ELMSR: A STRUCTURE EDITOR FOR ELM

ELMSR: A STRUCTURE EDITOR FOR ELM

By NARGES OSMANI, MSc

A Thesis Submitted to the School of Graduate Studies in Partial Fulfillment of the Requirements for the Degree Master of Computer Science

McMaster University

MASTER OF COMPUTER SCIENCE (2024)

Hamilton, Ontario, Canada (Computing and Software)

TITLE:	ElmSr: A Structure Editor for Elm
AUTHOR:	Narges Osmani MSc (Digital Humanities), University of Pisa, Pisa, Italy
SUPERVISOR:	Dr. Christopher K. Anand
NUMBER OF PAGES:	xii, 104

Abstract

Structure editors have been available for many decades, and for multiple programming languages. Historically, they have been recommended for teaching new programmers. Currently, they are recommended by advocates of Model Driven Development. However, they are not widely used, except for the special case of graphical structure editors commonly referred to as "block-based editors" such as Scratch. Although structure editors were first introduced for procedural languages, and they could be used for any type of language, current structure editors target object-oriented languages, almost exclusively, and build in many assumptions related to object orientation. The notable exception, Hazel, targets a functional language, exploits the strong typing typical of functional languages and emphasizes the use of typed holes.

This thesis introduces ElmSr, a structure editor developed for teaching the Elm programming language to novice programmers. As with most structure editors, ElmSr allows the developer to directly edit the Abstract Syntax Tree, without the intermediary of a compiler. As with Hazel, ElmSr's AST is typed, and transformations preserve types. Also typical of tree editors written in functional languages, ElmSr uses a zipper data structure to encode both the tree and a cursor position, making for efficient tree edits. Like other structure editors, ElmSr is designed to make common tasks simple and efficient. In our case, common tasks match the steps students are taught in the

"Algebraic Thinking" curriculum developed at McMaster University. Some steps are common to almost all programming, such as arithmetic expression entry and modification (which has been previously identified as a weak point for structure editors). Other steps, like definition integration, and support for function calls and control structures are specific to our curriculum. This aspect of usability was evaluated by comparing the number of keystrokes necessary for a benchmark task, using ElmSr, using VS Code following our structured development approach, and using VS Code solely for text entry. ElmSr was much more efficient than VS Code for structured editing, and still more efficient than linear code entry.

Acknowledgements

I extend my heartfelt appreciation to my supervisor, Dr. Christopher Kumar Anand, for their constant support, guidance, and encouragement throughout my master's journey. Your expertise and dedication have been invaluable to me.

A special thank you to my friends Sheida and Tina for their friendship and for always being there when I needed them.

I also want to extend my gratitude to Lucas and Emma for being there as a team during the initial stages of my thesis.

Finally, thank you to my parents for believing in me and supporting me every step of the way.

Last but not least, I want to express my deepest gratitude to my husband, Omid, for always being by my side and supporting me through this journey. Your love and encouragement have meant the world to me.

This thesis is a reflection of the love, support, and encouragement of each of you. I am incredibly grateful for your presence in my life.

Table of Contents

List of Figures

List of Tables

Definitions and Abbreviations

Definitions

Abstract Syntax Tree	A tree representation of the syntactic structure of source code, where each node represents a construct in the code.
Algebraic Thinking	An approach to teaching computer science and programming emphasizing the connections to algebra
Projectional Editor	Synonym for Structure Editor
Zipper	A data structure designed to simplify and make efficient the editing of tree data structures.

Abbreviations

AST	Abstract Syntax Tree

OO	Object-Oriented, adjectival phrase describing programming languages, etc.
DSL	Domain-Specific Language
CFG	Context-Free Grammar
CPS	Cornell Program Synthesizer
IPE	Incremental Programming Environment
RQ	Research Question
MPS	Meta Programming System

Chapter 1

Introduction

A structure or projectional editor is an editor in which an abstract syntax tree (AST) is edited without the need for a compiler. The presentation of the AST may be in the form of a text document, and visually may resemble the type of text editor which most commonly sits at the head of the toolchain, followed by the compiler, assembler, linker and run-time system. But since the AST is the object being edited, it is possible to avoid syntax and type errors and only present the user with editing actions which preserve the AST structure. While this has obvious advantages, structure editors have not replaced text editors because most prorgrammers still prefer the flexibility text editors offer, including the ability to transform a working program through malformed stages. Whether this is simply a matter of what they are used to, or whether it is fundamentally more efficient to take short-cuts through the space of ill-formed programs is an open question.

1.1 Purpose

This thesis explores the design of a structure editor, also called a projectional editor, for the Elm programming language. Elm is a compact language which is close to a common core for languages in the ML-family, of which the most well-known is Haskell. Most recent research into structure editors is built around the assumption, often unstated, that the target language will be object oriented (OO). Since OO programming represents the vast majority of programming in many commercial areas from enterprise computing, with, e.g., Java, web computing, with, e.g., Javascript, and data science and machine learning, with, e.g., Python, it is easy to make this assumption. But programs in functional programming languages have different structures compared to those in OO languages, and functional programmers have different expectations of their languages, so the advantages and disadvantages of structure editors may be different for functional languages. In particular, many functional programmers are strong advocates of very strong typing, and many programmers are attracted to the shared goal of eliminating as many errors as possible through the use of strong typing, see [25], and it is hypothesized that functional programmers have different priorities in program construction tasks [29]. Using strong typing to reduce the number of errors has the side-effect of reducing the overall number of expressible programs. This is likely to have deep and subtle impacts on structure editing.

This thesis will lay a foundation for exploring these differences, but identifying and quantifying such differences is a long-term research program, beyond the scope of this thesis. Instead, our focus is on creating interface conventions for such structure edtiors. As a proxy for usability at this stage, we consider the subset of program editing actions we teach to novices as part of our Algebraic Thinking curriculum, including those required to enter and modify algebraic expressions, because previous research by Voelter et al. [45] has identified this as a barrier to adoption of structure editors.

1.2 Research Questions

In the field of structure editing, there are key questions about usability and efficiency that require investigation. Particularly, the handling of algebraic expressions and the speed of code entry present significant challenges compared to traditional text editing. Consequently, we present two Research Questions (RQs):

RQ1. How can we optimize the usability of structure editors to provide a more intuitive and familiar experience, specifically when dealing with algebraic expressions utilizing infix operations?

This research question aims to explore the inherent challenges of structure editors in handling algebraic expressions, especially infix operations. It looks into the paradigm shift required by users who are accustomed to text editors when entering such expressions in a structure editor. The primary goal is to discover ways to enhance the user experience by making the editing process more intuitive and natural. Potential sub-questions could be:

- How does the input method for infix operations in a structure editor currently affect user experience?
- What features can be integrated into structure editors to make infix operations more user-friendly and familiar to those used to text editors?

- Can a hybrid approach, where typed algebraic expressions are parsed to build the AST, improve the user experience without compromising the integrity and advantages of a structure editor?

RQ2. Is it possible for a structure editor to achieve similar or superior efficiency in program construction as compared to a traditional text editor, despite inherent structural differences, and could the unique functionalities of structure editors pave the way for more efficient program entry methods?

This research question intends to investigate the efficiency trade-offs between text and structure editors. The question considers the structural differences that may act as barriers to achieving efficiency in structure editors. However, it also opens up the possibility that unique functionalities of structure editors could counterbalance these limitations or even lead to innovative and more efficient programming practices. Potential sub-questions could include:

- How does the program entry speed in structure editors compare to text editors, considering various complexity levels of the program?
- What structural constraints in structure editors could hinder efficiency in program entry, and how could these be mitigated?
- What unique features of structure editors could be exploited to enhance efficiency in program entry?

Through this study, we aim to uncover insights that could help to refine structure editing for both novice and experienced programmers, especially functional programmers.

1.3 Contributions

In this thesis, we have developed a structure editor for the Elm programming language, implemented in Elm itself. This thesis includes a thorough documentation of the design decisions to aid future developers. Specifically, we have reviewed the core concepts essential to this design, namely ASTs and Zippers.

Furthermore, we have provided an extensive review of the literature on structure editors. This includes a comprehensive table categorizing both current and historically significant structure editors, which serves as a quick reference for understanding the evolution and variety of editors in this field.

Additionally, we have developed simple benchmarks to assess the performance of various editors. These benchmarks, that are designed to be applicable to any functional programming language, have been applied to evaluate the performance of our structure editor in comparison with a conventional text editor.

1.4 Structure of Thesis

The rest of this thesis is organized as follows: Chapter 2 provides a literature review, discussing various methodologies for creating structure editors and different types of such editors. Chapter 3 introduces preliminary concepts essential for understanding the techincal details discussed in later chapters. Chapter 4, the core of the thesis, presents ElmSr, our structure editor for the Elm language, detailing design choices and capabilities. Chapter 5 discusses the tests conducted to evaluate ElmSr against a competitor text editor. Finally, Chapter 6 concludes the thesis, reviews research questions, and proposes future research directions.

Chapter 2

Literature Review

In this chapter, we explore the methods for constructing structure editors and the various categories they fall into, setting the stage for a discussion that positions our work within these categories.

2.1 Building Structure Editors

To the best of our knowledge, there are four broad ways of creating structure editors employed in the literature. In this section, we discuss these methods since they are an important factor in classifying related works in the field of structure editors.

2.1.1 Generating Editors from Language Grammars

In this approach, the focus is specifically on generating structure editors based on the grammar of the target programming language. Particularly, the formal description of the language (such as Context-Free Grammars (CFGs)[40]) is used to automatically produce an editor that offers the syntax and structure of that language visually, often as blocks or program units. Such units can be utilized by the language user to write programs. The emphasis in this approach is on the automatic generation of the editor interface from a given language specification.

2.1.2 Using Language Workbenches

While the emphasis of the previous approach is on the generation of the editor interface (syntax), language workbenches offer a broader range of capabilities for language design and implementation, and provide tools for defining syntax, semantics, type systems, transformations, and often editor interfaces for Domain-Specific Languages (DSLs). Therefore, they are suitable for more complex language engineering tasks beyond the scope of editor generation alone. Within the scope of this thesis, we are particularly interested in language workbenches that are structured (such as the Meta Programming System (MPS) [4],) meaning that they provide functionalities to edit the AST of the program directly.

2.1.3 Adapting Existing Text Editors

This method involves modifying or extending existing text editors to incorporate some features of structure editors. The process usually involves writing extensions or plugins that change the editor's behavior [3]. These extensions can provide a structure editing interface, where programming constructs are manipulated and traversed structurally rather than, or in addition to as text. For example, the extended editor might provide jumping between components of a case expression. Parsers are extensively used in such editors, for either parsing the entire program—despite the potential presence of incomplete expressions—or selectively bypassing these incomplete segments. The parser then generates an AST with which the editor can interact in a structured manner.

This approach is particularly interesting where the features, flexibility, and extensibility of a specific editor like Emacs [2] are desired, but adding structure editing features is also needed for specific use-cases. However, this approach might not offer all the benefits of structure editors, especially error prevention while entering expressions.

2.1.4 Development from Scratch

Apart from the above three methods, some structure editors are developed from scratch, mainly due to the fact that some highly customized features are needed. This approach, although it provides the greatest flexibility, needs much more effort compared to the other approaches and generally involves implementing the editor's functionality, user interface, and logic. Moreover, the choice of the technology stack and the software architecture can have significant impact on the performance and availability of the editor.

2.2 Types of Structure Editors

2.2.1 Strict Structure Editors

The initial class of structure editors was marked by a distinctive approach to programming interfaces, characterized by an emphasis on direct manipulation of programming constructs and the enforcement of structured program entry methodologies. The primary goal of many of these early structure editors was centered on the technology itself, rather than on user-friendliness. That is, the focus was the implementation of mechanisms to insert various programming constructs like loops and conditional statements as templates. These mechanisms included both menu selection interfaces [20] and keyboard shortcuts [19] — pressing 'f' for inserting a "for loop" template, or '=' for an assignment operation, for instance. These templates often included **fillable holes** for later completion by the user.

In many of these editors, the keyboard's role was usually two-fold. It was employed for typing literals, such as strings and numbers, and for activating structural commands via shortcuts. However, the choice and implementation of these shortcuts sometimes proved to be more complex than intuitive, often increasing the complexity and requiring users to remember so many shortcuts specific to such editors only. Also, choosing from many such shortcuts may lead to ambiguity.

A key aspect of these early structure editors was the requirement for users to have an awareness of the AST's tree-like structure, particularly in expression entry. For example, entering an expression like `5 * 2` required inserting the multiplication operator first, followed by the operands. This method needed an unnecessary engagement with the program's syntax tree even for simple structures. In an attempt to ease this process, some editors adopted a non-structured approach for expression entry or modification, accepting expressions in a text-like format. However, this approach often relied on parsers to transform textual input into an AST, which could dilute the benefits of structure editing (such as error prevention) at the level of expressions.

One of the early examples of structure editors was Emily [20] designed by William

J Hansen. In Emily, a program code started as a single non-terminal symbol, and developers constructed their programs by selecting syntax rules from a menu (using a light pen) to replace non-terminals. This approach maintained the tree structure of the text, enabling manipulation according to structural units. Emily's design choices, particularly the menu-driven construction of programs, were early explorations into how syntax-directed editing could facilitate program creation.

MENTOR [14, 15] is another influential early example in the domain of structure editors. Initially developed for Pascal, MENTOR integrated concepts of syntax-directed editing and structure-oriented manipulation, laying the groundwork for the development of language workbenches. Program entry in MENTOR involved incremental parsing of the program text into a tree structure that could then be manipulated in a structured manner. At the core of MENTOR was MENTOL, a tree manipulation language specifically designed for interacting with ASTs that allowed MENTOR to perform operations directly on the AST. The system evolved over time, with notable extensions like MENTOR-Ada [16], which inherited MENTOR's capabilities and applied them to the Ada programming language. These extensions continued to focus on representing programs as trees, maintaining the core principle of structure-oriented editing.

The Cornell Program Synthesizer (CPS) [41], implemented for a subset of PL/I[1] in the early 1980s, was designed to facilitate the construction and execution of small programs, particularly on microcomputers. This tool utilized menu-based interactions for entering language constructs. However, recognizing the difficulty of structured program entry, it was designed to combine structure editing with text-based input. At the expression level, programmers entered code as plain text in a *textual mode.*

[1]Programming Language One, see: https://en.wikipedia.org/wiki/PL/I

Upon exiting this mode, the text was parsed back into structured form, asking users to address any syntax errors before reverting to structure editing.

The Incremental Programming Environment (IPE) [30], introduced in 1981, is a part of a more general environment, Gandalf [32]. It notably enabled the definition of multiple language notations and the capability to partially hide parts of the AST. IPE applied structure editing at all levels of program construction, including expressions. Similar to our ElmSr editor, the environment visually highlighted the current position within the program tree, aiding programmers in navigating and understanding the program's structure. The syntax-directed editor in IPE was built from the language specifications which facilitated direct manipulation of program structure. However, other development tools such as debugger, translator and the linker/loader, operated invisible to the user and were automatically invoked to maintain consistency between the tree representation and the machine representation. Furthermore, IPE's compiler utilized the internal tree representation for code generation, avoiding the parsing from scratch (hence the 'incremental' in its name) and syntax analysis.

GNOME (Gandalf NOvice prograMming Environment) [17], introduced in 1984 as another sub-project of the Gandalf project, stands out particularly in the context of educational programming environments. The primary goal was to translate the experience gained from structure editing research into a practical tool for teaching programming to undergraduates. The structure editor format reduced the need for constant reference to language manuals and allowed students to focus more on computer science concepts. The GNOME project was part of a family of ALOE (A Language Oriented Editor) editors which supported the construction and manipulation of syntactically correct tree structures. Similar to other ALOE editors, GNOME was built using an editor generator tool called AloeGen from the grammar of the target language.

Finally, GANDALF [19, 32] encompasses a full set of advantages seen in earlier sub-systems, including the ability to automatically generate editors from language specifications. GANDALF introduced its own language designed specifically for specifying, in the language specification, interactions and views within the editor. It also offered programmers a uniform method for entering keyboard commands using Emacs-style keyboard shortcuts. Additionally, GANDALF provided tools for managing projects that involved multiple programmers and offered integrated version control functionalities.

2.2.2 Keyboard-Driven Structure Editors

In this section, we explore a class of structure editors designed to provide users with efficient means of manipulating code structures which is similar to text editors. Unlike strict structure editors, which often impose rigid constraints on the user's interactions, editors in this class prioritize flexibility using a broader interaction capabilities of the keyboard. Through a combination of intuitive keyboard shortcuts, context-sensitive commands, and interactive editing capabilities, these editors offer a more user-friendly and intuitive interface for code editing.

Some early works in this class provided textual editing capabilities and subsequently relied on **parsing** techniques to establish a structural comprehension and representation of the code. For instance, Barista [27] is a hybrid Java editor and an implementation toolkit that supports structure editing within an unrestricted text-editing environment and facilitates the creation of flexible structure editors [28]. Regarding editing capabilities, it provides the implementation facilities for structure views to project program text into alternate views, such as math equations. The authors highlight the limitations of traditional code editors that represent code as plain text, making it challenging to embed interactive tools, annotations, and alternative views within the code itself. To address this issue, Barista adopts an AST representation of the code internally and offers parsing techniques that treat the structure as if it were textual using a technique called "linearization" of the tree.

Regarding its framework capabilities, Barista expects user-defined interactions for each programming construct within the language to be defined by the creator of the editor (or the language). These interactions can include menu options, drag and drop functionality, and usable text-editing interactions. The framework follows a model-view-controller (MVC) architecture, with the model consisting of the modified AST structures and tokens. The view is a tree of interactive views that mimic the structure of the model, providing designers with control over layout, appearance, and interaction techniques. Controllers are implemented using the event-handling constructs offered by the Citrus programming language [26].

Another example is Grammar Cells [46], an approach for declaratively specifying textual notations and their interactions in projectional editors. By providing a formalism for defining consistent editor behaviors, this approach addresses the challenge of achieving consistency in editing experience for users. The paper presents a formal definition of grammar cells and their mapping to low-level editor behaviors.

Integrating structure editing capabilities into existing text editors offers several advantages which combines the flexibility and familiarity of text-based interfaces with the benefits of structure-aware editing. Duece [22] and GopCaml [18] followed this approach. Deuce offers a structure-aware code editor with direct manipulation capabilities built on top of Sketch-n-Sketch [23], a programming environment for creating SVG images. Its features include clickable widgets on top of the source code for structural selection and a user-friendly menu presenting potential transformations based on selections. Deuce's design has human-friendly structural interactions while retaining the freedom and familiarity of traditional text-based editing.

GopCaml introduces GopCaml-mode[2], a plugin for GNU Emacs, designed to enhance the editing experience for OCaml programmers by providing syntactic editor support. It addresses the limitations of standard text-based editors, like Emacs, in accurately capturing the structural complexities of OCaml syntax. Traditional editor interfaces struggle to effectively handle OCaml's syntax due to its complex structure. Existing plugins for OCaml support in editors like Emacs, Vim, and VS Code offer basic structural editing capabilities but rely solely on lexical analysis, limiting their ability to accurately represent OCaml syntax and support common code transformations. GopCaml-mode overcomes these challenges by implementing a zipper-based structural editing framework using the OCaml compiler infrastructure for AST definitions and parsing.

The aim of some works might not always align with the fundamental design choice of creating a structure editor. Some editors addressed specific challenges within their domain by tailoring their design to accommodate unique requirements. For example, Forest [47] is a structural code editor that integrates multiple cursors with structural editing commands that helps developers in performing repetitive edits across their

[2]https://gitlab.com/gopiandcode/gopcaml-mode

codebase. By splitting cursors and applying editing commands simultaneously, Forest enables users to perform edits similar to those achieved with refactoring scripts, but in a more interactive manner.

