Bachelor Thesis

Department of Computer Science
OST – University of Applied Sciences
Campus Rapperswil-Jona

Autumn Term 2025

Author	Fiona Pichler
Advisor	Prof. Dr. Farhad D. Mehta
External Co-Examiner	Sandy Maguire
Internal Co-Examiner	Cyrill Brunschwiler

Abstract

Text editing is a ubiquitous task. Its efficiency varies depending on the chosen editor and, more significantly, on a user's skill level. Modal editors offer strong benefits, like allowing users to keep their hands on the keyboard. However, their commands are often unintuitive and pose a steep learning curve for beginners. This thesis aims to make the initial experience with modal editing more pleasant and less frustrating.

To achieve this, a beginner-friendly modal editor based on Vim's keybindings is implemented. The editor is developed using algebra-driven design and Haskell. True to its name, Vim with training wheels supports new users through contextual help texts and colors, along with a tutorial that emphasizes understanding over rote memorization. It includes essential features such as editing and command undo functionality.

Management Summary

Background

Text editing is one of the most basic and common activities done by computer users. There are many programs available, but not all of them are as intuitive to a common user. While discussions go on over which is the most ergonomic, the fastest, the most efficient or simply the best way to edit text, it of course depends on the user and the use case.

Vim was released in 1991 as an improved clone of vi. Vim is a modal editor, where full functionality can only be used over the commandline. It is not intuitive to use for today's GUI-spoiled computer users. Nevertheless it is used by a large community. People appreciate it for being extensively customizable and open-source. The ability to work with modal editors helps users to navigate through a terminal and access systems and files by commandline. Many other programs like man, less and more use similar keybindings like Vim.

The entry into the world of modal editors is rather hard. There are only few tutorials and learning by doing can be especially frustrating, as every command has to be looked up and it needs time to get used to the workflows. After a dissuasive first experience many users turn back to the more familiar GUI-Editors. While experienced users have internalized the structure of combining commands, there is no simple guide for beginners and the structure needs to be discovered by oneself.

Objectives

The aim is to enhance the initial experience of using Vim and make it easier to start working with modal editors. This is done by providing a set of basic functions needed to use an editor as well as implementing a beginner-friendly editor and a tutorial focusing on the system of changing modes and combining commands rather than learning as many commands as possible.

Approach

Upon starting this project a lot of research on tutorials for Vim, vim-like editors and basic concepts of text editing was done. The usage and functionality of vim keybindings were analyzed to be able to imitate them. Relying on Vimtutor the basic functions a beginner needs to know could be defined. Since no base editor to build on, without a lot of overhead, could be found, an editor was implemented. Since Vim's commands can be combined like building blocks, they are well suited for using an algebraic approach. Algebra-Driven-Design and Haskell were used to implement the editor. The defined algebra for editing text was mapped to a TUI using vim keybindings. The concept to make the editor beginner friendly was to use more visual feedback for the user and help texts to remember commands relevant to the current mode. The visual feedback is implemented using colors for the modes and to show where the focus for keyboard input lays. The tutorial was kept simple, as the editor implements features to help learning by doing.

Results

The editor with the name "vim with training wheels" lives up to it's name, as it supports the user with additional feedback and a better overview of what is happening. It supports a basic set of functions to move the cursor, edit text, undo and redo an edit. The tutorial is a file to read and navigate through directly using vim with training wheels. The tutorial focuses on explaining the modes and combination of commands to help understand the big picture rather than only what different commands exist.

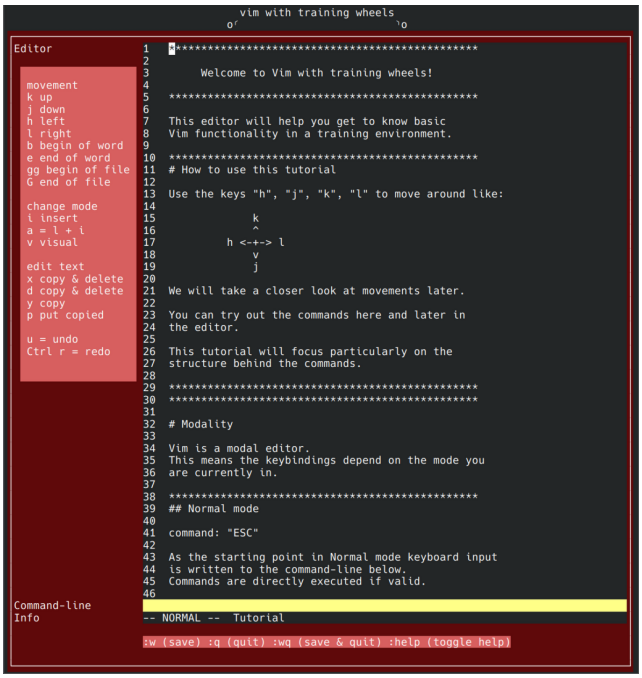

Figure 1: vim with training wheels

The repository for Vim with training wheels was already opened to external contributors for ZuriHac25. It got a lot of positive feedback and was used as a first project to get into Haskell.Brick and also Haskell in general. Since functionality is implemented very simple it can be easily understood and expanded.

