Preliminary Report on the Kale Visual Programming Environment

Maciej Goszczycki

16316981

Final Year Project - 2019/2020 - 15 credits

B.Sc. Computer Science and Software Engineering

Department of Computer Science

Maynooth University

Maynooth, Co. Kildare

Ireland

A thesis submitted in partial fulfilment of the requirements for the B.Sc. Computer Science and Software Engineering.

Supervisor: Dr. Barak A. Pearlmutter.

The latest version of this report can be found at `https://kale.maciej.ie/report.pdf`

Contents

Declaration

I hereby certify that this material, which I now submit for assessment on the program of study as part of B.Sc Computer Science and Software Engineering qualification, is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

I hereby acknowledge and accept that this thesis may be distributed to future final year students, as an example of the standard expected of final year projects.

Signed: **Date:** Monday 13th April 2020

Abstract

Contemporary text-based programming languages require the programmer to be constantly vigilant against syntax errors. This affects both novice and experienced programmers. One obvious way of eliminating syntax errors is to get rid of text. While there exist many visual programming environments, they generally eschew powerful editing capabilities in order to be friendly to beginners. This project explores how a Lisp-based visual programming environment might be used to mitigate syntax issues while remaining attractive to both novices and professional programmers.

Introduction

1.1 Motivation

Syntax errors are a fact of life in the programming industry. Programmers, from novice [1] to professional, spend significant time fighting or avoiding syntax errors. It is clear a visual programming environment such as Scratch could eliminate syntax errors. There exists a large body of existing visual programming environments [2] but most focus exclusively on making programming accessible to children and young adults. Most professional tools focus instead of error-detection, ignoring visual means of editing programs as slow and cumbersome. The Kale system is an attempt to show that it is possible to create a visual programming environment that is friendly to novices but powerful enough to be taken seriously by professionals.

Kale consists of a web based visual programming environment, designed from the ground up to fit many skill levels. It demonstrates that a drag-and-drop/blocks style interface can coexist with a keyboard-driven professional-focused editing experience. It generates usable code, runnable from within the interface.

1.2 Approach

The project consists of three major phases.

1.2.1 Layout engine development

In this phase, the goal is to implement a functional layout engine capable of displaying static expressions and handling sufficiently complex code.

1.2.2 Establishing a workflow

The next phase consists of adding editing capabilities to the project. These will be very keyboard-driven, to easily test out new ways of editing code and find any major flaws.

1.2.3 Simplifying the workflow

Once completed the final goal will be to simplify the editing experience by presenting the different editing operations in easy-to-use and accessible ways, such as mouse or touch driven editing.

1.3 Project

- The Kale visual programming environment can edit arbitrary expressions, using both mouse-driven and keyboard-driven workflows.
- A simple interpreted programming language enables testing the editing experience on functioning examples.
- As part of the development process I contributed patches to two different open-source projects, including adding `#rrggbbaa` notation support to the Popmotion library [3] and updating "styled-components" TypeScript typings to the a new major 5.0 version [4].

Technical Background

2.1 Topic material

2.1.1 Block based languages

The MIT Media lab's Scratch [5] is probably the most broadly successful block-based programming languages.

2.1.2 Frame based editing

Another attempt at mitigating syntax errors has been made in the form of "frame based editing". Kölling et al. [6] created a new editing environment based around the concept of frames, it serves a middle-ground between Kale and text-based editing.

2.2 Technical Material

2.2.1 Command Composition

Composing user commands is an important concern for any code editor. Chodarev [7] discussed this at length, pointing out that casual text-based editing environments rely on simple commands that operate only on elementary objects, manipulating programs a single character at a time. However this approach is quite inefficient, leading applications to adopt a secondary set of ad-hoc commands for performing specific combinations of operations and high-level objects (like deleting a word). One proposed alternative is Vim-like shortcuts, where the user combines a smaller set of motion and action commands to work on high-level objects.

2.2.2 Proportional fonts

Studies have shown that proportional (variable-pitch) fonts are faster to read than their `monospaced` counterparts [8, 2]. One notable example of a proportional font being used typeset code is the C++ Programming Language book by Bjarne Stroustrup.

2.2.3 TypeScript

TypeScript is a structurally and gradually typed superset of JavaScript being developed at Microsoft [9]. It provides stronger type-safety guarantees than JavaScript and allows programs to use modern and experimental JavaScript features such as class properties or optional chaining.

2.2.4 React and related libraries

To ease the burden of manually manipulating the Document Object Model (DOM), Kale uses the React JavaScript library [10]. React is a self-proclaimed "JavaScript library for making user interfaces" being developed by Facebook. It allows the program to write rendering code in a functional fashion, relying a "Virtual DOM to make DOM updates efficient". Together with JavaScript XML (JSX), a JavaScript extension supported by TypeScript, React makes it easy to write complex user interfaces (UIs) for the browser.

2.2.5 Scalable Vector Graphics (SVG)

SVG is a web technology which can be used to display vector images in the browser. It is scriptable by JavaScript and styleable with CSS, making it the perfect fit for dynamically rendering any highly custom UIs. It is used by Scratch for rendering their blocks interface.

2.2.6 Lisp

Lisp is a family of programming languages, known for its regular syntax, homoiconicity, and dynamic features.

The Problem

The programming language which Kale users will be writing should map well to the UI but be powerful enough to be able to express complex programs. It need not necessarily be an existing language. In fact, it might not be desirable to make it one.

A proficient user should be able to write programs and navigate Kale without having to reach for the mouse. This might involve mapping Kale concepts to standard shortcut keys, as well as developing new ones. It should be possible for users of existing programming environments to adjust to Kale with relative ease. Programs written in Kale should be readable, balancing accessibility and information density.

