row = col\mathbf{or}n = ''-''$ then
            It is an identity or the morphism does not exist
        end if
        if $identity.contains(row)\mathbf{and}col = n$ then
            domain of n is domain of row
        end if
        if $row = n\mathbf{and}idenitity.contains(col)$ then
            codomain of n is codomain of col
        end if
        if $identity.contains(row)\mathbf{and}identity.contains(col)$ then
            The category is invalid
        end ifElse
        if $stateA.contains(row)\mathbf{and}stateA.contains(n)$ then
            if $ifdomain(row) \neq domain(n)$ then
                The category is invalid
            end if
        end ifElse
        Add n to rechecks
        if $statesA.contains(row)$ then
            domain of n is domain of the row
        end if
        if $stateB.contains(col)$ then
            if $stateb.contains(n)anditisnotequaltocodomain(col)$ then
                it is an invalid category
            end ifElse
            codomain of n is codomain of n
        end ifElse
        add n to rechecks
    end for
end for
```



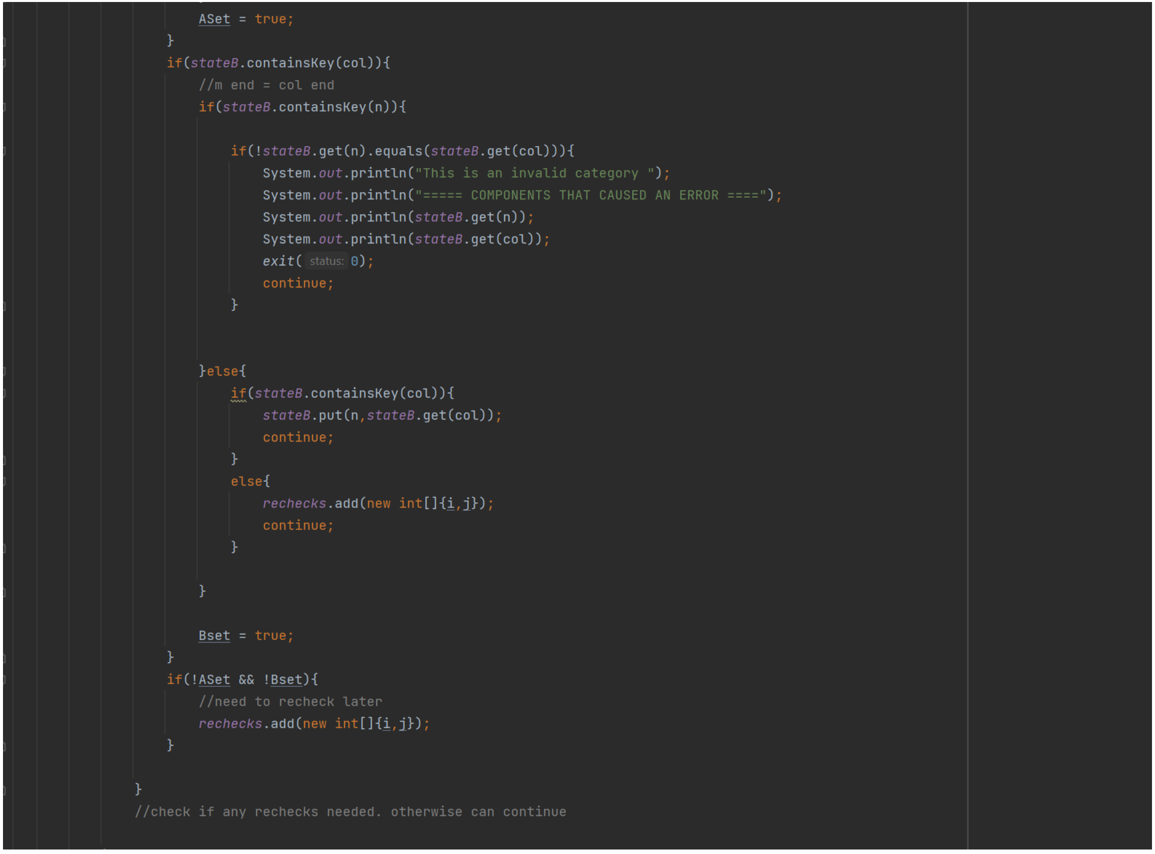
```
1 usage  Pradnesh Sanderan *
public static void formatMorphs(){
    for(int i=1;i<readLines.size();i++){
        String row = String.valueOf((i-1));
        String[] curr = readLines.get(i).split( regex: ",");
        for(int j =1;j<curr.length;j++){
            String n = curr[j];
            String col = String.valueOf((j-1));

            if(row.equals(col) || n.equals("-")){
                continue;
            }
            if(identityNames.contains(row) && col.equals(n)){
                //start m = row
                if(statesA.containsKey(n)){
                    if(!statesA.containsKey(row) || !statesA.get(n).equals(statesA.get(row))){
                        System.out.println("This is an invalid category");
                        System.out.println("===== COMPONENTS THAT CAUSED AN ERROR ====");
                        System.out.println(row);
                        System.out.println(statesA.get(n));
                        System.out.println(statesA.get(row));
                        continue;
                    }
                }
                else {
                    if(statesA.containsKey(row)){
                        statesA.put(n,statesA.get(row));
                        continue;
                    }
                    else {
                        rechecks.add(new int[]{i,j});
```



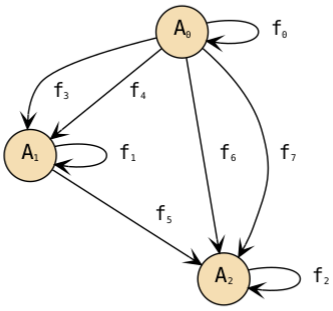
```
                        continue;
                    }
                    else {
                        rechecks.add(new int[]{i,j});
                        continue;
                    }