Conclusion

Vim with training wheels is an editor looking like vim with a mask on, that helps with orientation. Users are guided with help texts and visual feedback which help building confidence. It can be used for teaching computer science, for a first understanding of modal editors, as well as for new Haskell programmers as an example project. Further work could include more functions for example search and replace or building a web app to make the editor more accessible to users.

Contents

Part I

Product Documentation

Chapter 1

Objectives

1.1 Problem Statement

Learning to use a new program Nowadays most programs have a GUI, used with a mouse or on a touchscreen one can navigate through menus select and move. Spoiled by GUIs it is not easy to switch to an environment where navigation is done with a keyboard. But the good-old terminal is still being used, as it needs less resources to run and provides a lot of functionality. Learning ones way around the command-line usually includes using a text editor. While there are endless discussions on which is the most efficient, the fastest and the most ergonomic editor it depends on the user and the use case. Especially useful is the ability to use modal editors like Vim, since some concepts, like movement or quitting are not only used in Vim but also for other command-line programs like man or less and they are preinstalled on many systems.

The entry into the world of modal editors is rather hard, as often when opening an editor one is greeted with no information on how to use or at least exit again. The few tutorials usually don't explain the system behind the editor and the structure of the commands and the modes. This makes learning very frustrating, time consuming and dissuasive. While experienced users have internalized the structure of combining commands, there is no simple guide for beginners and the structure needs to be discovered by oneself.

1.2 Goals

The initial experience of using modal editors should be more pleasant. This can build confidence to the command-line as well as shorten the time to learn. Helping beginners to understand the definition of a modal editor and the structure behind the combinations of commands, can help to learn the usage faster, as many people remember details by understanding the logic behind, rather than by memorizing. This is achieved by providing a Beginner-friendly editor implementing Vim keybindings.

1.3 Constraints

This thesis focuses on Vim, since the author was more familiar with it and also since man and less use similar keybindings. There was no further analysis between using Vim or another modal editor. There was no added value in analyzing this further since the goal is to provide a positive first experience and understanding of a modal editor to the user. Since this thesis also especially focuses on the structure behind Vim commands and the modality. It makes sense to use Algebra-Driven-Design and Haskell to model and implement the functionality. The solution should implement a TUI, as it should be a command-line editor. Only the basic functionality will be implemented, considering time and to not overwhelm the user. Every thing should be open source, in the spirit of Vim.

1.4 Deliverables

The deliverable should contain a basic modal text editor, as well as an Algebra for editing text and a tutorial to learn using the editor. Furthermore a set of basic commands to be able to work with Vim and detailed documentation on implementation. The software should contain tests and should be well structured, simple and clean.

Chapter 2

Background

This chapter provides background information for the thesis. As well as some known tutorials.

2.1 Modal Editor

A modal editor operates in different modes. Depending on the mode different actions can be done. For example can a pressing ”j” either lead to put ”j” in the text or moving the cursor around.

2.2 Vim

Vim [9] is a modal command-line editor, it is pre-installed on many systems. Vim, was released in 1991 and gained popularity, since it is open-source and customizable. The keybindings can be used in many other programs via a plugin or directly.

2.3 Algebra-Driven Design

Algebra-Driven Design [6] is an approach to design and model a problem. The focus lays on the proper level of abstraction. Laws are created to model a problem resulting in an algebra. Analyzing equations and combinations the model can be better understood and becomes more reliable.

2.4 Current State of the Art

This is a selection of tutorials to learn Vim commands.

2.4.1 Vimtutor

Vim Tutor [10] is the tutorial included when installing vim. It consists of a file to read through and edit examples while reading. The most basic commands are introduced. It focuses mostly on introducing commands and leaves out concepts of which commands can be combined. Since there is no repetition commands can be forgotten again very quickly. It is good to read through and get an overview of commands.

2.4.2 Vim Adventures

Vim Adventures [5] is a web browser game where the user navigates through using Vim commands. It helps using h, j, k ,l vor movement and also to have get more detailed knowledge of what a command does. For example, some obstacles can be overcome by using the functionality that the position in a line is remembered when moving down, even if the line one is currently at has fewer chars than the line before.

2.4.3 Vim Golf

Vim Golf [2] focuses on efficiency. The user gets an input to edit and a target output. The goal is to get to the target output using the least possible commands. Vim Golf lives from the community. Once a member one can upload own challenges. When a challenge is completed the user gets a ranking and can look at the commands of some users above themselves in the ranking. This helps to find more efficient ways.

Chapter 3

Requirements

The requirements are structured using FURPS.

3.1 Actors

The System under development (SUD) knows only one Actor, the user. The goal of the user is to edit text and learn in a tutorial the basic commands to edit text using basic vim commands.