Hazelnut [33] and its successor works [34] are probably the most advanced structure editors that extensively handle keyboard interactions. Hazelnut extends a small bidirectionally-typed lambda calculus with holes and a cursor, enabling edit actions to operate over statically meaningful incomplete terms. This approach enables programmers to construct terms interactively, with the editor automatically placing terms inconsistent with expected types inside holes to defer type consistency checks until the term is completed. The focus of Hazelnut is on providing a principled solution to the problem of static meaning assignment in structure editors by ensuring that every edit state corresponds to a statically meaningful expression. The paper discusses various interpretations of holes and their connections with gradual typing. Hazelnut is implemented as a functional reactive program using `js_of_ocaml`.

In their "live functional programming with holes" paper [34], the authors further expand Hazlnut and introduce a live programming environment that provides continuous feedback about a program's dynamic behavior as it is being edited. Moreover, [35] introduces the concept of live literals (livelits), which enable users to fill holes in code directly through user-defined graphical user interfaces embedded into the code. Livelits provide a graphical representation for expressions that are more naturally manipulated graphically, such as colors, music, animations, and diagrams.

Another work in the Hazel family is tylr [31] which addresses the challenge of maintaining hierarchical term structures. The authors propose a novel approach called **tile-based editing**, which allows the disassembly of terms into linearly sequenced tiles and shards around user selections. This approach maintains term structure while offering linear selection and modification features, improving the selection and code restructuring tasks compared to traditional structure editors.

The tile-based editing approach in tylr operates on three levels of structure: terms, tiles, and shards. Terms follow the abstract syntax of the language, tiles correspond to groups of matching delimiters, and shards represent individual tokens and delimiters. The editor assists and guides user interactions through subsystems called the grouter and the backpack. The grouter aids the reassembly of tiles into terms, while the backpack guides user movement to ensure the proper reassembly of shards into tiles. This work was preliminary and had some limitations including the lack of multi-key input and multi-character tokens. We listed all the relevant papers of the Hazel family as "Hazel" in Table 2.1.

Beckman et al [7] argue that previous work on generating structure editors from language grammars relied primarily on menu and mouse-based interactions, which may not be as efficient or familiar to users accustomed to keyboard entry. Thus, their Partial Parsing approach addresses the challenge of automatically creating useful structure editors by introducing modifications to a parser of general-purpose programming language grammars to enable keyboard-centric interactions.

The paper discusses the structure of Tree-sitter grammars [9], which serve as a basis for auto-generating mappings from general-purpose programming language grammars to structure editors. The mappings map non-terminals to block compounds and terminals to text fields. Partial parsing utilizes autocomplete techniques to disambiguate language structures, which eliminates the need for a menu of language constructs. While the user enters language expressions via keyboard input, the system identifies language constructs that may start with the given input and prompts users to continue typing until only one construct remains or until they explicitly disambiguate between choices. The authors also mention techniques for minimizing the results set of partial parse trees to provide relevant options to the user.

The last paper we explore in this section is Javardise [39], which is a structure code editor designed for programming education. Javardise specifically targets a subset of the Java language and aims to provide an editing experience that closely resembles that of a conventional code editor. The editor ensures syntactic validity by disallowing syntax errors and automating certain elements such as balancing block brackets and inserting semicolons. Users interact with Javardise by typing elements of the actual source code in their regular order, but with constraints imposed to prevent syntactic errors and guide the typing activity.

The editing experience in Javardise is designed to maintain syntactically valid code at all times, except during transient moments when an identifier or literal is being written. The appearance of the source code in Javardise resembles that of a regular code editor, with automatic indentation and the appropriate placement of tokens. Certain tokens are editable and selectable, while others are not, to ensure the syntactic correctness of the code.

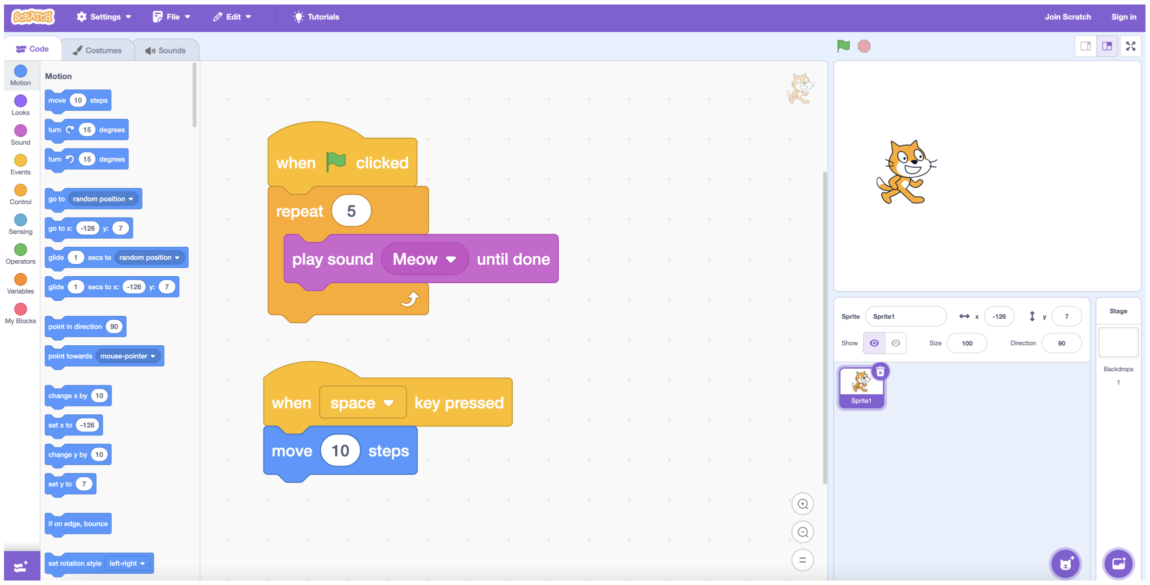
Javardise is implemented as a custom-made editor without relying on editor generation frameworks. The editor follows a model-view-controller architecture, where the abstract structure of the program is represented by an in-memory model and rendered through widgets in the view.

2.2.3 Block-based Editors

Block-based editors are coding environments that aim to simplify programming and make it more intuitive and accessible, particularly for novice programmers and children. The fundamental concept behind block-based programming is the use of *blocks* as the basic units of code. These blocks are visual, interactable elements that illustrate various programming constructs, such as loops, conditionals, variables, and functions.

One of the primary advantages of block-based editors is their accessibility and intuitiveness, especially for individuals without prior programming experience. In traditional text-based programming environments, learners must understand and remember specific syntax rules, which can be a significant barrier to learn. In contrast, block-based programming abstracts these syntactical complexities. Each block represents a specific piece of code or a programming concept, with its shape often indicating how it can be combined with other blocks that uses the human ability to recognize and organize shapes.

Additionally, block-based editors typically employ a rich use of colors to differentiate between various types of blocks. This color-coding not only makes the programming environment more visually engaging, especially for children, but also aids in understanding different programming structures. For example, loops might be represented in one color, conditionals in another to provide an immediate visual cue about the block's functionality. Figure 2.1, shows the user interface of the Scratch programming environment which consists of three main parts: a block palette (left), a coding area (middle) where the blocks can be dropped and arranged, and a stage area (right) where the programmer can see the output.



```
when [flag] clicked
repeat 5
    play sound Meow until done

when space key pressed
move 10 steps
```

Figure 2.1: User interface of Scratch

The primary mode of interaction in block-based programming environments is **drag-and-drop**. Users select blocks and drag them to appropriate positions to construct their programs. This type of interaction aligns well with the cognitive skills of young learners which makes programming feel more like a game or a puzzle for them. Moreover, the use of a keyboard in block-based programming environments is typically limited to specific tasks, such as entering names for variables or inputting string and numeric values.

The ease of use of block-based editors can come with trade-offs in efficiency. For simple operations or expressions, the process of dragging and dropping multiple blocks can become tedious, particularly for more experienced users. A critical consideration in designing educational programming tools has always been balancing simplicity and the potential for engagement with more advanced programming tasks or environments.

Alice [10, 36] stands out as a pioneering block-based editor designed specifically for

3D programming which aimed at tackling the challenges of authoring interactive 3D graphics. With its intuitive drag-and-drop interface and event-based, object-oriented paradigm, users can integrate objects into their scenes, manipulate their attributes, and invoke methods.

Scratch [38] is a widely recognized block-based programming environment designed to teach programming concepts to users of all ages, particularly targeting children between the ages of 8 and 16. The tool adopts a puzzle analogy and uses blocks as language constructs to facilitate the creation of interactive stories, games, and animations. Scratch provides a visual environment that enables users to snap together programming blocks to construct scripts, allowing for the creation of various interactive media.

While block-based editors like Scratch have proven effective in introducing programming concepts to novices, they may become cumbersome for more experienced users due to their mouse-driven input and low visual information density. Nonetheless, Scratch has achieved success in teaching programming fundamentals and, as of April 2024, it has a rank of 15 in TIOBE[3] index, which measures the popularity of programming languages through search engine data.

Snap! [21] an extended reimplementation (in JavaScript) of Scratch that has features such as first-class lists, procedures, sprites, costumes, sounds, and continuations. Snap! aims to provide a more advanced introduction to computer science suitable for high school or college students. While Scratch has been widely used as a programming language for children, it possesses limitations such as the absence of procedures and weak support for data structures. Snap! maintains the core simplicity and user-friendly nature of Scratch while introducing powerful features to support a

[3]https://www.tiobe.com/tiobe-index/

comprehensive introductory computer science curriculum.

Blockly [1] is a free and open-source software developed by Google and offers essential tools and components to construct block-based languages and editors. While primarily designed as a library, Blockly also has characteristics of a block-based editor itself. The introduction of Blockly paved the way for further research in the field of block-based editors. Users can interact with and manipulate blocks consistently as they construct their own programming environments. Blockly is implemented in JavaScript and has a smooth integration with it which makes it an easy-to-use solution that runs within web browsers and delivers a user experience similar to Scratch. Its effectiveness led Scratch to adopt Blockly as its implementation language starting with version 3.0.

One of Blockly's important features is its ability to generate code across various programming languages, including JavaScript, Python, and PHP. However, this translation is one-way, meaning that only the blocks can be converted to text code. As an attempt to make this conversion **bidirectional**, Droplet [6] was introduced that can convert blocks to code and vice versa. Droplet is used for the development of the Pencil Code[4] which is an educational programming language and website.

As an attempt to pave the way for creation of block-based editors, Kogi [42] is introduced as a tool designed to **derive block-based environments from context-free grammars**, particularly for domain-specific languages and other programming interfaces. Kogi aims to integrate the creation process by repurposing existing language artifacts which in turn significantly reduces the effort required to construct block-based environments from scratch. By transforming context-free grammars into abstract structures, Kogi generates block-based environments based on Blockly. The

[4]https://pencilcode.net

effectiveness of Kogi is demonstrated through four case studies that showcase its ability to create environments for DSLs such as state machines, sound synthesis, simple programming languages, and questionnaires.

Building upon Kogi's concept, Verano Merino et al. introduced S/Kogi [44] as an enhancement aimed at refining and optimizing block-based editors generated from language grammars. By analyzing input grammars and applying structural changes, S/Kogi reduces the complexity of the resulting block-based editors while adhering to established aesthetic guidelines. This refinement process significantly decreases the number of blocks, which enhances the usability of the editors across languages of varying complexities. S/Kogi represents a significant advancement in the field of block-based environment generation and user-friendly development processes.

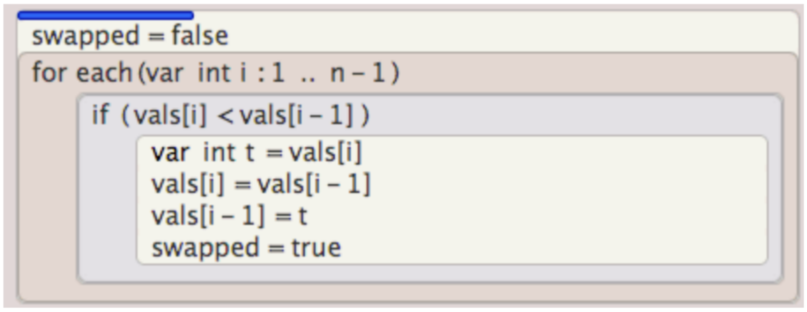
Finally, the same team of researchers introduced Blocklybench [43] as a workbench to facilitate the creation of block-based environments for both programming languages and semi-structured data languages. Blocklybench provides developers with a meta-block-based environment to describe specific elements of block-based environments using block notation. This tool enables developers to express key aspects such as layout, color, block connections, and code generators. Developing a block-based environment with Blocklybench involves several steps, beginning with the definition of language constructs (blocks), followed by the creation of categories to organize blocks for end-user accessibility. Finally, developers translate blocks into executable or data languages through the definition of code generators, which may involve multiple generators depending on language use cases. While Blocklybench offers a robust mechanism for developing block-based environments, it faces certain limitations, particularly in maintaining the link between blocks and code generation.

Integrating Blocklybench with approaches similar to S/Kogi's (such as annotating input grammars) could offer solutions to this challenge to enhance the overall efficiency and flexibility of block-based environments.

2.2.4 Frame-based Editors

Frame-based editors [8] represent another class of structure editors that blend the advantages of text-based and block-based editors. Block-based editors are great at reducing syntax errors and facilitate the recognition of code elements, especially for novice programmers. However, they often lack efficiency in quick manipulation of the code. On the other hand, text-based editors offer a more fluid editing experience for advanced users. To combine the best of both worlds, frame-based editors enhance user interaction with code by presenting each program element as a 'frame'. A frame, as shown in Figure 2.2, is a unified, indivisible entity that wraps a particular structure or statement within the program. These editors use simple graphical elements to represent concepts that are less effectively conveyed through mere text characters, such as scope. It should be noted that while program entry is primarily facilitated through keyboard shortcuts, such as pressing the `i` key to insert an 'if' frame, these editors do not fully support keyboard interactions.

The design of frame-based editors not only reduces common syntax errors, similar to block-based editors, but also preserves the textual representation of programming, which makes it suitable for both novices and advanced users. It also facilitates a smoother transition for novices advancing to more complex development environments. The authors in [37] evaluated the use of frame-based editors and suggested



```
swapped = false
for each (var int i : 1 .. n - 1)
    if (vals[i] < vals[i - 1])
        var int t = vals[i]
        vals[i] = vals[i - 1]
        vals[i - 1] = t
        swapped = true
```

Figure 2.2: Illustration of a couple of frames in the frame-based editor. The blue bar is the frame cursor.

that frame-based editing effectively reduces the burden of syntax for novice programmers and may improve performance on programming tasks.

2.3 Discussion

Table 2.1 summarizes the related works explored in this chapter. In Section 2.1, we have explored various methodologies for constructing structure editors and discussed how each approach influences the functionality and user interaction of these editors. It is important to note that a combination of these methodologies is possible, and an implementation might integrate more than one approach to create a desired editing environment. Moreover, it should be noted that some works, such as Blockly-Bench [43], are developed from scratch, but their final products can be utilized by others to create editors from language specifications.

For the implementation of ElmSr, we chose to **develop from scratch** to maintain full flexibility over the implementation, particularly of text-based interactions. Our choice of the Elm programming language facilitated this process due to its straightforward type system.

We also have addressed various types of structure editors. We discussed that initial strict structure editors often suffered from usability issues. Subsequently, user-friendly block-based editors emerged and have been particularly effective in teaching programming to novices. However, this simplicity presents significant drawbacks. On the one hand, they make expression entry tedious especially in expression-oriented programs. On the other hand, they can hinder novices from easily transitioning to more advanced programming environments due to their simple interfaces. In response to these challenges, frame-based editors were introduced, aiming to merge the advantages of both block-based and text-based systems.

Concurrently, developments in keyboard-driven structure editors tried to overcome the limitations of strict editors by enhancing keyboard interactions within the typical operations of structure editors. Our ElmSr editor falls into the category of **keyboard-driven structure editors**, as it supports extensive keyboard interactions similar to those expected in traditional text-based editors.

Table 2.1 also details how each editor supports the input of expressions. **Strict Structure** entry mode requires users to adhere to the AST's tree structure or employ drag-and-drop for each element of an expression. **Seamless structure** closely mimics the behavior of conventional text-based editors, while **free-form** allows for the free-form input of expressions which requires subsequent parsing to convert text into an AST. Additionally, the table lists the target languages for which these editors are designed.

Methods	Editor Type				Implementation				Expression Entry			Language Support
	Strict	Block-Based	Keyboard-Driven	Frame-Based	From Grammar	Using Workbench	Adapting Existing Editor	From Scratch	Strict Structure	Seamless Structure	Free-Form	
Emily [20]	●	-	-	-	-	-	-	●	●	-	-	PL/I
MENTOR [14]	●	-	-	-	-	-	-	●	●	-	-	Pascal, Ada
CPS [41]	●	-	-	-	-	-	-	●	-	-	●	PL/I
IPE [30]	●	-	-	-	●	-	-	-	●	-	-	any language
GNOME [17]	●	-	-	-	●	-	-	-	●	-	-	any language
GANDALF [19]	●	-	-	-	●	-	-	-	●	-	-	any language
Barista [27]	-	-	●	-	-	●	-	-	-	-	●	Java
Grammar Cells [46]	-	-	●	-	-	●	-	-	-	-	●	mbeddr C
Deuce [22]	-	-	●	-	-	-	●	-	-	-	●	a small functional lang.
GopCaml [18]	-	-	●	-	-	-	●	-	-	-	●	OCaml
Forest [47]	-	-	●	-	-	-	-	●	-	-	●	TypeScript

Hazel [33]	-	-	●	-	-	-	-	●	-	●	-	Elm/ML-like
Partial Parsing [7]	-	-	●	-	●	-	-	-	-	●	-	JavaScript
Javardise [39]	-	-	●	-	-	-	-	●	-	●	-	subset of Java
Alice [10]	-	●	-	-	-	-	-	●	●	-	-	Alice/Java
Scratch [38]	-	●	-	-	-	-	●[5]	-	●	-	-	Scratch
Snap! [21]	-	●	-	-	-	-	-	●	●	-	-	Snap!
Blockly [1]	-	●	-	-	-	-	-	●	●	-	-	many languages
Kogi [42]	-	●	-	-	●	-	-	-	●	-	-	domain-specific
S/Kogi [44]	-	●	-	-	●	-	-	-	●	-	-	domain-specific
Blocklybench [43]	-	●	-	-	-	-	-	●	●	-	-	any language/domain-specific
frame-based editor [8]	-	-	-	●	-	-	-	●	-	●	-	Stride
ElmSr (our editor)	-	-	●	-	-	-	-	●	-	●	-	Elm

Table 2.1: The comparison of the related work. ●: supports the property; -: does not support the property.

[5]From version 3.0, Scratch started using Blockly as the implementation library.

Chapter 3

Preliminaries

In this chapter, we introduce the concepts that are essential in understanding the thesis. We will examine the Elm programming language, delving deep into its architectural framework and core principles. We will then detail Elm's type classes and their role in our editor. The subsequent sections discuss the types of polymorphism in functional programming languages. We further explore the Algebraic Thinking approach in teaching problem-solving and computer science concepts. We then introduce two code refactoring techniques that are essential in the algebraic thinking curriculum. Finally, we discuss the general concept of AST with a focus on zipper concepts.

3.1 Elm Programming Language

Elm is a functional programming language that aims at designing responsive frontend web applications [11]. Unlike many mainstream languages, Elm is characterized by its strong static typing, emphasis on purity, and absence of runtime exceptions. It should be emphasized that Elm has a robust type system, which not only assists developers in writing correct code but also provides human-readable error messages that guide them towards solutions.

Elm provides developers with a rich set of tools and abstractions for building web interfaces. An Elm application can be compiled into an optimized JavaScript code, however the standard tooling[1] can wrap this code in an HTML file for ease of use and deployment.