Novice users should be able to construct Kale programs in a simple and intuitive manner on every device form-factor whether touch-screen or mouse-driven. It should be simple for

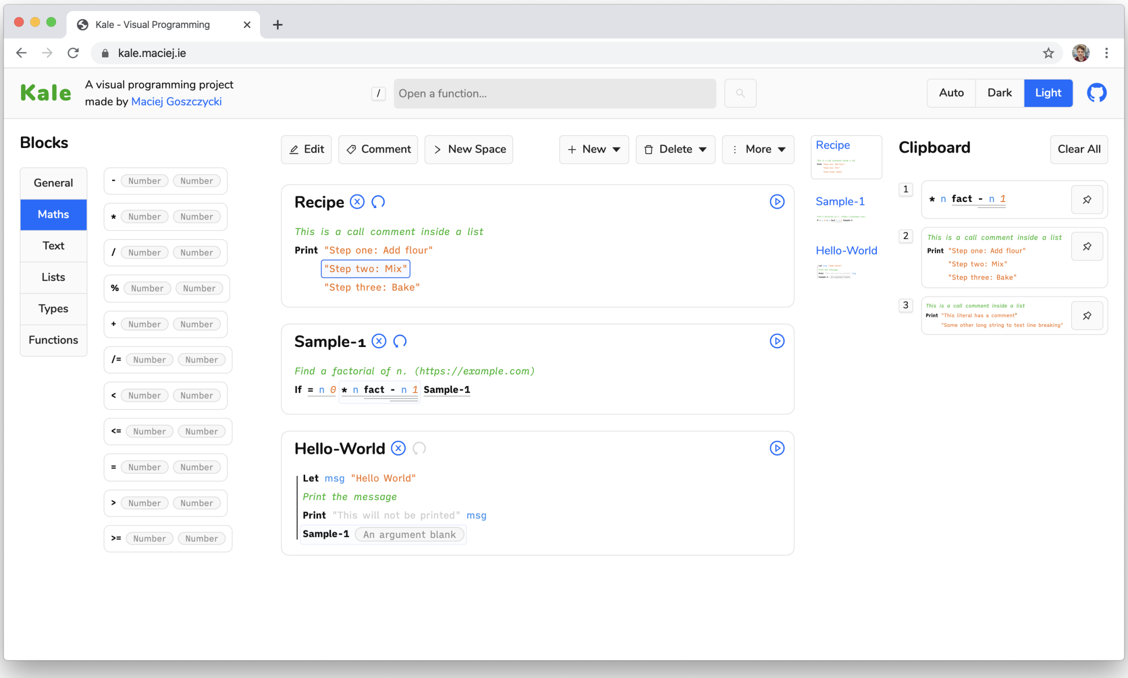

Figure 4.1: The Kale user interface

them to discover new functionality and edit existing programs without knowledge of higher-level operations.

Effectively manipulating Lisp programs often requires commands for high-level hierarchical selections and actions, which can be difficult for users to remember. Editors like Vim use the command composition pattern to enable these actions but this comes at a great cost: learning difficulty [7].

The Solution

4.1 Expression structure

Adapting programming language syntax, to be elegant to display and easy to consistently manipulate, requires radically simplifying the structure that is common in contemporary languages. Lisp [11] is a family of languages known for their regular syntax and homoiconicity, making it a perfect fit for Kale.[1] Each expression within Kale belongs to one of five categories.

4.1.1 Literals

"2 + foo is"

Kale relies on a very generic notion of literals. A literal is a string tagged with its type, which is an arbitrary string. In practice, Kale currently makes use of

[1] In fact Kale originally stood for Kick Ass Lisp Environment

only two types of literals: "text" and "number", but the system is designed to handle any literal type that can be serialised into a string.

4.1.2 Variables

`foo`

Variables are simplest of Kale's expression types. Each variable is a string, encoding the variable name it represents.

4.1.3 Function calls

`Print "2 + foo is" + 2 foo`

Are analogous to Lisp's S-expressions but with a constraint that the first value must be a function identifier. While this makes higher-level functions somewhat harder to write (similarly to Common Lisp), it allows for a more consistent navigation system by removing the distinction between a function call expression being selected or just the function itself.

4.1.4 Spaces

`?`

Because the fundamental way of moving around code in Kale is the currently selected expression, instead of a cursor, another way of inserting new expressions must be provided. One way of achieving this might be different commands, like "Paste after this expression", but this would significantly complicate Kale's command structure. Kale provides insertion points using spaces (or "blanks"). Spaces are similar to Snippet Placeholders used in modern editors, but are actually part of the expression tree (instead of being simple ephemeral placeholders used only during code completion). Spaces make it simple to create new Kale expression by simply "filling in the blanks" and simplify the command structure.

4.1.5 Expression lists

```
Print "2 + foo is" + 2 foo
?
```

In Lisp, new scope is introduced by the `let` macro, while elegant, the complex nesting level syntax is difficult to express well visually. Kale attempts to make scoping more visible while also providing a visually clean way of representing the `progn` (also called `do`) macro, through the expression list structure. Each list contains at least two expression and introduces a new lexical scope. This makes it easy to explain Kale's scoping rules visually. More precisely, each expression list adheres to two invariants:

- Every list must have at least two expressions. If there are one or fewer expressions, the list is automatically removed, leaving its content in its place. In Lisp this might be expressed as `(progn a)` ≡ `a`.
- A list's direct children cannot be lists themselves. Nested list's contents are merged into their parent list. This is because `(progn (progn a b) c)` ≡ `(progn a b c)`. Nested lists are almost never intentional in Kale and enforcing this invariant simplifies many commands.

4.2 Drag and drop

One of the main changes Kale introduces, compared to a normal Lisp editor, is drag and drop. It lets novice users effectively manipulate Kale programs without prior knowledge of commands like copy and paste. Unlike the majority of other visual programming environments, Kale distinguishes between two types of drag and drop actions: Replacement and Insertion. These can can be seen in Figure 4.3 and Figure 4.2 respectively.