3.2 Usecases

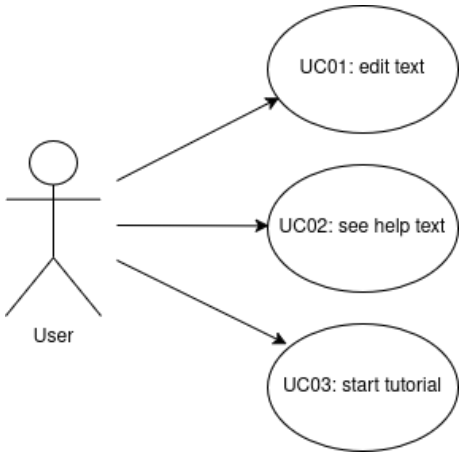

Figure 3.1: Use Case Diagram for the SUD

UC-ID	Title	Description	Priority
UC01	Editor	The SUD can be used as an editor for textfiles	1
UC02	Help / information	The SUD contains a Help page with information on the functions and the underlying algebra to help the user navigate and get deeper understanding of the commands.	1
UC03	Tutorial	The SUD has a tutorial to teach the user how to use the SUD.	2

3.3 Functional Requirements

3.3.1 Editor

FR-ID	Title	Description	Priority
FR01	Movement basic	The user can move the cursor using h (left), j (down), k (up), l (right)	1
FR02	Movement intermediate	The user can move the cursor using e (end of word), w (start of next word), $ (end of line), 0 (beginning of line)	1
FR03	Deleting chars	The user can copy-delete using x	1
FR04	Deleting lines	The user can copy-delete lines using dd	1
FR05	Deleting selections	The user can copy-delete using d + movement	1
FR06	insertion front	using i to enter insertion mode	1
FR07	insertion after	using a to enter insertion mode inserting after cursor	1
FR08	Visual mode	using v to enter visual mode	1
FR09	Normal mode	using Esc to enter normal mode	1
FR10	Saving file	using :w to write to file	1
FR11	quitting editor	using :q to quit editor	1

FR-ID	Title	Description	Priority
FR12	Movement advanced	The user can move the cursor using G, gg,	2
FR13	insertion end of line	using A to enter insertion mode after end of line	2
FR14	Copy	use y to copy selection	2
FR15	paste	use p to paste	2
FR16	undo	using u to undo last change	2
FR17	redo	using CTRL + r to redo the undone thing	2
FR18	Open new lines	Using o (open line below), O (open line above)	3
FR19	Replace one	The user can replace using r	3
FR20	Replace all	The user can replace using R	3
FR21	Change	The user can use c to delete specified and start insert mode	3

3.3.2 Help page

FR-ID	Title	Description	Priority
FR22	Cheatsheet	The help page contains a cheatsheet for the commands	1

3.3.3 Tutorial

FR-ID	Title	Description	Priority
FR23	Tutorial	The tutorial should teach the user the basic commands.	1

3.4 Non-Functional Requirements

NFR-ID	Title	Description	Priority
NFR01	Usability - Clear Instructions	The Editor should use clear language for help messages, tutorials and instructions	1
NFR02	Usability - Visual Feedback	The Editor should provide feedback when the user performs a task.	1
NFR03	Reliability - Stability	The SUD should run stable without issues	1
NFR04	Performance - Load-time	The SUD should start quickly (within 2-3 seconds)	1
NFR05	Performance - Input	The SUD should respond to input within 0.5 seconds.	1
NFR06	Supportability - Documentation	Code should be well-documented to facilitate future modifications	1

Chapter 4

Design

This chapter explains the architecture as well as the algebra for editing text.

4.1 Architecture

The architecture was kept simple. A Terminal user interface (TUI) is needed for interaction between the user and the editor, a file loader to interact with the local filesystem and a module to handle the TUI input and map it to algebra functions.

It was decided to implement a TUI to keep the commandline in focus and because Vim is a command-line editor. The Algebra implements the basic functionality to edit text. The functions can be combined which leads to more functionality.

The TUI has two pages. At startup opens a landing page, to welcome the user and let them decide between using the editor on a file or opening the tutorial. The file and also the tutorial are opened by changing to the editor page with the chosen input.

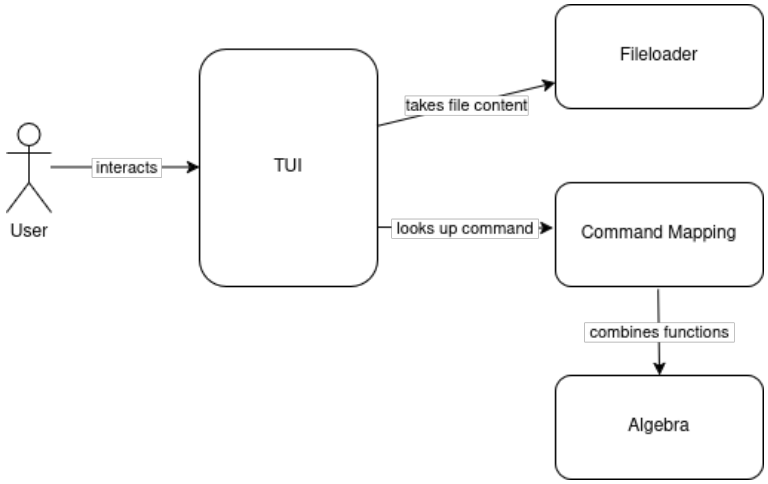

Figure 4.1: Basic Architecture of the Editor

4.2 Algebra

The algebra was defined following the book Algebra-Driven Design by Sandy Maguire[6]. The text to edit is modeled as a 3-tuple. Ideas on modeling the current position using pointers were dismissed for the simplicity of operating on lists in functional programming. Another idea on reversing the first list was dismissed also for simplicity, as only a thorough analysis would show wether this will be more efficient. Since for the analysis the most important functions need to be given, the initial concept was retained. The algebra was only defined for the task of editing the text buffer. an attempt was made to include the whole editor, but then discarded. The alegebra is left very basic. The basic functions are defined, but others left out, as every movement command like ”e” or ”b” need their own function which is a combination of simple movements and conditions. The attempt to include the whole editor provided a deep understanding of what should be implemented. The algebra can be found in the appendix.