Elm's architecture, commonly referred to as The Elm Architecture (TEA), structures applications in a unique Model-View-Update (MVU) pattern, which combined with Elm's emphasis on immutability and functional paradigms ensures consistent and maintainable application design. The architecture is further enhanced by the Elm Runtime which orchestrates various interactions. In this section we give an overview of The Elm Architecture and its components.

Model. In the Elm architecture, the `Model` represents the state of the Elm application. It is a data structure typically realized as a record or custom type that the developer defines based on what kind of stateful information the application requires. The `Model` captures all stateful information, be it user data, UI status, or other application data. As an example, the `Model` in a simple counter application might be an integer holding the current count (state):

```
type alias Model = { count : Int }
```

Elm enforces immutability, meaning once a data structure is created, it cannot be changed. Instead, any *changes* must be realized by creating a new copy with the modifications applied. Hence, updates to application's state generate a new instance

[1]`elm make`

of the `Model`.

view. The `view` function renders the application's current state into a user interface. It transforms the `Model` into a virtual DOM, which is then converted into an actual webpage DOM by the Elm Runtime. Depending on the specific Elm packages used for the user interface, the `view` function can build outputs in HTML, SVG, etc. The following is the type annotation for the `view` function:

```
view : Model -> Html Msg
```

It takes the current state of the `Model` and returns HTML that can produce messages of type `Msg`.

update. The `update` function implements the main logic of the application, reacting to different messages (events). Given the current `Model` and a message, it determines how the `Model` should change. This function potentially yields a new `Model` (due to immutability) and can also produce side effects as commands, if necessary.

As can be seen in the following annotation, the `update` function accepts a message of type `Msg` and the current `Model`, and returns a potentially updated `Model` and commands that can produce other messages of type `Msg`:

```
update : Msg -> Model -> ( Model , Cmd Msg )
```

Other than the above three components, Elm's central orchestrator is the **Elm Runtime** which is the environment in which the Elm code runs. It basically manages interactions, state and rendering, and bridges the gap between the application (the above functions) and the outside world (such as browser's DOM). The Elm Runtime is responsible for listening to events, updating the `Model` based on messages and the `update` function, executing side effects (`Cmd`), managing subscriptions (`Sub`), and rendering the view using the `view` function.

subscriptions. The `subscriptions` function is responsible for setting up event listeners for various events, like key presses, and mouse movements. When these events occur, they generate messages that are sent to the `update` function. Thus, the `subscriptions` can be defined as the following which takes the `Model` and returns a subscription (`Sub`) that can produce messages of type `Msg`.:

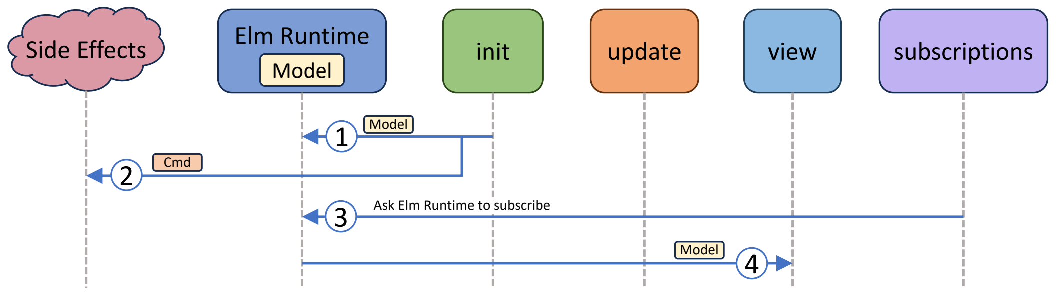
```
subscriptions : Model -> Sub Msg
```

init. The entry point of an Elm application is the `init` function. It takes the `Model` as the initial state of the application and can send out initial commands that can produce messages, if necessary.

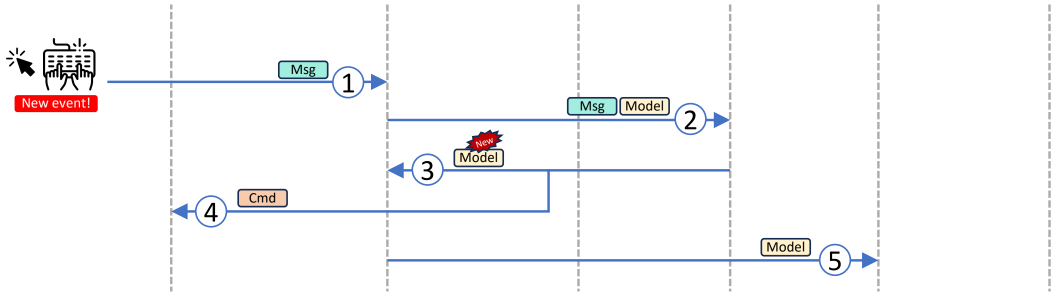
```
init : ( Model , Cmd Msg )
```

Figure 3.1 illustarates the lifecycle of the Elm application. Figure 3.1a shows that the process starts with the `init` function which sets the initial state of `Model` (①) and can produce the intial list of commands (②) as side-effects that are executed outside of the Elm Runtime. Then, `subscriptions` asks Elm Runtime to set the event listeners the application should subscribe to (③). Once the initial state is set and event listeners are configured, the `model` is passed to the `view` function (④) to render the user interface. At this point, the application waits for user events.

Figure 3.1b depicts the process of handling user events. When a user performs an action, such as a key press, a corresponding event is generated and translated it into a message (①). The message, along with the current state (`Model`), is passed to the `update` function (②). The `update` function, returns a **new** model (③) and commands for side effects (④). After the state transition, the Elm Runtime invokes the `view` function to render the new model (⑤). Any further user interaction produces new messages, continuing the above loop.



(a) Elm application's initialization sequence: Model setup, initial commands dispatching, event subscription configuration, and initial view rendering



(b) Processing user events: Message reception by the runtime, state update, and view re-rendering accordingly

Figure 3.1: The Elm application lifecycle

3.1.1 Type Classes

Type classes in programming languages are a type system construct that supports ad hoc polymorphism, which is the ability of a value to adopt any one of several types because it can be evaluated in different ways based on its context. Type classes define a set of functions that can have different implementations depending on the type of data they are used on. This allows functions to be written generically, making them flexible in handling different types.

Haskell, for instance, has a well-known type class system that allows for the creation of functions that can accept multiple types, and it can determine the appropriate type at compile time. It means that a programmer can implement a function that behaves differently (and appropriately) based on the type of the input.

Elm, on the other hand, has no user-defined type classes. Elm does, however, have a few built-in type classes such as `number`, `comparable` and `appendable`. These built-in classes, unlike user-defined type classes, are limited in scope but still play a vital role in Elm's type system. It should be noted that Elm doesn't have type classes in the same way as languages like Haskell. It is more accurate to say that Elm has a few built-in constrained types that function in ways similar to type classes in other languages. It is not possible for an Elm programmer to create their own type classes or extend the existing ones. Although they are limited, they are crucial in allowing values to be contextually evaluated which enables Elm to ensure type safety while offering some degree of type flexibility.

In the following, three built-in type classes of Elm are described:

- **number:** This can be either an `Int` or a `Float`. Elm will determine the precise type based on the context in which the number is used. In the following example, the type of `x` is inferred based on the context in which it is used. When it is added to an `Int`, it is considered an `Int`, and when it is added to a `Float`, it is considered a `Float`:

```
x = 3
sumInt = x + 2   -- 5 : Int
sumFloat = x + 2.5  -- 5.5 : Float
```

- **comparable:** This type class is used for types that can be ordered or compared using relational operators like `<`, `<=`, `==`, `/=`, `>=`, and `>`. These operators work on all types that are part of the **comparable** type class including `Int`, `Float`, `Char`, `String`, `Time.Posix`, and `List comparable` types. For example:

```
isLess = 2 < 3  -- True
isSame = "Hello" == "World"  -- False
```

- **appendable:** This type class is used for types that can be appended using the `++` operator. The types `String`, `List a`, and `Text` are contained in the **appendable** type class. For examle:

```
greeting = "Hello" ++ " World!"  -- "Hello World!"
numbers = [1, 2, 3] ++ [4, 5, 6]  -- [1, 2, 3, 4, 5, 6]
```

In our editor's implementation, we have chosen to adopt a similar strategy, and we have implemented only the number type class. Rather than implementing a generic handling of type classes (which Elm does not support for user-defined types), we have provided specific editor support for built-in number type class, and postponed the implementation of the other two for future works. This approach aligns with Elm's philosophy of simplicity and safety. As demonstrated in Section 4.7, we have built a supertype solution for numeric types to allow a smooth transition between integer and float types.

3.2 Polymorphism in functional programming

In this section, we discuss the two forms of polymorphism in functional programming.

Parametric Polymorphism

Parametric polymorphism allows implementing functions that operate uniformly on any type where the behavior is the same regardless of the type being used. This enables higher levels of abstraction and code reusability.

A common use of parametric polymorphism in Elm is the implementation of generic data structures and higher-order functions like `List.map`. Here's an example that illustrates the use of parametric polymorphism with a function that doubles every element in a list, using `List.map`:

Listing 3.1: Parametric Polymorphism Example

```
doubleList : List number -> List number
doubleList numbers = List.map (\n -> n * 2) numbers
```

The `List.map` function itself is a great example of parametric polymorphism:

`List.map : (a -> b)-> List a -> List b`

Here, `a` and `b` can be any types, making `List.map` applicable to a wide variety of situations. It abstracts the pattern of applying a function to every element in a list, regardless of the specific types involved.

Ad hoc polymorphism

Ad hoc polymorphism allows different implementations of a function depending on the type of arguments it receives. Essentially, it is a way to provide multiple implementations of a function under the same name for different types.

Elm does not natively support ad hoc polymorphism in the way that some other languages like Haskell do. However, in Elm, pattern matching with custom types is a way to define different behavior for different cases, though, it should be noted that it is not the same as ad hoc polymorphism.

In the following, such an example in Elm is illustrated where a custom `Shape` type with different constructors for different shapes is defined. A function to calculate the area can be defined specifically for this type, with different implementations for each shape:

Listing 3.2: Similar to Ad Hoc Polymorphism Example in Elm

```elm
type Shape = Circle Float | Square Float


area : Shape -> Float
area shape =
    case shape of
        Circle radius -> 3.14 * radius ^ 2
        Square side -> side ^ 2
```

3.3 Algebraic Thinking

The curriculum at the heart of this research is called “Algebraic Thinking” which has been used for 7 years in the outreach activities at McMaster Univeristy in collaboration with local teachers. It is an approach to teaching computer science, particularly to K-8 students [13].

This curriculum places the main emphasis on the general concept of functions before narrowing down to its specific applications like arithmetic. The approach challenges traditional teaching methods, which usually build on learners’ prior knowledge of arithmetic to explain programming concepts. Thus, instead of building on concepts like numbers and arithmetic and developing imperative programming, it builds on notions of shapes and transformations, and develops higher-level algebraic concepts using functional programming.

By teaching functions first, the curriculum aims to elevate learners to higher levels of mathematical abstraction early on. It demonstrates the utility of algebraic thinking, highlighting the importance of functions and then mapping arithmetic to this conceptual understanding of functions.

In practical terms, this teaching approach involves engaging students in creating graphics and animations using Elm and the GraphicSVG library[2]. Concepts are often introduced as "tips" for enhancing their creations, rather than as rigid, standalone lessons. This tactic motivates students, as they can immediately see the benefits of applying these new concepts.

As an example, students learn how to define an object with a GraphicSVG group and create copies of a graphic on a collage plane. As they progress, they are encouraged to reuse code instead of duplicating it. They also learn to modify variables to create variations in their graphics, such as different colors.

3.4 Code Refactoring

Refactoring in programming is the process of modifying existing code to improve its structure, design, or readability, without altering its external behavior or functionality. Several types of refactoring operations are commonly utilized in programming editors. For example, a simple refactoring operation is rename which is as simple as altering the name of a code element like a variable, function, or class to make it more descriptive or accurate. Another more complex is inline refactoring which is done when a function's body is as clear as its name, or if it is called just once. It often makes sense to remove the function and incorporate its body into the original call location. These refactoring strategies, among others, boost the code's readability,

[2]https://package.elm-lang.org/packages/MacCASOutreach/graphicsvg/latest/

maintainability, and in some cases, its performance.

However, in the context of our specific teaching style, Algebraic Thinking, two types of refactoring operations are particularly interesting:

Extract Parameter: In this operation, a constant or literal value within a function body is converted into a function parameter. This refactoring operation aligns with our teaching style by helping students develop concepts by generalizing examples into abstract patterns as realized by functions. This abstraction, rather than being a brain teaser, is a practical way of broadening a function's applicability. An example of creating a `color` parameter is shown below:

Before:

```
myShape = circle 10 |> fill red
```

After:

```
myShape color = circle 10 |> fill color
```

Extract Definition: This operation involves taking a segment of code, assigning it to a new definition (like a function definition) in the immediate or outer scope, and replacing the original segment with the new variable. This operation is a clear demonstration of how complex expressions can be decomposed into simpler, reusable parts — a crucial concept in our teaching style. By transforming an entire expression into a definition, students learn to view code not just as a linear sequence of instructions, but as a collection of discrete, reusable units that can be composed (hence the original definition of Algebra as taking apart and putting together [13]) to achieve complex outcomes. An example of such refactoring is shown below which defines 5 as a new definition:

Before:

```
let
    calculateArea r = pi * r * r
in
    calculateArea 5
```

After:

```
let
    radius = 5
    calculateArea r = pi * r * r
in
    calculateArea radius
```

These refactoring operations are supported by tools such as IntelliJ IDEA[3], Visual Studio Code, and Eclipse for widely-used languages like Java, Python, and C++. However, for less mainstream languages like Elm, the available tooling might be more limited.

3.5 Abstract Syntax Tree

An Abstract Syntax Tree (AST) is a tree representation of the structure of source code which captures the code's essence, representing statements, operations, operands, and their relationships in a condensed form. Within an AST for an expression, each internal node denotes an operator, and its children symbolize the corresponding operands. This representation can be generalized to other programming constructs like the if-then-else statement for which the construct (if-then-else) can be considered as the

[3]IntelliJ Idea's support of this refactoring: `https://www.jetbrains.com/help/idea/extract-parameter.html`

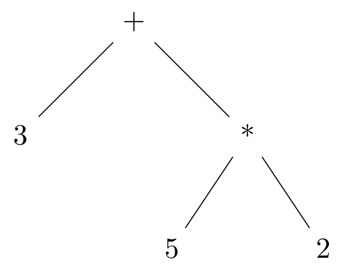


Figure 3.2: An example AST

operator and its components (condition, then branch, else branch) can be regarded as operands [5]. For instance, consider the arithmetic expression `3 + 5 * 2`. Figure 3.2 shows the AST representing this expression that prioritizes the multiplication operation due to its precedence.

On the other hand, a parse tree represents the syntactic structure of a source code based on a specific formal grammar. Each leaf node (terminals) represents a token from the expression and interior nodes (non-terminals) represent the grammatical constructs from the grammar that were used to recognize the sequence of tokens.

Unlike a parse tree, which represents every detail and construct in the source code, an AST abstracts away certain syntactic details and focuses on the semantic structure of the code. Therefore, ASTs are more suitable for structured editors to facilitate more intuitive manipulations.

3.6 Zipper

The Zipper data structure, as introduced by Gérard Huet in his seminal paper [24], provides a method to efficiently navigate and manipulate immutable tree structures. It is essentially a way to "focus" on a specific part of a tree, enabling operations at that location without having to reconstruct the whole tree. This method effectively turns the tree “inside out”, so that the element under focus is the root of a new tree, accompanied by its context.

The zipper elegantly solves the challenges of working with immutable data structures, particularly when changes need to be **localized**. Without a data structure like the Zipper, altering an element deep within an immutable tree would require the reconstruction of significant portions of the tree. This can be a huge problem when working with large trees. The zipper, however, by retaining the context, allows for such modifications in a more efficient manner.

Understanding the Zipper data structure is key for this thesis. Therefore, we will delve deeper into the Zipper's core concepts, explaining its definitions of tree, location, and path, all presented in the Elm language[4].

The `Tree` is defined as:

```
type Tree
    = Item String
    | Section (List Tree)
```

where the constructor `Item` of type `String` represents a leaf node that contains an item, and the constructor `Section` represents an internal node that contains a list of its child Trees.

A `Path` represents a way to reach a particular node in the tree and recursively captures the surroundings of a focused node. It can be defined as:

```
type Path
    = Top
```

[4]Note that the original definitions in Huet's paper were in OCaml. We focus on explaining the structure rather than the exact translation. For example *Item* in *Tree* definition in the original paper is *Item of item*, but we simply represented it as *String* in definitions.

```
    | Node (List Tree) Path (List Tree)
```

where `Top` represents the root of the tree, and `Node(l, p, r)` is a node in the path in which `l` is the list of left siblings (in reverse order), `p` is the parent's path, and `r` is the list of right siblings (in order).

A `Location` in the tree is a pair consisting of the current focused subtree and the path leading to that subtree which can be defined as:

```
type Location
    = Loc Tree Path
```

For example, the tree in Figure 3.2 is represented as:

```
treeExample : Tree
treeExample =
    Section
        [ Item "3"
        , Item "+"
        , Section
            [ Item "5"
            , Item "*"
            , Item "2"
            ]
        ]
```

and visually as in Figure 3.3.

If we were to focus on the "*" operator within this tree, the location would be:

```
Loc
    (Item "*")
```

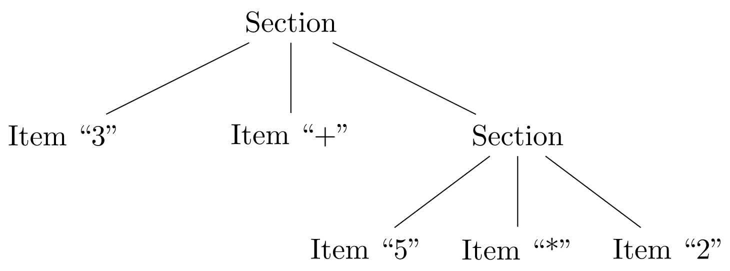


Figure 3.3: AST of Figure 3.2 represented as a Tree in Huet's paper

```
(Node
    [Item "5"]
    (Node
        [Item "+", Item "3"]
        Top
        [])
    [Item "2"]
)
```

In his paper, Huet suggests certain modifications to adapt the Zipper for use with ASTs. Initially, the tree structures in Huet's approach are untyped and do not have specific labels for tree nodes. To facilitate the manipulation of ASTs, it is recommended to label these tree nodes with proper names in the Path. For instance, the operands of a binary operator in an expression tree, like +, can be addressed with "left" and "right" in the Path to distinguish the location of the cursor. We will describe our interpretation of this adaptation in Section 4.2.

Chapter 4

ElmSr

In this chapter, we describe ElmSr, the novel structure editor specifically created for the Elm programming language. We elaborate on our design goals, ElmSr's high-level design and choices, and the granular implementation details.

4.1 Design Goals

This section outlines the fundamental design goals for our ElmSr editor, which aims to make the programming learning experience more efficient and enjoyable for novices.

DG1: User-Friendly Design. The editor should be intuitive and straightforward to use, specifically designed for children who are novice programmers. Moreover, if the shortcut keys are needed, they must be defined as intuitive key combinations, and they should not be overwhelming.

DG2: Error-Free Coding Environment: The editor should guide users to write correct and valid Elm code by providing different methods of code construction to support this goal.

DG3: Algebraic Thinking Curriculum Support: The editor should facilitate the implementation of the algebraic thinking curriculum. It should allow for unique features that align with this teaching style, such as converting constants in a function's body to a function parameter, or transforming an entire expression into a definition. These features support the teaching philosophy of emphasizing functions over arithmetic and promote code reusability and abstraction.

DG4: Providing Consistent Experience for Arithmetic Expressions: The editor should provide a uniform user experience for expression construction and editing, specifically in handling infix expressions, which ensures consistency and minimizes the need for users to learn complex operations. By simplifying the process, users can simply input their desired expressions, similar to typing in a traditional text editor.