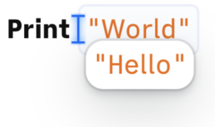

Figure 4.2: Inserting an expression

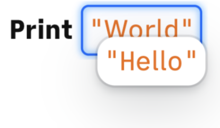

Figure 4.3: Replacing an expression

4.3 Structural underlines

Lisp (on which Kale's expression structured is heavily based on), is known for its high number of parentheses. To solve this, Kale expressions rely on on the concept of **Structural underlines**. Any function-call arguments to a single line function call expression are underlined. Any child underlines are placed *under* their parent underlines. This approach sacrifices some vertical space for a clear visual representation of an expression's nesting structure. Using structural underlines, the only characters on the screen are those that can be directly typed by the user. Operations like "Move Up" ⇧+P, are also cleanly represented by the underlines. Design exploration for this feature can be found in figures B.1 and B.2.

```
* + 2 * 4 6 8 + 3 5 7
```

Figure 4.4: Expression using structural underlines

```
*(+(2 *(4 6) 8) +(3 5 7))
```

Figure 4.5: Equivalent parenthesised expression

4.4 Command structure

Because Kale does not operate on elementary text elements, even the most basic commands can be much higher-level than ordinary editors since instead of a current cursor position, Kale operates on the selected expression. Even so, figuring out the correct set of commands proved to be a challenge.

A good example of the difficulties that arise in common text editors are the arrow key commands ↑ ↓ ← →. Initially, these were implemented as fundamental tree operations: "Select parent", "Select first child", "Select left sibling", and "Select right sibling" respectively. While logical, these operations were unintuitive to every user Kale was shown to, including the author. In the end **Smart Selection** was implemented, where ← and → use the pre-order[2] traversal, while ↑ and ↓ pre-order traverse only non-inline non-list expressions, mirroring the visual line motion a normal cursor might make.

[2]The pre-order traversal algorithm traverses the parent node first, then traverses the left and right tree by calling itself on each.

4.5 Discoverability

Most commands can be accessed in at least three ways: through their dedicated keyboard shortcut, the context menu, and the top-level editor menu. In case the user is keen on using the keyboard shortcuts but forgot a specific command, each editor menu-item shows the corresponding keyboard shortcut, and each shortcut can be triggered whilst the context menu is open. These keyboard shortcut indicators are also placed throughout Kale to help with discovering shortcuts for the Clipboard List or the Function Search menu.

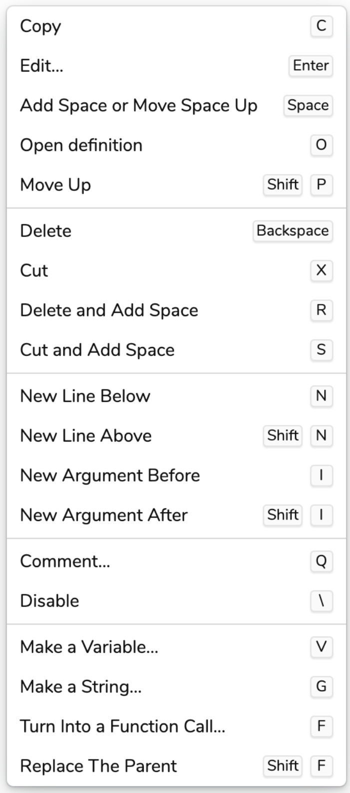

Figure 4.6: Kale's Context Menu

4.6 High-level manipulation

4.6.1 Commenting

Comments in text-based programming are normally quite ad-hoc. It is up to the reader to establish which comment applies to which expression. This problem is exacerbated by using comments to selectively disable pieces of code. Kale addresses both of these problems elegantly.

Similar to to other visual programming environments, Kale allows assigning comments directly to specific expressions, making them completely unambiguous and leaving it up to the layout engine to figure out where each comment should appear. However, in the future, rich or formatted text might be used also. Presently, every expression can be commented on. However, comments on spaces are displayed inside the space's "bubble", while comments on literals are not directly displayed, instead being accessible thought a special comment tooltip.

Disabling pieces of code is another common operation. Most modern programming editors provide a single command to format a line or selection like Ctrl + /. Kale's semantics driven selection mechanism neatly fits with the need to disable specific expressions. Using the "Disable" \ command, expressions can be easily disabled, marking them to be ignored by the interpreter. This effectively eliminates any ambiguity that exists in text-only comments.

4.6.2 Clipboard stack

Relinquishing control over the elementary elements of a program might potentially make more high-level manipulation problematic. To remedy this, Kale needed to provide a better way of transforming expressions. Andrew Blinn's Fructure [12] environment tackles this by letting user colour expressions, then using a special transformation mode where the user can enter new expression or use one of the previously coloured ones. While imbued with a certain sense of mathematical purity, this approach deals poorly with more complex refactorings and requires colour vision.

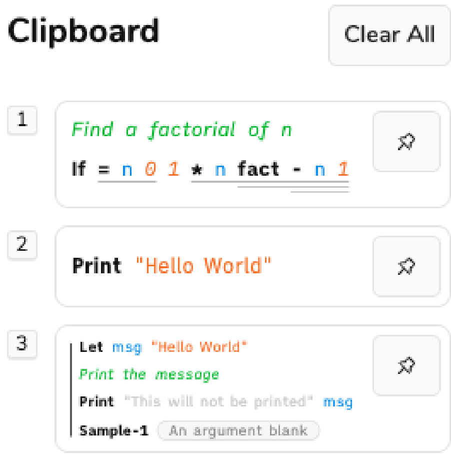

Figure 4.7: The Clipboard Stack

Kale's solution to this problem comes in the form of the **Clipboard Stack**. The clipboard stack is a stack of expressions shown on the right-hand side of the screen. Kale provides a "Copy" C command which copies the currently selected expression to the top of the clipboard stack. To facilitate more destructive refactoring, Kale provides a palette of deletion commands:

- "Delete" ⟵
- "Cut" X
- "Delete and Add Space" R
- "Cut and Add Space" S

Expressions on the clipboard stack can replace the currently selected expression by using one of nine "Paste" 0 – 9 commands, with each command's shortcut prominently displayed to the right of each expression it would paste.