4.3 Keymappings

Some keymappings were intentionally left out. For example the use of the arrow keys to move around. This should help users focus on the vim specific keybindings.

4.4 UI Design

The UI should be simple like Vim's but at the same time more approachable. The main idea was to keep Vim's design but add more information around it, providing help text and visual feedback. Working with colors helps to show which mode is used and were keyboard input is directed to. While in Vim beginners often forget about the mode and forget changing it, this problem can be solved with the colors to make the mode obvious.

4.5 Naming

The editor is called vim with training wheels. It was chosen after the first version of the editor was implemented. Since the mask with helping texts around the actual editor support the user like training wheels. The user gets more confident and a sense of achievement, using the non-intuitive commands. The name is very graphic, which led to the added ASCII art to give the title training wheels.

Chapter 5

Implementation

5.1 TUI

The TUI was implemented using brick [1] a TUI library for Haskell. Each window, is implemented as a separate app, that is called by the main function. This helps to keep the application states smaller and neat.

5.2 Algebra

While implementing the algebra some laws had to be changed, since the author had not thought about all the special cases like moving left when the first character is selected already. Changes on laws were applied but not all functions were added.

5.3 Modules

The program was divided into Haskell modules by task. For a better understanding a graph was generated using graphmod [3], a tool to automatically draw module dependency graphs. It is showed in figure 5.1

5.4 Testing Environment

The product was constantly tested using the following strategies. In general [4] was used for testing.

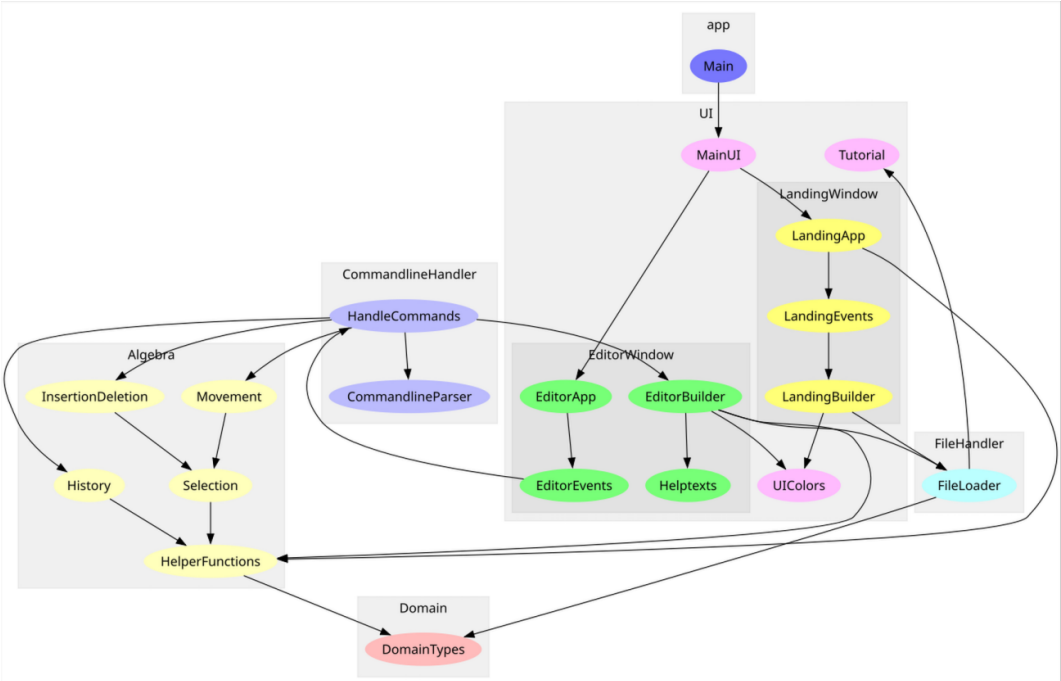

Figure 5.1: Dependency Graph of the Modules

5.4.1 Unit Tests

Each function of the algebra has at least one unit test. If some corner cases are known there are also unit tests for those. The tests were implemented simultaneously with the function.

5.4.2 Property Tests

Using quickspec [8] property tests were implemented following laws from the algebra. These tests were very helpful to discover corner cases and help to define the needed properties.

5.4.3 Manual Testing

Manual testing was done thoroughly by using the TUI and especially focusing on the visual parts. The program was tested by different users for feedback. These tests were not done systematically with instructions, to gather more corner cases especially for the landing page.

5.4.4 Limitations

Although it look at first like all functions can be tested, the implemented command handler which maps the algebra functions to the tui is directly editing the tui state. This should have been handled differently, to maintain testability for these combined functions. To implement another layer between the handler and the Algebra was considered an overhead and over complication for the program, but now those mappings rely on manual testing.