DG5: Semantic Focus: The editor should support a structure-based focus, enabling highlighting of code blocks or their components. This functionality can emphasize semantic connections within the code, aiding the user's understanding of program structure and logic.

DG6: Structure Manipulation: The editor should support easy and correct code modifications. This includes the ability to rename elements, change numbers, and manipulate code structures, offering a type-sensitive drop-down menu and copy and paste functionality for code insertions.

DG7: Navigation Consistency: The editor should provide a way to navigate through the code that aligns with the visual layout of the programming blocks, rather than strictly following the tree-like structure of the underlying code. This enhances the connection between the code and its visual representation.

4.2 Adapted AST and Zipper

In section 3.6, we explored the original concept of the zipper data structure and highlighted its utility in efficiently navigating and manipulating immutable tree structures. In this section, we describe our adjustments to the AST and the zipper to align with our specific requirements. Our adaptation involves a fully-typed AST and a corresponding recursive zipper path type structure.

4.2.1 Fully-Typed AST

In the ElmSr editor, we have designed a fully-typed AST. This means that each node within the tree doesn't just represent a language construct, but also carries specific type information with it. This approach was particularly feasible due to Elm's relatively simple type system, which allowed us to create a fully typed AST that is easy to understand, manipulate, and maintain. Elm's simplicity in type handling greatly facilitated the development of our structure editor.

Listing 4.1 represents an excerpt of the fully-typed AST. Note that, the code showcased here is just an excerpt and does not encompass the full scope of the ElmSr's supported types. The type `ElmType` represents any valid Elm type within an Elm code and `ElmNum` represents valid Elm expressions with a numerical outcome.

Listing 4.1: Typed AST

```
type ElmType
    = ElmNum
    | ElmBool
    ...
```

```
type ElmNum
    = NumHole
    | NumIf ElmBool ElmNum ElmNum
    | NumInfix String ElmNum ElmNum
    ...
```

For instance, consider the Elm expression `if a then 3 else 4`. In our AST, this entire `if-then-else` statement is encapsulated within a `NumIf` node that indicates the numerical outcome of the expression. Yet, each constituent part of the statement – `a`, `3`, and `4` – stands as a separate node with attached type information, with `a` being an `ElmBool`, and `3` and `4` being `ElmNum`.

It should be noted that to represent an `if-then-else` that evaluates to a specific type, our AST has a similar rule in the corresponding `ElmType` type definition. For example, for an `if-then-else` that evaluates to a boolean type (`Bool`), such as in the expression `if (5 > 2) then True elseFalse`, we have a corresponding rule in `ElmBool` type definition (i.e., `BoolIf ElmBool ElmBool ElmBool`.)

Typed Holes

We utilize the concept of typed holes to enhance the coding experience while maintaining code correctness consistently. Typed holes are specific locations within the code where the user needs to input an expression of a designated type. We have modeled these holes in our AST to support structured code completion. They are denoted as ***Type***`Hole` where ***Type*** can be any of the Elm types supported by our editor. For example, `NumHole` node corresponds to a location in the code where the user can input an expression of type Num as indicated in Listing 4.1.

Once the cursor is located at a hole, the user can simply type the desired expression. To assist the user in filling these holes accurately, ElmSr provides a user-friendly approach. When a user starts typing within a typed hole, a drop-down menu appears, offering a selection of options that are compatible with the type of the hole. This drop-down menu ensures that the user's input aligns with the type of the hole. As the user gradually constructs their code, the typed holes ensure that the code is syntactically correct. Of course, the presence of these holes constantly reminds the user that there are gaps to be filled before considering the code as complete.

Definitions

ElmSr's AST includes various types of definitions that can be used in an Elm code, some parts of which are outlined in Listing 4.2. For example, in our AST, `DefNumVar` indicates the definition of a numerical value. It consists of a variable name (of type `VarName`, an alias for a string) and the corresponding numerical value expressed as `ElmNum`. The constructors `DefBoolVar` and `DefListVar` are the same as `DefNumVar` except that they define a boolean and a list respectively. To define functions returning numerical results `DefNumFunc` is used which includes the function name (represented by `VarName`), a list of parameters (`List ElmParam`), and the numerical return value indicated by `ElmNum`. Finally, the constructor `DefCustomType` enables the definition of custom types. It includes the type constructor name (as `TyConstructor`), a list of type variables (`List TyVarName`), and a list of data constructors (`List Constructor`) that define the structure of the custom type.

We omit the detailed path definition for `ElmDef`, but it should be noted that `ElmDefPath` defines various path constructors that help us in tracking the cursor's location within different parts of each definition, including the definition name, value, and parameters.

Listing 4.2: Definitions

```
type ElmDef
    = DefNumVar VarName ElmNum
    | DefBoolVar VarName ElmBool
    | DefListVar VarName ElmList
    | DefNumFunc VarName (List ElmParam) ElmNum
    | DefCustomType TyConstructor (List TyVarName) (List ↩
      Constructor)
    ...
```

Module

In ElmSr, the `ElmModule` type is used to represent a module, which is essentially a list of definitions as can be seen in Listing 4.3. For the purpose of this thesis, we mainly focused on defining a single module. However, it is important to note that expanding our editor to include multiple modules is straightforward.

Furthermore, in our AST, similar to the "Top" constructor in Huet's zipper path, we have `EMTop` that indicates the highest level of the AST.

Listing 4.3: Module

```
type ElmModule
    = Module (List ElmDef)
type ElmModulePath
    = EMTop
```

The wrapper and type safety

The fully-typed AST defined above enhances code manipulations and minimizes potential type errors by utilizing facilities of the Elm compiler. For example, the Elm compiler can prevent attempting to enter a `String` where an `Int` is expected (according to the AST). However, in the design of our typed AST, there might be situations where the type of a subtree's head is ambiguous such as in constructs like list elements, or we have a heterogeneous set of elements that are required to be in sequences like lists. For example, the list of function arguments might include integers, strings, and even other functions.

One potential solution could be to embed all possible types into the AST, thus enforcing strict type safety. However, this approach restricts ElmSr to a finite set of types. For the editor to be versatile, allowing constructs like nested lists and recursive types are essential. These recursive constructs inherently lead to an infinite set of possible types that avoids achieving complete type safety in our editor. In contrast, languages like Haskell, with their support for existential types are able to address this issue. But within the constraints of Elm, such features are unavailable.

One way to unify these different types into a consistent format is by using a wrapper type as shown in Listing 4.4. Having this type defined, a numerical function application can be indicated as `NumFunApp String (List Wrapper)` where `(List Wrapper)` captures the list of arguments to be passed to the function.

Listing 4.4: Wrapper

```
type Wrapper
    = WrapNum ElmNum
    | WrapBool ElmBool
    | WrapList ElmList
    ...
```

It should be noted that the flexibility offered by the wrapper also raises concerns regarding type safety. Though the AST aims to be 'fully-typed', the wrapper, due to its generic nature, might lead to situations where explicit type checks could be bypassed. Given these constraints, the wrapper, however, seems to be a practical solution. It ensures that from a user's perspective, the interactions remain consistent and free from type-unsafe scenarios. Moreover, it introduces a trade-off between type generality and type specificity.

Zipper's Location and Path

Modeling the tree is only one aspect of this design. To manipulate and navigate the fully-typed AST, ElmSr employs an advanced variant of the zipper data structure. Traditional zipper structures allow efficient navigation and manipulation of data structures by maintaining a *cursor* and a list of *breadcrumbs*. However, they often lack type safety or the ability to handle diverse data structures. In contrast, the zipper implemented in ElmSr maintains full type safety and can accommodate the fully typed AST using `Location` type and the relevant path types described in the following.

ElmSr keeps track of the cursor's location using the zipper structure. As mentioned in Section 3.6, this location stores both the cursor's position and facilitates the reconstruction of the tree around that cursor. As shown in Listing 4.5, this type includes a multitude of constructors, one for each node type in the AST, indicating where the cursor is currently located and carries along a path object. For instance, if the cursor points to an `ElmBool` node, a `LocBool` will keep track of it, which contains the `ElmBool` node itself, and its path within the AST.

Listing 4.5: An excerpt of the Location Type definition

```
type CaretPosition
    = Start
    | End
    | Oper
type Location
    = LocNum ElmNum ElmNumPath CaretPosition
    | LocBool ElmBool ElmBoolPath CaretPosition
    | ...
    | LocMatch ElmMatch ElmMatchPath
    | LocList ElmList ElmListPath
    | LocItem ElmItem ElmListPath
    | LocDef ElmDef ElmDefPath
    | LocModule ElmModule ElmModulePath
    | LocParam ElmParam ElmDefPath
    | LocDataConstr Constructor ElmDataConstrPath ElmDefPath
```

The zipper's path, defined with types like `ElmNumPath`, `ElmBoolPath`, etc., as shown in Listing 4.6, assists in reconstructing the complete AST from the current cursor location. In the context of the previous example `if a then 3 else 4`, consider a situation where the cursor is located at `a`, the boolean condition of the statement. The location of the cursor would be represented as `LocBool a (CondOfNumIf_ parentPath 3 4)` according to `LocBool ElmBool ElmBoolPath` in the definition of `Location`, and `CondOfNumIf_ ElmNumPath ElmNum ElmNum` in the definition of `ElmBoolPath`.

The (CondOfNumIf_ parentPath 3 4) in the cursor location means the cursor's parent node is a NumIf node, with 3 and 4 being the then and else parts respectively. The parentPath in this case represents the path from the NumIf node to the root of the AST. This path would depend on the overall structure of the code.

Overall, the recursive path structure allows us to navigate from any node in the AST to the root while maintaining type safety. This is a powerful feature in AST manipulation as it allows precise navigation and manipulation while preventing type mismatches.

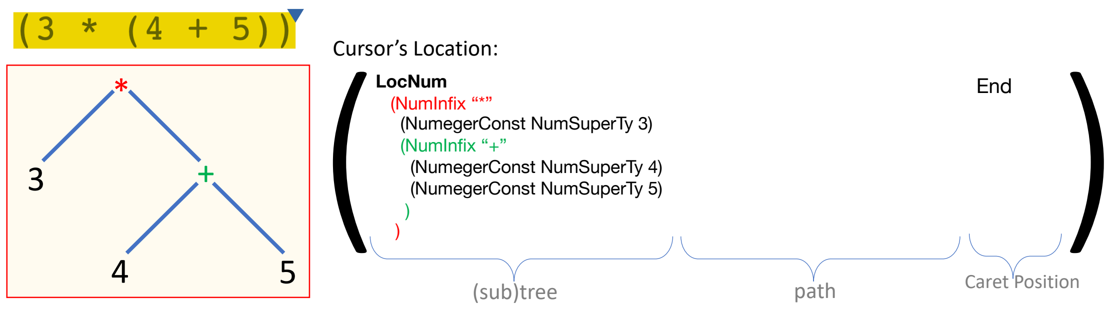
Listing 4.6: Recursive Path Types

```
type ElmNumPath
    = ThenOfNumIf_ ElmBool ElmNumPath ElmNum
    | ElseOfNumIf_ ElmBool ElmNum ElmNumPath
    | LeftOfNumInfix_ String ElmNumPath ElmNum
    | RightOfNumInfix_ String ElmNum ElmNumPath
    | ...
type ElmBoolPath
    | CondOfNumIf_ ElmNumPath ElmNum ElmNum
    | ...
```

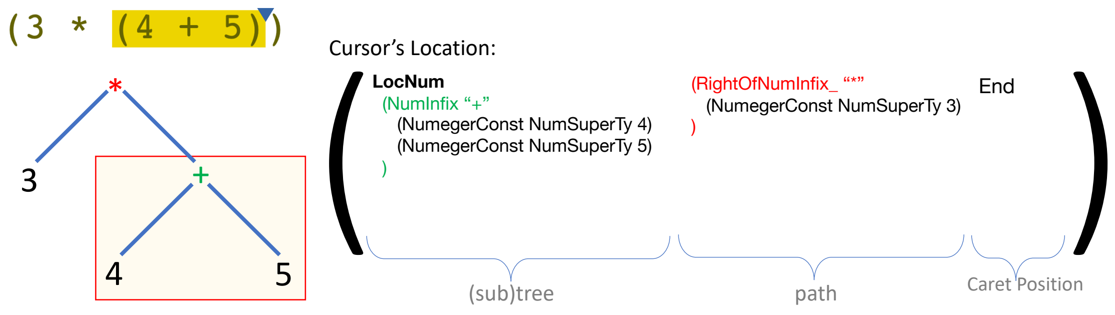
As another example, imagine a tree representing an arithmetic expression: (3 * (4 + 5)). If our focus is on the addition operation, the cursor will mark (4 + 5). In our model, the focus sub-tree is entirely (4 + 5), and the path is captured as RightOfNumInfix_ "*" (NumegerConst NumSuperTy 3)parentPath in which parentPath may contain the rest of the path[1]. Figure 4.1 visualizes how the zipper

[1]For example, if the expression was the body of a function "func", the path could be RightOfNumInfix_ "*" (NumegerConst NumSuperTy 3)(BodyOfNumFun "func" ...)

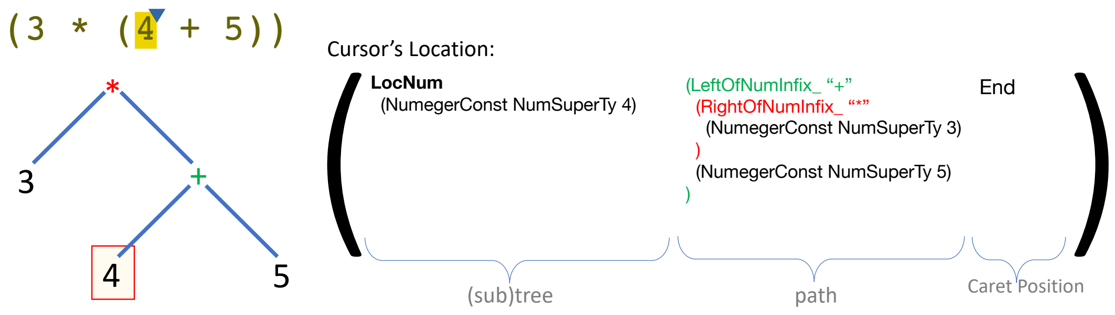
changes when the cursor moves over the expression. When required, the original expression can be reconstructed.



(a) The focus is on the whole expression



(b) The focus moves on `(4+5)`



(c) The focus moves on `4`

Figure 4.1: AST Zipper Diagram

Our approach provides an intuitive way to manipulate tree structures in our editor, while also allowing for efficient projection, modifications and reconstructions.

4.2.2 Modularized Projection

Our adapted zipper modeling approach has allowed for simplification of our projection logic. By modularizing the rendering process, the task of continous projection of the tree is divided into more manageable and intuitive subtasks, as described in the following:

- `pretty`*`Type`* Functions: These functions are responsible for rendering representations of the given type. Examples include `prettyNum` for numbers and `prettyBool` for boolean values. Each `pretty`*`Type`* function provides a visualization of the primary element under focus, and generates the output shape of that focused part of the tree. `pretty`*`Type`* functions either generate the shape or call the `mk`*** functions.

- `mk`*** Functions: These utility functions are designed for specific constructs in the programming language. Their role is to standardize the rendering of these constructs. `mkBinary`, for instance, would ensure that all binary operations, irrespective of their specific operators or operand types, are visualized consistently. Similarly, `mkIf` standardizes the rendering of conditional statements.

- `prettyWrap`*`Type`* Functions: Supporting the `pretty`*`Type`* functions, the `prettyWrap`*`Type`* suite is responsible for rendering the broader context surrounding the primary focus. They render the non-focussed parts of the program which "wrap" the focus, and include sibling nodes and other definitions in the module. Each function, such as `prettyWrapNum`, operates by recursively decomposing the path. During each recursive call, the path is segmented into two parts: the immediate outer context and the residual path. The immediate context is utilized for the current rendering operation, while the residual path is passed on to subsequent recursive calls. This strategy ensures that every layer of the path is accurately visualized, starting from the immediate context and progressively expanding outwards.

- `prettyLocation` Function: An essential component in our rendering mechanism is the `prettyLocation` function. This function is the main function in our AST visualization system, as it continuously renders the program code AST. Based on the location type, it calls the corresponding `pretty`***`Type`*** function and then calls the relevant `prettyWrap`***`Type`*** function to wrap its output. It actively projects the whole tree and highlights the location of the cursor. An example is shown in Figure 4.2 which illustrates how the projection of Listing 4.7 is rendered using the location of the cursor. The cursor is on `a2` which is a `LocNum` type of location (①). Listing 4.8 is the path of the cursor which can be described as follows: At the most granular level, `a2` is the right operand of an addition operation paired with `a1` (②). This specific addition, `(a1 + a2)`, subsequently acts as the left operand in another summation, complemented by the constant `45` on its right (③). This combined expression, `((a1+a2)+ 45)`, outlines the body of a function `"fun1"` with two parameters `a1` and `a2` (④). Zooming out to the module's broader landscape, this function `"fun1"` is placed between two definitions. Preceding it (the first list in `ModulePath`), there is `x = 2`, and after it (the second list in `ModulePath`), there is `y = 9` (⑤). Our projection function uses such an interpretation of the location to render the projection, as shown in Figure 4.2.