Sometimes it is beneficial to be able to paste the same expression multiple times. Expression on the clipboard stack can be pinned using the Pin-shaped button. Pinned expressions are not removed from the stack when their corresponding paste command is invoked. They are also not cleared by the "Clear All" button.

4.6.3 Smart space

In text-based programming languages various punctuation marks are used to indicate new statements and function arguments. This presents a challenge for an editor like Kale; How to let the user indicate that a new expression should be created. At first this was implemented as a set of expression-kind specific operations: "Create new child", "Create new sibling", and "Create new line". However this proved unintuitive as unlike normal punctuation, the shortcut keys for these operations did not correspond to any character produced on the screen, making them difficult to memorize. Additionally, while these actions were more powerful than their text counterparts, it was unintuitive how the command needed to create a new argument changed depending on whether a function call or one of its arguments was selected.

Kale's current solution to this problem is the "Smart Space" command. Smart space is a high-level operation that attempts to perform a reasonable action no matter the selection:

- If a function is selected create a new child space in the first argument.
- If a space is selected use the "Move Up" ⇧+P command, making the currently selected expression the last sibling of its current parent.
- Otherwise, create a new sibling space to the right of the selection, for example creating a new argument.

Note that this does not cover the "Create new line" operation, so this option still exists in the form of "New Line Below" N / "New Line Below" ⇧+N commands.

Implementation

5.1 Layout engine

A key component of the layout engine is the notion of "inline" expressions. An inline expression is one which can be easily displayed inside another inline expression. In Figure 5.1 inline expressions are those with a blue outline while those with a red outline are non-inline.

Literals, blanks and variables are always inline, while lists are always non-inline. For function calls a heuristic-driven algorithm is used to determine the inline status. The `isCallInline` algorithm determines a function call to be inline if:

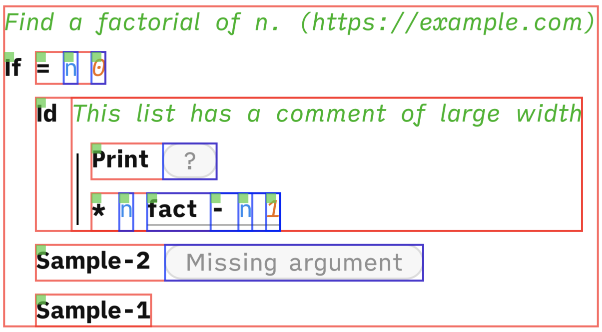

Figure 5.1: Kale's internal layout information[1]

1. No comment exists on the function call.
2. Every argument is also inline.
3. The sum total length of the argument widths is below 300 pixels.
4. The hight of the expression tree of every argument is below four.

Non-inline function calls are broken up, and their underline stack (explained below) is rendered onto the screen, while inline calls continue to build up the underline stack.

[1]The layout debug overlay is always available within Kale. To access it, right click any expression, then hold Ctrl (or Alt on macOS) to reveal a new hidden "Toggle the Debug Overlay" option.

5.1.1 Data structure

Keeping track of all the data required by the layout algorithm including enabling in-line editing and supporting drag and drop requires a complex layout data structure. Because parent expressions like function calls need access to their children's layout information to make layout decisions, it is not sufficient to simply treat Kale expressions as React components to be rendered. Instead, the `Layout` class is responsible for keeping track of state required of each expression and SVG elements by their parents.

```
class Layout {
    size: Size;
    nodes: ReactNode[] = [];
    underlines: Underline[] = [];
    areas: Area[] = [];
    inline = false;
    isUnderlined = false;
    expr: Expr | null = null;
    partOfExpr: Expr | null = null;
    text: Optional<TextProperties>;
}
```

`size` A key component of effectively laying out expressions is keeping track of their size. Note that since SVG elements have no inherent size, this is simply a suggestion of what the predicated size of a layout element will be.

`nodes` The nodes array is responsible for aggregating all the SVG nodes rendered to up to this point.

`underlines` Kale's inverted stack of underlines means individual expressions cannot know at what level and depth their underline will be rendered. Instead the underlines to be drawn at the first non-inline function call parent are lazily kept track of during the layout process. With each parent expression laid out, the level and the offset relative to the parent expression is updated. Once underlines are finally drawn, the stack is cleared.

```
interface Underline {
    level: number;
    offset: number;
    length: number;
}
```

Figure 5.2: The underline interface

`areas` Expression areas are the aggregation of all the layout element data collected up to a certain point in the rendering process. It has a variety of uses throughout Kale including, drag-and-drop hit-testing and inline data used for the ↑ and ↓ traversal. Kale supports two kinds of areas: expression areas and "gap" areas used for implementing drag and drop.

`inline` Stores the inline status of a layout element as explained above.

isUnderlined Not every inline layout element needs to create a new underline. Kale chooses to not underline atomic expressions like spaces, literals, and variable names.

expr Generating the areas field requires keeping track of the expression each layout element "belongs to". Setting this field indicates to the layout system that this layout element should be assigned an area and optionally an underline.

partOfExpr Drag-and-drop gap areas for most expressions allow inserting a new dropped expression between sibling expressions. However, when using drag and drop over function call expressions, it is desirable to be able to create new *child* expressions. This field complements the expr field and allows the function calls's text to have a drop areas where one normally would not exist.

text Creating the field editor requires re-creating the same text style as used during layout. This field is set by the text layout methods to keep track of the exact text-styling used.

5.1.2 Bounding box refinement

When laying out inline expressions the layout engine does not know how tall the underlines under under each expression will be until its first non-inline parent is rendered. Once this happens, bounding box refinement is performed: all inline children of the rendered non-inline expression have their heights adjusted to match their non-inline ancestor.

Figure 5.3 shows an example expression. Without the refinement process the bounding boxes of a's children are too short, as seen in Figure 5.4. This would lead to expression highlights potentially overlapping with the underlines. Figure 5.5 shows the bounding boxes after the refinement has been performed.