5.5 External Contribution

A copy of the project repository [7] was opened for external contribution during ZuriHac2025 [11] the biggest Haskell community event in the world. The program was heavily tested and some missing keymappings and parser functions were discovered. Some of them were fixed. Namely, not being able to use ”y” and ”d” without further input in visual mode. The efficiency of moving to the begin and the end of file, which was implemented very slowly before. Also a not used language extension was identified, a remnant from a function that is not used anymore. As the author is new to external contributors, some functions were accepted without tests included and others did not perfectly conform to the same system of what a module is responsible for and what should rather be handled in another module.

Chapter 6

Outcomes

This chapter describes the final product, it's limits and learnings.

6.1 Requirements

Most requirements are fulfilled, but not all of them.

6.1.1 Functional Requirements

For time reasons replacement mode was not implemented, as a priority three requirement and not being essential for a good user experience. All priority three functional requirements were dismissed to focus on the other implementations.

6.1.2 Non-Functional Requirements

NFR05 was not met at first when using "gg" and "G" to move to the beginning or to the end of a file. This was at first a desicion, to not further expand the algebra. Using existing functions resulted in very slow performance. This issue was solved by an external user and can not be oberseved any more in the vim-tw github repository.

6.2 Product

This bachelor thesis implemented a whole commandline editor.

6.2.1 Editor

Vim with training wheels is a fully functional text editor. Files can be opened, edited and saved. Commands can be undone and repeated using a count. The TUI supports the user with help texts that change according to the mode. The current mode is made obvious using colors. Also the focus for keyboard input is indicated by color. The editor scrolls automatically following the cursor, but only in vertical direction. Long lines can only be showed by extending the terminal window to the right. This works depending on the screen size. A text wrap is not implemented.

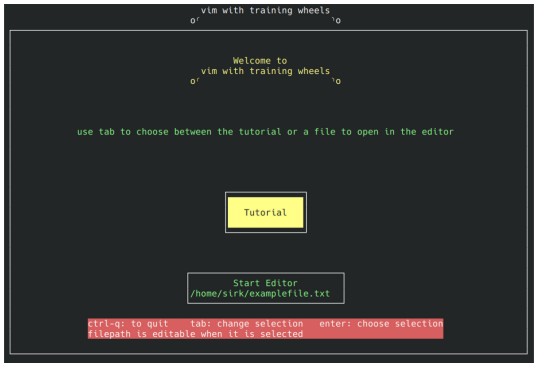

Figure 6.1: the Landing page of Vim with training wheels

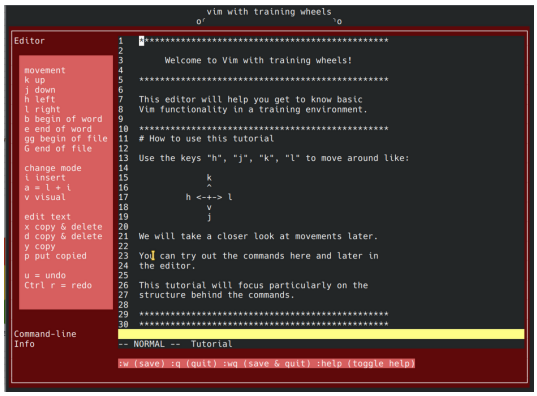

Figure 6.2: Vim with training wheels in normal mode

6.2.2 Tutorial

The tutorial is used by reading and navigating through a text using Vim with training wheels. Reading some explanations the user learns about modes, different types of com-

Figure 6.3: Vim with training wheels in insert mode

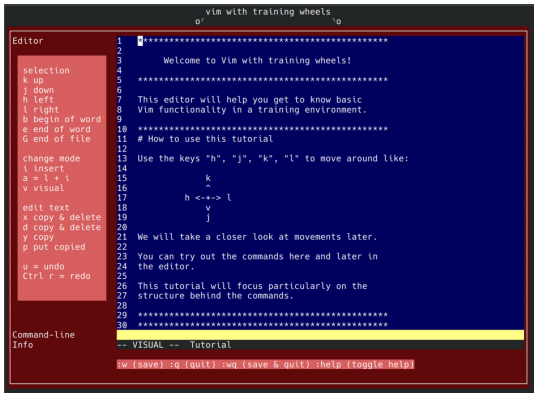

Figure 6.4: Vim with training wheels in visual mode

mands and the commands to use. Using ascii art the tutorial is designed more exciting. The tutorial text can be found in the appendix.

6.2.3 Haskell Project

Discovered at ZuriHac2025 [11] this project is not only a beginner friendly editor, but also a beginner friendly Haskell project to contribute to. The algebra is implemented very simply by using recursion and pattern matching. The TUI is only two applications of arranged widgets visualizing the state. There is still functionality that can easily be implemented by mostly copying existing code and making small changes. With Vim as the model there are many commands that could be added.

Chapter 7

Conclusion and Outlook

7.1 Conclusion

This thesis resulted in an educational program. Vim with training wheels can be used to learn and get accustomed to basic vim commands in a safe environment and also to practice Haskell skills implementing small functionalities. Using algebra-driven design the program's design was elaborated thoroughly, which led to an efficient implementation. The algebra could be more defined and also including the combination of the functions. Since this was the authors first project using algebra-driven design the final algebra resulted in not being over all functions. Although the design of using the 3-tuple to represent the buffer might not be best for every use case, it led to the simple almost pure use of Haskell.