Listing 4.7: Example code

```
x = 2
fun1 a1 a2 =
    ((a1 + a2) + 45)
y = 9
```

Listing 4.8: Cursor's Path when a2 is highlighted in Listing 4.7

```
(RightOfNumInfix "+" (NumVar "a1" NumSuperTy)
    (LeftOfNumInfix "+"
        (BodyOfNumFun "fun1" [NumParam "a1",NumParam "a2"]
            (ModulePath [DefNum "x" (NumegerConst NumSuperTy ↩
               2)] [DefNum "y" (NumegerConst NumSuperTy 9)]) ↩
               )
        (NumegerConst NumSuperTy 45)))
```

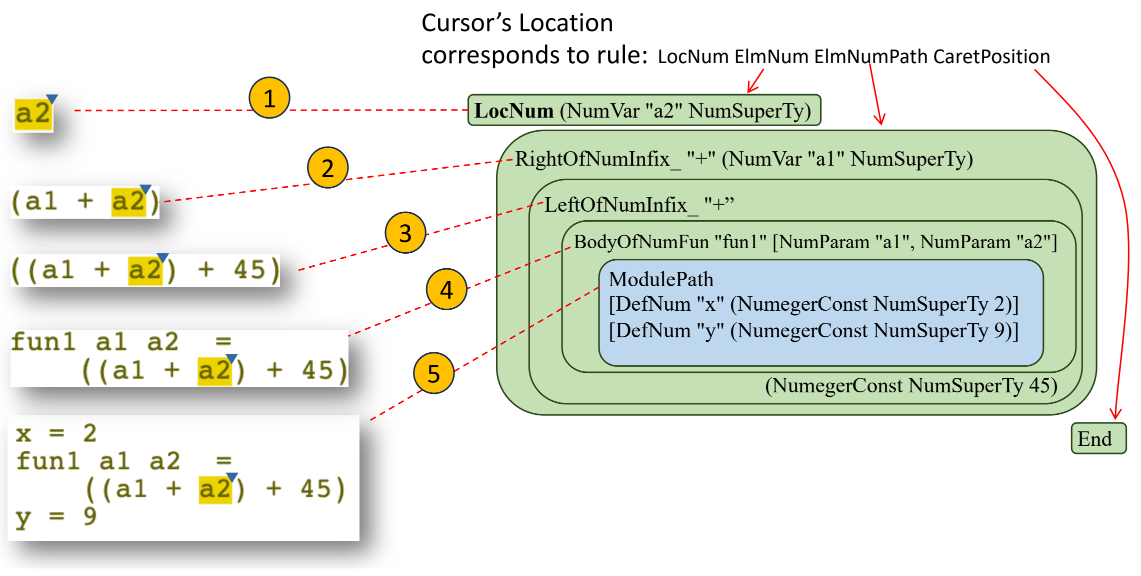

Cursor's Location
corresponds to rule: LocNum ElmNum ElmNumPath CaretPosition
a2
(a1 + a2)
((a1 + a2) + 45)
fun1 a1 a2 =
((a1 + a2) + 45)
x = 2
fun1 a1 a2 =
((a1 + a2) + 45)
y = 9
1
2
3
4
5
LocNum (NumVar "a2" NumSuperTy)
RightOfNumInfix_ "+" (NumVar "a1" NumSuperTy)
LeftOfNumInfix_ "+"
BodyOfNumFun "fun1" [NumParam "a1", NumParam "a2"]
ModulePath
[DefNum "x" (NumegerConst NumSuperTy 2)]
[DefNum "y" (NumegerConst NumSuperTy 9)]
(NumegerConst NumSuperTy 45)
End

Figure 4.2: Example of Rendering Listing 4.7 when the cursor is on a2

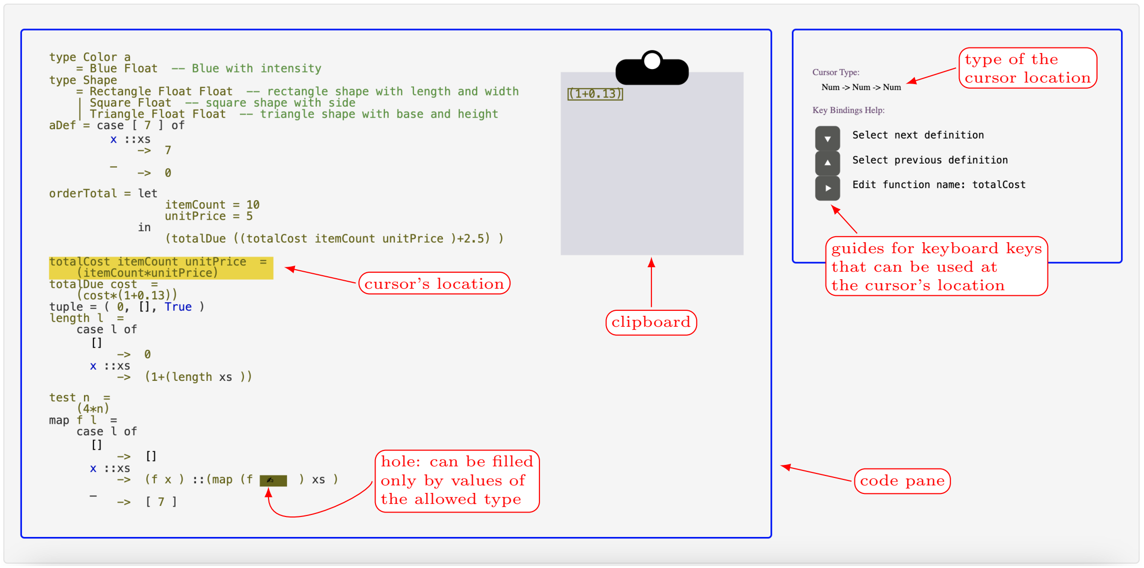
4.2.3 Discussion

During the development of ElmSr, the design of its AST prompted discussions about the trade-offs between fully-typed AST and the AST with basic types. Opting for a fully-typed AST ensures robust type safety throughout the editor's runtime, enhancing code reliability and preventing type-related errors during code input. This level of expressiveness allows for precise handling of code fragments which aligns closely with the semantics of the language and potentially enables optimizations. However, the complexities of managing expressions and definitions within a fully-typed AST pose implementation challenges and maintenance overheads.

On the other hand, adopting a basic-typed AST simplifies the structure, making it easier to navigate and manipulate for certain use cases that do not require detailed type information. With fewer specifications, the implementation complexity is reduced, resulting in a more straightforward development process and easier maintenance. However, the lack of specific type information in a basic-typed AST necessitates more extensive runtime checks to ensure type safety, potentially introducing inefficiencies and errors if not handled properly.

4.3 Visual Overview

Figure 4.3 illustrates ElmSr's graphical interface. The visual layout of our editor is divided into specific functional areas. On the left is the code pane, where users can enter and edit code. This pane highlights the current cursor location in yellow background which provides a clear visual cue for easier navigation. As shown in the figure, the entered code may include typed holes. Additionally, the code pane



```
type Color a
    = Blue Float  -- Blue with intensity
type Shape
    = Rectangle Float Float  -- rectangle shape with length and width
    | Square Float  -- square shape with side
    | Triangle Float Float  -- triangle shape with base and height
aDef = case [ 7 ] of
         x ::xs
             ->  7

         _
             ->  0

orderTotal = let
                 itemCount = 10
                 unitPrice = 5
             in
                 (totalDue ((totalCost itemCount unitPrice )+2.5) )

totalCost itemCount unitPrice  =
    (itemCount*unitPrice)
totalDue cost  =
    (cost*(1+0.13))
tuple = ( 0, [], True )
length l  =
    case l of
      []
          ->  0
      x ::xs
          ->  (1+(length xs ))

test n  =
    (4*n)
map f l  =
    case l of
      []
          ->  []
      x ::xs
          ->  (f x ) ::(map (f      ) xs )
      _
          ->  [ 7 ]
```

```
[(1+0.13)]
```

Cursor Type:
Num -> Num -> Num

Key Bindings Help:

▼	Select next definition
▲	Select previous definition
▶	Edit function name: totalCost

type of the cursor location
cursor's location
clipboard
guides for keyboard keys that can be used at the cursor's location
code pane
hole: can be filled only by values of the allowed type

Figure 4.3: ElmSr's Interface

features a clipboard that displays a list of copied items. Users can paste these items into type-compatible locations.

In the upper right corner, the editor displays the type of the code block at the cursor's current location. Below this is a guide to keyboard bindings applicable at the current cursor position, offering users a quick reference for using the various keyboard shortcuts available.

4.4 AST Navigation

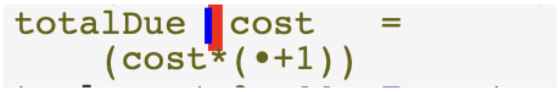
In this section, we dive into four core structural operations of our Elm editor: selection, navigation, manipulation, and deletion. Together, they help programmers interact with code in a meaningful way while maintaining type accuracy.

4.4.1 Selection

Similar to other structure/projectional editors, ElmSr's selection mechanism deviates from traditional text editors, where selection can occur freely across the text. In ElmSr, the focus is on a semantic selection. Essentially, the cursor focuses on a specific code structure — like a block or function — or its components. This type of selection offers a structural view of the code which in turn helps in a deep understanding of code anatomy. The choice of selecting the structure as a whole or drilling down to its components provides a flexible and intuitive way for young learners to interact with code elements. For example, in Figure 4.4e, the entire function body is selected. Using the arrow keys, the programmer can *select* different parts of that body expression, with each part highlighted, as illustrated in Figures 4.4f, 4.4d, and 4.4c.

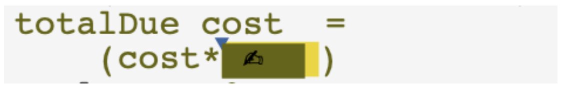
4.4.2 Navigation

The navigation in ElmSr mimics the visual layout of the code and its blocks. Instead of navigating according to the underlying tree structure, the editor enables programmers to move around as if they were looking at the code written on a page. For instance, hitting the right arrow key moves the cursor to the first child node of the currently selected subtree. This is illustarted as a transition between **state 1** and **state 2** in Figure 4.6. Pressing the up arrow key moves the cursor quickly to the other sibling node of the selected element (from **state 2** to **state 3**). If there are more than one siblings, the programmer can keep pressing the up arrow key to jump to the next siblings, and the down arrow key to jump to the previous sibling. If the expression is an infix expression, then, the user can keep pressing the right arrow key to move the caret forward in the expression(which is aligned with the direction of the code).



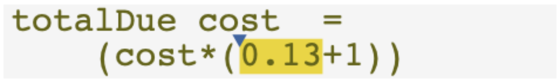
```
totalDue  cost   =
    (cost*(•+1))
```

(a) state 5



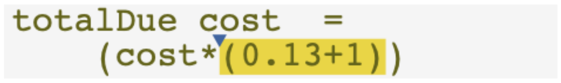
```
totalDue cost  =
    (cost*     )
```

(b) state 8



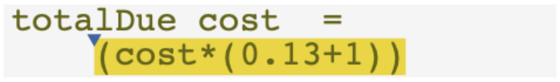
```
totalDue cost  =
    (cost*(0.13+1))
```

(c) state 4



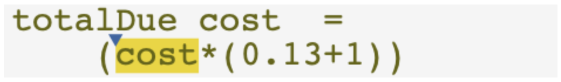
```
totalDue cost  =
    (cost*(0.13+1))
```

(d) state 3



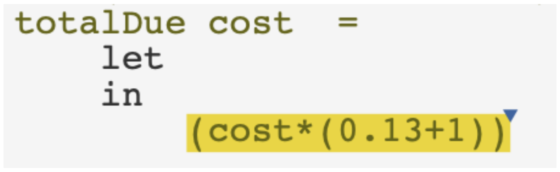
```
totalDue cost  =
    (cost*(0.13+1))
```

(e) state 1



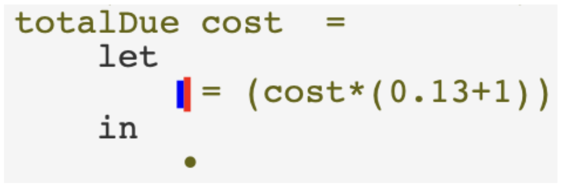
```
totalDue cost  =
    (cost*(0.13+1))
```

(f) state 2



```
totalDue cost  =
    let
    in
        (cost*(0.13+1))
```

(g) state 6



```
totalDue cost  =
    let
        = (cost*(0.13+1))
    in
        •
```

(h) state 7

Figure 4.4: Navigation and Keyboard Interactions in ElmSr Corresponding to Figure 4.6

Pressing the right arrow again further navigates down the tree, focusing on the first child of the selected node (from **state 3** to **state 4**). Conversely, pressing the left arrow key moves the cursor up the tree, returning to the parent of the current node (from **state 4** to **state 3**). However, if the cursor is on the components of an infix expression, pressing the left arrow key will move the caret in the left direction on the infix expression. This design makes it intuitive for users to navigate through their code and if needed, jump quickly to the neighboring components of each construct.

Figure 4.5 shows how continuously pressing the right arrow key moves the cursor on an infix expression. The continuous pressing of the left arrow key will return the cursor back to the initial state.

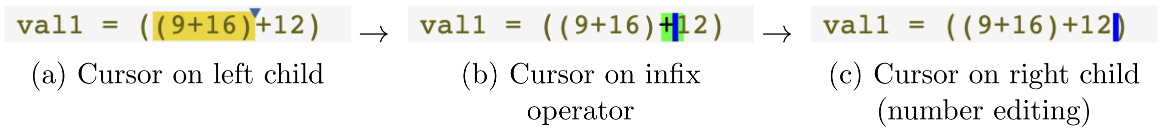


Figure 4.5: Pressing Right Arrow Key Multiple Times on an Infix Expression

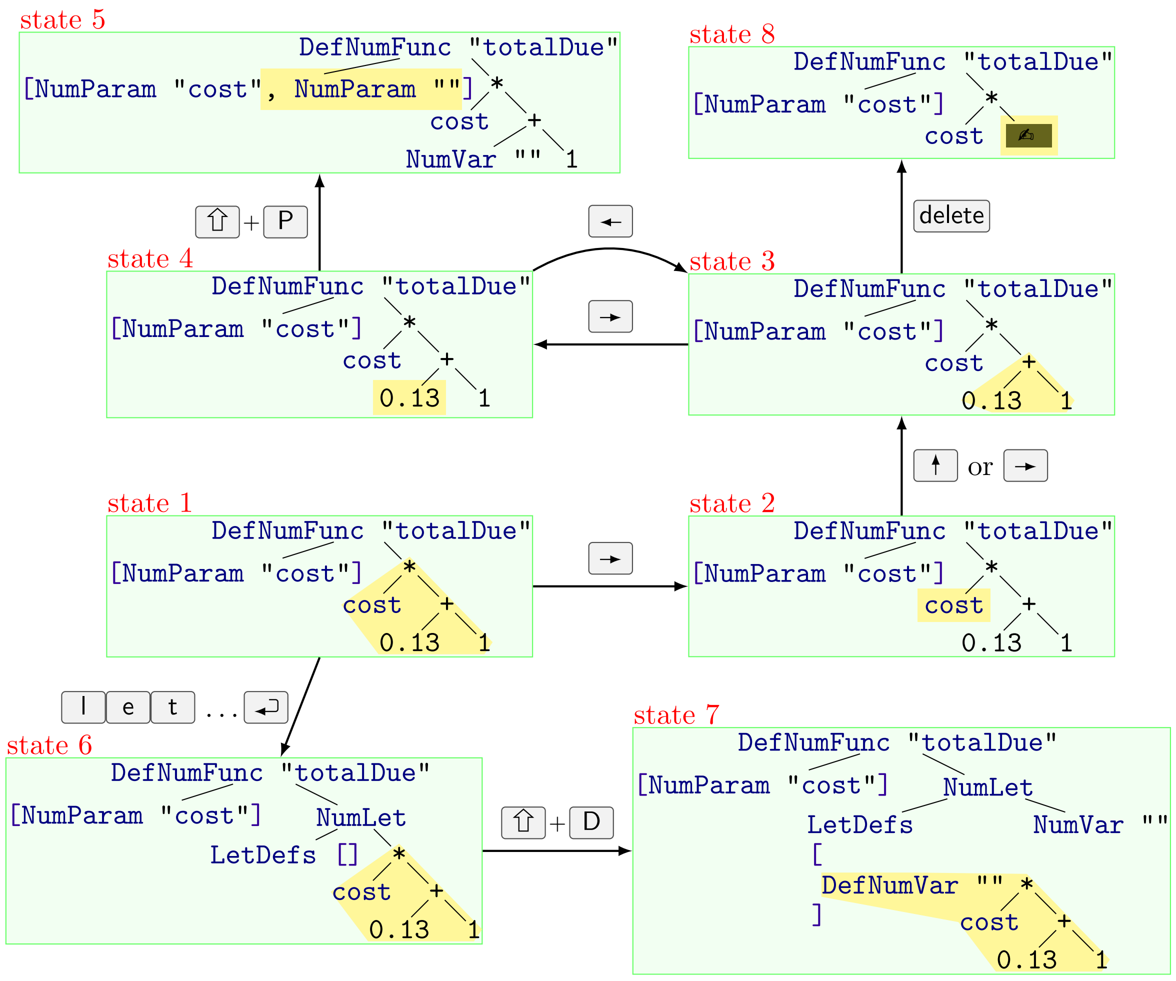


Figure 4.6: AST Navigation and Keyboard Interactions

4.5 AST Adjustment

This section delves into the operations that alter the AST of program code. A detailed discussion on expression construction and editing, particularly regarding infix expressions, is provided in Section 4.8.

ElmSr provides a wide range of functionalities for entering and modifying programming code. Users can rename variables, adjust values, and perform other tasks similar to standard text editors. Additionally, the editor facilitates the insertion of new code elements and structures via a type-sensitive drop-down menu. This allows programmers to add elements that are appropriate for the current context in the code, thereby reducing the likelihood of errors. Furthermore, ElmSr supports copy-and-paste operations through a clipboard which enables users to store code snippets in the clipboard for later use. The integration of the algebraic thinking curriculum further introduces AST manipulation operations which we will discuss in this section.

4.5.1 Deletion

When the delete key is pressed, the editor creates a 'hole' in the place of the deleted object as discussed in Section 4.2.1. This hole represents a missing element in the code and can be filled at a later time, while respecting the original type of the deleted element. This operation ensures that the type integrity of the program is maintained, and the overall structure of the program remains sound, even if certain parts of it are missing.

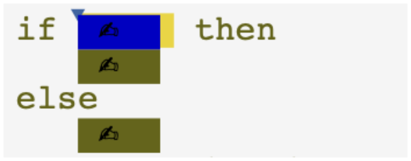
4.5.2 Undo/Redo

As discussed in Section 4.9.2, the Model component of ElmSr keeps track of the editing history through `undos` and `redos` lists. The `undos` list maintains an ordered sequence of *locations* representing the most recent changes, with the latest change positioned at the head of the list. Conversely, the `redos` list contains the reversed order of changes to facilitate redo operations. Pressing the ⌘+Z reverts the editor back to the previous state by accessing the head of the undos list, and ⌘+⇧+Z executes a redo operation.

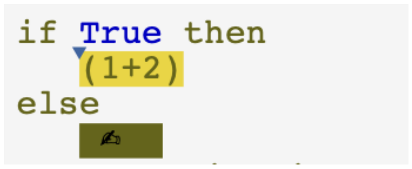
4.5.3 Copy/Paste

ElmSr provides a copy/paste functionality with a clipboard approach. Illustrating a clipboard in the editor is aimed at providing users with a robust and error-free method of copying and pasting code elements. To copy the expression at the cursor, the ⌘+C key combination can be used. This action places the copied item in the clipboard, ready for future pasting. Users have the flexibility to continue copying additional items. To paste a copied item at a specific location, users can activate the paste mode by pressing ⌘+V. Upon entering paste mode, ElmSr filters and shows a list of options suitable for the current location, based on the type of the location. Users can then select the desired item from the list to insert it into the location.

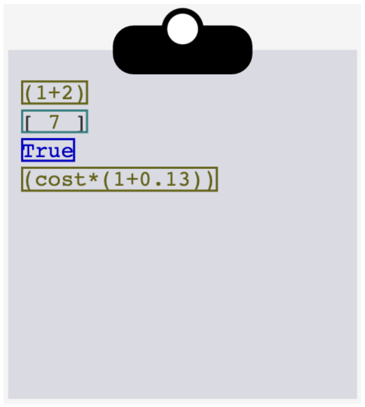
The clipboard functionality is particularly important in structure editors like ElmSr. Unlike traditional parser-based editors, where code manipulation is typically revolves around editing sequences of characters, ElmSr operates on a fully typed AST in which not every code element can be copied and pasted across different parts of the codebase. Copying and pasting code elements requires consideration of their compatibility with the target location. In this regard, the clipboard serves as a tool for managing this process by temporarily storing copied code elements, with color coding based on type, and providing users with a list of compatible options for pasting based on the type of the current location. Moreover, we think that the design of our clipboard can enhance the visual learning of novice programmers and aid in their comprehension of the type (as indicated by color coding) of program blocks. Figure 4.7 provides an illustration of how the copy/paste functionality appears in ElmSr.



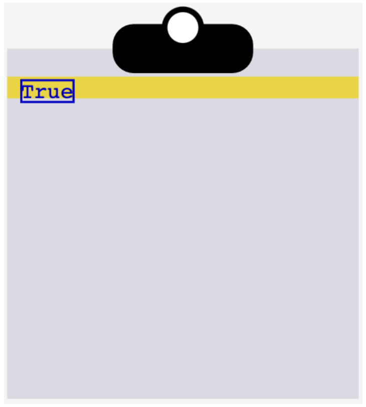
(a) `if-then-else` statement with holes



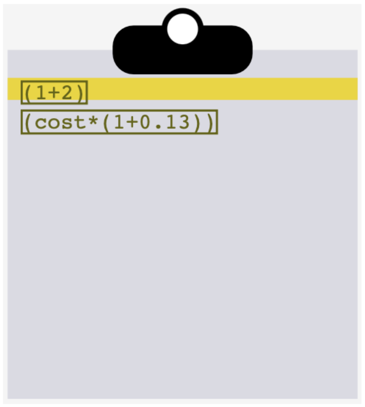
(b) after pasting into holes from clipboard



(c) list of items in the clipboard. Colors indicate the type of items.