Figure 5.3: Expression to be refined

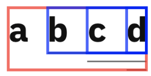

Figure 5.4: Before refinement

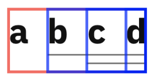

Figure 5.5: After refinement

5.1.3 Text metrics

Creating the Layout data structure for text elements requires information on the metrics of piece of rendered text. Unfortunately, the current state of web text-metrics APIs leaves a lot to be desired. While the <canvas> element provides a seemingly comprehensive TextMetrics[2] API, the reality is the vast majority of the metrics systems like Kale might be interested in consuming are currently only available in the latest browsers, behind experimental flags.

Instead Kale takes the same approach as the Scratch Blocks [13] library, using SVG's getComputedTextLength API and an invisible <svg> element onto which new pieces of text are rendered.

[2] https://developer.mozilla.org/en-US/docs/Web/API/TextMetrics

5.2 Drag and drop

The area system is used to create drag-and-drop areas between expressions. Each area keeps track of its related expression, and whether dropping an expression on it should cause the dropped expression to be added as a sibling or a child of that expression.

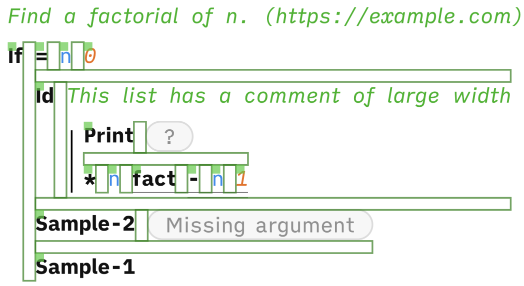

Figure 5.6: The expression from Figure 5.1 showing drag-and-drop insertion slots.

5.3 Expression IDs

Originally reference identity was used for handling the selection and edition functionality. However, this presented a challenge when more than one editor was opened. Editing an expression would change its reference identity, invalidating the current selection in other editors displaying the same function. To remedy this, each expression is allocated a unique ID number. Simply editing an expression, such as changing its children or editing its text leaves the ID intact, but creating a new expression, or copying one from the clipboard list creates a new ID.

5.4 Web deployment

Any code committed to Kale's Github repository[3] is automatically built by the Travis Continuous Integration (CI) service, which runs tests, bundles and optimises code using webpack, as well as deploys the resulting site to Github Pages.

Evaluation

With the currently implemented features, Kale forms a completely usable visual programming environment. Users are able to create arbitrary expressions, and manipulate them with relative ease without ever encountering a single syntax error. Kale is stable and the layout algorithm is quite efficient.

6.1 Editing

Many features work in tandem to make the editing experience as complete as it is now. Drag and drop is simple to use, supports touch-screen devices, and common idioms like pressing Ctrl to copy instead of move an expression. Most features can be accessed in variety of ways, catering to an extensive set of skill levels.

[3] https://github.com/mgoszcz2/kale

6.2 Editor features

Kale provides a substantial set of features one would expect from a modern programming editor. Auto-complete suggestions are provided when typing function expressions. Functions can be quickly opened or created by using an easily accessible fuzzy matching search box. The editor is available in both Light and Dark themes, to match the taste of the user; and provides a reasonably polished UI.

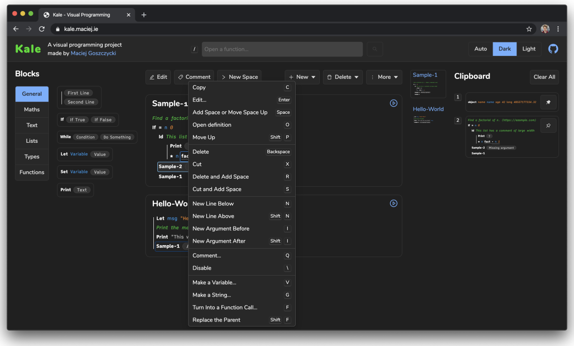

Figure 6.1: Kale Dark theme

6.3 User interface

On touch-screen devices, buttons and menus follow Apple Human Interface Guidelines[1] 44pt x 44pt tappable area recommendation. By using the Hypertext Markup Language (HTML) Pointer Events[2] API, all reasonable input devices are supported for drag and drop, including dragging multiple expression simultaneously, should the user so desire.

6.4 Workflow

Modern editors often provide a tabbing system to allow users to work on several files at once. While Kale's editing experience is completely function focused, a single function stack makes it hard to develop larger projects. A tabbing system, switching between several different function stacks, would greatly help scaling Kale to work on bigger, more challenging projects.

6.5 Kale language

The Kale language right now really serves as a placeholder for a more complete programming language Kale might operate on. It is sufficient for demonstrating the broader editing experience, but is somewhat hindered by the fact that Kale does expose an interface to set function parameters on user-defined functions. Developing the language more might further help to demonstrate the unique features Kale can provide. Even better, the Kale language could instead consist of a simple translation layer over an existing, established language, helping to broaden the appeal of the new editing paradigm.

6.6 Implementation

React proved to be excellent choice as the UI library. While there exists a certain impedance mismatch between the Kale layout algorithm and React's programming style, in practice this was not much of an issue, given how custom Kale's layout algorithm is. Typescript and React

[1] https://developer.apple.com/design/human-interface-guidelines/ios/visual-design/adaptivity-and-layout/

[2] https://www.w3.org/TR/pointerevents/

are mature technologies that made much of the rapid iteration, essential to developing Kale into the editor it is today, possible.

An earlier version of Kale was written in the Rust programming language. Rust is a modern, powerful and safe programming language, which recently gained a WebAssembly compilation target. Unfortunately Rust's immature web-library ecosystem, combined with its strict memory model, made it hard to implement a complete editor in.

Conclusion

In conclusion the project shows that a visual programming environment can be approachable by both novices and professionals and eliminate a class of programming errors, without limiting program creation in a substantial way.