7.2 Outlook

Since the project is already open for contribution and presented to a number of programmers it can grow from here on. It could be used in schools for student's first commandline experience. The algebra should be published with the program, to help understand the design and keeping it simple. An additional tutorial similar to vim golf would be nice, since to getting used to an editor it needs to really be used.

Bibliography

[1] *brick: A declarative terminal user interface library.* URL: `https://hackage.haskell.org/package/brick`. (accessed: 09.09.2025).

[2] Mandy Neumeyer Frank Hofmann. *Vimgolf – Vim mit wenigen Tastendrücken meistern.* URL: `https://www.linux-community.de/ausgaben/linuxuser/2018/09/unter-par/`. (accessed: 09.09.2025).

[3] *graphmod: Present the module dependencies of a program as a dot graph.* URL: `https://hackage.haskell.org/package/graphmod`. (accessed: 09.09.2025).

[4] *Hspec: A Testing Framework for Haskell.* URL: `https://hspec.github.io/`. (accessed: 09.09.2025).

[5] Doron Linder. *Vim Adventures.* URL: `https://vim-adventures.com/`. (accessed: 09.09.2025).

[6] Sandy Maguire. *Algebra-Driven Design.* Leanpub.com, 2020.

[7] Fiona Pichler. *vim-tw repository.* URL: `censored`. (accessed: 09.09.2025).

[8] *quickspec: Equational laws for free!* URL: `https://hackage.haskell.org/package/quickspec`. (accessed: 09.09.2025).

[9] *Vim - the ubiquitous text editor.* URL: `https://www.vim.org/`. (accessed: 09.09.2025).

[10] *Vim Tutor.* URL: `https://vimschool.netlify.app/introduction/vimtutor/`. (accessed: 09.09.2025).

[11] *zurihac2025.* URL: `https://zfoh.ch/zurihac2025/`. (accessed: 09.09.2025).

Part II

Appendix

vim Commands

This is an overview what commands do an with which other command they can be built.

Normal Mode

Movement Direction

Name	keystroke	Definition
Left	h	
Right	l	
Up	k	(count h until [\n] + h until [\n] + [count] l or until [\n])
Down	j	(count h until [\n] + l until [\n] + [count] l or until [\n])
Move to End of File	G	down and right until eof (j + l) * 1000
Move to start of file	gg	up and left until beginning of file (k + h) * 1000
move to certain number	[linenumber] G	(gg + [linenumber]j)

Movement Stepsize

Name	keystroke	Definition
End of Word	e	(l until [space])
Before next Word	w	{e + l}
End of Line	$	(l until [\n] + h)
Start of Line	0	(h until [\n] + l)

Text Manipulation

Name	keystroke	Definition
Delete char	x	copy selection + delete selection
Delete Line	dd	(0 + select $ + x)
Delete word	dw	select w + x
Delete	d $number $motion	select $number $motion + x
Replacement	R	start replace mode
replace one	r[char]	x + insert [char]
Put insert last deleted	p	insert [char]

Name	keystroke	Definition
replace	r	x + insert mode [char] + Esc
change (deletes specified and starts insert mode)	c $motion	d [motion] + insert mode
copy	y	add selection
paste	p	insert selection

Undo, Redo, LineUndo

https://vimhelp.org/undo.txt.html#undo-redo

Name	keystroke	Definition
Undo last thing	u	changes during insert mode are one command
Undo Line Manipulation	U	undo also undos on the whole line
redo	Ctrl + r	undo an undo

Insert Mode

Name	keystroke	Definition
Insert	i	
Append	a	
Append at End of Line	A	
open a new line below	o	
open a new line above	O	

Command Mode

Name	keystroke	Definition
search	/[phrase to search]	
find matching parenthesis	%	
substitute	`:s/old/new/g` (there are many more options)	
save file	:w	
quit vim	:q!	

Algebra

Datatypes

```
newtype AfterSelection :: AfterSelection [Char]

newtype BeforeSelection :: [Char]

newtype CurrentSelection :: [Char]

newtype CurrentBuffer :: (BeforeSelection, CurrentSelection,
AfterSelection)
```

Helper Functions

```
checkSelection :: CurrentBuffer -> [Char] -> bool
```

Example:
`([a,b],[c],[d,e]) , [c] -> True`

```
safehead :: [Char] -> [Char]
```

Example:
`([a,b,c]) -> [a]`

```
safetail :: [Char] -> [Char]
```

Example:
`([a,b,c]) -> [b,c]`

```
safelast :: [Char] -> [Char]
```

Example:
`([a,b,c]) -> [c]`

```
safeinit :: [Char] -> [Char]
```

Example:
`([a,b,c]) -> [a,b]`

```
getSelectionSize :: CurrentBuffer -> Natural
```

Example:
`([a,b],[c,d],[e]) -> 2`

```
getInlinePos :: CurrentBuffer -> Natural
```

Example:
`[a,b,\n,c,d][e][f,g,e] -> 2 (start index at 0)`

```
repeatTimes :: Natural -> a -> (a -> CurrentBuffer -> CurrentBuffer) ->
(CurrentBuffer -> CurrentBuffer)
```

Example:
->

The function repeatTimes takes a Natural, a variable and a function that take as input the same variable as well as a CurrenBuffer. The function is then repeated, taking the cariable as input. Natural is the number of times the function is repeated. This function helps to handle repeated commands.

```
discardArgument :: a -> CurrentBuffer -> CurrentBuffer
```

Example:
`function a -> function`

The function discardArgument is needed to use repeatTimes on functions that don't take an input.