(d) pasting at the *condition* of the `if-then-else`



(e) pasting at the *then branch* of the `if-then-else`

Figure 4.7: Copy/Paste with Clipboard

4.5.4 Drop-down Menu Generation

The ability to suggest valid type-compatible options while coding is a pivotal feature of our editor. These suggestions are presented in a drop-down menu that offers the programmer a range of possible substitutions based on the type context of the location with the aim of minimizing manual coding and preventing errors.

The drop-down menu is dynamically populated by scanning the program code to identify entities that match the type specification of the current location. In addition, fundamental Elm constructs such as `let-in`, `if-then-else`, and `case` expressions are included in the available options. Each time the drop-down menu is accessed, a buffer is displayed which allows users to input some characters related to their desired option. This buffer then narrows down the list of options based on the entered characters. Instances of the drop-down menu can be seen in Figure 4.12.

4.5.5 Supporting the Algebraic Thinking Curriculum

The editor's primary design goal is to support the Algebraic Thinking Curriculum. Through this, the editor aims to provide functionalities that emphasize this curriculum's principles, such as converting constants to function parameters (extract parameter) or abstracting expressions into definitions (extract definition) as discussed in Section 3.4.

As depicted as the transition between **state 4** and **state 5** in Figure 4.6, within a function body, the current cursor's location can be converted into a function parameter by pressing ⇧+P. Subsequently, the cursor is relocated to the designated position for entering the new parameter's name (Figure 4.4a .) Following this, the user can input the desired name, which replaces the placeholder *dot* name. The editor then searches the code to locate instances of the edited function and adds the new argument to the appropriate calling sites. If the expression subjected to the "extract parameter" action contains variables that are defined within the function, those variables are replaced with holes in the expression and it is then moved to the calling sites as an argument. Otherwise, the expression, as is, will be used as the argument passed to the function call.

Furthermore, expressions can be transformed into definitions within the immediate or outer scope by pressing the ⇧+D key combination, as shown in Figure 4.6 by the transition from **state 6** to **state 7**. This action declares the expression as a new definition, either at the module level or within the immediately enclosing *let-in* construct. Additionally, users can wrap an expression into a *let-in* construct using the drop-down menu. To do so, the user needs to select the expression and begin typing "let in"; the drop-down menu will then offer the option for selection (transition from **state 1** to **state 6**.)

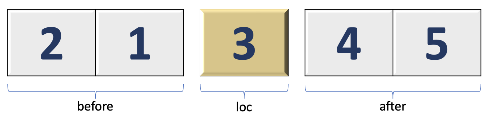
4.6 Number and Name Editing

This section explores how ElmSr models the editing of numbers and names (e.g., string names of variables, functions, etc.) using lists in a functional manner. In a pure functional language, a list serves as an abstract data type that stores items of the same type in a given order. A list, among other use-cases, can hold digits in a number or characters in a name. Efficient navigation within lists, both forward and backward, is crucial for optimizing the overall system performance. To achieve this, ElmSr employs the list zipper data structure to model lists effectively.

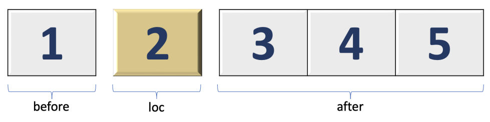
4.6.1 List Zipper

A list zipper maintains a focused element along with its surrounding context. The cursor's focus on a list is modeled as a focused element **loc** on which the cursor is and two Elm lists **before** and **after**. The *before* list stores the elements to the left of *loc* arranged in reverse order, and the *after* list holds the elements to the right in their original order. Storing elements in reverse order in the before list adheres to Elm's way of extracting the first element of a list by decomposing it into the head (first element) and tail (the rest of elements).

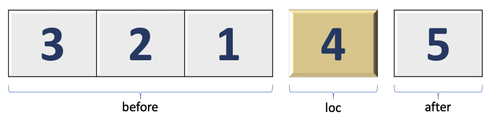
Figure 4.8 shows an example of moving forward and backward on the following list of integer elements: `[1, 2, 3, 4, 5]`. The focus is initially on 3 (Figure 4.8a). Moving the cursor to the left (Figure 4.8b) will relocate the cursor to element 2 which was the head of *before*. Note that *before* is stored in reverse order. In the meantime, 3 (current cursor) is prepended to the *after* list. As illustrated in Figure 4.8c, a forward move from the cursor originally on 3 prepends it to the *before* list and, as the new cursor, extracts the head of the *after* list.



(a) Original cursor



(b) Move to the left of the original cursor



(c) Move to the right of the original cursor

Figure 4.8: Demonstration of a list zipper for list `[1, 2, 3, 4, 5]`

4.6.2 Number Editing

Number editing in ElmSr works similarly to a typical text editor. The editor enters this mode either by pressing the right arrow key on a number, or just by starting to type a number on a hole, or a location that can accept a number as the left operand of an infix operator (Like in Column 2 of Figure 4.12).

In this mode, as depicted in Figure 4.9, digits are treated as individual integers forming a list, with caret movement and digit manipulation facilitated by a list zipper.

Our implementation of the number supertype, as discussed in Section 4.7 is employed in this mode which initially treats the number as an integer. However, pressing the . key transitions it to the mode of accepting floating-point numbers, allowing for the sequential addition of digits to the right of the decimal place.

It should be noted that, even while the editor is in number editing mode, users can still type characters and select from available functions and variables via the drop-down menu.

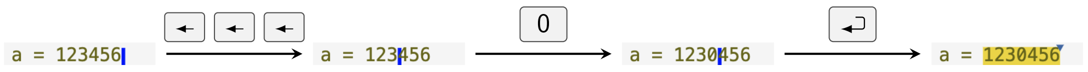


Figure 4.9: Number editing.

4.6.3 Name Editing

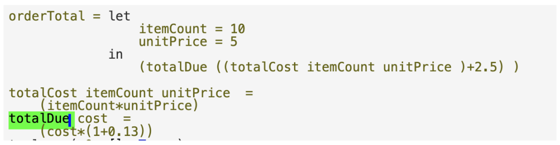
For entering names of variables, functions, and custom types, ElmSr employs a method similar to number editing which uses the list zipper for efficient manipulation. In particular, the `NameEditState` keeps track of the characters preceding and following the caret position, along with a set of existing names within the current scope:

Listing 4.9: Name Edit State

```
type NameEditState
   = NameEntry (List Char)  -- ^ before
```

```
                    (List Char)   -- ^ after
                    (Set String)  -- ^ names in scope
```

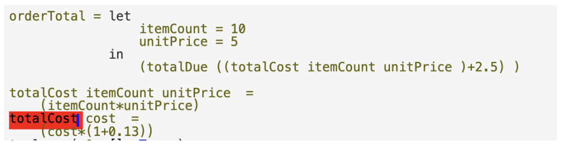
Figure 4.10 illustrates a scenario of editing the name of a function called `totalDue`. When the cursor is on a definition, pressing the → key, transitions the editor into the name edit state (Figure 4.10a). When a user selects a name already present in the current scope (`totalCost` in this case), the background of the name editing area changes to red, indicating a potential conflict (Figure 4.10b). On the other hand, if the chosen name is valid, the background switches to green (Figure 4.10c). In this case, the user can exit this state by pressing the ↵ key to commit the changes. Furthermore, any modifications made during name editing propagate to all occurrences of the specific definition throughout the codebase. As Figure 4.10d shows, the new name `totalPayable` replaces the old name in the function application inside the `let in` construct.



```
orderTotal = let
                 itemCount = 10
                 unitPrice = 5
             in
                 (totalDue ((totalCost itemCount unitPrice )+2.5) )

totalCost itemCount unitPrice  =
    (itemCount*unitPrice)
totalDue cost  =
    (cost*(1+0.13))
```

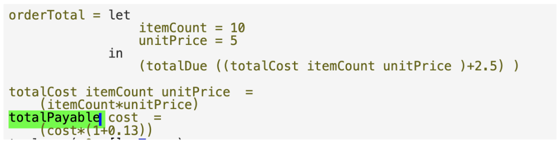
(a) After pressing → on a definition



```
orderTotal = let
                 itemCount = 10
                 unitPrice = 5
             in
                 (totalDue ((totalCost itemCount unitPrice )+2.5) )

totalCost itemCount unitPrice  =
    (itemCount*unitPrice)
totalCost cost  =
    (cost*(1+0.13))
```

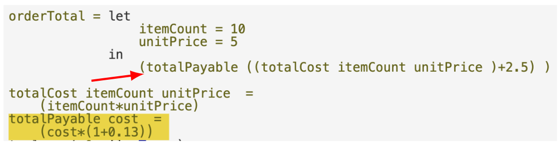
(b) Entering a name already in the scope



```
orderTotal = let
                 itemCount = 10
                 unitPrice = 5
             in
                 (totalDue ((totalCost itemCount unitPrice )+2.5) )

totalCost itemCount unitPrice  =
    (itemCount*unitPrice)
totalPayable cost  =
    (cost*(1+0.13))
```

(c) Entering an acceptable name



```
orderTotal = let
                 itemCount = 10
                 unitPrice = 5
             in
                 (totalPayable ((totalCost itemCount unitPrice )+2.5) )

totalCost itemCount unitPrice  =
    (itemCount*unitPrice)
totalPayable cost  =
    (cost*(1+0.13))
```

(d) After pressing ↵ key

Figure 4.10: A name editing scenario.

4.7 Numerical Supertype

Consider an example where the cursor navigates to an integer value deep within ElmSr's AST. The value, and all the corresponding nodes above and adjacent, adhere to integer operations, given the fully typed nature of the AST. When the user decides to modify the integer to a float (by pressing . in integer editing), the involved nodes must be transitioned into floats to uphold the correctness of the type system.

To handle cases where an integer may transition into a float, or vice versa, we introduce a datatype `ElmNumType` to represent categories of number types. `ElmNumType` has been designed with three distinct categories: `NumSuperTy`, `IntConcreteTy`, and `FloatConcreteTy`. The first one represents an uninstantiated numerical supertype that acts as a placeholder that could transform into either an integer or a float based on user interactions. The last two, `IntConcreteTy` and `FloatConcreteTy`, denote a concrete integer type and a concrete float type, representing an explicit integer value and explicit float value respectively, and offering no further flexibility.

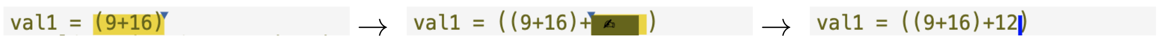
The `ElmNumType` construct acts as a flexible interface for type transitions which allows dynamic type modifications within the AST in response to user interactions, all while preserving type safety. This design choice of a supertype that inherently accommodates both integer and float types aligns with Elm's immutable data structure paradigm and eliminates the need for propagating changes throughout the AST.

4.8 Expression Construction

The primary focus of ElmSr is to facilitate seamless editing of expressions and enable easy, error-free insertion and manipulation of programming constructs, particularly infix operators. The aim is to mimic the interactions one would expect from a text-based (parser-based) editor.

Inserting new expressions into an empty hole works much like a text editor, where the programmer inputs characters sequentially or navigates within the expression to modify different parts of it. Users can extend existing expressions by incorporating infix operators such as addition (`+`), subtraction (`-`), division (`/`), and integer division (`//`). ElmSr employs visual cues to enhance the user's understanding of the expression's context and structure. For instance, subexpressions are automatically parenthesized after insertion. Furthermore, as the user moves the cursor over expressions, subexpressions are highlighted and a small blue triangle indicating the caret's position appears. While highlighting an expression signifies the cursor is on it, extending that expression, whether by appending or prepending, requires the user to know the exact caret location, similar to text-based editors. The following representation of an expression illustrates an example with the cursor on `2`, and the caret at the beginning: `1 + 2` , and the caret at the end of the same subexpression: `1 + 2` .

When intending to append a new expression to the cursor, users must continue pressing the right arrow key until the caret reaches the end of the expression where they wish to append. This caret movement is akin to that of text-editors. Once the caret is in position, users can begin typing the desired operator to extend the expression. A hole is inserted after the entered operator, prompting the user to input the right operand. Figure 4.11 illustrates this process.



```
val1 = (9+16)
```
$\longrightarrow$
```
val1 = ((9+16)+ )
```
$\longrightarrow$
```
val1 = ((9+16)+12)
```

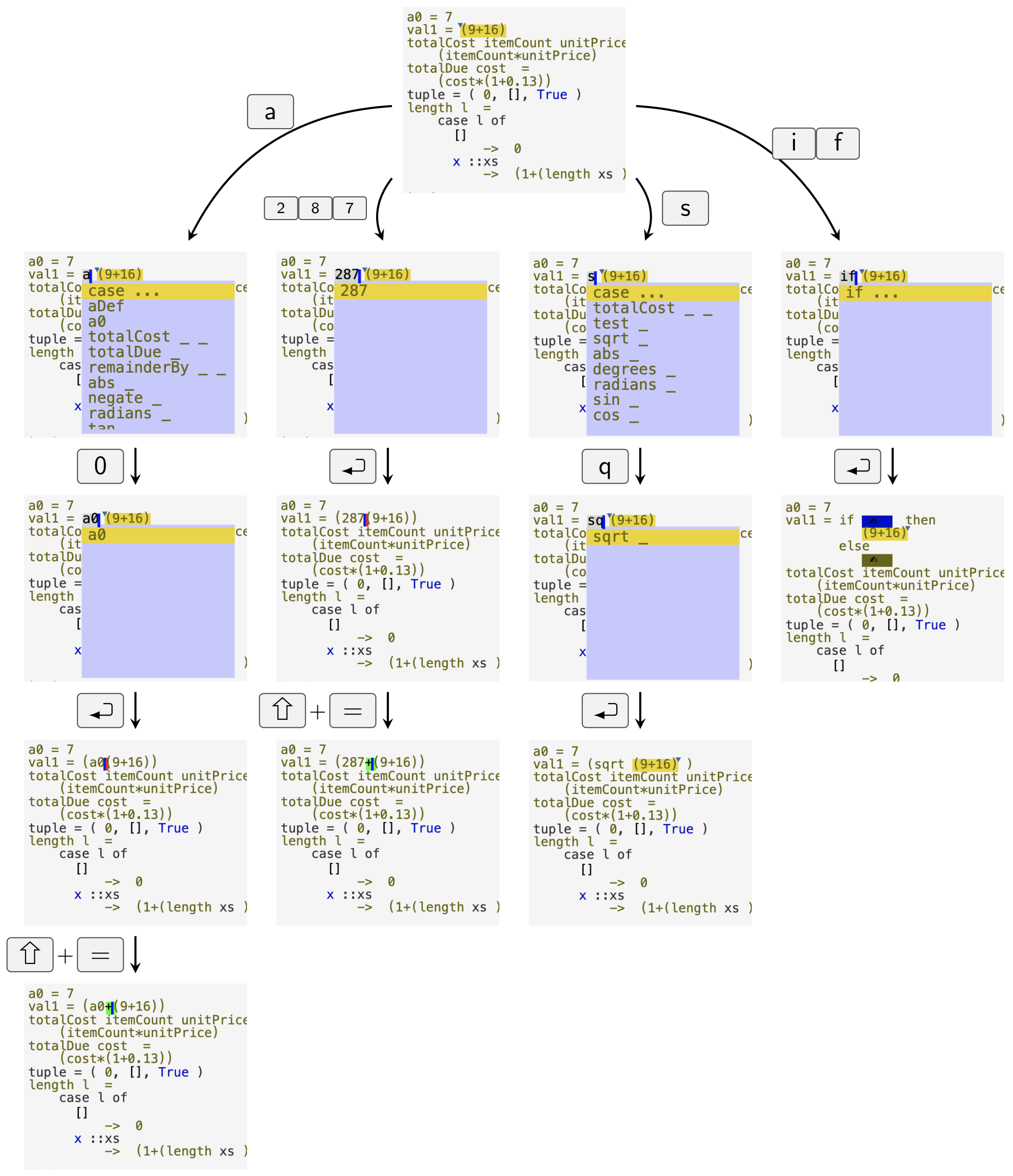
(a) Caret at the end (b) After pressing `+` (c) After pressing `1` and `2`

Figure 4.11: Appending to expression

Conversely, in the prepending scenario, users have the flexibility to initiate expression extension by entering either the operator or the operand first. If starting with the operator, a hole is inserted before the operator (as the left operand), and the cursor waits on the operator. The user can then navigate the cursor to the hole by pressing the left arrow key. This way of extending is similar to many traditional structure editors requiring the user to enter the operator first (sub-tree's root) to adhere to the tree structure. If the user opts to start with the operand first, the drop-down menu assists the user in proceeding. This operand can be a value allowed at that hole (like an integer) or an already defined variable. In either case, the user inputs their desired option in the drop-down buffer, and the drop-down filters accordingly. By selecting an option, the cursor moves to the operator and awaits user input. Column 1 of Figure 4.12 depicts a scenario where an already-defined variable `a0` is used as the left operand. In Column 2, the figure demonstrates entering the number `287` as the left operand. Here, the drop-down menu only displays the option to enter the number.

Besides appending or prepending, ElmSr also allows users to wrap an expression in a function call, a let-in statement, a case expression, or an `if-then-else` statement. Users simply start typing what needs to be entered and then select from the provided drop-down menu. For example, by typing `if` and selecting the option in the drop-down menu (Figure 4.12, column 4), the selected expression becomes the 'then' branch of the inserted `if-then-else` statement.

To make a subexpression the argument of a function, users need only enter the name of the function partially, and the drop-down will display matching function names for selection. The function can be either a user-defined function in the scope or a basic Elm function available in the scope. For instance, by typing "sqrt" and



```
a0 = 7
val1 = (9+16)
totalCost itemCount unitPrice
    (itemCount*unitPrice)
totalDue cost  =
    (cost*(1+0.13))
tuple = ( 0, [], True )
length l  =
    case l of
      []
        ->  0
      x ::xs
        ->  (1+(length xs )
```

Figure 4.12: Prepending to expression. Different scenarios starting from the top figure.

selecting the drop-down option (Figure 4.12, column 3), the expression becomes the argument of the function. If the function requires more than one argument, the expression is set as the first argument of the function call, and a hole is added in place for the remaining arguments.

In addition to numbers, infix operators can be added between the digits of an already entered number while the editor is in the number edit mode described in Section 4.6.2. In this case, the two parts of the number are converted into the left and right operands of the entered operator (Figure 4.13). Moreover, if the operator is removed, the digits are merged to reconstruct the number.

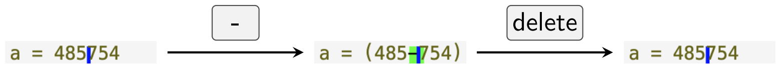


Figure 4.13: Number editing mode to infix conversion

4.9 Implementation Details

In this section, we will delve into details of the architecture of ElmSr and the implementation of its components.

4.9.1 Bootstrapping

Bootstrapping in the context of programming languages or compilers refers to the process of writing a compiler (or an interpreter) for a language in the same language.

The use of Elm for both the implementation of the editor and the target language to be edited has many advantages. Elm's strong type system and inherent functional nature contribute to the editor's robustness and correctness. The language's declarative style and reactive model fit seamlessly with the structure editor's objectives, making it an excellent choice for this purpose.

Beyond the technical benefits, this bootstrapping approach also creates a great development ecosystem. The editor's implementation in Elm means that anyone looking to maintain or extend it only needs proficiency in Elm itself. This single-language focus simplifies the learning curve for new contributors and facilitates the editor's continual growth and improvement.

Essentially, by using Elm to construct an editor for Elm, we're not only creating a powerful tool but also reinforcing the capabilities and robustness of the Elm language itself.

4.9.2 Architecture

The high-level architecture of ElmSr adheres to the Elm Architecture itself, which is thoroughly explained in Section 3.1. In this section, we delve into the unique features of ElmSr in terms of the Elm architecture.