7.1 Future Work

7.1.1 User studies

The main concern of this project was creating a property of the Kale environment, in the future, it would be beneficial to perform user studies on the Kale and the various design choices made along the way. Some things that would be investigated might be:

- The value of the structured underlines.
- The effectiveness of Kale in transitioning children away from Scratch.
- User experience of professional programmers.

7.1.2 Drag and drop improvements

Currently due to the limitations of drag and drop implementation, it is impossible to insert new expression as the last sibling of expression lists and function calls. This can be viewed as a limitation of the structural-underlines system as, unlike with parentheses, there is no clear point at which each sub-expression ends. In the future this might be solved by making the dropped expression the last child of the inner-most function call or providing a disambiguating UI while a drag is in progress.

7.1.3 Integrated debugging

Figure 7.1: A prototype of expression breakpoints

Right now the runtime capabilities provided by Kale are quite basic. While effort was put into making the error messages somewhat descriptive, there are no advanced debugging capabilities available. This is unfortunate as expression-oriented selection lends itself nicely to visualising the currently executing expression and setting new breakpoints. Modern text-oriented debuggers use various inline markers "inline breakpoints"[1]. to try to indicate what expression the execution is halted on. In Kale, setting a new breakpoint would follow from the selection mechanism and use the same highlighting machinery.

7.1.4 Fluid field editor

"Fluid entry" is a concept for simplifying entering new expressions. Right now to turn a space into another expression the user needs to either enter the "space popover" or memorise a set of shortcuts like "Make a Variable..." V or "Turn into a Function Call..." F. These shortcuts are currently essential to efficiently creating new Kale expressions, but for many expressions they feel like an extra step that would not be necessary under a normal text editor.

The idea behind the "fluid field editor" would be to eliminate these shortcuts. Instead, delaying the decision on what type of expression should be created, until the user has typed at least one character.

Typed character	Resulting expression
" or '	Text literal
0 – 9	Number literal
⇧+A – ⇧+Z	Function call
A – Z	Variable expression

7.1.5 Removing modality

Removing the field editing feature might be worth investigation. This could be achieved by implicitly entering the field editing mode when changing the selection. ← and → could then always move the cursor inside the current field, similarly to standard text editors, once. If the cursor reaches the end of a field, the next expression would be selected, following the current default behaviour.

[1] https://developers.google.com/web/updates/2019/04/devtools

Bibliography

[1] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. Understanding the syntax barrier for novices. *ITiCSE'11 - Proceedings of the 16th Annual Conference on Innovation and Technology in Computer Science*, pages 208–212, 2011. doi: 10.1145/1999747.1999807.

[2] Ion P. Beldie, Siegmund Pastoor, and Elmar Schwarz. Fixed versus Variable Letter Width for Televised Text. *Human Factors*, 1983. ISSN 00187208. doi: 10.1177/001872088302500303.

[3] Add support for the rrggbbaa hex color notation by mgoszcz2 · Pull Request #868 · Popmotion/popmotion, . URL `https://github.com/Popmotion/popmotion/pull/868`.

[4] Update styled-components to 5.0 by mgoszcz2 · Pull Request #42415 · DefinitelyTyped/DefinitelyTyped, . URL `https://github.com/DefinitelyTyped/DefinitelyTyped/pull/42415`.

[5] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Trans. Comput. Educ*, 10, 2010. doi: 10.1145/1868358.1868363. URL `http://doi.acm.org/10.1145/1868358.1868363.`

[6] M Kölling, N C C Brown, and A Altadmri. Frame-Based Editing. *Journal of Visual Languages and Sentient Systems*, page 3, 2017. doi: 10.18293/VLSS2017-012. URL `https://doi.org/10.18293/VLSS2017-012`.

[7] Sergej Chodarev. Commands composition user interface pattern. In *2015 IEEE 13th International Scientific Conference on Informatics, INFORMATICS 2015 - Proceedings*, pages 120–123. Institute of Electrical and Electronics Engineers Inc., jan 2016. ISBN 9781467398688. doi: 10.1109/Informatics.2015.7377819.

[8] A. J. Campbell, F. M. Marchhtti, and D. J. Mewhort. Reading speed and text production a note on right-justification techniques. *Ergonomics*, 1981. ISSN 13665847. doi: 10.1080/00140138108924885.

[9] Microsoft. TypeScript - JavaScript that scales. URL `https://www.typescriptlang.org/`.

[10] Facebook. React – A JavaScript library for building user interfaces. URL `https://reactjs.org/`.

[11] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 1960. ISSN 15577317. doi: 10.1145/367177.367199.

[12] Andrew Blinn. disconcision/fructure: a structured interaction engine. URL `https://github.com/disconcision/fructure`.

[13] MIT. LLK/scratch-blocks: Scratch Blocks is a library for building creative computing interfaces. URL https://github.com/LLK/scratch-blocks.

Appendices

Kale commands

Commands different by only a shift key ⇧ are closely related. Commands that change the selection are usually not listed in the context menu. Applicable mnemonics are represented by a **B**old letter.

A.1 Up / Down ↑ / ↓

Change the selection to the previous or next expression by pre-order traversal. This means that by running a command repeatedly, you will move through every possible selection to either side of the cursor. If there is no previous or next expression, the current selection is preserved.

Alternative shortcuts: K / J (based on the Vim editor).

A.2 Left / Right ← / →

Change the selection to previous or next non-inline non-list expression by pre-order traversal. This approximates moving between expressions that visually resemble lines, or broken up lines. If there is no previous or next expression, the current selection is preserved.

Alternative shortcuts: H / L (based on the Vim editor). See also the "Left / Right Sibling" ⇧+H / ⇧+L commands.

A.3 Add Space or Move Space Up Space

This is also known as the "smart space". If the currently selected expression is a space, run "Move Up" on the current selection. If the selection is a function call, add a new child space in the first argument. Otherwise insert a new sibling space to the right of the current selection.

Because this command will not create a new sibling to function calls, the "New Line Below /Above" N / ⇧+N command is also useful to know.

A.4 Edit ↲

This can also be invoked by double clicking on an expression. Edit the text inside the expression.