Selection

```
selectRight :: CurrentBuffer -> CurrentBuffer
```

Example:
`[a,b],[c],[d,e] -> [a,b],[c,d],[e]`

```
selectRightInline :: CurrentBuffer -> CurrentBuffer
```

Example:
`[a,b],[c],[\n,d,e] -> [a,b],[c],[\n,d,e]`

```
unselectRight :: CurrentBuffer -> CurrentBuffer
```

Example:
`[a,b],[c],[d,e] -> [a,b],[],[c,d,e]`

Law size of selectRight

```
∀((a,b,c) :: CurrentBuffer).
(c /= [] ) =>
getSelectionSize (selectRight (a,b,c)) = getSelectionSize (a,b,c) + 1
```

Law selectRight/unselectRight

```
∀((a,b,c) :: CurrentBuffer).
(c /= [] ) =>
selectRight . unselectRight (a,b,c) = (a,b,c)
```

Law size of selectRightInline

```
∀((a,b,c) :: CurrentBuffer).
(c /= [] && safehead c /= "\n") =>
getSelectionSize (selectRightInline (a,b,c)) = getSelectionSize (a,b,c) +
1
```

Law selectRightInline/unselectRight

```
∀((a,b,c) :: CurrentBuffer).
(a /= [] && safehead c /= "\n") =>
selectRightInline . unselectRight (a,b,c) = (a,b,c)
```

Law size of unselectRight

```
∀((a,b,c) :: CurrentBuffer).
(length b > 1 ) =>
getSelectionSize( unselectRight (a,b,c)) = getSelectionSize (a,b,c) - 1
```

```
selectLeft :: CurrentBuffer -> CurrentBuffer
```

Example:
`[a,b],[c],[d,e] -> [a],[b,c],[d,e]`

```
selectLeftInline :: CurrentBuffer -> CurrentBuffer
```

Example:
`[a,b,\n],[c],[e] -> [a,b,\n],[c],[e]`

```
unselectLeft :: CurrentBuffer -> CurrentBuffer
```

Example:
`[a,b],[c],[d,e] -> [a,b,c],[],[d,e]`

Law size of selectLeft

```
∀((a,b,c) :: CurrentBuffer).
(a /= [] ) =>
getSelectionSize( selectLeft (a,b,c)) = getSelectionSize (a,b,c) + 1
```

Law selectLeft/unselectLeft

```
∀((a,b,c) :: CurrentBuffer).
(a /= [] ) =>
selectLeft . unselectLeft (a,b,c) = f
```

Law size of selectLeftInline

```
∀((a,b,c) :: CurrentBuffer).
(a /= [] && safelast a /= "\n") =>
getSelectionSize( selectLeftInline (a,b,c)) = getSelectionSize (a,b,c) + 1
```

Law selectLeftInline/unselectLeft

```
∀((a,b,c) :: CurrentBuffer).
(a /= [] && safelast a /= "\n") =>
selectLeftInline . unselectLeft (a,b,c) = f
```

Law size of unselectLeft

```
∀((a,b,c) :: CurrentBuffer).
(length b > 1 ) =>
getSelectionSize( unselectLeft (a,b,c)) = getSelectionSize (a,b,c) - 1
```

Moving operators

```
moveRight :: CurrentBuffer -> CurrentBuffer
```

Example:
`[a,b],[c],[d,e]` -> `[a,b,c],[d],[e]`

```
moveLeft :: CurrentBuffer -> CurrentBuffer
```

Example:
`[a,b],[c],[d,e]` -> `[a],[b],[c,d,e]`

Law inverse/moveRight/moveLeft

```
∀((a,b,c) :: CurrentBuffer).
(c /= [] && safelast c /= "\n") =>
moveRight . moveLeft f = f
```

Law associativity/moveRight/moveLeft

```
∀((a,b,c) :: CurrentBuffer).
(a /= [] && safelast a /= "\n" && safehead c /= "\n") =>
moveRight . moveLeft (a,b,c) = moveLeft . moveRight (a,b,c)
```

Law moveRight/select

```
∀(f :: CurrentBuffer).
moveRight f = unselectLeft . selectRight f
```

Law moveLeft/select

```
∀(f :: CurrentBuffer).
moveLeft f = unselectRight . selectLeft f
```

```
moveLeftUntilNewline :: [Char] -> CurrentBuffer -> CurrentBuffer
```

Example:
->

```
moveRightUntilNewline :: [Char] -> CurrentBuffer -> CurrentBuffer
```

Example:
->

```
moveUp :: CurrentBuffer -> CurrentBuffer
```

Example:
[a,b,\n,c],[d],[e,f] -> [a][b][\n,c,d,e,f]

Law moveUp/definition

```
moveUp f = (repeatTimes moveLeft (countLinePos f)) (repeatTimes
moveLeftUntil 2) [\n] f
```

```
moveDown :: CurrentBuffer -> CurrentBuffer
```