Model

The `Model` represents the state of the entire ElmSr application, providing an up-to-date snapshot of all the necessary components in our editor. The main component of our Model is the zipper structure that maintains information regarding the cursor's current position within the AST. The state of the keyboard to track keypress events is also maintained in the Model which is particularly important for an editor like ElmSr, where edit operations are done through keyboard interactions. Moreover, the Model also contains fields for managing the screen display, key bind menu which stores the current state of a hint menu for keyboard shortcuts, and a history of undo and redo operations which preserves past cursor locations, enabling users to easily revert or repeat changes.

View

The `view` function takes the current model and generates the JavaScript and HTML to be displayed in the browser. It generates a visual representation of the current Elm code, including not only the current location of the cursor highlighted, but also any additional elements such as holes, key bind menu and the drop-down menu, if needed.

Given the nature of our editor, it is important to note that while the View generates HTML, the HTML is structured to represent the code's structure, not its text. This way, the users are interacting with the structure of the Elm code, not its textual representation. This is what makes our editor a structure editor. Currently, we opt to render the code into a conventional Elm code format. However, there is a potential for more innovative projections that might potentially be easier to understand for students. Although we haven't explored these novel visualizations yet, the structure-based approach of our editor lays the groundwork for these potential enhancements in the future. Details of editor projections can be found in Section 4.2.2.

Update

The `update` function is where the application's logic lives. Every keyboard interaction triggers a message (using `subscriptions`) that gets processed by the `update` function to modify the `Model`, which in turn updates the `view`.

The `update` function takes the current model and a message, and produces a new model state. Thus, the `update` function will handle all the keyboard interactions that trigger changes in the code structure, or editor's state.

For example, pressing the down arrow key on a definition moves the cursor to the next definition, or if the user wants to perform a refactoring operation, like extract parameter, this action would be initiated by ⇧+P. When this command is received, the `update` function will execute the refactoring operation on the current model (current `location` on AST), and produce a new model that reflects these changes (for the extract parameter case, the `location` of the cursor is moved to the name of the newly created parameter and waits for entering the name of the parameter).

Subscriptions

This function monitors essential external events required for ElmSr's operation. For example, it tracks window resize events, as well as keydown and keyup events. Each event captured triggers the generation of a corresponding message, subsequently processed by the `update` function.

4.9.3 Components

Our editor's implementation is partitioned into distinct modules. By isolating specific responsibilities within individual modules, we ensure that each component remains focused, concise, and independent, which enables efficient debugging, testing, and enhancement. This section offers an overview of each of these modules.

Types.elm

The `Types.elm` module serves as the central repository for the data types essential to the AST used in our editor. It categorizes the Elm types into numerical types, booleans, lists, custom type definitions, and typed holes, as embodied in the `ElmType` data type. Additionally, it defines the structure of an Elm module with the `ElmModule` type, which represents a module as a list of various Elm definitions.

More than just a mere type collection, this module also defines the `Model` type which holds a comprehensive snapshot of the editor's current state, reflecting both the user's actions and the editor's responses. Specifically, it retains the current location in the cursor, along with the history of changes through undos and redos lists. Additionally, it maintains flags indicating the up/down state of several essential keys, such as control, shift, alt, and so on. The information kept in this structure ensures that the editor has a real-time awareness of the user's interactions and the evolving state of the program code.

Cursor.elm

The `Cursor.elm` module is central to the navigation and manipulation of the editor's AST. The module primarily offers a suite of cursor functions designed for the traversal of the AST. Functions such as `cursorUp`, `cursorDown`, `cursorLeft`, and `cursorRight` facilitate movement within the tree with the help of our zipper structure. A unique aspect of our design is the directionality of tree growth, which is left to right. In this context, moving "left" corresponds to moving toward the (sub)tree's root, while moving "right" directs us toward the leaf nodes. Although conventional tree processing might involve navigation from a node to all its children, the nature of our editor emphasizes level-to-level traversal, enabling jumps between child nodes as required.

Additionally, this module offers utility functions like `addParameter`, which transforms an expression into a new parameter for a function, and `addDefinition`, which incorporates an expression as a fresh definition within the current context.

Eval.elm

The `Eval.elm` module has the primary responsibility of evaluating the definitions and expressions within the program code module. Its design ensures that users can instantly view the evaluated value for any given cursor location within the editor. For example, if the cursor is on the definition `e = (3 * (4 + 5))`, the evaluation will be `e = 27`.

A key component of this module is the "Stack", which maintains mappings of variable names to their corresponding values, be it integers, floats, booleans, and lists of items. This stack operates similarly to a function stack but abstractly. Although it is inherently recursive due to its nested scopes, this recursiveness is subtly masked to make its operation straightforward.

This module has a suite of evaluation functions, each tailored to a specific type in our AST. For example, the function `evalNum` is invoked when the cursor's location is of type `Num`.

Rename.elm

The `Rename.elm` module is dedicated to a specific, yet crucial functionality within our editor: renaming variables and definitions. Whenever there is a need to modify the name of a definition or a variable, this module steps in to ensure that all *uses* of that particular variable within a specific scope are updated accordingly.

To provide a glimpse into its operations, consider the `renameVarInNum` function as defined in Listing 4.10 (only first few lines are shown here). It inspects locations of the `Num` type to determine if they make use of the old variable name, and if so, it replaces them with the new name. As can be seen in Listing 4.10, if the examined node (`ei`) is a numeric constant, it remains unchanged. However, if it is a numeric variable (`NumVar`) bearing the old name, the function returns a new variable with the updated name. For binary operations (e.g., `NumAdd`), the function invokes itself recursively with the left and right operands to ensure all uses of the `oldName` are renamed throughout the depth of the operation.

The module has a collection of renaming functions tailored for different types and structures, such as booleans, renames in case matches, lists, and path. This ensures that name changes are propagated accurately throughout the entire program code, preserving the integrity and correctness of the program.

Listing 4.10: An excerpt of function `renameVarInNum`

```
renameVarInNum oldName newName ei =
    case ei of
        NumegerConst _ _ -> ei
        FloatConst _ _-> ei
        NumHole -> ei
        NumVar name ty -> if name == oldName then
                              NumVar newName ty
                          else
                              ei
        NumInfix op left right ->
            NumInfix op (renameVarInNum oldName newName left)↩
                (renameVarInNum oldName newName right)
```

Overlaps.elm

The `Overlaps.elm` module is the component of our system that ensures variable name uniqueness within a given scope. It is designed to produce sets of variable names already present within the scope, thereby preventing naming collisions when introducing or renaming elements in the code.

The core utility of this module lies in its `overlaps*` functions. These functions, when invoked, return a set of variable names that are currently active within a given scope. An empty set denotes the absence of any variables in the examined scope.

The functionality of the Overlaps module is closely related to the Cursor module. When a user intends to alter a name, be it of a definition, function, custom type, or any other construct, functions from the Overlaps module are called upon from the Cursor Module. They generate a set of pre-existing names within the scope, providing an immediate reference to names that should be avoided by the user. The use of this *set* is explained in Section 4.6.3.

Overlaps works similar to the Rename module. Its helper functions are recursively invoked to delve deep into the path, enabling a comprehensive extraction of names from all nested levels of the scope.

Scope.elm

This module works similarly to the `overlaps` module, but it prepares the list of all available options within the scope of the current location. This list is then used to populate the options of the drop-down menu.

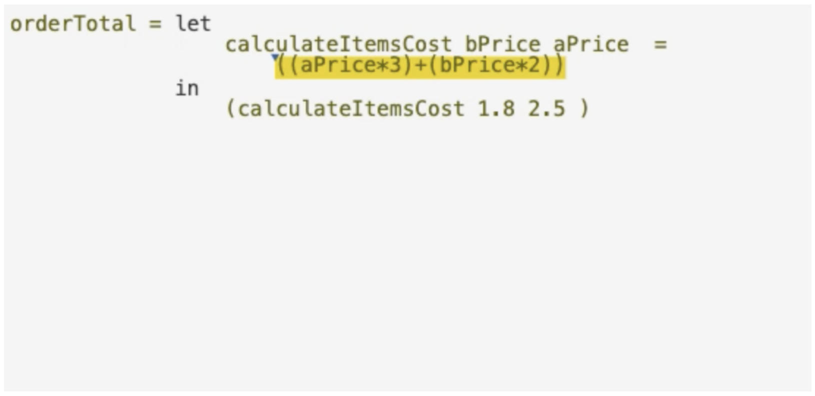
Chapter 5

Results

In Chapter 4 we explained the ElmSr editor and its internals. In this chapter we report that, as measured by keystrokes, ElmSr is more efficient than a conventional text editor at creating a function of moderate complexity and a lengthy expression.

5.1 Experimental Setup

Our experiments focused on comparing the efficiency of ElmSr with VS Code, a traditional text-based coding environment. The objective is to assess the keystroke counts required to implement an example code snippet and a long expression in the Elm programming language using the development approach taught in the algebraic-thinking curriculum. This is a structured approach, with much in common with structured approaches to program creation used in every programming language since the concept was introduced by Dijkstra, see [12]. The important point is that program editing is never just about typing out the program linearly, because everything we teach students, particularly code reusability acheived by abstraction in the form of



```
orderTotal = let
                calculateItemsCost bPrice aPrice  =
                    ((aPrice*3)+(bPrice*2))
             in
                (calculateItemsCost 1.8 2.5 )
```

Figure 5.1: Test Task 1, Stage 1

function-centric programming, leads to non-linear thinking and editing.

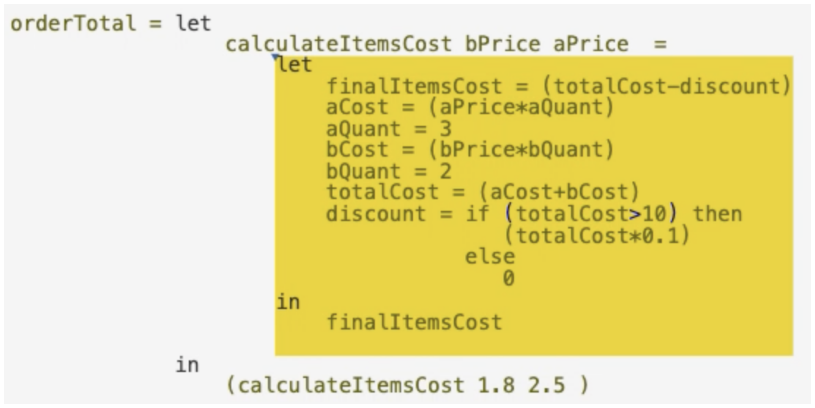
The first coding example is a practical demonstration of the algebraic thinking curriculum on how to transform a basic expression into a structured code snippet. The code starts with this initial expression: `2.5*3+1.8*2` that computes the total cost of purchasing `3` apples and `2` bananas with unit prices of `2.5` and `1.8`, respectively.

In the first stage, the expression is wrapped in a `let in` construct. Subsequently, it is converted into a function, with the prices of apple and banana declared as its parameters. Figure 5.1 shows the resulting code in ElmSr.

In the next stage, the function body is further wrapped in another `let in` construct and some local bindings are introduced to enhance code clarity. Then, a discount variable is defined which involves a conditional statement as shown in Figure 5.2.

In the final step, the body expression of the first `let in` is transformed into a function that accepts a `taxRate` parameter and calculates the total cost after applying taxes. The final code as displayed in ElmSr is depicted in Figure 5.3.

To further analyze the efficiency of code input in ElmSr, we designed a second test task, as depicted in Figure 5.4. This task involves inputting the expression

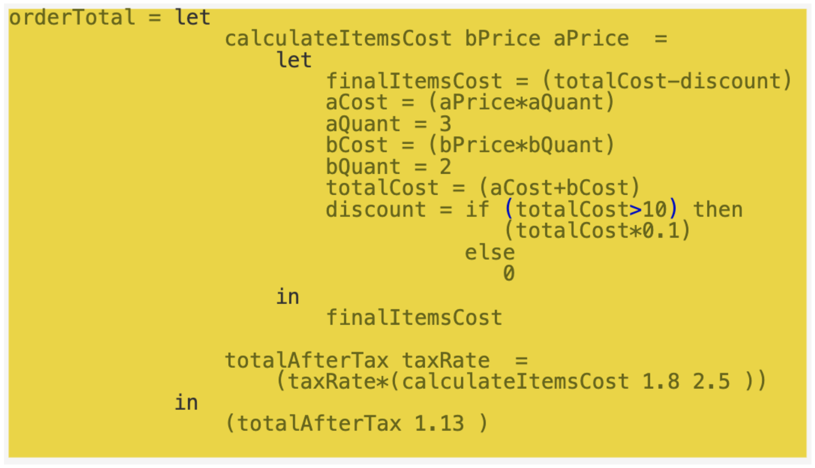


```
orderTotal = let
                 calculateItemsCost bPrice aPrice  =
                     let
                         finalItemsCost = (totalCost-discount)
                         aCost = (aPrice*aQuant)
                         aQuant = 3
                         bCost = (bPrice*bQuant)
                         bQuant = 2
                         totalCost = (aCost+bCost)
                         discount = if (totalCost>10) then
                                        (totalCost*0.1)
                                    else
                                        0
                     in
                         finalItemsCost

             in
                 (calculateItemsCost 1.8 2.5 )
```

Figure 5.2: Test Task 1, Stage 2

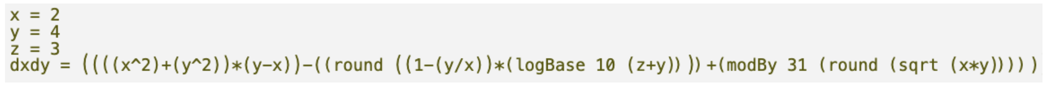
represented by the definition dxdy. While this expression does not correspond to any real-world application, it is representative of the type of code commonly written by our learners when engaging with complex graphics. It is important to note that this test is purely a linear coding exercise and does not involve the extraction of definitions or parameters.



```
orderTotal = let
                 calculateItemsCost bPrice aPrice  =
                     let
                         finalItemsCost = (totalCost-discount)
                         aCost = (aPrice*aQuant)
                         aQuant = 3
                         bCost = (bPrice*bQuant)
                         bQuant = 2
                         totalCost = (aCost+bCost)
                         discount = if (totalCost>10) then
                                        (totalCost*0.1)
                                    else
                                        0
                     in
                         finalItemsCost

                 totalAfterTax taxRate  =
                     (taxRate*(calculateItemsCost 1.8 2.5 ))
             in
                 (totalAfterTax 1.13 )
```

Figure 5.3: Test Task 1, Final Stage



```
x = 2
y = 4
z = 3
dxdy = ((((x^2)+(y^2))*(y-x))-((round ((1-(y/x))*(logBase 10 (z+y)) ))+(modBy 31 (round (sqrt (x*y)))) )
```

Figure 5.4: Test Task 2

5.2 Data Collection

The keystrokes necessary to accomplish the tasks in ElmSr were counted, and labelled according to sub-tasks such as extracting a definition (`mk def`), extracting a parameter (`mk param`), entering if-then-else statement (`if-then-else`), utilizing `let in` constructs (`let in`), entering literals like names or numbers (`name/number`), typing operators (`operator`), function application (`func App`), and navigating through the code (`navigation`). Each keystroke in each of these categories is recorded for further analysis.

Duplicating the same examples, using the same structured appraoch, with the same ordering of steps, in VS Code is challenging due to its lack of native support for extracting parameters or definitions. We have used Elm Plugin for VS Code which provides essential features such as error detection and format-on-save functionality as well as jumping to definitions. However, as of April 2024, it does not support the advanced editing operations most helpful for our recommended structured approach to creating programs.

In our experiment, we assumed that the user performing the tasks indicated above, enters no typos which would result in syntax or type errors while *manually* abstracting definitions, parameters, etc. For comparison, we recorded the keystroke counts in each sub-task category using VS Code.

5.2.1 Considerations

In the following, we list important considerations made during the data collection process.

- Key combinations such as ⇧ + D or capital letters are counted as two keystrokes.
- In VS Code, we employed what we believe to be the most efficient method available for each step without using the mouse. For instance, we utilized ⌘ + L to select the entire line containing an expression, ⌘ + ← to navigate to the beginning of the line, and used word selections.
- Entering a `let in` in VS Code involves typing "let" with a space after it and pressing the return key, triggering auto-completion. In ElmSr, wrapping an expression in a `let in` requires typing "let" and pressing the return key. Thus, the operation requires 5 and 4 keystrokes, in VS Code and ElmSr, respectively.
- In ElmSr, typing the full name of a function or variable is not necessary when using it, as the drop-down menu always offers suggestions based on context. However, we observed that this is not always the case for VS Code (with the Elm plugin), as sometimes names in the scope are not offered in the drop-down menu.
- In ElmSr, navigating between operands of an infix operator can be done in two different ways: using the text-like left/right arrow key navigation or the up/-down arrow keys to jump between operands. We always selected and recorded the fastest method available.

- In ElmSr, inserting an `if-then-else` requires typing "if" and pressing the return key, after which the necessary keywords and placeholders are automatically inserted. The navigation needed to jump to the holes is recorded in the `navigation` category.

- In VS Code, the operation of extracting a definition involves multiple steps as explained above. Therefore, the keystroke count for each `mk def` operation includes selecting the expression, cutting it, navigating to the desired location for the definition, pasting the expression, and typing an "=". Additionally, a `mk def` operation requires typing the name of the new definition twice (once to replace the expression and once as the name of the new definition), unless the dropdown menu provides the second name, in which case only one name entry is needed. Similarly, extracting a parameter (`mk param`) involves keystrokes for selection, cut, navigation to the new parameter location, navigation to paste at the function call site, and finally pasting. Both operations need only 2 keystrokes in ElmSr.

- ElmSr automatically wraps sub-expressions in parentheses, and hence there is no need for manual entry of them. In contrast, VS Code requires manual insertion of only the opening parenthesis (the closing parenthesis is inserted automatically), if needed, which is recorded as two keystrokes for ⇧+9.

- For function applications such as `round`, `modBy`, and `sqrt`, we recorded three keystrokes for the filtering of the dropdown menu and an additional keystroke for the return key to select the function from the menu. The function `logBase` requires five keystrokes in total.

- In VS Code, a space is required between the function name and the first argument in function applications when the argument is not parenthesized, as in `round 10.5`. However, when the argument is parenthesized, like in `round(x)`, we do not count the space, as it is not required. Nonetheless, we do count spaces that enhance code readability. In contrast, ElmSr does not require a navigation operation for function applications, because the cursor automatically moves to the placeholder for the first argument after a function is selected from the menu.

5.3 Results

Figure 5.5 displays the total keystroke counts for each category in both ElmSr and VS Code for the first task. The figure clearly demonstrates that ElmSr generally requires fewer keystrokes to accomplish the same sub-task. For instance, generating the structure for the only conditional statement in the task requires only 3 keystrokes in ElmSr: typing “if” prompts the drop-down menu with the `if` option highlighted, and pressing the return key inserts the statement at the cursor’s location. In contrast, the same operation needs 15 keystrokes in VS Code, including the keywords and whitespaces. ElmSr falls short only in the navigation category. Given its handling of the AST, it needs more navigation commands, especially for multi-line constructs. However, VS Code (dealing with text), allows for more direct cursor movement using simple arrow key commands to reach the desired location.

Figure 5.6 presents the total number of keystrokes required for the second task. It is evident that the number of keystrokes necessary for function application and operator entry in ElmSr is identical to that in VS Code. However, ElmSr *sometimes* needs additional keystrokes for selecting variable names from a dropdown menu. Specifically, after typing a variable name (e.g., `x` or `y`), users must explicitly press return key to confirm the selection from the menu if the variable concludes a sub-expression. For instance, when entering the expression `x*y`, the variable `x` is automatically highlighted in the menu and inserted in the location, upon typing 'x', followed by the `*` operator. However, to enter `y`, the user must press the return key after typing 'y' to finalize its selection.