If the expression is a space, open a new menu instead, letting you select the type of expression with which to replace the space. Once a selection is made editing proceeds as normal, with the exception of editing function expressions, which when the edit is complete, will create an appropriate number of spaces for their arguments.

If the expression is a list, no action is performed.

Alternative shortcut: E

A.5 Copy C

Copy the currently selected expression to the top of the Clipboard Stack

A.6 Paste 0 – 9

Replace the currently selected expression with an expression at the index matching the shortcut key from the Clipboard Stack, removing it if it is not pinned.

A.7 Select Parent P

Select the parent of the currently selected expression. If a parent does not exist, the current selection is preserved.

A.8 Move Up ⇧+P

Move the currently selected expression to be the last sibling of its parent. If a parent does not exist, no action is performed. This is similar to the "Barf" operation in the Emacs Par Edit mode.

A.9 New Line Below / Above N / ⇧+N

Insert a new list expression around the current selection. Note that "list merging" is performed, merging immediate list children of a list with their parent. For example, invoking this command on an expression which already has a list as its parent, will not create a new list, instead appending a new sibling to the selection to the current list.

A.10 Delete ⟵

Delete the currently selected expression, potentially replacing it with a new space if no other expression would remain in a function.

Alternative shortcut: D

A.11 Cut X

Copy the currently selected expression, then delete it. Similar to the Ctrl+X command in text editors.

A.12 Delete and Add Space R

Delete the currently selected expression, replacing it with a new space. This helps helps if you want to completetly **R**eplace an expression with something new.

A.13 Cut and Add Space

S

Copy the currently selected expression, then perform "Delete and Add Space". This helps if you want to Shuffle expressions around, replacing the current selection with something new, but using the old expression somewhere else.

A.14 Open Definition

O

Open an editor with the function name of the currently selected expression. If a function with a given name does not exist in the current workspace, create one. If an editor with a given function is already opened, do not create a new editor, instead move the focus to the closest editor displaying the selected function.

This command can also be triggered by middle-clicking the expression.

A.15 New Argument After / Before

I / ⇧+I

Create a new sibling space before or after the currently selected expression. This is command largely superseded by the "Smart Space" Space action but can be useful for creating multiple consecutive spaces or if a sibling is desired instead of a child and the currently selected expression is a function call.

A.16 Comment

Q

Edit the comment on the currently selected expression. If no comment exists, create one. Note that comments on spaces and literals are handled specially. Any empty comments are automatically removed.

A.17 Disable

\

Disable the currently selected expression, enabling it if it is already disabled. Spaces cannot be disabled.

A.18 Make a Variable...

V

Replace the currently selected expression with a new variable expression, then perform the "Edit" ↲ command.

A.19 Make a String...

G

Replace the currently selected expression with a new string literal expression, then perform the "Edit" ↲ command.

A.20 Turn into a Function Call F

If the currently selected expression is a space, replace it with a function call expression. Otherwise wrap the currently selected expression in a new function call expression, making it its first argument, and perform the "Edit" ↲ command.

A.21 Replace the Parent ⇧+F

Replace the parent of the currently selected expression with the selected expression and its siblings.

A.22 Left / Right Sibling ⇧+H / ⇧+L

Change the selection to left or right sibling of the currently selected expression. Unlike the ↑ ↓, this ignores any children the currently selected expression might have. If no left or right sibling exists, move the previous or next expression by pre-order traversal.

A.23 Open a Function /

Switch focus from the current editor to the "Open a Function..." search field. From within the field you can use ↑ and ↓ to move between suggestions, then press ↲ to confirm the selected suggestion.

A.24 Focus on Editor Above / Below ⇧+K / ⇧+J

Move the focus from the current editor to the one above or below it. If editor above / below exists, wrap around. Each jump is added to the editor jump list and can be reversed by using "Jump back" ⇧+O command.

A.25 Jump back ⇧+O

Move back through the editor jump stack populated by the "Focus on Editor Above / Below" ⇧+K / ⇧+J commands.

A.26 Close Editor ⇧+D

Close the current editor.