Example:
[a,b,c],[d],[e,\n,f,g] -> [a,b,c,d,e,\n,f][g][]

Law moveDown/definition

```
moveDown f = (repeatTimes moveLeft (countLinePos f)) moveRightUntil [\n] f
```

Insertion & Deletion

```
insertOnebeforeSelection :: CurrentBuffer -> Char -> CurrentBuffer
```

Example:
`[a,b],[c],[d,e], x -> [a,b],[x],[c,d,e]`

```
insertManybeforeSelection :: CurrentBuffer -> [Char] -> CurrentBuffer
```

Example:
`[a,b],[c],[d,e], [x,y,z] -> [a,b,x,y,z],[c],[d,e]`

```
deleteSelection :: CurrentBuffer -> CurrentBuffer
```

Example:
`[a,b],[c],[d,e] -> [a,b],[d],[e]`

```
deleteOneBeforeSelection :: CurrentBuffer -> CurrentBuffer
```

Example:
`[a,b],[c],[d,e] -> [a],[c],[d,e]`

Tutorial Text

```
**********************************************

     Welcome to Vim with training wheels!

**********************************************
This editor will help you get to know basic
Vim functionality in a training environment.
**********************************************
**********************************************

# How to use this tutorial

Use the keys "h", "j", "k", "l" to move around like:

             k
             ^
         h <-+-> l
             v
             j

We will take a closer look at movements later.

You can try out the commands here and later in
the editor.

This tutorial will focus particularly on the
structure behind the commands.

**********************************************
**********************************************

# Modality

Vim is a modal editor.
This means the keybindings depend on the mode you
are currently in.
```

```
**********************************************
## Normal mode

command: "ESC"

As the starting point in Normal mode keyboard input
is written to the command-line below.
Commands are directly executed if valid.

The training wheels help you see the command-line.

While in vim an "i" will always start insert mode,
With training wheels, the command needs to be the
only string on the command-line to be valid.

**********************************************
## Insert mode

command: "i"

For writing new text Insert mode is needed.
Focus lies on the text, and it can be directly
edited. There are more commands to start Insert mode
command: "a" start Insert mode after the cursor
command: "A" start Insert mode at the end of the line.

Use "Esc" to get back to Normal mode.

**********************************************
## Visual mode

command: "v"

Some commands require a selection as input. In
Visual mode the selections are visual. For example, to
be sure you delete the next two words in
the selection can be made in Visual mode
to then apply a command.

Selections use the same keybindings as movements.

Use "Esc" to clear the command-line and get back
to Normal mode.

**********************************************
## Command mode

command: ":"
```

Exit the editor, save the file, or toggle help, all
need Command mode.

Commands will only be executed after using "Enter".

In Vim Command mode is mostly invisible. The wheels
help you see it. There are also other commands in
Vim not starting with ":" but the training wheels
don't support those commands.

Use "Esc" to clear the command-line and get back
to Normal mode

****
****

Direct Commands

Some commands don't need further input. But they
can be repeated.

```
 count   +   command
```

For example use "7j" to execute "j" 7 times.
Most commands can be used more efficiently when
combined with a count.

****
****

Operator Commands

Some commands need further input like a motion.

```
 count   +   command   +   motion
```

For example "y", yank, which copies the selection
to a clipboard.

****
****

That's it for the main concept.

1. Keybindings depend on the mode you are in.

```
2. Commands can be used like building blocks for
   more efficient usage.

Let's move on to the different commands.
Many of them can be remembered by the first
letter of a verb describing their action.
**********************************************
**********************************************

# Command-line commands

Let's first look at how you can exit the editor
and safe the file.

":w" -> write to file which means saving it

":q" -> quit the program

":wq" -> write to file then quit the program

**********************************************
**********************************************

# Movements or Selection

             k
             ^
         h <-+-> l
             v
             j

As shown before the above keys are used to move
around, but we can be even more efficient:

"b" -> move to the beginning of the word
       instead of "h" moving left by character
       we can move left by word.

"e" -> move to end of word.
       move to the right by word

"0" -> move to the beginning of the line

"$" -> move to the end of the line

Movements can also be used as motions for
commands and as selections in visual mode.
```

```
**********************************************
**********************************************

# Meet your new best friend

Before you edit this tutorial's text by mistake,
take a look at a vim-beginners best friend


          ****   ****
       ***   ** **   ***
      **       *       **
      *                 *
      **      undo     **
        **           **
          ***     ***
             ** **
               *

"u" -> undo last edit
"Ctrl-r" -> redo the undo

Whatever happens, undo is there to take you back
to the good old times.

**********************************************
**********************************************

# Edit text

"x" -> delete the current character and copy it
       to the clipboard
"p" -> put the clipboard contents after the cursor

"d" + motion -> delete the selected and copy it
                to the clipboard
"dd" -> delete the whole line

"y" + motion -> yank the selected, which means
                copy it to the clipboard

"yy" -> yank the whole line

**********************************************
**********************************************

Unfortunately, the training wheels don't support
all Vim commands and functionality, but hopefully
they helped you get some comfort with using modes
```

and only the keyboard for editing text.

****
****