Furthermore, as observed in the results from the first task, ElmSr typically requires more keystrokes for navigation. However, ElmSr does not require the user to manually insert parentheses as it automatically wraps each sub-expression within parentheses. Hence, the user must only ensure the cursor is correctly positioned to determine where a new expression will be inserted. In contrast, VS Code requires the manual insertion of parentheses, which, for this task, involved 16 additional keystrokes.

As a point of comparison, the keystrokes required for linear program construction in VS Code are also recorded without the breakdown of sub-tasks. As shown in Table 5.1 ElmSr requires even fewer keystrokes than the linear construction of the code in VS Code for both tasks. Note that, dropdown options were utilized in linear construction when available.

Table 5.1: Total Keystrokes Count Comparison

Task No.	ElmSr	VS Code (Algebraic Thinking)	VS Code (linear)
Task 1	259	409	285
Task 2	75	-	83

To better understand the effort required for tasks as it progresses, we compared the number of keystrokes at each step and illustrated the time series in Figure 5.7

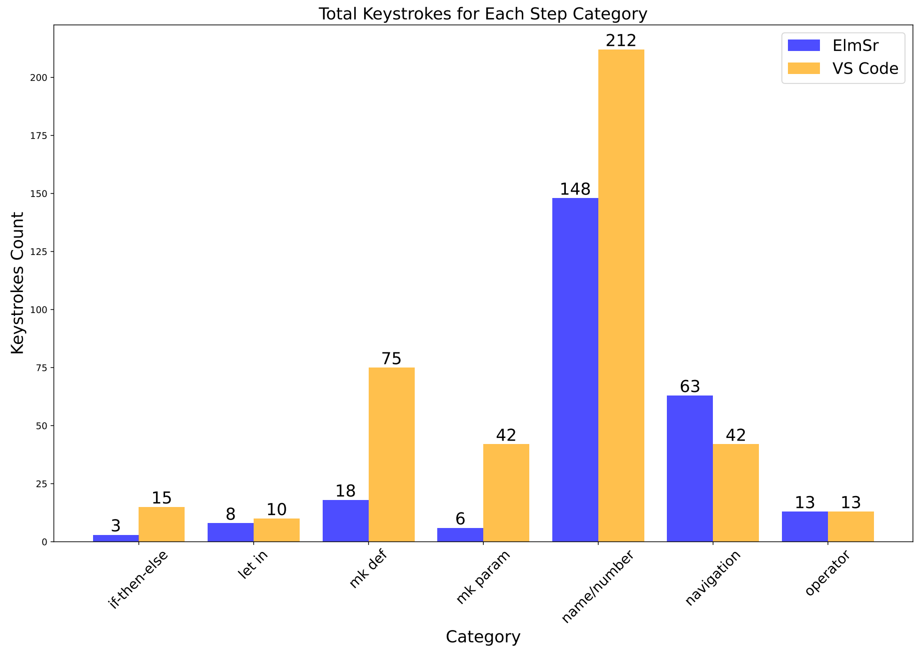


Figure 5.5: Total Keystrokes for Task 1

and Figure 5.8. The solid line represents the keystroke counts for ElmSr, while the dashed line represents those for VS Code. The colors of the markers indicate the categories. As evident, the effort in ElmSr consistently remains lower throughout task 1. It should be noted that when either tool does not require any operation for a corresponding task step in the other tool, we record this as a zero keystroke event, indicated by the `noop` category.

As illustrated in Figure 5.8, ElmSr demonstrates comparable effort to VS Code in entering lengthy expressions involving many infix operators and function applications. This indicates that despite its structured nature, ElmSr achieves similar efficiency in inputting complex expressions.

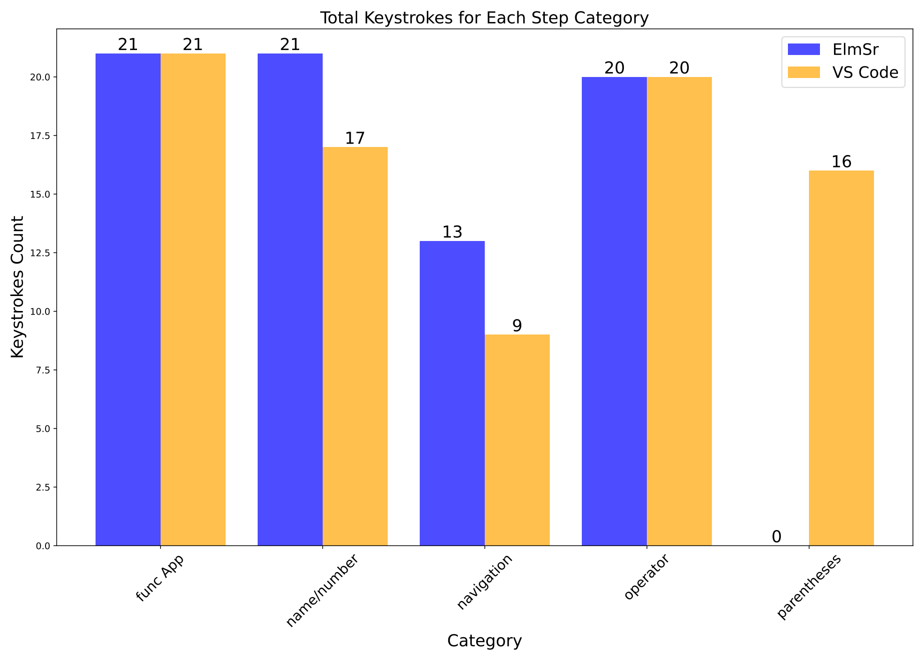


Figure 5.6: Total Keystrokes for Task 2

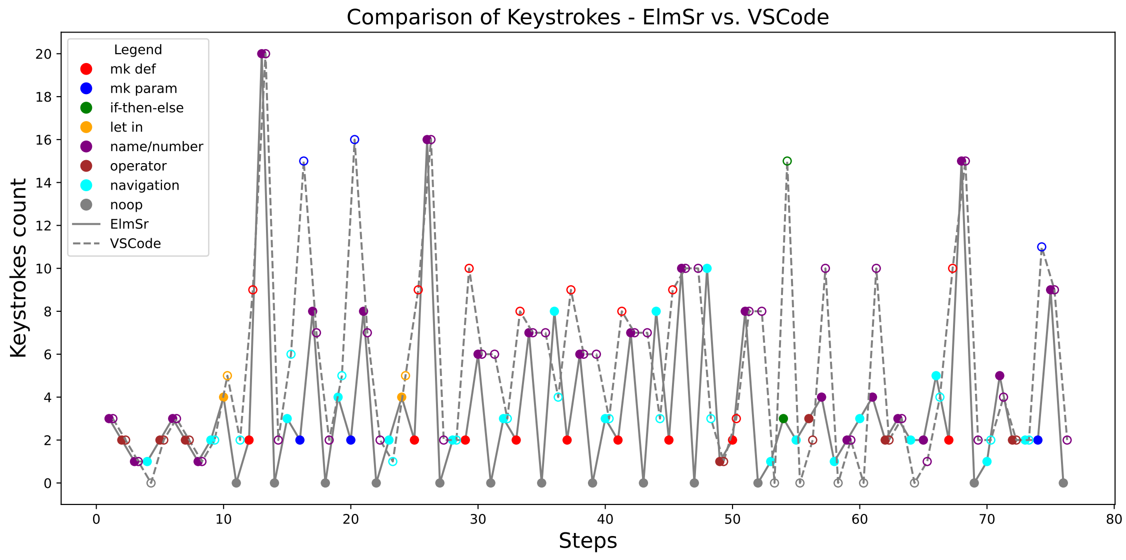


Figure 5.7: Comparison of keystrokes for Task 1. ElmSr keystroke counts are shown as solids on a solid line. VS Code counts are shown as outlined symbols on a dashed line. The lines are offset slightly to make it easier to identify overlap. The operations are identified by colour. In cases where the breakdown into steps for an operation differs, additional 0-counts are added so that subsequent operations align.

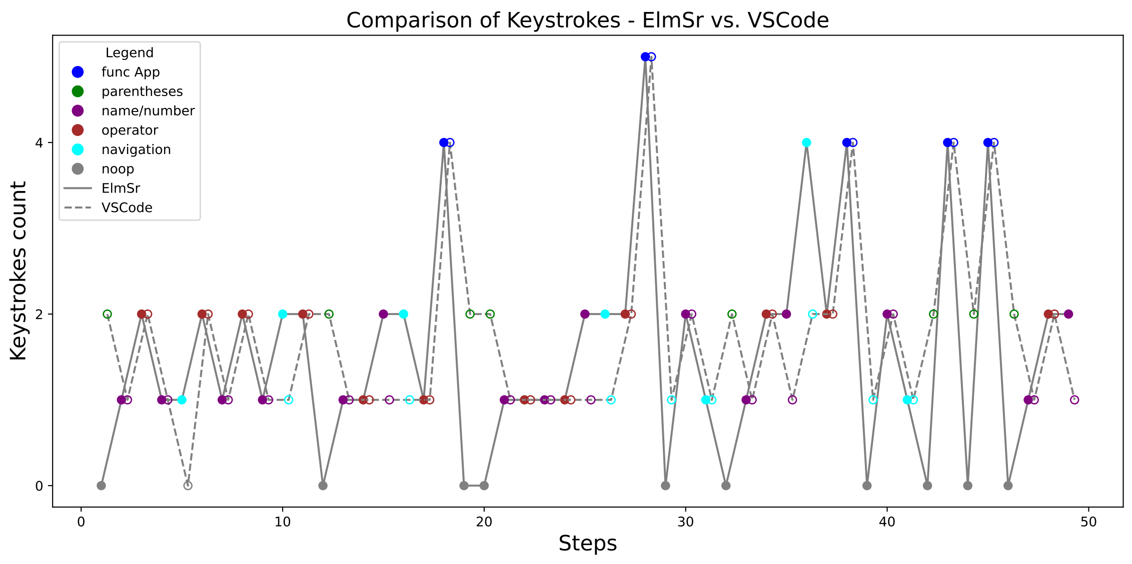


Figure 5.8: Comparison of keystrokes for Task 2. ElmSr keystroke counts are shown as solids on a solid line. VS Code counts are shown as outlined symbols on a dashed line. The lines are offset slightly to make it easier to identify overlap. The operations are identified by colour. In cases where the breakdown into steps for an operation differs, additional 0-counts are added so that subsequent operations align.

Chapter 6

Conclusion and Future Works

In this thesis, we set out to develop a structure editor for the functional programming language Elm and successfully achieved this goal. Unlike a simple text editor, where text entry, copy, paste, and delete are the primary functions, a structured editor must provide a comprehensive suite of actions to facilitate all legal program transformations, often in multiple steps. This requirement significantly increases the potential actions available in such an editor. Moreover, while actions like copy-and-paste are expected to be consistent across different programs for the sake of usability, structured editors face a lack of conventions or established expectations for many of the necessary program transformations.

To guide the design, we formulated two research questions and answered as follows:

RQ1. How can we optimize the usability of structure editors to provide a more intuitive and familiar experience, specifically when dealing with algebraic expressions utilizing infix operations?

We used the familiarity many users have with expression entry and autocompletion features in text editors to design editing actions that mimic a text-like expression entry and a well-implemented autocompletion system. This approach allowed us to enhance the intuitive feel of the editor and to make structured editing actions feel as natural and efficient as those in text editors.

RQ2. Is it possible for a structure editor to achieve similar or superior efficiency in program construction as compared to a traditional text editor, despite inherent structural differences, and could the unique functionalities of structure editors pave the way for more efficient program entry methods?

We developed two simple benchmarks to evaluate the efficiency of our structured editor compared to a traditional text editor. These benchmarks, which focused on the number of keystrokes required for various editing tasks, demonstrated that our structured editor is generally as efficient, if not more so, than conventional text editors. This efficiency was measured solely through keystroke counts, as mouse support has not yet been implemented in our editor.

Future developments for ElmSr are planned to extend its capabilities and address current limitations. Firstly, the existing implementation supports the `Number` type class in Elm, which is considerably simpler than Haskell's equivalent, as Elm only includes two number types: `Int` and `Float`. However, while there is some support for the `Comparable` type class to allow number comparisons and usage in conditional expressions, Elm's type system encompasses other types as well. Future work should either expand to include the full range of Elm's type classes or develop a novel approach to handling type classes. One potential direction could be the introduction of user-definable type classes or a different fixed set of type classes. This approach would shift significant design decisions from the editor to the library. It is important to note that Elm originally excluded user-defined type classes to simplify the compiler, so introducing them would definitely increase the complexity of the structured editor. Determining the best way to support this feature and implementing it will be crucial future tasks. Additionally, there is scope to support more complex constructs such as higher-order functions.

Regarding experimentation and evaluation, future work should focus on developing more comprehensive benchmarks and employing a wider range of metrics. Besides keystroke counts, other measures such as task completion times, or user interface-related measures, such as eye-tracking data, could provide deeper insights into the usability and efficiency of the editor.

Bibliography

[1] Blockly. `https://developers.google.com/blockly`, 2020. [Online; accessed 12-November-2023].

[2] GNU Emacs. `https://www.gnu.org/software/emacs/`, 2023. [Online; accessed 10-December-2023].

[3] ParEdit — parenthetical editing in Emacs. `https://paredit.org`, 2023. [Online; accessed 10-December-2023].

[4] Jetbrains MPS. `https://www.jetbrains.com/mps/`, 2023. [Online; accessed 10-December-2023].

[5] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., USA, 1986. ISBN 0201100886.

[6] D. Bau. Droplet, a blocks-based editor for text code. *Journal of Computing Sciences in Colleges*, 30(6):138–144, 2015.

[7] T. Beckmann, P. Rein, T. Mattis, and R. Hirschfeld. Partial parsing for structured editors. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*, pages 110–120, 2022.

[8] N. C. Brown, A. Altadmri, and M. Kölling. Frame-based editing: Combining the best of blocks and text programming. In *2016 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)*, pages 47–53. IEEE, 2016.

[9] M. Brunsfeld. Tree-sitter. `https://tree-sitter.github.io/tree-sitter/`, 2020. [Online; accessed 5-December-2023].

[10] M. J. Conway and R. Pausch. Alice: easy to learn interactive 3D graphics. *ACM SIGGRAPH Computer Graphics*, 31(3):58–59, 1997.

[11] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for GUIs. *ACM SIGPLAN Notices*, 48(6):411–422, 2013.

[12] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured programming*. Academic Press Ltd., GBR, 1972. ISBN 0122005503.

[13] C. d'Alves, T. Bouman, C. Schankula, J. Hogg, L. Noronha, E. Horsman, R. Siddiqui, and C. K. Anand. Using Elm to introduce algebraic thinking to k-8 students. *arXiv preprint arXiv:1805.05125*, 2018.

[14] V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, and J. Levy. *A structure-oriented program editor: a first step towards computer assisted programming*. IRIA. Laboratoire de Recherche en Informatique et Automatique, 1975.

[15] V. Donzeau-Gouge, G. Huet, B. Lang, and G. Kahn. *Programming environments based on structured editors: The MENTOR experience*. PhD thesis, Inria, 1980.

[16] V. Donzeau-Gouge, B. Lang, and B. Me'le'se. A tool for Ada program manipulations: Mentor-Ada. In *Proceedings of the 1985 annual ACM SIGAda international conference on Ada*, pages 297–308, 1985.

[17] D. B. Garlan and P. L. Miller. Gnome: An introductory programming environment based on a family of structure editors. *ACM Sigplan Notices*, 19(5):65–72, 1984.

[18] K. Gopinathan. Gopcaml: A structural editor for OCaml. *arXiv preprint arXiv:2207.07423*, 2022.

[19] A. N. Habermann and D. Notkin. Gandalf: Software development environments. *IEEE transactions on software engineering*, (12):1117–1127, 1986.

[20] W. J. Hansen. User engineering principles for interactive systems. In *Proceedings of the November 16-18, 1971, fall joint computer conference*, pages 523–532, 1972.

[21] B. Harvey and J. Mönig. Bringing "no ceiling" to scratch: Can one language serve kids and computer scientists. *Proc. Constructionism*, pages 1–10, 2010.

[22] B. Hempel, J. Lubin, G. Lu, and R. Chugh. Deuce: a lightweight user interface for structured editing. In *Proceedings of the 40th International Conference on Software Engineering*, pages 654–664, 2018.

[23] B. Hempel, J. Lubin, and R. Chugh. Sketch-n-sketch: Output-directed programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, pages 281–292, 2019.

[24] G. Huet. The zipper. *Journal of functional programming*, 7(5):549–554, 1997.

[25] J. Hughes. Why functional programming matters. *The computer journal*, 32(2): 98–107, 1989.

[26] A. J. Ko and B. A. Myers. Citrus: a language and toolkit for simplifying the creation of structured editors for code and data. In *Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 3–12, 2005.

[27] A. J. Ko and B. A. Myers. Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 387–396, 2006.

[28] A. J. Ko, H. H. Aung, and B. A. Myers. Design requirements for more flexible structured editors from a study of programmers' text editing. In *CHI'05 extended abstracts on human factors in computing systems*, pages 1557–1560, 2005.

[29] J. Lubin and S. E. Chasins. How statically-typed functional programmers write code. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, 2021.

[30] R. Medina-Mora and P. H. Feiler. An incremental programming environment. *IEEE Transactions on Software Engineering*, (5):472–482, 1981.

[31] D. Moon, A. Blinn, and C. Omar. tylr: a tiny tile-based structure editor. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Type-Driven Development*, pages 28–37, 2022.

[32] D. Notkin. The GANDALF project. *Journal of Systems and Software*, 5(2): 91–105, 1985.

[33] C. Omar, I. Voysey, M. Hilton, J. Aldrich, and M. A. Hammer. Hazelnut: a bidirectionally typed structure editor calculus. *ACM SIGPLAN Notices*, 52(1): 86–99, 2017.

[34] C. Omar, I. Voysey, R. Chugh, and M. A. Hammer. Live functional programming with typed holes. *Proceedings of the ACM on Programming Languages*, 3(POPL): 1–32, 2019.

[35] C. Omar, D. Moon, A. Blinn, I. Voysey, N. Collins, and R. Chugh. Filling typed holes with live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 511–525, 2021.

[36] J. Pierce, T. Cobb, and R. Pausch. Alice. In *ACM SIGGRAPH 98 Conference abstracts and applications*, page 140, 1998.

[37] T. W. Price, N. C. Brown, D. Lipovac, T. Barnes, and M. Kölling. Evaluation of a frame-based programming editor. In *Proceedings of the 2016 ACM Conference on International computing education research*, pages 33–42, 2016.

[38] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[39] A. L. Santos. Javardise: a structured code editor for programming pedagogy in Java. In *Companion Proceedings of the 4th International Conference on Art, Science, and Engineering of Programming*, pages 120–125, 2020.

[40] M. Sipser. Introduction to the theory of computation. *ACM Sigact News*, 27(1): 27–29, 1996.

[41] T. Teitelbaum and T. Reps. The Cornell program synthesizer: a syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, 1981.

[42] M. Verano Merino and T. Van Der Storm. Block-based syntax from context-free grammars. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, pages 283–295, 2020.

[43] M. Verano Merino and K. Van Wijk. Workbench for creating block-based environments. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*, pages 61–73, 2022.

[44] M. Verano Merino, T. Beckmann, T. Van Der Storm, R. Hirschfeld, and J. J. Vinju. Getting grammars into shape for block-based editors. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering*, pages 83–98, 2021.

[45] M. Voelter, J. Siegmund, T. Berger, and B. Kolb. Towards user-friendly projectional editors. In *International Conference on Software Language Engineering*, pages 41–61. Springer, 2014.

[46] M. Voelter, T. Szabó, S. Lisson, B. Kolb, S. Erdweg, and T. Berger. Efficient development of consistent projectional editors using grammar cells. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 28–40, 2016.

[47] P. Voinov, M. Rigger, and Z. Su. Forest: Structural code editing with multiple cursors. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 137–152, 2022.