Design exploration

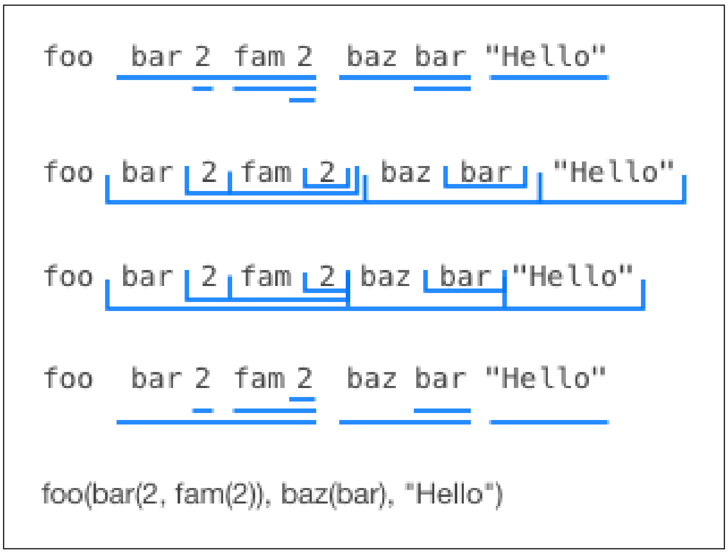

Figure B.1: Explicit atomic expression underlines

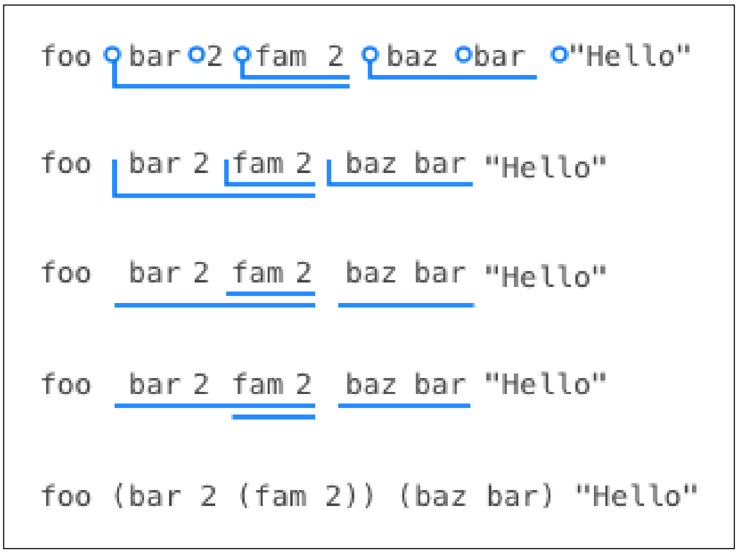

Figure B.2: Implicit atomic expression underlines

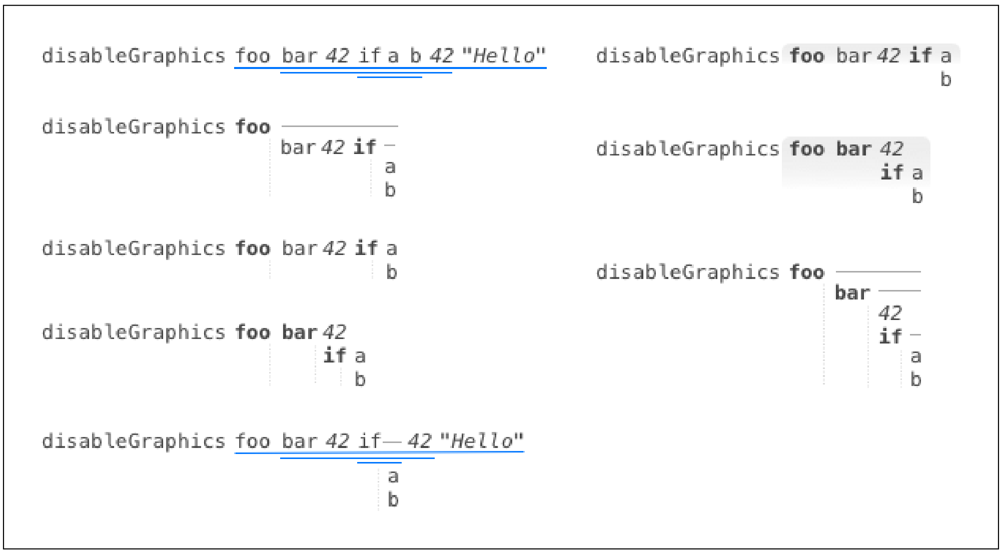

Figure B.3: Exploring line breaking UI

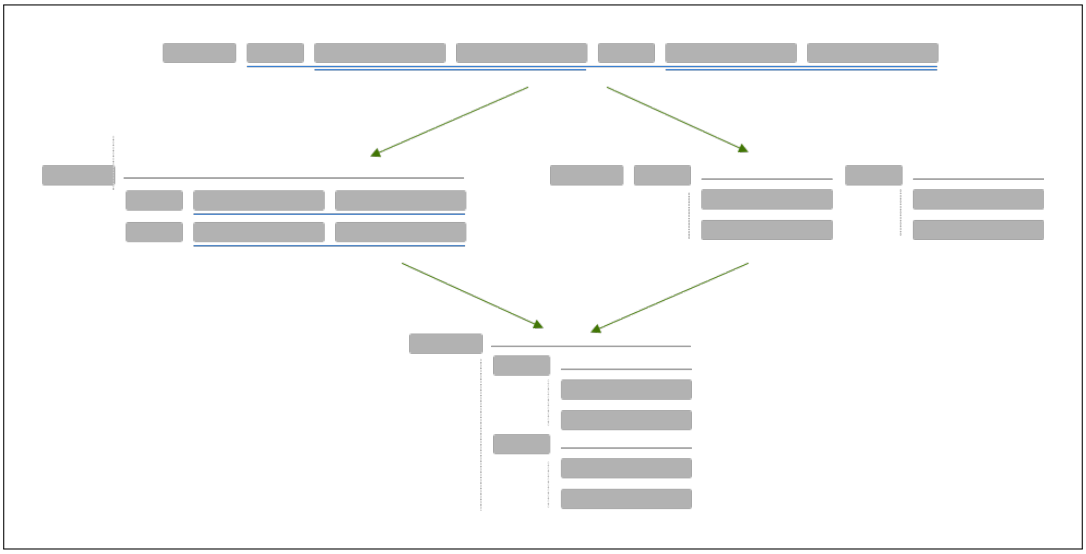

Figure B.4: Line breaking algorithm exploration

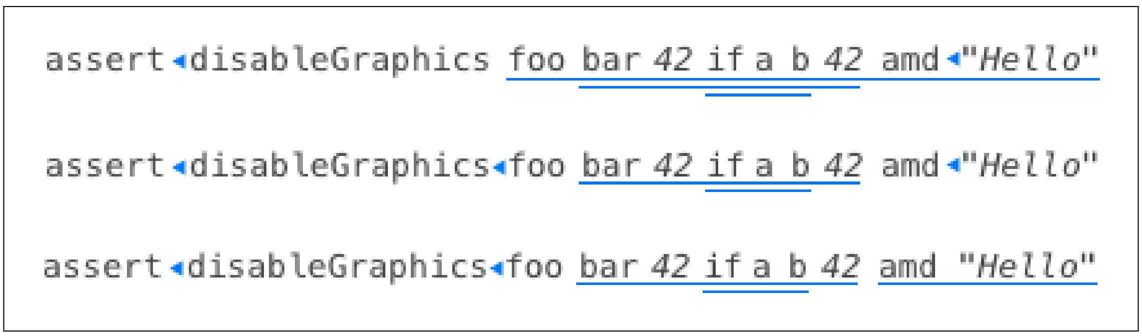

Figure B.5: Exploring more compact notations