                }
            } else if (row.equals(n) && identityNames.contains(col)) {
                //end m = identity
                if(stateB.containsKey(n)){
                    if(!stateB.containsKey(col) || !stateB.get(n).equals(stateB.get(col))){
                        System.out.println("This is an invalid category ");
                        System.out.println("===== COMPONENTS THAT CAUSED AN ERROR ====");
                        System.out.println(col);
                        System.out.println(stateB.get(n));
                        System.out.println(stateB.get(col));
                        exit( status: 0);
                        continue;
                    }
                }
                else {
                    if(stateB.containsKey(col)){
                        stateB.put(n,stateB.get(col));
                        continue;
                    }
                    else {
                        rechecks.add(new int[]{i,j});
                        continue;
                    }
                }
            }
            else if (identityNames.contains(col) && identityNames.contains(row)){
                System.out.println("This is an invalid category");
                System.out.println("===== COMPONENTS THAT CAUSED AN ERROR ====");
                System.out.println(col);
                System.out.println(row);
                exit( status: 0);
            }
```

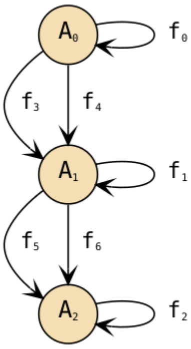


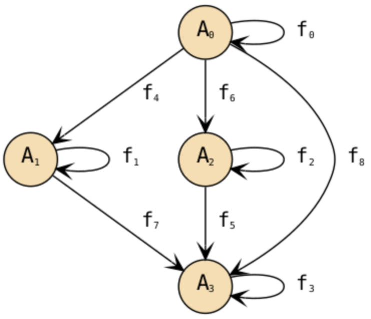
```
}
else {
    boolean ASet = false;
    boolean Bset = false;
    if (statesA.containsKey(row)){
        // m state A = row state A
        if(statesA.containsKey(n)){
            if(statesA.containsKey(row)){
                if(!statesA.get(n).equals(statesA.get(row))){
                    System.out.println("This is an invalid category");
                    System.out.println("===== COMPONENTS THAT CAUSED AN ERROR ====");
                    System.out.println(statesA.get(n));
                    System.out.println(statesA.get(row));
                    exit(status: 0);

                }
            }
            else {

                rechecks.add(new int[]{i,j});
                continue;
            }

        }else {
            if(statesA.containsKey(row)){
                statesA.put(n,statesA.get(row));
                continue;
            }
            else{
                rechecks.add(new int[]{i,j});
                continue;
            }

        }
        ASet = true;
    }
    if(stateB.containsKey(col)){
```

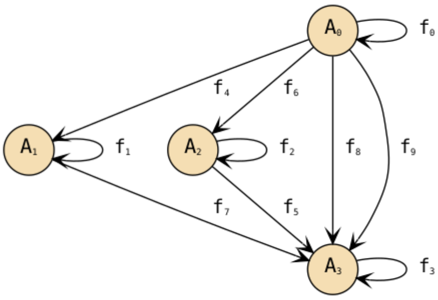


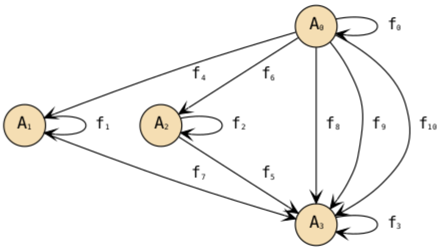
```
        ASet = true;
    }
    if(stateB.containsKey(col)){
        //m end = col end
        if(stateB.containsKey(n)){

            if(!stateB.get(n).equals(stateB.get(col))){
                System.out.println("This is an invalid category ");
                System.out.println("===== COMPONENTS THAT CAUSED AN ERROR ====");
                System.out.println(stateB.get(n));
                System.out.println(stateB.get(col));
                exit(status: 0);
                continue;
            }


        }else{
            if(stateB.containsKey(col)){
                stateB.put(n,stateB.get(col));
                continue;
            }
            else{
                rechecks.add(new int[]{i,j});
                continue;
            }

        }

        Bset = true;
    }
    if(!ASet && !Bset){
        //need to recheck later
        rechecks.add(new int[]{i,j});
    }

}
//check if any rechecks needed. otherwise can continue
```


A₀
f₀
f₃
f₄
A₁
f₁
f₆
f₇
f₅
A₂
f₂


A₀
f₀
f₃
f₄
A₁
f₁
f₅
f₆
A₂
f₂


A₀
f₀
f₄
f₆
A₁
f₁
A₂
f₂
f₈
f₇
f₅
A₃
f₃


A₀
f₀
f₄
f₆
A₁
f₁
A₂
f₂
f₈
f₉
f₇
f₅
A₃
f₃


A₀
f₀
f₄
f₆
A₁
f₁
A₂
f₂
f₈
f₉
f₁₀
f₇
f₅
A₃
f₃